ARM® Cortex™-A Series

Version: 4.0

Programmer's Guide

ARM®

ARM DEN0013D (ID012214)

ARM Cortex-A Series
Programmer's Guide

Release Information

The following changes have been made to this book.

Change history

Date	Issue	Confidentiality	Change
25 March 2011	A	Non-Confidential	First release
10 August 2011	B	Non-Confidential	Second release. Updated to include Virtualization, Cortex-A15 processor, and LPAE. Corrected and revised throughout
25 June 2012	C	Non-Confidential	Updated to include Cortex-A7 processor, and big.LITTLE. Index added. Corrected and revised throughout.
22 January 2014	D	Non-Confidential	Updated to include Cortex-A12 processor, Cache Coherent Interconnect, expanded GIC coverage, Multi-core processors, Corrected and revised throughout.

Proprietary Notice

In this document, where the term ARM is used to refer to the company it means "ARM or any of its subsidiaries as appropriate".

Web Address

http://www.arm.com

Contents
ARM Cortex-A Series Programmer's Guide

Preface

It is estimated that the number of mobile phones in the world will exceed the human population sometime in 2014. It is also estimated that over 90% of all these mobile devices have an ARM processor inside them.

This book provides an introduction to ARM technology for programmers using ARM Cortex-A series processors conforming to the ARMv7–A architecture. The book is meant to complement rather than replace other ARM documentation available for Cortex-A series processors, such as the ARM *Technical Reference Manuals* (TRMs) for the processors themselves, documentation for individual devices or boards or, most importantly, the *ARM® Architecture Reference Manual* (the ARM ARM).

The purpose of this book is to provide a single guide for programmers who want to develop applications for the Cortex-A series of processors, bringing together information from a wide variety of sources that will be useful to both assembly language and C programmers. Hardware concepts such as caches and Memory Management Units are covered, but only where this is valuable to the application writer. We will also look at the way operating systems such as Linux make use of ARM features, and how to take full advantage of the capabilities of the ARM processor, in particular writing software for multi-core processors.

Although much of the content of this book is also applicable to older ARM processors, it does not explicitly cover processors that implement older versions of the Architecture. The Cortex-R series and M-series processors are mentioned but not described. Our intention is to provide an approachable introduction to the ARM architecture, covering the feature set in detail and providing practical advice on writing both C and assembly language programs to run efficiently on a Cortex-A series processor.

This is not an introductory level book. It assumes some knowledge of the C programming language and microprocessors, but not of any ARM-specific background. In the allotted space, we cannot hope to cover every topic in detail. In some chapters, we suggest additional reading (referring either to books or web sites) that can give a deeper level of background to the topic in hand, but in this

book we focus on the ARM-specific detail. We do not assume the use of any particular tool chain. We will mention both GNU and ARM tools in the course of the book. We hope that the book is suitable for programmers who have a desktop PC or x86 background and are taking their first steps into the ARM processor based world.

The first chapters of the book cover the basic features of the ARM Cortex-A series processors. An introduction to the fundamentals of the ARM architecture, covering the various registers, and modes and some background on individual processors is provided in Chapter 2 and 3. Chapters 4 and 5 provide a brisk introduction to ARM assembly language programming, and assembly language instructions. We look at floating-point and the ARM Advanced SIMD extensions (NEON™) in Chapters 6 and 7. These chapters are only an introduction to the relevant topics. We then switch our focus to the memory system and look at Caches, Memory Management and Memory Ordering in Chapters 8, 9 and 10. Dealing with exceptions and interrupts are covered in Chapters 11 and 12.

The remaining chapters of the book provide more advanced programming information. Chapter 13 provides an overview of boot code. Chapter 14 looks at issues with porting C and assembly code to the ARMv7 architecture, from other architectures and from older versions of the ARM architecture. Chapter 15 covers the Application Binary Interface, knowledge of which is useful to both C and assembly language programmers. Profiling and optimizing of code is covered in Chapters 16 and 17. Many of the techniques presented are not specific to the ARM architecture, but we also provide some processor-specific hints.

Chapters 18 and 19 cover the area of multi-core processors. We take a detailed look at how these are implemented by ARM and how you can write code to take advantage of them. Power management is an important part of ARM programming and is covered in Chapter 20. The final chapters of the book provide a brief coverage of the ARM Security Extensions (TrustZone®) in Chapter 21, the ARM Virtualization extensions in Chapter 22, big.LITTLE™ technology in Chapter 23, and the hardware debug features available to programmers in Chapter 24. Appendix A gives a summary of the available ARM instructions. Appendix B gives a brief introduction to some of the tools and platforms available to those getting started with ARM programming, and Appendix C gives step-by-step instructions for configuring and building Linux for ARM systems.

Preface to the 4th Edition

The ARM architecture continues to evolve. ARM recently announced the ARMv8 architecture, which, although not covered by this book, has influenced its content. Small changes have been made to the ARMv7 instruction set architecture for compatibility reasons. Those changes are included in this version. We have also taken advantage of the opportunity to carry out a general revision of the contents of the Guide.

You will find some sections moved, some extensively rewritten, and some with minor changes. Chapters that covered the same areas have been combined. The chapter on Registers has been largely rewritten, to include the effects of the Security and Virtualization Extensions and the introduction of Privilege levels. The chapters on Exception Handling have been consolidated, so that they now tell a coherent story. A similar approach has been applied to multiprocessing, and parallelization. In all cases, what was two chapters is now one, with the contents updated. LPAE is now covered in the chapter on Virtualization. The chapter on Tools, Operating Systems and Boards has been moved to the Appendices, while the chapter on big.LITTLE technology has been extensively revised to keep pace with rapidly changing technology.

ARM has added another processor to the ARMv7 Cortex-A series, and the new Cortex-A12 processor is covered in this new edition. You will also notice that what we call the devices has changed. Processor now refers to the marketed device, such as the Cortex-A15 processor, elsewhere you will find references to cores, and clusters (of cores).

Some content that existed in the previous editions of the *Cortex™-A Series Programmer's Guide* has been removed. You will notice that the NEON/VFP Appendix and the chapter on Writing NEON Code have been removed. We originally said that the information on NEON would need a book of its own. It now has one, in the form of the *ARM® NEON™ Programmer's Guide*.

Additional information has also been provided in the form of programming hints and tips, for developers using the Cortex-A series processors. While these cannot cover every eventuality that the developer might encounter we hope that the examples included will prove useful.

The *ARM® Cortex™-A Series Programmer's Guide* has proved to be a very popular addition to the ARM documentation set, and now also forms the reference textbook for the *ARM Accredited Engineer* (AAE) examinations.

Glossary

Abbreviations and terms used in this document are defined here.

AAPCS ARM Architecture Procedure Call Standard.

ABI Application Binary Interface.

ACP Accelerator Coherency Port.

AEABI ARM Embedded ABI.

AHB Advanced High-Performance Bus.

AMBA® Advanced Microcontroller Bus Architecture.

AMP Asymmetric Multi-Processing.

APB Advanced Peripheral Bus.

ARM ARM The ARM *Architecture Reference Manual*.

ASIC Application Specific Integrated Circuit.

APSR Application Program Status Register.

ASID Address Space ID.

ATPCS ARM Thumb® Procedure Call Standard.

AXI Advanced eXtensible Interface.

BE8 Byte Invariant Big-Endian Mode.

BIU Bus Interface Unit.

BSP Board Support Package.

BTAC Branch Target Address Cache.

BTB Branch Target Buffer.

CISC Complex Instruction Set Computer.

CP15 Coprocessor 15. System control coprocessor.

CPSR Current Program Status Register.

DAP Debug Access Port.

DBX Direct Bytecode Execution.

DDR Double Data Rate (SDRAM).

DMA Direct Memory Access.

DMB Data Memory Barrier.

DPU Data Processing Unit.

DS-5™ The ARM Development Studio.

DSB Data Synchronization Barrier.

DSP Digital Signal Processing.

DSTREAM® An ARM debug and trace unit.

DVFS Dynamic Voltage/Frequency Scaling.

EABI Embedded ABI.

ECC Error Correcting Code.

ECT Embedded Cross Trigger.

EOF End Of File.

ETB Embedded Trace Buffer™.

ETM Embedded Trace Macrocell™.

FDT Flattened Device Tree.

FIQ An interrupt type (formerly fast interrupt).

FPSCR Floating-Point Status and Control Register.

GCC GNU Compiler Collection.

GIC Generic Interrupt Controller.

GIF Graphics Interchange Format.

GPIO General Purpose Input/Output.

Gprof GNU profiler.

Harvard architecture
Architecture with physically separate storage and signal pathways for instructions and data.

HCR Hyp Configuration Register.

HMP Heterogenous Multi-Processing.

ICU Instruction Cache Unit.

IDE Integrated development environment.

I/F Interface (abbreviation used in some diagrams).

IPA Intermediate Physical Address.

IRQ Interrupt Request (normally external interrupts).

ISA Instruction Set Architecture.

ISB Instruction Synchronization Barrier.

ISR Interrupt Service Routine.

Jazelle™ The ARM bytecode acceleration technology.

JIT Just In Time.

L1/L2 Level 1/Level 2.

LPAE Large Physical Address Extension.

LSB Least Significant Bit.

MESI	A cache coherency protocol with four states; Modified, Exclusive, Shared and Invalid.
MMU	Memory Management Unit.
MOESI	A cache coherency protocol with five states; Modified, Owned, Exclusive, Shared and Invalid.
MPU	Memory Protection Unit.
MSB	Most Significant Bit.
NEON™	The ARM Advanced SIMD Extensions.
NMI	Non-Maskable Interrupt.
Normal world	The execution environment when the core is in the Non-secure state.
Oprofile	A Linux system profiler.
QEMU	A processor emulator.
PCI	Peripheral Component Interconnect. A computer bus standard.
PCS	Procedure Call Standard.
PFU	Prefetch Unit.
PIPT	Physically Indexed, Physically Tagged.
PLE	Preload Engine.
PLI	Preload Instruction.
PMU	Performance Monitor Unit.
PoC	Point of Coherency.
PoU	Point of Unification.
PPI	Private Peripheral Input.
Privilege	The ability to perform certain tasks that cannot be done from User (*Unprivileged)* mode.
PSCI	Power State Coordination Interface.
PSR	Program Status Register.
RCT	Runtime Compiler Target.
RISC	Reduced Instruction Set Computer.
RVCT	RealView® Compilation Tools (the “ARM Compiler”).
SBZP	Should Be Preserved.
SCU	Snoop Control Unit.
Secure world	The execution environment when the core is in the Secure State.
SGI	Software Generated Interrupt.
SIMD	Single Instruction, Multiple Data.
SiP	System in Package.

SMP	Symmetric Multi-Processing.
SoC	System on Chip.
SP	Stack Pointer.
SPI	Shared Peripheral Interrupt.
SPSR	Saved Program Status Register.
Streamline	A graphical performance analysis tool.
SVC	Supervisor Call instruction. (Previously SWI)
SWI	Software Interrupt instruction. (Replaced with SVC)
SYS	System Mode.
TAP	Test Access Port (JTAG Interface).
TDMI®	Thumb, Debug, Multiplier, ICE.
TEX	Type Extension.
Thumb®	An instruction set extension to ARM.
Thumb-2	A technology extending the Thumb instruction set to support both 16-bit and 32-bit instructions.
TLB	Translation Lookaside Buffer.
TLS	Thread Local Storage.
TrustZone	The ARM security extension.
TTB	Translation Table Base.
TTBR	Translation Table Base Register.
UAL	Unified Assembly Language.
UART	Universal Asynchronous Receiver/Transmitter.
UEFI	Unified Extensible Firmware Interface.
U-Boot	A Linux Bootloader.
UNK	Unknown.
USR	User mode, a non-privileged processor mode.
VFP	The ARM floating-point instruction set. Before ARMv7, the VFP extension was called the Vector Floating-Point architecture, and was used for vector operations.
VIC	Vectored Interrupt Controller.
VIPT	Virtually Indexed, Physically Tagged.
VMID	Virtual Machine ID.
VMSA	Virtual Memory Systems Architecture.
XN	Execute Never.

Typographical conventions

This book uses the following typographical conventions:

italic	Highlights important notes, introduces special terminology, denotes internal cross-references, and citations.
bold	Used for terms in descriptive lists, where appropriate.
`monospace`	Denotes text that you can enter at the keyboard, such as commands, file and program names, instruction names, parameters and source code.
`monospace italic`	Denotes arguments to monospace text where the argument is to be replaced by a specific value.
< and >	Enclose replaceable terms for assembler syntax where they appear in code or code fragments. For example: `MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>`

Feedback on this book

We have tried to ensure that the Cortex-A Series Programmer's Guide is both easy to read and still covers the material in enough depth to provide the comprehensive introduction to using the processors that we originally intended.

If you have any comments on this book, don't understand our explanations, think something is missing, or think that it is incorrect, send an e-mail to `errata@arm.com`. Give:

- The title, The *Cortex-A Series Programmer's Guide*.
- The number, ARM DEN0013D.
- The page number(s) to which your comments apply.
- What you think needs to be changed.

ARM also welcomes general suggestions for additions and improvements.

References

Cohen, D. "*On Holy Wars and a Plea for Peace*", USC/ISI IEN April, 1980 http://www.ietf.org/rfc/ien/ien137.txt.

Furber, Steve. "*ARM System-on-chip Architecture*", 2nd edition, Addison-Wesley, 2000, ISBN: 9780201675191.

Hohl, William. "*ARM Assembly Language: Fundamentals and Techniques*" CRC Press, 2009. ISBN: 9781439806104.

Sloss, Andrew N.; Symes, Dominic.; Wright, Chris. "*ARM System Developer's Guide: Designing and Optimizing System Software*", Morgan Kaufmann, 2004, ISBN: 9781558608740.

ANSI/IEEE Std 754-1985, "*IEEE Standard for Binary Floating-Point Arithmetic*".

ANSI/IEEE Std 754-2008, "*IEEE Standard for Binary Floating-Point Arithmetic*".

ANSI/IEEE Std 1003.1-1990, "*Standard for Information Technology - Portable Operating System Interface (POSIX) Base Specifications*, Issue 7".

ANSI/IEEE Std 1149.1-2001, "*IEEE Standard Test Access Port and Boundary-Scan Architecture*".

ANSI/IEEE Std 1275-1994, "*IEEE Standard for Boot Firmware (Initialization Configuration) Firmware: Core Requirements and Practices*".

The *ARM® Architecture Reference Manual* (known as the ARM ARM) is a must-read for any serious ARM programmer. It is available (after registration) from the ARM web site. It fully describes the ARMv7 instruction set architecture, programmer's model, system registers, debug features and memory model. It forms a detailed specification to which all implementations of ARM processors must adhere.

References to the *ARM® Architecture Reference Manual* in this document are to:

ARM® Architecture Reference Manual - ARMv7-A and ARMv7-R edition (ARM DDI 0406).

Note

In the event of a contradiction between this book and the ARM ARM, the ARM ARM is definitive and must take precedence. In most instances, however, the ARM ARM and the Cortex-A Programmer's Guide cover two separate world views. The ARM ARM is for processor implementers, while this book is for processor users. The most likely scenario is that this book will describe something in a way that does not cover all architecturally permitted behaviors, or rewords an abstract concept in more practical terms.

ARM® Generic Interrupt Controller Architecture Specification version 2.0 (ARM IHI 0048).

ARM® Compiler Toolchain Assembler Reference (DUI 0489).

ARM® C Language Extensions (IHI 0053).

ARM® NEON™ Programmer's Guide (DEN 0018).

Procedure Call Standard for the ARM® Architecture (ARM IHI 0042).

Power State Coordination Interface (PSCI) Platform Design Document (ARM DEN 0022)

The individual processor Technical Reference Manuals provide a detailed description of the processor behavior. They can be obtained from the ARM web site documentation area http://infocenter.arm.com.

Chapter 1
Introduction

You can find ARM processors everywhere. More than 10 billion ARM processor based devices had been manufactured by the end of 2008. At the end of 2013, over 52 billion ARM processors had been shipped. It is likely that readers of this book own at least one product containing ARM processor based devices – a mobile phone, tablet device, personal computer, television or even a car. It might come as a surprise to programmers more used to the personal computer to learn that the highly successful x86 architecture occupies a much smaller position in terms of total microprocessor shipments, with over three billion devices.

The ARM architecture has advanced significantly since the first ARM1 silicon in 1985. ARM produces a whole family of processors that share common instruction sets and programmer's models and have some degree of backward compatibility.

Let's begin, however, with a brief look at the history of ARM.

1.1 History

The first ARM processor, the ARM1, was designed at Acorn Computers Limited by a team led by Sophie Wilson and Steve Furber, with the first silicon (which worked first time!) produced in April 1985. The ARM1 was quickly replaced by the ARM2, which added multiplier hardware, and was used in real systems, including the Acorn Archimedes personal computer.

Photograph courtesy of The Centre for Computing History, www.computinghistory.org.uk

Figure 1-1 Acorn Archimedes

ARM as a separate company was formed in Cambridge, England in November 1990, as Advanced RISC Machines Ltd. It was a joint venture between Apple Computers, Acorn Computers and VLSI Technology and has outlived two of its parents. The original 12 employees came mainly from the team within Acorn Computers. One reason for spinning ARM off as a separate company was that the processor had been selected by Apple Computers for use in its Newton product.

Figure 1-2 The Apple Newton

The new company quickly decided that the best way forward for their technology was to license their *Intellectual Property* (IP). Instead of designing, manufacturing and selling the chips themselves, they would sell rights to their designs to semiconductor companies. These companies would design the ARM processor into their own products, in a partnership model. This IP licensing business is how ARM continues to operate today. ARM was able to sign up licensees, with Sharp, Texas Instruments and Samsung prominent names among the first customers. In 1998, ARM floated on the London Stock Exchange and Nasdaq. At the time of writing, ARM has over 2000 employees and has expanded somewhat from its original remit of processor design. ARM also licenses Physical IP – libraries of cells (NAND gates, RAM and so forth), graphics and video accelerators and software development products such as compilers, debuggers and development boards.

1.2 System-on-Chip (SoC)

Designers today can assemble computer chips containing a billion or more transistors. Designing and verifying such complex circuits has become an extremely difficult task. It is increasingly rare for all of the parts of such systems to be designed by a single company. In response to this, ARM and other semiconductor IP companies design and verify components (so-called IP blocks or processors). These are licensed by semiconductor companies who use these blocks in their own designs and include microprocessors, DSPs, 3D graphics and video controllers, along with many other functions.

The semiconductor companies take these blocks and include many other parts to create a complete system on the chip, forming a *System-on-Chip* (SoC). The architects must select the appropriate core(s), memory controllers, on-chip memory, peripherals, bus interconnect and other logic (perhaps including analog or radio frequency components), in order to produce a system.

Application Specific Integrated Circuit (ASIC) is another term that we will use in the book. This is an Integrated Circuit design that is specific to a particular application. An ASIC might well contain an ARM core, memory and other components. Clearly there is a large overlap between ASICs and SoCs. The term SoC usually refers to a device with a higher degree of integration, including many of the parts of the system in a single device, possibly including analog, mixed-signal or radio frequency circuits.

Semiconductor companies investing tens of millions of dollars to create these devices will normally invest a great deal in software to run on their platform. It would be uncommon to produce a complex system with a powerful processor without at least having ported one or more operating systems to it and written device drivers for peripherals.

Of course, operating systems like Linux require significant amounts of memory to run, more than is usually possible on a single silicon device. The term System-on-Chip is therefore not always entirely accurate, because the SoC does not always contain the whole system. Apart from the issue of silicon area, it is also often the case that many useful parts of a system require specialist silicon manufacturing processes that preclude them from being placed on the same die.

1.3 Embedded systems

An embedded system is conventionally defined as a piece of computer hardware running software designed to perform a specific task. Examples of such systems might be TV set-top boxes, smartcards, routers, disk drives, printers, automobile engine management systems, MP3 players or photocopiers. These contrast with what is generally considered a computer system, that is, one that runs a wide range of general purpose software and possesses input and output devices such as a keyboard, and a graphical display of some kind.

This distinction is becoming increasingly blurred. Consider the cellular or mobile phone. A basic model might only perform the task of making phone calls, but modern smartphones can run a complex operating system to which many thousands of applications are available for download.

Embedded systems can contain very simple 8-bit microprocessors, such as an Intel 8051 or PIC micro-controllers, or some of the more complex 32 or 64-bit processors, such as the ARM family described in this book. They require RAM and some form of non-volatile storage to hold the program(s) to be executed by the system. Systems will almost always have additional peripherals, relating to the actual function of the device – typically including UARTs, interrupt controllers, timers, GPIO controllers, but also potentially quite complex blocks such as GPUs, or DMA controllers.

Software running on such systems is typically grouped into two separate parts: the *Operating System* (OS) and applications that run on top of the OS. In this book, we will concentrate mainly on examples from Linux. The source code for Linux is readily available for inspection by the reader and is likely to be familiar to many programmers. Nevertheless, lessons learned from Linux are equally applicable to other operating systems.

Memory Footprint

In many systems, to minimize cost (and power), memory size can be limited. You might be forced to consider the size of the program and how to reduce memory usage while it runs.

Real-time behavior

A feature of certain systems is that there are deadlines to respond to external events. This might be a “hard” requirement (a car braking system *must* respond within a certain time) or “soft” requirement (audio processing must complete within a certain time-frame to avoid a poor user experience – but failure to do so under rare circumstances might not render the system worthless).

Power In many embedded systems the power source is a battery, and programmers and hardware designers must take great care to minimize the total energy usage of the system. This can be done, for example, by slowing the clock, reducing supply voltage or switching off the core when there is no work to be done.

Cost Reducing the bill of materials can be a significant constraint on system design.

Time to market

In competitive markets, the time to develop a working product can significantly impact the success of that product.

Chapter 2
ARM Architecture and Processors

ARM does not manufacture silicon devices. Instead, ARM creates microprocessor designs, that are licensed to semiconductor companies and OEMs, who then integrate them into System-on-Chip devices.

To ensure compatibility between implementations, ARM defines architecture specifications that define how compliant products must behave. Processors implementing the ARM architecture conform to a particular version of the architecture.

The ARM architecture supports implementations across a very wide range of performance points. Its simplicity leads to very small implementations, and this enables very low power consumption.

The Cortex-A series processors covered in this book conform to the ARMv7-A architecture. There might also be multiple processors with different internal implementations and micro-architectures, different cycle timings and clock speeds that conform to the same version of the architecture, in that they execute the ARM instruction set defined for the architecture and pass the ARM Validation System tests.

2.1 Architectural profiles

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 left as implementation-defined. For example, the architecture does not define cache sizes or cycle timings for individual instructions. These are determined by the individual core and SoC.

Each architecture version can also define optional extensions. These might be implemented in a particular implementation of a processor. For example, in the ARMv7 architecture, the Advanced SIMD (NEON) technology is available as an optional extension, and is described in Chapter 7 *Introducing NEON*.

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

The profiles are as follows:

A The *Application* profile defines an architecture aimed at high performance processors, supporting a virtual memory system using a *Memory Management Unit* (MMU) and therefore capable of running fully featured operating systems. Support for the ARM and Thumb instruction sets is provided.

ARMv7-A, the Application profile, is implemented by all Cortex-A series processors, and by processors developed by companies who have licensed the ARM architecture. At the beginning of 2014, just under three billion Cortex-A Series chips had been shipped.

The ARMv8-A architecture, which is not described in this book, supports the AArch32 state, a 32-bit implementation of the architecture that is backwards compatible with ARMv7-A.

R The *Real-time* profile defines an architecture aimed at systems that require deterministic timing and low interrupt latency. There is no support for a virtual memory system, but memory regions can be protected using a simple *Memory Protection Unit* (MPU).

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.

2.2 Architecture history and extensions

The ARM architecture changed relatively little between the first test silicon in the mid-1980s through to the first ARM6 and ARM7 devices of the early 1990s. In the first version of the architecture, the majority of the load, store and arithmetic operations along with the exception model and register set was implemented by the ARM1. Version 2 added multiply and multiply-accumulate instructions and support for coprocessors, plus other innovations. These early processors only supported 26-bits of address space. Version 3 of the architecture separated the program counter and program status registers and added several new modes, enabling support for 32-bits of address space. Version 4 adds support for halfword load and store operations and an additional kernel-level privilege mode. Readers unfamiliar with the ARM architecture should not worry if parts of this description use terms they have not met, because all of these topics are covered in the following chapters.

The ARMv4T architecture, which introduced the Thumb (16-bit) instruction set, was implemented by the ARM7TDMI® and ARM9TDMI® processors, products that have shipped in their billions.

The ARMv5TE architecture added improvements for DSP-type operations and saturated arithmetic and to ARM/Thumb interworking.

ARMv6 made a number of enhancements, including support for unaligned memory accesses, significant changes to the memory architecture and for multi-core support, plus some support for SIMD operations operating on bytes or halfwords within the 32-bit registers. It also provided a number of optional extensions, notably Thumb-2 and Security Extensions (TrustZone). Thumb-2 extends Thumb to be a mixed length (16-bit and 32-bit) instruction set.

The ARMv7-A architecture makes the Thumb-2 extensions mandatory and adds the Advanced SIMD extensions (NEON), described in Chapter 7.

For a number of years, ARM adopted a sequential numbering system for processors with ARM9 following ARM8, that 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 Limited 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 2-1 on page 2-4 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.

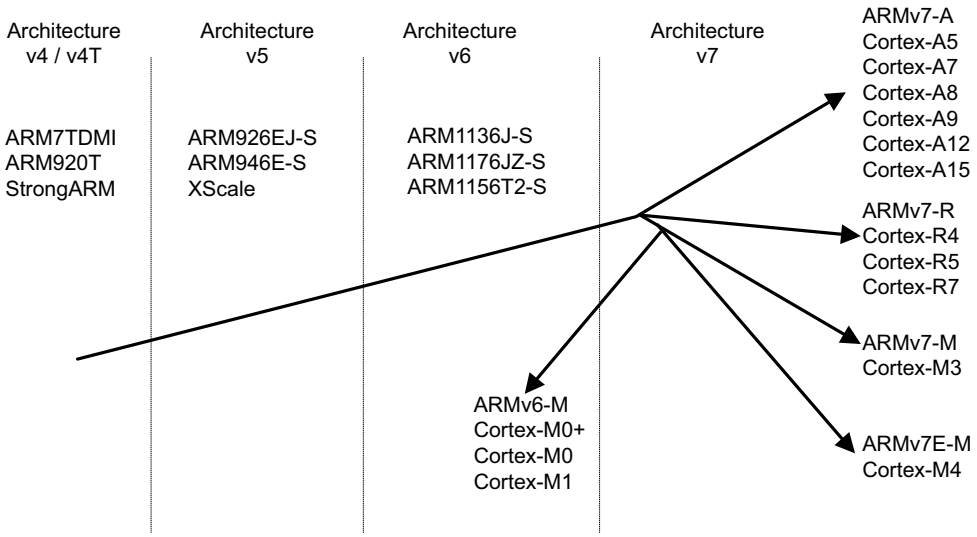


Figure 2-1 Architecture and processors

In Figure 2-2, we show 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 still be used on ARMv7 processors.

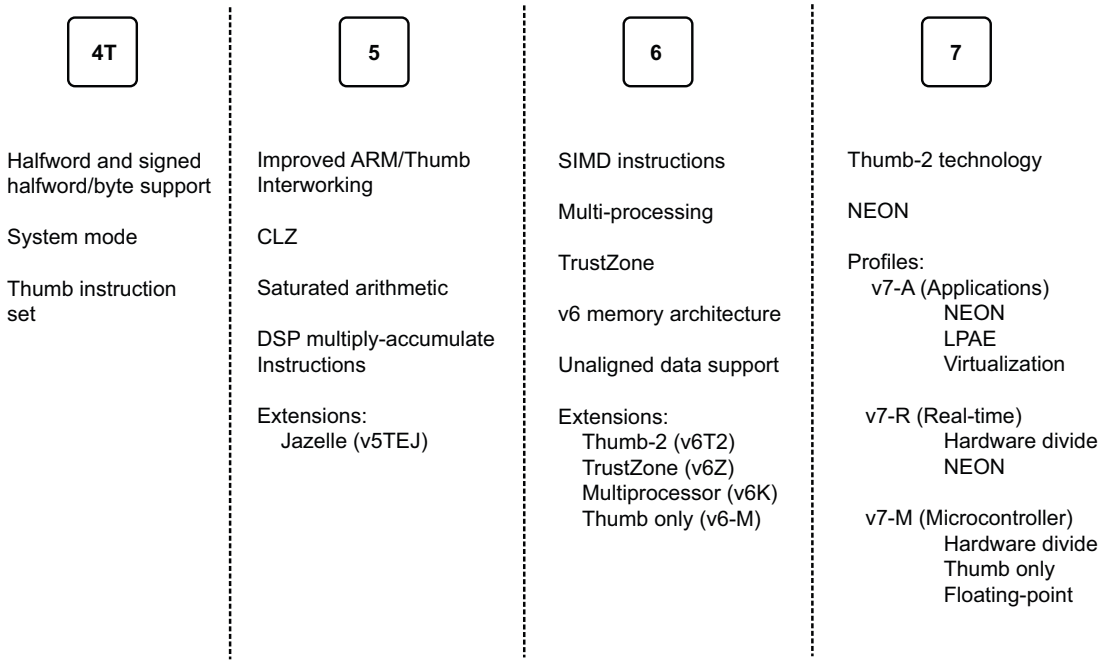


Figure 2-2 Architecture history

Individual chapters of this book will cover these topics in greater detail, but here we will briefly introduce a number of architecture elements.

2.2.1 DSP multiply-accumulate and saturated arithmetic instructions

These instructions, added in the ARMv5TE architecture, improve the capability for digital signal processing and multimedia software and are denoted by the letter E. The new instructions provide many variations of signed multiply-accumulate, saturated add and subtract, and count leading zeros and are present in all later versions of the architecture. In many cases, this made it possible to remove a simple separate DSP from the system.

2.2.2 Jazelle

Jazelle-DBX (Direct Bytecode eXecution) was introduced in ARMv5TEJ to accelerate Java performance while conserving power. A combination of increased memory availability and improvements in *Just-In-Time* (JIT) compilers have subsequently reduced its value in application processors. For this reason, many ARMv7-A processors do not implement this hardware acceleration.

Jazelle-DBX is best suited to providing high performance Java in systems with very limited memory (for example, feature phone or low-cost embedded use). In today's systems, it is mainly used for backwards compatibility.

2.2.3 Thumb Execution Environment (ThumbEE)

Introduced in ARMv7-A, ThumbEE is sometimes referred to as Jazelle-RCT (Runtime Compilation Target). It involves small changes to the Thumb instruction set that make it a better target for code generated at runtime in controlled environments, for example, by managed languages like Java, Dalvik, C#, Python or Perl.

ThumbEE is designed to be used by *Just-In-Time* (JIT) or *Ahead-Of-Time* (AOT) compilers, where it can reduce the code size of recompiled code. The use of ThumbEE is now deprecated by ARM.

2.2.4 Thumb

The ARMv7 architecture contains two main instruction sets, the ARM and Thumb instruction sets. Much of the functionality available is identical in the two instruction sets. The Thumb instruction set is a subset of the most commonly used 32-bit ARM instructions. Thumb instructions are each 16 bits long, and have a corresponding 32-bit ARM instruction that has the same effect. The main reason for using Thumb code is to reduce code density. Because of its improved density, Thumb code tends to cache better than the equivalent ARM code and can reduce the amount of memory required.. You can still use the ARM instruction set and you may want to, for particular code sections which require the highest performance. See *The ARM instruction sets* on page 4-3.

2.2.5 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.

2.2.6 Security Extensions (TrustZone)

The optional Security Extensions referred to as TrustZone introduced with ARMv6K have been implemented in all ARM Cortex-A processors. TrustZone provides a separate Secure world to isolate sensitive code and data from the normal world that contains the operating system and applications. The software in the Secure world is therefore intended to provide security services to the Normal (non-secure) world applications. TrustZone is described in Chapter 21.

2.2.7 VFP

Before ARMv7, the VFP extension was called the Vector Floating-Point Architecture, and supported vector operations. VFP is an extension that implements single-precision and optionally, double-precision floating-point arithmetic, compliant with the ANSI/IEEE Standard for Floating-Point Arithmetic.

2.2.8 Advanced SIMD (NEON)

The ARM NEON technology provides an implementation of the Advanced SIMD instruction set, with separate register files (shared with VFP). Some implementations have a separate NEON pipeline back-end. It supports 8, 16, 32 and 64-bit integer, and single-precision (32-bit) floating-point data, that can be operated on as vectors in 64-bit and 128-bit registers. NEON is described in Chapter 7, and in the *ARM® NEON™ Programmer's Guide.*

2.2.9 Coprocessors

The ARM architecture supports a way of extending the instruction set by using *Coprocessors*, to extend the functionality of an ARM processor. There are 16 coprocessors with numbers from 0 to 15, On earlier ARM cores, a dedicated hardware interface was provided to permit external coprocessors to be connected. On the Cortex-A series processors, only internal coprocessors are supported. These are CP15 (system control for such things as cache and MMU, CP14 for debug, and CP10 and 11 for NEON and VFP operations. Coprocessors are described in *Registers* on page 3-6. Coprocessor operations are described in *Miscellaneous instructions* on page 5-17.

2.2.10 Large Physical Address Extension (LPAE)

LPAE is optional in the v7-A architecture and is presently supported by the Cortex-A7, Cortex-A12, and Cortex-A15 processors. It enables 32-bit processors that are normally limited to addressing a maximum of 4GB of address space to access up to 1TB of address space by translating 32-bit virtual memory addresses into 40-bit physical memory addresses. See *Large Physical Address Extensions* on page 22-10.

2.2.11 Virtualization

The ARM Virtualization Extensions are optional extensions to the ARMv7-A architecture profile. The extensions support the use of a virtual machine monitor, known as the hypervisor, to switch from one operating system to another. When implemented in a single core or in a multi-core system, the Virtualization Extensions support running multiple virtual machines on a single cluster. See Chapter 22.

2.2.12 big.LITTLE

big.LITTLE processing was introduced in ARMv7 to enable devices to balance the requirements for processing power and power efficiency. big.LITTLE uses a high performance cluster, such as the Cortex-A15 processor, coupled with an energy efficient cluster, such as the

Cortex-A7 processor. The “big” cluster can be utilized for heavy workloads, while the “LITTLE” cluster can take over for the majority of mobile device workloads. big.LITTLE is described in Chapter 23.

2.3 Processor properties

In this section, we consider some ARM processors and identify which processor implements which architecture version. In *Cortex-A series processors* on page 2-10 we take a slightly more detailed look at some of the individual processors that implement architecture version v7-A. Some terminology will be used in this chapter that might be unfamiliar to the first-time user of ARM processors and will not be explained until later in the book.

Table 2-1 lists the architecture version implemented by a number of older ARM processors.

Table 2-1 Older ARM processors and architectures

Architecture version	Applications processor	Embedded processor
v4T	ARM720T™ ARM920T™ ARM922T™	ARM7TDMI™
v5TE	-	ARM946E-S™ ARM966E-S™ ARM968E-S
v5TEJ	ARM926EJ-S™	-
v6K	ARM1136J(F)-S™ ARM11™ MPCore™	-
v6T2	-	ARM1156T2-S™
v6K + security extensions	ARM1176JZ(F)-S™	-

Table 2-2 lists the architecture version implemented by the Cortex family of processors.

Table 2-2 Cortex processors and architecture versions

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

Table 2-3 on page 2-9 compares the properties of Cortex-A series processors. For processor cache information, see Table 8-1 on page 8-11.

Table 2-3 Some properties of Cortex-A series processors

	Processor					
	Cortex-A5	**Cortex-A7**	**Cortex-A8**	**Cortex-A9**	**Cortex-A12**	**Cortex-A15**
Release date	Dec 2009	Oct 2011	July 2006	March 2008	June 2013	April 2011
Typical clock speed	~1GHz	~1GHz on 28nm	~1GHz on 65nm	~2GHz on 40nm	~2GHz on 28nm	~2.5GHz on 28nm
Execution order	In-order	In-order	In-order	Out of order	Out of order	Out of order
Cores	1 to 4	1 to 4	1	1 to 4	1 to 4	1 to 4
Peak integer throughput	1.6DMIPS/ MHz	1.9DMIPS/ MHz	2DMIPS/MHz	2.5DMIPS/MHz	3.0DMIPS/ MHz	3.5DMIPS/ MHz
VFP architecture	VFPv4	VFPv4	VFPv3	VFPv3	VFPv4	VFPv4
NEON architecture	NEON	NEONv2	NEON	NEON	NEONv2	NEONv2
Half precision extension	Yes	Yes	No	Yes	Yes	Yes
Hardware Divide	No	Yes	No	No	Yes	Yes
Fused Multiply Accumulate	Yes	Yes	No	No	Yes	Yes
Pipeline stages	8	8	13	9 to 12	11	15+
Instructions decoded per cycle	1	Partial dual issue	2 (Superscalar)	2 (Superscalar)	2 (Superscalar)	3 (Superscalar)
Return stack entries	4	8	8	8	8	48
LPAE	No	Yes	No	No	Yes	Yes
Floating Point Unit	Optional	Yes	Yes	Optional	Yes	Optional
AMBA interface	64-bit AMBA 3	128-bit AMBA 4	64 or 128-bit AMBA 3	2× 64-bit AMBA 3	128-bit AMBA 4	128-bit AMBA 4
Generic Interrupt Controller (GIC)	Included	Optional	Not included	Included	External	Optional
Trace	Optional ETM	Optional ETM separate macrocell	Integrated ETM	Integrated PTM	Integrated PTM	Integrated PTM

2.4 Cortex-A series processors

In this section, we take a closer look at each of the processors that implement the ARMv7-A architecture. Only a general description is given in each case, for more specific information on each processor, see Table 2-3 on page 2-9 and Table 8-1 on page 8-11.

2.4.1 The Cortex-A5 processor

The Cortex-A5 processor is the smallest ARM multi-core applications processor. Devices based on this processor are typically low-cost, capable of delivering the internet to the widest possible range of devices from low-cost entry-level smartphones and smart mobile devices, to embedded, consumer and industrial devices.

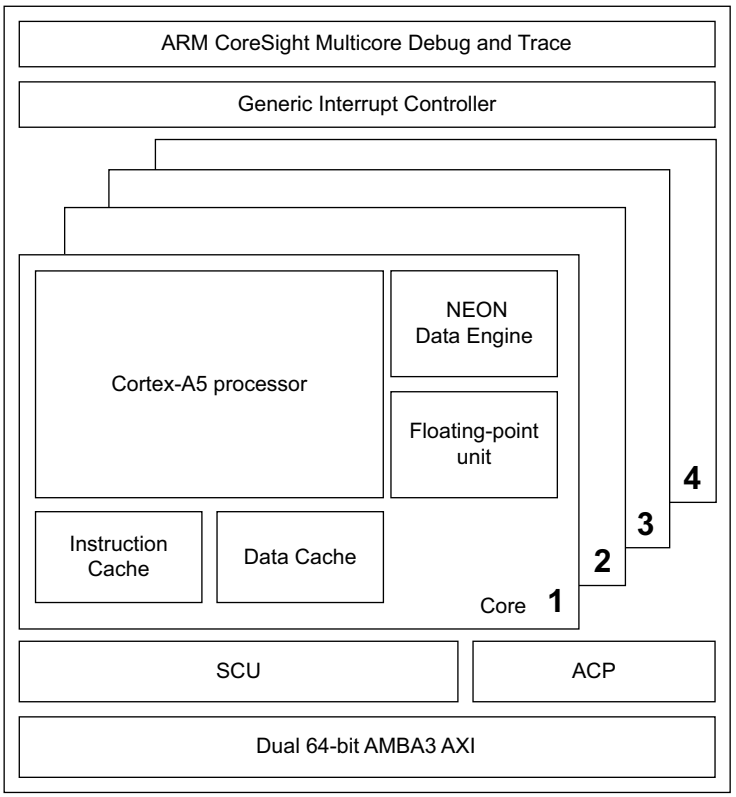


Figure 2-3 Cortex-A5 processor

The Cortex-A5 processor has the following features:

- Full application compatibility with other Cortex-A series processors.
- Multiprocessing capability for scalable, energy efficient performance.
- Optional Floating-point or NEON units for media and signal processing.
- High-performance memory system including caches and memory management unit.
- High value migration path from older ARM processors.

2.4.2 The Cortex-A7 processor

The ARM Cortex-A7 processor is the most energy efficient application processor developed by ARM and extends ARM's low-power leadership in entry level smart phones, tablets and other advanced mobile devices.

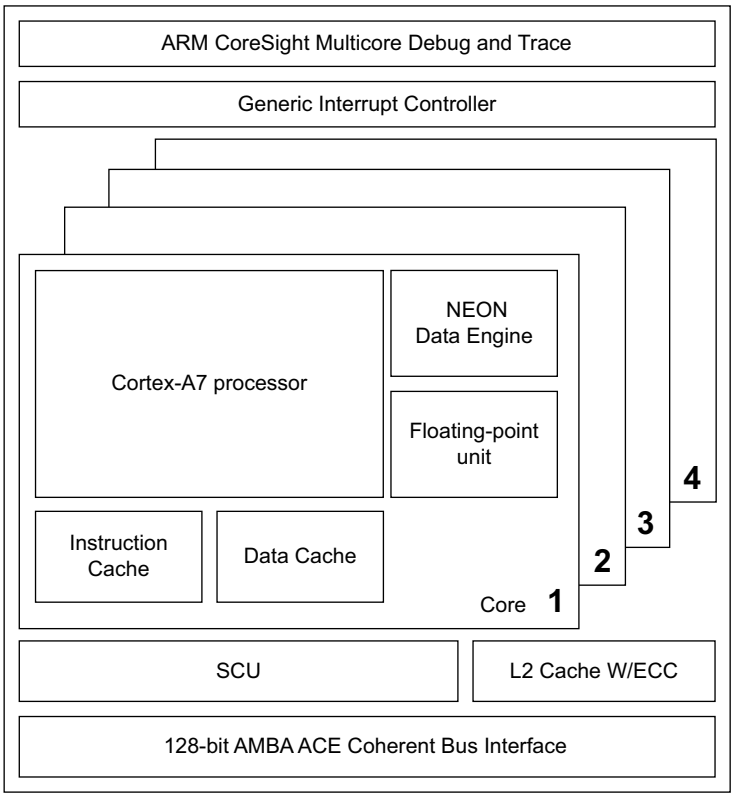


Figure 2-4 Cortex-A7 processor

The Cortex-A7 processor has the following features:

- Architecture and feature set identical to the Cortex-A15 processor, enabling big.LITTLE configuration.
- Less than 0.5mm^2, using 28nm process technology.
- Full application compatibility with all Cortex-A series processors.
- Tightly-coupled low latency level 2 cache (up to 4MB).
- Floating-point unit.
- NEON technology for multimedia and SIMD processing.

2.4.3 The Cortex-A8 processor

The ARM Cortex-A8 processor, has the ability to scale in speed from 600MHz to greater than 1GHz. The Cortex-A8 processor can meet the requirements for power-optimized mobile devices needing operation in less than 300mW; and performance-optimized consumer applications requiring 2000 Dhrystone MIPS. It is available in a number of different devices, including the S5PC100 from Samsung, the OMAP3530 from Texas Instruments and the

i.MX515 from Freescale. From high-end feature phones to netbooks, DTVs, printers and automotive-infotainment, the Cortex-A8 processor offers a proven high-performance solution with millions of units shipped annually

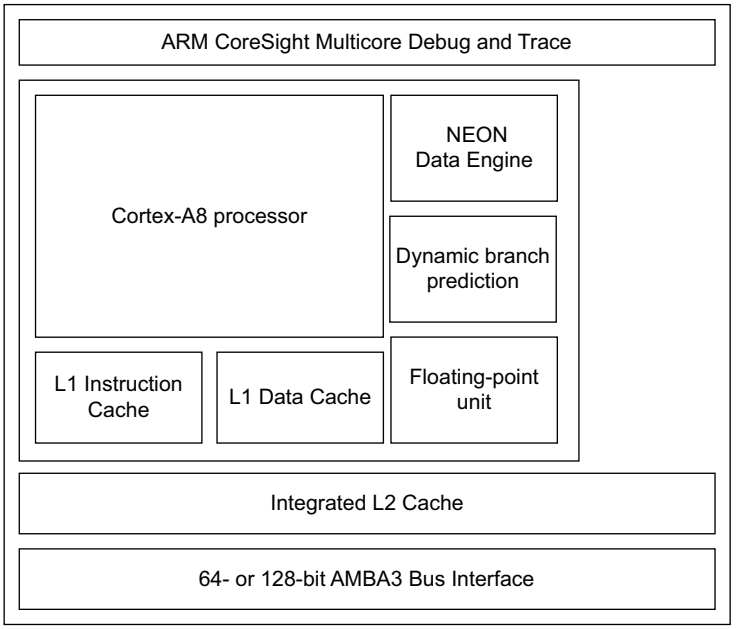


Figure 2-5 Cortex-A8 processor

The Cortex-A8 processor has the following features:

- Frequency from 600MHz to more than 1GHz.
- High performance superscalar architecture.
- NEON technology for multi-media and SIMD processing.
- Compatibility with older ARM processors.

2.4.4 The Cortex-A9 processor

The ARM Cortex-A9 processor is a power-efficient and popular high performance choice in low power or thermally constrained cost-sensitive devices.

It is currently shipping in large volumes for smartphones, digital TV, consumer and enterprise applications. The Cortex-A9 processor provides an increase in performance of greater than 50% compared to the Cortex-A8 processor. The Cortex-A9 processor can be configured with up to four cores delivering peak performance when required. Configurability and flexibility makes the Cortex-A9 processor suitable for wide variety of markets and applications.

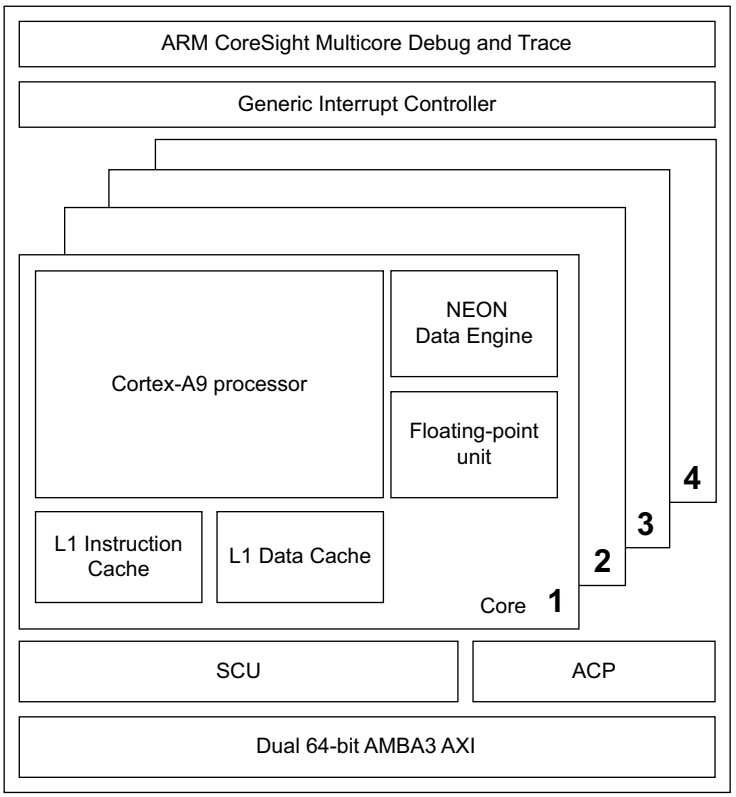


Figure 2-6 Cortex-A9 processor

Devices containing the Cortex-A9 processor include nVidia's dual-core Tegra-2, the SPEAr1300 from ST and TI's OMAP4 platform.

The Cortex-A9 processor has the following features:

- Out-of-order speculating pipeline.
- 16, 32 or 64KB four way associative L1 caches.
- Floating-point unit.
- NEON technology for multi-media and SIMD processing.
- Available as a speed or power optimized hard macro implementation.

2.4.5 The Cortex-A12 processor

The Cortex-A12 processor is a high performance mid-range mobile processing solution designed for mobile applications, for example, use in smartphones and tablet devices. The Cortex-A12 processor is a successor to the highly successful Cortex-A9 processor and is optimized for highest performance in the mainstream mobile power envelope leading to best-in-class efficiency.

The high performance and high-end feature set of the Cortex-A12 processor is suitable for many use cases. Mid-range devices can build on the success of high-end devices and continue driving the fastest growing market segment in mobile.

Architecturally, the Cortex-A12 processor is based on the latest ARMv7-A architecture and features extensions that are aligned with processors such as the Cortex-A15 processor.

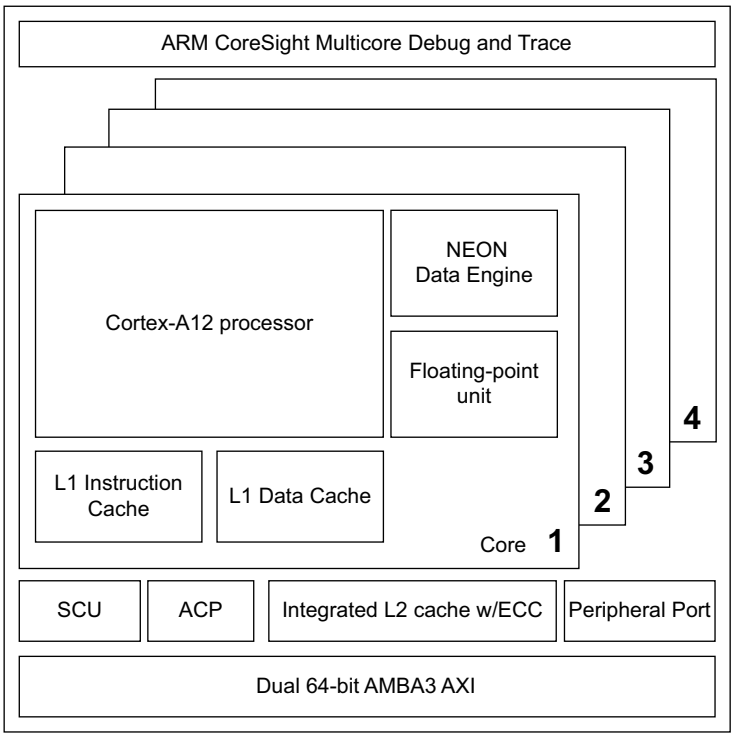


Figure 2-7 Cortex-A12 processor

The Cortex-A12 processor has the following features:

- 40-bit Large Physical Address Extensions (LPAE) addressing up to 1 TB of RAM.
- Full application compatibility with all Cortex-A series processors.
- NEON technology for multi-media and SIMD processing.
- Virtualization and TrustZone security technology

2.4.6 The Cortex-A15 processor

The ARM Cortex-A15 processor is designed to deliver unprecedented flexibility and processing capability. This processor is designed with advanced power reduction techniques, and enables products in a wide range of markets ranging from mobile computing, high-end digital home, servers and wireless infrastructure.

The Cortex-A15 MPCore processor has full application compatibility with all other Cortex-A series processors.

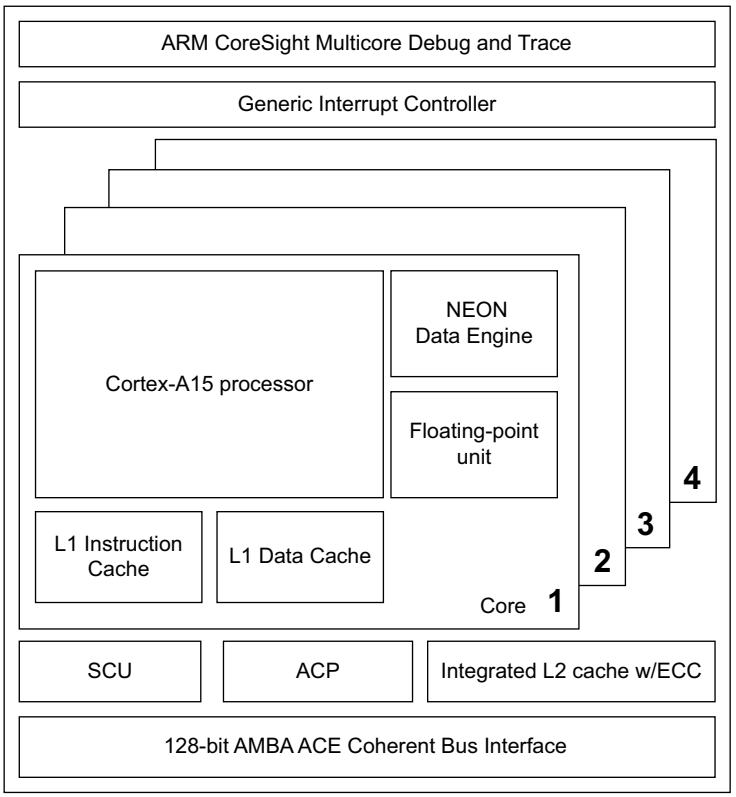


Figure 2-8 Cortex-A15 processor

The Cortex-A15 processor has the following features:

- Highly scalable, up to 2.5GHz performance.
- Full application compatibility with all Cortex-A series processors.
- Out-of-order superscalar processor.
- Tightly coupled low-latency level 2 cache (up to 4MB).
- Floating-point unit.
- NEON technology for multi-media and SIMD processing.
- Available as a quad-core hard macro implementation.

2.5 Key architectural points of ARM Cortex-A series processors

A number of key points are common to the Cortex-A family of devices:

- 32-bit RISC core, with 16 × 32-bit visible registers with mode-based register banking.
- Modified Harvard Architecture (separate, concurrent access to instructions and data).
- Load/Store Architecture.
- Thumb-2 technology as standard.
- VFP and NEON options.
- Backward compatibility with code from previous ARM processors.
- 4GB of virtual address space and a minimum of 4GB of physical address space.
- Hardware translation table walking for virtual to physical address translation.
- Virtual page sizes of 4KB, 64KB, 1MB and 16MB. Cacheability attributes and access permissions can be set on a per-page basis.
- Big-endian and little-endian data access support.
- Unaligned access support for basic load/store instructions.
- *Symmetric Multi-processing* (SMP) support on MPCore™ variants, that is, multi-core versions of the Cortex-A series processors, with full data coherency at the L1 cache level. Automatic cache and *Translation Lookaside Buffer* (TLB) maintenance propagation provides high efficiency SMP operation.
- *Physically indexed, physically tagged* (PIPT) data caches. See *Virtual and physical tags and indexes* on page 8-11.

All of these architectural points are described in the chapters which follow.

Chapter 3
ARM Processor Modes and Registers

The ARM architecture is a modal architecture. Before the introduction of Security Extensions it had seven processor modes, summarized in Table 3-1. There were six privileged modes and a non-privileged user mode. *Privilege* 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, MMU configuration and cache operations. Modes are associated with exception events, which are described in Chapter 11 *Exception Handling*.

Table 3-1 ARM processor modes before ARMv6

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	
Supervisor (SVC)	Entered on reset or when a Supervisor Call instruction (SVC) is executed	
Abort (ABT)	Entered on a memory access exception	
Undef (UND)	Entered when an undefined instruction executed	
System (SYS)	Mode in which the OS runs, sharing the register view with User mode	

The introduction of the TrustZone Security Extensions (see Figure 2-2 on page 2-4) created two security states for the processor that are independent of Privilege and processor mode, with a new Monitor mode to act as a gateway between the Secure and Non-secure states and modes existing independently for each security state.

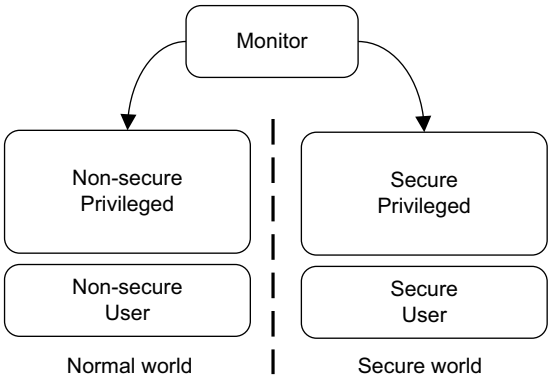


Figure 3-1 Secure and Non-secure worlds

The TrustZone Security Extensions are described in Chapter 21, but for the present, for processors that implement the TrustZone extension, system security is achieved by dividing all of the hardware and software resources for the device so that they exist in either the Secure world for the security subsystem, or the Normal (Non-secure) world for everything else. When a processor is in the Non-secure state, it cannot access the memory that is allocated for Secure state.

In this situation the Secure Monitor acts as a gateway for moving between these two worlds.

If Security Extensions are implemented, software executing in Monitor mode controls transition between Secure and Non-secure processor states.

The ARMv7-A architecture Virtualization Extensions add a hypervisor mode (Hyp), in addition to the existing privileged modes. Virtualization enables more than one Operating System to co-exist and operate on the same system. The ARM Virtualization Extensions therefore makes it possible to operate multiple Operating Systems on the same platform.

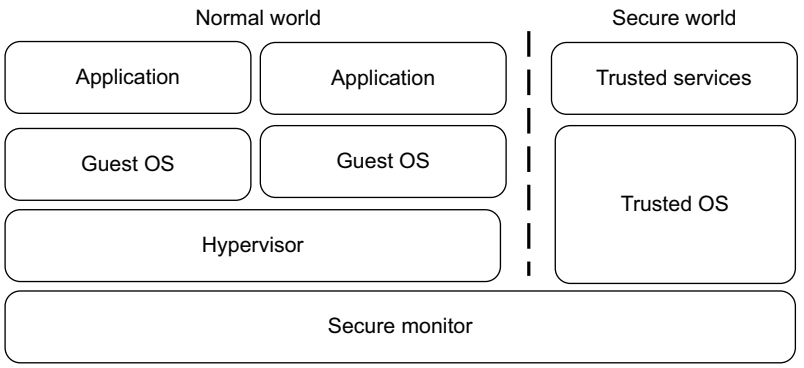


Figure 3-2 The hypervisor

If the Virtualization Extensions are implemented there is a privilege model different to that of previous architectures. In Non-secure state there can be three privilege levels, PL0, PL1 and PL2.

PL0 The privilege level of application software, that executes in User mode. Software executed in User mode is described as unprivileged software. This software cannot access some features of the architecture. In particular, it cannot change many of the configuration settings.

Software executing at PL0 can make only unprivileged memory accesses.

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

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

An Operating System is expected to execute across all PL1 modes and its applications executing in PL0 (User Mode).

PL2 Hyp mode is normally used by a hypervisor, that controls, and can switch between Guest Operating Systems that execute at PL1.

If Virtualization Extensions are implemented, a hypervisor will execute at PL2 (Hyp mode). A hypervisor will control and enable multiple Operating Systems to co-exist and execute on the same processor system.

These privilege levels are separate from the TrustZone Secure and Normal (Non-secure) settings.

Note

The privilege level defines the ability to access resources in the current security state, and does not imply anything about the ability to access resources in the other security state.

Figure 3-3 on page 3-4 shows the available processor modes under different processor states.

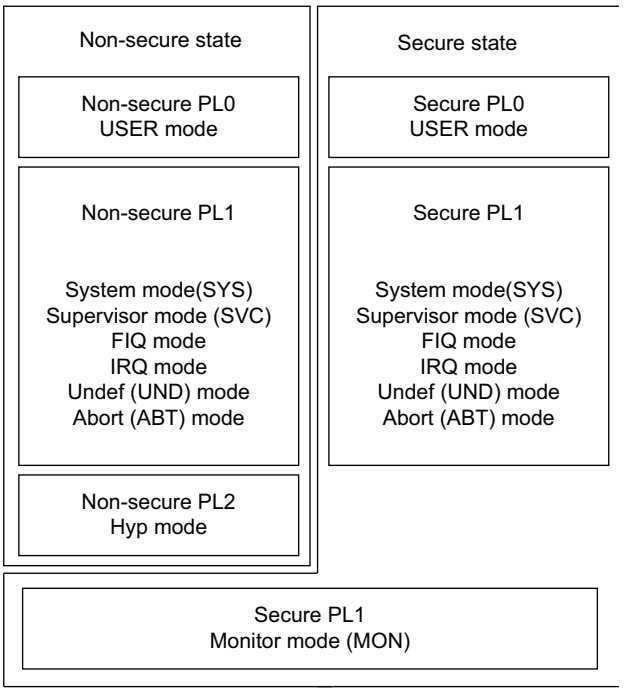


Figure 3-3 Privilege levels

The presence of particular processor modes and states depends on whether the processor implements the relevant architecture extension, as shown in Table 3-2.

Table 3-2 ARMv7 processor modes

Mode	Encoding	Function	Security state	Privilege level
User (USR)	10000	Unprivileged mode in which most applications run	Both	PL0
FIQ	10001	Entered on an FIQ interrupt exception	Both	PL1
IRQ	10010	Entered on an IRQ interrupt exception	Both	PL1
Supervisor (SVC)	10011	Entered on reset or when a Supervisor Call instruction (SVC) is executed	Both	PL1
Monitor (MON)	10110	Implemented with Security Extensions. See Chapter 21	Secure only	PL1
Abort (ABT)	10111	Entered on a memory access exception	Both	PL1
Hyp (HYP)	11010	Implemented with Virtualization Extensions. See Chapter 22	Non-secure	PL2
Undef (UND)	11011	Entered when an undefined instruction executed	Both	PL1
System (SYS)	11111	Privileged mode, sharing the register view with User mode	Both	PL1

A general purpose Operating System, such as Linux, and its applications, are expected to run in Non-secure state. The Secure state is expected to be occupied by vendor-specific firmware, or security-sensitive software. In some cases, software running in the Secure state is even more privileged than that running in Non-secure.

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 link to relevant section.

3.1 Registers

The ARM 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 a saved copy of the CPSR from the previously executed mode, called the *Saved Program Status Register* (SPSR).

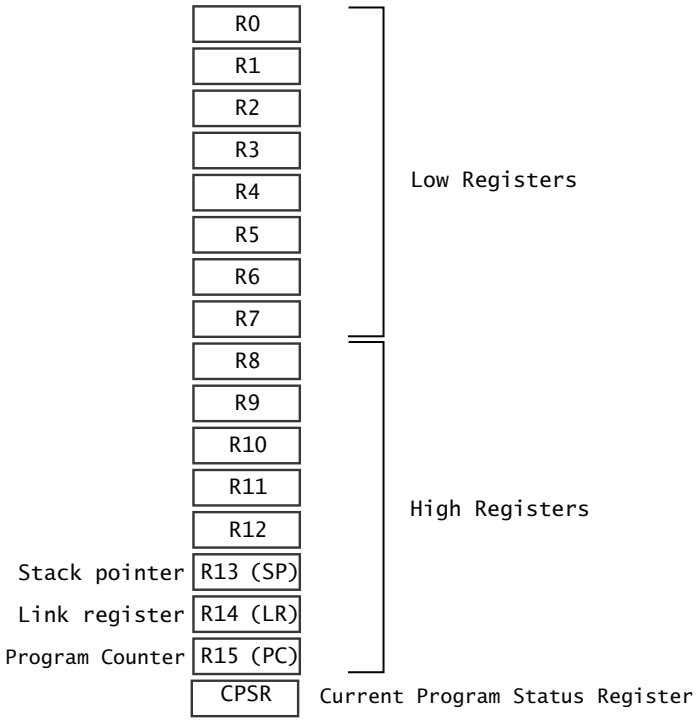


Figure 3-4 Programmer visible registers for user code

Although software can access the registers, depending on which 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 3-5 on page 3-7 are banked. They use physically distinct storage and are usually accessible only when a process is executing in that particular mode.

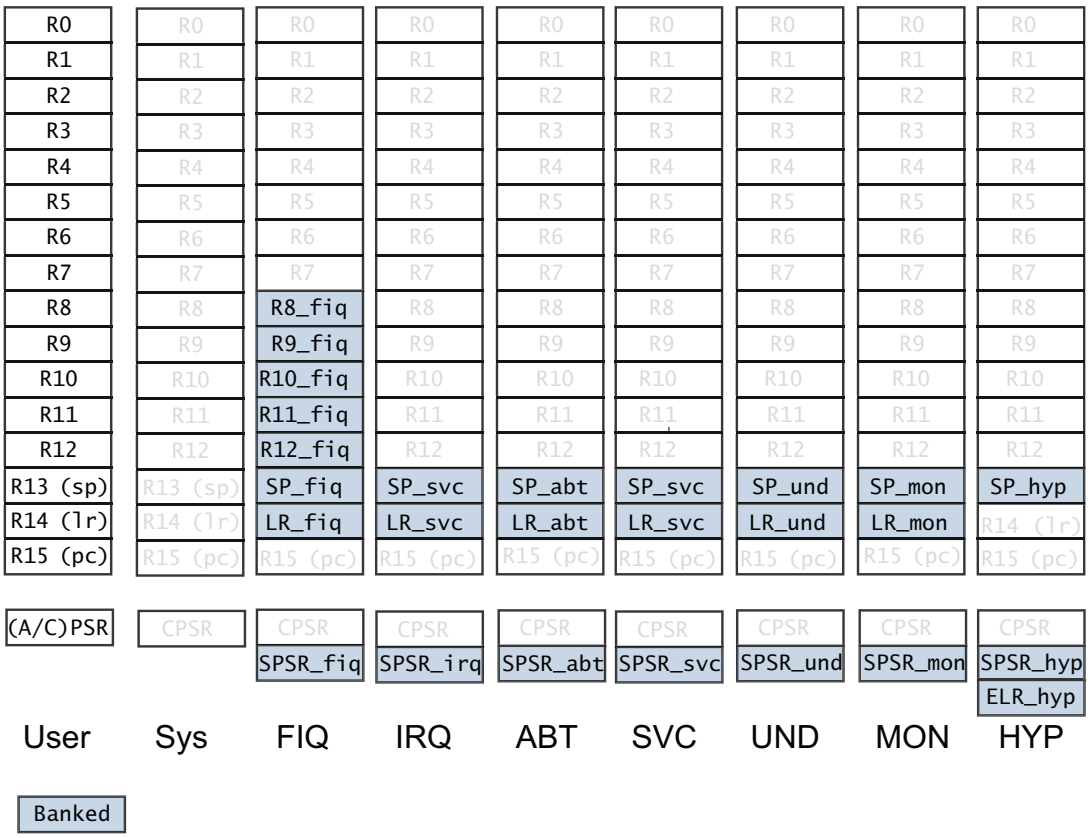


Figure 3-5 The ARM register set

In all modes, 'Low Registers' and R15 share the same physical storage location. Figure 3-5 shows that some 'High Registers' are banked for certain modes. For example, R8-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 and the SPSRs are banked.

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

R13 (in all modes) is the OS stack pointer, but it can be used as a general purpose register when not required for stack operations.

R14 (the Link Register) holds the return address from a subroutine entered when you use the branch with link (`BL`) instruction. It too 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 values of R15 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). When R15 is read in ARM state, bits [1:0] are zero and bits [31:2] contain the PC. In Thumb state, bit [0] 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 *ARM Architecture Procedure Call Standard* (AAPCS) or *ARM Embedded ABI* (AEABI) (see Chapter 15 *Application Binary Interfaces*) specifies how software should use the general purpose registers in order to interoperate between different toolchains or programming languages.

3.1.1 Hypervisor mode

Implementations that support the Virtualization Extensions have additional registers available in Hypervisor (Hyp) mode. Hypervisor mode operates at the PL2 level of privilege. It has access to its own versions of R13 (SP) and SPSR. It uses the User mode link register for storing function return addresses, and has a dedicated register, called ELR_hyp, to store the exception return address. Hyp mode is available only in the Normal world and provides facilities for virtualization, that are only accessible in this mode. See Chapter 22 *Virtualization*, for more information.

3.1.2 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 modes.

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

Figure 3-6 shows the bit assignments in the CPSR.

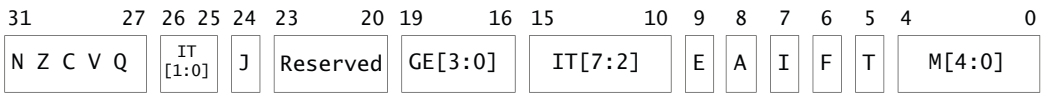


Figure 3-6 CPSR bits

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*).
- *J* – indicates whether the core is in Jazelle state.
- *GE*[3:0] – used by some SIMD instructions.
- *IT* [7:2] – `If-Then` conditional execution of Thumb-2 instruction groups.
- *E* bit controls load/store endianness.
- *A* bit disables asynchronous aborts.
- *I* bit disables IRQ.
- *F* bit disables FIQ.
- *T* bit – indicates whether the core is in Thumb state.
- *M*[4:0] – specifies the processor mode (FIQ, IRQ, as described in Table 3-1 on page 3-1).

The core 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 the PSR bits [4:0] that control the processor mode or the A, I and F bits that govern the exceptions to be enabled or disabled.

We will consider these bits in more detail in Chapter 5 and Chapter 11.

3.1.3 Coprocessor 15

CP15, the System Control coprocessor, provides control of many features of the core. It can contain up to sixteen 32-bit primary registers. Access to CP15 is privilege controlled and not all registers are available in User 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 name. For example, the CP15 System Control Register is called CP15.SCTLR.

Table 3-3 summarizes the function of some of the more relevant coprocessor registers used in Cortex-A series processors. We will consider some of these in more detail when we look at the operation of the cache and MMU.

Table 3-3 CP15 Register summary

Register	Description
Main ID Register (MIDR)	Gives identification information for the processor (including part number and revision).
Multiprocessor Affinity Register (MPIDR)	Provides a way to uniquely identify individual cores within a cluster.
CP15 c1 System Control registers	
System Control Register (SCTLR)	The main processor control register (see *System control register (SCTLR)* on page 3-12).
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.
Secure Configuration Register (SCR)	Used by TrustZone (Chapter 21).
CP15 c2 and c3, memory protection and control registers	
Translation Table Base Register 0 (TTBR0)	Base address of level 1 translation table (see Chapter 9).
Translation Table Base Register 1 (TTBR1)	Base address of level 1 translation table (see Chapter 9).
Translation Table Base Control Register (TTBCR)	Controls use of TTB0 and TTB1 (see Chapter 9).
CP15 c5 and c6, memory system fault registers	
Data Fault Status Register (DFSR)	Gives status information about the last data fault (see Chapter 11).
Instruction Fault Status Register (IFSR)	Gives status information about the last instruction fault (see Chapter 11).
Data Fault Address Register (DFAR)	Gives the virtual address of the access that caused the most recent precise data abort (see Chapter 11).
Instruction Fault Address Register (IFAR)	Gives the virtual address of the access that caused the most recent precise prefetch abort (see Chapter 11).
CP15 c7, cache maintenance and other functions	
Cache and branch predictor maintenance functions	See Chapter 8.
Data and instruction barrier operations	See Chapter 10.
CP15 c8, TLB maintenance operations	

Table 3-3 CP15 Register summary (continued)

Register	Description
CP15 c9, performance monitors	
CP15 c12, Security Extensions registers	
Vector Base Address Register (VBAR)	Provides the exception base address for exceptions that are not handled in Monitor mode.
Monitor Vector Base Address Register (MVBAR)	Holds the exception base address for all exceptions that are taken to Monitor mode.
CP15 c13, process, context and thread ID registers	
Context ID Register (CONTEXTIDR)	See description of ASID Chapter 9).
Software thread ID registers	See description of ASID Chapter 9).
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 shown in Example 3-1.

Example 3-1 CP15 Instruction syntax

```
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, because this would duplicate reference information that can readily be obtained from the *ARM Architecture Reference Manual* or product documentation.

AS an example of how we can read information from one of these registers, we will consider the read-only Main ID Register (MIDR), the format is shown in Figure 3-7.

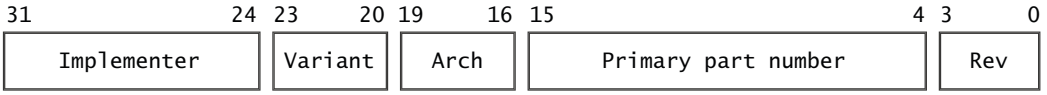


Figure 3-7 Main ID Register format

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

```
MRC p15, 0, R1, c0, c0, 0
```

The result, placed in register `R1`, tells software the processor it is running on. For an ARM Cortex processor the interpretation of the results is as follows:

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

- Bits [23:20] – variant, shows the revision number of the processor.
- Bits [19:16] – architecture, will be `0xF` for ARM architecture v7.
- Bits [15:4] – part number, for example `0xC08` for the Cortex-A8 processor.
- Bits [3:0] – revision, shows the patch revision number of the processor.

3.1.4 System control register (SCTLR)

The SCTLR is another of a number of registers that are accessed using CP15, and controls standard memory, system facilities and provides status information for functions implemented in the core.

The System Control Register is only accessible from PL1 or higher.

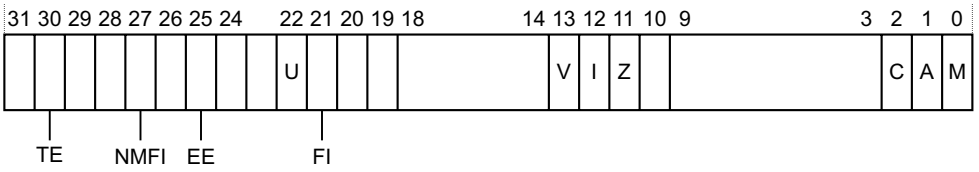


Figure 3-8 Simplified view of the system control register

The individual bits represent the following:

- TE – Thumb exception enable. This controls whether exceptions are taken in ARM or Thumb state.
- NMFI – Non-maskable FIQ (NMFI) support. See *External interrupt requests* on page 12-2.
- EE = Exception endianness. This defines the value of the CPSR.E bit on entry to an exception. See *Endianness* on page 14-2.
- U – Indicates use of the alignment model. See *Alignment* on page 14-5.
- FI – FIQ configuration enable. See *External interrupt requests* on page 12-2
- V – This bit selects the base address of the exception vector table. See *The Vector table* on page 11-7.
- *I* – Instruction cache enable bit. See Example 3-2 on page 3-13 and *Invalidating and cleaning cache memory* on page 8-17.
- *Z* – Branch prediction enable bit. See Example 3-2 on page 3-13 and *Invalidating and cleaning cache memory* on page 8-17
- *C* – Cache enable bit. See Example 3-2 on page 3-13 and *Invalidating and cleaning cache memory* on page 8-17
- A – Alignment check enable bit. See *Alignment* on page 14-5.
- M – Enable the MMU. See *Configuring and enabling the MMU* on page 9-3.

Part of the boot code sequence will typically be to set the Z bit in the CP15:SCTLR, System Control Register, that enables branch prediction. A code sequence to do this is shown in Example 3-2 on page 3-13.

Example 3-2 Setting bits in the SCTLR

```
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 Example 13-3 on page 13-3.

Chapter 4
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 core.

Many programmers writing at the application level will have little reason 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 disassembled code when debugging C programs, 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.*

4.1 Comparison with other assembly languages

An ARM processor is 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 cores 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 cores to perform arithmetic operations (such as `ADD`) and logical bit manipulation (such as `AND`). They also transfer program execution from one part of the program to another, in order 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 some 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 (but unlike the 68K), 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. Example 4-1 and Example 4-2 give a flavor of the differences between the different assembly languages.

Example 4-1 Instructions to add 100 to a value in a register

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

Example 4-2 Load a register with a 32-bit value from a register pointer

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

4.2 The ARM instruction sets

The ARMv7 architecture is a 32-bit processor architecture. It is also a load/store architecture, meaning that data-processing instructions operate only 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 two or even three 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, which 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-A 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 the ARM instruction set, with code size similar to that of Thumb instructions. 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.

In older ARM processors, systems often contained code which was compiled for ARM state and code which 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, required more instructions to carry out the same task, when compared with ARM code. Thumb code could typically encode smaller constant values within instructions and has shorter relative branches. See *Branches* on page 5-15. The available range for relative branches is approximately ±32MB for ARM instructions and ±16MB for the Thumb-2 extension. Thumb is further 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 were only half of the size, programs would be typically a third smaller than their ARM code equivalent. Thumb instructions are therefore used for reasons of code density 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 which 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) allows 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* on page 4-11.

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
```

4.3 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, that 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 to provide enough information to enable programmers to get started with the tools.

The names of GNU tool components will 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.

4.3.1 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 core.

4.3.2 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.

4.3.3 Sections

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

Directives with the same 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 will appear in `.bss`. The Block Started by Symbol (bss) segment defines the space for uninitialized static data.

4.3.4 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 will be on a word boundary. `.align n` gives 2^n alignment on ARM cores.

`.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.

`.global` ***symbol***

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 4-1 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 4-1 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	
|	: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	

4.3.5 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 will be 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.

4.3.6 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: *x*PSR, *x*PSR_all, *x*PSR_f, *x*PSR_*x*, *x*PSR_ctl, *x*PSR_fs, *x*PSR_fx, *x*PSR_f, *x*PSR_cs, *x*PSR_cf, *x*PSR_cx (where *x* = C current or S saved). See *Program Status Registers* on page 3-8.

Note

Chapter 15 *Application Binary Interfaces* describes how all of the registers are assigned a role within the procedure call standard and how the GNU Assembler lets you refer to the registers using their *Procedure Call Standard* (PCS) names. See Table 15-1 on page 15-2.

4.4 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 cores. 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.

4.4.1 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.

4.4.2 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.

Example 4-3 A simple example showing use of a label

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

In Example 4-3, `Loop` is a label and the conditional branch instruction (`BNE Loop`) will be 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`.

4.4.3 Directives

Most lines will normally have an assembly language instruction, to be converted by the tool into its binary equivalent, or a directive that tells the assembler to do something. It can also be a pseudo-instruction (one that will be converted into one or more real instructions by the assembler). We'll look at the actual instructions available in hardware in Chapter 5 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 you 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, we 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 we can use to get the address of this data. Similarly, we might have data items expressed in hex:

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

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

```
CtrlD EQU 4
TUBE EQU 0x30000000
```

We can then use these names 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 we change the address or value of something in a piece of code, we must 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 require 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 cores.

`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.

4.5 Interworking

When the core 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 core in a particular state can only execute instructions from that instruction set. We must make sure that the core 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 (described in Chapter 15). 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 3-6 on page 3-8. 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. Both ARM (aligned to a word boundary) and Thumb (aligned to a halfword boundary) instructions do not use bit [0] to form an instruction 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` will 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, we 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, we 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.

4.6 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 will be 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 Chapter 5. 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 code types, such as .c, .h, .cpp, .cxx, and .c++ can be built for Thumb or ARM, depending on GCC configuration and command-line switches (`-marm` or `-mthumb`).

4.7 Compatibility with ARMv8-A

With the introduction of the ARMv8-A architecture a number of changes have been made to the ARMv7 ISA to provide backward compatibility. The following instructions have been added to the ARMv7 ISA.

Table 4-2 New instructions

Instruction	Description
LDA	Load-Acquire Word
LDAB	Load-Acquire Byte
LDAEX	Load-Acquire Exclusive Word
LDAEXB	Load-Acquire Exclusive Byte
LDAEXD	Load-Acquire Exclusive Double
LDAEXH	Load-Acquire Exclusive Halfword
LDAH	Load-Acquire Halfword
STL	Store-Release Word
STLB	Store-Release Byte
STLEX	Store-Release Exclusive Word
STLEXB	Store-Release Exclusive Byte
STLEXD	Store-Release Exclusive Double
STLEXH	Store-Release Exclusive Halfword
STLH	Store-Release Halfword

These are described in Appendix A *Instruction Summary*.

Chapter 5
ARM/Thumb Unified Assembly Language Instructions

This chapter is a general introduction to ARM/Thumb assembly language. It does not provide detailed coverage of every instruction, descriptions of individual instructions can be found in Appendix A *Instruction Summary*.

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

- *Data processing operations* (ALU operations such as `ADD`).
- *Memory access* (load and stores to memory).
- *Control flow* (for loops, goto, conditional code and other program flow control).
- *System* (coprocessor, debug, mode changes and so forth).

We will take a brief look at each of these in turn. Before that, let us examine capabilities that are common to different instruction classes.

5.1 Instruction set basics

There are a number of features common to all parts of the instruction set.

5.1.1 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. You 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` (see *ROR* on page A-35). 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 that is able 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 `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, that 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.

5.1.2 Conditional execution

A feature of the ARM instruction set is that nearly all instructions can be 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 Example 5-1. The suffix LT indicates that the instruction must be executed only if the most recent flag-setting instruction returned *less than*; GE means *greater than or equal*.

Example 5-1 Example code showing branches (GNU)

```
@ Code using branches
  CMP     R0, R1
  BLT     .Lsmaller   @ if R0<R1 jump over
  MOV     R2, R1      @ R1 is greater 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, in Example 5-2

Example 5-2 Same example using conditional execution

```
  CMP      R0, R1
  MOVGE    R2, R1     @ R1 is less thanor equal to R0
  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 processors like the Cortex-A9, 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 in Example 5-2 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 will be 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-2 technology also introduced an 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 (see *IT* on page A-12).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 5-1 lists the 15 condition codes that can be attached to most instructions.

Table 5-1 Condition code suffixes

Suffix	Meaning	Flags
EQ	Equal	Z = 1
NE	Not equal	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
MI	Minus or negative result	N = 1
PL	Positive or zero result	N = 0
VS	Overflow	V = 1
VC	Now overflow	V = 0
HI	Unsigned higher	C = 1 AND Z = 0
LS	Unsigned lower or same	C = 0 OR Z = 1
GE	Signed greater than or equal	N = V
LT	Signed less than	N != V

Table 5-1 Condition code suffixes (continued)

Suffix	Meaning	Flags
GT	Signed greater than	Z = 0 AND N = V
LE	Signed less than or equal	Z = 1 OR N != V
AL	Always. This is the default	-

5.1.3 Status flags and condition codes

Program Status Registers on page 3-8, stated that 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 5-2 indicates the value of these bits for flag setting operations.

Table 5-2 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 will be 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, 2 is added to this value, the result is `0x80000001`, a large negative number. The V bit is set to indicate the overflow or underflow, from bit [30] to bit [31].

5.2 Data processing operations

These are essentially the fundamental arithmetic and logical operations of the core. *Multiplication operations* on page 5-8 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 core.

ARM cores 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 5-3 summarizes the data processing assembly language instructions, giving their mnemonic opcode, operands and a brief description of their function. Appendix A gives a fuller description of all of the available instructions.

Table 5-3 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 will be readily 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` 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 performs 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).

5.2.1 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: 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 in 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 *n*th 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. Example 5-3 on page 5-8 provides examples of differing operand 2 types used in ARM instructions.

Example 5-3 ARM instructions showing a variety of operand 2 types

```
add     R0, R1, #1              @ R0 = R2 + 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
```

5.2.2 Multiplication operations

The multiply operations are readily understandable. A key limitation to note 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 5-4 summarizes the multiplication assembly language instructions, giving their mnemonic opcode, operands and a brief description of their function.

Table 5-4 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 64-bit multiply	RdHiLo = Rn × Rm
UMLAL	RdLo, RdHi, Rn, Rm	Unsigned 64-bit MAC	RdHiLo += Rn × Rm
UMULL	RdLo, RdHi, Rn, Rm	Unsigned 64-bit multiply	RdHiLo = Rn × Rm

5.2.3 Additional multiplies

Multiplication operations provide the means 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 might 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.

5.2.4 Integer SIMD instructions

Single Instruction, Multiple Data (SIMD) instructions were first added in the ARMv6 architecture and 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. These must not be confused with the

significantly more powerful Advanced SIMD (NEON) operations that were introduced in the ARMv7 architecture and are covered in detail in Chapter 7 and the *ARM® NEON™ Programmer's Guide*.

Integer register SIMD instructions

The ARMv6 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 result and set the N, Z, C and V flags (as seen in Figure 3-6 on page 3-8). 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 is that subword quantities in each register are operated on in parallel (for example, four byte-sized `ADD`s 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.

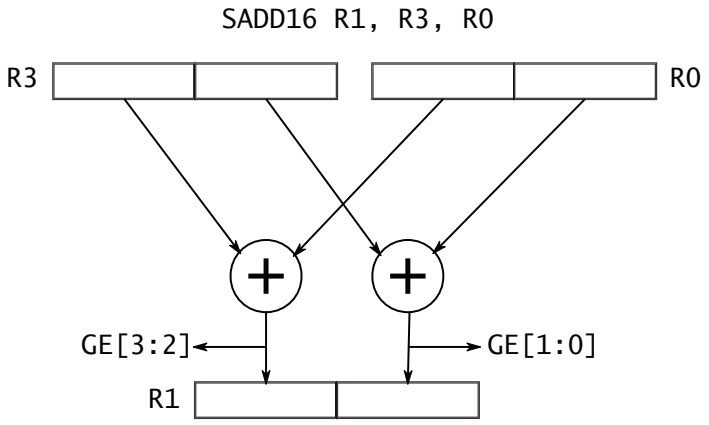


Figure 5-1 SIMD ADD v6

The `SADD16` instruction shown in Figure 5-1 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 overflow information is duplicated in the specified pair of bits.

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 and add or subtract 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 5-2 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.

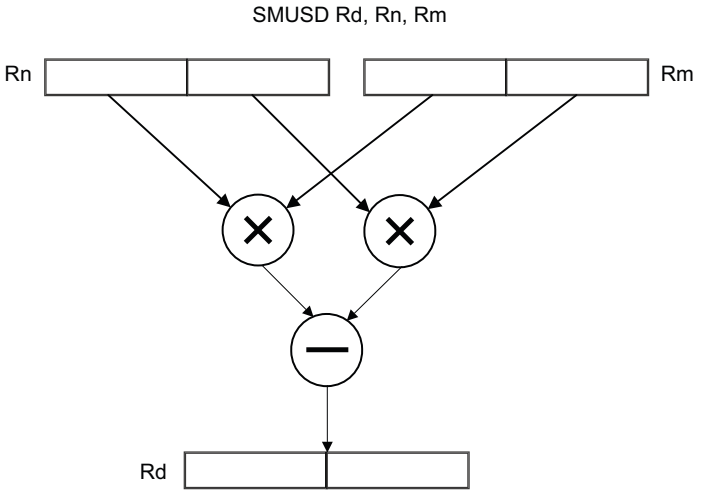


Figure 5-2 v6 SIMD signed dual multiply subtract

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 5-3 on page 5-11. 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`.

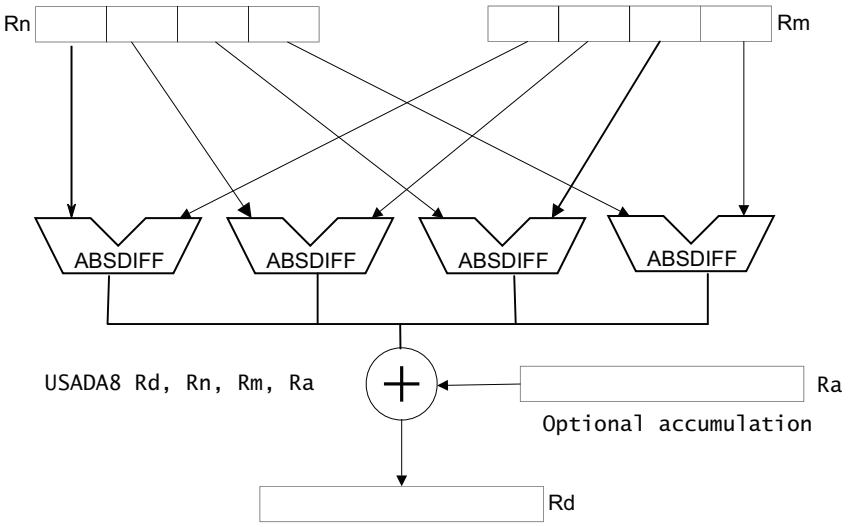


Figure 5-3 Sum of absolute differences

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 might 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 permit 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.

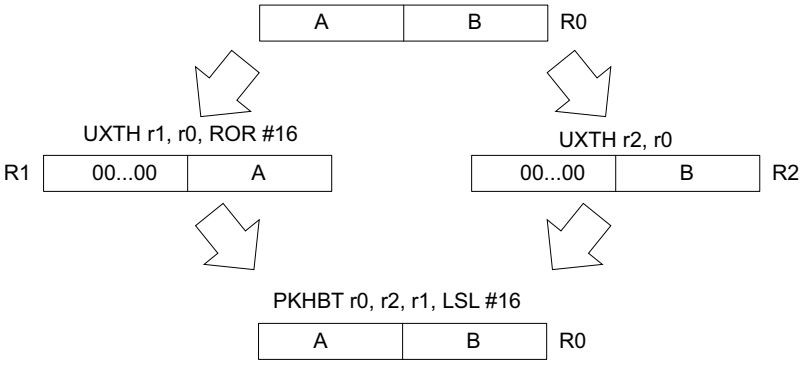


Figure 5-4 Packing and unpacking of 16-bit data in 32-bit registers

In the example shown in Figure 5-4, 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 future processing and 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 a single instruction intended to work on parts of registers is used.

Byte selection

The `SEL` instruction enables you 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.

5.3 Memory instructions

ARM cores perform *Arithmetic Logic Unit* (ALU) operations only on registers. The only supported memory operations are the *load* (that reads data from memory into registers) or *store* (that writes 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). See *LDR* on page A-18 for examples of this. 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. Unsigned numbers are zero-extended (that is, the most significant 16 or 24 bits of the register are set 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.

5.3.1 Addressing modes

There are multiple addressing modes that can be used for loads and stores. The number in parentheses refers to Example 5-4:

- *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).

Example 5-4 Examples of addressing modes

```
(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
```

5.3.2 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 operated on in this way and a word aligned address must be used.

The operands are a base register (with an optional `!` denoting write-back of the base register) with a list of registers between braces. 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, 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 might also be specified using aliases (FD, FA, ED and EA) which 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 5-5 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.

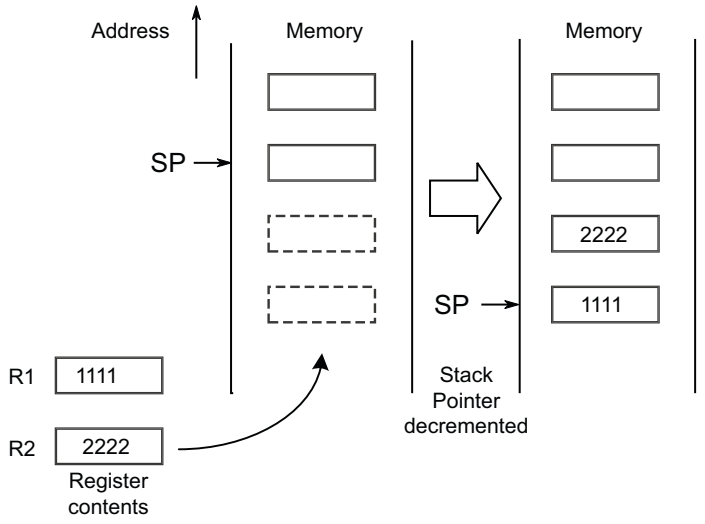


Figure 5-5 Stack push operation

5.4 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, that 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.

5.5 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 enable such algorithms.

5.5.1 Saturated arithmetic instructions

The ARM saturated arithmetic instructions can operate on byte, word or halfword sized values. For example, the 8 in the `QADD8` and `QSUB8` instructions indicate that they operate on byte sized values. The result of the operation will be 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.

5.6 Miscellaneous instructions

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

5.6.1 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-A series.

- Coprocessor 15 is a built-in coprocessor that provides control over many core features, including cache and MMU.
- Coprocessor 14 is a built-in coprocessor that controls the hardware debug facilities of the core, such as breakpoint units (described in Chapter 24).
- Coprocessors 10 and 11 give access to the floating-point and NEON hardware in the system (described in Chapter 6 and Chapter 7).

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 a coprocessor register from multiple registers,
- `STCL` – writes a coprocessor register to multiple registers,

These and other variants are described more fully in Appendix A *Instruction Summary*.

5.6.2 SVC

The `SVC` (supervisor call) instruction, when executed, causes a supervisor call exception. This is described in Chapter 11 *Exception Handling*. 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).

5.6.3 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`) can 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. See Figure 3-6 on page 3-8.
- `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.

5.6.4 Bit manipulation

There are instructions that enable 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.

5.6.5 Cache preload

Cache preloading is described in Chapter 17 *Optimizing Code to Run on ARM Processors*. 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. Implementations that do not support these operations will treat a preload as a `NOP`, but all of the Cortex-A family processors described in this book are able to preload the cache. Any illegal address specified as a parameter to the `PLD` instruction will not result in a data abort exception.

5.6.6 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 5-6 on page 5-19 illustrates the operation of the `REV` instruction, showing how four bytes within a register have their ordering within a word reversed.

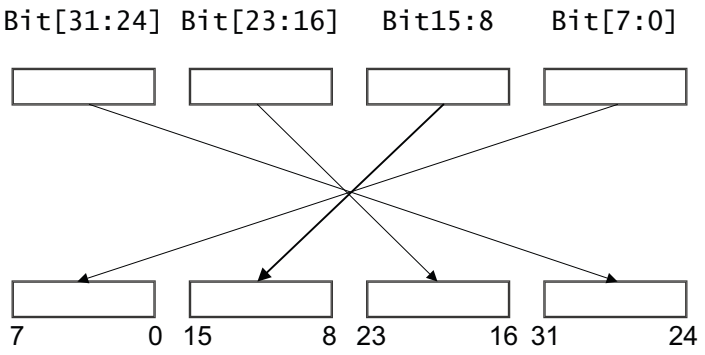


Figure 5-6 Operation of the REV instruction

5.6.7 Other instructions

A few other instructions are available:

- The breakpoint instruction (`BKPT`) will either cause a prefetch abort (see *Types of exception* on page 11-3) or cause the core 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 events* on page 24-2.
- Wait For Interrupt (`WFI`) puts the core into standby mode, that is described in Chapter 20 *Power Management*. The core stops execution until woken by an interrupt or debug event. If `WFI` is executed with interrupts disabled, an interrupt will still wake the core, but no interrupt exception is taken. The core proceeds to the instruction after the `WFI`. In older ARM processors, `WFI` was implemented as a CP15 operation.
- No operation (`NOP`) does nothing. It is not guaranteed to take time to execute, so the `NOP` instruction must 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 REV instruction. An interrupt or a bug event will cause the core to wake up. `WFE` (Wait for Event) is also described in Chapter 20.
- The `SEV` (Send Event) instruction is used to generate wake-up events that might wake-up other cores in the cluster.

Chapter 6
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.

6.1 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 has since been updated with the publication of IEEE-754-2008. The standard defines precisely what result will be 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* on page 6-4). IEEE-754 also specifies the outcome when an *exceptional* operation occurs. This means a calculation that 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

The specification also describes what action must be taken when one of these exceptional operations is detected. Possible outcomes include the generation of a *Not a Number* (NaN) result for invalid operations, positive or negative infinity, for overflow or division by zero, or *denormalized* numbers in the case of underflow. The standard additionally 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 (see *VFP* on page 2-6) 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 6-1 on page 6-3.

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 6-1 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. Another special case is that of denormalized values.

The sign bit lets you 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 6-1 summarizes this.

Table 6-1 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, but 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.

6.1.1 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* $+\infty$ .This selects rounding towards positive infinity.
- *Round toward* $-\infty$. 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.

6.1.2 ARM VFP

VFP is an optional but rarely omitted extension to the instruction sets in the ARMv7-A architecture. 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. If the Advanced SIMD (NEON) extension is implemented together with VFPv3, VFPv3-D32 is always present. 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 enable 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-A series processors is given in Table 2-3 on page 2-9.

In addition to the registers described, there are a number of other VFP registers:

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 through `/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.

Example 6-1 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.

Example 6-1 Example code illustrating usage of floating-point flags

```
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 6-2 ARM APSR flags

IEEE-754 relationship	ARM APSR flags			
	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 (and NEON) 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 6-3 describes floating-point comparisons rather than integer comparisons:

Table 6-3 Interpreting the flags

Code	Meaning (when set by vcmp)	Meaning (when set by cmp)	Flags tested
EQ	Equal to	Equal to	Z =1
NE	Unordered, or not equal to	Not equal to.	Z = 0
CS or HS	Greater than, equal to, or unordered	Greater than or equal to (unsigned).	C = 1
CC or LO	Less than.	Less than (unsigned).	C = 0
MI	Less than	Negative.	N = 1
PL	Greater than, equal to, or unordered	Positive or zero.	N = 0
VS	Unordered. (At least one argument was NaN.)	Signed overflow.	V = 1
VC	Not unordered. (No argument was NaN.)	No signed overflow.	V = 0
HI	Greater than or unordered	Greater than (unsigned).	(C = 1) && (Z = 0)
LS	Less than or equal to	Less than or equal to (unsigned).	(C = 0) || (Z = 1)
GE	Greater than or equal to.	Greater than or equal to (signed).	N==V
LT	Less than or unordered.	Less than (signed).	N!=V
GT	Greater than.	Greater than (signed).	(Z==0) && (N==V)
LE	Less than, equal to or unordered.	Less than or equal to (signed).	(Z==1) || (N!=V)
AL (or omitted)	Always executed.	Always executed.	None tested.

It is clear 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. It is the meaning of the flags that differs when you perform a vcmp rather than a cmp. 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.

6.1.3 Enabling VFP

If an ARMv7-A core includes VFP hardware, it must be explicitly enabled before applications can make use of it. Several steps are required to do this:

- The EN bit in the FPEXC register must be set.
- If access to VFP is required in the Normal world, access to CP10 and CP11 must be enabled in the *Non-Secure Access Control Register* (CP15.NSACR). This would normally be done inside the Secure bootloader.
- 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.

6.2 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. As does specifying the option `-mfpu=neon`.

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

- `-mfpu=vfpv3` or `-mfpu=vfpv3-d16` (for Cortex-A8 and Cortex-A9 processors).
- `-mfpu=vfpv4` or `-mfpu=vfpv4-d16` (for Cortex-A5 and Cortex-A15 processors).

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.

- `-mfloat-abi=softfp` (or `hard`) specify which ABI to use to enable the use of VFP.

`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. See *VFP and NEON register use* on page 15-4 for more information on efficient parameter passing.

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

6.3 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-A series processor are:

- `--fpu=vfpv3` or `--fpu=vfpv3_d16` (for the Cortex-A8 and Cortex-A9 processors).
- `--fpu=vfpv4` or `--fpu=vfpv4_d16` (for all other Cortex-A series processors).

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`.

6.4 VFP support in Linux

An application that uses VFP (or calls into a library that uses VFP) places some additional requirements on the Linux kernel. For the application to run correctly, the kernel must save and restore the VFP registers during context switches. The kernel might also have to decode and emulate VFP instructions where the VFP hardware is not present.

6.4.1 Context switching

In addition to saving and restoring integer registers, the kernel might also have to perform saving and restoring of VFP registers on a context switch. To avoid wasting cycles, this is done only when an application actually used VFP. Because the VFP initialization code leaves VFP disabled, the first time a thread actually tries to access the floating-point hardware, an undefined exception occurs. The `kernel` function that handles this, sees that VFP is disabled and that a new thread wants to use VFP. It saves the current VFP state and restores the state for the new thread.

On clusters, where threads can migrate to a different core, this simple system will no longer work correctly. Instead, the kernel saves the state if the VFP was used by the previous thread.

6.5 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 core can be sub-optimal on different hardware.

- Avoid mixing of VFP and NEON instructions on the Cortex-A9 processor, as there is a significant overhead in switching between data engines.
- 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 must not be placed in time-critical loops, if possible. For example, accesses to control registers on the Cortex-A9 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 must similarly be avoided in time-critical loops. For the Cortex-A8 processor, this is particularly true of register transfers from VFP registers to integer registers.
- 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.

Chapter 7
Introducing NEON

NEON technology provides *Single Instruction Multiple Data* (SIMD) operations in ARM processors implementing the Advanced SIMD architecture extensions and can be used to accelerate the performance of multimedia applications running on ARM Cortex-A series processors. These operations can significantly accelerate repetitive operations on large data sets. This can be useful in applications such as media codecs.

NEON is implemented as a separate hardware unit that is available as an option on Cortex-A series processors. Making the NEON hardware optional enables ARM SoCs to be optimized for specific markets. In most general-purpose applications processor SoCs, NEON will probably be included. However, for an embedded application such as a network router, NEON can be omitted, enabling a small saving in silicon area that translates to a small cost saving.

7.1 SIMD

SIMD is a computational technique for processing a number of data values (generally a power of two) using a single instruction, with the data for the operands packed into special wide registers. One instruction can therefore do the work of many separate instructions. This type of parallel processing instruction is commonly called SIMD (single instruction, multiple data), 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.

Single Instruction Multiple Data (SIMD)

A technique for processing multiple data values using a single instruction, with the data for the operands packed into wide registers. One instruction can therefore do the work of many. SIMD instructions are very powerful for computations on media data.

Single Instruction, Single Data (SISD)

A single core executes a single instruction stream, to operate on data stored in a single memory one operation at a time. There is often a central controller that broadcasts the instruction stream to all the processing elements. Almost all ARM processors prior to the ARMv6 architecture use SISD processing.

Multiple Instruction, Single Data (MISD)

A type of parallel computing architecture where many functional units perform different operations on the same data. Fault-tolerant computers executing the same instructions in order to detect and mask errors might be considered to belong to this type. One example of this was the Space Shuttle flight control computers

Multiple Instruction, Multiple Data (MIMD)

Multiple computer instructions, that might be the same, and that might be synchronized with each other, perform actions simultaneously on two or more pieces of data. A multi-core superscalar processor is an MIMD processor.

For code that can be parallelized, large performance improvements can be achieved. SIMD extensions exist on many 32-bit architectures – PowerPC has AltiVec, while x86 has several variants of MMX/SSE. SIMD is described in *Integer SIMD instructions* on page 5-8.

Many software programs operate on large datasets. The data items can be less than 32 bits in size. 8-bit pixel data is common in video, graphics and image processing, 16-bit samples in audio codecs. In such cases, the operations to be performed are simple, repeated many times and have little requirement for control code. SIMD can offer considerable performance improvements for this type of data processing. It is particularly beneficial for digital signal processing or multimedia algorithms, such as:

- Block-based data processing.
- Audio, video, and image processing codecs.
- 2D graphics based on rectangular blocks of pixels
- 3D graphics
- Color-space conversion.
- Physics simulations.

On a 32-bit core such as the Cortex-A series processors, it is relatively inefficient to perform large numbers of 8-bit or 16-bit single operations one at a time. The processor ALU, registers and datapath are designed for the 32-bit calculations. SIMD enables a single instruction to treat a register value as multiple data elements (for example, as four 8-bit values in a 32-bit register) and to perform multiple identical operations on those elements.

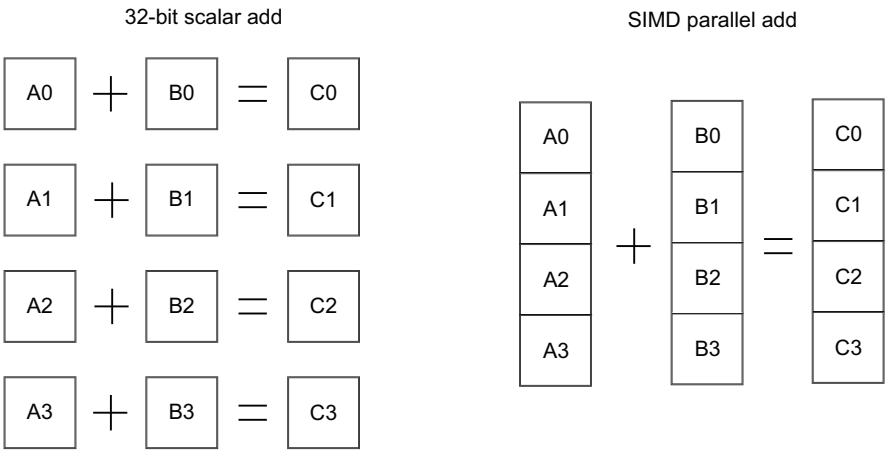


Figure 7-1 Comparing SIMD parallel add with 32-bit scalar add

To achieve four separate additions without using SIMD requires you to use four `ADD` instructions, as shown in Figure 7-1, and additional instructions to prevent one result from overflowing into the adjacent byte. SIMD requires only one instruction to do this.

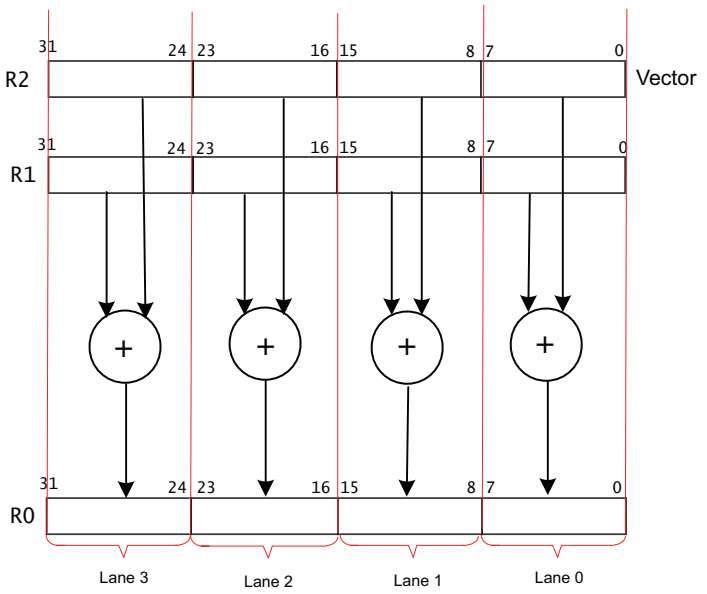


Figure 7-2 4-way 8-bit add operation

Figure 7-2 shows the operation of the SIMD `UADD8 R0, R1, R2` instruction. This operation performs a parallel addition of four pairs of 8-bit elements (called lanes) packed into vectors stored in general purpose registers R1 and R2, and places the result into a vector in register R0.

It is important to note that the addition operations are truly independent. Any overflow or carry from bit [7] of lane 0 does not affect bit [0] of lane 1 (bit [8] of the whole register), which is a separate calculation.

ARM NEON technology is designed to build on the concept of SIMD. NEON is a combined 64-bit and 128-bit SIMD instruction set that provides 128-bit wide vector operations, compared to the 32-bit SIMD in the ARMv6 architecture. NEON technology introduced in the ARMv7 architecture is at present only available with ARM Cortex-A and Cortex-R series processors. It is a SIMD technology targeted at advanced media and signal processing applications and embedded processors, and can accelerate multimedia and signal processing algorithms such as video encode/decode, 2D/3D graphics, gaming, audio and speech processing, image processing, telephony, and sound synthesis by at least twice the performance of ARMv6 SIMD.

NEON is included by default in the Cortex-A7, Cortex-A12, and Cortex-A15 processors, but is optional in all other ARMv7 Cortex-A series processors. NEON can execute MP3 audio decoding on processors running at 10 MHz. It features a comprehensive instruction set, a separate register file and independent execution hardware. NEON supports 8-, 16-, 32- and 64-bit integer and single-precision (32-bit) floating-point data and SIMD operations for handling audio and video processing as well as graphics and gaming processing. NEON supports up to 16 operations at the same time. The NEON hardware shares the same registers as used in VFP. It is implemented as part of the ARM processor, but has its own execution pipelines and a register bank that is distinct from the ARM core register bank. NEON data is organized into very long registers (64 or 128 bits wide). These registers can hold data items that are 8, 16, 32 or 64 bits long.

7.2 NEON architecture overview

NEON was designed as an additional load/store architecture to provide good vectorizing compiler support from languages such as C/C++. A rich set of NEON instructions operate on wide 64-bit and 128-bit vector registers, enabling a high level of parallelism. The NEON instructions are straightforward and easy to understand, that also makes hand-coding easy for applications that require the very highest performance.

A key advantage of NEON technology is that instructions form part of the normal ARM or Thumb code, making programming simpler than with an external hardware accelerator. There are NEON instructions available to read and write external memory, move data between NEON registers and other ARM registers and to perform SIMD operations.

The NEON architecture uses a 32 × 64-bit register file. These are actually the same registers used by the floating-point unit (VFPv3). It does not matter that the floating-point registers are re-used as NEON registers. All compiled code and subroutines will conform to the EABI, that specifies which registers can be corrupted and which registers must be preserved. The compiler is free to use any NEON or VFPv3 registers for floating-point values or NEON data at any point in the code.

The NEON architecture permits 64-bit or 128-bit parallelism. This choice was made to keep the size of the NEON unit manageable (a vector ALU can easily become quite large), while still offering good performance benefits from vectorization. The NEON architecture also does not specify instruction timings and might require different numbers of cycles to execute the same instruction on different processors.

7.2.1 Commonality with VFP

The ARM architecture can support a wide range of different NEON and VFP options, but in practice you see only the combinations:

- No NEON or VFP.
- VFP only.
- NEON and VFP.

These are vendor implementation options for the architecture, and so are fixed for a particular implementation of an ARM-based design.

The key differences between NEON and VFP are that NEON only works on vectors, does not support double-precision floating-point (double-precision is supported by the VFP), and does not support certain complex operations such as square root and divide. NEON has a register bank of thirty-two 64-bit registers. If both NEON and VFPv3 are implemented, this register bank is shared between them in hardware. This means that VFPv3 must be present in its VFPv3-D32 form, that has 32 double-precision floating-point registers. This makes support for context switching simpler. Code that saves and restores VFP context also saves and restores NEON context.

7.2.2 Data types

NEON instructions operate on elements of the following types:

- 32-bit single precision floating-point
- 8, 16, 32 and 64-bit unsigned and signed integers
- 8 and 16-bit polynomials.

Data type specifiers in NEON instructions comprise a letter indicating the type of data and a number indicating the width. They are separated from the instruction mnemonic by a point, for example, `VMLAL.S8`. So you have the following possibilities:

- Unsigned integer U8 U16 U32 U64.
- Signed integer S8 S16 S32 S64.
- Integer of unspecified type I8 I16 I32 I64.
- Floating-point number F16 F32.
- Polynomial over {0,1} P8.

Note

F16 is not supported for data processing operations. It is only supported as a format to be converted to, or from.

Polynomial arithmetic is useful when implementing certain cryptography or data integrity algorithms.

Adding two polynomials over {0,1} is the same as a bitwise exclusive OR. Polynomial additional results in different values to a conventional addition.

Multiplying two polynomials over {0,1} involves first determining the partial products as done in conventional multiply, then the partial products are exclusive ORed instead of being added conventionally. Polynomial multiplication results in different values to conventional multiplication because it requires polynomial addition of the partial products.

NEON technology is IEEE 754-1985 compliant, but only supports round-to-nearest rounding mode. This is the rounding mode used by most high-level languages, such as C and Java. Additionally, NEON instructions always treats denormals as zero.

7.2.3 NEON registers

The register bank can be viewed as either sixteen 128-bit registers (Q0-Q15) or as thirty-two 64-bit registers (D0-D31). Each of the Q0-Q15 registers maps to a pair of D registers, as shown in Figure 7-3.

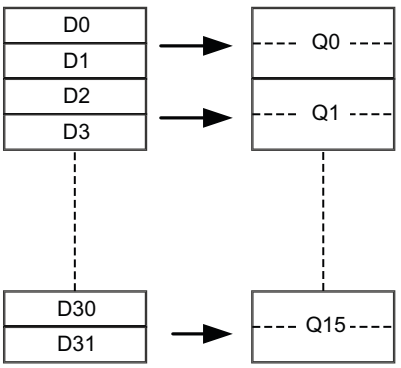


Figure 7-3 NEON register bank

The view of registers in Figure 7-4 on page 7-7 is determined by form of the instruction used. so that the software does not have to explicitly change state.

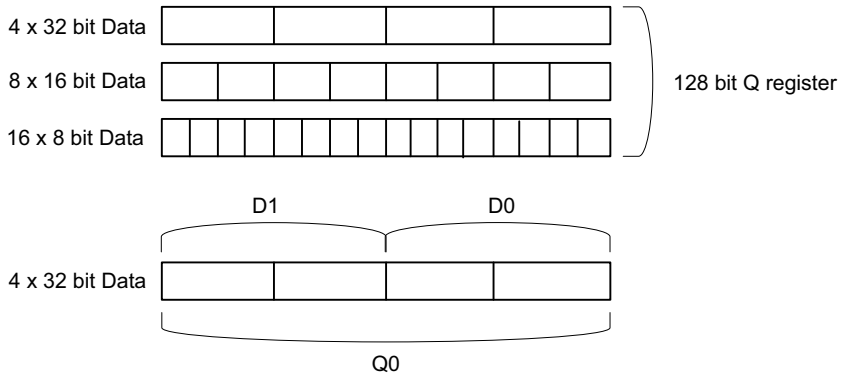


Figure 7-4 NEON registers

Individual elements can also be accessed as scalars.The advantage of the dual view is that it accommodates mathematical operations that widen or narrow the result. For example multiplying two D registers gives a Q register result. The dual-view enables the register bank to be used more efficiently.

NEON data processing instructions are typically available in Normal, Long, Wide, Narrow and Saturating variants.

- *Normal* instructions can operate on any vector types, and produce result vectors the same size, and usually the same type, as the operand vectors.
- *Long* instructions operate on doubleword vector operands and produce a quadword vector result. The result elements are usually twice the width of the operands, and of the same type. Long instructions are specified using an L appended to the instruction. Figure 7-5 shows this, with input operands being promoted before the operation.

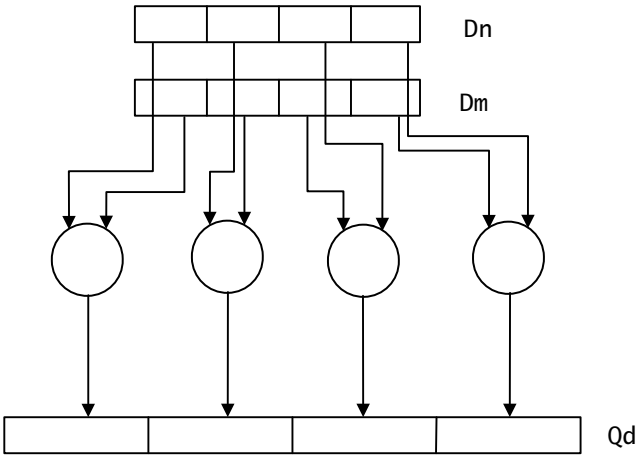


Figure 7-5 NEON long instructions

- *Wide* instructions operate on a doubleword vector operand and a quadword vector operand, producing a quadword vector result. The result elements and the first operand are twice the width of the second operand elements. Wide instructions have a W appended to the instruction. Figure 7-6 on page 7-8 shows this, with the input doubleword operands being promoted before the operation.

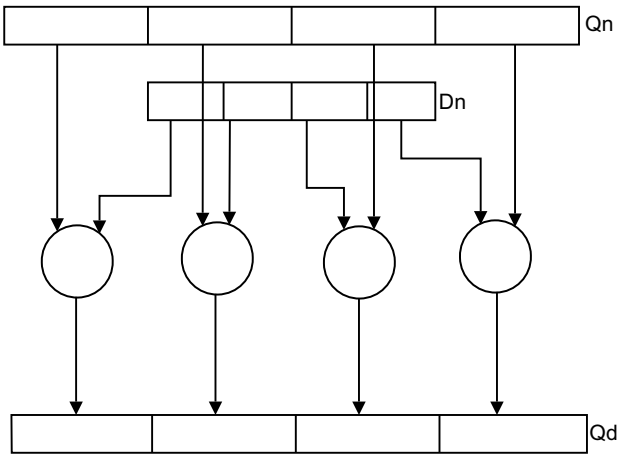


Figure 7-6 NEON wide instructions

- *Narrow* instructions operate on quadword vector operands, and produce a doubleword vector result. The result elements are usually half the width of the operand elements. Narrow instructions are specified using an N appended to the instruction. Figure 7-7 shows this, with input operands being demoted before the operation.

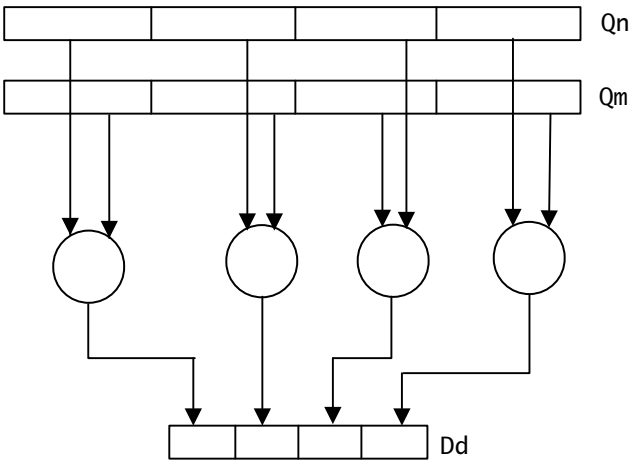


Figure 7-7 NEON narrow instructions

Some NEON instructions act on scalars together with vectors. The scalars can be 8-, 16-, 32-, or 64-bit. Instructions that use scalars can access any element in the register bank, although there are differences for multiply instructions. The instruction uses an index into a doubleword vector to specify the scalar value. Multiply instructions only support 16-bit or 32-bit scalars, and can only access the first 32 scalars in the register bank (that is, D0-D7 for 16-bit scalars or D0-D15 for 32-bit scalars).

7.2.4 NEON instruction set

All mnemonics for NEON instructions (as with VFP) begin with the letter “V”. Instructions are generally able to operate on different data types, with this being specified in the instruction encoding. The size is indicated with a suffix to the instruction. The number of elements is indicated by the specified register size.

For example, looking at the instruction

```
VADD.I8 D0, D1, D2
```

where:

- VADD indicates a NEON ADD operation.
- The I8 suffix indicates that 8-bit integers are to be added
- D0, D1 and D2 specify the 64-bit registers used (D0 for the result destination, D1 and D2 for the operands).

So this instruction performs eight additions in parallel.

There are operations that have different size registers for input and output.

```
VMULL.S16 Q2, D8, D9
```

This instruction performs four 16-bit multiplies of data packed in D8 and D9 and produces four 32-bit results packed into Q2.

The VCVT instruction converts elements between single-precision floating-point and 32-bit integer, fixed-point, and (if implemented) half-precision floating-point.

NEON includes load and store instructions that can load or store individual or multiple values to a register. In addition, there are instructions that can transfer blocks of data between multiple registers and memory. It is also possible to interleave or de-interleave data during such multiple transfers.

The following modifiers can be used with certain Advanced SIMD instructions (some modifiers can be used only with a small subset of the available instructions):

Q The instruction uses saturating arithmetic, so that the result is saturated within the range of the specified data type. The sticky QC bit in the FPSCR is set if saturation occurs in any lane. VQADD is an example of such an instruction.

H The instruction will halve the result. It does this by shifting right by one place (effectively a divide by two with truncation). VHADD is an example of such an instruction – it could be used to calculate the mean of two inputs.

D The instruction doubles the result and saturates. This is commonly required when multiplying numbers in Q15 format, where an additional doubling is required to get the result into the correct form.

R The instruction will perform rounding on the result, equivalent to adding 0.5 to the result before truncating. VRHADD is an example of this.

Instructions have the following general format:

```
V{<mod>}<op>{<shape>}{<cond>}{.<dt>}{<dest>}, src1, src2
```

where:

<mod> is one of the previously described Modifiers (Q, H, D, R)

<op> - operation (for example, ADD, SUB, MUL)

<shape> - Shape (L, W or N, as described in *NEON registers* on page 7-6)

<cond> - Condition, used with IT instruction

<.dt> - Data type

<dest> - Destination

<src1> - Source operand 1

<src2> - Source operand 2.

The NEON instruction set includes a range of vector addition and subtraction operations, including pairwise adding, that adds adjacent vector elements together.

There are a number of multiply operations, including multiply-accumulate and multiply-subtract and doubling and saturating options. There is no SIMD division operation, but such an operation can be performed using the VRECPE (Vector Reciprocal Estimate) and VCREPS (Vector Reciprocal Step) instructions to perform Newton-Raphson iteration.

Similarly, there is no vector square root instruction, but VRSQRTE, VRSQRTS, and multiplies can be used to compute square roots. Shift left, right and insert operations are also available, along with instructions that select minimum or maximum values. Common logic operations (AND, OR, EOR, AND NOT and OR NOT) can be performed. The instruction set also includes the ability to count numbers of bits set in an element or to count leading zeros or sign bits.

There are a number of different instructions to move data between registers, or between elements. It is also possible for instructions to swap or duplicate registers, to perform reversal, matrix transposition and extract individual vector elements.

7.3 NEON C Compiler and assembler

Code targeted at NEON hardware can be written in C or in assembly language and a range of tools and libraries is available to support this.

In many cases, it might be preferable to use NEON code within a larger C/C++ function, rather than in a separate file to be processed by the assembler. This can be done using NEON intrinsics.

7.3.1 Vectorization

A vectorizing compiler can take your C or C++ source code and parallelize it in a way that enables efficient usage of NEON hardware. This means you can write portable C code, while still obtaining the levels of performance made possible by NEON. The C language does not specify parallelizing behavior, so it can be necessary to provide hints to the compiler about this. For example, it might be necessary to use the `__restrict` keyword when defining pointers. This has the effect of guaranteeing that pointers will not address overlapping regions of memory. It can also be helpful to ensure that the number of loop iterations is a multiple of four or eight. Automatic vectorization is specified with the GCC option `-ftree-vectorize` (along with `-mfpu=neon`). Using the ARM Compiler, you must specify optimization level `-O2` (or `-O3`), `-Otime` and `--vectorize`.

7.3.2 Detecting NEON

As NEON hardware can be omitted from a processor implementation, it might be necessary to test for its presence.

Build-time NEON selection

This is the easiest way to select NEON. In armcc (RVCT 4.0 and later), or GCC, the predefined macro `__ARM_NEON__` is defined when a suitable set of processor and FPU options is provided to the compiler. The armasm equivalent predefined macro is `TARGET_FEATURE_NEON`.

This could be used to have a C source file that has both NEON and non-NEON optimized versions.

Run-time NEON detection

To detect NEON at run-time requires help from the operating system, since the ARM architecture intentionally does not expose processor capabilities to user-mode applications.

Under Linux, `/proc/cpuinfo` contains this information in human-readable form.

On Tegra2 (a dual-core Cortex-A9 processor with FPU), `cat /proc/cpuinfo` reports:

```
…
Features    : swp half thumb fastmult vfp edsp thumbee vfpv3 vfpv3d16
…
```

The ARM quad-core Cortex-A9 processor with NEON gives a slightly different result:

```
…
Features        : swp half thumb fastmult vfp edsp thumbee neon vfpv3
…
```

As the `/proc/cpuinfo` output is text based, it is often preferred to look at the auxiliary vector `/proc/self/auxv`. This contains the kernel `hwcap` in a binary format. The `/proc/self/auxv` file can be easily searched for the `AT_HWCAP` record, to check for the `HWCAP_NEON` bit (4096).

Some Linux distributions, for example, Ubuntu 09.10 or later, take advantage of NEON transparently. The `ld.so` linker script is modified to read the `hwcap` using `glibc`, and add an additional search path for NEON-enabled shared libraries. In the case of Ubuntu, a new search path `/lib/neon/vfp` contains NEON-optimized versions of libraries from `/lib`.

Chapter 8
Caches

The word cache derives from the French verb *cacher*, “to hide”.The application of this word to a processor is obvious – a cache is where the processor stores instructions and data, hidden from the programmer and system. In many cases, it would be true to say that the cache is transparent to, or hidden from you. But very often, as we shall see, it is important to understand the operation of the cache in detail.

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 at the same speeds as the core, but such RAM is very expensive in comparison to standard DRAM blocks, that can have thousands of times more capacity. In many ARM processor-based systems, access to external memory will take tens or even hundreds of core cycles.

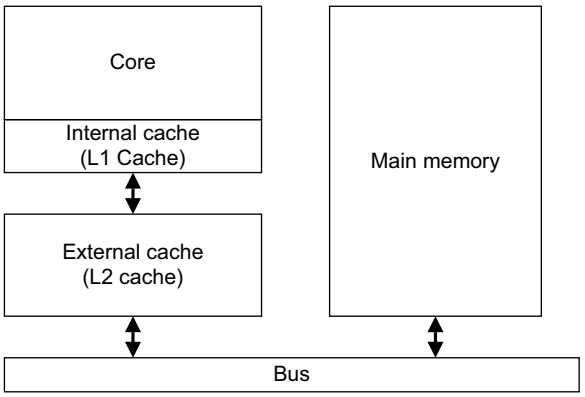


Figure 8-1 A basic cache arrangement

Essentially, a cache is a small, fast block of memory that (conceptually at least) sits between the core and main memory. It holds copies of items in main memory. Accesses to the cache memory happen significantly faster than those to main memory. Because 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. If it finds 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 need to drive external signals.

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. Efficient use of this limited resource is a key part of writing efficient applications to run on a core.

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. It is this fact that access to RAM by the core exhibits such locality, and is not truly random, that enables caches to be successful.

The write buffer is a block that decouples 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. Like the cache, it sits between the core and main memory. This enables the core to move on and execute the next instructions without having to stop and wait for the slow main memory to actually complete the write operation.

8.1 Why do caches help?

Caches speed things up, as we have seen, because program execution is not random. Programs tend to access the same sets of data repeatedly and execute the same sets of instructions repeatedly. By moving code or data into faster memory when it is first accessed, 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. It is any subsequent accesses to the cached values that are faster, and it is from this that the performance increase derives. The core hardware will check all instruction fetches and data reads or writes in the cache, although obviously you must mark some parts of memory (those containing peripheral devices, for example) as non-cacheable. Because the cache holds only a subset of main memory, you require a way to determine (quickly) whether the address you are looking for is in the cache.

8.2 Cache drawbacks

It might appear that caches and write buffers are automatically 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 main memory, it fills rapidly as a program executes. When the cache is full, existing code or data must be removed 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 to be found 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.

Furthermore, you 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 the processor can update the cache contents, which 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 particular problem when you have multiple cores or memory agents like an external DMA controller. Coherency issues are described later in the book.

8.3 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 farther 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, you typically sub-divide this 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 can improve system performance where there is an access time difference between different parts of the memory system.

In ARM processor-based systems, level 1 (L1) caches are typically connected directly to the core logic that fetches instructions and handles load and store instructions. These are Harvard caches, that is, there are separate caches for instructions and for data that effectively appear as part of the core.

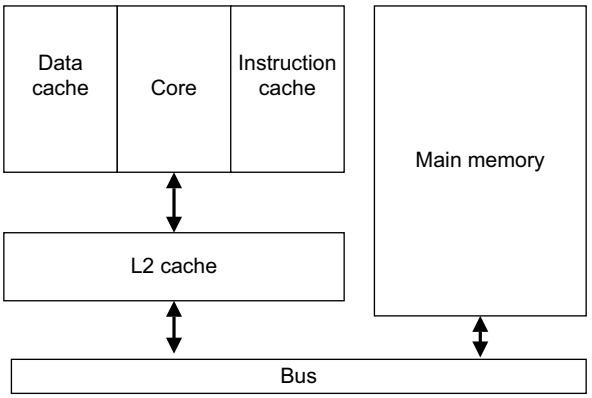


Figure 8-2 Typical Harvard cache

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.

Many ARM systems have, in addition, a 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). It can be inside the core itself, or be implemented as an external block, placed between the core and the rest of the memory system. The ARM L2C-310 is an example of such an external L2 cache controller block.

In addition, cores can be implemented in clusters in which each core has its own level 1 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 that share that memory. This is described in more detail when we look at multi-core processors.

8.4 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 and therefore there are two caches, an instruction cache (I-cache) and a data cache (D-cache). In many ARM systems, you can have distinct instruction and data level 1 caches backed by a unified level 2 cache.

The cache requires to hold an address, some data and some 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 RAMs used to hold tag values are 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 several locations are typically grouped 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 storage as a part of the tag. This will be covered in more detail later in this chapter. 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 – you require the address of a whole 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, you will have a valid bit, that marks the line as containing data that can be used. (This means that the address tag represents some real value.) In a data cache you might also have one or more dirty bits that mark whether the cache line (or part of it) holds data that is not the same as (newer than) the contents of main memory.

8.4.1 Cache terminology

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

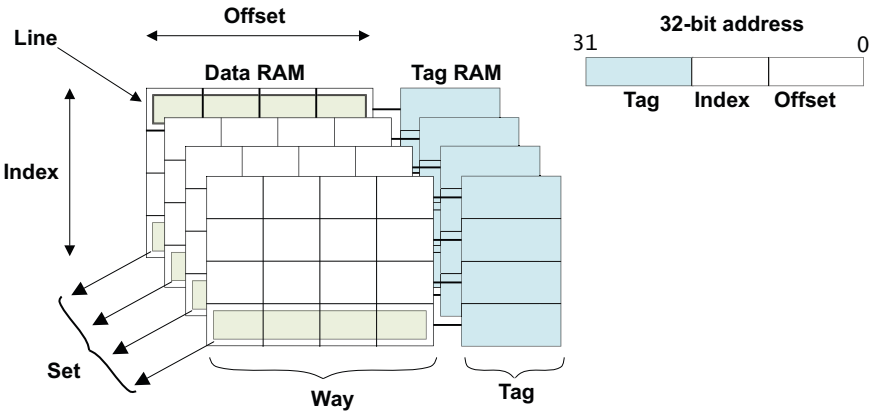


Figure 8-3 Cache terminology

- 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(s) 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.

8.4.2 Direct mapped caches

There are different ways of implementing caches, the simplest of which is 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 shows a small cache, with four words per line and four lines.

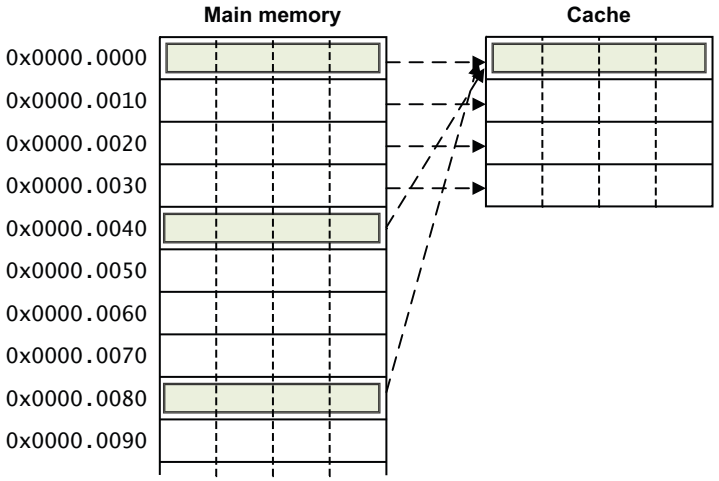


Figure 8-4 Direct mapped cache operation

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]) will be stored as a tag value.

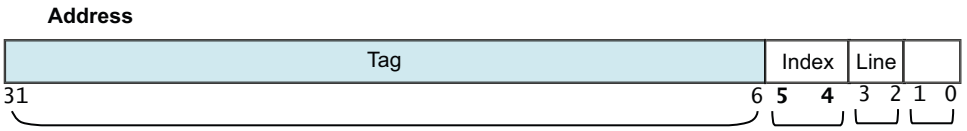


Figure 8-5 Cache address

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. If the line contains valid data, but does not generate a hit (that is, the tag shows that the cache holds a different address in main memory) 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 a problem called *thrashing* can easily occur. Consider a loop that repeatedly accesses address `0x00`, `0x40` and `0x80`, as in the following code:

```
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 on page 8-7.

- When you first read 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.
- When you then read 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 you lose the data from address `0x40` to `0x4F` from the cache.
- The result is written to `0x00`. Depending on the allocation policy this can cause another line fill. The data from `0x80` to `0x8F` might be lost.
- The same thing will happen on each iteration of the loop and our software will perform poorly. Direct mapped caches are therefore not typically used in the main caches of ARM cores, but you do see them in some places – for example in the branch target address cache of the ARM1136 processor.

Cores 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 `memcpy()`- or `memset()`-like functions 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 will be 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, you should not rely on them.

8.4.3 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 comes at the cost of increased hardware complexity and a slight increase in power (because multiple tags are compared on each cycle).

With this kind of cache organization, the cache is divided into a number of equally-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- or 4-ways, but some ARM implementations have used higher numbers.

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, you must look at each of the tags in the set.

In Figure 8-6, a 2-way cache is shown. Data from address `0x00` (or `0x40`, or `0x80`) might be found in line 0 of either (but not both) of the two cache ways.

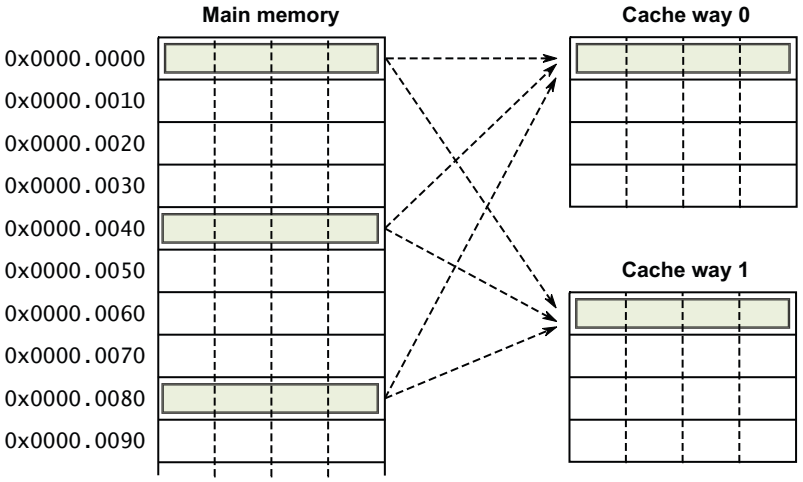


Figure 8-6 A 2-way set-associative cache

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, those associated with MMU TLBs – see Chapter 9). 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.

8.4.4 A real-life example

Before going on to look at write buffers, let's consider an example that is more realistic than those shown in the previous two diagrams. Figure 8-7 on page 8-10 is a 4-way set associative 32KB data cache, with an 8-word cache line length. This kind of cache structure can be found on the Cortex-A7 or Cortex-A9 processors.

The cache line length is eight words (32 bytes) and you have 4-ways. 32KB divided by 4 (the number of ways), divided by 32 (the number of bytes in each line) gives you a figure of 256 lines in each way. This means that you require eight bits to index a line within a way (bits [12:5]). Here, you must use bits [4:2] of the address to select from the eight words within the line, though the number of bits which are required to index into the line depends on whether you are accessing a word, halfword, or byte. The remaining bits [31:13] in this case will be used as a tag.

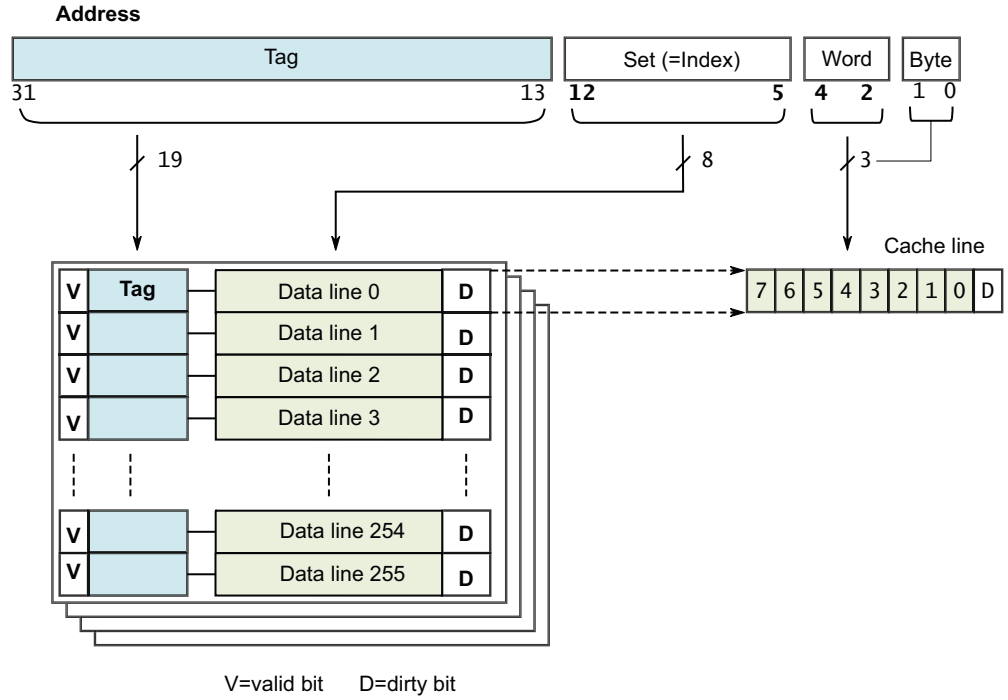


Figure 8-7 A 32KB 4-way set associative cache

8.4.5 Cache controller

This is a hardware block that has the task of managing the cache memory, in a way that is (largely) 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 to be found 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 (a hit) 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 – 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 a software developer.

The core need not 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 you perform a load instruction that misses in the cache and triggers a cache linefill, the core first retrieves that part of the cache line which contains the requested data. This critical data is supplied to the core pipeline, while the cache hardware and external bus interface then read the rest of the cache line, in the background.

8.4.6 Virtual and physical tags and indexes

This section assumes some knowledge of the address translation process. Readers unfamiliar with virtual addressing might want to revisit this section after reading Chapter 9.

A real-life example on page 8-9 was a little imprecise about specification of exactly which address is used to perform cache lookups. Early ARM processors such as the ARM720T or ARM926EJ-S processors used virtual addresses to provide both the index and tag values. This has the advantage that the core can do a cache look-up without the need for a virtual to physical address translation. The drawback is that changing the virtual to physical mappings in the system means that the cache must first be cleaned and invalidated, and this can have a significant performance impact. *Invalidating and cleaning cache memory* on page 8-17 goes into more detail about these terms.

ARM11 family processors use a different cache tag scheme. Here, the cache index is still derived from a virtual address, but the tag is taken from the physical address. The advantage of a physical tagging scheme is that changes in virtual to physical mappings do not now require the cache to be invalidated. This can have significant benefits for complex multi-tasking operating systems that can frequently modify translation table mappings. Using a virtual index has some hardware advantages. It means that the cache hardware can read the tag value from the appropriate line in each way in parallel without actually performing the virtual to physical address translation, giving a fast cache response. Such a cache is often described as *Virtually Indexed, Physically Tagged* (VIPT). Cache properties of Cortex-A series processors, including the use of these tagged caches are given in Table 8-1. Other properties of the Cortex-A series processors are listed in Table 2-3 on page 2-9.

Table 8-1 Cache features of Cortex-A series processors

	Processor					
	Cortex-A5	**Cortex-A7**	**Cortex-A8**	**Cortex-A9**	**Cortex-A12**	**Cortex-A15**
L2 Cache	External	Integrated	Integrated	External	Integrated	Integrated
L2 Cache size	-	128KB to 1MB[a]	0KB to 1MB[a]	-	256KB to 8MB	512KB to 4MB[a]
Cache Implementation (Data)	PIPT	PIPT	PIPT	PIPT	PIPT	PIPT
Cache Implementation (Instruction)	VIPT	VIPT	VIPT	VIPT	VIPT	PIPT
L1 Cache size (data)[a]	4K to 64K[a]	8KB to 64KB[a]	16/32KB[a]	16KB/32KB/64KB[a]	32KB	32KB
Cache size (Inst)[a]	4K to 64K[a]	8KB to 64KB[a]	16/32KB[a]	16KB/32KB/64KB[a]	32KB or 64KB	32KB
L1 Cache Structure	2-way set associative (Inst) 4-way set associative (Data)	2-way set associative (Inst) 4-way set associative (Data)	4-way set associative	4-way set associative (Inst) 4-way set associative (Data)	4-way set associative (Inst) 4-way set associative (Data)	2-way set associative (Inst) 2-way set associative (Data)
L2 Cache Structure	-	8-way set associative	8-way set associative	-	16-way set associative	16-way associative

Table 8-1 Cache features of Cortex-A series processors (continued)

	Processor					
	Cortex-A5	**Cortex-A7**	**Cortex-A8**	**Cortex-A9**	**Cortex-A12**	**Cortex-A15**
Cache line (words)	8	8	16	8	-	16
Cache line (bytes)	32	64	64	32	64	64
Error protection	None	None	L2 ECC	None	L1 None, L2 ECC	Optional for L1 and L2

a. Configurable

However, there is a drawback to a VIPT implementation. For a 4-way set associative 32KB or 64KB cache, bits [12] and [13] of the address are required to select the index. If 4KB pages are used in the MMU, bits [13:12] of the virtual address might not be equal to bits [13:12] of the physical address. There is therefore scope for potential cache coherency problems if multiple virtual address mappings point to the same physical address. This is resolved by placing certain restrictions on such multiple mappings that kernel translation table software must obey. This is described as a *page coloring* issue and exists on other processor architectures for the same reasons.

This problem is avoided by using a *Physically Indexed, Physically Tagged* (PIPT) cache implementation. The Cortex-A series of processors use such a scheme for their data caches. It means that page coloring issues are avoided, but at the cost of hardware complexity.

8.5 Cache policies

There are a number of different choices that can be made in cache operation. Consider what causes a line from external memory to be placed into the cache (*allocation policy*) and how the controller decides which line within a set associative cache to use for the incoming data (*replacement policy*). What happens when the core performs a write that hits in the cache (*write policy*) must also be controlled.

8.5.1 Allocation policy

When the core performs 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 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, as we shall see in *Write policy* on page 8-14.

8.5.2 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.

- *Round-robin* or *cyclic* replacement means that you have a counter (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.
- *Least Recently Used* (LRU) replacement is used to replace the cache line or page that was least recently used.

Most ARM processors support both Round-robin and Pseudo random policies. The Cortex-A15 processor also supports LRU.

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.

8.5.3 Write policy

When the core executes a store instruction, a cache lookup on the address(es) 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. Because there are more writes to main memory, a write-through policy is slower than a write-back policy for some use cases, where the same area of memory is updated frequently. If large contiguous blocks of memory are being written to, and the writes can be buffered, it may be just as efficient to write-through. If the memory is not expected to be read from anytime soon (think large memory copies or memory initialization,) then it is better to not fill up the cache with such writes.
- *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 core, you must consider coherency issues. These are described in *Cache coherency* on page 18-9.

8.6 Write and Fetch buffers

A write buffer is a hardware block inside the core (but sometimes in other parts of the system as well), implemented using a number of buffers. 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 with executing the next instructions. 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. It does this by freeing the core from having 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 will be 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 that sometimes the behavior of the write buffer is not what you want 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 really want a stream of bytes to be written and you don’t want the stores to be combined.*ARM memory ordering model* on page 10-3, describes memory types supported by the ARM architecture and how to use these to control how 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.

8.7 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. One might 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 (typically for all parts of the memory system that contain code and more generally for RAM and ROM, but not peripherals). Performance will be considerably increased in Cortex-A series processors if instruction memory is cached. Placing frequently accessed data together in memory can also be helpful. For example, a frequently accessed array might 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 will be used, there will be little or no performance gain. This can be mitigated by accessing data in a manner that is ‘cache-friendly’.For instance, accesses to sequential addresses, for example, accessing a row of an array, benefit from cache behavior, Non-predictable or non-sequential access patterns, for example, linked lists, do not.

Smaller code might cache better than larger code and this can sometimes give seemingly paradoxical results. For example, a piece of C code might fit entirely within cache when compiled for Thumb (or for the smallest size) but not when compiled for ARM (or for maximum performance) and as a consequence can actually run faster than the more optimized version. Cache considerations are described in much more detail in Chapter 17 *Optimizing Code to Run on ARM Processors*.

8.8 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 MMU related activity such as changing access permissions, cache policies, or virtual to physical address mappings.

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 will be 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 bit(s) 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 cache clean operations can be performed by cache set, or way, or by virtual address.

Copying code from one location to another (or other forms of self-modifying code) might require you either to clean and/or to invalidate the cache. The memory copy code will use load and store instructions and these will operate on the data side of the core. 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. In systems where the TrustZone Security Extensions are in use, there can be hardware limitations applied to non-secure use of some of these operations.

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, or when the data is going to be overwritten, and you don't care about losing the previous values. You can perform the operation on the entire cache, or on individual lines. These individual lines can be specified either by giving a virtual 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 and we will look at this in *Level 2 cache controller* on page 8-22. A typical example of such code can be found in *Setting up caches, MMU and branch predictors* on page 13-3.

Example 8-1 Preparing the caches

```
setup_caches
      MRC p15, 0, r1, c1, c0, 0          ; Read System Control Register (SCTLR)
      BIC r1, r1, #1                     ; mmu 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.MMU, L1$ disable
;-----------------------------------------------
     MRC p15, 0, r1, c1, c0, 0          ; Read System Control Register (SCTLR)
     BIC r1, r1, #1                     ; mmu 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$, TLB, 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
     MCR     p15, 0, r0, c8, c7, 0      ; Invalidate entire Unified Main TLB
     ISB                                ; instr sync 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
;-----------------------------------------------
```

Of course, these operations will be accessed through kernel code – in GCC on Linux, you will use the `__clear_cache()` function implemented in `arch/arm/mm/cache-v7.S`.

```
void __clear_cache(char* beg, char* end);
```

The start address (char* beg) is inclusive, while the end address (char* end) is exclusive.

Equivalent functions exist in other operating systems, Google Android has `cacheflush()`, for example.

A common situation where cleaning or invalidation can be required is DMA (Direct Memory Access). When it is required to make changes made by the core 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 core, the affected addresses must be invalidated in the cache.

8.9 Point of coherency and unification

For set/way based clean and invalidate, the operation is performed on a specific level of cache. For operations that use a virtual address, the architecture defines two conceptual points:

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 will be the main external system memory.

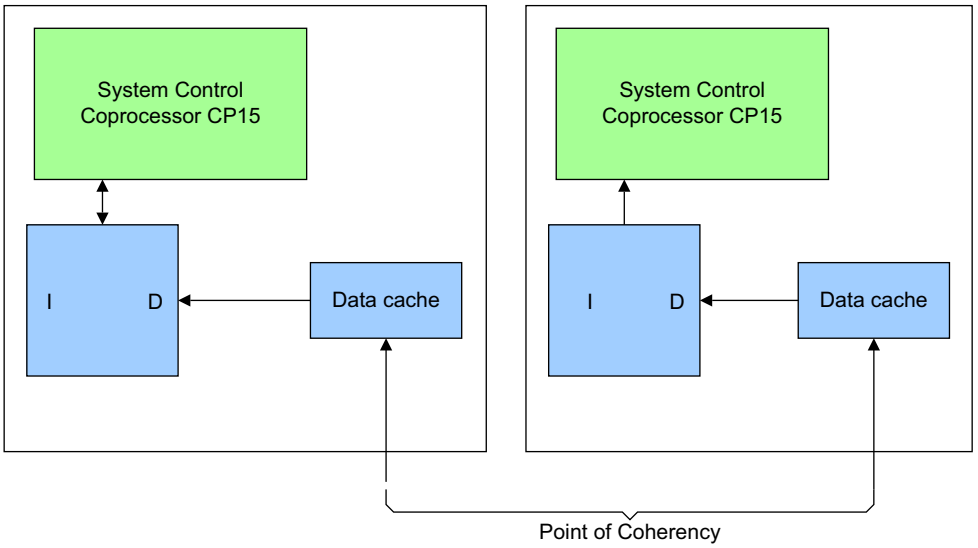


Figure 8-8 Point of Coherency

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. For example, a unified level 2 cache would be the point of unification in a system with Harvard level 1 caches and a TLB for cacheing translation table entries. If no external cache is present, main memory would be the Point of unification.

In the Cortex-A9 processor the PoC and PoU is essentially the same place, at the L2 interface.

Since the Cortex-A8 processor incorporates a L2 cache under CP15 control PoU and PoC are in different places, PoU is in the L2 cache and PoC is outside the L2 interfaces.

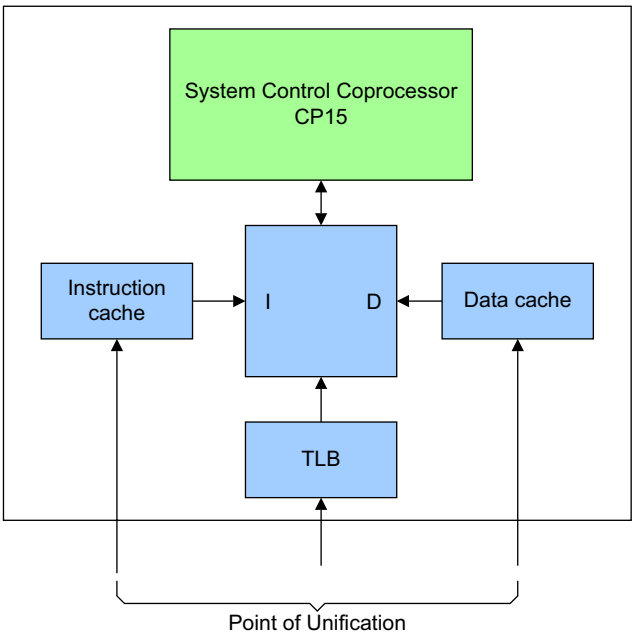


Figure 8-9 Point of Unification

Readers unfamiliar with the terms hardware translation table walk or Translation Lookaside Buffer (TLB) will find these described in Chapter 9. If no external cache is present, main memory would be the PoU.

In the case of a cluster, or a big.LITTLE combination, the PoU is where instruction and data caches and translation table walks of all the cores within the cluster are guaranteed to see the same copy of a memory location.

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:

- Clean the relevant data cache entries by address.
- Invalidate instruction cache entries by address.

In addition, the use of memory barriers will be required.

8.9.1 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 cache level
    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
```

Similarly, you can use the clean data cache entry and invalidate TLB operations to ensure that all writes to the translation tables are visible to the MMU.

8.10 Level 2 cache controller

At the start of this chapter, we briefly described the partitioning of the memory system and explained how many systems have a multi-level cache hierarchy. The Cortex-A5 and Cortex-A9 processors, however, do not have an integrated level 2 cache. Instead, the system designer can opt to connect another cache controller, such as 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 cores, or indeed between the core and other agents, such as 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.

8.10.1 Level 2 cache maintenance

Virtual and physical tags and indexes on page 8-11 described how you might require the ability either to clean or invalidate some or all of a 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 core, or through CP15, where the level 2 cache is implemented inside the core. The registers themselves are not cached, which makes this feasible. Where such operations are performed by having the core perform memory-mapped writes, the core must have a way of determining when the operation is complete. It does this by polling an additional memory-mapped register within the L2 cache controller.

The ARM L2C-310 Level 2 cache controller operates only on physical addresses. Therefore, to perform cache maintenance operations, it might be necessary for the program to perform a virtual to physical address translation. The L2C-310 provides a *cache sync* operation that forces the system to wait for pending operations to complete.

8.11 Parity and ECC in caches

So-called *soft errors* are an increasing concern. Smaller transistor geometries and lower voltages give circuits an increased sensitivity to perturbation by cosmic rays and other background radiation, alpha particles from silicon packages, or from electrical noise. 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. In some systems, mean-time-between-failure could be measured in seconds if appropriate protection against soft errors was not employed.

The ARM architecture provides support for parity and *Error Correcting Code* (ECC) in the caches. Parity means that there is an additional bit that marks whether the number of bits with the value one is even or odd, depending on the scheme chosen. This provides a simple check against single bit errors.

An ECC scheme enables detection of multiple bit failures and possible recovery from soft errors, but recovery calculations can take several cycles. Implementing a core that is tolerant of level 1 cache RAM accesses taking multiple clock cycles significantly complicates the design. ECC is therefore more commonly used only on blocks of memory (for example, the Level 2 cache), outside the core. The Cortex-A15, however, supports ECC and parity inside the core.

Parity is checked on reads and writes, and can be implemented on both tag and data RAMs. Parity mismatch generates a prefetch or data abort exception, and the fault status address registers are updated appropriately.

Chapter 9
The Memory Management Unit

An important function of a *Memory Management Unit* (MMU) is to enable you to manage tasks as independent programs running in their own private virtual memory space. A key feature of such a virtual memory system is address relocation, or the translation of the virtual address issued by the processor to a physical address in main memory.

The ARM MMU is responsible for translating addresses of code and data from the virtual view of memory to the physical addresses in the real system. The translation is carried out by the MMU hardware and is transparent to the application. In addition, the MMU controls such things as memory access permissions, memory ordering and cache policies for each region of memory.

In multi-tasking embedded systems, we typically require a way to partition the memory map and assign permissions and memory attributes to these regions of memory. In situations where we are running more complex operating systems, like Linux, we require even greater control over the memory system.

The MMU enables tasks or applications to be written in a way that requires them to have no knowledge of the physical memory map of the system, or about other programs that might be running simultaneously. This enables you to use the same virtual memory address space for each program. It also lets you work with a contiguous virtual memory map, even if the physical memory is fragmented. This virtual address space is separate from the actual physical map of memory in the system. Applications are written, compiled and linked to run in the virtual memory space. Virtual addresses are those used by you, and the compiler and linker, when placing code in memory. Physical addresses are those used by the actual hardware system.

It is the responsibility of the operating system to program the MMU to translate between these two views of memory. Figure 9-1 on page 9-2 shows an example system, illustrating the virtual and physical views of memory. Different processors and/or devices in a single system might have different virtual and physical address maps, for example, some multi-core boards and PCI devices.

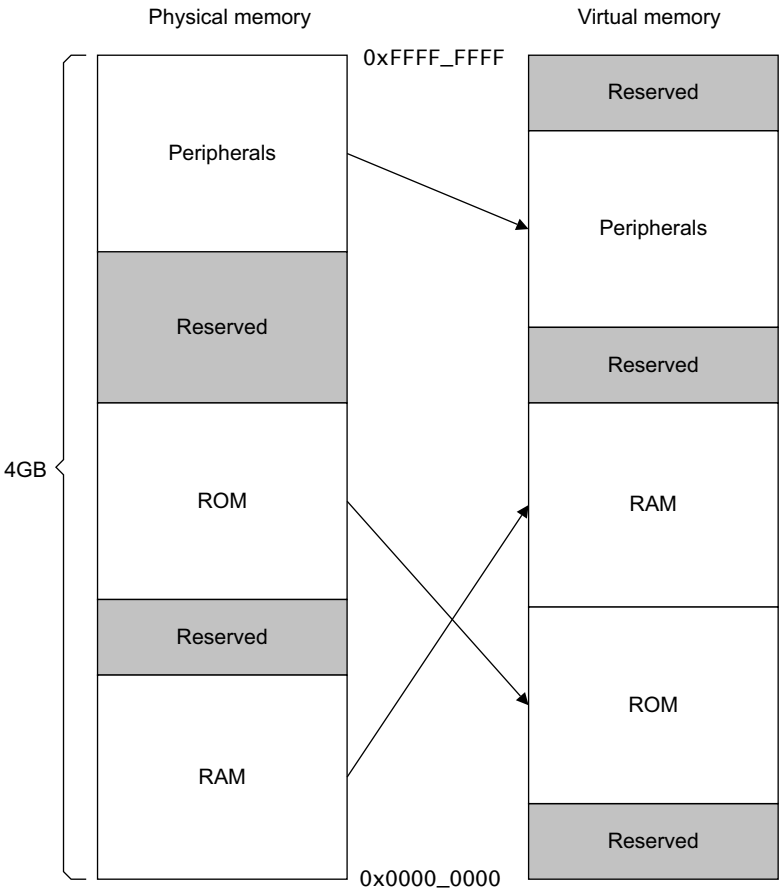


Figure 9-1 Virtual and physical memory

When the MMU is disabled, all virtual addresses map directly to the corresponding physical address (a *flat mapping*). If the MMU cannot translate an address, it generates an abort exception on the processor and provides information to the processor about what the problem was. This feature can be used to map memory or devices on-demand, one page at a time.

9.1 Virtual memory

The MMU enables you to build systems with multiple virtual address maps. Each task can have its own virtual memory map. The OS kernel places code and data for each application in physical memory, but the application itself does not require the location.

The address translation carried out by the MMU is done using *translation tables*. These are tree-shaped table data structures created by software in memory, that the MMU hardware traverses to accomplish virtual address translation.

Note

In the ARM architecture, the concept referred to in generic computer terminology as *page tables* has a more specific meaning. The ARM architecture uses multi-level page tables, and defines *translation tables* as a generic term for all of these. An entry in a translation table contains all the information required to translate a page in virtual memory to a page in physical memory. The specific mechanism of traversal and the table format are configurable by software and are explained later.

Translation table entries are organized by virtual address. In addition to describing the translation of that virtual page to a physical page, they also provide access permissions and memory attributes necessary for that page.

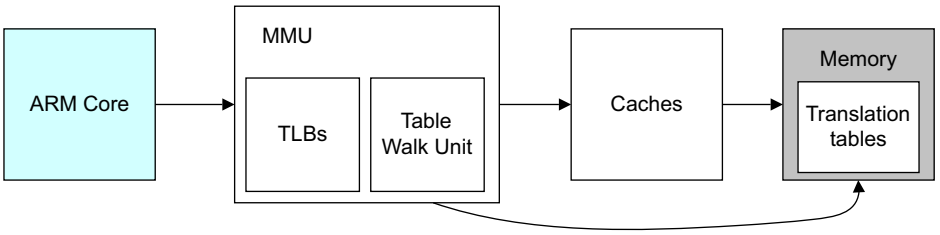


Figure 9-2 The Memory Management Unit

Addresses generated by the core are virtual addresses. When the MMU is enabled all memory accesses made by the core pass through it. The MMU essentially replaces the most significant bits of this virtual address with some other value, to generate the physical address (effectively defining a base address of a piece of memory).The same translation tables are used to define the translations and memory attributes that apply to both instruction fetches and to data accesses. Dedicated hardware within the MMU enables it to read the translation tables in memory. This process is known as *translation table walking*.

9.1.1 Configuring and enabling the MMU

Before the MMU is enabled, the translation tables must be written to memory. The TTBR register must be set to point to the tables. The following code sequence can then be used to enable the MMU:

```
MRC p15, 0, R1, c1, C0, 0       ;Read control register
ORR R1, #0x1                    ;Set M bit
MCR p15, 0,R1,C1, C0,0          ;Write control register and enable MMU
```

Care must be taken if enabling the MMU changes the address mapping of the region in which code is currently being executed. Barriers (See *Memory barriers* on page 10-6) may be necessary to ensure correct operation.

9.2 The Translation Lookaside Buffer

The *Translation Lookaside Buffer* (TLB) is a cache of recently executed page translations within the MMU. On a memory access, the MMU first checks whether the translation is cached in the TLB. If the requested translation is available, you have a TLB hit, and the TLB provides the translation of the physical address immediately. If the TLB does not have a valid translation for that address, you have a TLB miss and an external translation table walk is required. This newly loaded translation can then be cached in the TLB for possible reuse.

The exact structure of the TLB differs between implementations of the ARM processors. What follows is a description of a typical system, but individual implementations might vary from this. There are one or more micro-TLBs that are situated close to the instruction and data caches. Addresses with entries that hit in the micro-TLB require no additional memory look-up and no cycle penalty. However, the micro-TLB has only a small number of mappings, typically eight on the instruction side and eight on the data side. This is backed by a larger main TLB (typically 64 entries), but there might be some penalty associated with accesses that miss in the micro-TLB but that hit in the main TLB. Figure 9-3 shows how each TLB entry contains physical and virtual addresses, but also attributes (such as memory type, cache policies and access permissions) and potentially an ASID value, described in *Address Space ID* on page 9-17.

The TLB is like other caches and so has a TLB line replacement policy, but this is effectively transparent to users. If the translation table entry is a valid one, the virtual address, physical address and other attributes for the whole page or section are stored as a TLB entry. If the translation table entry is not valid, the TLB will not be updated. The ARM architecture requires that only valid translation table descriptors are cached within the TLB.

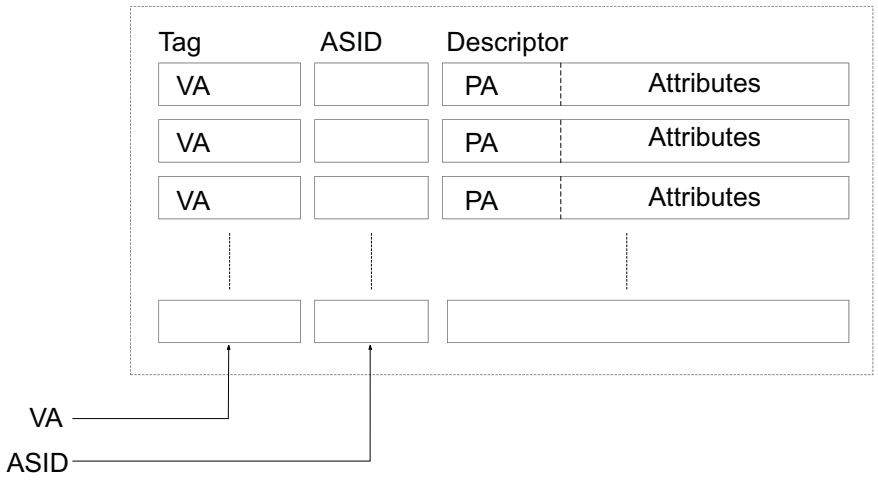


Figure 9-3 Illustration of TLB structure

9.2.1 TLB coherency

When the operating system changes translation table entries, it is possible that the TLB could contain stale translation information. The OS must take steps to invalidate TLB entries. There are several CP15 operations available that permit a global invalidate of the TLB or removal of specific entries.

As speculative instruction fetches and data reads might cause translation table walks, it is essential to invalidate the TLB when a valid translation table entry is changed. Invalid translation table entries cannot be cached in the TLB so they can be changed without invalidation.

The Linux kernel has a number of functions that use these CP15 operations, including `flush_tlb_all()` and `flush_tlb_range()`. Such functions are not typically required by device drivers.

9.3 Choice of page sizes

Page size is essentially controlled by the operating system, but it is worth being aware of the considerations involved when selecting a size. Smaller page sizes enable finer control of a block of memory and potentially can reduce the amount of unused memory in a page. If a task requires 7KB of data space, there is less unused space if it is allocated two 4KB pages as opposed to a 64KB page or a 1MB section. Smaller page sizes also enable finer control over permissions, cache properties and so forth.

However, with increased page sizes, each entry in the TLB holds a reference to a larger piece of memory. It is therefore more likely that a TLB hit will occur on any access and so there will be fewer translation table walks to slow external memory. For this reason, 16MB supersections can be used with large pieces of memory that do not require detailed mapping. In addition, each L2 translation table requires 1KB of memory.

9.4 First level address translation

Consider the process by which a virtual address is translated to a physical address using level 1 translation table entries on an ARM core. The first step is to locate the translation table entry associated with the virtual address.

The first stage of translation uses a single level 1 translation table, sometimes called a master translation table. The L1 translation table divides the full 4GB address space of a 32-bit core into 4096 equally sized sections, each of which describes 1MB of virtual memory space. The L1 translation table therefore contains 4096 32-bit (word-sized) entries.

Each entry can either hold a pointer to the base address of a level 2 translation table or a translation table entry for translating a 1MB section. If the translation table entry is translating a 1MB section determined by the encoding, (See Figure 9-5 on page 9-8), it gives the base address of the 1MB page in physical memory.

The lower bits are the same in both addresses (defining an offset in physical memory from the base address). The ARM MMU supports a multi-level translation table architecture with two levels of translation tables, level 1 (L1) and level 2 (L2). Unless the Large Physical Address Extensions (See *Large Physical Address Extensions* on page 22-10) are implemented, both L1 and L2 translation tables use the *Short-descriptor* translation table format that feature:

- 32-bit page descriptors.
- Up to two levels of translation tables.
- Support for 32-bit physical addresses
- Support for the following memory sizes:
  - 16 MB or 1 MB sections.
  - 64KB or 4KB page sizes.

The base address of the L1 translation table is known as the *Translation Table Base Address* and is held in CP15 c2. It must be aligned to a 16KB boundary. The Translation table locations are defined by the *Translation Table Base Register*s (TTRB0 and TTRB1).

When the MMU performs a translation, the top 12 bits of the requested virtual address act as the index into the translation table.

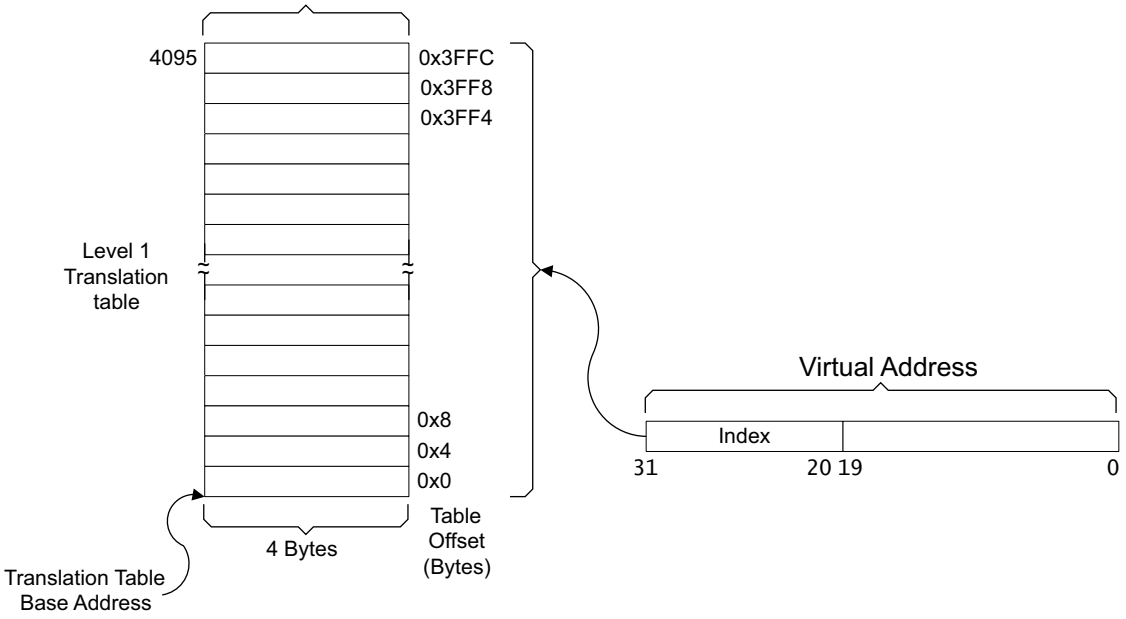


Figure 9-4 Finding the address of the level 1 translation table entry

To take a simple example, shown in Figure 9-4, suppose the L1 translation table is stored at address `0x12300000`. The processor issues virtual address `0x00100000`. The top 12 bits [31:20] define which 1MB of virtual address space is being accessed. In this case `0x001`, so the MMU must read table entry 1. To get the offset into the table you must multiply the entry number by entry size:

`0x001` * 4 bytes = address offset of `0x004`

The address of the entry the MMU reads the physical address from is `0x12300000` + `0x004` = `0x12300004`.

Now that you have the location of the translation table entry, you can use it to determine the physical memory address

Figure 9-5 shows the format of L1 translation table entries in CP15 c2.

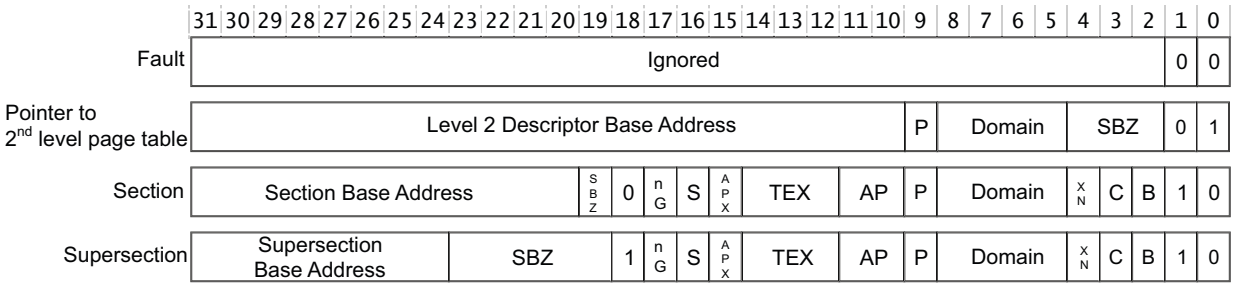


Figure 9-5 Level 1 translation table entry format

First level translation tables contain first level descriptors.

L1 translation table entries can be one of four possible types:

- A 1MB section translation entry, mapping a 1MB region to a physical address.

- An entry that points to an L2 translation table. This enables a 1MB piece of memory to be sub-divided into pages.
- A 16MB supersection. This is a special kind of 1MB section entry, that requires 16 entries in the translation table, but can reduce the number of entries allocated in the Translation Lookaside Buffer for this region.
- A fault entry that generates an abort exception. This can be either a prefetch or data abort, depending on the type of access. This effectively indicates virtual addresses that are unmapped.

The least significant two bits [1:0] in the entry define whether the entry is a fault entry, a translation table entry, or a section entry. Bit [18] is used to distinguish between a normal section and supersection.

A supersection is a 16MB piece of memory, that must have both its virtual and physical base address aligned to a 16MB boundary. Because L1 translation table entries each describe 1MB, you require 16 consecutive, identical entries within the table to mark a supersection. *Choice of page sizes* on page 9-6 described why supersections can be useful.

Figure 9-6 shows the simplest case in which the physical address of a 1MB section is directly generated from the contents of a single entry in the level 1 translation table.

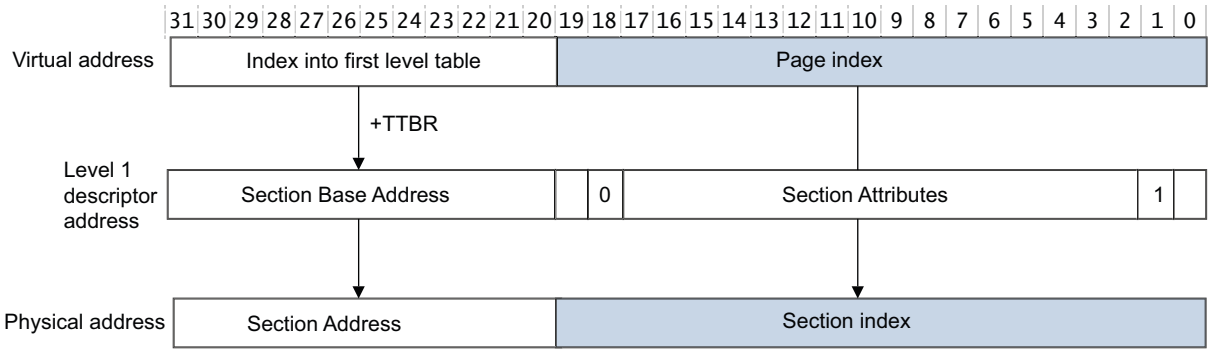


Figure 9-6 First level address translation

The translation table entry for a section (or supersection) contains the physical base address used to translate the virtual address. Many other bits are given in the translation table entry, including the Access Permissions (AP) and Cacheable (C) or Bufferable (B) types that we will consider in *Memory attributes* on page 9-14. This is all of the information required to access the corresponding physical address and in these cases, there is no requirement for the MMU to look beyond the L1 table.

Figure 9-7 on page 9-10 summarizes the translation process for an address translated by a section entry in the L1 translation table.

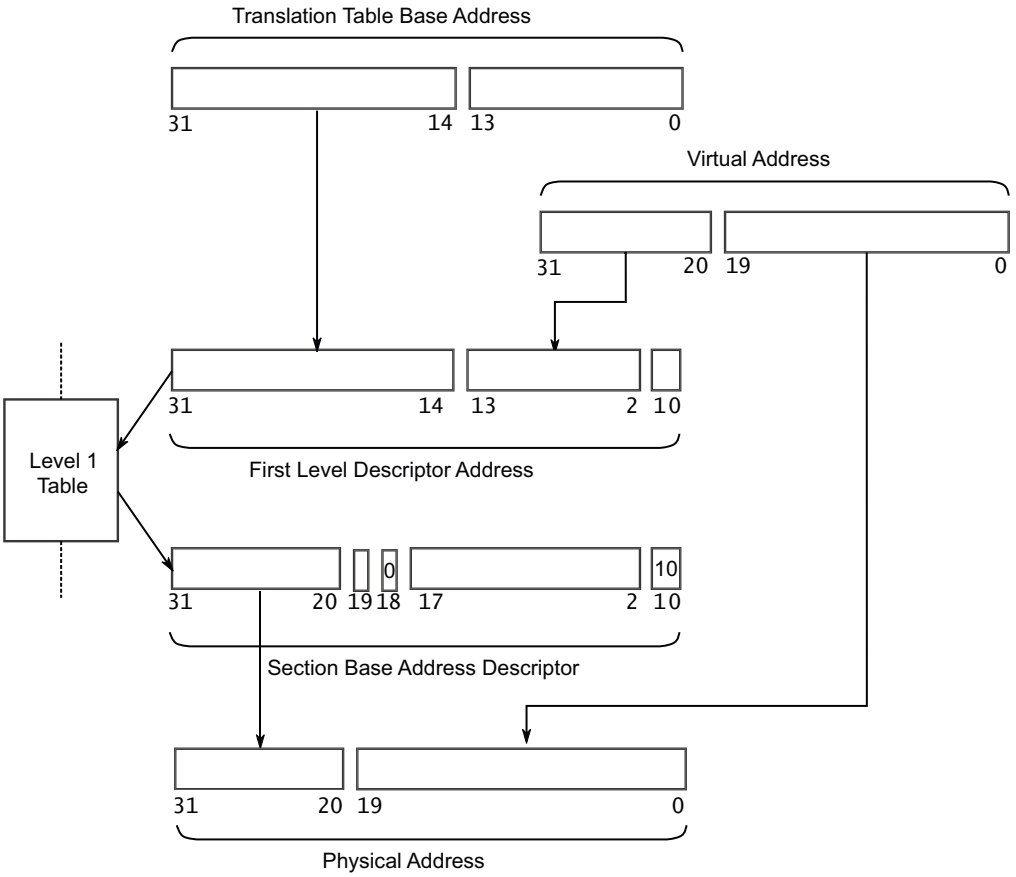


Figure 9-7 Generating a physical address from a level 1 translation table entry

In a translation table entry for a 1MB section of memory, the upper 12 bits of the translation table entry replace the upper 12 bits of the virtual address when generating the physical address, as Figure 9-5 on page 9-8 shows.

9.5 Level 2 translation tables

An L2 translation table has 256 word-sized (4 byte) entries, requires 1KB of memory space and must be aligned to a 1KB boundary. Each entry translates a 4KB block of virtual memory to a 4KB block in physical memory. A translation table entry can give the base address of either a 4KB or 64KB page.

There are three types of entry used in L2 translation tables, identified by the value in the two least significant bits of the entry:

- A large page entry points to a 64KB page. You should note that, since each entry points to a 4KB address space, large page entries must be repeated 16 times.
- A small page entry points a 4KB page.
- A fault page entry generates an abort exception if accessed.

Figure 9-8 shows the format of L2 translation table entries.

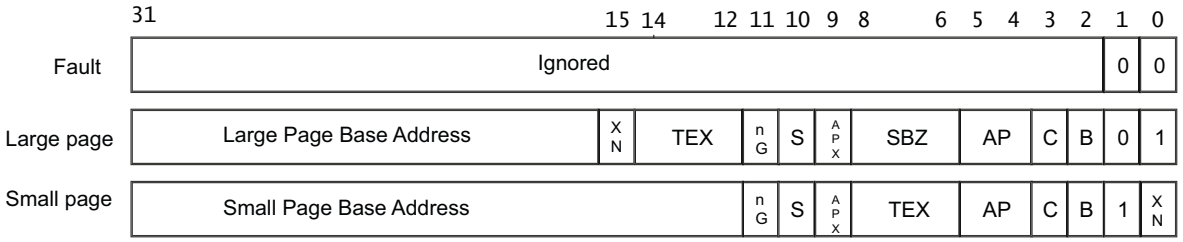


Figure 9-8 Format of a level 2 translation table entry

As with the L1 translation table entry, a physical address is given, along with other information about the page. Type extension (TEX), Shareable (S), and Access Permission (AP, APX) bits are used to specify the attributes necessary for the ARMv7 memory model. Along with TEX, the C and B bits control the cache policies for the memory governed by the translation table entry. The nG bit defines the page as being global (applies to all processes) or non-global (used by a specific process). These are described in more detail in *Memory attributes* on page 9-14.

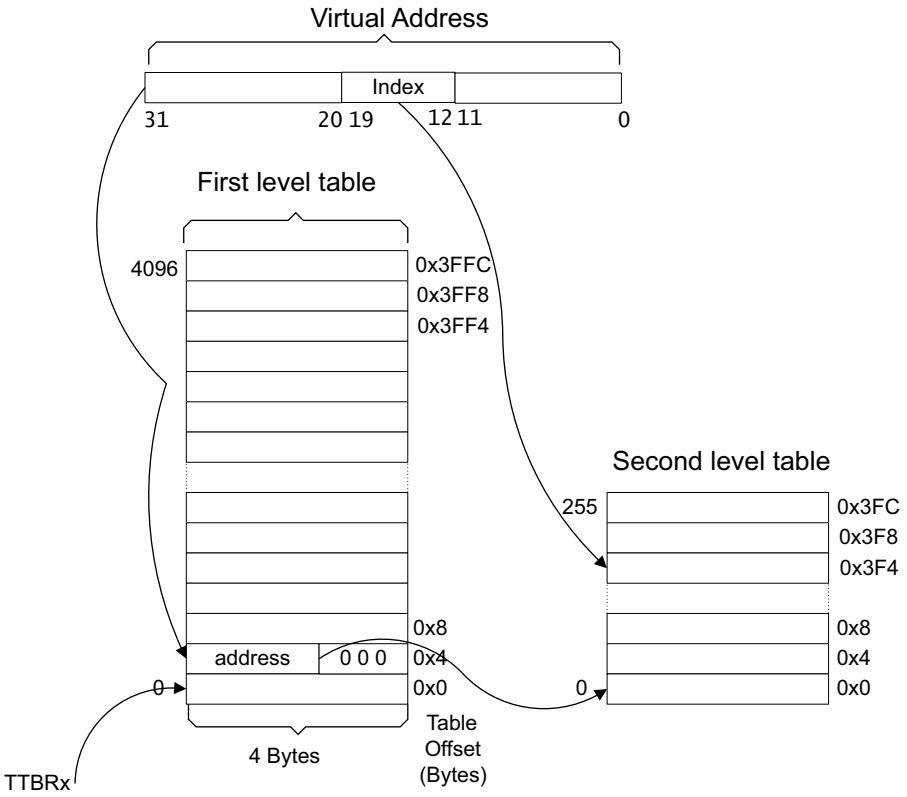


Figure 9-9 Generating the address of the level 2 translation table entry

In Figure 9-9 we see how the address of the L2 translation table entry that we require is calculated by taking the (1KB aligned) base address of the level 2 translation table (given by the level 1 translation table entry) and using 8 bits of the virtual address (bits [19:12]) to index within the 256 entries in the L2 translation table.

Figure 9-10 on page 9-13 summarizes the address translation process when using two layers of translation tables. Bits [31:20] of the virtual address are used to index into the 4096-entry L1 translation table, whose base address is given by the CP15 TTB register. The L1 translation table entry points to an L2 translation table that contains 256 entries. Bits [19:12] of the virtual address are used to select one of those entries that then gives the base address of the page. The final physical address is generated by combining that base address with the remaining bits of the virtual address.

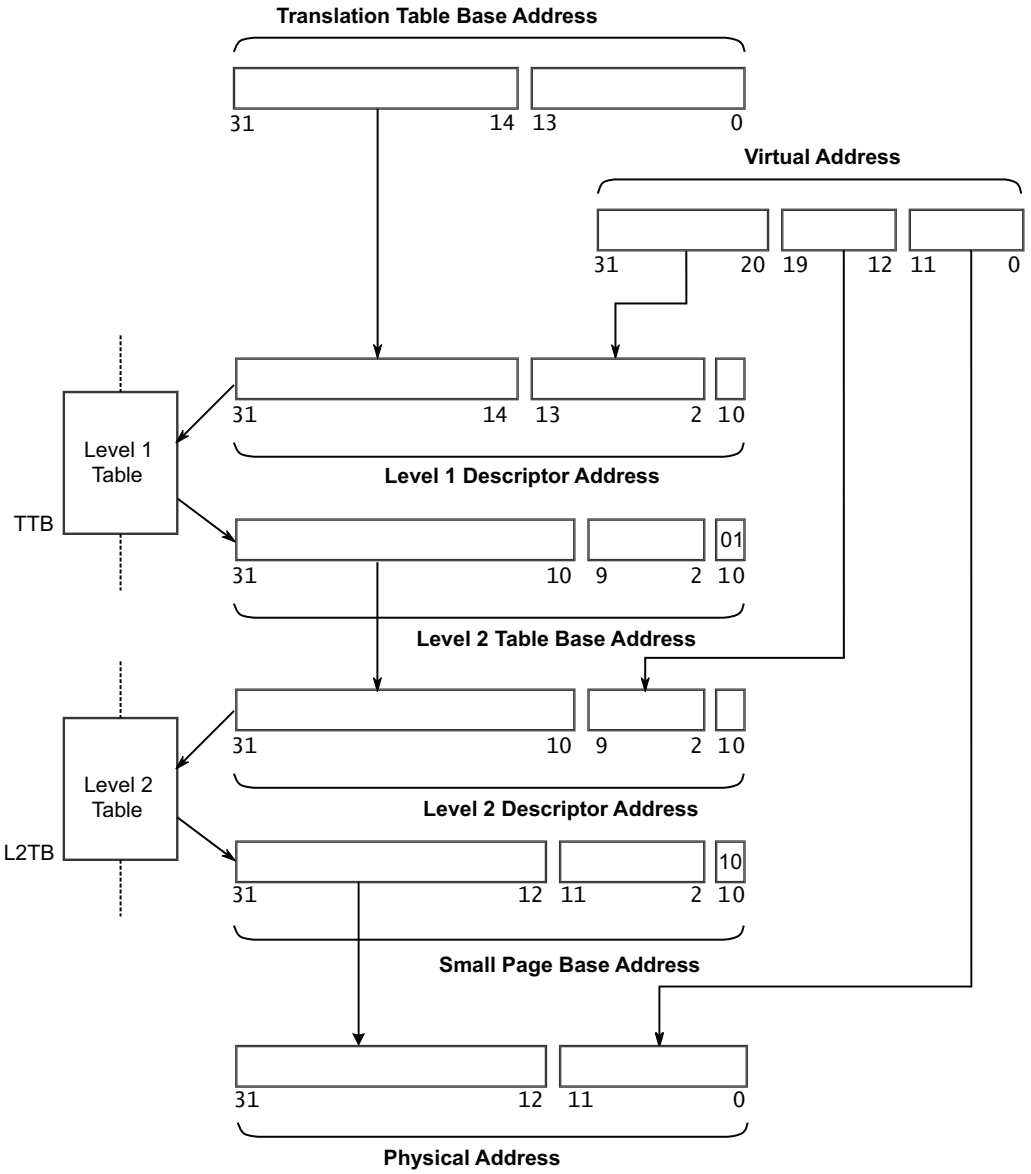


Figure 9-10 Summary of generation of physical address using the L2 translation table entry

9.6 Memory attributes

We have seen how translation table entries enable the MMU hardware to translate virtual to physical addresses. However, they also specify a number of attributes associated with each page, including access permissions, memory type and cache policies.

9.6.1 Memory Access Permissions

The Access Permission (AP and APX) bits in the translation table entry give the access permissions for a page. See Table 9-1.

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

Information about the address of the faulting location and the reason for the fault is stored in CP15 (the fault address and fault status registers). The abort handler can then take appropriate action – for example, modifying translation tables to remedy the problem and then returning to the application to retry the access. Alternatively, the application that generated the abort might have a problem and must be terminated.

Table 9-1 Summary of Access Permission encodings

APX	AP	Privileged	Unprivileged	Description
0	00	No access	No access	Permission fault
0	01	Read/Write	No access	Privileged Access only
0	10	Read/Write	Read	No user-mode write
0	11	Read/Write	Read/Write	Full access
1	00	-	-	Reserved
1	01	Read	No access	Privileged Read only
1	10	Read	Read	Read only
1	11	-	-	Reserved

9.6.2 Memory types

Earlier ARM architecture versions enabled you to specify the memory access behavior of pages 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. The new memory types added to the ARM architecture in ARMv6 and extended in the ARMv7 architecture are designed to meet these requirements.

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 9-2.

Table 9-2 Memory attributes

Memory type	Shareable/ Non-shareable	Cacheable	Description
Normal	Shareable	Yes	Designed to handle normal memory that is shared between multiple cores.
	Non-shareable	Yes	Designed to handle normal memory that is used only by a single core.
Device	-	No	Designed to handle memory-mapped peripherals.[a] All memory accesses to Device memory occur in program order.
Strongly-ordered	-	No	All memory accesses to Strongly-ordered memory occur in program order. All Strongly-ordered accesses are assumed to be shared.

a. Shared memory was originally used to distinguish between accesses directed to the “peripheral private port” found on several ARM11 processors. This use is now deprecated and processors implementing LPAE treat all device accesses as Shareable.

More descriptions of these memory types can be found in *ARM memory ordering model* on page 10-3.

Table 9-3 shows how the TEX, C and B bits within the translation table entry are used to set the memory types of a page and also the cache policies to be used. The meaning of each of the memory types is described in Chapter 10, while the cache policies were described in Chapter 8.

Table 9-3 Memory type and cacheable properties encoding in translation table entry

TEX	C	B	Description	Memory type
000	0	0	Strongly-ordered	Strongly-ordered
000	0	1	Shareable device	Device[a]
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[a]
010	-	-	Reserved	-
011	-	-	Reserved	-
1XX	Y	Y	Cached memory XX = Outer policy YY = Inner policy	Normal

a. LPAE treats all device accesses as Shareable

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 (perhaps for level 2 or level 3 caches) 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-A15 and Cortex-A8 processors, inner cache properties set by the translation table entry apply to both L1 and L2 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, of course.

9.6.3 Execute Never

When set, the *Execute Never* (XN) bit in the translation table entry prevents speculative instruction fetches taking place from desired 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 which might be caused by speculative instruction fetches.

9.6.4 Domains

The ARM architecture has an unusual feature that enables regions of memory to be tagged with a domain ID. There are 16 domain IDs provided by the hardware and CP15 c3 contains the *Domain Access Control Register* (DACR) that holds a set of 2-bit permissions for each domain number. This enables each domain to be marked as no-access, *manager* mode or *client* mode. No-access causes an abort on any access to a page in this domain, irrespective of page permissions. Manager mode ignores all page permissions and enables full access. Client mode uses the permissions of the pages tagged with the domain.

Note

The use of domains is deprecated in the ARMv7 architecture, and will eventually be removed, but in order for access permissions to be enforced, it is still necessary to assign a domain number to a section and to ensure that the permission bits for that domain are set to client. Typically, you would set all domain ID fields to 0 and set all fields in the DACR to ‘Client’.

9.7 Multi-tasking and OS usage of translation tables

In most systems using Cortex-A series processors, you will have a number of applications or tasks running concurrently. Each task can have its own unique translation tables residing in physical memory. Typically, much of the memory system is organized so that the virtual-to-physical address mapping is fixed, with translation table entries that never change. This typically is used to contain operating system code and data, and also the translation tables used by individual tasks.

Whenever an application is started, the operating system will allocate it a set of translation table entries that map both the code and data used by the application to physical memory. If the application has to map in code or extra data space (for example through a `malloc()` call), the kernel can subsequently modify these tables. When a task completes and the application is no longer running, the kernel can remove any associated translation table entries and re-use the space for a new application. In this way, multiple tasks can be resident in physical memory. On a task switch, the kernel switches translation table entries to the page in the next thread to be run. In addition, the dormant tasks are completely protected from the running task. This means that the MMU can prevent the running task from accessing the code or data of other tasks.

9.7.1 Address Space ID

When we described the translation table bits in *Level 2 translation tables* on page 9-11 we noted a bit called nG (non-global). If the nG bit is set for a particular page, the page is associated with a specific application. When the MMU performs a translation, it uses both the virtual address and an ASID value.

The ASID is a number assigned by the OS to each individual task. This value is in the range 0-255 and the value for the current task is written in the ASID register (accessed using CP15 c13). When the TLB is updated and the entry is marked as non-global, the ASID value will be stored in the TLB entry in addition to the normal translation information. Subsequent TLB look-ups will only match on that entry if the current ASID matches with the ASID that is stored in the entry. You can therefore have multiple valid TLB entries for a particular page (marked as non-global), but with different ASID values. This significantly reduces the software overhead of *context switches*, as it avoids the requirement to flush the on-chip TLBs. The ASID forms part of a larger (32-bit) process ID register that can be used in task-aware debugging.

Note

A *context switch* denotes the scheduler transferring execution from one process to another. This typically requires saving the current process state and restoring the state of the next process waiting to be run.

Figure 9-11 on page 9-18 illustrates this. Here, you have multiple applications (A, B and C), each of which is linked to run from virtual address 0. Each application is located in a separate address space in physical memory. There is an ASID value associated with each application so you can have multiple entries within the TLB at any particular time, that will be valid for virtual address 0.

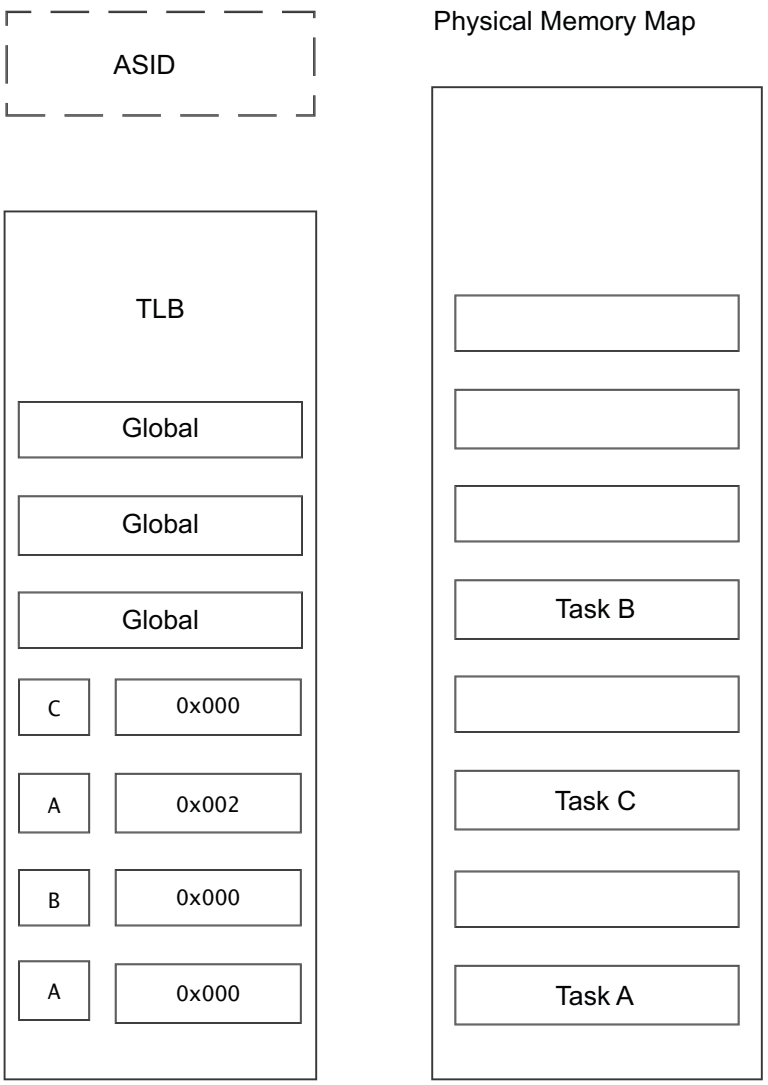


Figure 9-11 ASIDs in TLB mapping the same virtual address

9.7.2 Translation Table Base Register 0 and 1

An additional potential difficulty associated with managing multiple applications with their individual translation tables is that there could be multiple copies of the L1 translation table, one for each application. Each of these will be 16KB in size. Most of the entries will be identical in each of the tables, as typically only one region of memory will be task-specific, with the kernel space being unchanged in each case. Furthermore, if a global translation table entry is to be modified, the change will be required in each of the tables.

To help reduce the effect of these problems, a second translation table base register is provided. CP15 contains two *Translation Table Base Registers*, TTBR0 and TTBR1. A control register (the TTB Control Register) is used to program a value in the range 0 to 7. This value (denoted by N) tells the MMU how many of the upper bits of the virtual address it must check to determine which of the two TTB registers to use.

When N is 0 (the default), all virtual addresses are mapped using TTBR0. With N in the range 1-7, the hardware looks at the most significant bits of the virtual address. If the N most significant bits are all zero, TTBR0 is used, otherwise TTBR1 is used.

For example, if N is set to 7, any address in the bottom 32MB of memory will use TTBR0 and the rest of memory will use TTBR1. As a result, the application-specific translation table pointed to by TTBR0 will contain only 32 entries (128 bytes). The global mappings are in the table pointed to by TTBR1 and only one table must be maintained.

When these features are used, a context switch will typically require the operating system to change the TTBR0 and ASID values, using CP15 instructions. However, as these are two separate, non-atomic operations, some care is required to avoid problems associated with speculative accesses occurring using the new value of one register together with the older value of the other. OS programmers making use of these features should become familiar with the sequences recommended for this purpose in the *ARM Architecture Reference Manual*.

9.7.3 The Fast Context Switch Extension

The *Fast Context Switch Extension* (FCSE) was added to the ARMv4 architecture but has been deprecated since ARMv6. It enabled multiple independent tasks to run in a fixed, overlapping area at the bottom of the virtual memory space without having to clean the cache or TLB on a context switch. It did this by modifying virtual addresses by substituting a process ID value into the top seven bits of the virtual address (but only if that address lay within the bottom 32MB of memory). Some ARM documentation distinguishes *Modified Virtual Addresses* (MVA) from *Virtual Addresses* (VA). This distinction is useful only when the FCSE is used.

Chapter 10
Memory Ordering

Older implementations of the ARM architecture execute all instructions in program order and each instruction is completely executed before the next instruction is 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 to re-order memory accesses. 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.

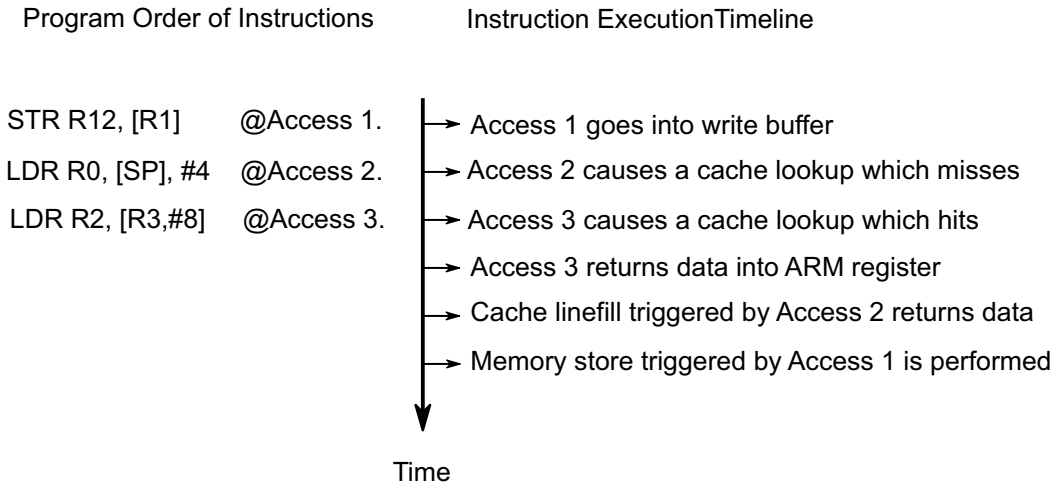


Figure 10-1 Memory ordering example

In Figure 10-1, three instructions are listed in program order. The first instruction performs a write to external memory that in this example, goes to the write buffer (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 in order 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 ARMv7-A 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 (such as `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 required, this is communicated to the core through the memory type attribute of the translation table entry that describes that memory. Enforcing ordering rules on the core limits the possible hardware optimizations and therefore reduces performance and increases power consumption.

10.1 ARM memory ordering model

Cortex-A 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 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 10-1 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 10-1 Memory type access order

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

10.1.1 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 will be preserved. Accesses will be atomic, and will not be interrupted part way through.
- Both read and write accesses can have side-effects on the system. 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 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 core 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 will be 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 or translation table walks with respect to such explicit memory accesses.

10.1.2 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.

The properties of Normal memory are as follows:

- The core can repeat read and some write accesses.
- The core can pre-fetch or speculatively access additional memory locations, with no side-effects (if permitted by MMU access permission settings). The core will not perform speculative writes, however.
- Unaligned accesses can be performed.
- Multiple accesses can be merged by core 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.

Regions of Normal memory must also have cacheability attributes described. See Chapter 8 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 core level 1 cache. An implementation might not have any outer cache, or it can apply the outer cacheability attribute to an L2 or L3 cache. For example, in a system containing a Cortex-A9 processor and the L2C-310 level2 cache controller, the L2C-310 is considered to be the outer cache. The Cortex-A8 L2 cache can be configured to use either inner or outer cache policy.

Shareability

Normal memory must also be designated 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.

The Outer Shareable attribute enables the definition of systems containing multiple levels of coherency control. For example, an Inner Shareable domain could consist of an Cortex-A15 cluster and Cortex-A7 cluster. Within a cluster, the data caches of the cores are coherent for all data accesses that have the Inner Shareable attribute. The Outer Shareable domain, meanwhile, might consist of this cluster and a graphics processor with multiple cores. An Outer Shareable domain can consist of multiple Inner Shareable domains, but and Inner Shareable domain can only be part of one Outer Sharable domain.

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 do not have to take care of the effects of data or caches. Without the Shareable attribute, in situations where cache coherency is not maintained between cores 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).

10.2 Memory barriers

A memory barrier is an instruction that requires the core to apply an ordering constraint between memory operations that occur before and after the memory barrier instruction in the program. Such instructions can also be called *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 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, core 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 must 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 system. This description is a simplified version of that given in the *ARM Architecture Reference Manual*, this section is intended to introduce the use 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 core 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 core, or of instruction fetches.

Instruction Synchronization Barrier (ISB)

This flushes the pipeline and prefetch buffer(s) in the core, 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 ASID changes or TLB 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 applies to, as follows:

SY This is the default and means that the barrier applies to the full system, including all cores 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* on page 8-19).

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 core 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 only 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 quad core Cortex-A9 cluster. The cluster forms 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. If the DMB ISH variant is used, the same is not guaranteed for external observers such as DMA controllers or DSPs.

10.2.1 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 shown in Example 10-1:

Example 10-1 Code example showing memory ordering issues

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 you were 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. We might have code similar to that in Example 10-2:

Example 10-2 Possible ordering hazard with postbox

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.

10.2.2 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.

Example 10-3 shows the type of code that can cause a problem.

Example 10-3 Deadlock

```
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
```

Without multiprocessing extensions the ARMv7 architecture does not strictly require the 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 for the core forces its store to be observed before it will read from Flag.

Cores that implement the multiprocessing extensions are required to complete accesses in a finite time (that is, their write buffers must drain) and so the barrier instruction is not required.

10.2.3 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 core that is attempting to acquire a mutex might find that some other core already has the lock. Instead of having the core 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. We 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.

10.2.4 Linux use of barriers

Barriers are required to enforce ordering of memory operations. Normally you will not have to understand, or explicitly use memory barriers. This is because they are already included within kernel locking and scheduling primitives. Nevertheless, writers of device drivers or those seeking an understanding of kernel operation might find a detailed description useful.

Both the compiler and core micro-architecture optimizations permit the order of instructions and associated memory operations to be changed. Sometimes, however, you want to enforce a specified order of execution of memory operations. For example, you can write to a memory mapped peripheral register. This write can have side effects elsewhere in the system. Memory operations that are in before or after this operation in our program can appear as if they can be re-ordered, as they operate on different locations. In some cases, however, you want to ensure that all operations complete before this peripheral write completes. Or, you might want to make sure that the peripheral write completes before any additional memory operations are started. Linux provides some functions to do this, as follows:

- Instruct the compiler that re-ordering is not permitted for a particular memory operation. This is done with the `barrier()` function call. This controls only the compiler code generation and optimization and has no effect on hardware re-ordering.
- Call a memory barrier function that maps to ARM processor instructions that perform the memory barrier operations. These enforce a particular hardware ordering. The available barriers are as follows (in a Linux kernel compiled with Cortex-A SMP support):
  - The read memory barrier `rmb()` function ensures that any read that appears before the barrier is completed before the execution of any read that appears after the barrier.
  - The write memory barrier `wmb()` function ensures that any write that appears before the barrier is completed before the execution of any write that appears after the barrier.
  - The memory barrier `mb()` function ensures that any memory access that appears before the barrier is completed before the execution of any memory access that appears after the barrier.
- There are corresponding SMP versions of these barriers, called `smp_mb()`, `smp_rmb()` and `smp_wmb()`. These are used to enforce ordering on Normal cacheable memory, between cores inside the same cluster, for example, each core in a Cortex-A15 cluster. They can be used with devices and they work even for normal non-cacheable memory. When the kernel is compiled without `CONFIG_SMP`, each invocation of these are expanded to `barrier()` statements.

  All of the locking primitives provided by Linux include any required barrier.

For these memory barriers, it is almost always the case that a pair of barriers is required. For more information, see `http://www.kernel.org/doc/Documentation/memory-barriers.txt`.

10.3 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.

Therefore dirty data that is 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 cache(s) before starting the DMA. Because all ARMv7-A processors can do speculative memory accesses, it will also be necessary to invalidate after using the DMA.

10.3.1 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.

10.3.2 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.

Chapter 11
Exception Handling

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 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* on page 11-3.

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 as such – all of 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 very 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.

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 as MMU generated aborts or OS calls using the SVC instruction. You will recall from Chapter 3 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 Chapter 3, before continuing with this chapter.

11.1 Types of exception

Table 3-1 on page 3-1, describes how the ARMv7-A and ARMv7-R architectures support a number of processor modes, six privileged modes called *FIQ*, *IRQ*, *Supervisor*, *Abort*, *Undefined* and *System*, and the non-privileged *User* mode. The *Hyp* mode and *Monitor* modes can be added to the list if the Virtualization Extensions and Security Extensions are implemented. 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.

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 There are two types of interrupts provided on ARMv7-A cores, 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 core – 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.

Both FIQ and IRQ are physical signals to the core, and when asserted, the core will take the corresponding exception if it is currently enabled. On almost all systems, various interrupts sources will be connected using an interrupt controller. The interrupt controller arbitrates and prioritizes interrupts, and in turn provides a serialized single signal that is then connected to the FIQ or IRQ signal of the core. For more information see *The Generic Interrupt Controller* on page 12-7.

Because the occurrence of IRQ and FIQ interrupts are not directly related to the software being executed by the core at any given time, they are classified as *asynchronous* exceptions.

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 (indicating perhaps that the specified address does not correspond to real memory in the system). Alternatively, the abort can be generated by the Memory Management Unit (MMU) of the core. An operating system can use MMU aborts to dynamically allocate memory to applications.

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 when a load or store instruction executes 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 MMU are always synchronous. The architecture does not require particular classes of externally aborted accesses to be synchronous.

For example, on a particular implementation, an external abort reported on a translation table walk might be treated as precise, but this is not required for all cores. 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 will be 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

Execution of certain instructions can generate exceptions. Such instructions are typically executed to request a service from software that runs at a higher privilege level:

- The Supervisor Call (SVC) instruction enables User mode programs to request an Operating System service.
- The Hypervisor Call (HVC) instruction, available if the Virtualization Extensions are implemented, enables the guest OS to request Hypervisor services.
- The Secure Monitor Call (SMC) instruction, available if the Security Extensions are implemented, enables the Normal world to request Secure world services.

Any attempt to execute an instruction that is not recognized by the core generates an UNDEFINED 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 can therefore be 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 11-3 on page 11-9.

Separate vector tables can be configured for Secure PL1, Non-secure PL1, Secure monitor and Non-secure PL2 privilege levels. The table the core uses to look up the handler depends on current execution privilege, and also on what privilege or security state it is configured to be taken to.

You can write the exception handlers in either ARM or Thumb code. The CP15 SCTLR.TE bit is used to specify whether exception handlers will use ARM or Thumb. When handling exceptions, the prior mode, state, and registers of the core must be preserved so that the program can be resumed after the exception has been handled.

11.1.1 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, 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 (which has higher priority) 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.

More than one exception can potentially be generated at the same time, but some combinations are mutually exclusive. A prefetch abort marks an instruction as invalid and so cannot occur at the same time as an undefined instruction or `SVC` (and of course, an `SVC` instruction cannot also be an undefined instruction). These instructions cannot cause any memory access and therefore cannot cause a data abort. The architecture does not define when asynchronous exceptions, FIQ, IRQ or asynchronous aborts must be taken, but the fact that taking an IRQ or data abort exception does not disable FIQ exceptions means that FIQ execution will be prioritized over IRQ or asynchronous abort handling.

Exception handling on the core is controlled through the use of an area of memory called the vector table. This lives 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.

The situation is more complicated for cores with Security Extensions. Here there are three vector tables, Non-secure, Secure and Secure Monitor. For cores with the Virtualization Extension there are four, adding a Hypervisor vector table. For cores with an MMU, all these vector addresses are virtual.

Table 11-1 Summary of exception behavior

Normal Vector offset	High vector address	Non-secure	Secure	Hypervisor[a]	Monitor
`0x0`	0xFFFF0000	Not used	Reset	Reset	Not used
`0x4`	0xFFFF0004	UNDEFINED instruction	UNDEFINED instruction	UNDEFINED instruction from Hyp mode.	Not used
`0x8`	0xFFFF0008	Supervisor Call	Supervisor Call	Secure Monitor Call	Secure Monitor Call
`0xC`	0xFFFF000C	Prefetch Abort	Prefetch Abort	Prefetch Abort from Hyp mode.	Prefetch Abort

Table 11-1 Summary of exception behavior (continued)

Normal Vector offset	High vector address	Non-secure	Secure	Hypervisor[a]	Monitor
0x10	0xFFFF0010	Data Abort	Data Abort	Data Abort from Hyp mode,	Data Abort
0x14	0xFFFF0014	Not used	Not used	Hyp mode entry	Not used
0x18	0xFFFF0018	IRQ interrupt	IRQ interrupt	IRQ interrupt	IRQ interrupt
0x1C	0xFFFF001C	FIQ interrupt	FIQ interrupt	FIQ interrupt	FIQ interrupt

a. Hypervisor entry exception (described in Chapter 22 *Virtualization*) is available only in cores that support Virtualization Extensions and is unused in other cores.

11.1.2 Exception mode summary

Table 11-2 lists the state of the interrupt disabling I and F bits of the CPSR on entering an exception handler.

Table 11-2 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	HYP	-
IRQ interrupt	IRQ	I = 1
FIQ interrupt	FIQ	F = 1 I = 1

11.1.3 The Vector table

The first column in Table 11-1 on page 11-6 gives the vector offset within the vector table associated with the particular type of exception. This is a table of instructions that the ARM core jumps to when an exception is raised. These instructions are located in a specific place in memory. The default vector base address is `0x00000000`, but most ARM cores permit the vector base address to be moved to `0xFFFF0000` (or `HIVECS`). All Cortex-A series processors permit this, and it is the default address selected by the Linux kernel. Cores that implement the Security Extensions can additionally set the vector base address, separately for Secure and Non-secure states, using the CP15 Vector Base Address registers.

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). Therefore, the vector table entry almost always contains one of the following two 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).

When the core is operating in Hyp mode, it uses Hyp mode vector entries, that are taken from a dedicated vector table belonging to the hypervisor. Hypervisor mode is entered through a special exception process called *Hyp trap entry* that makes use of the previously reserved `0x14` address within the vector table. A dedicated register (the Hyp Syndrome Register) gives information to the hypervisor about the exception or other reason for entering the hypervisor (a trapped CP15 operation, for example).

11.1.4 FIQ and IRQ

FIQ is 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 allows state to be retained between calls to the FIQ handler, again increasing execution speed by potentially removing the need to push some registers before using them.

A further 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 which have all their memory mapped and no need to make SVC calls to access kernel functions (so FIQ can be used only by code which does not need to use the kernel API).

FIQ is not typically used by Linux. As the kernel is architecture-independent, it does not have a concept of multiple forms of interrupt. Some systems running Linux can still make use of FIQ, but as the Linux kernel never disables FIQs they have priority over anything else in the system and so some care is required.

11.1.5 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 must be modified as shown in Table 11-3 on page 11-9, 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 11-3 also supplies an example return instruction from the exception that includes this adjustment.

Table 11-3 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

11.2 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 a 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.
4. Sets the PC to point to the relevant instruction from the exception vector table.

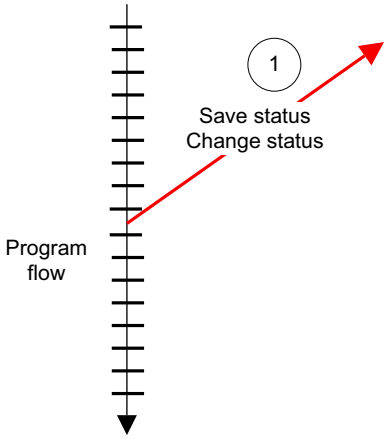


Figure 11-1 Taking the exception

When in the new mode the core will access the register associated with that mode, as shown in Figure 3-5 on page 3-7.

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.

11.2.1 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.

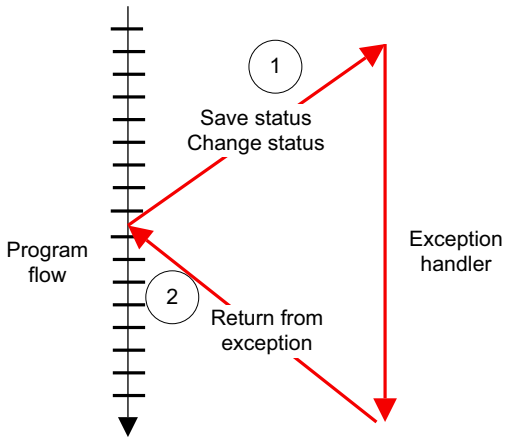


Figure 11-2 Returning from an exception

2. Set the PC to the return address offset, see Table 11-1 on page 11-6.

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 (See *RFE* on page A-35) restores the PC and SPSR from the stack of the current mode.

  ```
  RFEFD sp!
  ```

11.3 Other exception handlers

This section briefly describes handlers for aborts, undefined instructions and SVC exceptions and considers how interrupts are handled by the Linux kernel. Reset handlers are covered in Chapter 13 *Boot Code*.

11.3.1 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.

In systems that support virtual memory using an MMU, the abort handler can load the required virtual page into physical memory. In effect, it tries to fix the cause of the original abort and then return to the instruction that aborted and re-execute it. Chapter 9 gives more information about how Linux does this.

CP15 registers provide the address of the memory access that caused an abort (the Fault Address Register) and the reason for the abort (Fault Status Register). The reason might be lack of access permission, an external abort or an address translation fault. 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 that caused the abort exception. By examining these registers, the last instruction executed and possibly other things in the system, for example, translation table entries, the abort handler can determine what action to take.

11.3.2 Undefined instruction handling

An undefined instruction exception is taken if the core tries to execute an instruction with an opcode, described in the ARM architecture specification as UNDEFINED, 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 is possible that the VFP hardware cannot handle the particular instruction and wants to call software to emulate it. Alternatively, the VFP hardware might be disabled, and you take the exception so that it can be enabled and then 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 that makes use of undefined or coprocessor instructions, the handler for the exception must 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 Chapter 24 *Debug* for more information on breakpoints. See also the description of the Linux context switch for VFP in *Context switching* on page 6-10.

11.3.3 SVC exception handling

A supervisor call (`SVC`) is typically used to enable 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 exception handler might have to determine whether the core was in ARM or Thumb state when the exception occurred.

SVC handlers, especially, might have to read the instruction set state. This is done by examining the SPSR T-bit. This bit is set for Thumb state and clear for ARM state.

Both ARM and Thumb instruction sets have the SVC instruction. When calling SVCs from Thumb state, you must consider the following:

- The instruction address is at LR-2, rather than LR-4.
- The instruction itself is 16-bit, and so requires a halfword load,
- The SVC number is held in 8 bits instead of the 24 bits in ARM state.
- The SVC number is held in 8 bits instead of the 24 bits in ARM state.

Code to illustrate SVC usage with the Linux kernel is shown in Example 11-1.

Example 11-1 Linux kernel SVC usage

```
_start:
     MOV     R0, #1              @ STDOUT
     ADR     R1, msgtext         @ Address
     MOV     R2, #13             @ Length
     MOV     R7, #4              @ sys_write
     SVC     #0
     ....
  .align 2
  msgtxt:
     .asciz "Hello World\n"
```

The SVC #0 instruction makes the ARM core take the SVC exception, the mechanism to access a kernel function. Register R7 defines which system call you want (in this case, sys_write). The other parameters are passed in registers; for sys_write you have R0 telling where to write to, R1 pointing to the characters to be written and R2 giving the length of the string.

11.4 Linux exception program flow

Linux utilizes a cross-platform framework for exception handling that does not distinguish between different privileged core modes when handling exceptions. Therefore, the ARM implementation uses an exception handler stub to enable the kernel to handle all exceptions in SVC mode. All exceptions other than SVC and FIQ use the stub to switch to SVC mode and invoke the correct exception handler.

11.4.1 Boot process

During the boot process, the kernel will allocate a 4KB page as the vector page. It maps this to the location of the exception vectors, virtual address `0xFFFF0000` or `0x00000000`. This is done by `devicemaps_init()` in the file `arch/arm/mm/mmu.c`. This is invoked very early in the ARM system boot. After this, `trap_init` (in `arch/arm/kernel/traps.c`), copies the exception vector table, exception stubs and kuser helpers into the vector page. The exception vector table obviously has to be copied to the start of the vector page, the exception stubs being copied to address `0x200` (and kuser helpers copied to the top of the page, at `0x1000 - kuser_sz`), using a series of `memcpy()` operations, as shown in Example 11-2.

Example 11-2 Copying exception vectors during Linux boot

```
unsigned long vectors = CONFIG_VECTORS_BASE;

memcpy((void *)vectors, __vectors_start, __vectors_end - __vectors_start);
memcpy((void *)vectors + 0x200, __stubs_start, __stubs_end - __stubs_start);
memcpy((void *)vectors + 0x1000 - kuser_sz, __kuser_helper_start, kuser_sz);
```

When the copying is complete, the kernel exception handler is in its runtime dynamic status, ready to handle exceptions

11.4.2 Interrupt dispatch

There are two different handlers, `__irq_usr` and `__irq_svc`. These save all of the core registers and use a macro `get_irqnr_and_base` that indicates if there is an interrupt pending. The handlers loop around this code until no interrupts remain. If there is an interrupt, the code will branch to `do_IRQ` that exists in `arch/arm/kernel/irq.c`.

At this point, the code is the same in all architectures and you call an appropriate handler written in C.

There is, however, an additional point to consider. When the interrupt is completed, you would normally have to check whether or not the handler has done something that requires the kernel scheduler to be called. If the scheduler decides to go to a different thread, the one that was originally interrupted stays dormant until it is selected to run again.

Chapter 12
Interrupt Handling

Older versions of the ARM architecture enabled 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 *Generic Interrupt Controller* v2 (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.

12.1 External interrupt requests

Types of exception on page 11-3, described how all ARM cores 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 core 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 and if the corresponding input is asserted.

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 IE` or `CPS 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-A series processors, it is possible to configure the core 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 core is reset. They will still be masked automatically on taking an FIQ exception.

12.1.1 Assigning interrupts

A system will always have an interrupt controller that accepts and arbitrates interrupts from multiple sources. This typically contains a number of registers enabling software running on the core to mask individual interrupt sources, to acknowledge interrupts from external devices, to assign priorities to individual interrupt sources and to determine which 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* on page 12-7.

12.1.2 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. This is not generally suitable for complex embedded systems, but it is useful to examine before proceeding to a more realistic example, in this case of a non re-entrant interrupt handler.

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.

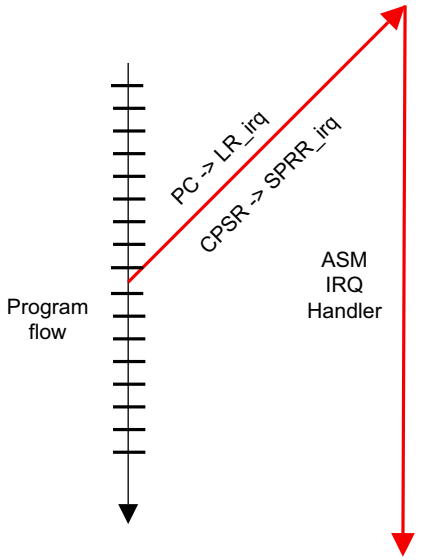


Figure 12-1 Save the context of the program

2. The instruction at the IRQ entry in the vector table (a branch to the interrupt handler) is executed.

3. 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 will be popped from stack when the handler finishes execution.

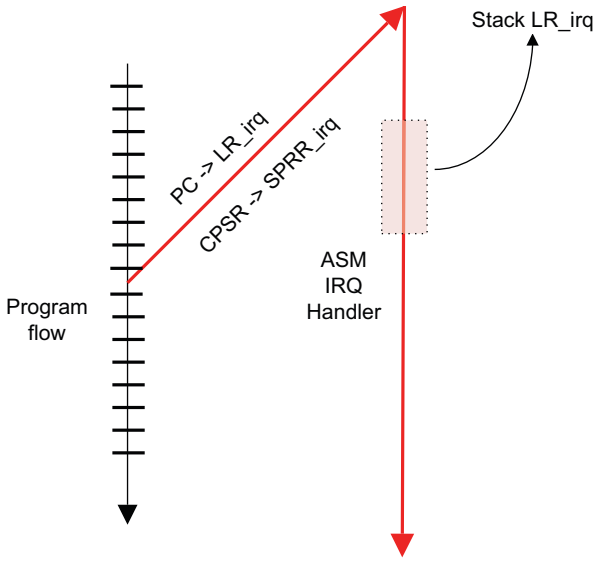


Figure 12-2

4. The interrupt handler determines which interrupt source must be processed and calls the appropriate device driver.

5. 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.

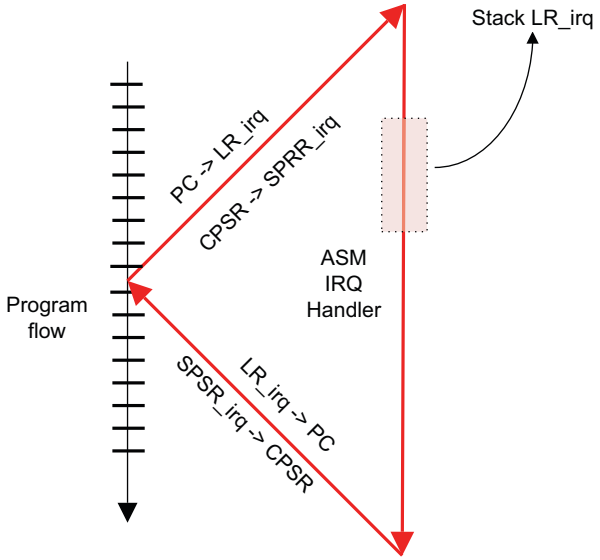


Figure 12-3

The same sequence is also applicable to an FIQ interrupt.

A very simple interrupt handler is shown in Example 12-1.

Example 12-1 Simple interrupt handler

```
IRQ_Handler
    PUSH   {r0-r3, r12, lr}    @ Store AAPCS registers and LR onto the IRQ mode stack
    BL                         @ identify_and_clear_source
    BL                         @ C-irq_handler
    POP    {r0-r3, r12, lr}    @ Restore registers and
    SUBS   pc, lr, #4          @ return from exception using modified LR
```

12.1.3 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 modes, 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. When the original interrupt attempts to return to the main program, it will cause the system to fail. The nested handler must change into an alternative kernel mode before re-enabling interrupts in order to prevent this.

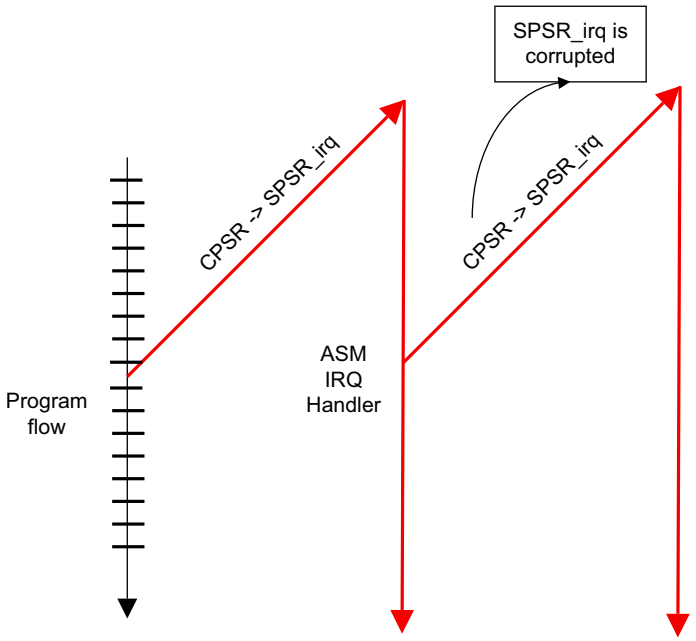


Figure 12-4 Nested interrupts

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 version has completed.

In Figure 12-4 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:

```
SRSFD  sp!, #0x12
```

Additionally, using the BL instruction within the interrupt handler code will cause LR_IRQ corruption. 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 alternative kernel 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 the core SVC mode, leaving the CPSR I bit set (interrupts are still disabled).
4. The interrupt handler saves the exception return address on the stack (a stack for the new mode, located in kernel memory) 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 alternative kernel 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 in Example 12-2.

Example 12-2 Nested interrupt handler

```
IRQ_Handler
    SUB     lr, lr, #4

    SRSFD    sp!, #0x1f      @ use SRS to save LR_irq and SPSR_irq in one step onto the
                             @ System mode stack

  CPS        #0x1f           @ Use CPS to switch to system mode

  PUSH       {r0-r3, r12}    @S tore remaining AAPCS registers on the System mode stack
  AND        r1, sp, #4      @ Ensure stack is 8-byte aligned. Store adjustment and
                             @ LR_sys 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_sys
  ADD         sp, sp, r1     @ Unadjust stack
  POP         {r0-r3, r12}   @ Restore AAPCS registers
  RFEFD       sp!            @ Return using RFE from the System mode stack.
```

12.2 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. It also provides support for the TrustZone Security Extensions described in Chapter 21 *Security*. 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 marked in 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.

More detailed information on GIC behavior can be found in the TRMs for the individual processor types and in the ARM *Generic Interrupt Controller Architecture* specification.

12.2.1 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. See *Handling interrupts in an SMP system* on page 18-14 for more details.

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:

12.2.2 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.

12.2.3 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.

As well as returning the interrupt ID, the read causes the interrupt to be marked as active in the distributor. Once 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 above 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.

Chapter 13
Boot Code

This chapter considers the boot code running in an ARM processor based system, and focuses on two distinct areas:

- Code to be run immediately after the core comes out of reset, on a so-called bare-metal system, that is, one in which code is run without the use of an operating system. This is a situation that is often encountered when first starting up a chip or system.
- How a bootloader loads and runs the Linux kernel.

13.1 Booting a bare-metal system

When the core has been reset, it will commence execution at the location of the reset vector in the exception vector table (at either address `0x00000000` or `0xFFFF0000`, see Table 11-3 on page 11-9). The reset handler code must do some, or all of the following:

- In a multi-core system, enable non-primary cores to sleep See *Booting SMP systems* on page 18-17.
- Initialize exception vectors.
- Initialize the memory system, including the MMU.
- Initialize core mode stacks and registers.
- Initialize any critical I/O devices.
- Perform any necessary initialization of NEON or VFP.
- Enable interrupts.
- Change core mode or state.
- Handle any set-up required for the Secure world (see Chapter 21).
- 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. The initial vector table will be in non-volatile memory and can contain branch to self instructions (other than the reset vector) as no exceptions are expected at this point. Typically, the reset vector contains a branch to the boot code in ROM. The ROM can be aliased to the address of the exception vector. The ROM then writes to some memory-remap peripheral that maps RAM into address 0 and the real exception vector table is copied into RAM. This means the part of the boot code that handles remapping must be position-independent, as only PC-relative addressing can be used. Example 13-1 shows an example of typical code that can be placed in the exception vector table.

Example 13-1 Typical exception vector table code

```
start
  B    Reset_Handler
  B    Undefined_Handler
  B    SWI_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. Example 13-2 on page 13-3 gives a simple example, showing code that initializes the stack pointers for FIQ and IRQ modes.

Example 13-2 Code to initialize the stack pointers

```
  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, MMU and branch predictors. An example of such code is shown in Example 13-3. We begin by disabling the MMU and caches and invalidating the caches and TLB. This example code is for the Cortex-A9 processor. Some of the Cortex-A processors automatically invalidate the L1 and/or L2 caches at reset, others require manual invalidation. You must check the TRM for a particular core to determine which options have been implemented.

The MMU TLBs must be invalidated. The branch target predictor hardware might not have to be explicitly invalidated, but it must be enabled by boot code. Branch prediction can safely be enabled at this point; this will improve performance.

Example 13-3 Setting up caches, MMU and branch predictors

```
  @ Disable MMU
  MRC   p15, 0, r1, c1, c0, 0        @ Read Control Register configuration data
  BIC   r1, r1, #0x1
  MCR   p15, 0, r1, c1, c0, 0        @ Write Control Register configuration data

  @ Disable L1 Caches
  MRC   p15, 0, r1, c1, c0, 0        @ Read Control Register configuration data
  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 L1 Caches
  @ Invalidate Instruction cache
  MOV   r1, #0
  MCR   p15, 0, r1, c7, c5, 0

  @ 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
  LDR 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
  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

  @ Invalidate TLB
  MCR   p15, 0, r1, c8, c7, 0

  @ Branch Prediction Enable
  MOV   r1, #0
  MRC   p15, 0, r1, c1, c0, 0         @ Read Control Register configuration data
  ORR   r1, r1, #(0x1 << 11)          @ Global BP Enable bit
  MCR   p15, 0, r1, c1, c0, 0         @ Write Control Register configuration data
```

After this, you can create some translation tables, as shown in the example code of Example 13-4. The variable `ttb_address` is used to denote the address to be used for the initial translation table. This must be a 16KB area of memory (whose start address is aligned to a 16KB boundary), to which an L1 translation table can be written by this code.

Example 13-4 Create translation tables

```
  @ Enable D-side Prefetch
  MRC   p15, 0, r1, c1, c0, 1         @ Read Auxiliary Control Register
  ORR   r1, r1, #(0x1 <<2)            @ Enable D-side prefetch
  MCR   p15, 0, r1, c1, c0, 1 ;       @ Write Auxiliary Control Register
  DSB
  ISB
  @ DSB causes completion of all cache maintenance operations appearing in program
  @ order before the DSB instruction
  @ An ISB instruction causes the effect of all branch predictor maintenance
  @ operations before the ISB instruction to be visible to all instructions
  @ after the ISB instruction.
  @ Initialize PageTable

  @ We will create a basic L1 page table in RAM, with 1MB sections containing a flat
(VA=PA) mapping, all pages Full Access, Strongly Ordered

  @ It would be faster to create this in a read-only section in an assembly file

  LDR   r0, =2_00000000000000000000110111100010 @ r0 is the non-address part of
descriptor
  LDR   r1, ttb_address
  LDR   r3, = 4095                    @ loop counter
write_pte
  ORR   r2, r0, r3, LSL #20           @ OR together address & default PTE bits
  STR   r2, [r1, r3, LSL #2]          @ write PTE to TTB
  SUBS  r3, r3, #1                    @ decrement loop counter
  BNE   write_pte

  @ for the very first entry in the table, we will make it cacheable, normal,
write-back, write allocate
  BIC   r0, r0, #2_1100               @ clear CB bits
  ORR   r0, r0, #2_0100               @ inner write-back, write allocate
  BIC   r0, r0, #2_111000000000000    @ clear TEX bits
  ORR   r0, r0, #2_101000000000000    @ set TEX as write-back, write allocate
  ORR   r0, r0, #2_10000000000000000  @ shareable
  STR   r0, [r1]

  @ Initialize MMU
  MOV   r1,#0x0
  MCR   p15, 0, r1, c2, c0, 2         @ Write Translation Table Base Control Register
  LDR   r1, ttb_address
```

```
  MCR   p15, 0, r1, c2, c0, 0         @ Write Translation Table Base Register 0

  @ In this simple example, we don't use TRE or Normal Memory Remap Register.
  @ Set all Domains to Client
  LDR r1, =0x55555555
  MCR p15, 0, r1, c3, c0, 0           @ Write Domain Access Control Register

  @ Enable MMU
  MRC   p15, 0, r1, c1, c0, 0         @ Read Control Register configuration data
  ORR   r1, r1, #0x1                  @ Bit 0 is the MMU enable
  MCR   p15, 0, r1, c1, c0, 0         @ Write Control Register configuration data
```

The L2 cache, if present, and if running without an operating system, might also require invalidating and enabling at this point. NEON or VFP access must also be enabled. If the system makes use of the TrustZone Security Extensions, it might have to switch to the Normal world when the Secure world is initialized. See Chapter 21 *Security* for details of this.

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, and set up application stack and heap spaces. It might also be necessary to initialize C library functions, call top-level constructors (for C++ code) and other standard embedded C initialization.

A common approach is to permit a single core within the cluster to perform system initialization, while the same code, if run on a different core, will cause it to sleep, that is, enter `WFI` state, as described in Chapter 20. The other cores might be woken after core 0 has created a simple set of L1 translation table entries, as these could be used by all cores in the system. Example 13-5 shows example code that determines which core it is running on and either branches to initialization code (if running on core 0), or goes to sleep otherwise. The secondary cores are typically woken up later by an SMP OS.

Example 13-5 Determining which core is running

```
  @ 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
```

13.2 Configuration

There are a number of control register bits within the core that will typically be set by boot code. In all cases, for best performance, code must run with the MMU, instruction and data caches and branch prediction enabled. Translation table entries for all regions of memory that are not peripheral I/O devices must be marked as L1 Cacheable and (by default) set to read-allocate, write-back cache policy. For multi-core systems, pages must be marked as Shareable and the broadcasting feature for CP15 maintenance operations must be enabled.

In addition to the CP15 registers required by the ARM architecture, cores typically have registers that control implementation-specific features. Programmers of boot code should refer to the relevant technical reference manual for the correct usage of these.

13.3 Booting Linux

It is useful to understand what happens from the core coming out of reset and executing its first instruction at the exception base address `0x00000000` or `0xFFFF0000` if `HIVECS` (known as high vectors) is selected, until the Linux command prompt appears. (See *The Vector table* on page 11-7.)

When the kernel is present in memory, the sequence on an ARM processor based system is similar to what might happen on a desktop computer. However, the bootloading process can be very different, as ARM processor based phones or more deeply embedded devices can lack a hard drive or PC-like BIOS.

Typically, what happens when you power the system on is that hardware specific boot code runs from flash or ROM. This code initializes the system, including any necessary hardware peripheral code and then launches the bootloader (for example U-Boot). This initializes main memory and copies the compressed Linux kernel image into main memory (from a flash device, memory on a board, MMC, host PC or elsewhere). The bootloader passes certain initialization parameters to the kernel. The Linux kernel then decompresses itself and initializes its data structures and running user processes, before starting the command shell environment. Let's take a more detailed look at each of those processes.

13.3.1 Reset handler

There is typically a small amount of system-specific boot monitor code that configures memory controllers and performs other system peripheral initialization. It sets up stacks in memory and typically copies itself from ROM into RAM, before changing the hardware memory mapping so that RAM is mapped to the exception vector address, rather than ROM. In essence this code is independent of that operating system is to be run on the board and performs a function similar to a PC BIOS. When it has completed execution, it will call a Linux bootloader, such as U-Boot.

13.3.2 Bootloader

Linux requires a certain amount of code to be run out of reset, to initialize the system. This performs the basic tasks required for the kernel to boot:

- Initializing the memory system and peripherals.
- Loading the kernel image to an appropriate location in memory (and possibly also an initial RAM disk).
- Generate the boot parameters to be passed to the kernel (including machine type).
- Set up a console (video or serial) for the kernel.
- Enter the kernel.

The exact steps taken vary between different bootloaders, so for detailed information, refer to documentation for the one that you want to use. U-Boot is a widely used example, but other bootloader possibilities include Apex, Blob, Bootldr and Redboot.

When the bootloader starts, it is typically not present in main memory. It must start by allocating a stack and initializing the core (for example invalidating its caches) and installing itself to main memory. It must also allocate space for global data and for use by `malloc()` and copy exception vector entries into the appropriate location.

13.3.3 Initialize memory system

This is very much a board or system specific piece of code. The Linux kernel has no responsibility for the configuration of the RAM in the system. It is presented with the physical memory layout, but has no other knowledge of the memory system. In many systems, the available RAM and its location are fixed and the bootloader task is straightforward. In other systems, code must be written that discovers the amount of RAM available in the system.

13.3.4 Kernel images

The kernel image from the build process is typically compressed in zImage format (the conventional name given to the bootable kernel image). Its head code contains a magic number, used to verify the integrity of the decompression, plus start and end addresses. The kernel code is position independent and can be located anywhere in memory. Conventionally, it is placed at a `0x8000` offset from the base of physical RAM. This gives space for the parameter block placed at a `0x100` offset (used for translation tables etc).

Many systems require an initial RAM disk (initrd), as this lets you have a root filesystem available without other drivers being setup. The bootloader can place an initial ramdisk image into memory and pass the location of this to the kernel using `ATAG_INITRD2` (a tag that describes the physical location of the compressed RAM disk image) and `ATAG_RAMDISK`.

The bootloader will typically setup a serial port in the target, enabling the kernel serial driver to detect the port and use it for a console. In some systems, another output device such as a video driver can be used as a console. The kernel command line parameter `console=` can be used to pass the information.

13.3.5 Kernel parameters using ATAGs

Historically, the parameters passed to the kernel are in the form of a tagged list, placed in physical RAM with register R2 holding the address of the list. Tag headers hold two 32-bit unsigned `int`s, with the first giving the size of the tag in words and the second providing the tag value (indicating the type of tag). For a full list of parameters that can be passed, consult the appropriate documentation. Examples include `ATAG_MEM` to describe the physical memory map and `ATAG_INITRD2` to describe where the compressed ramdisk image is located. The bootloader must also provide an ARM Linux machine type number (`MACH_TYPE`). This can be a hard-coded value, or the boot code can inspect the available hardware and assign a value accordingly.

There is a more flexible, or generic method for passing this information using *Flattened Device Trees* (FDTs).

13.3.6 Kernel parameters using Flattened Device Trees

The Linux device tree or FDT support was introduced for the PowerPC kernel as a part of the merger of a 32-bit and 64-bit kernel to standardize the firmware interface by using an Open Firmware interface for all PowerPC platforms, servers, desktop and embedded. It has become the configuration methodology used in the Linux kernel for PowerPC, Micro Blaze and SPARC architectures.

A device tree is a data structure that describes the hardware configuration. It includes information about processors, memory sizes and banks, interrupt configuration, and peripherals. The data structure is organized as a tree with a single root node named /. With the exception of the root node, each node has a single parent. Each node has a name and can have any number of child nodes. Nodes can also contain named properties values with arbitrary data, and they are expressed in key-value pairs.

The device tree data format follows the conventions defined in IEEE standard 1275. To simplify system description, a device tree source format (`.dts`) is used to express device tree data.

A device tree node must comply with the following syntax:

```
[label:] node-name[@unit-address] {
  [properties definitions]
  [child nodes]
}
```

Nodes are defined with a name and a unit-address. Braces mark the beginning and end of the node definition.

You can use a *Device Tree Compiler* (DTC) tool to convert the device tree source file (`.dts`) to the device tree blob (`dtb`) format. The `dtb`, or blob, is known as the *Flattened Device Tree* and is a firmware independent description of the system, in a compressed format that requires no firmware calls to retrieve its properties. The Linux kernel loads the dtb before it loads the operating system.

The chosen node is a placeholder for any environment information that does not belong anywhere else, such as boot arguments for the kernel and default console. Chosen node properties are usually defined by the boot loader, but the `dts` file can specify a default value.

The following code fragment shows a root node description for an ARM Versatile Platform Board. The model and compatible properties are assigned the name of the platform in the form <*manufacturer*>,<*model*>. This string concatenation is the universal identifier for the machine and must be defined at the top node.

```
/ {
    model = "arm,versatilepb";
    compatible = "arm,versatilepb";
    #address-cells = <1>;
    #size-cells = <1>;

memory {
        name = "memory";
        device_type = "memory";
        reg = <0x0 0x08000000>;
};

chosen {
        bootargs = “console=ttyAMA0 debug”;
        }
};
```

13.3.7 Kernel entry

Kernel execution must commence with the core in a fixed state. The bootloader calls the kernel image by branching directly to its first instruction, the `start` label in `arch/arm/boot/compressed/head.S`. The MMU and data cache must be disabled. The core must be in Supervisor mode, with CPSR I and F Bits set (IRQ and FIQ disabled). R0 must contain 0, R1 the `MACH_TYPE` value and R2 the address of the tagged list of parameters.

The first step in getting the kernel working is to decompress it. This is mostly architecture independent. The parameters passed from the bootloader are saved and the caches and MMU are enabled. Checks are made to see if the decompressed image will overwrite the compressed image, before calling `decompress_kernel()` in `arch/arm/boot/compressed/misc.c`, The cache is then cleaned and invalidated before being disabled again. We then branch to the kernel startup entry point in `arch/arm/kernel/head.S`.

13.3.8 Platform-specific actions

A number of architecture specific tasks are now undertaken. The first checks core type using `__lookup_processor_type()` that returns a code specifying which core it is running on. The function `__lookup_machine_type()` is then used (unsurprisingly) to look up machine type. A basic set of translation tables is then defined which map the kernel code. The cache and MMU are initialized and other control registers set. The data segment is copied to RAM and `start_kernel()` is called.

13.3.9 Kernel start-up code

In principle, the rest of the startup sequence is the same on any architecture, but in fact some functions are still hardware dependent.

1. IRQ interrupts are disabled with `local_irq_disable()`, while `lock_kernel()` is used to stop FIQ interrupts from interrupting the kernel. It initializes the tick control, memory system and architecture-specific subsystems and deals with the command line options passed by the bootloader.
2. Stacks are set up and the Linux scheduler is initialized.
3. The various memory areas are set-up and pages allocated.
4. The interrupt and exception table and handlers are setup, along with the GIC.
5. The system timer is setup and at this point IRQs are enabled. Additional memory system initialization occurs and then a value called *BogoMips* is used to calibrate the core clock speed.
6. Internal components of the kernel are set up, including the filesystem and the initialization process, followed by the thread daemon that creates kernel threads.
7. The kernel is unlocked (FIQ enabled) and the scheduler started.
8. The function `do_basic_setup()` is called to initialize drivers, sysctl, work queues and network sockets. At this point, the switch to User mode is performed.

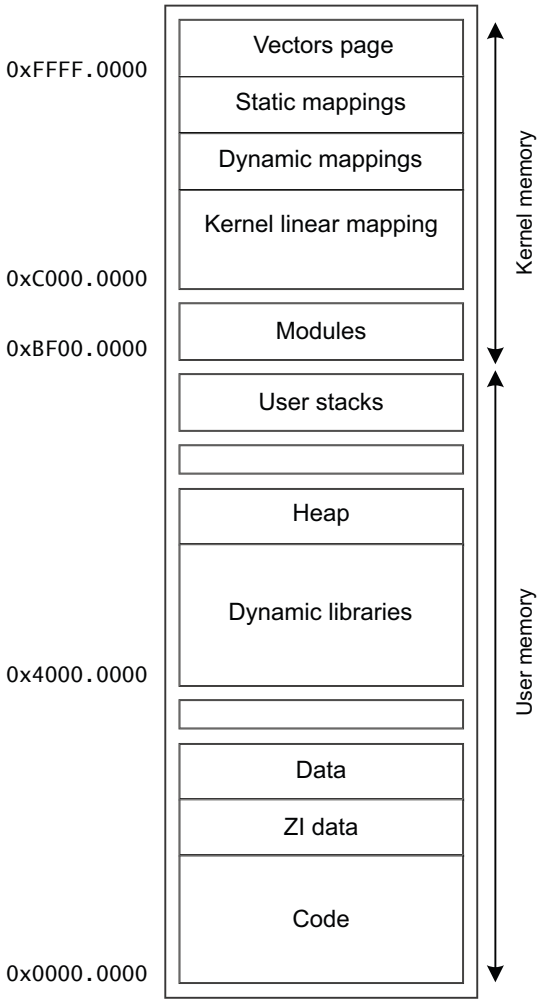


Figure 13-1 Linux virtual memory view

The memory map used by Linux is shown in Figure 13-1. ZI refers to zero-initialized data. There is a broad split between kernel memory, above address `0xBF000000` and user memory, below that address. Kernel memory uses global mappings, while user memory uses non-global mappings, although both code and data can be shared between processes. As already mentioned, application code starts at `0x1000`, leaving the first 4KB page unused, to enable trapping of NULL pointer references.

Chapter 14
Porting

New projects will normally use an existing operating system and re-use code from existing applications. New code might be targeted at the ARMv7-A architecture, but might eventually require porting to a different board. There are a number of issues associated with porting code from a different architecture to run on an ARM processor, or from older versions of the ARM architecture to ARMv7-A.

For many applications, particularly those coded with portability issues in mind, this will mean recompiling the source code. For example, a large amount of Linux application software is designed to run in many different environments and tends to make fewer assumptions about the underlying hardware. However, there are a number of areas where C code is not fully portable. In particular, low level, hardware-specific code such as device drivers might require more effort than porting applications.

There is an additional consideration when porting code between processors: that of efficiency. It might be the case that optimizations applied to code running on another processor, or to older versions of the ARM architecture, do not apply to that code when running on ARMv7-A. Equally, there might be scope for making code smaller or faster on ARMv7-A class processors. Optimizations might differ between processors or systems. Code that is optimal for one processor might not be optimal for others. ARM-specific optimization is covered in Chapter 17.

14.1 Endianness

The use of the terms *little-endian* and *big-endian* was introduced by Danny Cohen in his 1980 paper “On Holy Wars and a Plea for Peace”. Cohen has also been responsible for many advances in the fields of networks and computer graphics. It is a reference to *Gulliver's Travels*, a famous satire from the early 18th century, by Irish writer Jonathan Swift, in which a war is fought between the fictional countries of Lilliput and Blefuscu over the correct end to open a boiled egg.

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 discuss these.

Consider the following simple piece of code (Example 14-1):

Example 14-1 Endian access

```
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 14-1 on page 14-3 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.

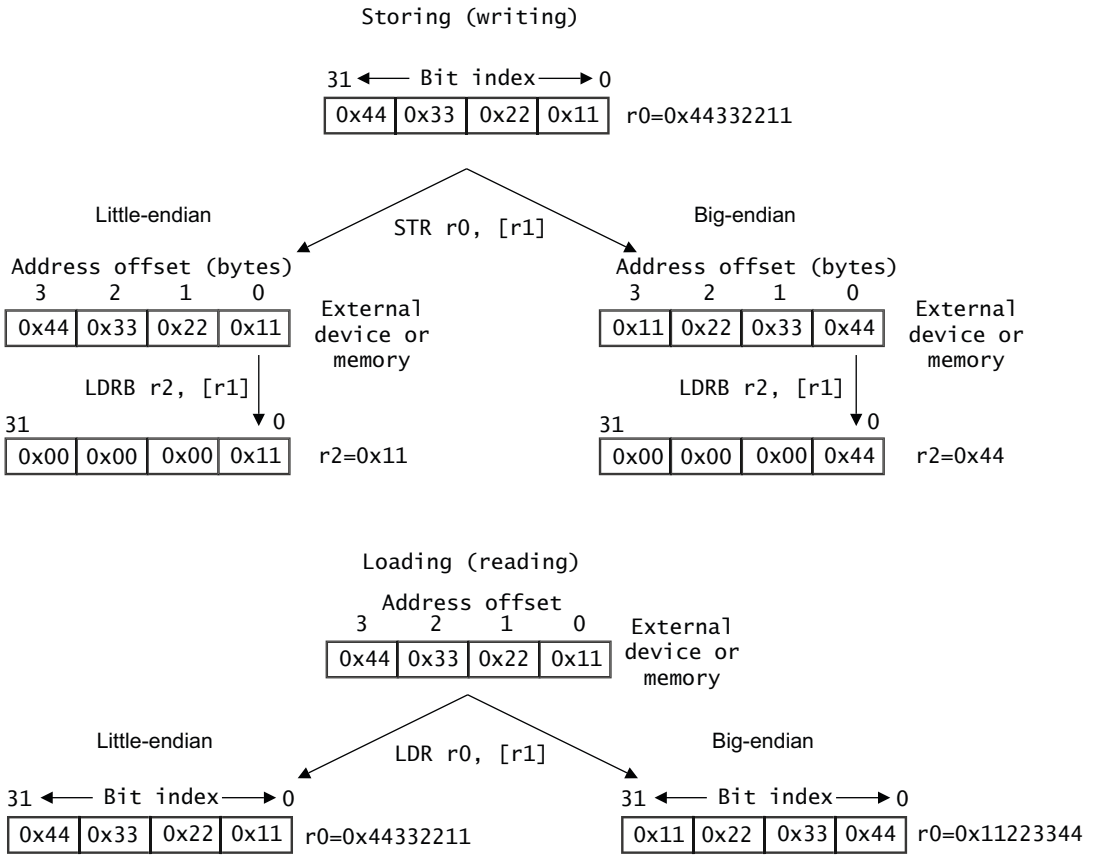


Figure 14-1 Different endian behaviors

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. The PowerPC or the venerable 68K, on the other hand, are generally big-endian, although the Power architecture can also handle little-endian. Several common file formats and networking protocols specify different endianness. For example, .BMP and .GIF files are little-endian, while .JPG is big-endian, and TCP/IP is big-endian, but USB and PCI are 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-A 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* on page 5-18) can be used to reverse bytes within an ARM register, providing simple conversion between big and little-endian formats.

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` assembly instruction provided to do this. The CP15:SCTLR (System Control Register, c1), contains the EE bit (see *Coprocessor 15* on page 3-9) that defines the endian mode to switch to on an exception and also the endianness of translation table lookups.

It would be difficult if exception code had to worry about which endian state the core was in on arrival at the handler. In practice, however, it can be difficult to tell the compiler that part of the system is of a different endian configuration to the rest of memory.

Modern ARM processors support a big-endian format known architecturally as BE8 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. BE8 corresponds to what most other computer architectures call big-endian.

Example 14-2 provides a simple piece of code that behaves differently when run on architectures with different endianness.

Example 14-2 Non-portable code

```
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.

14.2 Alignment

The alignment of accesses is significant on ARM cores. On older ARM processors, accesses to addresses that are not aligned are possible, but with a different behavior to those using the ARMv7 architecture. On ARM7 and ARM9 processors, an unaligned `LDR` is performed in the memory system in the same way as an aligned access, but with the data returned being rotated so that the data at the requested address is placed in the least significant byte of the loaded register. Some older compilers and operating systems were able to use this behavior for clever optimizations. This can represent a portability problem when moving code from an ARMv4 or ARMv5 to ARMv7 architecture.

ARM MMUs can be configured to automatically detect such unaligned accesses and abort them (using the CP15:SCTL A bit), see *System control register (SCTLR)* on page 3-12 and *Coprocessor 15* on page 3-9.

For the Cortex-A series of processors, unaligned accesses are supported, although you must enable this by setting the U bit in the CP15:SCTL register, indicating that unaligned accesses are permitted. This means that instructions to read or write words or halfwords can access addresses that are not aligned to word or halfword boundaries. However, load and store multiple instructions (`LDM` and `STM`) and load and store double-word (`LDRD` or `STRD`) must be aligned to at least a word boundary. Loads and stores of floating-point values must always be aligned. Additional alignment constraints might be imposed by the ABI that are stronger than those imposed by the ARM architecture.

These unaligned accesses can take additional cycles in comparison with aligned accesses and therefore alignment is also a performance issue. In addition, such accesses are not guaranteed to be atomic. This means that a external agent (another core in the system) might perform a memory access that appears to occur part way through the unaligned access. For example, it might read the accessed location and see the *new* value of some bytes and the *old* value of others.

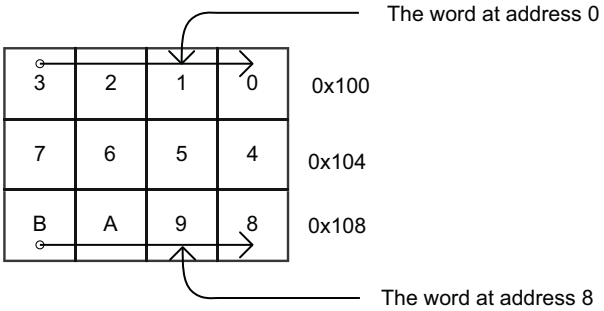


Figure 14-2 Aligned words at address 0 or 8

A word aligned address is one that is a multiple of four, for example `0x100`, `0x104`, `0x108`, `0x10C`, `0x110`. Figure 14-2 shows examples of aligned accesses.

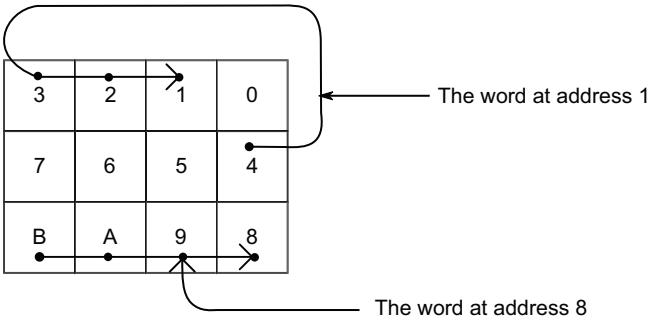


Figure 14-3 An unaligned word

An unaligned word at address 1 is shown in Figure 14-3. It takes three bytes from the word at 0 and one byte from the word at 4.

A simple example where alignment effects can have significant performance effects is the use of `memcpy()`. Copying small numbers of bytes between word aligned addresses will be compiled into `LDM` or `STM` instructions. Copying larger blocks of memory aligned to word boundaries will typically be done with an optimized library function that will also use `LDM` or `STM`. Copying blocks of memory whose start or end points do not fall on a word boundary can result in a call to a generic `memcpy()` function that can be significantly slower. Although, if the source and destination are similarly unaligned then only the start and end fragments are non-optimal. Whenever explicit typecasting is performed, that cast always carries alignment implications.

14.3 Miscellaneous C porting issues

In this section we consider some other possible causes of problems when porting C code.

14.3.1 unsigned char and signed char

The piece of code in Example 14-3 illustrates a very simple example of a possible issue when porting code to ARM.

Example 14-3 Use of unsigned char

```
char c = -1;
if (c > 0)  printf("c is positive \n");
else    printf("c is negative \n");
```

When this is compiled for some architectures (for example, for x86) the result is the one you might intuitively expect, which is that it reports the variable `c` as negative, but compiling the code on an ARM Compiler will produce code that reports `c` as positive, and typically a warning will be emitted by the compiler too.

The ANSI C standard specifies a range for both signed (at least -127 to +127) and unsigned (at least 0 to 255) chars. Simple `chars` are not specifically defined and it is compiler dependent whether they are signed or unsigned. Although the ARM architecture has the `LDRSB` instruction, that loads a signed byte into a 32-bit register with sign extension, the earliest versions of the architecture did not. It made sense at the time for the compiler to treat simple `chars` as unsigned, whereas on the x86 simple `chars` are, by default, treated as signed.

One workaround for users of GCC is to use the `-fsigned-char` command line switch or `--signed-chars` for RVCT, that forces all `chars` to become signed, but a better practice is to write portable code by declaring `char` variables appropriately. Unsigned `char` must be used for accessing memory as a block of bytes or for small unsigned integers. Signed `char` must be used for small signed integers and simple `char` must be used only for ASCII characters and strings. In fact, on an ARM core, it is usually better to use `int`s rather than `chars`, even for small values, for performance reasons. You can read more on this in Chapter 17 *Optimizing Code to Run on ARM Processors*.

A second piece of code, in Example 14-4, illustrates another possible problem with chars. Here `EOF` is compared with an `unsigned char`. On an ARM core, the `while` loop will never complete. The value of `EOF` is defined as -1 and when it is converted to be compared with a `char` (which is unsigned and therefore in the range 0 to 255), it can never be equal and so the loop does not terminate.

Example 14-4 Use of EOF

```
char c;
while ((c = getchar()) != EOF) putchar(c);
```

You must declare the variable as `int` instead of `char` to avoid the problem, in fact, this is how the functions in `stdio.h` are defined.

Similar cases to look out for include the use of `getopt()` and `getc()` – both are defined as returning an `int`.

14.3.2 Compiler packing of structures

Compilers are not permitted to re-order members of a structure and have to follow the alignment restrictions of the core architecture. This means that compilers might have to add unused bytes into user defined structures, for best performance and code size. Such padding is architecture specific and can therefore lead to portability problems if assumptions have been made about the location and size of this padding.

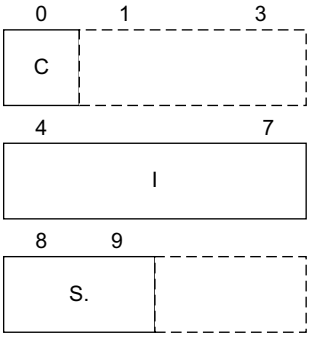
Marking a structure as `__packed` in the ARM Compiler or using the attribute `__packed__` in GCC, will remove any padding. This reduces the size of the structure and can be useful when porting code or for structures being passed from external hardware, but can reduce performance and increase code size, although generally it will be relatively efficient on Cortex-A series processors.

If you have some simple `struct` code, as shown in Example 14-5:

Example 14-5 A typical C struct

```
struct test
        {
           unsigned char  c;
           unsigned int i ;
           unsigned short s;
        }
```

Then the arrangement of data within the `struct` will be as in Figure 14-4.



Unpacked structure layout

Figure 14-4 Unpacked structure

If you now mark the structure as packed, as in Example 14-6:

Example 14-6 Packed structure

```
struct test
        {
           unsigned char  c;
           unsigned int i ;
           unsigned short s;
        } __attribute__((__packed__));
```

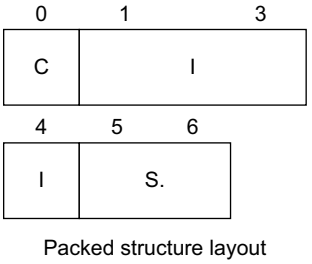


Figure 14-5 Packed structure

In Figure 14-5 byte 0 now holds `C`, bytes 1-4 hold `I` and bytes 5-6 will hold `S`. To access `S` or `I` will require the core to perform an unaligned access.

14.3.3 Use of the stack

Code that makes assumptions about the stack structure can require porting effort. For example, functions with a variable number and type of arguments can receive their variables through the stack. The `<stdarg.h>` macros dealing with accessing these arguments will walk through the stack frame and provide compatibility between systems, but code that does not use standard libraries or macros can have a problem.

14.3.4 Other issues

A function prototype is a declaration that omits the function body but gives the function name, argument and return types. It effectively gives a way to specify the interface of a function separately from its definition. Incorrectly prototyped functions can behave differently between different compilers.

Compilers can allocate different numbers of bytes to `enum`. Care is therefore required when enumerations are used; cross-linking of code and libraries between different compilers might not be possible if `enums` are used.

Code written for 8-bit or 16-bit cores might assume that integer variables are 16-bit. On ARMv7-A processors, they will always be 32 bits. The program might rely on 16-bit behaviors. In general, this is easily fixed by the use of the C short type. Use of `short ints` can be less efficient, as we shall see in the chapter on optimization and in cases where the code does not rely on 16-bit behavior, it is usually better to promote these variables to a 32-bit `int`.

14.4 Porting ARM assembly code to ARMv7-A

So far in this chapter, we have looked at porting C code from other architectures to ARM. It is sometimes necessary to port assembly code from older ARM processors to the Cortex-A series. Sometimes, it can be difficult to determine which ARM architecture variant your code was originally targeting. GCC has a series of macros, with names like `__ARM_ARCH_6__` which are mutually exclusive. The ARM Compiler has a similar set of macros such as `__TARGET_ARCH_7_A`. In general, ARM assembler code is backward compatible and will work unmodified.

There are a few special cases to look for:

CP15 Operations

The architecture specifies a consistent mapping of CP15 to designed system control operations. In general, you should attempt to understand the purpose of code that performs CP15 instructions and ensure that this code is appropriate for the ARMv7-A Architecture. In addition, a number of CP15 registers (for example CP15:ACTL, the Auxiliary Control Register) are implementation specific. Code that references such registers will always require attention when being ported.

SWP

The `SWP` (or `SWPB`) instruction was used to implement atomic operations in older versions of the ARM architecture, but is deprecated and its use strongly discouraged in the ARMv7 Architecture. There is no encoding for `SWP` in Thumb at all, so `SWP` is not permitted when building for Thumb. In the ARMv7 Architecture, `SWP` is disabled by default, but can be re-enabled by setting CP15:SYSCTL bit [10]. See *System control register (SCTLR)* on page 3-12. Code that uses `SWP` must be rewritten to make use of `LDREX` or `STREX` (and possibly also barrier instructions – see Chapter 10). Alternatively, the GCC `__sync_...` intrinsics could be used. It is usually preferable to use library functions for such things as spinlocks, semaphores, and mutexes, rather than writing such primitives yourself. The mechanisms used are different from those used by SWP, so it is necessary to port all code accessing an atomic object, not just some of it. Usually an atomic object will be managed by a piece of library code shared between the threads that access it, so this is not typically a problem. Cortex-A series cores treat `SWP` as Undefined out of reset.

14.4.1 Memory access ordering and memory barriers

The ARMv7 architecture has a weakly-ordered memory model. Code for older processors that makes implicit assumptions about ordering of memory accesses might not operate correctly on ARMv7 devices. This might be particularly true for code that interacts with other devices, such as a DMA controller or other bus master. Such code must be inspected and modified, possibly by the insertion of appropriate barrier instructions or by the use of suitable atomic primitives. See Chapter 10 for a more detailed description of memory ordering and memory barriers.

14.5 Porting ARM code to Thumb

We also consider problems associated with porting code written for the ARM instruction set to the Thumb instruction set. As we have seen, use of Thumb is often preferred because of its combination of small code size with higher performance relative to the older 16-bit only Thumb instruction set.

14.5.1 Use of PC as an operand

Instructions that use PC (R15) as an explicit operand can cause problems when assembling in Thumb. As we have seen, it is not possible to encode any arbitrary 32-bit address into any instruction as an operand. For this reason, it is common to address data stored in a literal pool using offsets relative to the current instruction location. In ARM code, the PC value can be used (with some adjustment) to determine the address of the currently executing instruction, for this purpose and this enables position-independent coding. However, the PC value obtained in this way can show some inconsistencies between ARM and Thumb states and can also depend on the type of instruction executed. For this reason, ARM assembly code that directly references the PC register might require modifying to work correctly in Thumb.

It is better to avoid explicit PC arithmetic and instead to use an instruction such as:

```
LDR     r0, =<value>
```

This automatically puts `<value>` somewhere in the text section and assembles an appropriate PC-relative `LDR` instruction. It will, however, only do this if `<value>` cannot be encoded in a `MOV` instruction.You can still do a load from a local text section label that you declare explicitly, as shown below, but again, the assembler or linker can be permitted to perform the PC offset calculation:

```
LDR     r0, data
...
data:   .long   <value>
```

Sometimes the required PC-relative address offset is too large to encode in a single `LDR` instruction, causing the assembler to complain that a literal pool is out of range. This can be resolved by explicitly placing a literal pool, using the `.ltorg` directive that tells the assembler where to insert literal data. You must ensure that the literal data is not located where it might be executed as code. This typically means the `.ltorg` directive is placed after an unconditional branch or function return instruction.

14.5.2 Branches and interworking

When using Thumb, the system will typically have both ARM and Thumb functions. Even if you compile your application for Thumb, you might still have to think about such things as libraries and prebuilt binaries. The core must know which instruction set is to be used for the code being executed after a branch, procedure call or return. This interworking between instruction sets is described in *Interworking* on page 4-11. When writing C code, the linker takes care of this for us, but a little more care is required when porting assembly code.

The target instruction set state is determined in different ways depending on the type of branch. We can consider a number of different instructions:

Function return

Legacy code might use the `MOV PC, LR` instruction. This is unsafe for systems that contain a mix of ARM and Thumb code and must be replaced by `BX LR` for code running on all later architectures.

Function return from the stack

This is done using the `LDMFD SP!, {registers, pc}` instruction that will continue to work correctly in the ARMv7-A architecture, although a newer, equivalent form, `POP {<registers>, pc}` is also available. This is used when registers that must be preserved by the function are PUSHed at the start of the function.

Branch A simple `B` instruction will work in the same fashion on all ARM architectures. If ARM and Thumb instructions are mixed in a single source file (this is unusual), there is no automatic instruction set switch for local symbols. The assembler might introduce a *veneer* depending on whether it knows that the destination is in a different instruction set and is definitely a code symbol (such as a `.type` <symbol>, %function or `.thumb_func`). Because a symbol appears in a code section it is not assumed to be a code symbol unless specifically tagged in this way. If the label is in a different file, the linker will take care of any necessary instruction set change. Similar considerations apply for a function call (`BL`).

Note

Veneers are small pieces of code that are automatically inserted by the linker when it detects that a branch target is out of range or is a conditional branch to code in the other state, for example, from Thumb to ARM or ARM to Thumb. The veneer becomes an intermediate target of the original branch with the veneer itself then being a branch to the target address. Often these veneers can be inlined. The linker can reuse a veneer generated for a previous call, for other calls to the same function if it is in range from both calls. Occasionally, such veneers can be a performance factor. If you have a loop that calls multiple functions through veneers, you will get many pipeline flushes and therefore sub-optimal performance. Placing related code together in memory can avoid this. The ARM linker can be made to export information on this by specifying the `--info veneers` option.

PC modification

Care might be required with other instructions that modify the PC and produce a branch effect. For example, `MOV PC, register` must be replaced with `BX` register in systems that contain both ARM and Thumb code.

Function call to register address

If code contains a sequence like `MOV LR, PC` followed by `MOV PC, register`, this will not work in a system that has both ARM and Thumb code. You must replace it with the single instruction `BLX <register>`.

When a destination or return address is variable or calculated at run-time, take care to appropriately set the Thumb bit (bit [0]) in the address correctly and to use the correct type of branch, to make sure that the call (and return, if applicable) switches instruction set appropriately.

If an external label or function defined in another object is referenced, the linker will produce an address with the Thumb bit (bit [0]) set appropriately. However, if you reference a symbol internal to the object, things are more complicated. For C functions, or code tagged as Thumb, bit [0] will be set appropriately, but it will not be set appropriately for other symbols. In particular, GNU Assembler local labels will not have the Thumb bit set appropriately, nor will the GNU current assembly location symbol ".".

Therefore, when coding in assembler, if an address will be passed to any other function or object, for example, as a return address, method address or callback, you must handle the Thumb bit setting yourself, setting bit [0] of the address where required.

14.5.3 Operand combinations

Thumb and ARM Assembly code have different restrictions on instruction operands. You might find that existing ARM code can produce assembler errors when targeted for Thumb.

Branch out of range errors occur when the distance between the current instruction and the branch target is too large to be encoded in a Thumb instruction. To resolve this, it might be necessary to use a different type of branch, move code sections or to use two separate branches, a so-called *trampoline*.

Similarly, *index out of range* errors might be produced on load and store operations, and to resolve these it might be necessary to manually add part (or all) of the required index offset to the base register in a separate explicit instruction.

Generally, use of SP is limited to stack operations, other forms of use might not be permitted in Thumb code. This means that `PUSH`, `POP`, `LDMFD SP!`, `STMFD SP!`, `ADD`, `SUB` or `MOV` instructions that use the SP are permitted, but other operations should be treated as possible problems. Similarly, operations that directly operate on the PC must be checked (other than the usual function or exception return operations, or literal pool loads).

14.5.4 Other ARM/Thumb differences

There are a number of other differences that can require attention by the assembly language programmer.

- The `RSC` instruction is not available in Thumb. Therefore, code that uses `RSC` must be re-coded using `RSB` or `SBC`, or be built in ARM state.
- Most ARM instructions can optionally be made conditional. This is not the case in Thumb, other than for branches. Instead, small instruction sequences can be executed conditionally by preceding them with the `IT` instruction. For compatibility with both ARM and Thumb, the IT block construct is always understood when using unified assembler syntax. Manually modifying code to use `IT` instructions can be tedious. Fortunately, the assembler command-line option `-mimplicit-it=<when>`, where `<when>` can be one of `never`, `arm`, `thumb` or `always`. In this case you don't have to add `IT` instructions, the assembler will work out the right thing to do for the given target. When assembling for Thumb, however, it is sensible to use `-mimplicit-it=thumb`.

Chapter 15
Application Binary Interfaces

The C compiler is able to generate code from many separately compiled modules. For programs to execute successfully these modules must be able to interoperate, with the operating system code and any code that is written in assembler or any other compiled language. For that reason, we must define a set of conventions to govern inter-operability between separate pieces of code.

The *Application Binary Interface* (ABI) for the ARM architecture specification describes a set of rules that an ARM executable must adhere to in order to execute in a specific environment. It specifies conventions for executables, including file formats and ensures that objects from different compilers or assemblers can be linked together successfully. There are variants of the ABI for specific purposes, for example, the Linux ABI for the ARM architecture or the Embedded ABI (EABI).

The *ARM Architecture Procedure Call Standard* (AAPCS) is part of the ABI (although the ABI actually calls it the *Procedure Call Standard for the ARM Architecture*). It specifies conventions for register and stack usage by the compiler and during subroutine calls. Knowledge of this is vital for inter-working C and assembly code and can be useful for writing optimal code. The AAPCS supersedes the previous *ARM-Thumb Procedure Call Standard* (ATPCS).

The AAPCS specifies rules that must be adhered to by callers to enable a callee function to run and what callee routines must do in order to ensure that callers can continue function correctly when the callee returns. It describes the way that data is laid out in memory and how the stack is laid out, plus permitted variations for processor extensions. It defines how code that has been separately compiled or assembled works together.

15.1 Procedure Call Standard

As we have seen, there are sixteen 32-bit integer registers available in the core. These are labeled R0 - R15. Table 15-1 shows the role assigned to registers within the procedure call standard.

Table 15-1 APCS registers

Register	PCS name	PCS role
R0	a1	argument 1/scratch register/result
R1	a2	argument 2/scratch register/result
R2	a3	argument 3/scratch register/result
R3	a4	argument 4/scratch register/result
R4	v1	register variable
R5	v2	register variable
R6	v3	register variable
R7	v4	register variable
R8	v5	register variable
R9	tr/sb/v6	static base/ register variable
R10	v7	register variable
R11	v8	register variable
R12	IP	scratch register/new -sb in inter-link-unit calls
R13	SP	SP always points at the top of the stack[a]
R14	LR	link register/scratch register
R15	PC	program counter

a. The ARM instruction set provides instructions which can facilitate the implementation of different types of stacks but the ABI only uses full descending stacks.

For the purposes of function calls, the registers are divided into three groups:

- *Argument* registers R0-R3 (a1-a4). These can be used as scratch registers or as caller-saved register variables that can hold intermediate values within a routine, between calls to other functions.
- Callee-saved registers, normally used as register variables. Typically, the registers R4-R8, R10 and R11 (v1-v5, v7 and v8) are used for this purpose.
- Registers that have a dedicated role.

  The function of the program counter, link register and stack pointer ought to be clear. If not, read *Registers* on page 3-6.

  The IP (R12) register can be used by the linker, as a scratch register between a routine and any subroutine it calls, or as an additional local variable within a function. Because the `BL` instructions cannot address the full 32-bit address space, the linker might have to insert a

veneer between the caller and callee. Veneers can also be used for ARM-Thumb inter-working or dynamic linking. Veneers are permitted to modify the contents of IP (R12).

Register R9 has a role that is specific to a particular environment. It can be used as the static base register (SB) to point to position-independent data, or as the thread register (TR) where thread-local storage is used. In code that has no requirement for such a special register, it can be used as an extra callee-saved variable register, v6.

The first four word-sized parameters passed to a function will be transferred in registers R0-R3. Sub-word sized arguments, for example, `char`, will still use a whole register. Arguments larger than a word will be passed in multiple registers. If more arguments are passed, the fifth and subsequent words will be passed on the stack. Passing arguments on the stack always requires additional instructions and memory accesses and therefore reduces performance. For optimal code, you must always try to limit arguments to four words or fewer. If this is not possible, the most commonly used parameters must be defined in the first four positions of the function definition. If the arguments are part of a structure then it might be more efficient to pass a pointer to the structure instead. C++ uses the first argument to pass the `this` pointer to member functions, so only three arguments can be passed in registers.

There are additional rules about 64-bit types. 64-bit types must always be 8-byte aligned in memory. Recall that in *Alignment* on page 14-5 we described how there are limitations on use of `LDRD` and `STRD` double-word instructions to unaligned addresses. In addition, 64-bit arguments to functions must be passed in an even + consecutive odd register pair (for example, R0 + R1 or R2 + R3).

If 64-bit arguments are passed on the stack, they must be at an 8-byte aligned location. Again, this is because of restrictions on `LDRD` and `STRD` instructions. If such 64-bit arguments are listed in a sub-optimal fashion, there can be wasted space in registers or on the stack. When considering such issues, it is important to take into account the `this` pointer in R0 present in all non-static C++ member functions.

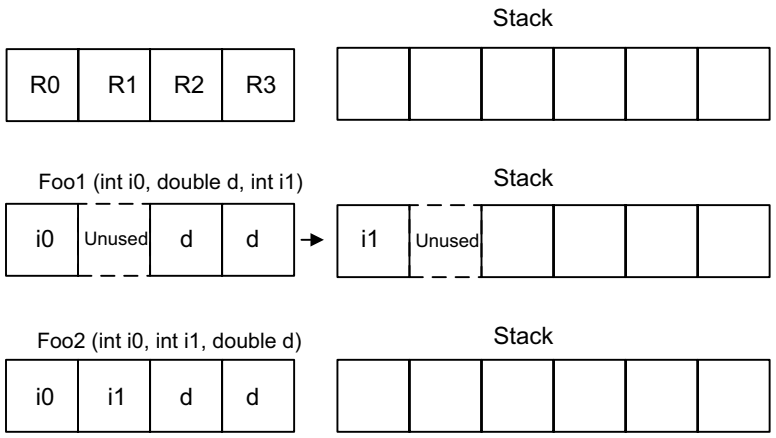


Figure 15-1 Efficient parameter passing

Figure 15-1 shows some examples of how sub-optimal listing of arguments can cause unnecessary spilling of variables to the stack. The figure shows how two different function calls, that pass identical parameters, make use of the registers and stack.

The first function passes an `int`, a `double` and an additional `int`. The first parameter is passed in R0. The second argument is 64-bits and must be passed in an even and consecutive odd register (or in an 8-byte aligned location on the stack). It is therefore passed in R2 and R3. This means that the final argument is passed on the stack.

As the stack pointer must be 8-byte aligned (a constraint imposed by the ABI to simplify the interface at function boundaries), there will be an additional unused word pushed and popped. In the second function call, you can pass the two `int` values in registers R0 and R1 and the double value in R2 and R3. This means that no values are spilled to the stack, which gives more efficient code, requiring both fewer instructions and fewer memory accesses.

The registers R4-R11 (v1-v8) are used to hold the values of the local variables of a subroutine. A subroutine is required to preserve (on the stack), the contents of the registers R4-R8, R10, R11 and SP (and R9 in PCS variants that designate R9 as v6), if they are used by that subroutine.

A caller function will have code like this:

```
@ may need to preserve r0-r3
@ does not need to preserve r4-r11
  BL Func
```

while the callee function will have code like this:

```
Func:

@ Must preserve r4-r11, lr (if used)
@ May corrupt r0-r3, r12
  PUSH {r4-r11, lr}
  ...
  ...
  ...
  POP {r4-r11, pc}
@ Returns value in r0 - char, short or int
@ Returns value in r0 & r1 - double
```

The `PUSH`/`POP` instructions must maintain 8-byte stack alignment, and so use an even number of registers. Leaf functions do not have to do so. The example callee code as shown pushes or pops R4-R11 and LR and PC, which would not preserve an 8-byte aligned stack. It is shown like this to indicate which registers must be saved. In practice, the compiler will normally push an extra register, depending on whether the function is leaf and which registers are modified by the function. Actual instructions will usually be one of `PUSH`/`POP` {r4, lr/pc}, `PUSH`/`POP` {r4-r8, lr/pc}, `PUSH`/`POP` {r4-r10, lr/pc} or `PUSH`/`POP` {r4-r12, lr/pc}. In each case, you would `PUSH` lr and `POP` pc.

15.1.1 VFP and NEON register use

Readers unfamiliar with ARM floating-point might want to refer to Chapter 6 *Floating-Point* before reading this section.

VFPv3 has 32 single-precision registers s0-s31 that can also be accessed in pairs as double-precision registers d0-d15. There are an additional 16 double-precision registers, d16-d31. NEON can also view these as quadword registers q0-q15. Registers s16-s31 (d8-d15, q4-q7) must be preserved across subroutine calls; registers s0-s15 (d0-d7, q0-q3) do not have to be preserved (and can be used for passing arguments or returning results in standard procedure-call variants). Registers d16-d31 (q8-q15), do not have to be preserved.

The Procedure Call Standard specifies two ways in which floating-point parameters can be passed. For software floating-point, they will be passed using ARM registers R0-R3 and on the stack, if required. An alternative, where floating-point hardware exists in the core, is to pass parameters in the VFP or NEON registers.

This hardware floating-point variant behaves in the following way:

- Integer arguments are treated in exactly the same way as in softfp. So, if you consider the function *f* below, you will see that the 32-bit value `a` will be passed to the function in R0, and because the value `b` must be passed in an even or odd register pair, it will go into R2 or R3, leaving R1 unused.

  ```
  void f(uint32_t a, uint64_t b)
      r0: a
      r1: unused
      r2: b[31:0]
      r3: b[63:32]
  ```

- FP arguments fill d0-d7 (or s0-s15), independently of any integer arguments. This means that integer arguments can flow onto the stack and FP arguments will still be slotted into FP registers (if there are enough available).

- FP arguments are able to back-fill, so it's less common to get the unused slots that we see in integer arguments. Consider the following examples:

  ```
  void f(float a, double b)
      d0:
        s0: a
        s1: unused
      d1: b
  ```

  Here, `b` is aligned automatically by being assigned to `d1` (that occupies the same physical registers as VFP s2 or VFP s3).

  ```
  void f( float a, double b, float c)
      d0:
        s0: a
        s1: c
      d1: b
  ```

  In this example, the compiler is able to place `c` into `s1`; it does not have to be placed into `s4`.

  In practice, this is implemented (and described) by using separate counters for `s`, `d` and `q` arguments, and the counters always point at the next available slot for that size. In the second FP example, `a` is allocated first because it is first in the list, and it goes into first available `s` register, which is `s0`. Next, `b` is allocated into the first available `d` register, which is `d1` because `a` is using part of `d0`. When `c` is allocated, the first available `s` register is `s1`. A subsequent double or single argument would go in `d2` or `s4`, respectively.

- There is an additional case when filling FP registers for arguments. When an argument must be spilled to the stack, no back-filling can occur, and stack slots are allocated in exactly the same way for additional parameters as they are for integer arguments.

  ```
  void f( double a, double b, double c, double d,
          double e, double f, float g, double h,
          double i, float j)
      d0: a
      d1: b
      d2: c
      d3: d
      d4: e
      d5: f
      d6:
        s12: g
        s13: unused
      d7: h
      *sp:    i
      *sp+8:  j
      *sp+12: unused (4 bytes of padding for 8-byte sp alignment)
  ```

  Arguments `a-f` are allocated to `d0-d5` as expected.

The single-precision `g` is allocated to `s12`, and h goes to `d7`.

The next argument, `i`, cannot fit in registers, so it is stored on the stack. It would be interleaved with stacked integer arguments if there were any. However, while `s13` is still unused, `j` must go on the stack because we cannot back-fill to registers when FP arguments have hit the stack.

- Double-precision and quad-precision registers can also be used to hold vector data. This would not occur in typical C code.
- No VFP registers are used for variadic procedures, that is, a procedure that does not have a fixed number of arguments. They are instead treated as in `softfp`, in that they are passed in integer registers (or the stack). Single-precision variadic arguments are converted to doubles, as in `softfp`.

15.1.2 Linkage

If the platform has hardware support for NEON technology or a *Floating Point Unit* (FPU), the highest performance is achieved by passing NEON and FPU parameters and return values in NEON and FPU registers. This is called *hardware floating-point linkage*. In some situations, using the general-purpose registers for parameter passing might be preferred, to simplify software compatibility between platforms with and without hardware floating-point support. This is called *software floating-point linkage*.

Floating-point arithmetic might use either hardware coprocessor instructions, or library functions. Floating-point linkage, however, is concerned with passing arguments between functions that use floating-point variables.

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. Hardware floating-point linkage uses the VFP coprocessor registers to pass the arguments and return value. The benefit of using software floating-point linkage is that the code can be run on a core with or without a VFP coprocessor and is therefore more portable. The benefit of using hardware floating-point linkage is that it is more efficient than software floating-point linkage, but only runs on systems that have a VFP coprocessor.

You cannot mix objects with different floating-point linkage in a single image. Any dynamic libraries loaded while the application is running must also use the same linkage.

Any system that supports both NEON and VFP instructions uses a common register bank for these instructions, therefore configuration options that affect the floating-point calling convention also affect how NEON parameters are passed and returned.

15.1.3 Stack and heap

The stack implementation is full-descending, with the current top of the stack pointed to by R13 (SP). The stack must be contiguous in the virtual memory space. Detection of a stack overflow is usually handled with memory management. The stack must always be aligned to a word boundary, except at an external interface, when it must be double-word aligned.

The heap is an area (or areas) of memory that are managed by the process itself, for example, with the C `malloc()` function. It is typically used for the creation of dynamic data objects.

15.1.4 Returning results

A function that returns a `char`, `short`, `int` or single-precision `float` value will do so using R0. A function that returns a 64-bit value (a double-precision `float` or `long long`) does so in R0 and R1. As mentioned in *VFP and NEON register use* on page 15-4, floating-point and NEON return

values will be in `s0`, `d0`, or `q0` when using hardware linkage. If the software uses hardware linkage it will return floating-point values in `s0`, `d0`, or `q0`. If the software uses software linkage, it will return single-precision float in r0.

15.2 Mixing C and assembly code

One example of why it can be useful to understand the AAPCS is to write assembly code that is compatible with C code. One way to do this is write separate modules and assemble them with GNU Assembler. They can be defined as `extern` functions in C and called. Provided the AAPCS rules are followed, there should be no problem.

We can also insert assembly code into our C code, through the GCC `asm` statement. This is very simple to use. For example, we can implement a `NOP` as shown in Example 15-1.

Example 15-1 NOP

```
asm("nop");
```

The time taken to carry out a `NOP` is undefined.

In fact, it is likely that this `NOP` will have no effect, because the C compiler might optimize it away, or the core might discard the instruction. When you include assembly language code with inline assembler, the resulting code is still subject to optimization by the C compiler. This is a very important point that must be taken into account whenever inline assembly is used. Even if the compiler does not optimize-away the NOP instruction, the core itself can filter out the NOP from the instruction stream so that it never reaches the execute stage.

Inline assembly code has a different syntax to regular assembly code. Registers and constants have to be specified differently, if they refer to C expressions.

A slightly more complicated example takes the value of an `int` and uses the `USAD8` assembly instruction to calculate the sum of difference of bytes and then store the result in a different `int`. Example 15-2 shows the relevant code.

Example 15-2 Using the USAD8 instruction

```
asm volatile (“usad8 %0, %1, %2” : “=r” (result): “r”(value1), “r”(value2));
```

The general format of such inline assembly code is:

```
asm volatile (assembler instructions : output operands (optional) : input operands
(optional) : clobbered registers (optional) );
```

The colons divide the statement up into parts. The first part “`usad8 %0, %1, %2`” is the actual assembly instruction. The second part is an (optional) list of output values from the sequence. If more than one output is required, commas are used to separate the entries. You might then optionally have a list of input values for the sequence, with the same format as the output. If you don’t specify an output operand for an assembly sequence, it is quite likely that the C compiler optimizer will decide that it is not serving any useful purpose and optimize it away! A way to avoid this is to use the `volatile` attribute that tells GCC not to optimize the sequence.

In the actual assembly language statement, operands are referenced by a percent sign followed by the symbolic name in square brackets. The symbolic name references the item with the same name in either the input or output operand list. This name is completely distinct from any other symbol within your C code (although clearly it is less confusing to use symbols that do have a meaning within your code). Alternatively, the name can be omitted and the operand can be specified using a percent sign followed by a digit indicating the position of the operand in the list (that is, `%0`, `%1` … `%9`), as shown in the example.

There is an optional fourth part to an asm statement, called the *clobber* list. This enables you to specify to the compiler what will be changed by the assembly code. We can specify registers (for example, R0), the condition code flags (cc) or memory.

This makes the compiler store affected values before and reload them after executing the instruction.

The constraints mentioned when we talked about input and output operand lists relate to the fact that assembly language instructions have specific operand type requirements. When passing parameters to inline assembly statements, the compiler must know how they are represented. For example, the constraint "r" specifies one of the registers R0-R15 in ARM state, while "m" is a memory address and "w" is a single precision floating-point register. These characters have an = placed before them to indicate a write-only output operand, a "+" for a read/write output operand (that is, one that is both input and output to the instruction). The "&" modifier instructs the compiler not to select any register for the output value that is used for any of the input operands.

You can force the inline assembler to use a particular register to hold a local variable by using something like the code shown in Example 15-3.

Example 15-3 Inline assembler local variable usage

```
void func (void) {
register unsigned int regzero asm("r0");
```

and later

```
asm volatile("rev r0, r0");
```

This usage can interfere with the compiler optimization and does not guarantee that the register will not be re-used, for example when the local variable is no longer referenced. Hard coding register usage is always bad practice. It is almost always better to use local variables instead.

Example 15-4 gives a longer example of inline assembler, taken from the Linux kernel. It shows how a series of inline assembly language instructions can be used. The code manipulates the CPSR, to change modes. This would not be possible using C code.

Example 15-4 Inline assembler

```
void __naked get_fiq_regs(struct pt_regs *regs)
{
  register unsigned long tmp;
  asm volatile (
  "mov   ip, sp\n\
  stmfd  sp!, {fp, ip, lr, pc}\n\
  sub    fp, ip, #4\n\
  mrs    %0, cpsr\n\
  msr    cpsr_c, %2           @ select FIQ mode\n\
  mov    r0, r0\n\
  stmia  %1, {r8 - r14}\n\
  msr    cpsr_c, %0           @ return to SVC mode\n\
  mov    r0, r0\n\
  ldmfd  sp, {fp, sp, pc}"
  : "=&r" (tmp)
  : "r" (&regs->ARM_r8), "I" (PSR_I_BIT | PSR_F_BIT | FIQ_MODE));
}
```

The ARM compiler tools have a similar concept, albeit with different syntax. In addition to inline assembly, they also support *embedded* assembly that is assembled separately from the C code and produces a compiled object that is combined with the object from the C compilation.

Chapter 16
Profiling

Donald Knuth once famously observed that “Premature optimization is the root of all evil”. However, Knuth's comment does not argue against *appropriate* optimization.

Code optimization is an important part of the work of software engineers– modifying software so that it runs more quickly, uses less power and makes less use of memory or other resources. To do this, you must first identify which part or parts of the code should be optimized. It is worth noting, however, that a programmer’s best guesses about where the time is spent can be notoriously inaccurate, hence the importance of 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. We can distinguish two basic approaches to gathering information.

Time based sampling

Here, 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

Here, 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 use of annotation options to specify which events are to be recorded, which events shown, or thresholds to avoid displaying large numbers of functions with low count numbers.

16.1 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 Example 16-1 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.

Example 16-1 Example flat profile

```
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
```

It might be useful at this point to consider some example profiling tools which can be used in Cortex-A series processors.

16.1.1 Gprof

GProf is a GNU tool that provides an easy way to profile your C/C++ application and find the locations that require work.

Using GCC, you can generate profile information by compiling with a special flags. The source code has to be compiled with the `-pg` option. For line-by-line profiling, the `-g` option would also be required. You then execute the compiled program and the profiling data is collected. You then run `gprof` on the resulting statistics file to view the data, in a variety of convenient formats.

When you compile with `-pg` the compiler adds profiling instrumentation that collects data at function entry and exits at runtime. It therefore profiles only the user application code and not anything that happens in code that has not been built for profiling (for example, libc) or the kernel. Gprof can give misleading results if the performance limitations of the code come from kernel or I/O issues (memory fragmentation or file accesses for example).

It might be necessary to remove GCC optimization flags, as some compiler optimizations can cause problems while profiling. It is also the case that the use of the profiling flags will actually slow the program down. This can be an important consideration in some types of real-time system, where it might be that the interaction of real-time events has a significant effect on the performance of the profiled code. A binary file called `gmon.out` containing profiling information is generated. This file can then be operated on by the `gprof` tool.

16.1.2 OProfile

OProfile is a whole system profiling tool that runs on Linux and includes the kernel in its metrics. Unlike `gprof`, it works using a statistical sampling method. OProfile can examine the system at regular intervals, determine what code is running, and update appropriate counters. If a long enough profile is taken, with a sufficient sample rate, an accurate picture of the execution is obtained. Like other profilers that make use of interrupts, code that disables interrupts can cause inaccuracies. For this reason, the Linux function `spinlock_irq_restore()` that re-enables interrupts after a spinlock has been relinquished can erroneously appear to be a major system bottleneck, as the time for which interrupts were disabled can be counted against it. OProfile can also be made to trigger on hardware events and will record all system activity including kernel and library code execution.

OProfile does not require code to be recompiled with any special flags (provided symbol information is available). It provides useful hardware information about such things as clock cycles and cache misses. Call graphs are statistically generated, so might not be completely accurate.

16.1.3 DS-5 Streamline

DS-5 Streamline is a graphical performance analysis tool that can be used to analyze the performance of an Linux or Android system. It is a component of ARM DS-5 and combines a kernel driver, target daemon and Eclipse-based user interface. (For more information about DS-5, see *ARM DS-5* on page B-10.)

DS-5 Streamline takes sampling data and system trace information and produces reports that present the data visually and in a statistical form. It uses hardware performance counters and kernel metrics to provide an accurate representation of system resources. DS-5 Streamline enables the application source code to be annotated to augment its graphical display with additional textual or visual information, such as labelled timing markers or screen shots from the target.

For example, DS-5 Streamline has a display mode called Core map. In this mode the process trace view changes from an intensity map of time to a view that highlights core affinity. Figure 16-1 on page 16-5 shows the DS-5 Streamline view of a multi-threaded Linux application running on a multi-core cluster. Threads are being assigned to the cores by the Linux kernel:

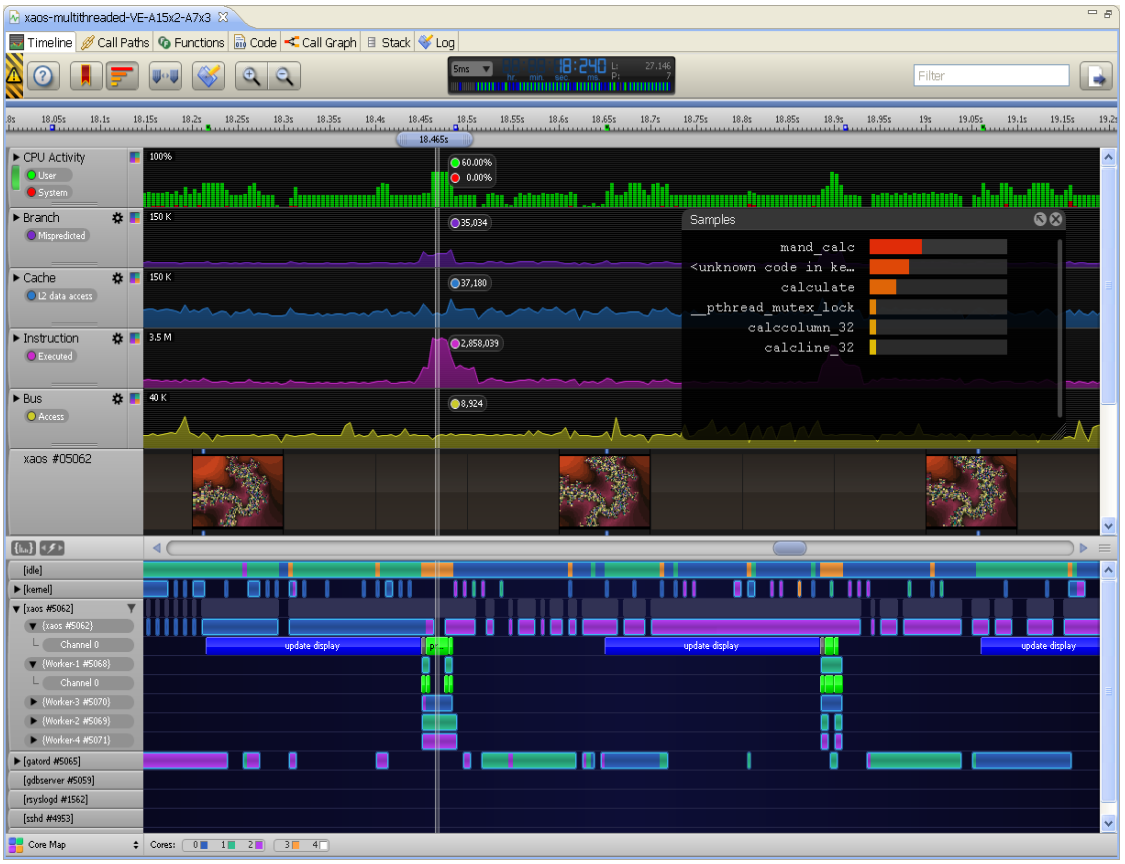


Figure 16-1 Core map in ARM DS-5 Streamline

16.1.4 ARM performance monitor

The performance monitor hardware is able to count several events, using multiple counters. Normally, you combine together multiple values to generate useful parameters to optimize. For example, you can choose to count the total number of clock cycles and the number of instructions executed and use this to derive a cycles per instruction figure that is a useful proxy for the efficiency with which the core is operating. We can generate information about cache hit or miss rates (separately for both L1 data and instruction caches) and examine how code changes can affect these.

Cortex-A series processors contain event counting hardware that can be used to profile and benchmark code, including generation of cycle and instruction count figures and to derive figures for cache misses and so forth. The performance counter block contains a cycle counter that can count core cycles, or be configured to count every 64 cycles. There are also a number of configurable 32-bit wide event counters that can be set to count instances of events from a wide-ranging list (for example, instructions executed, or MMU TLB misses). These counters can be accessed through debug tools, or by software running on the core, through the CP15 *Performance Monitoring Unit* (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.

It is important to understand that information generated by such counters might not be exact. In a superscalar, out-of-order core, 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 standard countable events, common to all ARMv7-A processors are listed in Table 16-1. The TRM for the specific processor being used provides more information on the lists of events that can be monitored, that can include a large number of additional possibilities in addition to those listed here.

Table 16-1 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
0x02	Instruction fetch that causes a Level 1 instruction TLB refill
0x03	Data Read or Write operation that causes a Level 1 instruction TLB refill
0x04	Data Read or Write operation that causes a Level 1 data cache access
0x05	Data Read or Write operation that causes a Level 1 data TLB refill
0x06	Memory-reading instruction executed
0x07	Memory-writing instruction 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
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
0x13	Data memory access
0x14	Level 1 instruction cache access
0x15	Level 1 data cache write-back
0x16	Level 1 data cache write-back
0x17	Level 2 data cache refill
0x18	Level 2 data cache write-back
0x19	Bus access
0x1A	Local memory error
0x1B	Instruction speculatively executed
0x1C	Instruction write to TTBR
0x1D	Bus cycle
0x1E-0x3F	Reserved

Multi-core clusters include a significant number of additional events relating to SMP capabilities, described in Chapter 18 *Multi-core processors*. These include numbers of barrier instructions executed, measures of coherent cache activity and counts of exclusive access failures.

In Linux, these counters are normally accessed through the kernel using the Linux OProfile tool or the Linux perf events framework. However, User mode access to these counters can be enabled for direct access if required.

16.1.5 Linux perf events

Linux contains patches (as of Linux-2.6.34) to support the Ingo Molnar Perf Events framework. It handles multiple events, including core cycles and cache-misses but also measures kernel events like context switches. It provides simple, platform independent access to the ARM performance counter registers and also to software events. Through this framework, tools can produce graphs and statistics about such things as function calls. You can also record execution traces, profile on a per-core, per-application and per-thread basis and generate summaries of events.

16.1.6 Ftrace

Ftrace is an increasingly widely used trace tool on Linux. It provides visualization of the flow within the kernel by tracing each function call. It enables interactions between parts of the kernel to be viewed and enables to developers to investigate potential problem paths where interrupts or pre-emption are disabled for long periods.

16.1.7 Valgrind and Cachegrind

Valgrind is a widely used tool, commonly used for detection of memory leaks and other memory handling problems; however, it can also be used for profiling memory usage and is potentially able to give much more detailed results than OProfile. Valgrind translates a program into a core-neutral intermediate representation. Other tools associated with Valgrind are able to operate on this and then Valgrind translates the code back into native machine code. This process makes the code run an order of magnitude slower, but enables checks for accesses to undefined memory, off-by-one errors and so forth to be performed by tools like `Memcheck`.

For memory access optimization, the Cachegrind tool can be used. Because Valgrind simulates the execution of the program, you can use Cachegrind to record all uses of program memory. By simulating the operation of the core caches, you can generate statistics about cache usage. Some care is required – this is a simulation of the cache and it is possible that it does not represent the real cache hardware completely accurately. Nevertheless, it can be a very useful tool to examine cache and memory usage by an application. When writing high level applications, it can be difficult to have much appreciation for (or control over) addresses used by a program. The linker will typically generate many of the virtual addresses used within an image, while the runtime loader will control positioning of libraries and so forth. Finally, the kernel is responsible for placement of code and data in physical memory. Memory profiling tools can therefore be a useful aid.

Chapter 17
Optimizing Code to Run on ARM Processors

Optimization does not necessarily mean optimizing to make programs faster. In embedded systems, you might prefer to optimize for battery life, code density or memory footprint. Writing code that is more efficient delivers not only higher levels of performance, but can also be crucial in conserving battery life. If you can get a job done faster, in fewer cycles, you can turn off the power for longer periods.

Many compilers provide options to help with this. For example, both the ARM Compiler and the GNU GCC compiler have -0 flags to specify the level of optimization.

Although cycle timing information can be found in the *Technical Reference Manual* (TRM) for the processor that you are using, it is very difficult to work out how many cycles even a trivial piece of code will take to execute. The movement of instructions through the pipeline is dependent on the progress of the surrounding instructions and can be significantly affected by memory system activity. Pending loads or instruction fetches that miss in the cache can stall code for tens of cycles. Standard data processing instructions (logical and arithmetic) will take only one or two cycles to execute, but this does not give the full picture. Instead, you must use profiling tools, or the system performance monitor built-in to the core, to extract useful information about performance.

17.1 Compiler optimizations

The ARM Compiler and GNU GCC give you a wide range of options that aim 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 in order to produce the best image file.

17.1.1 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.

17.1.2 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.

Example 17-1 illustrates how this works:

Example 17-1 Common sub-expression

```
i = a * b + c;
j = a * b * d;
```

The compiler can treat this code as if it had been written as in Example 17-2. It must be noted though, that it can only do this if neither `a` nor `b` is volatile.

Example 17-2 Common sub-expression elimination

```
tmp = a * b;
i = tmp + c;
j = tmp * d;
```

This reduces both the instruction count and cycle count.

17.1.3 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. Furthermore, there is a branch instruction to iterate over the loop, that can take a number of cycles to execute. We can avoid this penalty by unrolling loops, partially or fully.

Consider the simple code shown in Example 17-4, to initialize an array.

Example 17-3 Initializing an array

```
for (i = 0; i < 10; i++)
{
   x[i] = i;
}
```

Each iteration of the loop contains an assembler sequence of the form in Example 17-4.

Example 17-4 Loop termination assembly code

```
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 in Example 17-5.

Example 17-5 Unrolled loop

```
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, we 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-A 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. We 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.

17.1.4 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 `-O`*n*, 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. Furthermore, this option enables `-ftree-vectorize` - causing the compiler to attempt to automatically generate NEON code from standard C or C++. See Chapter 20 *Writing NEON Code*.
- `-funroll-loops`. This option is independent of the `-O`*n* 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 the above 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-a8`. This selects instructions for the architecture ARMv5TE but tunes the selected instructions for execution on a Cortex-A8 processor.

```
-mfpu=<fpu>
```

If your target platform supports hardware floating-point or NEON, specify this to ensure that the compiler can make use of these instructions. For a Cortex-A5 target, you would specify `-mfpu=neon-vfpv4`.

```
-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 17-1 shows a few examples of code generation for floating-point operations.

Table 17-1 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
neon	hard	VFPv3 and Advanced SIMD code, where the floating-point and SIMD types use the hardware FP registers

17.1.5 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.

17.2 ARM memory system optimization

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 you can do the job with less access to external memory, you can save power by keeping everything on-chip. Furthermore, by accessing the external memory less frequently, you improve the performance of the system, enabling software to run faster, or the processor to be clocked more slowly or for shorter periods, to save power.

17.2.1 Data cache optimization

In most Cortex-A series processors, there is a significant gap in performance between memory accesses that hit in the cache and those that do not. Cache misses can take tens of cycles to resolve. Cache hits return data within a few cycles and the compiler can often schedule instructions in a way that hides latency. For most algorithms, therefore, ensuring that cache misses are minimized is the most important possible optimization. The most important improvements are those that affect the level 1 cache.

Consider the problem of data cache misses. Optimization is particularly significant for pieces of code that use a dataset larger than the available cache size. It is important for you to understand the arrangement of data in memory and how that corresponds to data cache accesses. Code must 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, which gives best performance.

Several techniques to improve this locality can be considered.

17.2.2 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. Example 17-6 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.

Example 17-6 Matrix vector product code

```
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) will be 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* on page 17-9. Therefore, elements (0,0), (0,1), (0,2) etc. 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 Example 17-7 you can make a big improvement. Similarly, you can unroll the outer loop two (or four, or eight) times as well.

Example 17-7 Code using tiles

```
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* on page 17-12.

Figure 17-1 illustrates the changing cache access pattern that results from changes to the C code.

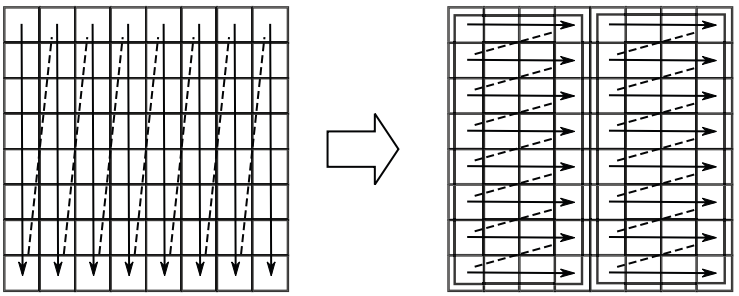


Figure 17-1 Effect of tiling on cache usage

17.2.3 Loop interchange

In many programs, there will be nested loops, a very 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 will be 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.

17.2.4 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.

As we saw in Chapter 14 *Porting*, 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.

17.2.5 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.

17.2.6 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-A 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 (and OProfile) 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.

17.2.7 Optimizing L2 and outer cache usage

Everything said about optimizations for using the L1 cache also applies to the L2 cache accesses. Best performance results from having a working dataset that is smaller than the L2 cache size, and where the data is used more than once; there is little benefit caching data that is used only once, other than possibly producing more optimal bus accesses. If the dataset is larger than the cache size, you can consider similar techniques to those already described for the L1 cache. There is, however, an additional point to consider with outer caches, which is that they might well be shared with other cores and therefore the effective size for an individual processor can be less than the actual size. In addition, when writing generic code to run on a number of ARM families, it can be difficult to make optimal use of the L2 cache. The presence of such a cache is not guaranteed and its size can vary significantly between systems.

17.2.8 Optimizing TLB usage

In general, the scope for optimizing usage of the Translation Lookaside Buffer (see Chapter 9) is much less than for optimizing cache accesses. The key points are to minimize the number of pages in use (this obviously gives fewer TLB misses) and to use large MMU mappings (supersections or sections in preference to 4KB pages) as this reduces the cost of individual translation table walks (one external memory access rather than two) and also means that a larger amount of memory is represented within an individual TLB entry (also giving fewer TLB misses). In practice, however, an operating system like Linux uses 4KB pages everywhere, so the main optimization technique available is to separate the frequently accessed code and data from the infrequently accessed code and data (for example exception handling code can be moved to a different page) and trying to limit the number of frequently accessed pages to below

the maximum number supported by the processor hardware. The main optimization would be to try to process multiple cache lines' worth of data per page, so that the L1 cache is the limiting factor rather than TLB entries.

17.2.9 Data abort optimization

Chapter 9 *The Memory Management Unit*, in the context of Linux, describes how data aborts will be generated by page faults on the first time that a memory page is accessed and again when the page is first written to. This means that the kernel abort handler is called to take appropriate action and there is a certain performance overhead to this. Simplistically, you can reduce this overhead by using fewer pages. Again, code optimizations that make code smaller will help, as will reducing the size of the data space.

17.2.10 Prefetching a memory block access

ARM Cortex-A series processors contain sophisticated cache systems, and support for speculation and out of order execution that can hide latencies associated with memory accesses. However, accesses to the external memory system are usually sufficiently slow that there will still be some penalty. If you can prefetch instructions or data into the cache before you require them, you can hide this latency.

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. There is also a `PLI` instruction that enables you to hint to the processor that an instruction load from a particular address is likely in the near future. This can cause the processor to preload the instructions to its cache.

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.

17.3 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.

17.3.1 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* on page 17-13).

17.3.2 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 (Example 17-8 and Example 17-9).

Example 17-8 Loop fusion

```
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:

Example 17-9 Fused loops

```
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.

17.3.3 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 will be 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.

Chapter 15 stated that 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.

17.3.4 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 in Example 17-10, for signed halfwords (shorts).

Example 17-10 Addition of 2 signed shorts (assembly code)

```
ADD      R0, R1, R2
SXTH     R0, R0
```

Or for unsigned halfwords as in Example 17-11.

Example 17-11 Addition of 2 unsigned shorts (assembly code)

```
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.

17.3.5 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.

17.3.6 Division and modulo

Not all ARM processors have hardware support for division (See Table 2-3 on page 2-9). For these processors, C division typically calls a library routine that takes tens of cycles to run for divides of 32-bit integers,

Note

Division is slower than multiplication even in hardware. In performance-critical code it is almost always worth replacing it if possible. This must be done as a trade-off against code maintainability.

Where possible, divides must be avoided, or removed from loops. Division with a fixed divisor, that is, one that is known at compile time, is faster than dividing two variable quantities. The compiler can replace a divide by a shift-multiply pair in this case. A 32 × 32 multiply by fixed constant, then shift right to adjust the most significant word.

Modulo arithmetic is another case to be aware of, as this will also use division library routines.

The code

```
minutes = (minutes + 1) % 60;
```

will run significantly faster on machines with no hardware divide, if coded as

```
if (++minutes == 60) minutes=0;
```

that substitutes a two cycle add and compare in place of a call to a library function.

17.3.7 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`.

17.3.8 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.

Chapter 14 *Porting* provides more information about the use of an inline assembler.

17.3.9 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.

17.3.10 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.

17.3.11 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.

Chapter 18
Multi-core processors

Up to this point, we have considered the ARM processor core as a single entity. Most Cortex-A series processors, however, can include up to four processing cores.

Multi-core systems can potentially deliver higher performance, because more processing units (cores) are available. This enables multiple tasks to be executed in parallel, potentially reducing the amount of time required to perform the allocated task.

Multi-processing can be defined as running two or more sequences of instructions simultaneously within a single device containing two or more cores. The concept of multi-processing has been a subject of research for a number of decades, and has seen widespread commercial use over the past 15 years. Multi-processing is now a widely adopted technique in both systems intended for general-purpose application processors and in areas more traditionally defined as embedded systems.

The overall energy consumption of a multi-core system can be significantly lower than that of a system based on a single processor core. Multiple cores enable execution to be completed faster and so some elements of the system might be completely powered down for longer periods. Alternatively, a system with multiple cores might be able to operate at a lower frequency than that required by a single processor to achieve the same throughput. A lower power silicon process or a lower supply voltage can result in lower power consumption and reduced energy usage. Most current systems do not permit the frequency of cores to be changed independently. However, each core can be dynamically clock gated, giving additional power and energy savings.

Multi-core systems also add flexibility and scalability to system designs. A system that contains one or two cores could be scaled up for more performance by adding additional cores, without requiring redesign of the whole system or significant changes to software.

Having multiple cores at our disposal also enables more options for system configuration. For example, you might have a system that uses separate cores, one to handle a hard real-time requirement and another for an application requiring high, uninterrupted performance. These could be consolidated into a single multi-processor system.

A multi-core device is also likely to be more responsive than one with a single core. When interrupts are distributed between cores there will be more than one core available to respond to an interrupt and fewer interrupts per core to be serviced. Multiple cores will also enable an important background process to progress simultaneously with an important but unrelated foreground process.

Multi-core systems can also extract more performance from high latency memory systems (for example, DDR memory) by enabling a memory controller to queue up and optimize requests to the memory. Processors working on coherent data can benefit from reductions in linefills and evictions. When the data is not shared, it is likely that performance will be adversely affected. An L2 cache can mean improved utilization for shared memory regions (including file caches), shared libraries and kernel code. Additionally, if the number of cores is increased, then without a corresponding increase in memory bandwidth performance will also deteriorate.

In the past, much software was written to operate within the context of a single core. Some operating systems provide support for time-slicing. this gives the illusion of multiple processes or tasks running simultaneously. It is important to clearly understand the difference between multi-threading, for example POSIX threads, or Java and multi-processing. A multi-threaded application can be run on a single core, but only with multi-processing can the threads truly execute in parallel.

Migrating multi-threaded software from a single core system to a multi-core one can trigger problems with incorrect programs that could not be exposed by running the same program time-sliced on a single core. It can also cause very infrequent bugs to become very frequently triggered. What it cannot do is to cause correctly written multi-threaded programs to misbehave, only expose previously unnoticed errors.

18.1 Multi-processing ARM systems

From early in the history of the architecture, ARM processors were likely to be implemented in systems that contained other processors. This commonly meant a *heterogeneous* system, perhaps containing an ARM processor plus a separate DSP processor. Such systems have different software executing on different cores and the individual processors can have differing privileges and views of memory. Many widely used ARM systems, such as the TI OMAP series, or the Freescale i.MX, are examples of this.

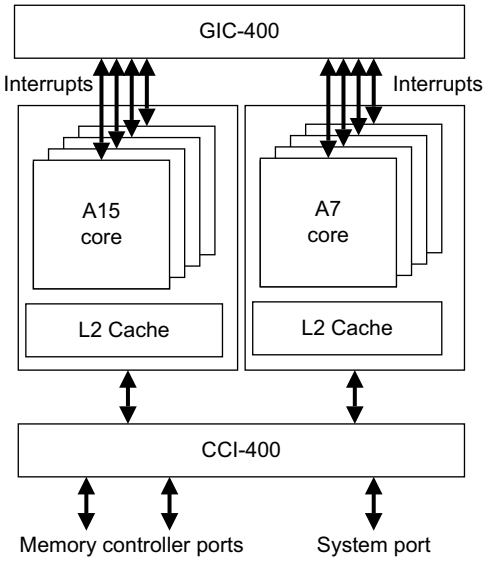


Figure 18-1 Example of a multi-cluster system

We can distinguish between systems that contain:

- A single processor containing a single core, such as the Cortex-A8 processor.
- A multi-core processor, that contains several cores capable of independent instruction execution, that can be externally viewed as a single unit or cluster, either by the system designer or by an operating system that can abstract the underlying resources from the application layer.
- Multiple clusters (such as that shown in Figure 18-1), in which each cluster contains multiple cores.

ARM was among the first companies to introduce multi-core processors to the SoC market, when it introduced the ARM11 MPCore processor in 2004. All the processors described in this book, with the exception of the Cortex-A8 processor, are examples of such multi-core systems.

An ARM multi-core processor can contain between one and four cores. Each core can be individually configured to take part (or not) in a data cache coherency management scheme. A *Snoop Control Unit* (SCU) device inside the processor has the task of automatically maintaining level 1 data cache coherency, between cores within the cluster without software intervention.

ARM multi-core processors include an integrated interrupt controller. Multiple external interrupt sources can be independently configured to target one or more of the individual processor cores. Furthermore, each core is able to signal (or *broadcast*) any interrupt to any other core or set of cores in the system, from software (software triggered interrupts). These mechanisms enable the OS to share and distribute interrupts across all cores and to coordinate activities using the low-overhead signaling mechanisms provided.

Cortex-A MPCore processors also provide hardware mechanisms to accelerate OS kernel operations such as system-wide cache and TLB maintenance operations. (This feature is not found in the ARM11 MPCore.)

Each of the Cortex-A series multi-core processors have the following features:

- Configurable between one and four cores (at design time).
- Level 1 data cache coherency.
- Integrated interrupt controller.
- Local timers and watchdogs.
- An optional Accelerator Coherency Port (ACP).

Figure 18-2 illustrates the structure of the Cortex-A9 MPCore processor, though a generalized form of this description also applies to other multi-core processors. These features are described in more detail in the course of this chapter and those that follow.

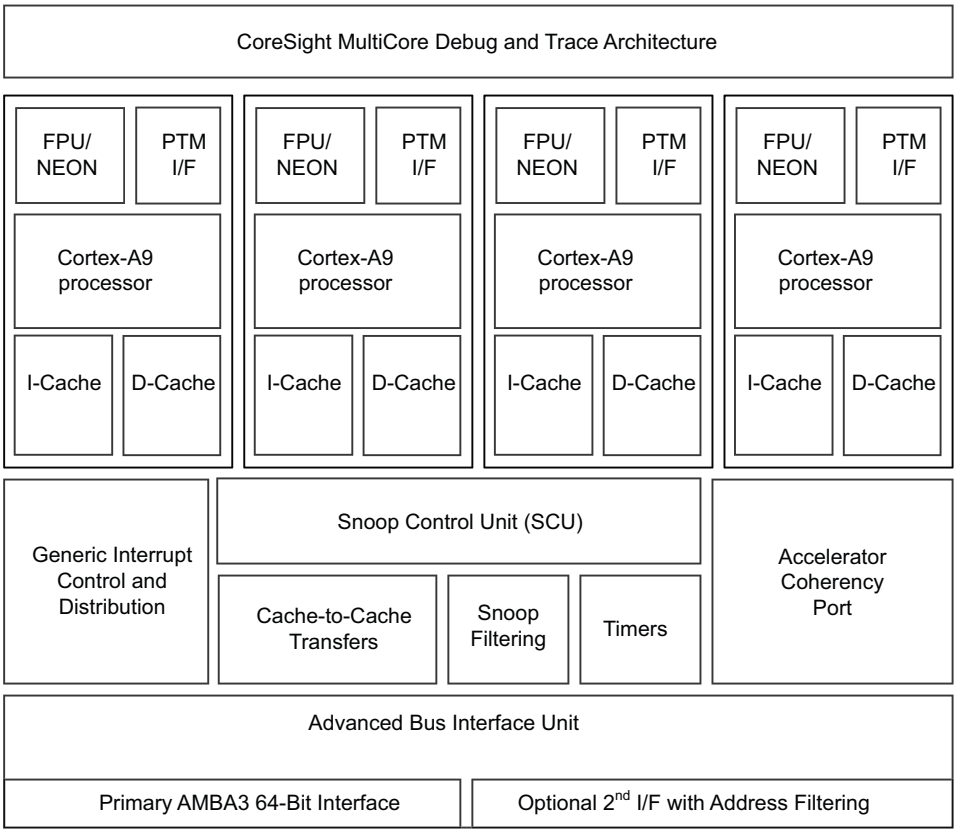


Figure 18-2 Cortex-A9 MPCore processor

18.2 Symmetric multi-processing

Symmetric multi-processing (SMP) is a software architecture that dynamically determines the roles of individual processors. Each core in the cluster has the same view of memory and of shared hardware. Any application, process or task can run on any core and the operating system scheduler can dynamically migrate tasks between cores to achieve optimal system load.

We expect that readers will be familiar with the fundamental operating principles of an OS, but OS terminology will be briefly reviewed here. An application that executes under an operating system is known as a *process*. It performs many operations through calls to the system library that provides certain functions from library code, but also acts a wrapper for system calls to kernel operations. Individual processes have associated resources, including stack, heap and constant data areas, and properties such as scheduling priority settings. The kernel view of a process is called a *task*.

For the purposes of describing SMP operation, we will use the term *kernel* to represent that portion of the operating system that contains exception handlers, device drivers and other resource and process management code. We will also assume the presence of a task scheduler that is typically called using a timer interrupt. The scheduler is responsible for time-slicing the available cycles on cores between multiple tasks, dynamically determining the priority of individual tasks and deciding which task to run next.

Threads are separate tasks executing within the same process space that enable separate parts of the application to execute in parallel on different cores. They also permit one part of an application to keep executing while another part is waiting for a resource.

In general, all threads within a process share a number of global resources (including the same memory map and access to any open file and resource handles). Threads also have their own local resources, including their own stacks and register usage that will be saved and restored by the kernel on a context switch. The fact that these resources are local does not, however, mean that the local resources of any thread are guaranteed to be protected from incorrect accesses by other threads. Threads are scheduled individually and can have different priority levels even within a single process.

An SMP-capable operating system provides an abstracted view of the available core resources to the application. Multiple applications can run concurrently in an SMP system without re-compilation or source code changes. A conventional multitasking OS enables the system to perform several task or activities at the same time, in both single-core and multi-core processors. There is, however, a key distinction. In the single (uniprocessor) case, multitasking is performed through time-slicing of a single core, giving the illusion of many tasks being performed at the same time. In a multi-core system, we can have true concurrency; multiple tasks are actually run at the same time, in parallel, on separate cores. The role of managing the distribution of such tasks across the available cores is performed by the operating system.

Typically, the OS task scheduler can distribute tasks across available cores in the system. This feature, known as load balancing, is aimed at obtaining better performance, or energy savings or even both. For example, with certain types of workloads, energy savings can be achieved if the tasks making up the workload are scheduled on fewer cores. This would allow more resources to be left idling for longer periods, thereby saving energy.

In other cases, the performance of the workload could be increased if the tasks were spread across more cores. These tasks could make faster forward progress, without getting perturbed by each other than if they ran on fewer cores.

In another case, it might be worth running tasks on more cores at reduced frequencies as compared to fewer cores at higher frequencies. Doing this could provide a better trade-off between energy savings and performance.

The scheduler in an SMP system can dynamically re-prioritize tasks. This *dynamic task prioritization* enables other tasks to run while the current task sleeps. In Linux, for example, tasks whose performance is bound by I/O activity can have their priority decreased in favor of tasks whose performance is limited by processor activity. The I/O-bound task will typically schedule I/O activity and then sleep pending such activity.

Interrupt handling can also be load balanced across cores. This can help improve performance or save energy. Balancing interrupts across cores or reserving cores for particular types of interrupts can result in reduced interrupt latency. This might also result in reduced cache use which will help improve performance.

Using fewer cores for handling interrupts could result in more resources idling for longer periods, resulting in an energy saving at the cost of reduced performance. The Linux kernel does not support automatic interrupt load balancing. However, the kernel provides mechanisms to change the binding of interrupts to particular cores. There are open source projects such as irqbalance `https://github.com/Irqbalance/irqbalance` which use these mechanisms to arrange a spread of interrupts across the available cores. irqbalance is made aware of system attributes such as the shared cache hierarchy (which cores have a common cache) and power domain layout (which cores can be powered off independently). It can then determine the best interrupt to core binding.

An SMP system will by definition have shared memory between cores in the cluster. To maintain the required level of abstraction to application software, the hardware must take care of providing a consistent and coherent view of memory for you.

Changes to shared regions of memory must be visible to all cores without any explicit software coherency management. Likewise, any updates to the memory map (for example because of demand paging, allocation of new memory or mapping a device into the current virtual address space) of either the kernel or applications must be consistently presented to all cores.

18.3 Asymmetric multi-processing

In an *Asymmetric Multi-processing* (AMP) system, you can statically assign individual roles to a core within a cluster, so that in effect, you have separate cores, each performing separate jobs, within each cluster. This can be referred to as a function-distribution software architecture and typically means that you have separate operating systems running on the individual cores. The system can appear to you as a single-core system with dedicated accelerators for certain critical system services. In general AMP does not refer to systems in which tasks or interrupts are associated with a particular core.

In an AMP system, each task can have a different view of memory and there is no scope for a core that is highly loaded to pass work to one that is lightly loaded. There is no requirement for hardware cache coherency in such systems, although there will typically be mechanisms for communication between the cores through shared resources, possibly requiring dedicated hardware. The system described in *Cache coherency* on page 18-9 can help reduce the overheads associated with sharing data between the systems.

Reasons for implementing an AMP system using a multi-core processor might include security, a requirement for guaranteeing meeting of real-time deadlines, or because individual cores are dedicated to perform specific tasks.

There are classes of systems that have both SMP and AMP features. This means that we have two or more cores running an SMP operating system, but the system has additional elements that do not operate as part of the SMP system. The SMP sub-system can be regarded as one element within the AMP system. Cache coherency is implemented between the SMP cores, but not necessarily between SMP cores and AMP elements within the system. In this way, independent subsystems can be implemented within the same cluster.

It is entirely possible (and normal) to build AMP systems in which individual cores are running different operating systems (these are called Multi-OS systems).

The selection of software MP model is determined by the characteristics of the applications running in the system. In networking systems, for example, it can be convenient to provide a separation between control-plane (AMP) and data-plane (SMP) sections of the system. In other systems, it might be desirable to isolate those parts of the system that require hard real-time response from applications that require raw performance and to implement these on separate cores.

Note

Where synchronization is required between these separate cores, it can be provided through message passing communication protocols, for example, the *Multicore Communications Association API* (MCAPI). These can be implemented by using shared memory to pass data packets and by the use of software triggered interrupts to implement a so-called door-bell mechanism.

18.4 Heterogeneous multi-processing

The term *Heterogeneous multi-processing* (HMP) finds application in many different contexts. It is often conflated with AMP to describe systems that are composed of different types of processors, such as a multi-core ARM applications processor and an application specific processor (such as a baseband controller chip or an audio codec chip).

ARM uses HMP to mean a system composed of clusters of application processors that are 100% identical in their instruction set architecture but very different in their microarchitecture. All the processors are fully cache coherent and a part of the same coherency domain.

This is best explained using the ARM implementation of HMP technology known as big.LITTLE. In a big.LITTLE system energy efficient LITTLE cores are coherently coupled with high performance big cores to form a system that can accomplish both high intensity and low intensity tasks in the most energy efficient manner.

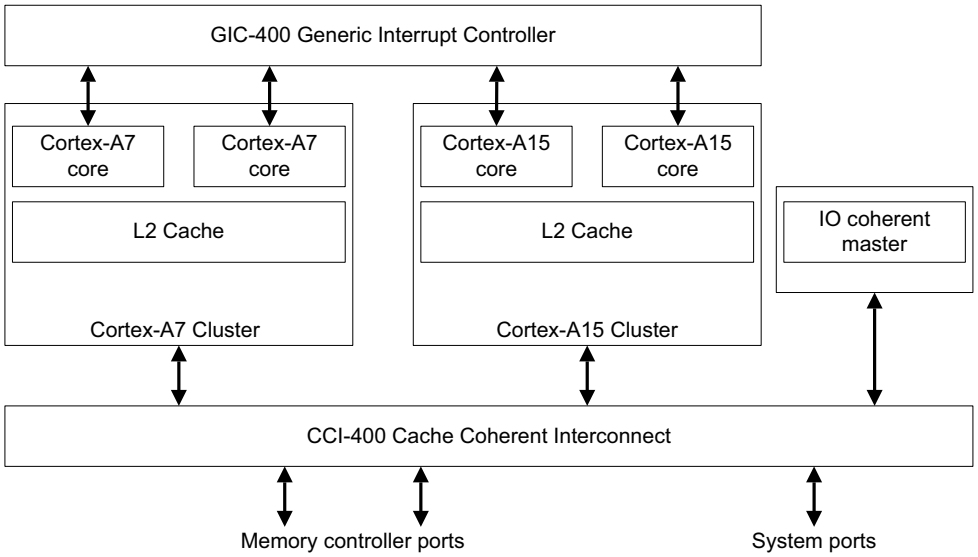


Figure 18-3 A typical big.LITTLE system

The central principle of big.LITTLE is that application software can run unmodified on either type of processor. For a detailed overview of big.LITTLE technology, including the software execution models see Chapter 23.

18.5 Cache coherency

Coherency is about ensuring all processors, or bus masters in the system see the same view of memory. It means that changes to data held in the cache of one core are visible to the other cores, making it impossible for cores to see stale copies of data (the old data from before it was changed by the first core). For example, if you have a processor that is creating a data structure then passing it to a DMA engine to move, both the processor and DMA must see the same data. If that data were cached in the core and the DMA reads from external memory, the DMA will read old, stale data.

There are three mechanisms to maintain coherency:

Disable caching

This is the simplest mechanism but might cost significant core performance. To get the highest performance processors are pipelined to run fast, and to run from caches that offer a very low latency. Caching of data that is accessed multiple times increases performance significantly and reduces DRAM accesses and power. Marking data as “non-cached” could impact performance and power.

Software managed coherency

Software managed coherency is the traditional solution to the data sharing problem. Here the software, usually device drivers, must clean or flush dirty data from caches, and invalidate old data to enable sharing with other processors or masters in the system. This takes processor cycles, bus bandwidth, and power.

Where there are high rates of sharing between requesters the cost of software cache maintenance can be significant, and can limit performance.

Hardware managed coherency

Hardware Coherency is the most efficient solution. Any data marked ‘shared’ in a hardware coherent system will always be up to date. All cores and bus masters in that sharing domain see the exact same value.

While hardware coherency might add some complexity to the interconnect and clusters, it greatly simplifies the software and enables applications that would not be possible with software coherency.

18.5.1 MESI and MOESI protocols

There are a number of standard ways by which cache coherency schemes can operate. Most ARM processors use the MOESI protocol, while the Cortex-A9 uses the MESI protocol.

Depending on which protocol is in use, the SCU marks each line in the cache with one of the following attributes: M (Modified), O (Owned), E (Exclusive), S (Shared) or I (Invalid). These are described below:

Modified The most up-to-date version of the cache line is within this cache. No other copies of the memory location exist within other caches. The contents of the cache line are no longer coherent with main memory.

Owned This describes a line that is dirty and in possibly more than one cache. A cache line in the owned state holds the most recent, correct copy of the data. Only one core can hold the data in the owned state. The other cores can hold the data in the shared state.

Exclusive The cache line is present in this cache and coherent with main memory. No other copies of the memory location exist within other caches.

Shared The cache line is present in this cache and coherent with main memory. Copies of it can also exist in other caches in the coherency scheme.

Invalid The cache line is invalid.

The rules for the standard implementation of the protocol are as follows:

- A write can only be done if the cache line is in the Modified or Exclusive state. If it is in the Shared state, all other cached copies must be invalidated first. A write moves the line into the Modified State.
- A cache can discard a Shared line at any time, changing to the Invalid state. A Modified line is written back first.
- If a cache holds a line in the Modified state, reads from other caches in the system will get the updated data from the cache. Conventionally, this is done by first writing the data to main memory and then changing the cache line to the Shared state, before performing a read.
- A cache that has a line in the Exclusive state must move the line to the Shared state when another cache reads that line.
- The Shared state might not be precise. If one cache discards a Shared line, another cache might not be aware that it could now move the line to Exclusive status.

ARM multi-core processors also implement optimizations that can copy clean data and move dirty data directly between participating L1 caches, without having to access (and wait for) external memory. This activity is handled in multi-core systems by the *Snoop Control Unit* (SCU).

18.5.2 Snoop Control Unit

The SCU maintains coherency between the L1 data cache of each core. Since executable code changes much less frequently, this functionality is not extended to the L1 instruction caches. The coherency management is implemented using a MOESI-based protocol, optimized to decrease the number of external memory accesses. In order for the coherency management to be active for a memory access, all of the following must be true:

- The SCU is enabled, through its control register located in the private memory region. See *Private memory region* on page 18-19. The SCU has configurable access control, restricting which processors can configure it.
- The core performing the access is configured to participate in the Inner Shareable domain, configured using the operating system at boot time, by setting the somewhat misleadingly named SMP bit in the CP15:ACTLR, Auxiliary Control Register.

  The following code example sets the SMP bit in either a Cortex-A7 or Cortex-A15 ACTLR:

```
MRC     p15, 0, r0, c1, c0, 1   ; Read ACTLR
ORR     r0, r0, #0x040          ; Set bit[6] SMP (coherency)
MCR     p15, 0, r0, c1, c0, 1   ; Write ACTLR
DSB
```

- The MMU is enabled.
- The page being accessed is marked as Normal Shareable, with a cache policy of *write-back*, *write-allocate*. Device and Strongly-ordered memory, however, are not cacheable, and write-through caches behave like uncached memory from the point of view of the core.

The SCU can only maintain coherency within a single cluster. If there are additional processors or other bus masters in the system, explicit software synchronization is required when these share memory with the MP block. Alternatively, the *Accelerator Coherency Port* (ACP) can be used.

18.5.3 Accelerator Coherency Port (ACP)

The *Accelerator Coherency Port* (ACP) is a feature of the Cortex-A5, Cortex-A9, Cortex-A12 and Cortex-A15 processors.

It provides an AXI slave interface into the Snoop Control Unit of the processor. The AXI bus interface is defined in the ARM AMBA specification.

This slave interface can be connected to an external uncached AXI master, such as a DMA engine, for example, which is able to initiate both read and write memory transfers to the ACP. It enables such a device to snoop the L1 caches of all cores, avoiding the requirement for synchronization through external memory. A cached device can also be connected, but this requires manual coherency management through software.

The behavior of accesses performed on the ACP is as follows:

- Addresses used by the ACP are physical addresses that can be snooped by the SCU to be fully coherent.
- ACP reads can hit in the L1 D-cache of any core.
- Writes on the ACP invalidate any stale data in L1 and write-through to L2.

The ACP enables an external device to see core-coherent data without knowledge of where the data is in the memory hierarchy. Memory transfers are automatically coherent in the same way as happens between the L1 D-caches of the cores in the cluster.

Use of the ACP can both increase performance and save power, as there will be reduced traffic to external memory and faster execution.

Programmers writing device drivers that use the ACP do not have to be concerned with coherency issues, because no cache cleaning or invalidation is required to ensure coherency. However, the use of memory barriers (DMB) or external cache synchronization operations can still be necessary, if a particular ordering must be enforced.

18.5.4 The Cache Coherent Interface (CCI)

Extending hardware coherency to a multi-cluster system requires a coherent bus protocol. In 2011 ARM released the AMBA 4 ACE specification that introduces the *AXI Coherency Extensions* (ACE) on top of the popular AXI protocol. The full ACE interface enables hardware coherency between clusters and enables an SMP operating system to extend to more cores. In the ACE protocol, three coherency channels are added in addition to the normal five channels of AXI. If you have two clusters, any shared access to memory can snoop into the cache of the other cluster to see if the data is already on chip, if not, it is fetched from external memory.

The AMBA 4 ACE-Lite interface is designed for I/O (or one-way) coherent system masters like DMA engines, network interfaces and GPUs. These devices might not have any caches of their own, but they can read shared data from the ACE processors.

The CoreLink™ CCI-400 is one of the first implementations of AMBA 4 ACE and supports up to two ACE processor clusters enabling up to eight cores to see the same view of memory and run an SMP OS.

If we return to our example of a multi-cluster system:

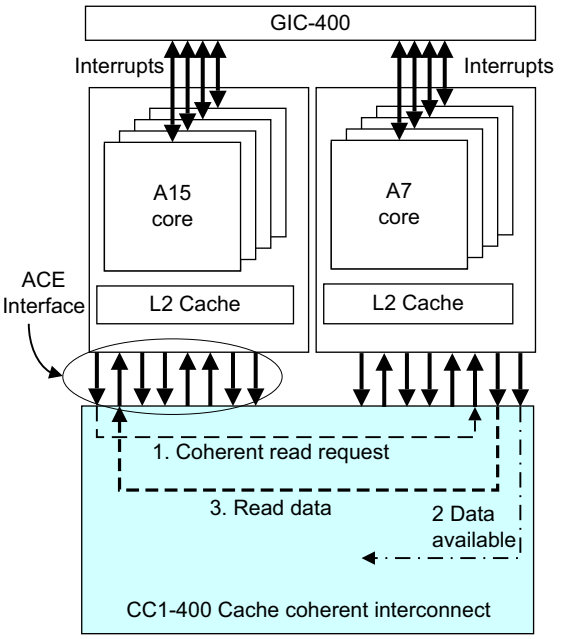


Figure 18-4 Cache coherency in a multi-cluster system

Figure 18-4 shows the steps in a coherent data read from the Cortex-A7 cluster to the Cortex-A15 cluster. This starts with the Cortex-A7 cluster issuing a Coherent Read Request. The CCI-400 hands over the request to the Cortex-A15 processor to snoop into Cortex-A15 cluster cache.

When the request from the CCI-400 is received, the Cortex-A15 cluster checks the data availability and reports this information back. If the requested data is in the cache, the CCI-400 moves the data from the Cortex-A15 cluster to the Cortex-A7 cluster, resulting in a cache linefill in the Cortex-A7 cluster. The CCI-400 and the ACE protocol enable full coherency between the Cortex-A15 and Cortex-A7 clusters, enabling data sharing to take place without external memory transactions.

18.6 TLB and cache maintenance broadcast

An individual core can broadcast Translation Lookaside Buffer and cache maintenance operations to other cores in the Inner Shareable coherency domain. This can be required whenever shared translation tables are modified, for example. This behavior might have to be enabled by you. For example, in the Cortex-A9 processor, this is controlled by the FW bit in the *Auxiliary Control Register* (ACTLR). Maintenance operations can only be broadcast and received when the processor is configured to participate in the Inner Shareable domain, using the SMP bit in ACTLR. Only Inner Shareable operations are broadcast, for example:

- To invalidate TLB entry by virtual address.
- To clean or invalidate data cache line by virtual address.
- To invalidate instruction cache line by virtual address.

Some care is required with cache maintenance activity in multi-core systems that include a L2C-310 L2 cache (or similar). Cleaning or invalidating the L1 cache and L2 cache will not be a single atomic operation. A core might therefore perform cache maintenance on a particular address in both L1 and L2 caches only as two discrete steps. If another core were to access the affected address between those two actions, a coherency problem can occur. Such problems can be avoided by following two simple rules.

- When cleaning, always clean the innermost (L1) cache first and then clean the outer cache(s).
- When invalidating, always invalidate the outermost cache first and the L1 cache last.

18.7 Handling interrupts in an SMP system

Multi-core processors include an integrated interrupt controller that implements the GIC architecture (see *The Generic Interrupt Controller* on page 12-7 for additional details). This controller provides 32 private interrupts per core, of which the lower 16 are *Software Generated Interrupts* (SGI) that can be generated only through software operations, and the rest are *Private Peripheral Interrupts* (PPI). It also provides a configurable number of *Shared Peripheral Interrupts* (SPI), up to 224 in current multi-core implementations). It supports interrupt prioritization, pre-emption and routing to different cores.

In a multi-core processor, the GIC control registers are memory-mapped and located within the Private memory region, see *Private memory region* on page 18-19.

The Interrupt Processor Targets registers configure that cores individual interrupts are routed to. They are ignored for private interrupts.

The registers controlling the private interrupts (0-31) are banked, so that each core can have its own configuration for these. This includes priority configuration and the enabling or disabling of individual interrupts.

The *Software Generated Interrupt* (SGI) Register can assert a private SGI on any core, or a groups of cores. The priority of a software interrupt is determined by the priority configuration of the receiving core, not the one that sends the interrupt. The interrupt acknowledge register bits [12:10] will provide the ID of the core that made the request. The target list filter field within this register provides shorthand for an SGI to be sent to all processors, all but self or to a target list.

For a software generated interrupt in a multi-core processor, the Interrupt Acknowledge Register also contains a bitfield holding the ID of the core that generated it. When the interrupt service routine has completed, it must write-back the value previously read from the Interrupt Acknowledge Register into the End Of Interrupt Register. For an SGI, this must also match the ID of the signalling core.

In both AMP and SMP systems, it is likely that cores will trigger interrupts on other cores (or themselves), a so called *softirq*. These can be used for kernel synchronization operations, or for communicating between AMP cores. For operations requiring more information passed than a raised interrupt, you can use a shared buffer to store messages. Before the core can receive an interrupt, some initialization steps are required both in the distributor and in the core interface.

18.8 Exclusive accesses

In an SMP system, data accesses frequently have to be restricted to one modifier at any particular time. This can be true of peripheral devices, but also for global variables and data structures accessed by more than one thread or process. Furthermore, library code that is used by multiple threads must be designed to ensure that concurrent access or execution is possible – it must be *reentrant*.

Note

Code is *reentrant* if it can be interrupted in the middle of its execution and then be called again before the previous invocation has completed.

Protection of such shared resources is often through a method known as *mutual exclusion*. The section of code that is being executed by a core while accessing such a shared resource is known as the *critical section*.

In a single core system, mutual exclusion can be achieved by disabling interrupts when inside critical sections. This is not sufficient in a multi-core system, as disabling interrupts on one core will not prevent others from entering the critical section. It is also not ideal since interrupts cannot be disabled from within User mode. Of course, there are other problems with this technique, including reduced responsiveness to real-time events, particularly if the critical section is long.

In a multi-core system, we can use a *spinlock* – effectively a shared flag with an atomic (indivisible) mechanism to test and set its value. We perform this operation in a tight loop to wait until another thread (or core) clears the flag. We require hardware assistance in the form of special machine instructions to implement this. Application developers should not worry about the low-level implementation detail, but should instead become familiar with the lock and unlock calls available in their OS or threading library API.

The ARM architecture provides three instructions relating to exclusive access. Variants of these instructions, that operate on byte, halfword, word or doubleword sized data, are also provided. The instructions rely on the ability of the core or memory system to tag particular addresses to be monitored for exclusive access by that core, using an *exclusive access monitor*.

- `LDREX` (Load Exclusive) performs a load of memory, but also tags the physical address to be monitored for exclusive access by that core.
- `STREX` (Store Exclusive) performs a conditional store to memory, succeeding only if the target location is tagged as being monitored for exclusive access by that core. This instruction returns the value of 1 in a general purpose register if the store does not take place, and a value of 0 if the store is successful.
- `CLREX` (Clear Exclusive) clears any exclusive access tag for that core.

Load Exclusive and Store Exclusive operations must be performed only on Normal memory (see *Normal memory* on page 10-4) and have slightly different effect depending on whether the memory is marked as Shareable or not. If the core reads from Shareable memory with an `LDREX`, the load happens and that physical address is tagged to be monitored for exclusive access by that core. If any other core writes to that address and the memory is marked as Shareable, the tag is cleared.

If the memory is not Shareable then any attempt to write to the tagged address by the one that tagged it results in the tag being cleared. If the core does an additional `LDREX` to a different address, the tag for the previous `LDREX` address is cleared. Each core can only have one address tagged.

STREX can be considered as a conditional store. The store is performed only if the physical address is still marked as exclusive access (this means it was previously tagged by this core and no other core has since written to it). STREX returns a status value showing if the store succeeded. STREX always clears the exclusive access tag.

The use of these instructions is not limited to multi-core systems. In fact, they are frequently employed in single core systems, to implement synchronization operations between threads running on the same core.

In hardware, the core includes a device named the *local monitor*. This monitor observes the core. When the core performs an exclusive load access, it records that fact in the local monitor. When it performs an exclusive store, it checks that a previous exclusive load was performed and fails the exclusive store if this was not the case. The architecture enables individual implementations to determine the level of checking performed by the monitor. The core can only tag one physical address at a time. An LDREX from a particular address can be followed shortly after by an STREX to the same location, before an LDREX from a different address is performed. This is because the local monitor does not have to store the address of the exclusive tag (although it can do, if the processor implementer decides to do this). The architecture enables the local monitor to treat any exclusive store as matching a previous LDREX address. For this reason, use of the CLREX instruction to clear an existing tag is required on context switches.

Where exclusive accesses are used to synchronize with external masters outside the core, or to regions marked as Sharable even between cores in the same cluster, it is necessary to implement a global monitor within the hardware system. This acts as a wrapper to one or more memory slave devices and is independent of the individual cores. This is specific to a particular SoC and might not exist in any particular system. An LDREX/STREX sequence performed to a memory location that has no suitable exclusive access monitor will fail, with the STREX instruction always returning 1.

18.9 Booting SMP systems

Initialization of the external system might have to be synchronized between cores. Typically, only one of the cores in the system has to run code that initializes the memory system and peripherals. Similarly, the SMP operating system initialization typically runs on only one core – the *primary core*. When the system is fully booted, the remaining cores are brought online and this distinction between the primary core and the others (secondary cores) is lost.

If all of the cores come out of reset at the same time, they will normally all start executing from the same reset vector. The boot code will then read the cluster ID to determine which core is the primary. The primary core will perform the initialization and then signal to the secondary ones that everything is ready. An alternative method is to hold the secondary cores in reset while the primary core does the initialization. This requires hardware support to co-ordinate the reset.

In an AMP system, the bootloader code will determine the suitable start address for the individual cores, based on their cluster ID (as each core will be running different code). Care might be required to ensure correct boot order in the case where there are dependencies between the various applications running on different cores.

18.9.1 Processor ID

Booting provides a simple example of a situation where particular operations must be performed only on a specific core. Other operations perform different actions dependent on the core on which they are executing.

The CP15:MPIDR *Multiprocessor Affinity Register* provides an identification mechanism in a multi-core system.

This register was introduced in version 7 of the ARM architecture, but was in fact already used in the same format in the ARM11 MPCore. In its basic form, it provides up to three levels of affinity identification, with 8 bits identifying individual blocks at each level (Affinity Level 0, 1 and 2).

This information can also be of value to an operating system scheduler, as an indication of the order of magnitude of the cost of migrating a process to a different core, processor or cluster.

The format of the register was slightly extended with the ARMv7-A multiprocessing extensions. This extends the previous format by adding an identification bit to reflect that this is the new register format, and adds the *U* bit that indicates whether the current core is the only core in a single-core implementation or not.

18.9.2 SMP boot in Linux

The boot process for the primary core is as described in *Boot process* on page 11-14. The method for booting the secondary cores can differ somewhat depending on the SoC being used. The method that the primary core invokes in order to get a secondary core booted into the operating system is called `boot_secondary()` and must be implemented for each *mach* type that supports SMP. Most of the other SMP boot functionality is extracted out into generic functions in `linux/arch/arm/kernel`.

The method below describes the process on an ARM Versatile Express development board (`mach-vexpress`).

While the primary core is booting, the secondary cores will be held in a standby state, using the `WFI` instruction. It will provide a startup address to the secondary cores and wake them using an *Inter-Processor Interrupt* (IPI), meaning an SGI signalled through the GIC (see *Handling interrupts in an SMP system* on page 18-14). Booting of the secondary cores is serialized, using the global variable `pen_release`. Conceptually, you can think of the secondary cores being in a

holding pen and being released one at a time, under control of the primary core. The variable `pen_release` is set by the kernel code to the ID value of the processor to boot and then reset by that core when it has booted. When an inter-processor interrupt occurs, the secondary core will check the value of `pen_release` against their own ID value using the MPIDR register.

Booting of the secondary core will proceed in a similar way to the primary. It enables the MMU (setting the TTB register to the new translation tables already created by the primary). It enables the interrupt controller interface to itself and calibrates the local timers. It sets a bit in `cpu_online_map` and calls `cpu_idle()`. The primary processor will see the setting of the appropriate bit in `cpu_online_map` and set `pen_release` to the next secondary core.

18.10 Private memory region

In the Cortex-A5, and Cortex-A9 MPCore processors, all of the internal peripherals are mapped to the private address space. This is an 8KB region location within the memory map at an address determined by the hardware implementation of the specific device used (this can be read using the CP15 Configuration Base Address Register).

The registers in this region are fixed in little-endian byte order, so some care is required if the CPSR E bit is set when accessing it. Some locations within the region exist as banked versions, dependent on the processor ID. The Private memory region is not accessible through the ACP. Table 18-1 shows the layout of this Private memory region.

Table 18-1 Private memory region layout

Base Address offset	Function
0x0000	Snoop Control Unit (SCU)
0x0100	Interrupt controller CPU Interface
0x0200	Global Timer
0x0600	Local Timer/Watchdog
0x1000	Interrupt Controller Distributor

18.10.1 Timers and watchdogs

Each core in a multi-core processor implements a standard timer and a watchdog, both private to that core.

These can be configured to trigger after a number of core cycles, using a 32-bit start value and an 8-bit pre-scaler. They can be operated using interrupts, or by periodic polling, supported by the Timer and Watchdog Interrupt Status Registers. They stop counting while the core is in debug state. The timer can be configured in *single-shot* or *auto-reload* mode. The watchdog can be operated in classic watchdog fashion, where it asserts the core reset signal, for that specific core on time-out. Alternatively, it can be used as a second timer.

The Cortex-A9, and Cortex-A5 processors also include a global timer, shared between all cores, but with banked comparator and auto-increment registers for each core. It is a single, incrementing 64-bit counter, accessible only through 32-bit accesses. It can be configured to trigger an interrupt when the comparator value is reached. The auto-increment feature causes the processor comparator register to be incremented after each match. This is typically used by the OS scheduler, to trigger the scheduler on each core, at different times.

Chapter 19
Parallelizing Software

Chapter 18 *Multi-core processors*, described how an SMP system can enable you to run multiple threads efficiently and concurrently across multiple cores. In this case, the parallelization is, in effect, handled on our behalf by the OS scheduler.

In many cases, however, this is insufficient and you must take steps to rewrite code to take advantage of speed-ups available through parallelization. An obvious example is where a single application requires more performance than can be delivered by a single core. More commonly, we can have the situation where an application requires much more performance than all of the others within a system, when it is said to be *dominant*. This prevents efficient energy usage, as we cannot perform optimal load-balancing. An unbalanced load distribution does not permit efficient dynamic voltage and frequency scaling.

The operating system cannot automatically parallelize an application. It is limited to treating that application as a single scheduling unit. In such cases, the application itself has to be split into multiple smaller tasks by you. Of course, this means each of these tasks must be able to be independently scheduled by the OS, as separate threads. A thread is a part of a program that can be run independently and concurrently with other parts of a program. If you *decompose* an application into smaller execution entities that can be separately scheduled, the OS can spread the threads of the application across multiple cores.

19.1 Amdahl's law

Amdahl's Law defines the theoretical maximum speedup achievable by parallelizing an application. The maximum speedup is given by the formula:

Max speedup = 1/ ((1-P) + (P/N))

where:

P = parallelizable proportion of program,

N = Number of cores.

This is, of course, an abstract, academic view. In practice, this provides a theoretical maximum speedup, as there are a number of overheads associated with concurrency. Synchronization overheads occur when a thread must wait for another task or tasks before it can continue execution. If a single task is slow, the whole program must wait. In addition, you might have critical sections of code, where only a single task is able to run at a time. There might also be occasions when all tasks are contending for the same resource or where no other tasks can be scheduled to run by the OS.

19.2 Decomposition methods

The best approach to decomposition of an application into smaller tasks capable of parallel execution depends on the characteristics of the original application. Large data-processing algorithms can be broken down into smaller pieces by sub-division into a number of similar threads that execute in parallel on smaller portions of a dataset. This is known as *data decomposition*.

Consider the example of color-space conversion, from RGB to YUV. Start with an array of pixel data. The output is a similar array giving chrominance and luminance data for each pixel. Each output value is calculated by performing a small number of multiplies and adds. Crucially, the output Y, U and V values for each pixel depend only on the input R, G and B values for that pixel. There is no dependency on the data values of other pixels. Therefore, the image can be divided into smaller blocks and you can perform the calculation using any number of instances of your code. This does not require any change to your original algorithm, only changes to the amount of data supplied to each thread.

Split the image into stripes (1/N arrays, where we have N threads) so that each thread works on a stripe. The level of detail of the stripes can be an important consideration, given that it is clearly better for cacheability if each thread works on a contiguous block of pixels in array order. The code does not have to be modified to take care of scheduling, the operating system takes care of it. Color space conversion would be a task where the NEON unit could significantly improve performance. Splitting the task across several cores can provide additional parallelization gains than using NEON instructions alone.

A different approach is that of *task decomposition*. Here, areas of code that are independent of each other and capable of being executed concurrently can be identified. This is a little more difficult, as you must consider the discrete operations being carried out and the interactions between them. A simple example might be the start-up sequence of a program. One task might be to check that you have a valid license for the software. Another task might be to display a start-up banner with a copyright message. These are independent tasks with no dependency on each other and can be performed in separate threads. Again, no change is required to the source code that carries out these isolated tasks. These must be supplied to the OS kernel Scheduler as separate execution threads.

Of course, not all algorithms are able to be handled through data or task decomposition. Instead, you must analyze the program with the aim of identifying functional blocks. These are independent pieces of code with defined inputs and outputs that have some scope to be parallelized. Such functional blocks often depend on input from other blocks (they have a *serial dependency*), but do not have a corresponding dependency on time (a *temporal dependency*). This is (in some respects) analogous to the hardware pipelining employed in the core itself.

The software for an MPEG video encoder provides a good example of this. Input data, in the form of an analog video signal is sampled and processed through a pipeline of discrete functional blocks. First, both inter-frame and intra-frame redundancies are removed. Then, quantization takes place to reduce the number of bits required to represent the video. After this, motion vector compensation takes place, run length compression and finally the encoded sub-stream is stored.

At the same time that data from one frame is being run-length compressed and stored, you can also start to process the next frame. Within a frame, the motion vector compensation process can be parallelized. You can use multiple parallel threads to operate on a frame (an example of data decomposition).

When decomposing an application using these techniques, you must consider the overheads associated with task creation and management. An appropriate level of granularity is required for best performance. If you make your datasets too small, too big, or have too many datasets, it can reduce performance. In the color-space conversion example, it would not be sensible to have a separate thread for each pixel, even though this is logically possible.

19.3 Threading models

When an algorithm has been analyzed to determine the potential changes that can be made for parallelization, you must modify code to map the algorithm to smaller, threaded execution units. There are two widely-used threading models, the *workers' pool* model and the *fork-join* model, not to be confused with the UNIX `fork` system call. The latter creates (spawns) a new thread whenever one is required (that is, threads are created on-demand.) The operating system then schedules the various threads across the available cores.

Each of the newly spawned threads is typically considered to be either a *detached* thread, or a *joinable* thread. A detached thread executes in the background and terminates when it has completed, without any message to the parent process. Of course, communication to or from such processes can be implemented manually by you, through the available signaling mechanisms, or using global variables. A joinable thread, in contrast, will communicate back to the main thread, at a point set by you. The parent process might have to wait for all joinable threads to return before proceeding with the next execution step.

In the fork-join model, individual threads have explicit start and end conditions. There is an overhead associated with managing their creation and destruction and latencies associated with the synchronization point. This means that threads must be sufficiently long-lived to justify these costs.

If you know that some execution threads will be repeatedly required to consume input data, you can instead use the workers' pool threading model. Here, you create a pool of worker threads at the start of the application. The pool can consist of multiple instances of the same algorithm, where the distributor (also called producer or boss) will dispatch the task to the first available worker (consumer) thread. Alternatively, the workers' pool can contain several different data processing operators and data-items will be tagged to show which worker can consume the data.

The number of worker threads can be changed dynamically to handle peaks in the workload. Each worker thread performs a task until it is finished, then interrupts the boss to be assigned another task. Alternatively, the boss can periodically poll workers to see whether one is ready to receive another task. The work queue model is similar. The boss places tasks in a queue, and workers check the queue and take tasks to perform. An additional variant is to have multiple bosses, sharing the pool of workers. The boss threads place tasks onto a queue, from where they are taken by the worker threads.

In each of these models, it must be understood that the amount of work to be performed by a thread can be variable and unpredictable. Even for threads that operate on a fixed quantity of data, it can be the case that data dependencies cause different execution times for similar threads. There is always likely to be some synchronization overhead associated with the requirement for a parent thread to wait for all spawned threads to return (in the fork-join model) or for a pool of workers to complete data consumption before execution can be resumed.

19.4 Threading libraries

We have looked at how to make our target application capable of concurrent execution. We must now consider actual source code modifications. This is normally done using a threading library, normally utilizing multi-threading support available in the OS. When modifying existing code, you must take care to ensure that all shared resources are protected by proper synchronization. This includes any libraries used by the code, as all libraries are not *reentrant*. In some cases, there can be separate reentrant libraries for use in multi-threaded applications. A library that is designed to be used in multi-threaded applications is called *thread-safe*. If a library is not known to be thread-safe, only one thread can be permitted to make calls to the library functions.

The most commonly used standard in this area is POSIX threads (Pthreads), a subset of the wider POSIX standard. POSIX (IEEE std. 1003) is the Portable Operating System Interface, a collection of OS interface standards. Its goal is to assure interoperability and portability of code between systems. Pthreads defines a set of API calls for creating and managing threads. Pthreads libraries are available for Linux, Solaris, and Windows.

There are several other multi-threading frameworks, such as OpenMP that can simplify multi-threaded development by providing high-level primitives, or even automatic multi-threading. OpenMP is a multi-platform, multi-language API that supports shared memory multi-processing through a set of libraries and compiler directives plus environment variables that affect run-time behavior.

Pthreads provides a set of C primitives that enable you to create, manage, and terminate threads and to control thread synchronization and scheduling attributes. Let you examine, in general terms, how you can use Pthreads to build multi-threaded software to run on our SMP system. We'll deal with the following types:

- *pthread_t* – thread identifier
- *pthread_mutex_t* – mutex
- *sem_t* - semaphore.

You must modify your code to include the appropriate header files.

```
#include <pthread.h>
#include <semaphore.h>
```

You must also link your code using the pthread library with the switch `-lpthread`.

To create a thread, you must call `pthread_create()`, a library function that requires four arguments. The first of these is a pointer to a `pthread_t`, which is where you will store the thread identifier. The second argument is the attribute that can point to a structure that modifies the thread's attributes (for example scheduling priority), or be set to NULL if no special attributes are required. The third argument is the function the new thread will start by executing. The thread will be terminated if this function returns. The fourth argument is a `void *` pointer supplied to the thread. This can receive a pointer to a variable or data structure containing relevant information to the thread function.

A thread can complete either by returning, or calling `pthread_exit()`. Both will terminate the thread. A thread can be detached, using `pthread_detach()`. A detached thread will automatically have its associated data structures (but not explicitly allocated data) released on exit.

For a thread that has not been detached, this resource cleanup will happen as part of a `pthread_join()` call from another thread. The library function `pthread_join()` enables you to make a thread stall and wait for completion of another thread. Take care, as so-called *zombie* threads can be created by joining a thread that has already completed. It is not possible to join a detached thread (one that has called `pthread_detach()`).

Mutexes are created with the `pthread_mutex_init()` function. The functions `pthread_mutex_lock()` and `pthread_mutex_unlock()` are used to lock or unlock a mutex. `pthread_mutex_lock()` blocks the thread until the mutex can be locked. `pthread_mutex_trylock()` checks whether the mutex can be claimed and returns an error if it cannot, rather than blocking. A mutex can be deleted when no longer required with the `pthread_mutex_destroy()` function.

Semaphores are created in a similar way, using `sem_init()` – one key difference being that you must specify the initial value of the semaphore. `sem_post()` and `sem_wait()` are used to increment and decrement the semaphore.

The GNU tools for ARM cores support full thread-local storage using the Native POSIX Thread library (NPTL) that enables efficient use of POSIX threads with the Linux kernel. There is a one-to-one correspondence between threads created with `pthread_create()` and kernel tasks

Example 19-1 provides a simple example of using the Pthreads library.

Example 19-1 Pthreads code

```
void *thread(void *vargp);
int main(void)
{
  pthread_t tid;
  pthread_create(&tid, NULL, thread, NULL);
  /* Parallel execution area */
  pthread_join(tid, NULL);
  return 0;
}
/* thread routine */
void *thread(void *vargp)
{
  /* Parallel execution area */
  printf("Hello World from a POSIX thread!\n");
  return NULL;
}
```

19.4.1 Inter-thread communications

Semaphores can be used to signal to another thread. A simple example would be where one thread produces a buffer containing shared data. It could use a semaphore to indicate to another thread that the data can now be processed.

For more complex signaling, a message passing protocol might be required. Threads within a process use the same memory space, so an easy way to implement message passing is by posting in a previously agreed-upon mailbox and then incrementing a semaphore.

19.4.2 Threaded performance

There are a few general points to consider when writing a multi-threaded application:

- Each thread has its own stack space and care might be required with the size of this if large numbers of threads are in use.
- Multiple threads contending for the same mutex or semaphore creates contention and wasted core cycles. There is a large body of research on programming techniques to reduce this performance loss.

- There is an overhead associated with thread creation. Some applications avoid this by creating a thread pool at startup. These threads are used on demand and then returned to the thread pool for later re-use, rather than being closed completely.

19.4.3 Thread affinity

Thread affinity refers to the practice of assigning a thread to a particular core or cores. When the scheduler wants to run a particular thread, it will use only the selected core(s) even if others are idle. This can be a problem if too many threads have an affinity set to a specific core. By default, threads are able to run on any core in an SMP system.

ARM DS-5 Streamline is able to reveal the affinity of a thread by using a display mode called Core map. This mode can be used to visualize how tasks are divided up by the kernel and shared amongst several cores. See *DS-5 Streamline* on page 16-4.

19.4.4 Thread safety and reentrancy

Functions that can be used concurrently by more than one thread concurrently must be both thread-safe and reentrant. This is particularly important for device drivers and for library functions.

For a function to be reentrant, it must fulfill the following conditions:

- All data must be supplied by the caller.
- The function must not hold static or global data over successive calls.
- The function cannot return a pointer to static data.
- The function cannot itself call functions that are not reentrant.

For a function to be thread-safe, it must protect shared data with locks. This means that the implementation must be changed by adding synchronization blocks to protect concurrent accesses to shared resources, from different threads. Reentrancy is a stronger property, this means that not every thread-safe function is reentrant.

There are number of common library functions that are not reentrant. For example, the function `ctime()` returns a pointer to static data that is over-written on each call.

19.5 Performance issues

There are several multi-core specific issues relating to performance of threads:

Bandwidth The connection to external memory is shared between all cores within a cluster. The individual cores run at speeds far higher than the external memory and so are potentially limited (in I/O intensive code) by the available bandwidth.

Thread dependencies and priority inversion

The execution of a higher priority thread can be stalled by a lower priority thread holding a lock to some shared data. Alternatively, an incorrect split in thread functionality can lead to a situation where no benefit is seen because the threads have fully serialized dependencies.

Cache contention and false sharing

If multiple threads are using data that reside within the same coherent cache lines, there can be cache line migration overhead even if the actual variables are not shared.

19.5.1 Bandwidth concerns

Bandwidth issues can be optimized in a number of ways. Clearly, the code itself must be optimized using the techniques described in Chapter 17, to minimize cache misses and therefore reduce the bandwidth utilization.

Another option is to pay attention to thread allocation. The kernel scheduler does not pay any attention to data usage by threads; instead it makes use of priority to decide which threads to run. You can provide hints that enable more efficient scheduling through the use of thread affinity.

19.5.2 Thread dependencies

In real systems you can have threads with higher or lower priority that both access a shared resource. This gives scope for some potential difficulties. The term *starvation* is used to describe the situation where a thread is unable to get access to a resource after repeated attempts to claim it.

Priority inversion is said to occur when a lower priority task has a lock on a resource that a higher priority requires in order to be able to execute. In other words, a lower priority task prevents the higher priority task from executing. Priority inheritance resolves this by temporarily raising the priority of the task that has the lock to the highest level. This causes that task to execute as quickly as possible and relinquish the shared resource as soon as it can.

Operating systems (particularly real time operating systems) have ways to avoid such problems automatically. One method is not to permit lower-priority threads from directly accessing resources required by higher-priority threads, they might have to use a higher-priority proxy thread to perform the operation. A similar approach is to temporarily increase the priority of the low-priority thread while it is holding the critical resource, ensuring that the scheduler will not pre-empt execution of that thread while in the critical selection.

A program that relies on threads executing in a particular sequence to work correctly might have a *race condition*. Single-core real-time systems often implicitly rely on tasks being executed in a priority based order. Tasks will then execute to completion, without pre-emption. Later tasks can rely on earlier tasks having completed. This can cause problems if such software is moved to a multi-core system without careful checking for such assumptions. A lower-priority task can run at the same time as a higher-priority task and the expected execution order of the original single-core system is no longer guaranteed. There are number of ways to resolve this. A simple

approach is to set task affinity to make those tasks run on the same core. This requires little change to the legacy code, but does break the symmetry of the system and remove scope for load balancing. A better approach is to enforce serial execution through the use of the kernel synchronization mechanisms that give you explicit control over the execution flow and better SMP performance, but does require the legacy code to be modified.

19.5.3 Cache thrashing

Cortex-A series processors use physically tagged caches that remove the requirement for flushing caches on context switch. In an SMP system, it is possible for tasks to migrate between the different cores in the system. The scheduler starts a task on a core. It runs for a certain period and is then replaced by a different task. When that task is restarted at a later time by the scheduler, this could be on a different core. This means that the task does not get the potential benefit of cache data already being present in the core cache. Memory intensive tasks that quickly fill the data cache might thrash each others' cached data. This has an impact on both performance (slower execution because of the higher number of cache misses) and system energy usage (because of additional interaction with external memory). Multi-core optimizations for cache line migration mitigate the effects of this. In addition, the OS scheduler can try to reduce the problem by aiming to keep tasks on the same core. As we have seen, you can also do this by setting core affinity to threads and processes.

19.5.4 False sharing

This is a problem of systems with shared coherent caches and is effectively a form of involuntary memory contention. It can happen when a core regularly accesses data that is never changed, and shares a cache line with data that will be altered by another core. The MESI protocol can end up migrating data that is not truly shared between different parts of the memory system, costing clock cycles and power. Even though there is no actual coherency to be maintained, the MESI protocol invalidates the cache line, forcing it to be re-loaded on each write. However, the cache-to-cache migration capability of multi-core clusters reduces the overhead. Therefore, you must avoid having cores operating on independent data that is stored within the same cache line and increasing the level of detail for inner loop parallelization.

19.5.5 Deadlock and livelock

When writing code that includes critical sections, it is important to be aware of common problems that can break correct execution of the program:

- *Deadlock* is the situation where two (or more) threads are each waiting for another thread to release a resource. Such threads are effectively blocked, waiting for a lock that can never be released.
- *Livelock* occurs when multiple threads are able to execute, without blocking indefinitely (the deadlock case), but the system as a whole is unable to proceed, because of a repeated pattern of resource contention.

Both deadlocks and livelocks can be avoided either by correct software design, or by use of lock-free software techniques.

19.6 Synchronization mechanisms in the Linux kernel

When porting software from a single core environment to run on multi-core cluster, there can be situations where you must modify code to enforce a particular order of execution or to control parallel access to shared peripherals or global data. The Linux kernel (like other operating systems) provides a number of different synchronization primitives for this purpose. Most such primitives are implemented using the same architectural features as application-level threading libraries like Pthreads. Understanding which of these is best suited for a particular case will give software performance benefits. Serialization and multiple threads contending for a resource can cause suboptimal use of the increased processing throughput provided by the multiple cores. In all cases, minimizing the size of the critical section provides best performance.

19.6.1 Completions

Completions are a feature provided by the Linux kernel that can be used to serialize task execution. They provide a lightweight mechanism with limited overhead that essentially provides a flag to signal completion of an event between two tasks. The task that is waiting can sleep until it receives the signal, using `wait_for_completion (struct completion *comp)` and the task that is sending the signal typically uses either `complete (struct completion *comp)`, that will wake up one waiting process, or `complete_all (struct completion *comp)` that wakes all processes that are waiting for the event. Kernel version 2.6.11 added support for completions that can time out and for interruptible completions.

19.6.2 Spinlocks

A spinlock provides a simple binary locking mechanism, designed for protection of critical sections. It implements a busy-wait loop. A spinlock is a generic synchronization primitive that can be accessed by any number of threads. More than one thread might be spinning for obtaining the lock. However, only one thread can obtain the lock. The waiting task executes `spin_lock (spinlock_t *lock)` and the signaling task uses `spin_unlock(spinlock_t *lock)`. Spinlocks do not sleep and disable pre-emption.

19.6.3 Semaphores

Semaphores are a widely used method to control access to shared resources, and can also be used to achieve serialization of execution. They provide a counting locking mechanism that can cope with multiple threads attempting to lock. They are designed for protection of critical sections and are useful when there is no fixed latency requirement. However, where there is a significant amount of contention for a semaphore, performance will be reduced. The Linux kernel provides a straightforward API with functions `down(struct semaphore *sem)` and `up(struct semaphore *sem)`; to lower and raise the semaphore.

Unlike spinlocks, which spin in a busy wait loop, semaphores have a queue of pending tasks. When a semaphore is locked, the task yields, so that some other task can run. Semaphores can be binary (in which case they are also mutexes) or counting.

19.6.4 Lock-free synchronization

The use of lock-free data structures, such as circular buffers, is widespread and can avoid the overheads associated with spinlocks or semaphores. The Linux kernel also provides two synchronization mechanisms that are lock-free, the *Read-Copy-Update* (RCU) and seqlocks. Neither of these mechanisms is normally used in device drivers.

If you have multiple readers and writers to a shared resource, using a mutex might not be very efficient. A mutex would prevent concurrent read access to the shared resource because only a single thread is permitted inside the critical section. Large numbers of readers might delay a

writer from being able to update the shared resource. RCUs can help in the case where the shared resource is mainly accessed by readers. Reader threads execute with little synchronization overhead. A thread that writes the shared resource has a much higher overhead, but is executed relatively infrequently. The writer thread must make a copy of the shared resource (access to shared resources must be done though pointers). When the update is complete, it publishes the new data structure, so that it is visible to all readers. The original copy is preserved until the next context switch on all cores. This guarantees that all current read operations can complete. RCUs are more complex to use than standard mutexes and are typically used only when traditional solutions are not suitable. Examples include shared file buffers or networking routing tables and garbage collection.

Seqlocks are also intended to provide quick access to shared resources, without use of a lock. They are optimized for short critical sections. Readers are able to access the shared resource with no overhead, but must explicitly check and re-try if there is a conflict with a write. Writes, of course, still require exclusive access to the shared resource. They were originally developed to handle things like system time, a global variable that can be read by many processes and is written only by a timer-based interrupt (on a frequent basis, obviously) The timer write has a high priority and a hard deadline, in order to be accurate. Using a seqlock instead of a mutex enables many readers to share access, without locking out the writer from accessing the critical section.

19.7 Profiling in SMP systems

ARM multi-core processors contain additional performance counter functions, that enable counting of the following SMP cache events:

- Coherent linefill missed in all cores.
- Coherent linefill hit in other core caches.

ARM DS-5 Streamline configures a default set of hardware performance counters that are a best-fit for optimizing applications. See *DS-5 Streamline* on page 16-4 for more information.

Chapter 20
Power Management

Many ARM systems are mobile devices, powered by batteries. In such systems, optimization of power usage (in fact, it would be more accurate to look at total energy 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 cores 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 core 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 you move to more advanced manufacturing processes – 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 core clock speed and the numbers of transistors that change state per cycle. Clearly, higher clock speeds and more complex cores 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 attempting 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 examples of these techniques are:

- *Idle management* on page 20-3.
- *Hotplug* on page 20-6.

- *Dynamic Voltage and Frequency Scaling* on page 20-7.

Performance management for something like a big.LITTLE system intersects directly with power management. *Operating System Power Management* (OSPM) might have to turn cores and clusters on and off as it transfers computation from a big core to a LITTLE one, or from a LITTLE core to a big one.

20.1 Idle management

When a core is idle the 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 will be 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 will be 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 becomes 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, when a core or cluster has been placed into a low power state, it can be reactivated at any time by a core wakeup event. That is, an event that can wake up a core from a low power state, such as an 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.

20.1.1 Power and clocking

One way in which you can reduce energy usage is to remove power, which removes both dynamic and static currents (sometimes called *power gating*) or to stop the clock of the core which removes dynamic power consumption only and can be referred to as *clock gating*.

ARM cores typically support a number of levels of power management, as follows:

- *Standby*.
- *Power down* on page 20-4.

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.

20.1.2 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 core 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 Synchronization 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, instruction pipeline or NEON blocks 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.

20.1.3 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.

20.1.4 Power down

In this state the core is powered off. Software on the device must 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.

For power down states, the interface requires a return address. This is the address at which the calling OS expects resumption of execution on wakeup at its level of privilege. From a powered down state, the core will restart at the reset vector, typically in secure mode. After initializing, the Secure world must resume the OS that called the power down interface, at the required return address.

20.1.5 Dormant mode

In dormant mode, the core logic is powered down, but the cache RAMs are left powered up. Often the RAMs will be 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 that permits individual cores within the cluster 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. Because 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.

20.2 Hotplug

CPU hotplug is a technique that can dynamically switch cores on or off. Hotplug can be used by the OSPM to change available compute capacity based on current compute requirements. Hotplug is also sometimes used for reliability reasons. There are a number of differences between hotplug and use of a power-down state for idle:

1. When a core is hot unplugged, the supervisory software stops all use of that core in interrupt and thread processing. The core is no longer considered to be available by the calling OS.
2. The OSPM has to issue an explicit command to bring a core back online, that is, hotplug a core. The appropriate supervisory software will only start scheduling on or enabling interrupts to that core after this command.

Operating systems typically perform much of the kernel boot process on one primary core, bringing secondary cores online at a later stage. Secondary boot behaves very similarly to hotplugging a core into the system.The operations in both cases are almost identical.

20.3 Dynamic Voltage and Frequency Scaling

Many systems operate under conditions where their workload is very variable. It would be useful to have the ability to reduce or increase the core performance to match the expected core workload. If you could clock the core more slowly when it is less busy, you could save dynamic power consumption.

Dynamic Voltage and Frequency Scaling (DVFS) is an energy saving technique that exploits:

- The linear relationship between power consumption and operational frequency.
- The quadratic relationship between power consumption and operational voltage.

This relationship is given as:

$P = C \times V^2 \times f$

Where:

P Is the dynamic power.

C Is the switching capacitance of the logic circuit in question.

V Is the operational voltage.

f Is the operational frequency.

Power savings are achieved by modulating the frequency of a core in proportion to its current load. Some frequencies can *be served at* lower voltages and a net quadratic power saving is achieved.

If the core is running more slowly, it is also the case that its supply voltage can be reduced somewhat. The advantage of reducing supply voltage is that it reduces both dynamic and static power. Compared to the alternative of running fast, then entering standby, then running fast and so forth, the approach of running slowly at a lower supply can save energy. To do this successfully requires two difficult engineering problems to be solved. The SoC requires a way in which software running on the ARM core can reliably modify the clock speed and supply voltage of the core, without causing problems in the system. This requires such things as voltage level-shifters and split power supplies on chip to cope with the variable supply, plus synchronizers between voltage domains to cope with timing changes. Of equal importance is the ability of software running in the system to make accurate predictions about future workloads to set the voltage and clock speed accordingly.

There is an implementation-specific relationship between the operational voltage for a given circuit and the range of frequencies that circuit can safely operate at. A given frequency of operation together with its corresponding operational voltage is expressed as a tuple and is known as an *Operating Performance Point* (OPP). For a given system, the range of attainable OPPs are collectively termed as the system DVFS curve.

Operating systems use DVFS to save energy and, where necessary, keep within thermal limits. The load on a core modulates its frequency of operation. The OS provides DVFS policies to manage the power consumed and the required performance. A policy aimed at high-performance selects higher frequencies and uses more energy. A policy aimed at saving energy selects lower frequencies and therefore results in lower performance.

20.4 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 core is not required to take any specific action when it executes them. In the Cortex-A 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 core is significantly reduced – only static leakage currents will be drawn, and there will be no dynamic power consumption.

The `WFI` instruction has the effect of suspending execution until the core 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 core 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, 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.

20.5 Power State Coordination Interface

The ARM architecture provides a convenient method of partitioning a software stack through its architectural definition of privilege levels, and through the Security and Virtualization Extensions. This introduces different modes of execution in the architecture, which in turn provide a way to partition the systems that compose the software stack used on a device. As we have seen, the ARMv7 architecture has the following privilege levels in which software from different software vendors can operate:

- PL0 - For User Application vendors, such as apps downloaded from an App Store.
- PL1 - Rich OS vendors, such as the Linux Kernel used by Android.
- PL2 - Hypervisor Vendors.
- Secure PL0 - Trusted OS applications, from Trusted OS vendors.
- Secure PL1 - Trusted OS.
- Secure PL1 - OEMs providing secure firmware.

As operating systems from different vendors can be simultaneously executing in an ARM system, it is necessary to have a method of collaboration when performing power control. This means that, if the operating system that is managing power, be it at supervisor (PL1) or hypervisor level (PL2), wants to enter an idle state, power up/down a core, or perform a big.LITTLE migration, operating systems at other levels of privilege must react to this request. Equally, if a core is roused from a power state by a wake up event, it might be necessary for operating systems running at different levels of privilege to perform actions such as restoring state. There are no freely available interfaces that allow interoperation, and integration amongst the various operating systems. This presents difficulties for the OS vendors.

ARM provides a software interface, the *Power State Coordination Interface* (PSCI) so that the OSPM can place a core into a low power state when it has no work for it. Using this interface operating systems and firmware can implement power management techniques such as idle, hotplug and state migration on big.LITTLE systems. The messages sent using this interface are received by all relevant levels of execution. That is, if Virtualization and Security Extensions are implemented, a message sent by a Rich OS must be received by a hypervisor. If the latter sends it, the message must be received by the secure firmware that then coordinates with a Trusted OS. This enables each operating system to determine whether context saving is required.

PSCI specifies the following functions:

CPU_SUSPEND

Suspend the execution on a core. This call is intended for use in idle subsystems where the core is expected to return to execution through a wake up event.

CPU_OFF

Power down a core. This call is intended for use in hotplug. A core that is powered down by CPU_OFF can only be powered up again by a CPU_ON.

CPU_ON Power up a core. This call is used to power up cores that either:

- Have not yet been booted into the calling OS.
- Have been previously powered down with a CPU_OFF call.

MIGRATE

This is used to request the Trusted OS of a single core to migrate its context to a specific core.

Chapter 21
Security

The term *security* is used in the context of computer systems to cover a wide variety of features. In this chapter, we will use a narrower definition. A secure system is one that protects assets (resources that require protecting, for example passwords, or credit card details) from a range of plausible attacks to prevent them from being copied or damaged or made unavailable (denial of service). Confidentiality is a key security concern for assets such as passwords and cryptographic keys. Defense against modification and proof of authenticity is vital for security software and on-chip secrets used for security. Examples of secure systems might include entry of Personal Identification Numbers (PINs) for such things as mobile payments, digital rights management, and e-Ticketing.

Security is harder to achieve in the world of open systems where a wide range of software can be downloaded onto a platform. This gives the potential for malevolent or untrusted code to tamper with the system.

ARM processors include specific hardware extensions to enable construction of secure systems. Creating secure systems is outside the scope of this book. In the remainder of this chapter, we present the basic concepts behind the ARM Security Extensions (TrustZone). If your system is one that makes use of these extensions, be aware that this imposes some restrictions on the operating system and on unprivileged code (in other words, code that is not part of the secure system). TrustZone is of little or no use without memory system support.

It must, of course, be emphasized, that no security is absolute!

21.1 TrustZone hardware architecture

The TrustZone hardware architecture aims to provide resources that enable a system designer to build secure systems. It does this through a range of components and infrastructure additions. Low-level programmers must have some awareness of the restrictions placed on the system by the TrustZone architecture, even if they are not intending to make use of the security features.

In essence, system security is achieved by dividing all of the device's hardware and software resources, so that they exist in either the Secure world for the security subsystem, or the Normal world for everything else. System hardware ensures that no Secure world resources can be accessed from the Normal world. A secure design places all sensitive resources in the Secure world, and has robust software running that can protect assets against a wide range of possible attacks.

The use of the term *Non-Secure* is used in the *ARM Architecture Reference Manual* as a contrast to *Secure* state, but this does not imply that there is a security vulnerability associated with this state. We will refer to this as *Normal* operation here. The use of the word *world* is to emphasize the relationship between the Secure world and other states the device is capable of.

The additions to the architecture enable a single physical core to execute code from both the Normal world and the Secure world in a time-sliced fashion. The memory system is similarly divided. An additional bit, indicating whether the access is Secure or Non-Secure (the NS bit) is added to all memory system transactions, including cache tags and access to system memory and peripherals. This can be considered as an additional address bit, giving a 32-bit physical address space for the Secure world and a completely separate 32-bit physical address space for the Normal world.

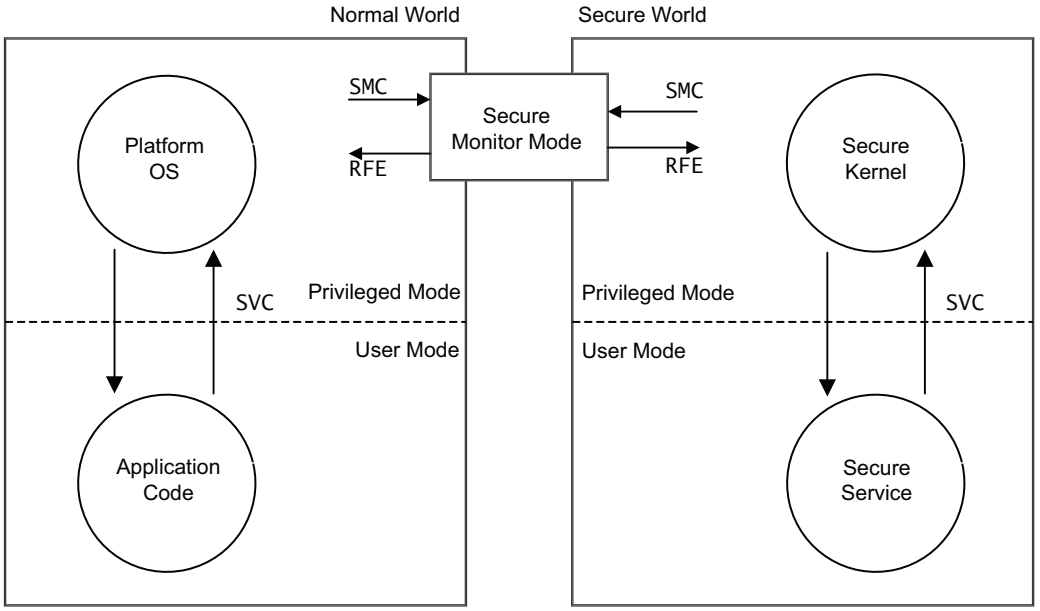


Figure 21-1 Switching between Normal and Secure worlds

As the cores execute code from the two worlds in a time-sliced fashion, context switching between them is done using an additional core mode (like the existing modes for IRQ, FIQ etc.) called Monitor mode. A limited set of mechanisms by which the core can enter Monitor mode from the Normal world is provided. Entry to monitor can be through a dedicated instruction, the Secure Monitor Call (`SMC`) instruction, or by hardware exception mechanisms. IRQ, FIQ and

external aborts can all be configured to cause the core to switch into Monitor mode. In each case, this will appear as an exception to be dealt with by the Monitor mode exception handler. Figure 21-1 on page 21-2 provides a conceptual summary of this switching.

Figure 21-2 shows how, in many systems, FIQ is reserved for use by the secure world (it becomes, in effect, a non-maskable secure interrupt). An IRQ that occurs when in the Normal world is handled in the normal way, described in the chapters on exception handling. An FIQ that occurs while executing in the Normal world is vectored directly to Monitor mode. Monitor mode handles the transition to Secure world and transfers directly to the Secure world FIQ handler. If the FIQ occurs when in the Secure world, it is handled through the Secure vector table and routed directly to the Secure world handler. IRQs are typically disabled during execution in the Secure world.

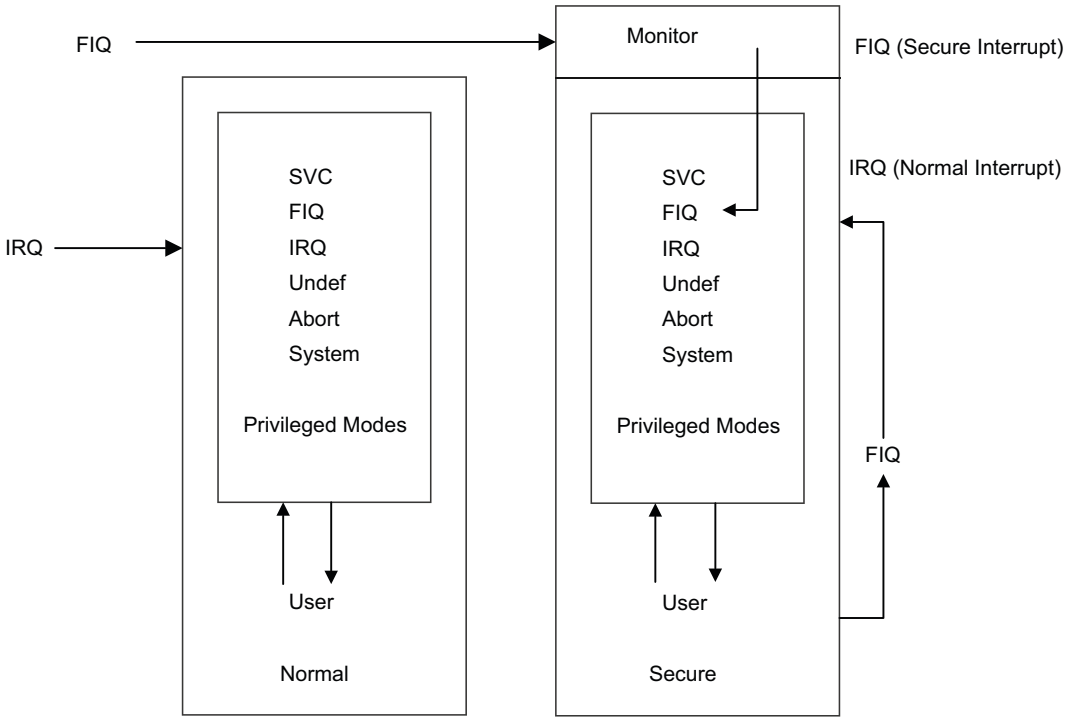


Figure 21-2 Normal and Secure worlds

The software handler for Monitor mode is implementation specific, but will typically save the state of the current world and restore the state of the world being switched to, much like a normal context switch.

The NS bit in the *Secure Configuration Register* (SCR) in CP15 indicates which world the core is currently in. In Monitor mode, the core is always executing in the Secure world, regardless of the value of the SCR NS-bit that is used to signal which world you were previously in. The NS-bit also enables code running in Monitor mode to snoop security banked registers, to see what is in either world.

TrustZone hardware also effectively provides two virtual MMUs, one for each virtual core. This enables each world to have a local set of translation tables, with the Secure world mappings hidden and protected from the Normal world. The translation table descriptions include a NS bit that is used to determine whether accesses are made to the secure or non-secure physical address space. Although the translation table entry bit is still present, the Normal virtual core hardware

does not use this field, and memory accesses are always made with NS = 1. The Secure virtual core can therefore access either Secure or Normal memory. Cache and TLB hardware permits Normal and Secure entries to co-exist.

It is good practice for code that modifies translation table entries and that does not care about TrustZone based security, to always set the translation table NS-bit to zero. This means that it will be equally applicable when the code is executing in the Secure or Normal worlds.

The ability to direct aborts, IRQ and FIQ directly to the monitor, enables trusted software to route the interrupt request accordingly, enabling a design to provide secure interrupt sources immune from manipulation by the Normal world software. Similarly, the Monitor mode routing means that from the point of view of Normal world code, an interrupt that occurs during Secure world execution appears to occur in the last Normal world instruction that occurred before the Secure world was entered.

A typical implementation is to use FIQ for the Secure world and IRQ for the Normal world. Exceptions are configured to be taken by the current world (whether Secure or Normal), or to cause an entry to the monitor. The monitor has its own vector table. Because of this, the core has three sets of exception vector tables. It has a table for the Normal world, one for the Secure world, and one for Monitor mode.

The hardware must also provide the illusion of two separate cores within CP15. Sensitive configuration CP15 registers can only be written by Secure world software. Other settings are normally banked in the hardware, or by the Monitor mode software, so that each world sees its own version.

Implementations that use TrustZone will typically have a light-weight kernel (Trusted Execution Environment) that hosts services (for example, encryption) in the Secure world. A full OS runs in the Normal world and is able to access the secure services using the `SMC` instruction. In this way, the Normal world gets access to functions of the service, without any ability to see keys or other protected data.

21.1.1 Multi-core systems with security extensions

Each core in a multi-core system has the programmer's model features described for single cores earlier in this book. Any number of the cores in the cluster can be in the Secure world at any point in time, and cores are able to transition between the worlds independently of each other. The Snoop Control Unit is aware of security settings. Additional registers are provided to control whether Normal world code can modify SCU settings. Similarly, the generic interrupt controller that distributes prioritized interrupts across the Multi-core cluster must also be modified to be aware of security concerns.

Theoretically, the Secure world OS on an SMP system could be as complicated as the Normal world OS. However, this is undesirable when aiming for security. In general, it is expected that a Secure world OS will actually only execute on one core of an SMP system (with security requests from the other cores being routed to this chosen core). This does provide some bottleneck issues. To some extent these will be balanced by the Normal world OS performing load balancing against the core that it will see as busy for unknown reasons. Beyond that this limitation has to be seen as one of the compromises that can be reached to hit a particular target level of security.

21.1.2 Interaction of Normal and Secure worlds

If you are writing code in a system that contains some secure services, it can be useful to understand how these are used. As we have seen, a typical system will have a light-weight kernel, *Trusted Execution Environment* (TEE) hosting services (for example, encryption) in the

Secure world. This interacts with a full OS in the Normal world that can access the secure services using the SMC call. In this way, the Normal world is able to have access to functions of the service, without getting to see keys (for example).

Generally applications developers won't directly interact with TrustZone (or TEEs or Trusted Services). Instead, one makes use of a high level API (for example, it might be called `reqPayment()`) provided by a Normal world library. The library would be provided by the same vendor as the Trusted Service (for example, a credit card company), and would handle the low level interactions. Figure 21-3 shows this interaction and illustrates the flow from user application calling the API that makes an appropriate OS call, which then passes to the TrustZone driver code, and then passes execution into the TEE through the Secure monitor.

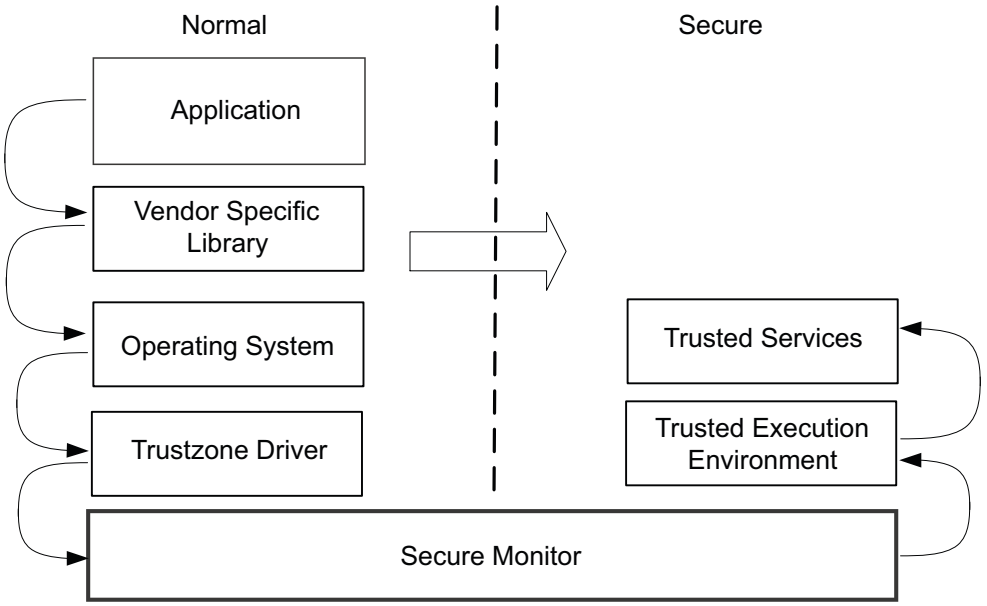


Figure 21-3 Interaction with TrustZone

It is common to share data between the Secure and Normal worlds. For example, in the Secure world you might have a signature checker. The Normal world can request that the Secure world verifies the signature of a downloaded update, using the `SMC` call. The Secure world requires access to the memory used by the Normal world to store the package. The Secure world can use the NS-bit in its translation table descriptors to ensure that it used non-secure accesses to read the data. This is important because data relating to the package might already be in the caches, because of the accesses done by the Normal world. These accesses with addresses marked as non-secure. As mentioned previously the security attribute can be thought of as an additional address bit. If the core used secure access to try to read the package, it would not hit on data already in the cache.

If you are a Normal world programmer, in general, you can ignore something happening in the Secure world, as its operation is hidden from you. One side effect is that interrupt latency can increase slightly, if an interrupt goes off in the Secure world, but this increase is small compared to the overall latency on a typical OS.

If you do have to access a secure application, you will require a driver-like function to talk to the Secure world OS and Secure applications, but the details of creating that Secure world OS and applications are beyond the scope of this book. Those writing code for the Normal world only require the particular protocol for the secure application being called.

Finally, the TrustZone System also controls availability of debug provision. Separate hardware over full JTAG debug and trace control is separately configurable for Normal and Secure software worlds, so that no information about the Secure system leaks.

Chapter 22
Virtualization

Modern compute subsystems are powerful but often under utilized. The domains that these systems address increasingly require multiple software environments working simultaneously on the same physical processor systems. Software applications and their environments might have to be completely separated for reasons of robustness, differing requirements for real-time behavior, or to have them isolated from each other. This is done by providing *virtual* cores for the software to execute on.

Implementing such virtual cores in an efficient fashion requires both dedicated hardware extensions (to accelerate switching between virtual machines) and hypervisor software. The volume of legacy software to be ported to new hardware might make it more expedient to provide a virtual machine matching the legacy OS requirements than to port the OS and drivers to new hardware and OS versions. The Virtualization Extensions to the ARMv7 architecture provide a standard hardware accelerated implementation enabling the creation of high performance hypervisors. Writing a hypervisor is outside the scope of this book, as such software is typically produced by specialized teams. In this chapter, we will provide an overview of how the Virtualization Extensions work and the effects of this on OS and application code, in a fashion similar to our previous discussion of the TrustZone Security Extensions.

The term hypervisor actually dates back to the 1960s and was first used by IBM to describe software running on mainframes. Today's hypervisors are conceptually very similar and might be thought of as operating at a level higher than the supervisor or operating system. They typically permit multiple instances of one or more different operating systems, called Guest Operating Systems (*Guest OS*), to run on the system. The hypervisor provides a virtual system to each of the Guest operating systems and monitors their execution. The Guest OS does not typically need to be aware that it is running under a hypervisor or that other operating systems might be using the system. The term *Virtual Machine Monitor* (VMM) is sometimes used to describe a hypervisor.

There are two types of hypervisors being deployed on ARM. Type1 is as described with each Virtual Machine (VM) containing Guest OS. In Type 2, the hypervisor is an extension of the Host OS with each subsequent Guest OS contained in a separate VM.

One possible use of virtual machine hypervisors is in so-called cloud computing solutions, where software can be partitioned into client or server devices or where large amounts of data or content exist. This scenario is likely to increase the amount of addressable physical memory required in the system. For this reason, the Virtualization Extensions require that the core also implement the *Large Physical Address Extension* (LPAE) described on page 22-10. This enables each of the multiple software environments, as well as different processes within, to access separate windows of physical addresses. LPAE provides an additional level of MMU translation table, so that each 32-bit virtual memory address can be mapped within a 40-bit physical memory range. In this scenario, this permits software to allocate enough physical memory to each virtual machine, even when total demands on memory exceed the range of 32-bit addressing. It would also be possible for a single operating system kernel to handle up to 40 bits of physical address space, with up to 4GB available at any given time. In theory, this can mean up to ~3GB per process.

22.1 ARMv7-A Virtualization Extensions

For the most part, this book assumes that a system is owned and managed by a single privileged Operating System that deploys many unprivileged applications. Most main stream Operating Systems are also built on this assumption. Virtualization is the concept where more than one Operating System is enabled to co-exist and operate on a same system. The ARM Virtualization Extensions make it possible to operate multiple Operating Systems on the same system, while offering each such Operating System an illusion of sole ownership of the system by virtue of introduction of new architectural features. These are:

- A hypervisor mode, in addition to the existing privileged modes. This PL2 mode is even more privileged than PL1 modes. Hyp mode is expected to be occupied by Hypervisor software managing multiple guest operating systems occupying PL1 and PL0 modes. Hyp mode only exists in the Normal (Non-secure) world.
- An additional memory translation, called Stage 2 is introduced. Previously, the Virtual Address (VA) is translated to Physical Address (PA) by the PL1 and PL0 MMU described in Chapter 9. This translation is now known as Stage 1 and the old Physical Address is now called Intermediate Physical Address (IPA). The IPA is subjected to another level of translation in Stage 2 to get the final PA corresponding to the VA.
- Interrupts can be configured to be taken (routed) to the Hypervisor. The hypervisor will then take care of delivering interrupts to the appropriate guest.
- A Hypervisor Call instruction (HVC) for guests to request Hypervisor services.

ARM Virtualization Extensions aim to run conventional Operating Systems as guests on a virtualized system with no or little modification.

22.1.1 Types of virtualization

Virtualization solutions can be broadly classified as *bare metal* or *hosted* according to their design. Regardless of the classification, the functional role of a Hypervisor remains the same, arbitration of platform resources, and seamless operation of individual guests with minimal porting effort and run time sacrifices.

In bare metal virtualization, the Hypervisor is the most privileged non-secure software to boot on the core. It initializes various aspects of PL2 for housekeeping, for example its own translation tables for PL2 translation regime. The Hypervisor will then initialize settings for each guest operating system that it will launch. This will involve setting up stage 2 translation tables for individual guests, interrupt delivery mechanisms etc., and goes on to launching them. The Hypervisor will schedule guest operating systems on available cores, handle exceptions on behalf of the guests, and deliver interrupts to appropriate guests at run time.

Figure 22-1 on page 22-4 contrasts conventional systems with systems deploying bare-metal virtualization.

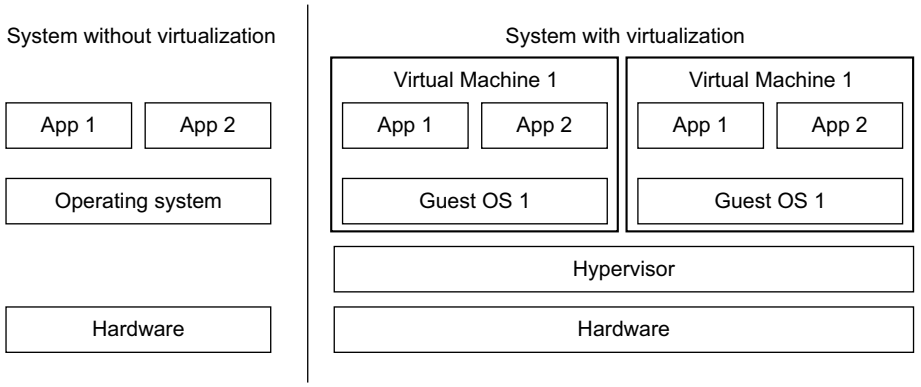


Figure 22-1 Bare metal virtualization

22.1.2 Memory translation

A number of memory translation regimes are possible in a virtualized system in the Normal world. A translation regime is a broad term encompassing the privileges and execution mode of the core and the set of translation tables used. These translations are carried out using the MMUs and the Translation Table structures created by the software that controls the translation. A stage of translation comprises of a set of translation tables that translates an input address to an output address. The input and output addresses take different names depending on the stage of translation.

PL1&0 Stage 1

This is the translation regime that an Operation System will setup and control, exactly as described in Chapter 9 *The Memory Management Unit*. This translation regime is applied when the core is in one of execution modes that falls under PL1 (Kernel) and PL0 (application) privilege levels. On a conventional system, this regime is used to translate virtual addresses to physical addresses.

In a virtualized system however, this physical address is treated as an *Intermediate Physical Address* (IPA) because it is subjected to another stage (Stage 2) of translation. In the presence of an oncoming Stage 2, this translation regime qualifies as Stage 1.

IPAs cannot be used to address system memory. Although called an intermediate physical address, from the perspective of a guest, the output of this translation regime is what it sees and uses as physical address.

PL1&0 Stage 2

This comprises of a set of translation tables that the Hypervisor sets up for each of the guests it manages. This regime will translate the IPA that was output by Stage 1, and translates it to a physical address that can finally be used to address system memory.

The Virtualization Extensions add a set of core registers to control the Stage 2 translation tables. The hypervisor saves and restores these registers whenever it schedules a different guest on the core. Individual guests that are managed by the Hypervisor have no control over, nor are aware of the presence of a Stage 2 translation. When virtualization is in effect, all PL1&0 Stage 1 translations are implicitly subjected to this Stage 2 translation. It is also important to note that a Stage 2 translation is applied even if Stage 1 translation is turned off by the guest, As a result, Stage 2 translation is always applied when virtualization is in effect.

Because it is managed by the Hypervisor, exceptions arising from Stage 2 translation are taken in Hyp mode so that Hypervisor can respond to them. Exceptions from Stage 1, as before, are taken to PL1 Abort mode for the respective guest to handle.

What Stage 2 translation achieves is virtualization of the guest's view of physical memory. Every virtual address used in PL1&0 stage is first translated to an IPA by Stage 1. This IPA is translated to the actual physical address by Stage 2. The guest is unaware of, and cannot control, translation by Stage 2. By appropriately setting up the Stage 2 translation tables the hypervisor can manage and allocate physical memory to guests. This can be thought of a natural extension of what an operating system does, managing and allocating physical memory to its applications. In that respect, the guests could be considered as applications of the Hypervisor.

PL2 This comprises of a set of translation tables that the Hypervisor sets up for itself to manage its own virtual memory. This regime will translate virtual addresses used in Hyp mode to Physical Addresses. No additional stages are applied to this translation, therefore it is not qualified by stages. The Virtualization Extension includes a set of registers for the Hypervisor to manage its own translation tables, much like an operating system kernel manages its own translation tables

22.2 Hypervisor software

This section lists some of the functions that a Hypervisor performs. As we have seen, these functions do not depend on the type of virtualization solution deployed.

22.2.1 Memory management

The hypervisor is responsible for memory management for itself and for the guest operating systems it manages. The entire physical memory is at the direct disposal of the Hypervisor.

The PL2 MMU is used in Hyp mode to translate the virtual addresses that the Hypervisor uses to address physical memory. Apart from setting up and managing its own translation tables, a Hypervisor has to create and manage Stage 2 translation tables for each of its guests.

In addition to both Secure and Non Secure versions of the TTBR, there are also a *Virtualization Translation Table Base Register* (VTTBR) and *Hyp Translation Table Base Register* (HTTBR) where the VTTBR is used to point to second stage translation tables (See *Large Physical Address Extensions* on page 22-10) and the HTTBR is used to point to tables used for Hypervisor memory mapping. Secure functions are the same as the Non-secure TTBR and can use the short-descriptor format, but VTTBR and HTTBR are long descriptor only.

Stage 2 translation tables setup by the Hypervisor translate intermediate physical memory addresses to physical memory addresses. Any aborts resulting from attempting to translate addresses in Stage 2 are taken in Hyp mode. The Hypervisor is responsible for receiving the aborts, and handling them appropriately. For intended and legitimate faults, the Hypervisor might take remedial measures such as emulating a device or allocating more memory to a guest operating systems. For unexpected faults the Hypervisor can choose to terminate the guest operating system or report aborts to the guest in turn.

22.2.2 Device emulation

Platform devices are memory-mapped, and guest accesses to devices are subject to at least Stage 2 translation when virtualization is in effect. When Virtualization is in effect, there are use cases for a Hypervisor to emulate a device in software, or to hide it from the guests.

Device emulation is necessary where more than one guest is aware of a platform device (using the physical address), and attempt to access it. Because of the shared nature, guests cannot be permitted direct accesses to it without arbitration. In such cases the Hypervisor can prohibit access to the said device region to all such guests, by means of their respective stage 2 translation table descriptors.

Alternatively, the hypervisor might choose to hide a device from a chosen set of guests either because the device is not actually present or has already been assigned to a different guest. (Guests would typically detect a platform device by reading its ID register.) By employing the same trapping mechanism as before, the Hypervisor can return dummy values for guest reads and ignore writes, effectively giving the guest the impression that the device does not exist on the platform.

One of the important jobs of the Hypervisor is to schedule guests on available cores in the system. The task is accomplished in much the same way that an operating system schedules different tasks. In a virtualized system the Hypervisor schedules a guest operating system on a core to execute. A scheduled guest assumes sole ownership of the core, and schedules its tasks on the core, as it would do on a conventional system. Guests will not be aware of the fact that a Hypervisor is functioning and that they are being constantly scheduled in and out of the core.

22.2.3 Device assignment

Device emulation is necessary but turns out to be expensive as all accesses to the device by the guest have to be trapped and emulated in software. The Hypervisor has the option of assigning individual devices to individual guests so that the guest can own and operate the device without requiring Hypervisor arbitration. The challenge is to hide from the guest the fact that the device is actually located at a different physical address, and generates a different interrupt ID than that which the guest is expecting. Transparent Stage 2 mappings, and interrupt virtualization can circumvent these challenges.

22.2.4 Exception handling

Asynchronous exceptions (IRQ, FIQ and asynchronous aborts) can be routed to Hypervisor mode. This behavior can be selected through dedicated control bits in the *Hyp Configuration Register* (HCR). An additional bit in this register also enables synchronous exceptions (Undef, SVC or precise aborts) to be routed to the Hypervisor Trap entry. The hypervisor can also be called through the `HVC` instruction. If this is performed while in privilege level PL1, it causes a hypervisor trap exception and entry into Hypervisor mode. If performed when already in Hypervisor mode, it causes an HVC exception.

The Virtualization Extensions introduce the concept of virtual exceptions. The exceptions we have met before are real events occurring in the system. Virtual exceptions are events of similar nature, that is, interrupts and aborts, but are manufactured by the Hypervisor, either in response to an actual physical exception, or manufactured (as a result of Device emulation) with no relation to a physical exception. These are taken in their conventional exception mode, for example, a virtual abort is taken in Abort mode. Physical exceptions can be configured to be consumed by the Hypervisor, only virtual exceptions are delivered to guests.

This concept is useful only when the corresponding physical exception is routed to Hyp mode. This means that the hypervisor has control of masking and generating a virtual exception. When a real physical exception occurs, and is routed to the hypervisor software, the Hypervisor then signals a virtual exception to the current Guest OS. The Guest OS handles the exception as it would do for an equivalent physical exception and need not be aware that the hypervisor has been involved. The virtual exception is signalled through the virtual GIC, or by using dedicated HCR bits. When exceptions are routed to the hypervisor in this fashion, the CPSR A, I and F bits no longer mask physical exceptions, instead they mask the handling of virtual exceptions within the Guest OS.

22.2.5 Interrupt handling

When virtualization is in effect, interrupts reaching a core can be targeted to a guest that is running on a different core, or even to a guest that does not currently occupy a core at all (suspended because of inactivity). In a virtualized system the hypervisor must be the primary arbitrator, and configures the core so that interrupts delivered to a core are handled in Hyp mode. This enables the Hypervisor to be in charge of receiving and delivering interrupts to guests.

The *Hypervisor Configuration Register* (HCR) in the core, that is accessible to guests, has a set of three bits corresponding to IRQ, FIQ and abort virtual exceptions. When any of these bits are set while the core is in PL1&0 modes, This has the same effect as the core taking the corresponding exception in a conventional system, that is, they will not be taken in Hypervisor mode again. Physical exceptions take the core to Hyp mode.

If, after handling the exception, the hypervisor chooses to deliver the exception to the guest, it sets the required bits and resumes the guest, the guest, being in one of PL1&0 modes, will take the exception as if it were running standalone.

22.2.6 Scheduling

One of the important jobs of the hypervisor is to schedule guests on available cores in the system. This task is accomplished in much the same way that an operating system schedules different tasks. In a virtualized system the hypervisor schedules a guest on a core to execute. A scheduled guest assumes sole ownership of the core, and schedules its tasks on the core, as it would do on a conventional system. Guests are not aware of the fact that a Hypervisor is functioning beneath and is being constantly scheduled in and out of the core.

22.2.7 Context switch

When the Hypervisor schedules another guest on a core, it must perform a *context switch*, that is, save the context of the currently running guest to memory, and then restore the context of the new guest from memory. The goal is to recreate the environment for the new guest on the current core before it resumes, creating the illusion of uninterrupted execution. By performing a context switch, the Hypervisor ensures that the execution environment follows the guest, and offers the illusion of a virtual core that the guest always occupies.

The following elements of a guest context must be saved and restored:

- The general purpose registers of the core including the banked registers of all modes,
- System register contents for such things as memory management and access control.
- The pending and active states of private interrupts on the core.
- In case of guests using core private timers, the timer registers must be saved and restored so that they generate interrupts at the expected intervals.

The physical memory assigned to a guest, that it sees as RAM, stays in place and does not have to be saved or restored. By using two stages of memory translation, the physical memory that the guest uses stays private and distinct from any others, even for identical guests.

22.3 Relationship between virtualization and ARM Security Extensions

We discussed the concept of privilege levels, PL0, PL1 and PL2, in Chapter 3 *ARM Processor Modes and Registers*. Figure 22-2 illustrates how in the Normal world, you have PL0 (User mode), PL1 (for exception modes) and PL2 for the hypervisor, while in the Secure world, you have only PL0 and PL1, with secure monitor mode at PL1.

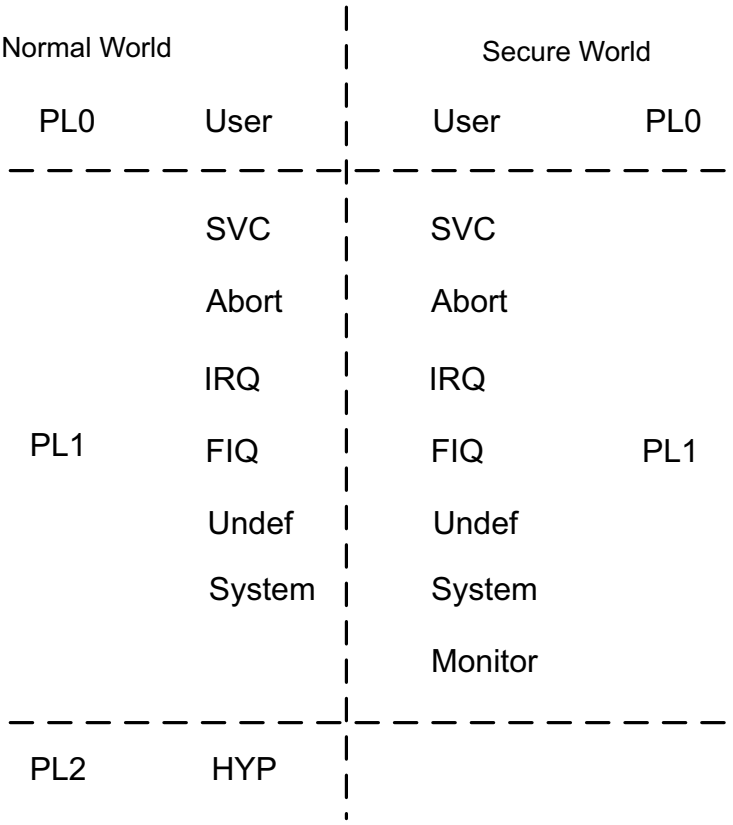


Figure 22-2 Privilege levels and security

In general, the hypervisor has no impact on the Secure world. It is unable to access secure parts of memory or otherwise interfere in the operations of the secure system.

22.4 Large Physical Address Extensions

Processors that implement the ARMv7-A *Large Physical Address Extension* (LPAE), expand the range of accessible physical addresses from 4GB (2^{32} bytes) to 1024GB (2^{40} bytes) a terabyte, by translating 32-bit virtual memory addresses into 40-bit physical memory addresses. To do this they use the *Long-descriptor* format. The existing short-descriptor format translation tables are still supported, as are the security extensions described in Chapter 21.

The Virtualization Extensions provide an additional second stage of address translation when running virtual machines. The first stage of this translation produces an *Intermediate Physical Address* (IPA) and the second stage then produces the physical address. The second stage of this conversion process is controlled by the Hypervisor, TLB entries can also have an associated *Virtual Machine ID* (VMID), in addition to an ASID. Again, it is possible to disable the stage 2 MMU and have a flat mapping from IPA to PA.

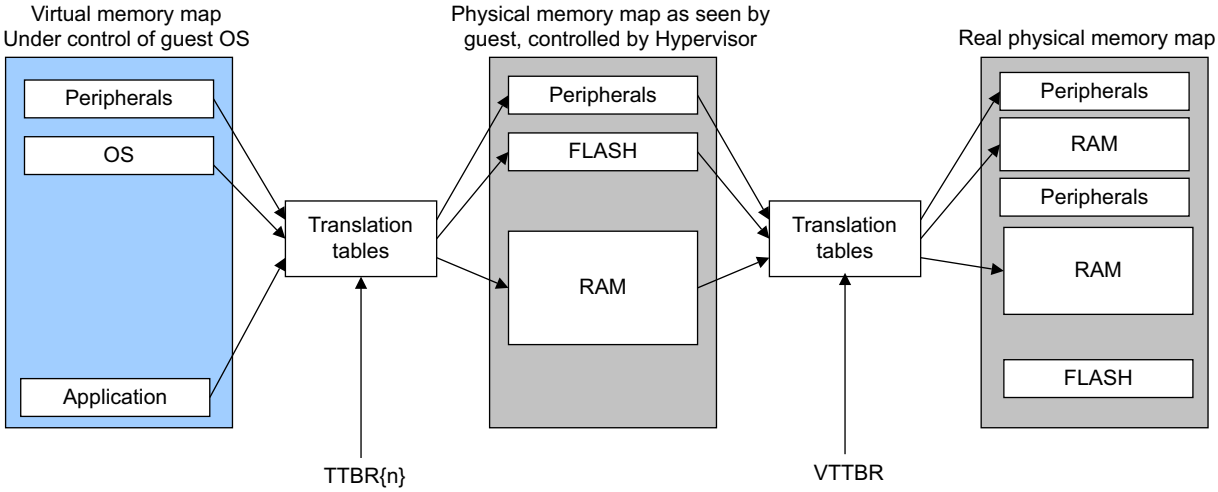


Figure 22-3 Stage 2 translation

Long-descriptor format memory management includes the following features:

- 64-bit page descriptors.
- Up to three levels of translation tables.
- Supports specifying up to 40-bit physical addresses.
- 1GB, 2MB and 4KB block or page sizes are supported.
- Optional second stage memory translation used in virtualization
- An additional access permission setting – *Privileged eXecute Never* (PXN). This marks a page as containing code that can be executed only in a non-privileged (user) mode. This setting is also added to the legacy descriptor format. There is also a privileged execute setting (PX) that means that code can be executed only in privileged mode.

There are certain similarities between the short-descriptor format and the long-descriptor format:

- Both have an input 32-bit virtual address from the processor.
- Both use TTBR0/TTBR1 to point to level 1 translation tables (with TTBCR selecting which part of the address space is covered by each).

- Long descriptors also provides a way to indicate “Outer-shareable” or “Inner-shareable” as opposed to “Shareable”.

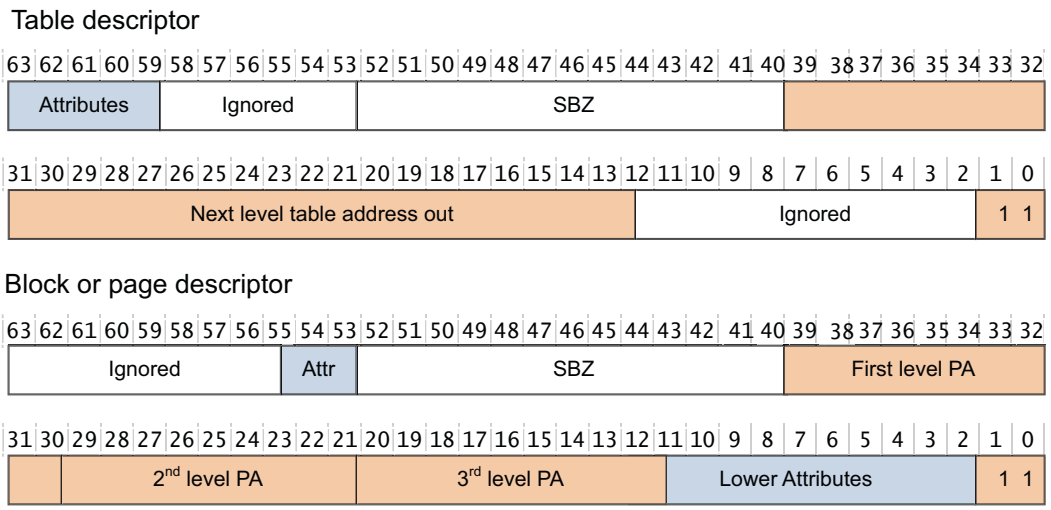


Figure 22-4 Format of long-descriptor table entries

22.4.1 Long descriptor translation tables

Long descriptor translation tables output a 40-bit intermediate physical address.

- The first level translation table is 4 entries – one entry for each 1 GB of virtual memory
  - Indexed by [31;30] of the VA.
- First level page descriptors contain either:
  - The upper 10 bits of physical address for that 1 GB of virtual memory
  - A pointer to the second level translation table
- Second level translation table is 512 entries – one entry for each 2MB of virtual memory (in the 1GB address range of the first level table entry).
  - Indexed by bits [29:21] of the VA.
- Second level page descriptors contain:
  - The upper 19 bits of physical address for that 2MB of virtual memory
  - A pointer to the third level translation table
- Third level translation table is 512 entries – one entry for each 4KB of virtual memory (in the 2MB address range of the second level table entry)
  - Indexed by bits [20:12] of the VA.
- Third level page descriptors contain:
  - Upper 28 bits of physical address for that 4KB of virtual memory.

This process is shown in Figure 22-5 on page 22-12.

The translation table walking steps are similar to those previously described for processors that do not implement LPAE. Individual translation table entries are now 64-bit in size.

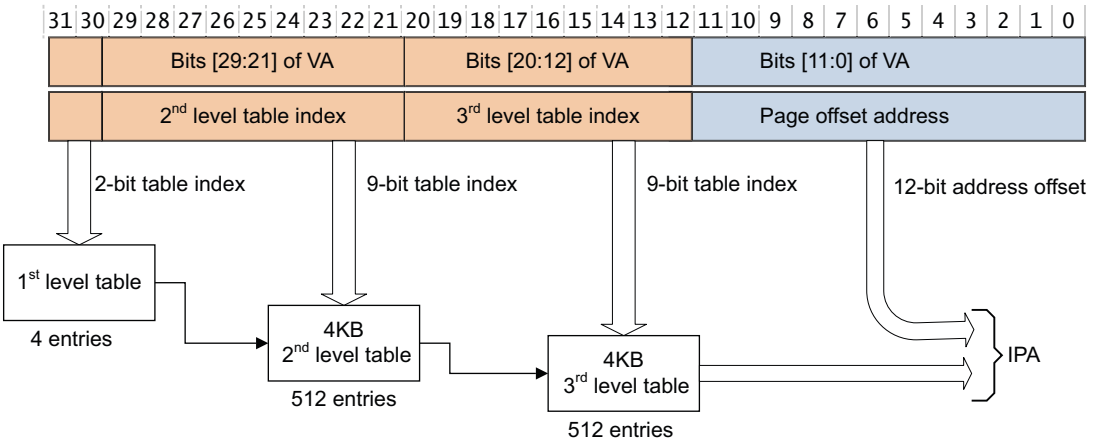


Figure 22-5 Long -descriptor table walk

Recall that long-descriptor translation tables output a 40-bit physical address. Therefore, the physical base address given by this translation will be of the form [39:0], with bit [39] of the descriptor producing bit [39] of the physical address. It is not necessary to map all of the physical address space, but all of the 4GB virtual address space has to be taken care of. There must be a descriptor for each virtual address – although it might be a "fault" descriptor.

Chapter 23
big.LITTLE

Modern software stacks place conflicting requirements on mobile systems. On the one hand is a demand for very high performance for tasks such as games, while on the other is a continuing requirement to be frugal with energy reserves for low intensity applications like audio playback.

Traditionally, it has not been possible to have a single processor design that can be capable of both high peak performance as well as high energy efficiency. This meant that a lot of energy was wasted because the high performance core would be used for low intensity tasks leading to reduced battery life. Performance would itself be affected by the thermal limits at which the cores could run for sustained periods.

big.LITTLE technology from ARM solves this problem by coupling together an energy efficient "LITTLE" core with a high performance "big" core. big.LITTLE is an example of a heterogeneous processing system. Such systems typically include several different processor types with different microarchitectures, like general purpose processors and specialized ASICs.

big.LITTLE takes the heterogeneity one step further in that it includes general purpose processors that are different in their micro-architecture but compatible in their instruction set architecture. A term that is often used with such systems is Heterogeneous Multi-processing (HMP). What makes HMP different from AMP is that all the processors in an HMP system are fully coherent and run the same operating system image.

The basic premise is that software can run on big or LITTLE processors depending on performance requirements. When peak performance is required software can run on big processors. At most other times, software can run on LITTLE processors. Through this combination, big.LITTLE provides a solution that is capable of delivering the high peak performance demanded by the latest mobile devices, within the thermal bounds of the system, with maximum energy efficiency.

23.1 Structure of a big.LITTLE system

Both types of core in a big.LITTLE system are fully cache coherent and share the same instruction set architecture (ISA). The same application binary will run unmodified on either. Differences in the internal microarchitecture of the processors enable them to provide the different power and performance characteristics that are fundamental to the big.LITTLE concept. These are typically managed by the operating system.

big.LITTLE software models require transparent and efficient transfer of data between big and LITTLE clusters. Hardware coherency enables this, transparently to the software, such as the ARM CoreLink CCI-400 described in *The Cache Coherent Interface (CCI)* on page 18-11. Without hardware coherency, the transfer of data between big and LITTLE cores would always occur through main memory - this would be slow and not power efficient. In addition, it would require complex cache management software, to enable data coherency between big and LITTLE clusters.

In addition, such a system also requires a shared interrupt controller, such as the GIC-400, enabling interrupts to be migrated between any cores in the clusters. All cores can signal each other using distributed interrupt controllers such as the CoreLink GIC-400. Task switching is handled entirely within the OS scheduler, and is invisible to the application software. A typical system is shown in Figure 23-1.

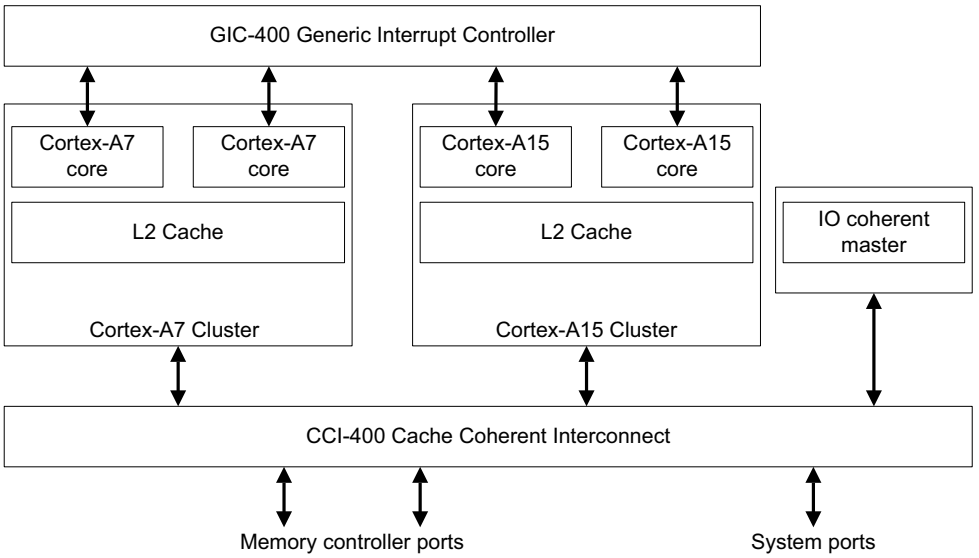


Figure 23-1 Typical big.LITTLE system

23.1.1 big.LITTLE configurations

A number of big.LITTLE configurations are possible, Figure 23-1 uses a Cortex-A7 processor as the LITTLE cluster and a Cortex-A15 processor as the big cluster, though other configurations are possible.

The LITTLE cluster is capable of handling most low intensity tasks such as audio playback, web-page scrolling, operating system events, and other always on, always connected tasks. As such, it is likely that the LITTLE cluster is where the software stack will remain until intensive tasks such as gaming or video processing are run.

The big cluster can be utilized for heavy workloads such as high performance graphics. A coupling of these two cluster types provides opportunities to save energy as well as satisfy the increasing performance demands of applications stacks in mobile devices.

23.2 Software execution models in big.LITTLE

There are two primary execution models for big.LITTLE:

- Migration.
- Global Task Scheduling.

Migration models are a natural extension to power-performance management techniques such as DVFS, (see *Dynamic Voltage and Frequency Scaling* on page 20-7).

The Migration model has two types:

- Cluster migration.
- CPU migration.

A migration action is similar to a DVFS operating point transition. Operating points on the DVFS curve of a core will be traversed in response to load variations. When the current core (or cluster) has attained the highest operating point, if the software stack requires more performance, a core (or cluster) migration action is effected (see Figure 23-2). Execution then continues on the other core (or cluster) with the operating points on this core (or cluster) being traversed. When performance is not required, execution can switch back.

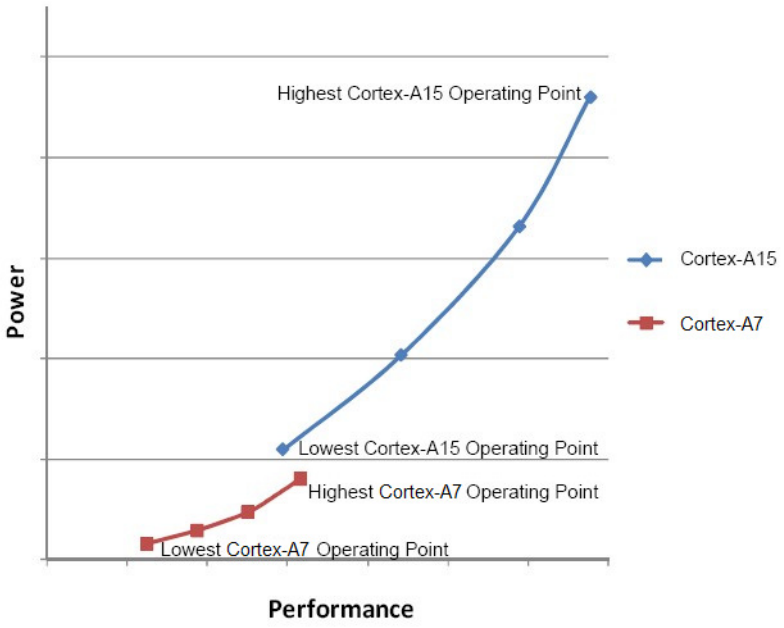


Figure 23-2 Typical Cortex-A15 and Cortex-A7 DVFS curves

The following sections describe the execution models:

- *Cluster migration* on page 23-5.
- *CPU migration* on page 23-5.
- *Global Task Scheduling* on page 23-5.

23.2.1 Cluster migration

Only one cluster, either big or LITTLE, is active at any one time, except very briefly during a cluster context switch to the other cluster. To achieve the best power and performance efficiency, the software stack runs mostly on the energy-efficient LITTLE cluster and only runs for short time periods on the big cluster. This model requires the same number of cores in both clusters.

This model does not cope well with unbalanced software workloads, that is, workloads that place significantly different loads on cores within a cluster. In such situations, cluster migration will result in a complete switch to the big cluster even though not all the cores need that level of performance. For this reason Cluster migration is less popular than other methods.

23.2.2 CPU migration

In this model, each big core is paired with a LITTLE core. Only one core in each pair is active at any one time, with the inactive core being powered down. The active core in the pair is chosen according to current load conditions. Using the example in Figure 23-3, the operating system sees four logical cores. Each logical core can physically be a big or LITTLE core. This choice is driven by *Dynamic Voltage and Frequency Scaling* (DVFS). This model requires the same number of cores in both the clusters.

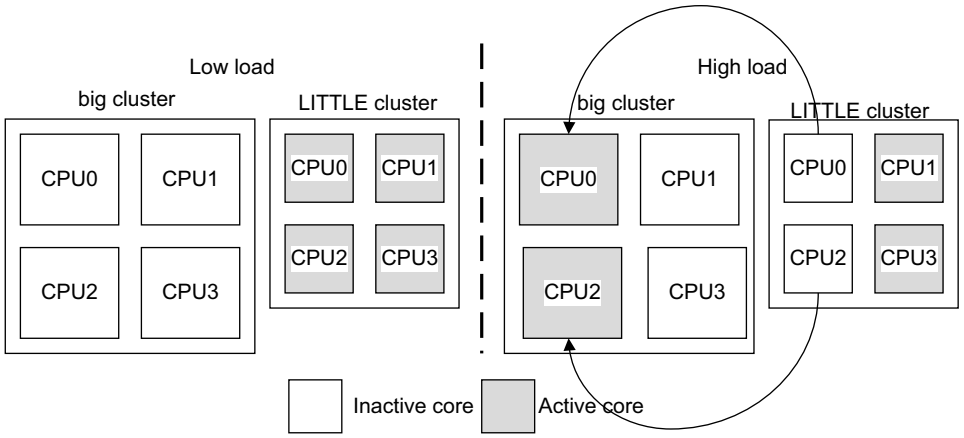


Figure 23-3 CPU migration

The system actively monitors the load on each core. High load causes the execution context to be moved to the big core, and conversely, when the load is low, the execution is moved to the LITTLE core. Only one core in the pairing can be active at any time. When the load is moved from an outbound core to an inbound core, the former is switched off. This model allows a mix of big and LITTLE cores to be active at any one time, and support the use of asymmetric topologies, that is, an unequal number of big and LITTLE cores.

The *In Kernel Switcher* (IKS) solution from Linaro is an example of this model.

23.2.3 Global Task Scheduling

Through the development of big.LITTLE technology, ARM has evolved the software models starting with various migration models through to *Global Task Scheduling* (GTS) that forms the basis for all future development in big.LITTLE technology. The ARM implementation of GTS is called big.LITTLE *Multi-processing* (MP).

In this model the operating system task scheduler is aware of the differences in compute capacity between big and LITTLE cores. Using statistical data, the scheduler tracks the performance requirement for each individual software thread, and uses that information to decide which type of core to use for each. Unused cores can be powered off. If all cores in a cluster are off, the cluster itself can be powered off. This model can work on a big.LITTLE system with any number of cores in any cluster. This is shown in Figure 23-4. This approach has a number of advantages over the migration models.

The system can have different numbers of big and LITTLE cores.

- Any number of cores can be active at any one time. When peak performance is required the system can deploy all cores.
- It is possible to isolate the big cluster for the exclusive use of intensive threads, while light threads run on the LITTLE cluster. This enables heavy compute tasks to complete faster, as there are no additional background threads.
- It is possible to target interrupts individually to big or LITTLE cores.

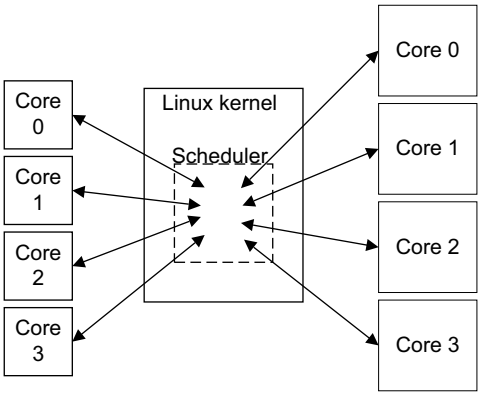


Figure 23-4 Global Task Scheduling

23.3 big.LITTLE MP

For big.LITTLE MP the fundamental requirement is for the scheduler to decide when a software thread can run on a LITTLE core or a big core. The scheduler does this by comparing the tracked load of software threads against tunable load thresholds, an *up migration threshold* and a *down migration threshold* as shown in Figure 23-5.

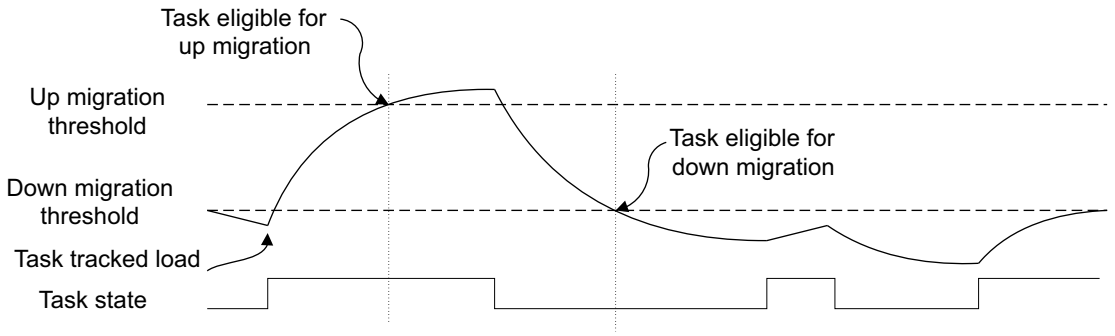


Figure 23-5 Migration thresholds

When the tracked load average of a thread currently allocated to a LITTLE core exceeds the up migration threshold, the thread is considered eligible for migration to a big core. Conversely, when the load average of a thread that is currently allocated to a big core drops below the down migration threshold, it is considered eligible for migration to a LITTLE core. In big.LITTLE MP these basic rules govern task migration between big and LITTLE cores. Within the clusters, standard Linux scheduler load balancing applies. This tries to keep the load balanced across all the cores in one cluster.

The model is refined by adjusting the tracked load metric based on the current frequency of a core. A task that is running when the core is running at half speed, will accrue tracked load at half the rate that it would if the core was running at full speed. This enables big.LITTLE MP and DVFS management to work together in harmony.

big.LITTLE MP uses a number of mechanisms to determine when to migrate a task between big and LITTLE cores:

23.3.1 Fork migration

This operates when the `fork` system call is used to create a new software thread. At this point, clearly no historical load information is available. The system defaults to a big core for new threads on the assumption that a ‘light’ thread will quickly migrate down to a LITTLE core as a result of *Wake migration*.

Fork migration benefits demanding tasks without being expensive. Threads that are low intensity and persistent, such as Android system services, will only be moved to big cores at creation time, quickly moving to more suitable LITTLE cores thereafter. Threads that are clearly demanding throughout, will not be penalized by being made to launch on LITTLE cores first. Threads that are episodic but tend to require performance on the whole will benefit from being launched on the big cluster and will continue to run there as required.

23.3.2 Wake migration

When a task that was previously idle becomes ready to run, the scheduler must decide which cluster will execute the task. To choose between big and LITTLE, big.LITTLE MP uses the tracked load history of a task. Generally, the assumption is that the task will resume on the same

cluster as before. The load metric is not updated for a task that is sleeping. Therefore, when scheduler checks the load metric of a task at wake up, before choosing a cluster to execute it on, the metric will have the value it had when the task last ran. This property means that tasks that a periodically busy will always tend to wake up on a big core. A task has to actually modify its behavior to change cluster.

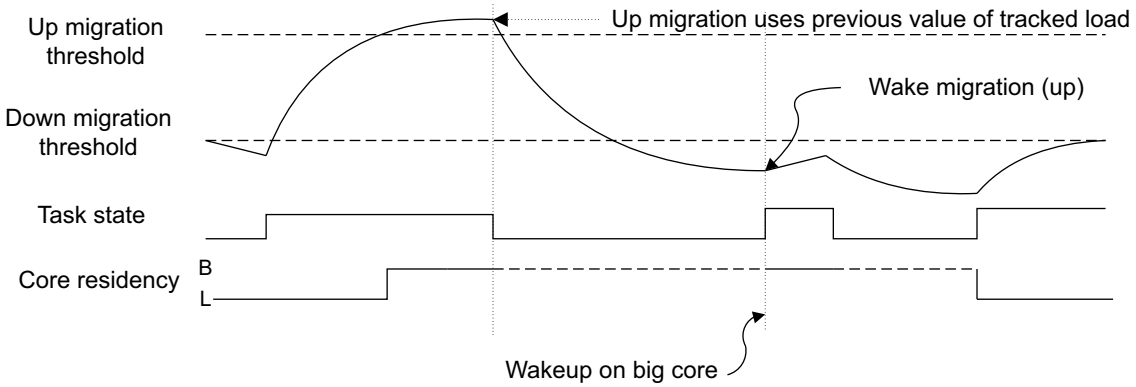


Figure 23-6 Wake migration on a big core

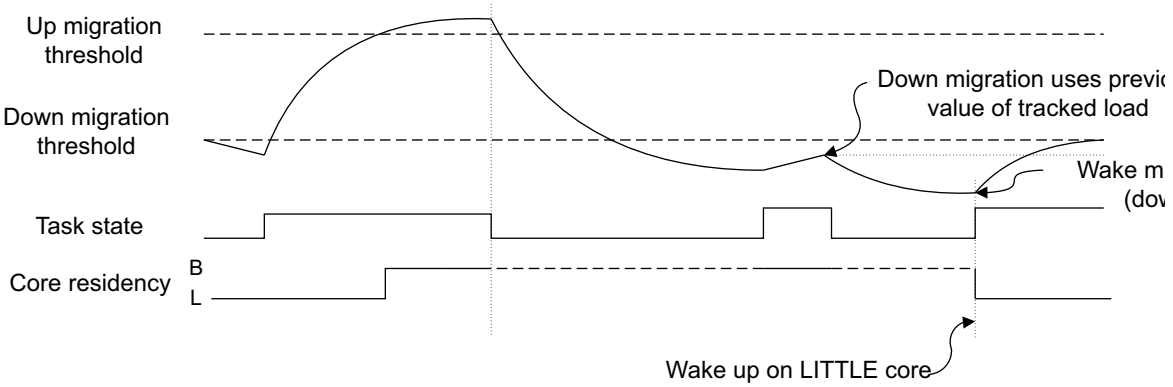


Figure 23-7 Wake migration on a LITTLE core

If a task modifies its behavior, and the load metric has crossed either of the up or down migration thresholds, the task can be allocated to a different cluster. Figure 23-6 and Figure 23-7 illustrate this process. Rules are defined that ensure that big cores generally only run a single intensive thread and run it to completion, so upward migration only occurs to big cores which are idle. When migrating downwards, this rule does not apply and multiple software threads can be allocated to a little core.

23.3.3 Forced migration

Forced migration deals with the problem of long running software threads that do not sleep, or do not sleep very often. The scheduler periodically checks the current thread running on each LITTLE core. If the tracked load exceeds the up migration threshold the task is transferred to a big core, as in Figure 23-8 on page 23-9.

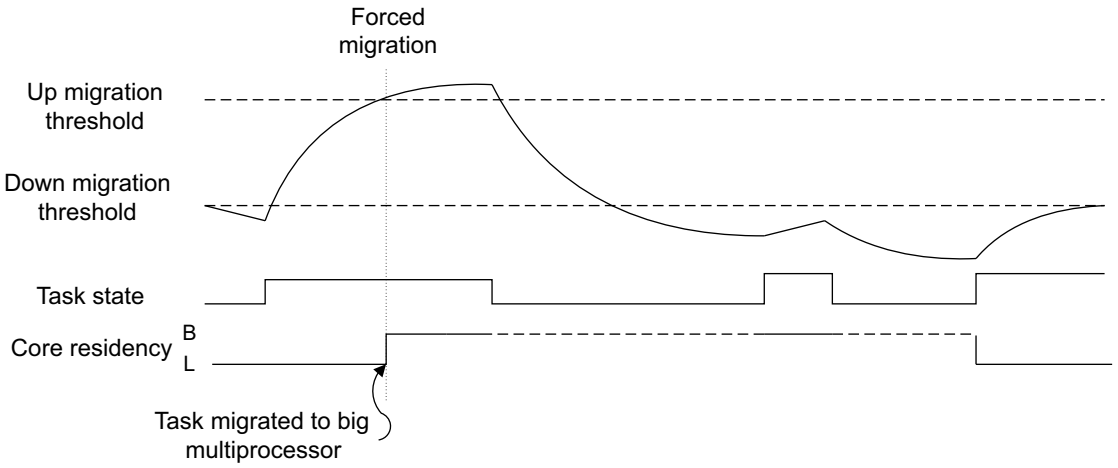


Figure 23-8 Forced migration

23.3.4 Idle pull migration

Idle pull migration is designed to make best use of active big cores. When a big core has no task to run, a check is made on all LITTLE cores to see if a currently running task on a LITTLE core has a higher load metric that the up migration threshold. Such a task can then be immediately migrated to the idle big core. If no suitable task is found, then the big core can be powered down. This technique ensures that big cores, when they are running, always take the most intensive tasks in a system and run them to completion.

23.3.5 Offload migration

Offload migration requires that normal scheduler load balancing be disabled. The downside of this is that long-running threads can concentrate on the big cores, leaving the LITTLE cores idle and under-utilized. Overall system performance, in this situation, can clearly be improved by utilizing all the cores.

Offload migration works to periodically migrate threads downwards to LITTLE cores to make use of unused compute capacity. Threads that are migrated downwards in this way remain candidates for up migration if they exceed the threshold at the next scheduling opportunity.

23.4 Using big.LITTLE

For a representative sample of common use cases, a big.LITTLE system running the ARM big.LITTLE MP solution shows significant power savings when compared to a system composed of only Cortex-A15 processors.

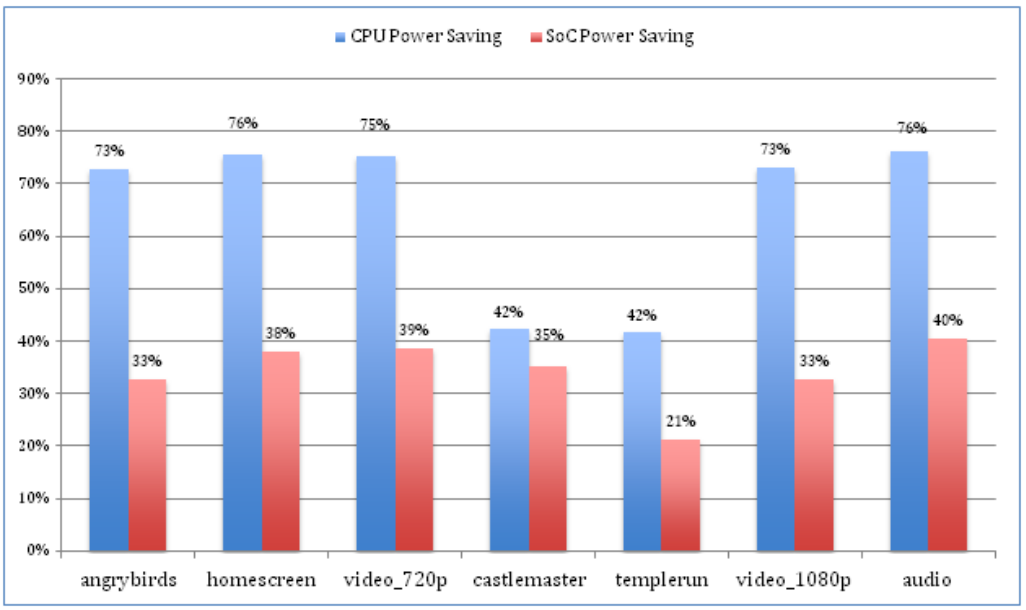


Figure 23-9 Typical big.LITTLE MP power savings compared to the Cortex-A15

shows how big.LITTLE MP benefits benchmarks. The comparison is between a big.LITTLE system composed of four LITTLE cores and four big cores and a system composed only of four big cores.

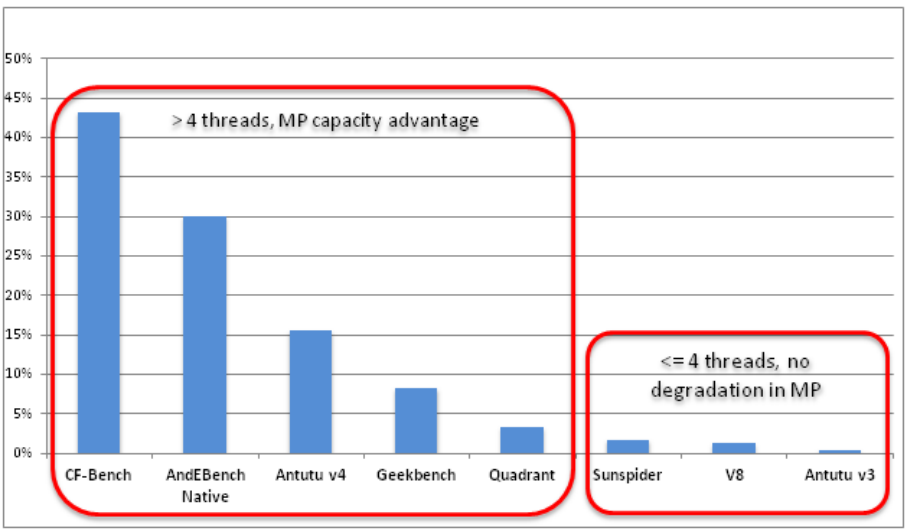


Figure 23-10 big.LITTLE benchmark improvements

The software thread affinity management techniques discussed earlier result in substantial performance gains for threaded benchmarks where the number of threads is greater than four. In this situation, on the system under test big.LITTLE MP enables the use of more cores to aid

the benchmark. Offload migration assists with spreading the number of compute intensive benchmark threads to the LITTLE cores when the big cores are busy or overloaded. Idle-pull migration results in the best utilisation of the big processors which effectively work as accelerators.

For benchmarks with fewer threads, using big.LITTLE MP either provides no degradation or a small improvement. Compared to the test system with only four big cores, dynamic software thread affinity management will promote better utilisation of the big cores which will not be encumbered with low intensity and frequent running threads (such as system services) or interrupts.

ARM big.LITTLE MP technology has been tested with Android on multiple silicon implementations. The code is self-contained and available as a drop-in into the software stack without significant modification or tuning. The only requirement is that the platform board-support package is well tuned in terms of DVFS and idle power management, allowing the scheduler extensions to focus on getting the job done.

The big.LITTLE MP scheduler extensions are available in two forms:

- As a part of monthly *Linaro Stable Kernel* (LSK) releases. These releases contain a complete Android software stack based on a very recent Linux Stable Kernel. The stack is available in source form and also as a pre-built binary set complete with boot firmware, boot loaders, ramdisk images and an Android root filesystem image.

  See `https://releases.linaro.org` Select the latest release and `android/vexpress-lsk` for details on the LSK.

- As an isolated patch set against the LSK kernel.

  See `https://wiki.linaro.org/ARM/VersatileExpress`

  Select **Attachments** and then chose **get** for the latest `lsk.tar.bz2` version of attachment:big-LITTLE-MP-scheduler-patchset.

Chapter 24
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 processors 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 processors 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 processor 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.

24.1 ARM debug hardware

Cortex-A 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 you 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 core 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, originally designed to standardize testing of electronic devices on boards, but now widely re-used for core 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 core, step through code and resume execution. We can set breakpoints on specific instructions (causing the debugger to take control when the core 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 breakpointed instruction. Hardware breakpoints use comparators built into the core 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 (typically four in the Cortex-A family). 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.

24.1.1 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 single step through the function instead.

24.1.2 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 core 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 core to enter Debug state. The core is halted and isolated from the rest of the system. This means that the debugger displays memory as seen by the core, and the effects of memory management and cache operations will become visible.

In Debug state, the core 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 core context and control all subsequent instruction execution. Both the core and system state can be modified. Because the core is stopped, no interrupts will be 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. Since the core is still operating, interrupts can still be serviced.

24.1.3 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 enable 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 semi-hosting 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 semi-hosting, for example, by using `fputc()`. This would involve replacing the library code that used an `SVC` call with code that could output a character.

24.2 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 current very high core clock speeds, even a few seconds of operation can mean trillions of cycles of execution. Clearly, to make sense of this volume of information would be extremely difficult. The second, related problem is that current cores 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 core 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 core. 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.

24.2.1 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-core systems (both asymmetric and SMP) that can share debug access and trace pins, with full control

of which cores 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 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. A brief description of some of the available CoreSight components follows:

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 core into debug state. To read or write memory without a DAP might require the debugger to stop the core and have it execute Load or Store instructions. The DAP gives an external debug tool access to all of the JTAG scan chains in a system and therefore to all debug and trace configuration registers of the available cores.

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 must also be stopped (regardless of the 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 core ETM, as the integer 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 cores (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 usage 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.

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.

You must consult documentation specific to the device you are using to determine what trace capabilities are present and the tools available to make use of them.

24.3 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 core – 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 core 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 core under control of an external hardware debugger, the core instead takes an abort exception and this can recognize that the abort was generated by a debug event and call the Monitor code.

24.4 Debugging Linux applications

Linux is a multi-tasking operating system in which each process has its own process address space, complete with private translation table mappings. This can make debug of some kinds of problems quite tricky.

We can broadly define two different debug approaches used in Linux systems.

- Linux applications are typically debugged using a GDB debug server running on the target, communicating with a host computer, usually through Ethernet. The kernel continues to operate normally while the debug session takes place. This method of debug does not provide access to the built-in hardware debug facilities. The target system is permanently in a running state. The server receives a connection request from the host debugger and then receives commands and provides data back to the host.

  The host debugger sends a load request to the GDB server, which responds by starting a new process to run the application being debugged. Before execution begins, it uses the system call `ptrace()` to control the application process. All signals from this process are forwarded to the GDB server. Any signals sent to the application will go instead to the GDB server that can deal with the signal or forward it to the application being debugged. To set a breakpoint, the GDB server inserts code that generates the **SIGTRAP** signal at the required location in the code. When this is executed, the GDB server is called and can then perform classic debugger tasks such as examining call stack information, variables or register contents.

- For kernel debug, a JTAG-based debugger is used. The system is halted when a breakpoint is executed. This is the easiest way to examine problems such as device driver loading or incorrect operation or the kernel boot failure. Another common method is through `printk()` function calls. The `strace` tool shows information about user system calls. Kgdb is a source-level debugger for the Linux kernel that works with GDB on a separate machine and enables inspection of stack traces and view of kernel state (such as PC value, timer contents, and memory). The device `/dev/kmem` enables run-time access to the kernel memory.

  Of course, a Linux-aware JTAG debugger can be used to debug threads. It is usually possible only to halt all processes; one cannot halt an individual thread or process and leave others running. A breakpoint can be set either for all threads, or it can be set only on a specific thread.

  As the memory map depends on which process is active, software breakpoints can usually only be set when a particular process is mapped in.

The ARM DS-5 Debugger is able to debug Linux applications using gdbserver and Linux kernel and Linux kernel modules using JTAG. The debug and trace features of DS-5 are described in the next section.

24.4.1 The call stack

Application code uses the call stack to pass parameters, store local data and store return addresses. The data each function pushes on the stack is organized into a *stack frame*. When a debugger stops a core, it might be able to analyze the data on the stack to provide you with a call stack, that is, a list of function calls leading up to the current situation. This can be extremely useful when debugging, as it enables you to determine why the application has reached a particular state.

In order to reconstruct the call stack, the debugger must be able to determine which entries on the stack contain return address information. This information might be contained in *debugger information* (DWARF debug tables) if the code was built with these included, or by following

a chain of *frame pointers* pushed on the stack by the application. To do this, the code must be built to use frame pointers. If neither of these types of information are present, the call stack can not be constructed.

In multi-threaded applications, each thread has its own stack. The call stack information will therefore only relate to the particular thread being examined.

24.5 DS-5 debug and trace

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.

DS-5 Debugger provides a wide range of debug features such as:

- 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 and Android applications.
- Debug of Linux kernel and Android modules, boot code and kernel porting.
- Application rewind, that allows you to debug backwards as well as forwards through Linux and Android applications.

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. 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 for bare-metal SMP systems, including cross-triggering and core-dependent views and breakpoints, PTM trace, and up to 4 GB trace with DSTREAM. This support is described in the following sections.

In addition, DS-5 Debugger supports ARM CoreSight ETM, PTM and STM, 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 ETB.

24.5.1 Debugging Linux or Android applications using DS-5

DS-5 Debugger takes care of downloading and connecting to the debug server. 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.

To debug a Linux or Android application you can use a TCP or serial connection:

- To `gdbserver` running on a model that is pre-configured to boot Linux or Android.
- To `gdbserver` running on a hardware target.

This type of development requires `gdbserver` to be installed and running on the target.

24.5.2 Debugging Linux kernel modules

Linux kernel modules provide a way to extend the functionality of the kernel, and are typically used for things such as device and filesystem drivers. Modules can either be built into the kernel or can be compiled as a loadable module and then dynamically inserted and removed from a running kernel during development without having to frequently recompile the kernel. However, some modules must be built into the kernel and are not suitable for loading dynamically. An example of a built-in module is one that is required during kernel boot and must be available prior to the root filesystem being mounted.

You can use DS-5 Debugger to set source-level breakpoints in a module provided that the debug information is loaded into the debugger. Attempts to set a breakpoint in a module before it is inserted into the kernel results in the breakpoint being pended.

When debugging a module, you must ensure that the module on your target is the same as that on your host. The code layout must be identical, but the module on your target does not have to contain debug information.

Built in module

To debug a module that has been built into the kernel using DS-5 Debugger, the procedure is the same as for debugging the kernel itself:

1. Compile the kernel together with the module.
2. Load the kernel image on to the target.
3. Load the related kernel image with debug information into the debugger.
4. Debug the module as you would for any other kernel code.

Loadable module

The procedure for debugging a loadable kernel module is more complex. From a Linux terminal shell you can use the `insmod` and `rmmod` commands to insert and remove a module. Debug information for both the kernel and the loadable module must be loaded into the debugger. When you insert and remove a module, DS-5 Debugger automatically resolves memory locations for debug information and existing breakpoints.

To do this, DS-5 Debugger intercepts calls within the kernel to insert and remove modules. This introduces a small delay for each action while the debugger stops the kernel to interrogate various data structures.

24.5.3 Debugging Linux kernels using DS-5

To debug a Linux kernel module you can use a debug hardware agent connected to the host workstation and the running target.

24.5.4 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 24-1 Threading call stacks in the DS-5 Debug Control view

24.5.5 Debugging shared libraries

Shared libraries enable parts of your application to be dynamically loaded at runtime. You must ensure that the shared libraries on your target are the same as those on your host. The code layout must be identical, but the shared libraries on your target do not have to contain debug information.

You can set standard execution breakpoints in a shared library but not until it is loaded by the application and the debug information is loaded into the debugger. Pending breakpoints however, enable you to set execution breakpoints in a shared library before it is loaded by the application.

When a new shared library is loaded DS-5 Debugger re-evaluates all pending breakpoints, those with addresses that it can resolve, are set as standard execution breakpoints. Unresolved addresses remain as pending breakpoints.

The debugger automatically changes any breakpoints in a shared library to a pending breakpoint when the library is unloaded by your application.

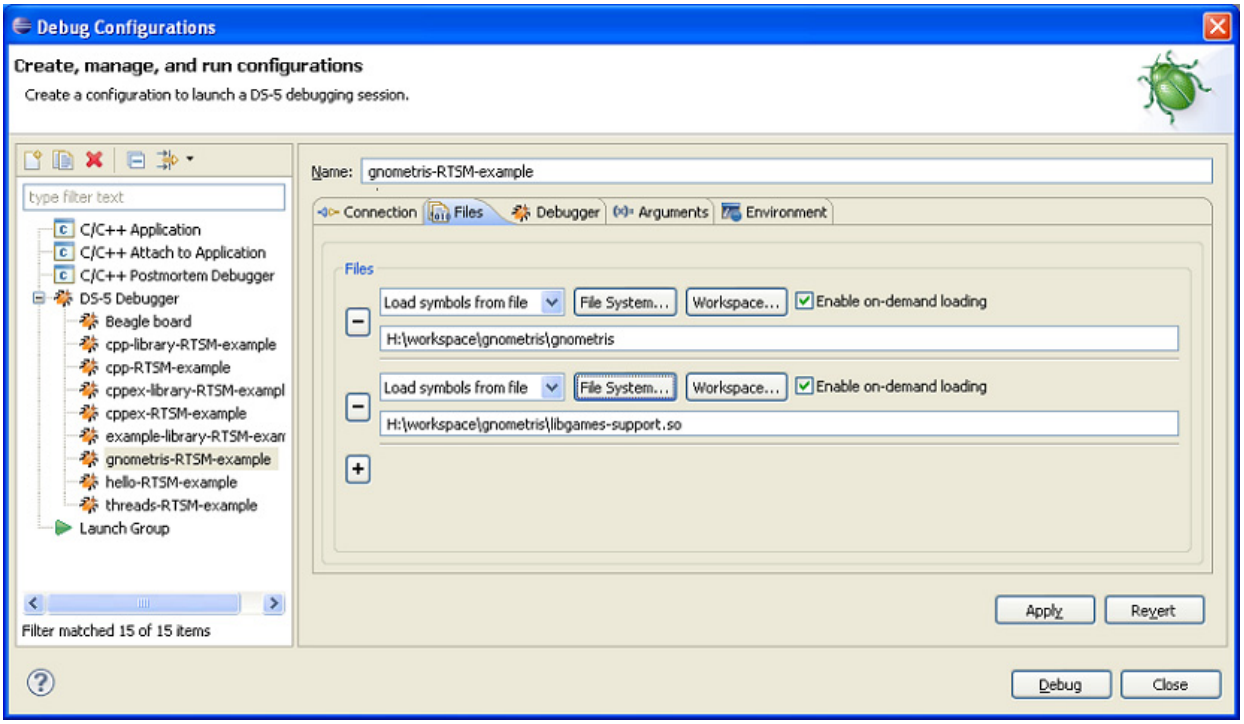


Figure 24-2 Adding shared libraries for debug using DS-5

24.5.6 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 core. 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.

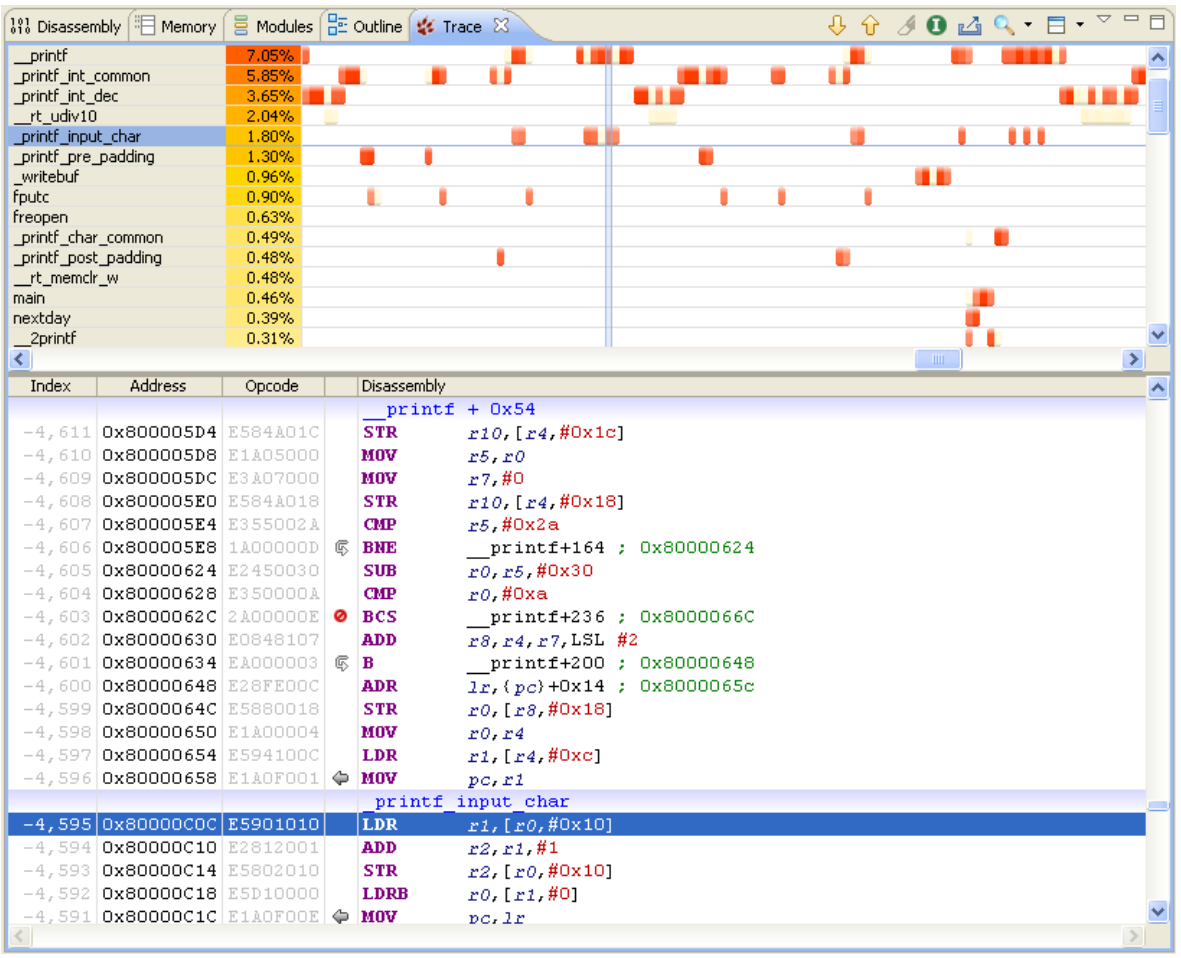


Figure 24-3 DS-5 Debugger Trace view

Trace-based profiling

Based on trace data received from a trace buffer such as the ETB, 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 core 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.

Appendix A
Instruction Summary

A summary of the instructions available in ARM/Thumb assembly language is given in this Appendix.

For most instructions, additional explanation can be found in Chapter 5. The optional condition code field (denoted by `cond`) is described in *Conditional execution* on page 5-3. The format of the flexible `operand2` used by data processing operations is described in *Operand 2 and the barrel shifter* on page 5-7, while the various addressing mode options for loads and stores is given in *Addressing modes* on page 5-13.

This Appendix is intended for quick reference. If more detail about the precise operation of an instruction is required, refer to the *ARM Architecture Reference Manual*, or to the official ARM documentation (for example the *ARM Compiler Toolchain Assembler Reference*) that can be found at `http://infocenter.arm.com/help/index.jsp`.

A.1 Instruction Summary

Instructions are listed in alphabetic order, with a description of the syntax, operands and behavior of the instruction. Not all usage restrictions are documented here, nor do we show the associated binary encoding or any detail of changes associated with older architecture versions.

A.1.1 ADC

`ADC` (Add with Carry) adds together the values in `Rn` and `Operand2`, with the carry flag.

Syntax

```
ADC{S}{cond} {Rd}, Rn, <Operand2>
```

where:

`S` (if specified) means that the condition code flags will be updated depending on the result of the instruction.

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`Rd` is the destination register.

`Rn` is the register holding the first operand.

`Operand2` is a flexible second operand. See *Operand 2 and the barrel shifter* on page 5-7.

A.1.2 ADD

`ADD` adds together the values in `Rn` and `Operand2` (or `Rn` and `imm12`).

Syntax

```
ADD{S}{cond} {Rd,} Rn, <Operand2>
```

```
ADD{cond} {Rd,} Rn, #imm12
```
(Only available in Thumb)

where:

`S` (if specified) means that the condition code flags will be updated depending on the result of the instruction.

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`Rd` is the destination register.

`Rn` is the register holding the first operand.

`Operand2` is a flexible second operand. See *Operand 2 and the barrel shifter* on page 5-7.

`imm12` is in the range 0-4095.

A.1.3 ADR

`ADR` (Address) is an instruction that loads a program or register-relative address (short range). It generates an instruction that adds or subtracts a value to the PC (in the PC-relative case). Alternatively, it can be some other register for a label defined as an offset from a base register defined with the `MAP` directive (see the ARM tools documentation for more detail).

Syntax

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

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`Rd` is the destination register.

`label` is a PC or register-relative expression.

A.1.4 ADRL

`ADRL` (Address) is a pseudo-instruction that loads a program or register-relative address (long range). It always generates two instructions.

Syntax

```
ADRL{cond} Rd, label
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`Rd` is the destination register.

`label` is a PC-relative expression. The offset between label and the current location has some restrictions.

The `ADRL` pseudo-instruction can generate a wider range of addresses than `ADR`.

A.1.5 AND

`AND` does a bitwise AND on the values in `Rn` and `Operand2`.

Syntax

```
AND{S}{cond} {Rd,} Rn, <Operand2>
```

where:

`S` (if specified) means that the condition code flags will be updated depending on the result of the instruction.

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`Rd` is the destination register.

`Rn` is the register holding the first operand.

`Operand2` is a flexible second operand. See *Operand 2 and the barrel shifter*.

A.1.6 ASR

`ASR` (Arithmetic Shift Right) shifts the value in `Rm` right, by the number of bit positions specified and copies the sign bit into vacated bit positions on the left. Permitted shift values are in the range 1-32. It can be considered as giving the signed value of a register divided by a power of two.

Syntax

```
ASR{S}{cond} {Rd,} Rm, Rs
ASR{S}{cond} {Rd,} Rm, imm
```

where:

`S` (if specified) means that the condition code flags will be updated depending on the result of the instruction.

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`Rd` is the destination register.

`Rm` is the register holding the operand to be shifted.

`Rs` is the register that holds a shift value to apply to the value in `Rm`. Only the least significant byte of the register is used.

`imm` is a shift amount, in the range 1-32.

A.1.7 B

`B` (Branch) transfers program execution to the address specified by label.

Syntax

```
B{cond}{.W} label
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`label` is a PC-relative expression.

`.W` is an optional instruction width specifier to force the use of a 32-bit instruction in Thumb.

A.1.8 BFC

`BFC` (Bit Field Clear) clears bits in a register. A number of bits specified by width are cleared in `Rd`, starting at `lsb`. Other bits in `Rd` are unchanged.

Syntax

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

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3

`Rd` is the destination register.

`lsb` specifies the least significant bit to be cleared.

`width` is the number of bits to be cleared.

A.1.9 BFI

`BFI` (Bit Field Insert) copies bits into a register. A number of bits in `Rd` specified by width, starting at `lsb`, are replaced by bits from `Rn`, starting at bit[0]. Other bits in `Rd` are unchanged.

Syntax

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

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`Rd` is the destination register.

`Rn` is the register that contains the bits to be copied.

`lsb` specifies the least significant bit in `Rd` to be written to.

`width` is the number of bits to be copied.

A.1.10 BIC

`BIC` (bit clear) does an AND operation on the bits in `Rn`, with the complements of the corresponding bits in the value of `Operand2`.

Syntax

```
BIC{S}{cond} {Rd}, Rn, <Operand2>
```

where:

`S` (if specified) means that the condition code flags will be updated depending on the result of the instruction.

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`Rd` is the destination register.

`Rn` is the register holding the first operand.

`Operand2` is a flexible second operand. See *Operand 2 and the barrel shifter* on page 5-7.

A.1.11 BKPT

`BKPT` (Breakpoint) causes the core to enter Debug state.

Syntax

```
BKPT #imm
```

where:

`imm` is an integer in the range 0 – 65535 (ARM) or 0 – 255 (Thumb). This integer is not used by the core itself, but can be used by Debug tools.

A.1.12 BL

`BL` (Branch with Link) transfers program execution to the address specified by label and stores the address of the next instruction in the LR (R14) register.

Syntax

```
BL{cond} label
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

label is a PC-relative expression.

A.1.13 BLX

BLX (Branch with Link and eXchange) transfers program execution to the address specified by label and stores the address of the next instruction in the LR (R14) register. BLX can change the core state from ARM to Thumb, or from Thumb to ARM. BLX label always changes the core state from Thumb to ARM, or ARM to Thumb. BLX Rm will set the state based on bit[0] of Rm:

- Rm bit[0]=0 ARM state.
- Rm bit[0]=1 Thumb state.

Syntax

```
BLX{cond} label
BLX{cond} Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

label is a PC-relative expression.

Rm is a register that holds the address to branch to.

A.1.14 BX

BX (Branch and eXchange) transfers program execution to the address specified in a register. BX can change the core state from ARM to Thumb, or from Thumb to ARM. BX Rm will set the state based on bit[0] of Rm:

- Rm bit[0] = 0 ARM state.
- Rm bit[0] = 1 Thumb state.

Syntax

```
BX{cond} Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rm is a register that holds the address to branch to.

A.1.15 BXJ

BXJ (Branch and eXchange Jazelle) enter Jazelle State or perform a BX branch and exchange to the address contained in Rm.

Syntax

```
BXJ{cond} Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rm is a register that holds the address to branch to if entry to Jazelle fails.

A.1.16 CBNZ

CBNZ (Compare and Branch if Nonzero) causes a branch if the value in Rn is not zero. It does not change the PSR flags. There is no ARM or 32-bit Thumb versions of this instruction.

Syntax

```
CBNZ Rn, label
```

where:

label is a pc-relative expression.

Rn is a register that holds the operand.

A.1.17 CBZ

CBZ (Compare and Branch if Zero) causes a branch if the value in Rn is zero. It does not change the PSR flags. There is no ARM or 32-bit Thumb versions of this instruction.

Syntax

```
CBZ Rn, label
```

where:

label is a PC-relative expression.

Rn is a register that holds the operand.

A.1.18 CDP

CDP (Coprocessor Data Processing operation) performs a coprocessor operation. The purpose of this instruction is defined by the coprocessor implementer.

Syntax

```
CDP{cond} coproc, #opcode1, CRd, CRn, CRm{, #opcode2}
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

coproc is the name of the coprocessor the instruction is for. This is usually of the form pn, where n is an integer in the range 0 to 15.

opcode1 is a 4-bit coprocessor-specific opcode.

opcode2 is an optional 3-bit coprocessor-specific opcode.

CRd, CRn, CRm are coprocessor registers.

A.1.19 CDP2

CDP2 (Coprocessor Data Processing operation) performs a coprocessor operation. The purpose of this instruction is defined by the coprocessor implementer.

Syntax

```
CDP2{cond} coproc, #opcode1, CRd, CRn, CRm{, #opcode2}
```

where:

`cond` is an optional condition code See *Conditional execution* on page 5-3.

`coproc` is the name of the coprocessor the instruction is for. This is usually of the form `pn`, where `n` is an integer in the range 0 to 15.

`opcode1` is a 4-bit coprocessor-specific opcode.

`opcode2` is an optional 3-bit coprocessor-specific opcode.

`CRd`, `CRn`, `CRm` are coprocessor registers.

A.1.20 CHKA

`CHKA` (Check array) is a ThumbEE instruction. If the value in the first register is less than or equal to, the second, the IndexCheck handler is called. This instruction is only available in 16-bit ThumbEE and only when Thumb-2EE support is present.

Syntax

```
CHKA Rn, Rm
```

where:

`Rn` holds the size of the array.

`Rm` contains the array index.

A.1.21 CLREX

`CLREX` (Clear Exclusive) moves a local exclusive access monitor to its open-access state.

Syntax

```
CLREX{cond}
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

A.1.22 CLZ

`CLZ` (Count Leading Zeros) counts the number of leading zeros in the value in `Rm` and returns the result in `Rd`. The result returned is 32 if no bits are set in `Rm`, or 0 if bit [31] is set.

Syntax

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

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`Rd` is the destination register.

`Rm` is the register holding the operand.

A.1.23 CMN

CMN (Compare Negative) performs a comparison by adding the value of Operand2 to the value in Rn. The condition flags are changed, based on the result, but the result itself is discarded.

Syntax

```
CMN{cond} Rn, <Operand2>
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rn is the register holding the first operand.

Operand2 is a flexible second operand. See *Operand 2 and the barrel shifter* on page 5-7.

A.1.24 CMP

CMP (Compare) performs a comparison by subtracting the value of Operand2 from the value in Rn. The condition flags are changed, based on the result, but the result itself is discarded.

Syntax

```
CMP{cond} Rn, <Operand2>
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rn is the register holding the first operand.

Operand2 is a flexible second operand. See *Operand 2 and the barrel shifter* on page 5-7.

A.1.25 CPS

CPS (Change Processor State) can be used to change the processor mode or to enable or disable individual exception types.

Syntax

```
CPS #mode
CPSIE iflags{, #mode}
CPSID iflags{, #mode}
```

where:

mode is the number of a mode for the processor to enter.

IE Interrupt or Abort Enable.

ID Interrupt or Abort Disable.

iflags specifies one or more of:

- a = asynchronous abort.
- i = IRQ.
- f = FIQ.

A.1.26 DBG

DBG (Debug) is a hint operation, treated as a NOP by the processor, but can provide information to debug systems.

Syntax

```
DBG{cond} {option}
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

option is in the range 0-15.

A.1.27 DMB

DMB (Data Memory Barrier) requires that all explicit memory accesses in program order before the DMB instruction are observed before any explicit memory accesses in program order after the DMB instruction. See Chapter 10 for a detailed description.

Syntax

```
DMB{cond} {option}
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

option is covered in depth in Chapter 10.

A.1.28 DSB

DSB (Data Synchronization Barrier) requires that no further instruction executes until all explicit memory accesses, cache maintenance operations, branch prediction and TLB maintenance operations before this instruction complete. See Chapter 10 for a detailed description.

Syntax

```
DSB{cond} {option}
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

option is covered in depth in Chapter 10.

A.1.29 ENTERX

ENTERX causes a change from Thumb state to ThumbEE state, or has no effect in ThumbEE state. It is not available in the ARM instruction set.

Syntax

```
ENTERX
```

A.1.30 EOR

EOR performs an Exclusive OR operation on the values in Rn and Operand2.

Syntax

```
EOR{S}{cond} {Rd,} Rn, <Operand2>
```

where:

S (if specified) means that the condition code flags will be updated depending on the result of the instruction.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register holding the first operand.

Operand2 is a flexible second operand. See *Operand 2 and the barrel shifter* on page 5-7.

A.1.31 ERET

ERET (Exception Return) loads the PC from the ELR-hyp and loads the CPSR from SPSR-hyp when executed in Hyp mode.

When executed in a Secure or Non-Secure PLI mode, ERET behaves as:

- MOVS PC, LR in the ARM instruction set.
- SUBS PC, LR, #0 in the Thumb instruction set.

Syntax

```
ERET{cond} {q}
```

where:

cond is the optional condition code. See *Conditional execution* on page 5-3.

q specifies optional assembler qualifiers on the instruction. The following qualifiers are defined:

- .N(narrow), specifies that the assembler must select a 16-bit encoding for the instruction.
- .W(wide), specifies that the assembler must select a 32-bit encoding for the intruction.

If neither .W nor .N are specified, the assembler can select either 16-bit or 32-bit encodings. if both are available, it must select a 16-bit encoding.

A.1.32 HB

HB (Handler Branch) branches to a specified handler (available in ThumbEE only).

Syntax

```
HB{L} #HandlerID
HB{L}P #imm, #HandlerID
```

where:

L indicates that the instruction saves a return address in the LR.

P means that the instruction passes the value of imm to the handler in R8.

imm is an immediate value in the range 0-31 (if L is present), otherwise in the range 0-7.

HandlerID is the index number of the handler to be called, in the range 0-31 (if P is specified), otherwise in the range 0-255.

A.1.33 ISB

ISB (Instruction Synchronization Barrier) flushes the processor pipeline and ensures that context altering operations (such as ASID or other CP15 changes, branch prediction or TLB maintenance activity) *before* the ISB, are visible to the instructions fetched *after* the ISB.

See Chapter 10 for a detailed description of barriers.

Syntax

```
ISB{cond} {option}
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

option can be SY (full system), which is the default and so can be omitted.

A.1.34 IT

IT (If-then) makes up to four following instructions conditional (known as the IT block). The conditions can all be the same, or some can be the logical inverse of others. IT is a pseudo-instruction in ARM state.

Syntax

```
IT{x{y{z}}} {cond}
```

where:

cond is a condition code. See *Conditional execution* on page 5-3 that specifies the condition for the first instruction in the IT block.

x, y and z specify the condition switch for the second, third and fourth instructions in the IT block, for example, ITTET.

The condition switch can be either:

- T (Then) applies the condition cond to the instruction.
- E (Else) applies the inverse condition of cond to the instruction.

A.1.35 LDA

LDA (Load-Acquire Word) loads a word from memory and writes it to a register. This instruction was introduced to provide backward compatibility for the ARMv8 architecture AArch32 state.

Note

LDA imposes ordering restrictions on memory accesses to the same shareability domain.

For further information, see Section E2.7.3 in the ARMv8 *Architecture Reference Manual* (ARM DDI 0487)

Syntax

```
LDA{cond}{q} <Rt>, [<Rn>]
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`q` specifies optional assembler qualifiers on the instruction. The following qualifiers are defined:

- `.N`(narrow), specifies that the assembler must select a 16-bit encoding for the instruction.
- `.W`(wide), specifies that the assembler must select a 32-bit encoding for the intruction.

If neither `.W` nor `.N` are specified, the assembler can select either 16-bit or 32-bit encodings. if both are available, it must select a 16-bit encoding.

`Rt` is the destination register.

`Rn` is the base register.

A.1.36 LDAB

`LDAB` (Load-Acquire Byte) loads a byte from memory, zero-extends it to form a 32-bit word and writes it to a register. This instruction was introduced to provide backward compatibility for the ARMv8 architecture AArch32 state.

Note

`LDAB` imposes ordering restrictions on memory accesses to the same shareability domain.

For further information, see Section E2.7.3 in the ARMv8 *Architecture Reference Manual* (ARM DDI 0487)

Syntax

```
LDAB{cond}{q} <Rt>, [<Rn>]
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`q` specifies optional assembler qualifiers on the instruction. The following qualifiers are defined:

- `.N`(narrow), specifies that the assembler must select a 16-bit encoding for the instruction.
- `.W`(wide), specifies that the assembler must select a 32-bit encoding for the intruction.

If neither `.W` nor `.N` are specified, the assembler can select either 16-bit or 32-bit encodings. if both are available, it must select a 16-bit encoding.

`Rd` is the destination register.

`Rm` is the base register.

A.1.37 LDAEX

`LDAEX` (Load-Acquire Exclusive Word) loads a word from memory and writes it to a register. This instruction was introduced to provide backward compatibility for the ARMv8 architecture AArch32 state.

Note

LDAEX imposes ordering restrictions on memory accesses to the same shareability domain.

For further information, see Section E2.7.3 in the ARMv8 *Architecture Reference Manual* (ARM DDI 0487)

Syntax

```
LDAEX{cond}{q} <Rt>, [<Rn>]
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

q specifies optional assembler qualifiers on the instruction. The following qualifiers are defined:

- .N(narrow), specifies that the assembler must select a 16-bit encoding for the instruction.
- .W(wide), specifies that the assembler must select a 32-bit encoding for the intruction.

If neither .W nor .N are specified, the assembler can select either 16-bit or 32-bit encodings. if both are available, it must select a 16-bit encoding.

Rt is the destination register.

Rn is the base register.

A.1.38 LDAEXB

LDAEXB (Load-Acquire Exclusive Byte) loads a byte from memory, zero-extends it to form a 32-bit word and writes it to a register. This instruction was introduced to provide backward compatibility for the ARMv8 architecture AArch32 state.

Note

LDAEXB imposes ordering restrictions on memory accesses to the same shareability domain.

For further information, see Section E2.7.3 in the ARMv8 *Architecture Reference Manual* (ARM DDI 0487)

Syntax

```
LDAEXB{cond}{q} <Rt>, [<Rn>]
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

q specifies optional assembler qualifiers on the instruction. The following qualifiers are defined:

- .N(narrow), specifies that the assembler must select a 16-bit encoding for the instruction.
- .W(wide), specifies that the assembler must select a 32-bit encoding for the intruction.

If neither .W nor .N are specified, the assembler can select either 16-bit or 32-bit encodings. if both are available, it must select a 16-bit encoding.

Rt is the destination register.

Rm is the base register.

A.1.39 LDAEXD

LDAEXD (Load-Acquire Exclusive Double) loads a doubleword from memory and writes it to two registers. This instruction was introduced to provide backward compatibility for the ARMv8 architecture AArch32 state.

Note

LDAEXD imposes ordering restrictions on memory accesses to the same shareability domain.

For further information, see Section E2.7.3 in the ARMv8 *Architecture Reference Manual* (ARM DDI 0487)

Syntax

```
LDAEXD{cond}{q} <Rt>, <Rt2>, [<Rn>]
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

q specifies optional assembler qualifiers on the instruction. The following qualifiers are defined:

- .N(narrow), specifies that the assembler must select a 16-bit encoding for the instruction.
- .W(wide), specifies that the assembler must select a 32-bit encoding for the intruction.

If neither .W nor .N are specified, the assembler can select either 16-bit or 32-bit encodings. if both are available, it must select a 16-bit encoding.

Rt is the first destination register. It must be an even numbered register and not R14..

Rt2 is the second destination register. Rt2 must be R(t + 1).

Rn is the base register.

A.1.40 LDAEXH

LDAEXH (Load-Acquire Exclusive Halfword) loads a halfword from memory, zero-extends it to form a 32-bit word and writes ito a register. This instruction was introduced to provide backward compatibility for the ARMv8 architecture AArch32 state.

Note

LDAEXH imposes ordering restrictions on memory accesses to the same shareability domain.

For further information, see Section E2.7.3 in the ARMv8 *Architecture Reference Manual* (ARM DDI 0487)

Syntax

```
LDAEXH{cond} <Rt,> [<Rn>]
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

q specifies optional assembler qualifiers on the instruction. The following qualifiers are defined:

- .N(narrow), specifies that the assembler must select a 16-bit encoding for the instruction.

- `.W`(wide), specifies that the assembler must select a 32-bit encoding for the intruction.

If neither `.W` nor `.N` are specified, the assembler can select either 16-bit or 32-bit encodings. if both are available, it must select a 16-bit encoding.

`Rt` is the destination register.

`Rn` is the base register.

A.1.41 LDAH

`LDAH` (Load-Acquire Halfword) loads a halfword from memory, zero-extends it to form a 32-bit word and writes it to a register. This instruction was introduced to provide backward compatibility for the ARMv8 architecture AArch32 state.

Note

`LDAH` imposes ordering restrictions on memory accesses to the same shareability domain.

For further information, see Section E2.7.3 in the ARMv8 *Architecture Reference Manual* (ARM DDI 0487)

Syntax

```
LDAH{cond}{q} <Rt>, [<Rn>]
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`q` specifies optional assembler qualifiers on the instruction. The following qualifiers are defined:

- `.N`(narrow), specifies that the assembler must select a 16-bit encoding for the instruction.
- `.W`(wide), specifies that the assembler must select a 32-bit encoding for the intruction.

If neither `.W` nor `.N` are specified, the assembler can select either 16-bit or 32-bit encodings. if both are available, it must select a 16-bit encoding.

`Rd` is the destination register.

`Rm` is the base register.

A.1.42 LDC

`LDC` (Load Coprocessor Registers) reads a coprocessor register from memory (or multiple registers, if `L` is specified).

Syntax

```
LDC{L}{cond} coproc, CRd, [Rn]
LDC{L}{cond} coproc, CRd, [Rn, #{-}offset]{!}
LDC{L}{cond} coproc, CRd, [Rn], #{-}offset
LDC{L}{cond} coproc, CRd, label
```

where:

`L` specifies that more than one register can be transferred (called a long transfer). The length of the transfer is determined by the coprocessor, but cannot be more than 16 words.

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

coproc is the name of the coprocessor the instruction is for. This is usually of the form pn, where n is an integer in the range 0 to 15.

CRd is the coprocessor register to be stored.

Rn is the register holding the base address for the memory operation.

Offset is a multiple of 4, in the range 0-1020, to be added or subtracted from Rn. If ! is present, the address including the offset is written back into Rn.

label is a word-aligned PC-relative address label.

A.1.43 LDC2

LDC2 (Load Coprocessor Registers) reads a coprocessor register from memory (or multiple registers, if L is specified).

Syntax

```
LDC2{L}{cond} coproc, CRd, [Rn]
LDC2{L}{cond} coproc, CRd, [Rn, #{-}offset]{!}
LDC2{L}{cond} coproc, CRd, [Rn], #{-}offset
LDC2{L}{cond} coproc, CRd, label
```

where:

L specifies that more than one register can be transferred (called a long transfer). The length of the transfer is determined by the coprocessor, but cannot be more than 16 words.

cond is an optional condition code. See *Conditional execution* on page 5-3.

coproc is the name of the coprocessor the instruction is for. This is usually of the form pn, where n is an integer in the range 0 to 15.

CRd is the coprocessor register to be stored.

Rn is the register holding the base address for the memory operation.

Offset is a multiple of 4, in the range 0 – 1020, to be added or subtracted from Rn. If ! is present, the address including the offset is written back into Rn.

label is a word-aligned PC-relative address label.

A.1.44 LDM

LDM (Load Multiple registers) loads one or more registers from consecutive addresses in memory at an address specified in a base register.

Syntax

```
LDM{addr_mode}{cond} Rn{!},reglist{^}
```

where:

addr_mode is one of:

- IA – Increment address After each transfer. This is the default, and can be omitted.
- IB – Increment address Before each transfer (ARM only).
- DA – Decrement address After each transfer (ARM only).

- DB – Decrement address Before each transfer.

It is also possible to use the corresponding stack oriented addressing modes (FD, ED, EA, FA). For example LDMFD is a synonym of LDMDB.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rn is the base register, giving the initial address for the transfer.

! if present, specifies that the final address is written back into Rn.

Reglist is a list of one or more registers to be stored, enclosed in braces. It can contain register ranges. It must be comma separated if it contains more than one register or register range.

^ if specified (in a mode other than User or System) means one of two possible special actions will be taken:

- Data is transferred into the User mode registers instead of the current mode registers (in the case where Reglist does not contain the PC).
- If Reglist does contain the PC, the normal multiple register transfer happens and the SPSR is copied into the CPSR. This is used for returning from exception handlers.

A.1.45 LDR

LDR (Load Register) loads a value from memory to an ARM register, optionally updating the register used to give the address.

A variety of addressing options are provided. For full details of the available addressing modes, see *Addressing modes* on page 5-13.

Syntax

```
LDR{type}{T}{cond} Rt, [Rn {, #offset}]
LDR{type}{cond} Rt, [Rn, #offset]!
LDR{type}{T}{cond} Rt, [Rn], #offset
LDR{type}{cond} Rt, [Rn, +/-Rm {, shift}]
LDR{type}{cond} Rt, [Rn, +/-Rm {, shift}]!
LDR{type}{T}{cond} Rt, [Rn], +/-Rm {, shift}
```

where:

type can be any one of:

- B – unsigned Byte. (Zero extend to 32 bits on loads.)
- SB – signed Byte. (Sign extend to 32 bits.)
- H – unsigned Halfword. (Zero extend to 32 bits on loads.)
- SH – signed Halfword. (Sign extend to 32 bits.)

or omitted, for a Word load.

T specifies that memory is accessed as if the processor was in User mode (not available in all addressing modes).

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rn is the register holding the base address for the memory operation.

! if present, specifies that the final address is written back into Rn.

offset is a numeric value.

Rm is a register holding an offset value to be applied.

shift is either a register or immediate based shift to apply to the offset value.

A.1.46 LDR (pseudo-instruction)

LDR (Load Register) pseudo-instruction loads a register with a 32-bit immediate value or an address. It generates either a MOV or MVN instruction (if possible), or a PC-relative LDR instruction that reads the constant from the literal pool.

Syntax

```
LDR{cond}{.W} Rt, =expr
LDR{cond}{.W} Rt, label_expr
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

.W specifies that a 32-bit Thumb instruction must be used.

Rt is the register to load.

expr is a numeric value.

label_expr is a label, optionally plus or minus a numeric value.

A.1.47 LDRD

LDRD (Load Register Dual) calculates an address from a base register value and a register offset, loads two words from memory, and writes them to two registers.

Syntax

```
LDRD{cond} Rt, Rt2, [{Rn},+/-{Rm}]{!}
LDRD{cond} Rt, Rt2, [{Rn}],+/-{Rm}
```

where:

cond is an optional condition code. See See *Conditional execution* on page 5-3.

Rt is the first destination register. This register must be even-numbered and not R14.

Rt is the second destination register. This register must be <R(t+1)>.

Rn is the base register. The SP or the PC can be used.

+/- is + or omitted if the value of <Rm> is to be added to the base register value (add == TRUE), or – if it is to be subtracted (add == FALSE).

Rm contains the offset that is applied to the value of <Rn> to form the address.

A.1.48 LDREX

LDREX (Load register exclusive). Performs a load from a location and marks it for exclusive access. Byte, halfword, word and doubleword variants are provided.

Syntax

```
LDREX{cond} Rt, [Rn {, #offset}]
LDREXB{cond} Rt, [Rn]
```

```
LDREXH{cond} Rt, [Rn]
LDREXD{cond} Rt, Rt2, [Rn]
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`Rt` is the register to load.

`Rt2` is the second register for doubleword loads.

`Rn` is the register holding the address.

`offset` is an optional value, permitted in Thumb only.

A.1.49 LEAVEX

`LEAVEX` causes a change from ThumbEE state to Thumb state, or has no effect in Thumb state. It is not available in the ARM instruction set.

Syntax

```
LEAVEX
```

A.1.50 LSL

`LSL` (Logical Shift Left) shifts the value in `Rm` left by the specified number of bits, inserting zeros into the vacated bit positions.

Syntax

```
LSL{S}{cond} Rd, Rm, Rs
LSL{S}{cond} Rd, Rm, imm
```

where:

`S` (if specified) means that the condition code flags will be updated depending on the result of the instruction.

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`Rd` is the destination register.

`Rm` is the register holding the operand to be shifted.

`Rs` is the register that holds a shift value to apply to the value in `Rm`. Only the least significant byte of the register is used.

imm is a shift amount, in the range 0-31.

A.1.51 LSR

`LSR` (Logical Shift Right) shifts the value in `Rm` right by the specified number of bits, inserting zeros into the vacated bit positions.

Syntax

```
LSR{S}{cond} Rd, Rm, Rs
LSR{S}{cond} Rd, Rm, imm
```

where:

S (if specified) means that the condition code flags will be updated depending on the result of the instruction.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm is the register holding the operand to be shifted.

Rs is the register that holds a shift value to apply to the value in Rm. Only the least significant byte of the register is used.

imm is a shift amount, in the range 1-32.

A.1.52 MCR

MCR (Move to Coprocessor from Register) writes a coprocessor register, from an ARM register. The purpose of this instruction is defined by the coprocessor implementer.

Syntax

```
MCR{cond} coproc, #opcode1, Rt, CRn, CRm{, #opcode2}
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rt is the ARM register to be transferred.

coproc is the name of the coprocessor the instruction is for. This is usually of the form pn, where n is an integer in the range 0 to 15.

opcode1 is a 4-bit coprocessor-specific opcode.

opcode2 is an optional 3-bit coprocessor-specific opcode.

CRn, CRm are coprocessor registers.

A.1.53 MCR2

MCR2 (Move to Coprocessor from Register) writes a coprocessor register, from an ARM register. The purpose of this instruction is defined by the coprocessor implementer.

Syntax

```
MCR2{cond} coproc, #opcode1, Rt, CRn, CRm{, #opcode2}
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rt is the ARM register to be transferred.

coproc is the name of the coprocessor the instruction is for. This is usually of the form pn, where n is an integer in the range 0 to 15.

opcode1 is a 4-bit coprocessor-specific opcode.

opcode2 is an optional 3-bit coprocessor-specific opcode.

CRn, CRm are coprocessor registers.

A.1.54 MCRR

MCRR (Move to Coprocessor from Registers) transfers a pair of ARM register to a coprocessor. The purpose of this instruction is defined by the coprocessor implementer.

Syntax

```
MCRR{cond} coproc, #opcode3, Rt, Rt2, CRm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rt and Rt2 are the ARM registers to be transferred.

coproc is the name of the coprocessor the instruction is for. This is usually of the form pn, where n is an integer in the range 0 to 15.

CRm is a coprocessor register.

Opcode3 is an optional 4-bit coprocessor-specific opcode.

A.1.55 MCRR2

MCRR2 (Move to Coprocessor from Registers) transfers a pair of ARM register to a coprocessor. The purpose of this instruction is defined by the coprocessor implementer.

Syntax

```
MCRR2{cond} coproc, #opcode3, Rt, Rt2, CRm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rt and Rt2 are the ARM registers to be transferred.

coproc is the name of the coprocessor the instruction is for. This is usually of the form pn, where n is an integer in the range 0 to 15.

CRm is a coprocessor register.

Opcode3 is an optional 4-bit coprocessor-specific opcode.

A.1.56 MLA

MLA (Multiply Accumulate) multiplies Rn and Rm, adds the value from Ra, and stores the least significant 32 bits of the result in Rd.

Syntax

```
MLA{S}{cond} Rd, Rn, Rm, Ra
```

where:

S (if specified) means that the condition code flags will be updated depending on the result of the instruction.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register holding the first multiplicand.

Rm is the register holding the second multiplicand.

Ra is the register holding the accumulate value.

A.1.57 MLS

MLS (Multiply and Subtract) multiplies Rn and Rm, subtracts the result from Ra, and stores the least significant 32 bits of the final result in Rd.

Syntax

```
MLS{S}{cond} Rd, Rn, Rm, Ra
```

where:

S (if specified) means that the condition code flags will be updated depending on the result of the instruction.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register holding the first multiplicand.

Rm is the register holding the second multiplicand.

Ra is the register holding the accumulate value.

A.1.58 MOV

MOV (Move) copies the value of Operand2 into Rd.

Syntax

```
MOV{S}{cond} Rn, <Operand2>
MOV{cond} Rd, #imm16
```

where:

S (if specified) means that the condition code flags will be updated depending on the result of the instruction.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Operand2 is a flexible second operand. See Section 6.2.1.

imm16 is an immediate value in the range 0-65535.

A.1.59 MOVT

MOVT (Move Top) writes imm16 to Rd[31:16]. It does not affect Rd[15:0].

Syntax

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

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Operand2 is a flexible second operand. See Section 6.2.1.

imm16 is an immediate value in the range 0-65535.

A.1.60 MOV32

MOV32 is a pseudo-instruction that loads a register with a 32-bit immediate value or address. It generates two instructions, a MOV, MOVT pair.

Syntax

```
MOV32 Rd, expr
```

where:

Rd is the destination register.

expr is a 32-bit constant, or address label.

A.1.61 MRC

MRC (Move to Register from Coprocessor) reads a coprocessor register to an ARM register. The purpose of this instruction is defined by the coprocessor implementer.

Syntax

```
MRC{cond} coproc, #opcode1, Rt, CRn, CRm{, #opcode2}
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rt is the ARM register to be transferred.

coproc is the name of the coprocessor the instruction is for. This is usually of the form pn, where n is an integer in the range 0 to 15.

opcode1 is a 4-bit coprocessor-specific opcode.

opcode2 is an optional 3-bit coprocessor-specific opcode.

CRn, CRm are coprocessor registers.

A.1.62 MRC2

MRC2 (Move to Register from Coprocessor) reads a coprocessor register to an ARM register. The purpose of this instruction is defined by the coprocessor implementer.

Syntax

```
MRC2{cond} coproc, #opcode1, Rt, CRn, CRm{, #opcode2}
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rt is the ARM register to be transferred.

coproc is the name of the coprocessor the instruction is for. This is usually of the form pn, where n is an integer in the range 0 to 15.

opcode1 is a 4-bit coprocessor-specific opcode.

opcode2 is an optional 3-bit coprocessor-specific opcode.

CRn, CRm are coprocessor registers.

A.1.63 MRRC

MRRC (Move to Registers from Coprocessor) transfers a value from a Coprocessor to a pair of ARM registers. The purpose of this instruction is defined by the Coprocessor implementer.

Syntax

```
MRRC{cond} coproc, #opcode3, Rt, Rt2, CRm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3. MRRC instructions might not specify a condition code in ARM state.

Rt and Rt2 are the ARM registers to be transferred.

coproc is the name of the coprocessor the instruction is for. This is usually of the form pn, where n is an integer in the range 0 to 15.

CRm is a coprocessor register.

Opcode3 is an optional 4-bit coprocessor-specific opcode.

A.1.64 MRRC2

MRRC2 (Move to Registers from Coprocessor) transfers a value from a Coprocessor to a pair of ARM registers. The purpose of this instruction is defined by the Coprocessor implementer.

Syntax

```
MRRC2{cond} coproc, #opcode3, Rt, Rt2, CRm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3. MRRC2 instructions might not specify a condition code in ARM state.

Rt and Rt2 are the ARM registers to be transferred.

coproc is the name of the coprocessor the instruction is for. This is usually of the form pn, where n is an integer in the range 0 to 15.

CRm is a coprocessor register.

Opcode3 is an optional 4-bit coprocessor-specific opcode.

A.1.65 MRS

MRS (Move Status register or Coprocessor Register to General purpose register) can be used to read the CPSR/APSR, CP14 or CP15 Coprocessor registers.

Syntax

```
MRS{cond} Rd, psr
MRS{cond} Rn, coproc_register
MRS{cond} APSR_nzcv, DBGDSCRint
MRS{cond} APSR_nzcv, FPSCR
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`Rd` is the destination register.

`psr` is one of: APSR, CPSR or SPSR.

`coproc_register` is the name of a CP14 or CP15 readable register.

`DBGDSCRint` is the name of a CP14 register that can be copied to the APSR.

A.1.66 MSR

`MSR` (Move Status Register or Coprocessor Register from General Purpose Register) can be used to write all or part of the CPSR/APSR or CP14 or CP15 registers.

Use of the `MSR` instruction to set the endianness bit in the CPSR in User Mode is deprecated. ARM strongly recommends that software executing in User Mode uses the `SETEND` instruction.

Syntax

```
MSR{cond} APSR_flags, Rm
MSR{cond} coproc_register
MSR{cond} APSR_flags, #constant
MSR{cond} psr_fields, #constant
MSR{cond} psr_fields, Rm
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

Rm and Rn are the source registers.

`flags` can be one or more of nzcvq (ALU flags) or g (SIMD flags).

`coproc_register` is the name of a CP14 or CP15 readable register.

`constant` is an 8-bit pattern rotated by an even number of bits within a 32-bit word. (Not available in Thumb.)

`psr` is one of: APSR, CPSR or SPSR.

`fields` is one or more of:

- `c` control field mask byte, PSR[7:0]
- `x` extension field mask byte, PSR[15:8]
- `s` status field mask byte, PSR[23:16]
- `f` flags field mask byte, PSR[31:24].

A.1.67 MUL

`MUL` (Multiply) Multiplies `Rn` and `Rm`, and stores the least significant 32 bits of the result in `Rd`.

Syntax

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

where:

S (if specified) means that the condition code flags will be updated depending on the result of the instruction.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register holding the first multiplicand.

Rm is the register holding the second multiplicand.

A.1.68 MVN

MVN (Move Not) performs a bitwise NOT operation on the operand2 value, and places the result into Rd.

Syntax

```
MVN{S}{cond} Rn, <Operand2>
```

where:

S (if specified) means that the condition code flags will be updated depending on the result of the instruction.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Operand2 is a flexible second operand. See *Operand 2 and the barrel shifter* on page 5-7.

A.1.69 NOP

NOP (No Operation) does nothing.

Syntax

```
NOP{cond}
```

where:

NOP does not have to consume clock cycles. It can be removed by the processor pipeline. It is used for padding, to ensure following instructions align to a boundary.

A.1.70 ORN

ORN (OR NOT) performs an OR operation on the bits in Rn with the complement of the corresponding bits in the value of Operand2.

Syntax

```
ORN{S}{cond} {Rd,} Rn, <Operand2>
```

where:

S (if specified) means that the condition code flags will be updated depending on the result of the instruction.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register holding the first operand.

Operand2 is a flexible second operand. See *Operand 2 and the barrel shifter* on page 5-7.

A.1.71 ORR

Performs an OR operation on the bits in Rn with the corresponding bits in the value of Operand2.

Syntax

```
ORR{S}{cond} {Rd,} Rn, <Operand2>
```

where:

S (if specified) means that the condition code flags will be updated depending on the result of the instruction.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register holding the first operand.

Operand2 is a flexible second operand. See *Operand 2 and the barrel shifter* on page 5-7.

A.1.72 PKHBT

PKHBT (Pack Halfword Bottom Top) combines bits[15:0] of Rn with bits[31:16] of the shifted value from Rm.

Syntax

```
PKHBT{cond} {Rd,} Rn, Rm{, LSL #leftshift}
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register holding the first operand.

Rm is the register holding the second operand.

leftshift is a number in the range 0-31.

A.1.73 PKHTB

PKHTB (Pack Halfword Top Bottom) combines bits[31:16] of Rn with bits[15:0] of the shifted value from Rm.

Syntax

```
PKHTB{cond} {Rd,} Rn, Rm {, ASR #rightshift}
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`Rd` is the destination register.

`Rn` is the register holding the first operand.

`Rm` is the register holding the second operand.

`rightshift` is a number in the range 1-32.

A.1.74 PLD

`PLD` (Preload Data) is a hint instruction that can cause data to be preloaded into the cache.

Syntax

```
PLD{cond} [Rn {, #offset}]
PLD{cond} [Rn, +/-Rm {, shift}]
PLD{cond} label
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`Rn` is a base address.

`offset` is an immediate value that defaults to 0 if not specified.

`Rm` contains an offset value and must not be PC (or SP, in Thumb state).

`shift` is an optional shift.

`label` is a PC-relative expression.

A.1.75 PLDW

`PLDW` (Preload data with intent to write) is a hint instruction that can cause data to be preloaded into the cache. It is available only in processors that implement multi-processing extensions.

Syntax

```
PLDW{cond} [Rn {, #offset}]
PLDW{cond} [Rn, +/-Rm {, shift}]
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`Rn` is a base address.

`offset` is an immediate value that defaults to 0 if not specified.

`Rm` contains an offset value and must not be PC (or SP, in Thumb state).

`shift` is an optional shift.

A.1.76 PLI

`PLI` (Preload instructions) is a hint instruction that can cause instructions to be preloaded into the cache.

Syntax

```
PLI{cond} [Rn {, #offset}]
PLI{cond} [Rn, +/-Rm {, shift}]
PLI{cond} label
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`Rn` is a base address.

`offset` is an immediate value that defaults to 0 if not specified.

`Rm` contains an offset value and must not be PC (or SP, in Thumb state).

`shift` is an optional shift.

`label` is a PC-relative expression.

A.1.77 POP

`POP` is used to pop registers off a full descending stack. `POP` is a synonym for `LDMIA sp!, reglist`.

Syntax

```
POP{cond} reglist
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`reglist` is a list of one or more registers, enclosed in braces.

A.1.78 PUSH

`PUSH` is used to push registers on to a full descending stack. `PUSH` is a synonym for `STMDB sp!, reglist`.

Syntax

```
PUSH{cond} reglist
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`reglist` is a list of one or more registers, enclosed in braces.

A.1.79 QADD

`QADD` (Saturating signed Add) does a signed addition and saturates the result to the signed range $-2^{31} \leq x \leq 2^{31}-1$. If saturation occurs, the Q flag is set.

Syntax

```
QADD{cond} {Rd,} Rm, Rn
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the register holding the operands.

A.1.80 QADD8

QADD8 (Saturating signed bytewise Add) does a signed bytewise addition (4 adds) and saturates the results to the signed range $-2^7 \leq x \leq 2^7-1$. The Q flag is not affected by this instruction.

Syntax

```
QADD8{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

A.1.81 QADD16

QADD16 (Saturating signed bytewise Add) does a signed halfword-wise addition (2 adds) and saturates the results to the signed range $-2^7 \leq x \leq 2^7-1$. The Q flag is not affected by this instruction.

Syntax

```
QADD16{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

A.1.82 QASX

QASX (Saturating signed Add Subtract eXchange) exchanges halfwords of Rm, then adds the top halfwords and subtracts the bottom halfwords and saturates the results to the signed range $-2^{15} \leq x \leq 2^{15}-1$. The Q flag is not affected by this instruction.

Syntax

```
QASX{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

A.1.83 QDADD

QADD (Saturating signed Add) does a signed doubling addition and saturates the result to the signed range $-2^{31} \leq x \leq 2^{31}-1$. If saturation occurs, the Q flag is set.

Syntax

```
QDADD{cond} {Rd,} Rm, Rn
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

The value in Rn is multiplied by 2, saturated and then added to the value in Rm. A second saturate operation is then performed.

A.1.84 QDSUB

QDSUB (Saturating signed Doubling Subtraction) does a signed doubling subtraction and saturates the result to the signed range $-2^{31} \leq x \leq 2^{31}-1$. If saturation occurs, the Q flag is set.

Syntax

```
QDSUB{cond} {Rd,} Rm, Rn
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

The value in Rn is multiplied by 2, saturated and then subtracted from the value in Rm. A second saturate operation is then performed.

A.1.85 QSAX

QSAX (Saturating signed Subtract Add Exchange) exchanges the halfwords of Rm, then subtracts the top halfwords and adds the bottom halfwords and saturates the results to the signed range $-2^{15} \leq x \leq 2^{15}-1$. The Q flag is not affected by this instruction.

Syntax

```
QSAX{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

A.1.86 QSUB

QSUB (Saturating signed Subtraction) does a signed subtraction and saturates the result to the signed range $-2^{31} \leq x \leq 2^{31}-1$. If saturation occurs, the Q flag is set.

Syntax

```
QDSUB{cond} {Rd,} Rm, Rn
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

The value in Rn is subtracted from the value in Rm. A saturate operation is then performed.

A.1.87 QSUB8

QSUB8 (Saturating signed bytewise Subtract) does bytewise subtraction (4 subtracts), with saturation of the results to the signed range $-2^{7} \leq x \leq 2^{7}-1$. The Q flag is not affected by this instruction.

Syntax

```
QSUB8{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

A.1.88 QSUB16

QSUB16 (Saturating signed halfword Subtract) does halfword-wise subtraction (2 subtracts), with saturation of the results to the signed range $-2^{15} \leq x \leq 2^{15}-1$. The Q flag is not affected by this instruction.

Syntax

```
QSUB16{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

A.1.89 RBIT

RBIT (Reverse bits) reverses the bit order in a 32-bit word.

Syntax

```
RBIT{cond} Rd, Rn
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`Rd` is the destination register.

`Rn` is the register holding the operand.

A.1.90 REV

`REV` (Reverse) converts 32-bit big-endian data into little-endian data, or 32-bit little-endian data into big-endian data.

Syntax

```
REV{cond} {Rd}, Rn
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`Rd` is the destination register.

`Rn` is the register holding the operand.

A.1.91 REV16

`REV16` (Reverse byte order halfwords) converts 16-bit big-endian data into little-endian data, or 16-bit little-endian data into big-endian data.

Syntax

```
REV16{cond} {Rd}, Rn
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`Rd` is the destination register.

`Rn` is the register holding the operand.

A.1.92 REVSH

`REVSH` (Reverse byte order halfword, with sign extension) does a reverse byte order of the bottom halfword, and sign extends the result to 32 bits.

Syntax

```
REVSH{cond} Rd, Rn
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`Rd` is the destination register.

Rn is the register holding the operand.

A.1.93 RFE

RFE (Return from Exception) is used to return from an exception where the return state was saved with SRS. If ! is specified, the final address is written back into Rn.

Syntax

```
RFE{addr_mode}{cond} Rn{!}
```

where:

addr_mode is one of:

- IA – Increment address After each transfer. This is the default, and can be omitted.
- IB – Increment address Before each transfer (ARM only).
- DA – Decrement address After each transfer (ARM only).
- DB – Decrement address Before each transfer.

cond is an optional condition codes. See *Conditional execution* on page 5-3, and is permitted only in Thumb, using a preceding IT instruction.

Rn specifies the base register.

A.1.94 ROR

ROR (Rotate right Register) rotates a value in a register by a specified number of bits. The bits that are rotated off the right end are inserted into the vacated bit positions on the left.

Syntax

```
ROR{S}{cond} {Rd,} Rm, Rs
ROR{S}{cond} {Rd,} Rm, imm
```

where:

S (if specified) means that the condition code flags will be updated depending on the result of the instruction.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register holding the operand.

Rm is the register holding the operand to be shifted.

Rs is the register that holds a shift value to apply to the value in Rm. Only the least significant byte of the register is used.

imm is a shift amount, in the range 1 – 31.

A.1.95 RRX

RRX (Rotate Right with extend) performs a shift right one bit on a register value. The old carry flag is shifted into bit[31]. If the S suffix is present, the old bit[0] is placed in the carry flag.

Syntax

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

where:

S (if specified) means that the condition code flags will be updated depending on the result of the instruction.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm is the register holding the operand to be shifted.

A.1.96 RSB

RSB (Reverse Subtract) subtracts the value in Rn from the value of Operand2. This is useful because Operand2 has more options than Operand1 (which is always a register).

Syntax

```
RSB{S}{cond} {Rd,} Rn, <Operand2>
```

where:

S (if specified) means that the condition code flags will be updated depending on the result of the instruction.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register holding the first operand.

Operand2 is a flexible second operand. See *Operand 2 and the barrel shifter* on page 5-7.

A.1.97 RSC

RSC (Reverse Subtract with Carry) subtracts Rn from Operand2. If the carry flag is clear, the result is reduced by one.

Syntax

```
RSC{S}{cond} {Rd,} Rn, <Operand2>
```

where:

S (if specified) means that the condition code flags will be updated depending on the result of the instruction.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register holding the first operand.

Operand2 is a flexible second operand. See *Operand 2 and the barrel shifter* on page 5-7.

A.1.98 SADD8

SADD8 (Signed bytewise Add) does a signed bytewise addition (4 adds).

Syntax

```
SADD8{cond} {Rd,} Rn, Rm
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`Rd` is the destination register.

`Rm` and `Rn` are the registers holding the operands.

A.1.99 SADD16

`SADD16` (Signed bytewise Add) does a signed halfword-wise addition (2 adds).

Syntax

```
SADD16{cond} {Rd,} Rn, Rm
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`Rd` is the destination register.

`Rm` and `Rn` are the registers holding the operands.

A.1.100 SASX

`SASX` (Signed Add Subtract Exchange) exchanges halfwords of `Rm`, then adds the top halfwords and subtracts the bottom halfwords.

Syntax

```
SASX{cond} {Rd,} Rn, Rm
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`Rd` is the destination register.

`Rm` and `Rn` are the registers holding the operands.

A.1.101 SBC

`SBC` (Subtract with Carry) subtracts the value of `Operand2` from the value in `Rn`. If the carry flag is clear, the result is reduced by one.

Syntax

```
SBC{S}{cond} {Rd,} Rn, <Operand2>
```

where:

`S` (if specified) means that the condition code flags will be updated depending on the result of the instruction.

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register holding the first operand.

Operand2 is a flexible second operand. See *Operand 2 and the barrel shifter* on page 5-7.

A.1.102 SBFX

SBFX (Signed Bit Field Extract) writes adjacent bits from one register into the least significant bits of a second register and sign extends to 32 bits.

Syntax

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

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register that contains the bits to be extracted.

lsb specifies the least significant bit of the bitfield.

width is the width of the bitfield.

A.1.103 SDIV

SDIV (Signed Divide). divides a 32-bit signed integer register value by a 32-bit signed integer register value, and writes the result to the destination register. This instruction is not present in all variants of the ARMv7-A architecture.

Syntax

```
SDIV{cond}{q} {Rd,} Rn, Rm
```

where:

cond is the optional condition code. See *Conditional execution* on page 5-3.

q specifies optional assembler qualifiers on the instruction. The following qualifiers are defined:

- .N (narrow), specifies that the assembler must select a 16-bit encoding for the instruction.
- .W (wide), specifies that the assembler must select a 32-bit encoding for the intruction.

If neither .W nor .N are specified, the assembler can select either 16-bit or 32-bit encodings. If both are available, it must select a 16-bit encoding.

Rd the destination register.

Rn is the register that contains the dividend.

Rm is the register that contains the divisor.

A.1.104 SEL

SEL (Select) selects bytes from Rn or Rm, depending on the APSR GE flags.

If GE[0] is set, Rd[7:0] comes from Rn[7:0], else from Rm[7:0].

If GE[1] is set, Rd[15:8] comes from Rn[15:8], else from Rm[15:8].

If GE[2] is set, Rd[23:16] comes from Rn[23:16], else from Rm[23:16].

If GE[3] is set, Rd[31:24] comes from Rn[31:24], else from Rm[31:24].

Syntax

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

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register that contains the bits to be extracted.

Rm is the register holding the second operand.

A.1.105 SETEND

SETEND (Set endianness) selects little-endian or big-endian memory access. See *Endianness* on page 14-2 for more details.

Syntax

```
SETEND LE
SETEND BE
```

A.1.106 SEV

SEV (Send Event) causes an event to be signaled to all cores in an MPCore. See *Assembly language power instructions* on page 20-8 for more detail.

Syntax

```
SEV{cond}
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

A.1.107 SHADD8

SHADD8 (Signed halving bytewise Add) does a signed bytewise addition (4 adds) and halves the results.

Syntax

```
SHADD8{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

A.1.108 SHADD16

SHADD16 (Signed halving bytewise Add) does a signed halfword-wise addition (2 adds) and halves the results.

Syntax

```
SHADD16{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

A.1.109 SHASX

SHASX (Signed Halving Add Subtract Exchange) exchanges halfwords of Rm, then adds the top halfwords and subtracts the bottom halfwords and halves the results.

Syntax

```
SHASX{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

A.1.110 SHSAX

SHSAX (Signed Halving Subtract Add Exchange) exchanges halfwords of Rm, then subtracts the top halfwords and adds the bottom halfwords and halves the results.

Syntax

```
SHSAX{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

A.1.111 SHSUB8

SHSUB8 (Signed halving bytewise subtraction) does a signed bytewise subtraction (4 subtracts) and halves the results.

Syntax

```
SHSUB8{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands

A.1.112 SHSUB16

SHSUB16 (Signed Halving halfword-wise Subtract) does a signed halfword-wise subtraction (2 subtracts) and halves the result.

Syntax

```
SHSUB16{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the register holding the operands.

A.1.113 SMC

SMC (Secure Monitor Call) is used by the ARM Security Extensions. This instruction was formerly called SMI. See Chapter 21 *Security* for more details.

Syntax

```
SMC{cond} #imm4
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

imm4 is an immediate value in the range 0-15, which is ignored by the processor, but can be used by the SMC exception handler.

A.1.114 SMLAxy

The SMLAxy (Signed Multiply Accumulate; 32 <= 32 + 16 × 16) instruction multiplies the 16-bit signed integers from the selected halves of Rn and Rm, adds the 32-bit result to the value from Ra, and writes the result in Rd.

Syntax

```
SMLA<x><y>{cond} Rd, Rn, Rm, Ra
```

where:

<x> and <y> can be either B or T. B means use the bottom half (bits [15:0]) of a register, T means use the top half (bits [31:16]) of a register. <x> specifies which half of Rn to use, <y> does the same for Rm.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register holding the first multiplicand.

Rm is the register holding the second multiplicand.

Ra is the register that holds the accumulate value.

A.1.115 SMLAD

SMLAD (Dual Signed Multiply Accumulate; 32 <= 32 + 16 × 16 + 16 × 16) multiplies the bottom halfword of Rn with the bottom halfword of Rm, and the top halfword of Rn with the top halfword of Rm. It then adds both products to the value in Ra and writes the sum to Rd.

Syntax

```
SMLAD{X}{cond} Rd, Rn, Rm, Ra
```

where:

{X} if present, means that the most and least significant halfwords of the second operand are swapped, before the multiplication.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register holding the first multiplicand.

Rm is the register holding the second multiplicand.

Ra is the register that holds the accumulate value.

A.1.116 SMLAL

SMLAL (Signed Multiply Accumulate 64 <= 64 + 32 × 32) multiplies Rn and Rm (treated as signed integers) and adds the 64-bit result to the 64-bit signed integer contained in RdHi and RdLo.

Syntax

```
SMLAL{S}{cond} RdLo, RdHi, Rn, Rm
```

where:

S (if specified) means that the condition code flags will be updated depending on the result of the instruction.

cond is an optional condition code. See *Conditional execution* on page 5-3.

RdLo and RdHi are the destination registers.

Rn is the register holding the first multiplicand.

Rm is the register holding the second multiplicand.

A.1.117 SMLALxy

SMLALxy (Signed Multiply Accumulate; 64 <= 64 + 16 × 16) multiplies the signed integer from the selected half of Rm by the signed integer from the selected half of Rn, and adds the 32-bit result to the 64-bit value in RdHi and RdLo.

Syntax

```
SMLAL<x><y>{cond} RdLo, RdHi, Rn, Rm
```

where:

<x> and <y> can be either B or T. B means use the bottom half (bits [15:0]) of a register, T means use the top half (bits [31:16]) of a register. <x> specifies which half of Rn to use, <y> does the same for Rm.

cond is an optional condition code. See *Conditional execution* on page 5-3.

RdLo and RdHi are the destination registers.

Rn is the register holding the first multiplicand.

Rm is the register holding the second multiplicand.

A.1.118 SMLALD

SMLALD (Dual Signed Multiply Accumulate Long; 64 <= 64 + 16 × 16 + 16 × 16) multiplies the bottom halfword of Rn with the bottom halfword of Rm, and the top halfword of Rn with the top halfword of Rm and adds both products to the value in RdLo, RdHi and stores the result in RdLo and RdHi.

Syntax

```
SMLALD{X}{cond} RdLo, RdHi Rn, Rm
```

where:

{X} if present, means that the most and least significant halfwords of the second operand are swapped, before the multiplication.

cond is an optional condition code. See *Conditional execution* on page 5-3.

RdLo and RdHi are the destination registers.

Rn is the register holding the first multiplicand.

Rm is the register holding the second multiplicand.

A.1.119 SMLAWy

SMLAW (Signed Multiply with Accumulate Wide; 32 <= 32 × 16 + 32) multiplies the signed integer from the selected half of Rm by the signed integer from Rn, adds the 32-bit result to the 32-bit value in Ra, and writes the result in Rd.

Syntax

```
SMLAW<y>{cond} Rd, Rn, Rm, Ra
```

where:

<y> can be either B or T. B means use the bottom half (bits [15:0]) of Rm, T means use the top half (bits [31:16]) of Rm.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register holding the first multiplicand.

Rm is the register holding the second multiplicand.

Ra is the register that holds the accumulate value.

A.1.120 SMLSLD

SMLSLD (Dual Signed Multiply Subtract Accumulate Long; 64 <= 64 + 16 × 16 - 16 × 16) multiplies Rn[15:0] with Rm[15:0] and Rn[31:16] with Rm[31:16]. It then subtracts the second product from the first, adds the difference to the value in RdLo, RdHi, and writes the result to RdLo, RdHi.

Syntax

```
SMLSLD{X}{cond} RdLo, RdHi Rn, Rm
```

where:

{X} if present, means that the most and least significant halfwords of the second operand are swapped, before the multiplication.

cond is an optional condition code. See *Conditional execution* on page 5-3.

RdLo and RdHi are the destination registers and hold the value to be accumulated.

Rn is the register holding the first multiplicand.

Rm is the register holding the second multiplicand.

A.1.121 SMMLA

SMMLA (Signed top word Multiply with Accumulate; 32 <= top word (32 × 32 + 32)) multiplies Rn and Rm, adds Ra to the most significant 32 bits of the product, and writes the result in Rd.

Syntax

```
SMMLA{R}{cond} Rd, Rn, Rm, Ra
```

where:

R, if present means that 0x80000000 is added before extracting the most significant 32 bits. This rounds the result.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register holding the first multiplicand.

Rm is the register holding the second multiplicand.

Ra is the register holding the accumulate value.

A.1.122 SMMLS

SMMLS (Signed top word Multiply with Subtract; 32 <= top word (32 × 32 - 32)) multiplies Rn and Rm, subtracts the product from the value in Ra shifted left by 32 bits, and stores the most significant 32 bits of the result in Rd.

Syntax

```
SMMLS{R}{cond} Rd, Rn, Rm, Ra
```

where:

R, if present means that 0x80000000 is added before extracting the most significant 32 bits. This rounds the result.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register holding the first multiplicand.

Rm is the register holding the second multiplicand.

Ra is the register holding the accumulate value.

A.1.123 SMMUL

SMMUL (Signed top word Multiply; 32 <= top word (32 × 32)) multiplies Rn and Rm, and writes the most significant 32 bits of the 64-bit result to Rd.

Syntax

```
SMMUL{R}{cond} Rd, Rn, Rm
```

where:

R, if present means that 0x80000000 is added before extracting the most significant 32 bits. This rounds the result.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register holding the first multiplicand.

Rm is the register holding the second multiplicand.

A.1.124 SMUAD

SMUAD (Dual Signed Multiply and Add products) multiplies Rn [15:0] with Rm [15:0] and Rn [31:16] with Rm [31:16]. It then adds the products and stores the sum to Rd.

Syntax

```
SMUAD{X}{cond} Rd, Rn, Rm
```

where:

X, if present means that the most and least significant halfwords of the second operand are exchanged before the multiplications occur.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register holding the first multiplicand.

Rm is the register holding the second multiplicand.

A.1.125 SMUSD

SMUSD (Dual Signed Multiply and Subtract products) multiplies Rn [15:0] with Rm [15:0] and Rn [31:16] with Rm [31:16]. It then subtracts the products and stores the sum to Rd.

Syntax

```
SMUSD{X}{cond} Rd, Rn, Rm
```

where:

X, if present means that the most and least significant halfwords of the second operand are exchanged before the multiplications occur.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register holding the first multiplicand.

Rm is the register holding the second multiplicand.

A.1.126 SMULxy

The SMULxy (Signed Multiply (32 <= 16 × 16) instruction multiplies the 16-bit signed integers from the selected halves of Rn and Rm, and places the 32-bit result in Rd.

Syntax

```
SMUL<x><y>{cond} {Rd}, Rn, Rm
```

where:

<x> and <y> can be either B or T. B means use the bottom half (bits [15:0]) of a register, T means use the top half (bits [31:16]) of a register. <x> specifies which half of Rn to use, <y> does the same for Rm.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register holding the first multiplicand.

Rm is the register holding the second multiplicand.

A.1.127 SMULL

The SMULL (signed multiply long; 64 <= 32 × 32) instruction multiplies Rn and Rm (treated as containing as two's complement signed integers) and places the least significant 32 bits of the result in RdLo, and the most significant 32 bits of the result in RdHi.

Syntax

```
SMULL{S}{cond} RdLo, RdHi, Rn, Rm
```

where:

S (if specified) means that the condition code flags will be updated depending on the result of the instruction.

cond is an optional condition code. See *Conditional execution* on page 5-3.

RdLo and RdHi are the destination registers.

Rn is the register holding the first multiplicand.

Rm is the register holding the second multiplicand.

A.1.128 SMULWy

SMULWy (Signed Multiply Wide; 32 <= 32 × 16) multiplies the signed integer from the chosen half of Rm with the signed integer from Rn, and places the upper 32-bits of the 48-bit result in Rd.

Syntax

```
SMULW<y>{cond} {Rd}, Rn, Rm
```

where:

<y> is either B or T. B means use the bottom half (bits [15:0]) of Rm, T means use the top half (bits [31:16]) of Rm.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register holding the first multiplicand.

Rm is the register holding the second multiplicand.

A.1.129 SRS

SRS (Store Return State) stores the LR and the SPSR of the current mode, at the address contained in the SP of the mode specified by modenum. The optional ! means that the SP value is updated. This is compatible with the normal use of the STM instruction for stack accesses.

Syntax

```
SRS{addr_mode}{cond} sp{!}, #modenum
```

where:

addr_mode is one of:

- IA – Increment address After each transfer. This is the default, and can be omitted.
- IB – Increment address Before each transfer (ARM only).
- DA – Decrement address After each transfer (ARM only).
- DB – Decrement address Before each transfer.

It is also possible to use the corresponding stack oriented addressing modes (FD, ED, EA, FA).

cond is an optional condition code. See *Conditional execution* on page 5-3.

modenum gives the number of the mode whose SP is used.

A.1.130 SSAT

SSAT (Signed Saturate) performs a shift and saturates the result to the signed range $-2^{sat-1} \leq x \leq 2^{sat-1}-1$. If saturation occurs, the Q flag is set.

Syntax

```
SSAT{cond} Rd, #sat, Rm{, shift}
```

where:

<y> is either B or T. B means use the bottom half (bits [15:0]) of Rm, T means use the top half (bits [31:16]) of Rm.

cond is an optional condition code. See *Conditional execution* on page 5-3

Rd is the destination register.

sat specifies the bit position to saturate to, in the range 1 to 32.

Rm is the register holding the second multiplicand.

shift is optional shift amount and can be either ASR #n where n is in the range (1 – 32 ARM state, 1 – 31 Thumb state) or LSL #n where n is in the range (0-31).

A.1.131 SSAT16

SSAT16 (Signed Saturate, parallel halfwords) saturates each signed halfword to the signed range $-2^{sat-1} \leq x \leq 2^{sat-1} -1$. If saturation occurs, the Q flag is set.

Syntax

```
SSAT16{cond} Rd, #sat, Rn
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

sat specifies the bit position to saturate to, in the range 1 to 32.

Rn is the register holding the operand.

A.1.132 SSAX

SSAX (Signed Subtract Add Exchange) exchanges halfwords of Rm, then subtracts the top halfwords and adds the bottom halfwords.

Syntax

```
SSAX{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the register holding the operands.

A.1.133 SSUB8

SSUB8 (Signed halving bytewise Subtraction) does a signed bytewise subtraction (4 subtracts).

Syntax

```
SSUB8{cond} {Rd,} Rn, Rm
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`Rd` is the destination registers.

`Rm` and `Rn` are the register holding the operands.

A.1.134 SSUB16

`SSUB16` (Signed halfword-wise Subtract) does a signed halfword-wise subtraction (2 subtracts).

Syntax

```
SSUB16{cond} {Rd}, Rn, Rm
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`Rd` is the destination registers.

`Rm` and `Rn` are the register holding the operands.

A.1.135 STC

`STC` (Store Coprocessor Registers) writes a coprocessor register to memory (or multiple registers, if `L` is specified).

Syntax

```
STC{L}{cond} coproc, CRd, [Rn]
STC{L}{cond} coproc, CRd, [Rn, #{-}offset]{!}
STC{L}{cond} coproc, CRd, [Rn], #{-}offset
STC{L}{cond} coproc, CRd, label
```

where:

`L` specifies that more than one register can be transferred (called a long transfer). The length of the transfer is determined by the coprocessor, but cannot be more than 16 words.

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`coproc` is the name of the coprocessor the instruction is for. This is usually of the form pn, where n is an integer in the range 0 to 15.

`CRd` is the coprocessor register to be stored.

`Rn` is the register holding the base address for the memory operation.

`offset` is a multiple of four, in the range 0-1020, to be added or subtracted from `Rn`. If `!` is present, the address including the offset is written back into `Rn`.

`label` is a word-aligned PC-relative address label.

A.1.136 STC2

STC2 (Store Coprocessor registers) writes a coprocessor register to memory (or multiple registers, if L is specified).

Syntax

```
STC2{L}{cond} coproc, CRd, [Rn]
STC2{L}{cond} coproc, CRd, [Rn, #{-}offset]{!}
STC2{L}{cond} coproc, CRd, [Rn], #{-}offset
STC2{L}{cond} coproc, CRd, label
```

where:

L specifies that more than one register can be transferred (called a long transfer). The length of the transfer is determined by the coprocessor, but cannot be more than 16 words.

cond is an optional condition code. See *Conditional execution* on page 5-3.

coproc is the name of the coprocessor the instruction is for. This is usually of the form pn, where n is an integer in the range 0 to 15.

CRd is the coprocessor register to be stored.

Rn is the register holding the base address for the memory operation.

offset is a multiple of four, in the range 0 – 1020, to be added or subtracted from Rn. If ! is present, the address including the offset is written back into Rn.

label is a word-aligned PC-relative address label.

A.1.137 STL

STL (Store-Release Word) stores a word from a register to memory. This instruction was introduced to provide backward compatibility for the ARMv8 architecture AArch32 state.

Note

STL imposes ordering restrictions on memory accesses to the same shareability domain.

For further information, see Section E2.7.3 in the ARMv8 *Architecture Reference Manual* (ARM DDI 0487)

Syntax

```
STL{cond}{q} <Rt>, [<Rn>]
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

q specifies optional assembler qualifiers on the instruction. The following qualifiers are defined:

- .N(narrow), specifies that the assembler must select a 16-bit encoding for the instruction.
- .W(wide), specifies that the assembler must select a 32-bit encoding for the intruction.

If neither .W nor .N are specified, the assembler can select either 16-bit or 32-bit encodings. if both are available, it must select a 16-bit encoding.

Rd is the source register.

Rm is the base register.

A.1.138 STLB

STLB (Store-Release Byte) stores a byte from a register to memory. This instruction was introduced to provide backward compatibility for the ARMv8 architecture AArch32 state.

Note

STLB imposes ordering restrictions on memory accesses to the same shareability domain.

For further information, see Section E2.7.3 in the ARMv8 *Architecture Reference Manual* (ARM DDI 0487)

Syntax

```
STLB{cond}{q} <Rt>, [<Rn>]
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

q specifies optional assembler qualifiers on the instruction. The following qualifiers are defined:

- .N(narrow), specifies that the assembler must select a 16-bit encoding for the instruction.
- .W(wide), specifies that the assembler must select a 32-bit encoding for the intruction.

If neither .W nor .N are specified, the assembler can select either 16-bit or 32-bit encodings. if both are available, it must select a 16-bit encoding.

Rt is the source register.

Rn is the base register.

A.1.139 STLEX

STLEX (Store-Release Exclusive Word) stores a word from a register to memory if the executing core has exclusive access to the memory addressed. This instruction was introduced to provide backward compatibility for the ARMv8 architecture AArch32 state.

Note

STLEX imposes ordering restrictions on memory accesses to the same shareability domain.

For further information, see Section E2.7.3 in the ARMv8 *Architecture Reference Manual* (ARM DDI 0487)

Syntax

```
STLEX{cond}{q} <Rd>, <Rt>, [<Rn>]
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

q specifies optional assembler qualifiers on the instruction. The following qualifiers are defined:

- .N(narrow), specifies that the assembler must select a 16-bit encoding for the instruction.
- .W(wide), specifies that the assembler must select a 32-bit encoding for the intruction.

If neither .W nor .N are specified, the assembler can select either 16-bit or 32-bit encodings. if both are available, it must select a 16-bit encoding.

Rd is the destination register for the returned status value. The value returned is:.

0 If the operation updates memory.

1 If the operation fails to update memory.

Rt is the source register.

Rn is the base register.

A.1.140 STLEXB

STLEXB (Store-Release Exclusive Byte) stores a byte from a register to memory if the executing core has exclusive access to the memory addressed. This instruction was introduced to provide backward compatibility for the ARMv8 architecture AArch32 state.

Note

STLEXB imposes ordering restrictions on memory accesses to the same shareability domain.

For further information, see Section E2.7.3 in the ARMv8 *Architecture Reference Manual* (ARM DDI 0487)

Syntax

```
STLEXB{cond}{q} <Rd>, <Rt>, [<Rn>]
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

q specifies optional assembler qualifiers on the instruction. The following qualifiers are defined:

- .N(narrow), specifies that the assembler must select a 16-bit encoding for the instruction.
- .W(wide), specifies that the assembler must select a 32-bit encoding for the intruction.

If neither .W nor .N are specified, the assembler can select either 16-bit or 32-bit encodings. if both are available, it must select a 16-bit encoding.

Rd is the destination register for the returned status value. The value returned is:.

0 If the operation updates memory.

1 If the operation fails to update memory.

Rt is the source register.

Rn is the base register.

A.1.141 STLEXD

STLEXD (Store-Release Exclusive Double) stores a doubleword from two registers to memory. This instruction was introduced to provide backward compatibility for the ARMv8 architecture AArch32 state.

Note

STLEXD imposes ordering restrictions on memory accesses to the same shareability domain.

For further information, see Section E2.7.3 in the ARMv8 *Architecture Reference Manual* (ARM DDI 0487)

Syntax

```
STLEXD{cond}{q} <Rd>, <Rt>, <Rt2>, [<Rn>]
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`q` specifies optional assembler qualifiers on the instruction. The following qualifiers are defined:

- `.N`(narrow), specifies that the assembler must select a 16-bit encoding for the instruction.
- `.W`(wide), specifies that the assembler must select a 32-bit encoding for the intruction.

If neither `.W` nor `.N` are specified, the assembler can select either 16-bit or 32-bit encodings. if both are available, it must select a 16-bit encoding.

`Rd` is the destination register for the returned status value. The value returned is:.

0 If the operation updates memory.

1 If the operation fails to update memory.

`Rt` is the first source register. `Rt` must be an even numbered register and not R14.

`Rt2` is the second destination register. `Rt2` must be `R(t + 1)`.

`Rn` is the base register.

A.1.142 STLEXH

`STLEXH` (Store-Release Exclusive) stores a halfword from a register to memory if the executing core has exclusive access to the memory addressed. This instruction was introduced to provide backward compatibility for the ARMv8 architecture AArch32 state.

Note

`STLEXH` imposes ordering restrictions on memory accesses to the same shareability domain.

For further information, see Section E2.7.3 in the ARMv8 *Architecture Reference Manual* (ARM DDI 0487)

Syntax

```
STLEXH{cond}{q} <Rd>, <Rt>, [<Rn>]
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`q` specifies optional assembler qualifiers on the instruction. The following qualifiers are defined:

- `.N`(narrow), specifies that the assembler must select a 16-bit encoding for the instruction.
- `.W`(wide), specifies that the assembler must select a 32-bit encoding for the intruction.

If neither `.W` nor `.N` are specified, the assembler can select either 16-bit or 32-bit encodings. if both are available, it must select a 16-bit encoding.

Rd is the destination register for the returned status value. The value returned is:.

0 If the operation updates memory.

1 If the operation fails to update memory.

Rt is the source register.

Rm is the base register.

A.1.143 STLH

STLH (Store-Release Halfword) stores a word from a register to memory. This instruction was introduced to provide backward compatibility for the ARMv8 architecture AArch32 state.

Note

STLH imposes ordering restrictions on memory accesses to the same shareability domain.

For further information, see Section E2.7.3 in the ARMv8 *Architecture Reference Manual* (ARM DDI 0487)

Syntax

```
STLH{cond}{q} <Rt>, [<Rn>]
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

q specifies optional assembler qualifiers on the instruction. The following qualifiers are defined:

- .N(narrow), specifies that the assembler must select a 16-bit encoding for the instruction.
- .W(wide), specifies that the assembler must select a 32-bit encoding for the intruction.

If neither .W nor .N are specified, the assembler can select either 16-bit or 32-bit encodings. if both are available, it must select a 16-bit encoding.

Rt is the source register.

Rn is the base register.

A.1.144 STM

STM (Store Multiple registers) writes one or more registers to consecutive addresses in memory to an address specified in a base register.

Syntax

```
STM{addr_mode}{cond} Rn{!},reglist{^}
```

where:

addr_mode is one of:

- IA – Increment address After each transfer. This is the default, and can be omitted.
- IB – Increment address Before each transfer (ARM only).
- DA – Decrement address After each transfer (ARM only).

- DB – Decrement address Before each transfer.

It is also possible to use the corresponding stack oriented addressing modes (FD, ED, EA, FA). For example STMFD is a synonym of STMDB.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rn is the base register, giving the initial address for the transfer.

! if present, specifies that the final address is written back into Rn.

^ if specified (in ARM state and a mode other than User or System) means that data is transferred into or out of the User mode registers instead of the current mode registers.

reglist is a list of one or more registers to be stored, enclosed in braces. It can contain register ranges. It must be comma separated if it contains more than one register or register range.

A.1.145 STR

STR (Store Register) stores a value to memory from an ARM register, optionally updating the register used to give the address.

A variety of addressing options are provided. For full details of the available addressing modes, see *Addressing modes* on page 5-13.

Syntax

```
STR{type}{T}{cond} Rt, [Rn {, #offset}]
STR{type}{cond} Rt, [Rn, #offset]!
STR{type}{T}{cond} Rt, [Rn], #offset
STR{type}{cond} Rt, [Rn, +/-Rm {, shift}]
STR{type}{cond} Rt, [Rn, +/-Rm {, shift}]!
STR{type}{T}{cond} Rt, [Rn], +/-Rm {, shift}
```

where:

type can be any one of:

- B – unsigned Byte (Zero extend to 32 bits on loads.)
- H – unsigned Halfword (Zero extend to 32 bits on loads.)

or omitted, for a Word load.

T specifies that memory is accessed as if the processor was in User mode (not available in all addressing modes).

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rn is the register holding the base address for the memory operation.

! if present, specifies that the final address is written back into Rn.

offset is a numeric value.

Rm is a register holding an offset value to be applied.

shift is either a register or immediate based shift to apply to the offset value.

A.1.146 STRD

STRD (Store Register Dual) calculates an address from a base register value and a register offset, and stores two words from two registers to memory. It can use offset, post-indexed, or pre-indexed addressing.

Syntax

```
STRD{cond} Rt, Rt2, [Rn {,#+/-<imm>}]
STRD{cond} Rt, Rt2, [<Rn>, #+/-<imm>
STRD{cond} Rt, Rt2, [<Rn>, #+/-<imm>]!
STRD{cond} Rt, Rt2, [{Rn},+/-{Rm}]{!}
STRD{cond} Rt, Rt2, [{Rn}],+/-{Rm}
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rt is the first source register. For an ARM instruction Rt must be even-numbered and not R14.

Rt is the second source register. For an ARM instruction Rt2 must be <R(t+1)>.

Rn is the base register. The SP can be used. In the ARM instruction set for offset addressing only, the PC can be used. However, use of the PC is deprecated.

+/- is + or omitted if the value of <Rm> is to be added to the base register value (add = TRUE), or – if it is to be subtracted (add = FALSE). #0 and #-0 generate different instructions.

imm is the immediate offset used to form the address. imm can be omitted, meaning an offset of 0.

Rm contains the offset that is applied to the value of <Rn> to form the address.

A.1.147 STREX

STREX (Store register exclusive). Performs a store to a location marked for exclusive access, returning a status value if the store succeeded. Byte, halfword, word and doubleword variants are provided.

Syntax

```
STREX{cond} Rd, Rt, [Rn {, #offset}]
STREXB{cond} Rd, Rt, [Rn]
STREXH{cond} Rd, Rt, [Rn]
STREXD{cond} Rd, Rt, Rt2, [Rn]
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register for the return status.

Rt is the register to store.

Rt2 is the second register for doubleword stores.

Rn is the register holding the address.

offset is an optional value, permitted in Thumb only.

A.1.148 SUB

SUB (Subtract) subtracts the value Operand2 from Rn (or subtracts imm12 from Rn).

Syntax

```
SUB{S}{cond} {Rd,} Rn, <Operand2>
```

```
SUB{cond}{Rd,}, Rn, #imm12 (Only available in Thumb)
```

where:

S (if specified) means that the condition code flags will be updated depending on the result of the instruction.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register holding the first operand.

Operand2 is a flexible second operand. See *Operand 2 and the barrel shifter* on page 5-7.

imm12 is in the range 0-4095.

A.1.149 SVC

SVC (SuperVisor Call) causes an SVC exception (was called SWI in older documentation).

Syntax

```
SVC{cond} #imm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

imm is an integer in the range 0 - 0xFFFFFF (ARM) or 0 - 0xFF (Thumb). This integer is not used by the processor itself, but can be used by exception handler code.

A.1.150 SWP

SWP (Swap registers and memory) performs the following two actions. Data from memory is loaded into Rt. Rt2 is saved to memory, at the address given by Rn. Use of this instruction is deprecated and its use is disabled by default.

Syntax

```
SWP{B}{cond} Rt, Rt2, [Rn]
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

B is an optional suffix. If specified, a byte is swapped. If not present, a word is specified.

Rt is the destination register.

Rt2 is the source register and can be the same as Rt.

Rn is the register holding the address and cannot be the same as Rt or Rt2.

A.1.151 SXT

SXT (Signed Extend) extracts the specified byte and extends to 32-bit.

Syntax

```
SXT<extend>{cond} {Rd,} Rm {,rotation}
```

where:

extend must be one of:

- B16 – extends two 8-bit values to two 16-bit values.
- B – extends an 8-bit value to a 32-bit value.
- H – extends a 16-bit value to a 32-bit value.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm is the register that contains the value to be extended.

rotation can be one of ROR #8, ROR #16 or ROR #24 (or can be omitted).

A.1.152 SXTA

SXTA (Signed Extend and Add) extracts the specified byte, adds the value from Rn and extends to 32-bit.

Syntax

```
SXTA<extend>{cond} {Rd,} Rn, Rm {,rotation}
```

where:

extend must be one of:

- B16 – extends two 8-bit values to two 16-bit values.
- B – extends an 8-bit value to a 32-bit value.
- H – extends a 16-bit value to a 32-bit value.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register holding the value to be added.

Rm is the register that contains the value to be extended.

rotation can be one of ROR #8, ROR #16 or ROR #24 (or can be omitted).

A.1.153 SYS

SYS (System coprocessor instruction) is used to execute special coprocessor instructions such as cache, branch predictor, and TLB operations. The instructions operate by writing to special write-only coprocessor registers.

Syntax

```
SYS{cond} instruction {,Rn}
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`instruction` is a write-only system coprocessor register name.

`Rn` is the register holding the operand.

A.1.154 TBB

`TBB` (Table Branch Byte) causes a PC-relative forward branch using a table of single byte offsets. `Rn` provides a pointer to the table, and `Rm` supplies an index into the table. The branch length is twice the value of the byte returned from the table. The target of the branch table must be in the same execution state. There is no ARM or 16-bit Thumb version of this instruction.

Syntax

```
TBB [Rn, Rm]
```

where:

`Rn` is the base register that holds the address of the table of branch lengths.

`Rm` is a register that holds the index into the table.

A.1.155 TBH

`TBH` (Table Branch Halfword) causes a PC-relative forward branch using a table of halfword offsets. `Rn` provides a pointer to the table, and `Rm` supplies an index into the table. The branch length is twice the value of the halfword returned from the table. The target of the branch table must be in the same execution state.

There is no ARM or 16-bit Thumb version of this instruction.

Syntax

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

where:

`Rn` is the base register that holds the address of the table of branch lengths.

`Rm` is a register that holds the index into the table.

A.1.156 TEQ

`TEQ` (Test Equivalence) does a bitwise AND operation on the value in `Rn` and the value of `Operand2`. This is the same as an `ANDS` instruction, except that the result is discarded.

Syntax

```
TEQ{cond} Rn, <Operand2>
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

Rn is the register holding the first operand.

Operand2 is a flexible second operand. See *Operand 2 and the barrel shifter* on page 5-7.

A.1.157 TST

TST (Test) does an Exclusive OR operation on the value in Rn and the value of Operand2. This is the same as an EORS instruction, except that the result is discarded.

Syntax

```
TST{cond} Rn, <Operand2>
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rn is the register holding the first operand.

Operand2 is a flexible second operand. See *Operand 2 and the barrel shifter* on page 5-7.

A.1.158 UADD8

UADD8 (Unsigned bytewise Add) does an unsigned bytewise addition (4 adds).

Syntax

```
UADD8{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

A.1.159 UADD16

UADD16 (Unsigned halfword-wise Add) does an unsigned halfword-wise addition (2 adds).

Syntax

```
UADD16{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

A.1.160 UASX

UASX (Unsigned Add Subtract Exchange) exchanges halfwords of Rm, then adds the top halfwords and subtracts the bottom halfwords.

Syntax

```
UASX{cond} {Rd,} Rn, Rm
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`Rd` is the destination register.

`Rm` and `Rn` are the registers holding the operands.

A.1.161 UBFX

`UBFX` (Unsigned Bit Field Extract) writes adjacent bits from one register into the least significant bits of a second register and zero extends to 32 bits.

Syntax

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

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

`Rd` is the destination register.

`Rn` is the register that contains the bits to be extracted.

`lsb` specifies the least significant bit of the bitfield.

`width` is the width of the bitfield.

A.1.162 UDIV

`UDIV` (Unsigned Divide). divides a 32-bit unsigned integer register value by a 32-bit unsigned integer register value, and writes the result to the destination register. This instruction is not present in all variants of the ARMv7-A architecture.

Syntax

```
UDIV{cond}{q} {Rd,} Rn, Rm
```

where:

`cond` is the optional condition code. See *Conditional execution* on page 5-3.

`q` specifies optional assembler qualifiers on the instruction. The following qualifiers are defined:

- `.N` (narrow), specifies that the assembler must select a 16-bit encoding for the instruction.
- `.W` (wide), specifies that the assembler must select a 32-bit encoding for the instruction.

If neither `.W` nor `.N` are specified, the assembler can select either 16-bit or 32-bit encodings. If both are available, it must select a 16-bit encoding.

`Rd` the destination register.

`Rn` is the register that contains the dividend.

`Rm` is the register that contains the divisor.

A.1.163 UHADD8

UHADD8 (Unigned Halving bytewise Add) does an unsigned bytewise addition (4 adds) and halves the results.

Syntax

```
UHADD8{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

A.1.164 UHADD16

UHADD16 (Unsigned Halving halfword-wise Add) does an unsigned halfword-wise addition (2 adds) and halves the results.

Syntax

```
UHADD16{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

A.1.165 UHASX

UHASX (Unsigned Halving Add Subtract Exchange) exchanges halfwords of Rm, then adds the top halfwords and subtracts the bottom halfwords and halves the results.

Syntax

```
UHASX{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

A.1.166 UHSAX

UHSAX (Unsigned Halving Subtract Add Exchange) exchanges halfwords of Rm, then subtracts the top halfwords and adds the bottom halfwords and halves the results.

Syntax

```
UHSAX{cond} {Rd}, Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

A.1.167 UHSUB8

UHSUB8 (Unsigned Halving bytewise Subtraction) does an unsigned bytewise subtraction (4 subtracts) and halves the results.

Syntax

```
UHSUB8{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-32.

Rd is the destination register.

Rm and Rn are the register holding the operands.

A.1.168 UHSUB16

UHSUB16 (Unsigned Halving halfword-wise Subtract) does an unsigned halfword-wise subtraction (2 subtracts) and halves the result.

Syntax

```
UHSUB16{cond} {Rd}, Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

A.1.169 UMAAL

UMAAL (Unsigned Multiply Accumulate Long; 64 <= 32 + 32 + 32 x 32) multiplies Rn and Rm (treated as unsigned integers) adds the two 32-bit values in RdHi and RdLo, and stores the 64-bit result to RdLo, RdHi.

Syntax

```
UMAAL{cond} RdLo, RdHi, Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

RdLo and RdHi are the destination accumulator registers.

Rn is the register holding the first multiplicand.

Rm is the register holding the second multiplicand.

A.1.170 UMLAL

UMLAL (Unsigned Multiply Accumulate 64 <= 64 + 32 x 32) multiplies Rn and Rm (treated as unsigned integers) and adds the 64-bit result to the 64-bit unsigned integer contained in RdHi and RdLo.

Syntax

```
UMLAL{S}{cond} RdLo, RdHi, Rn, Rm
```

where:

S (if specified) means that the condition code flags will be updated depending on the result of the instruction.

cond is an optional condition code. See *Conditional execution* on page 5-3.

RdLo and RdHi are the destination accumulator registers.

Rn is the register holding the first multiplicand.

Rm is the register holding the second multiplicand.

A.1.171 UMULL

UMULL (Unsigned Multiply; 64 <= 32 x 32) multiplies Rn and Rm (treated as unsigned integers) and stores the least significant 32 bits of the result in RdLo, and the most significant 32 bits of the result in RdHi.

Syntax

```
UMULL{S}{cond} RdLo, RdHi, Rn, Rm
```

where:

S (if specified) means that the condition code flags will be updated depending on the result of the instruction.

cond is an optional condition code. See *Conditional execution* on page 5-3.

RdLo and RdHi are the destination registers.

Rn is the register holding the first multiplicand.

Rm is the register holding the second multiplicand.

A.1.172 UQADD8

UQADD8 (Saturating Unsigned bytewise Add) does an unsigned bytewise addition (4 adds) and saturates the results to the unsigned range $0 \leq x \leq 2^8-1$. The Q flag is not affected by this instruction.

Syntax

```
UQADD8{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

A.1.173 UQADD16

UQADD16 (Saturating Unsigned halfword-wise Add) does an unsigned halfword-wise addition (2 adds) and saturates the results to the unsigned range $0 \leq x \leq 2^{16}-1$. The Q flag is not affected by this instruction.

Syntax

```
UQADD16{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the register holding the operands.

A.1.174 UQASX

UQASX (Saturating Unsigned Add Subtract Exchange) exchanges halfwords of Rm, then adds the top halfwords and subtracts the bottom halfwords and saturates the results to the unsigned range $0 \leq x \leq 2^{16}-1$. The Q flag is not affected by this instruction.

Syntax

```
UQASX{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

A.1.175 UQSAX

UQSAX (Saturating Unsigned Subtract Add Exchange) exchanges the halfwords of Rm, then subtracts the top halfwords and adds the bottom halfwords and saturates the results to the signed range $0 \leq x \leq 2^{16}-1$. The Q flag is not affected by this instruction.

Syntax

```
QSAX{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

A.1.176 UQSUB8

UQSUB8 (Saturating Unsigned bytewise Subtract) does bytewise subtraction (4 subtracts), with saturation of the results to the unsigned range $0 \leq x \leq 28-1$. The Q flag is not affected by this instruction.

Syntax

```
UQSUB8{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

A.1.177 UQSUB16

UQSUB16 (Saturating Unsigned halfword Subtract) does halfword-wise subtraction (two subtracts), with saturation of the results to the unsigned range $0 \leq x \leq 216-1$. The Q flag is not affected by this instruction.

Syntax

```
UQSUB16{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

A.1.178 USAD8

USAD8 (Unsigned Sum of Absolute Differences) finds the 4 differences between the unsigned values in corresponding bytes of Rn and Rm and adds the absolute values of the 4 differences, and stores the result in Rd.

Syntax

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

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register holding the first operand.

Rm is the register holding the second operand.

A.1.179 USADA8

USADA8 (Unsigned Sum of Absolute Differences Accumulate) finds the 4 differences between the unsigned values in corresponding bytes of Rn and Rm and adds the absolute values of the 4 differences to the value in Ra, and stores the result in Rd.

See *Sum of absolute differences* on page 5-10 for more information.

Syntax

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

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register holding the first operand.

Rm is the register holding the second operand.

Ra is the register that holds the accumulate value.

A.1.180 USAT

USAT (Unsigned Saturate) performs a shift and saturates the result to the signed range $0 \leq x \leq 2^{sat}-1$. If saturation occurs, the Q flag is set.

Syntax

```
USAT{cond} Rd, #sat, Rm{, shift}
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

sat specifies the bit position to saturate to, in the range 0 to 31.

Rm is the register holding the operand.

shift is optional shift amount and can be either ASR #n where n is in the range (1 – 32 ARM state, 1-31 Thumb state) or LSL #n where n is in the range (0 – 31).

A.1.181 USAT16

USAT16 (Unigned Saturate, parallel halfwords) saturates each unsigned halfword to the signed range $0 \leq x \leq 2^{sat}-1$. If saturation occurs, the Q flag is set.

Syntax

```
USAT16{cond} Rd, #sat, Rn
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

sat specifies the bit position to saturate to, in the range 0 to 31.

Rn is the register holding the operand.

A.1.182 USAX

USAX (Unsigned Subtract Add Exchange) exchanges halfwords of Rm, then subtracts the top halfwords and adds the bottom halfwords.

Syntax

```
USAX{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

A.1.183 USUB8

USUB8 (Unsigned bytewise Subtraction) does an unsigned bytewise subtraction (4 subtracts).

Syntax

```
USUB8{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

A.1.184 USUB16

USUB16 (Unsigned halfword-wise Subtract) does an unsigned halfword-wise subtraction (2 subtracts).

Syntax

```
USUB16{cond} {Rd,} Rn, Rm
```

where:

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm and Rn are the registers holding the operands.

A.1.185 UXT

UXT (Unsigned Extend) extracts the specified byte and zero extends to a 32-bit value.

Syntax

```
UXT<extend>{cond} {Rd,} Rm {,rotation}
```

where:

extend must be one of:

- B16 – extends two 8-bit values to two 16-bit values.
- B – extends an 8-bit value to a 32-bit value.
- H – extends a 16-bit value to a 32-bit value.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rm is the register that contains the value to be extended.

rotation can be one of ROR #8, ROR #16 or ROR #24 (or can be omitted).

A.1.186 UXTA

UXTA (Unsigned Extend and Add) extracts the specified byte, adds the value from Rn and zero extends to a 32-bit value.

Syntax

```
UXTA<extend>{cond} {Rd,} Rn, Rm {,rotation}
```

where:

extend must be one of:

- B16 – extends two 8-bit values to two 16-bit values.
- B – extends an 8-bit value to a 32-bit value.
- H – extends a 16-bit value to a 32-bit value.

cond is an optional condition code. See *Conditional execution* on page 5-3.

Rd is the destination register.

Rn is the register holding the value to be added.

Rm is the register that contains the value to be extended.

rotation can be one of ROR #8, ROR #16 or ROR #24 (or can be omitted).

A.1.187 WFE

WFE (Wait for Event). If the Event Register is not set, WFE suspends execution until one of the following events occurs:

- an IRQ interrupt (even when CPSR I-bit is set)
- an FIQ interrupt (even when CPSR F-bit is set)
- an asynchronous abort (not when masked by the CPSR A-bit)
- Debug Entry request, even when debug is disabled.
- an Event signaled by another processor using the SEV instruction.

If the Event Register is set, WFE clears it and returns immediately.

Syntax

```
WFE{cond}
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

A.1.188 WFI

`WFI` (Wait For Interrupt) suspends execution until one of the following events occurs:

- An IRQ interrupt (even when CPSR I-bit is set).
- An FIQ interrupt (even when CPSR F-bit is set).
- An asynchronous abort (not when masked by the CPSR A-bit).
- Debug Entry request, even when debug is disabled.

If the Event Register is set, `WFI` clears it and returns immediately.

Syntax

```
WFI{cond}
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

A.1.189 YIELD

`YIELD` indicates to the hardware that the current thread is performing a task that can be swapped out (for example, a spinlock). Hardware could use this hint to suspend and resume threads in a multithreading system.

Syntax

```
YIELD{cond}
```

where:

`cond` is an optional condition code. See *Conditional execution* on page 5-3.

Appendix B
Tools, Operating Systems and Boards

ARM processors can be found in a very wide range of devices, running a correspondingly wide range of software. Many readers will have ready access to appropriate hardware, tools and operating systems, but it might be useful to some readers to present an overview of some of these readily available compilation tools, ARM processor based hardware and Linux operating system distributions.

B.1 Linux distributions

Linux is a UNIX-like operating system kernel, originally developed by Linus Torvalds, who continues to maintain the official kernel. It is open source, distributed under the GNU Public License, widely-used and available on a large number of different processor architectures.

The Linux kernel is bundled with libraries and applications to make up a complete operating system in what is called a “Linux distribution”. A number of free Linux distributions exist for ARM processors, including Debian, Ubuntu, Fedora and Gentoo.

B.1.1 Linux for ARM systems

Support for the ARM architecture has been included in the standard Linux kernel for many years. Development of this port is ongoing, with significant input from ARM to provide kernel support for new processors and architecture versions.

You might wonder why a book about the Cortex-A series processors contains information about Linux. There are several reasons for this. Linux source code is available to all readers and represents a useful learning resource. In addition there are many useful resources with existing code and explanations. Many readers will be familiar with Linux, as it can be run on most processor architectures. By explaining how Linux features like virtual memory, multi-tasking, shared libraries and so forth are implemented, readers will be able to apply their understanding to other operating systems commonly used on ARM processors. The scalability of Linux is another factor – it can run on the most powerful ARM processors. and its derivative, uClinux, is also commonly used on much smaller processors, including the Cortex-M3 or ARM7TDMI processors. It can run on both the ARM and Thumb Instruction Set Architectures, using little-endian or big-endian data accesses and with or without a memory management unit.

One of the benefits of a modern operating system is that you do not have to know much detail about the underlying hardware in order to develop application software for it.

B.1.2 Linux terminology

Here, we define some terms which we will use when describing how the Linux kernel interacts with the underlying ARM architecture:

Process The kernel view of an executing unprivileged application is called a *process*. The same application (for example, `bin/bash`) can be running in several simultaneous instances in the system – and each of these instances will be a separate process. The process has resources associated with it, such as a memory map and file descriptors. A process can consist of one or more *threads*.

Thread A *thread* is a context of software execution within a process. It is the entity which is scheduled by the kernel, and actually executes the instructions that make up the application. A process can consist of multiple threads, each executing with their own program counter, stack pointer and register set – all existing within the same memory map and operating on the file descriptors held by the process as a whole. In a multi-processor system, threads inside the same process can execute concurrently on separate processors. Different threads within the same process can be configured to have different scheduling priorities.

There are also threads executing inside the kernel, to manage various tasks asynchronously, such as file cache management, or watchdog tickling, which is not as exciting as it sounds.

Scheduler This is a vital part of the kernel which has a list of all the current threads. It knows which threads are ready to be run and which are currently not able to run. It dynamically calculates priority levels for each thread and schedules the highest

priority thread to be run next. It is called after an interrupt has been handled. The scheduler is also explicitly called by the kernel using the `schedule()` function, for example, when an application executing a *system call* has to sleep. The system will have a timer based interrupt which results in the scheduler being called at regular intervals. This enables the OS to implement time-division multiplexing, where many threads share the processor, each running for a certain amount of time, giving the illusion that many applications are running simultaneously.

System calls Linux applications in ARMv7-A run in User (unprivileged) mode or at PL0 level for systems that implement the Security Extensions. Many parts of the system are not directly accessible in User mode. For example, the kernel might prevent User mode programs from accessing peripherals, kernel memory space and the memory space of other User mode programs. Access to some features of the system control coprocessor (CP15) is not permitted in User mode. The kernel provides an interface (using the `SVC` instruction) which permits an application to call kernel services, this is what forms a system call. Execution is transferred to the kernel through the `SVC` exception handler, which returns to the user application when the system call is complete.

Libraries Linux applications are, with very few exceptions, not loaded as complete pre-built binaries. Instead, the application relies on external support code linked from files called shared libraries. This has the advantage of saving memory space, in that the library only has to be loaded into RAM once and is more likely to be in the cache as it can be used by other applications. Also, updates to the library do not require every application to be rebuilt. However, this dynamic loading means that the library code must not rely on being in a particular location in memory.

Files These are essentially blocks of data which are referred to using a pathname attached to them. Device nodes have pathnames like files, but instead of being linked to blocks of data, they are linked to device drivers which handle real I/O devices like an LCD display, disk drive or mouse. When an application opens, reads from or writes to a device, control is passed to specific routines in the kernel that handle that device.

B.1.3 Embedded Linux

Linux-based systems cover the range from servers, using the desktop, through mobile devices, right down to high-performance micro-controllers in the form of uClinux for processors lacking an MMU. However, while the kernel source code base is the same, different priorities and constraints mean that there can be some fundamental differences between the Linux running on your desktop and the one running in your set-top box, as well as between the development methodologies used.

In a desktop system, a form of bootloader executes from ROM, either BIOS or UEFI. This has support for mass-storage devices and can then load a second-stage loader, for example GRUB, from a CD, a hard drive or even a USB memory stick. From this point on, everything is loaded from a general-purpose mass storage device.

In an embedded device, the initial bootloader is likely to load a kernel directly from on-board flash into RAM and execute it. In severely memory constrained systems, it might have a kernel built to *execute in place* (XiP), where all of the read-only portions of the kernel remain in ROM, and only the writable portions use RAM. Unless the system has a hard drive (or for fault tolerance reasons), the root filesystem on the device is likely to be located in flash. This can be a read-only filesystem, with portions that have to be writable overlaid by `tmpfs` mounts, or it can be a read-write filesystem. In both cases, the storage space available is likely to be significantly less than in a typical desktop computer. For this reason, they might use software components

such as uClibc and BusyBox to reduce the overall storage space required for the base system. A general desktop Linux distribution is usually supplied preinstalled with a lot of software that you might find useful at some point. In a system with limited storage space, this is not really optimal. Instead, you want to be able to select exactly the components you require to achieve what you want with your system. Various specific embedded Linux distributions exist to make this easier.

In addition, embedded systems often have lower performance than general purpose computers. In this situation, speed of development can be significantly increased by compiling software for the target device on a faster desktop computer and then moving it across to the target device. This process is called *cross-compiling*.

B.1.4 Board Support Package

Getting Linux to run on a particular platform requires a *Board Support Package* (BSP). We can divide the platform-specific code into a number of areas:

- Architecture-specific code. This is found in the `arch/arm/` directory of the Linux kernel source code and forms part of the kernel porting effort carried out by the ARM Linux maintainers.
- Processor-specific code. This is found in `the arch/arm/mm/` and `arch/arm/include/asm/` directories of the Linux kernel source code. This takes care of MMU and cache functions (for example, translation table setup, Translation Lookaside Buffer and cache invalidation and memory barriers). On SMP processors, spinlock code will be enabled.
- Generic device drivers are found under `drivers/`.
- Platform-specific code will be placed in `the arch/arm/mach-*/ directory of the Linux kernel source code`. This is code which is most likely to be altered by people porting to a new board containing a processor with existing Linux support. The code will define the physical memory map, interrupt numbers, location of devices and any initialization code specific to that board.

B.1.5 Linaro

Linaro is a non-profit organization which works on a range of open source software running on ARM processors, including kernel related tools and software and middleware. It is a collaborative effort between a number of technology companies to provide engineering help and resources to the open source community.

Linaro does not produce a Linux distribution, nor is it tied to any particular distribution or board. Instead, Linaro works on generic ARM technology to provide a common software platform for use by board support package developers. Its focus is on tools to helping developers write and debug code, on low-level software which interacts with the underlying hardware and on key pieces of middleware. Linaro's members have a reduced time to market and can deliver unique open source based products using ARM technology.

Linaro engineers work on the kernel and tools, graphics and multimedia and power management. Linaro provides patches to upstream projects and makes monthly source tree tarballs available, with an occasional integrated build every six months to consolidate the work.

In this way, code can easily be transferred to the mainline Linux kernel and other open source projects. Evaluation builds of Android and Ubuntu, plus generic Linux, toolchain and other downloads for ARMv7 processors are available from `http://www.linaro.org/downloads/`.

See `http://www.linaro.org/` for more information about Linaro.

B.2 Useful tools

Thissection takes a brief look at some available tools which can be useful to developers of ARM architecture based Linux systems. These are all extensively documented elsewhere. Here, we merely point out that these tools can be useful, and provide short descriptions of their purpose and function.

B.2.1 QEMU

QEMU is a fast, open source machine emulator. It was originally developed by Fabrice Bellard and is available for a number of architectures, including ARM. It can run operating systems and applications made for one machine (for example, an ARM processor) on a different machine, such as a PC or Mac. It uses dynamic translation of instructions and can achieve useful levels of performance, enabling it to boot complex operating systems like Linux.

B.2.2 BusyBox

BusyBox is a piece of open source software which provides many standard Unix tools, in a very small executable, which is ideal for many embedded systems and could be considered to be a *de facto* standard. It includes most of the Unix tools which can be found in the GNU Core Utilities, and many other useful tools including `init`, `dhclient`, `wget` and `tftp`. Less commonly used command switches are removed.

BusyBox calls itself the “Swiss Army Knife of Embedded Linux” – a reference to the large number of tools packed into a small package. BusyBox is a single binary executable which combines many applications. This reduces the overheads introduced by the executable file format and enables code to be shared between multiple applications without having to be part of a library.

B.2.3 Scratchbox

The general principle of cross-compiling is to use one system (the host) to compile software which runs on some other system (the target).

The target is a different architecture to the host and so the host cannot natively run the resulting image. For example, you might have a powerful desktop x86 machine and want to develop code for a small battery-powered ARM processor based device which has no keyboard. Using the desktop machine will make code development simpler and compilation faster. There are some difficulties with this process. Some build environments will try to run programs on the target machine during compilation, and of course this is not possible. In addition, tools which during the build process try to discover information about the machine (for software portability reasons), do not work correctly when cross-compiling.

Scratchbox is a cross-compilation toolkit which solves these problems and gives the necessary tools to cross-compile a complete Linux distribution. It can use either QEMU or a target board to execute the cross-compiled binaries it produces.

B.2.4 U-Boot

Das U-Boot (Universal Bootloader) is a universal bootloader that can easily be ported to new hardware processors or boards. It provides serial console output which makes it easy to debug and is designed to be small and reliable. In an x86 system, we have BIOS code which initializes the processor and system and then loads an intermediate loader such as GRUB or syslinux, which then in turn loads and starts the kernel. U-Boot essentially covers both functions.

B.2.5 UEFI and Tianocore

The *Unified Extensible Firmware Interface* (UEFI) is the specification of a programmable software interface that sits on top a computer's hardware and firmware. Rather than all of the boot code being stored in the motherboard's BIOS, UEFI sits in non-volatile memory. A computer boots into UEFI, a set of actions are carried out, before loading an operating system, such as Windows or Linux. While BIOS is limited to 16-bit processes and 1MB of memory addressing, UEFI can function in 32-bit and 64-bit modes, enabling much more RAM to be addressed by more complex processes. It also can be architecture independent and provide drivers for components that are also independent of what kind of processor you have. The UEFI forum is a non-profit collaborative trade organization formed to promote and manage the UEFI standard.

UEFI is processor architecture independent and the Tianocore EFI Development Kit 2 (EDK2) is available under a BSD license. It contains UEFI support for ARM platforms, including ARM Versatile Express boards.

See `http://www.uefi.org` and `http://sourceforge.net/apps/mediawiki/tianocore` for more information.

B.3 Software toolchains for ARM processors

There are a wide variety of compilation and debug tools available for ARM processors. In this section, we will focus on two toolchains (a collection of programming tools), the GNU toolchain which includes the GNU Compiler (`gcc`), and the ARM Compiler toolchain which includes the `armcc` compiler.

Figure B-1 shows how the various components of a software toolchain interact to produce an executable image.

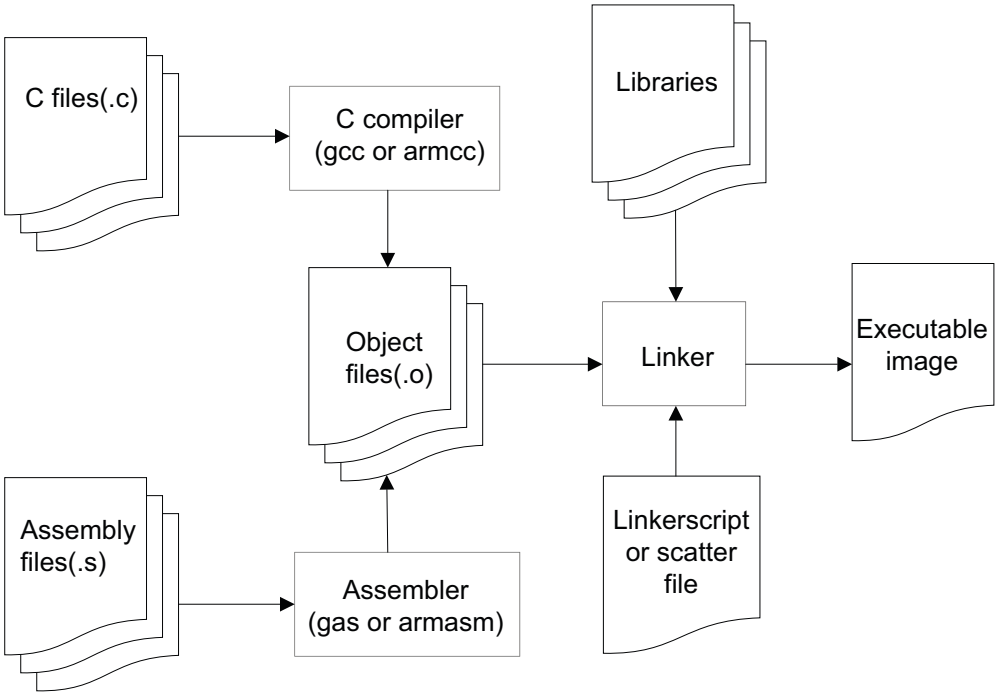


Figure B-1 Using a software toolchain to produce an image

B.3.1 GNU toolchain

The GNU toolchain is used both to develop the Linux kernel and to develop applications (and indeed other operating systems). Like Linux, the GNU tools are available on a large number of processor architectures and are actively developed to make use of the latest features incorporated in ARM processors.

The toolchain includes the following components:

- GNU make.
- GNU Compiler Collection (GCC).
- GNU binutils linker, assembler and other object/library manipulation tools.
- GNU Debugger (GDB).
- GNU build system (autotools).
- GNU C library (glibc or eglibc).

glibc is available on all GNU Linux host systems and provides portability, wide compliance with standards, and is performance optimized. However, it is quite large for some embedded systems (approaching 2MB in size) so other libraries may be preferred in smaller systems. For example, uClibc provides most features and is around 400KB in size, and produces significantly smaller application binaries. Android does not use glibc, but instead has its own BSD-derived system C library called Bionic.

Prebuilt versions of GNU toolchains

If you are using a complete Linux distribution on your target platform, and you are not cross-compiling, you can install the toolchain packages using the standard package manager. For example, on a Debian-based distribution such as Ubuntu you can use the command:

```
sudo apt-get install gcc g++ gcc-doc
```

Additional required packages such as `binutils` will also be pulled in by this command, or you can add them explicitly on the command line. In fact, if `g++` is specified this way, `gcc` is automatically pulled in. This toolchain will then be accessible in the way you would expect in any Linux system, by calling `gcc`, `g++`, `as`, or similar.

If you are cross-compiling, you must install a suitable cross-compilation toolchain. The cross-compilation toolchain consists of the GNU Compiler Collection (GCC) but also the GNU C library (`glibc`) which is necessary for building applications (but not the kernel).

Ubuntu distributions from Maverick (10.10) onwards include specific packages for this. These can be run using the command:

```
sudo apt-get install gcc-arm-linux-gnueabi
```

The resulting toolchain will be able to build Linux kernels, applications and libraries for the same Ubuntu version that is used on the build platform. It will however, have a prefix added to all of the individual tool commands in order to avoid problems distinguishing it from the native tools for the workstation. For example, the cross-compiling `gcc` will be accessible as `arm-linux-gnueabi-gcc`.

If your workstation uses an older Ubuntu distribution or an alternative Linux distribution, another toolchain must be used.

Linaro provide up-to-date source packages for ARM toolchains from `http://www.linaro.org/downloads/`. These can be used for generating both cross and native toolchains.

B.3.2 ARM Compiler toolchain

The ARM Compiler toolchain can be used to build programs from C, C++, or ARM assembly language source. It generates optimized code for the 32-bit ARM and mixed length (16-bit and 32-bit) Thumb instruction sets, and supports full ISO standard C and C++. It also supports the NEON SIMD instruction set with the vectorizing (multiple operations simultaneously) NEON compiler.

The ARM Compiler toolchain comprises the following components:

armcc The ARM and Thumb compiler. This compiles your C and C++ code. It supports inline and embedded assembly code, and also includes the NEON vectorizing compiler, invoked using the command:

```
armcc --vectorize
```

armasm The ARM and Thumb assembler. This assembles ARM and Thumb assembly language sources.

armlink The linker. This combines the contents of one or more object files with selected parts of one or more object libraries to produce an executable program.

armar The librarian. This enables sets of ELF format object files to be collected together and maintained in libraries. You can pass such a library to the linker in place of several ELF files. You can also use the library for distribution to a third party for further application development.

fromelf The image conversion utility. This can also generate textual information about the input image, such as disassembly and its code and data size.

C libraries The ARM C libraries provide:

- an implementation of the library features as defined in the C and C++ standards
- extensions specific to the ARM Compiler
- GNU extensions
- common nonstandard extensions to many C libraries
- POSIX extended functionality
- functions standardized by POSIX (See *Threading libraries* on page 19-6).

C++ libraries

The ARM C++ libraries provide:

- helper functions when compiling C++
- additional C++ functions not supported by the Rogue Wave library.

Rogue Wave C++ libraries

The Rogue Wave library provides an implementation of the standard C++ library.

B.4 ARM DS-5

ARM DS-5 is a professional software development solution for Linux, Android and bare-metal embedded systems running on 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 *Integrated Development Environment* (IDE).

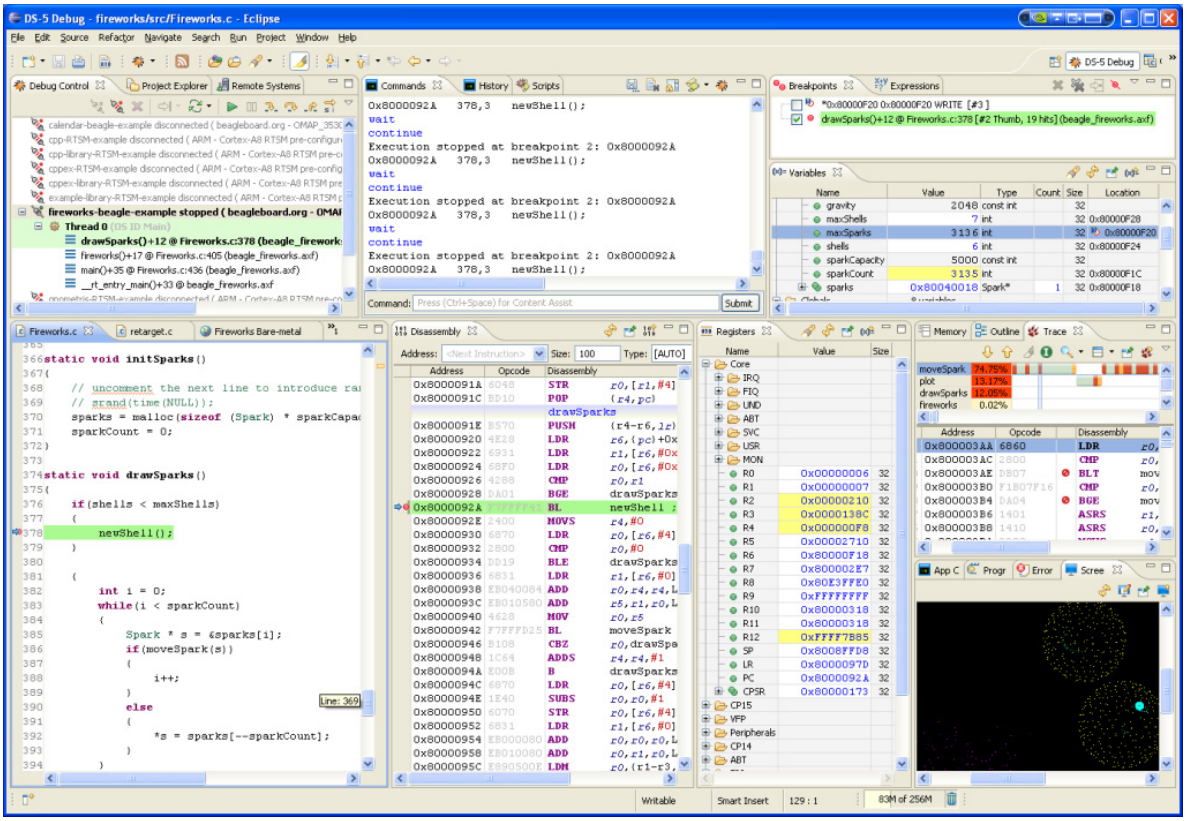


Figure B-2 DS-5 Debugger

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. Both GCC and the ARM Compiler are provided. See *ARM Compiler toolchain* on page B-8 for more information about the ARM Compiler.
- *Fixed Virtual Platforms* (FVPs) of complete ARM Cortex-A8 and Cortex-A9 MPCore processor-based devices. Typical simulation speeds are above 250 MHz.

- DS-5 Debugger, shown in Figure B-2 on page B-10, together with a supported debug target, enables debugging of bare-metal applications, Linux or Android kernels, Linux or Android applications and Linux or Android kernel modules. 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 and a number of Example projects, including bare-metal startup code examples for the range of ARM processors, and Linux applications example projects that can run models or (using JTAG-based debug hardware) on real hardware.
- DS-5 Streamline, a system-wide software profiling and performance analysis tool for ARM processor based Linux and Android platforms. DS-5 Streamline supports SMP configurations, native Android applications and libraries.

  Streamline only requires a standard TCP/IP network connection to the target in order to acquire and analyze system-wide performance data from Linux and Android systems, therefore making it an affordable solution to make software optimization possible from the early stages of the development cycle.

  See *DS-5 Streamline* on page 16-4 for more information.

B.5 Example platforms

In this section we'll mention a few widely available, off-the-shelf ARM processor based platforms which are suitable for use by students or hobbyists for Linux development. This list is likely to become outdated quickly, as newer and better boards are frequently announced. Of course, for mobile application development, your nearest smartphone is a good development platform.

B.5.1 BeagleBone Black

BeagleBone Black is a community-supported development platform for developers and hobbyists that uses a IGHz Cortex-A8 processor with 512MB DDR3 RAM and 4 USB ports and can be used to run Linux and Android

B.5.2 Gumstix

This derives its name from the fact that the board is the same size as a stick of chewing gum. The Gumstix Overo uses the OMAP3503 device from TI, containing a Cortex-A8 processor clocked at 600MHz and runs Linux 2.6 with the BusyBox utilities and OpenEmbedded build environment.

B.5.3 PandaBoard

PandaBoard is a single-board computer based on the Texas Instruments OMAP4430 device, including a dual-core 1GHz ARM Cortex-A9 processor, a 3D Accelerator video processor and 1GB of DDR2 RAM. Its features include Ethernet and Bluetooth plus DVI and HDMI interfaces.

B.5.4 Arndale Octa Board

The Arndale Octa board uses a 1.8GHz quad- core Cortex-A15 and 1.2GHz quad-core Cortex-A7 processors in a big.LITTLE configuration which enables energy efficient computing for less intensive tasks. It supports 1080p video encoding and decoding, 3D graphics display and high resolution image signal processing using an ARM Mali™ T628 GPU.

B.5.5 Altera Cyclone V SoC

The Altera Cyclone integrates an ARM-based hard processor system (HPS) with a 925MHz dual-core Cortex-A9 processor, peripherals and memory interfaces and 1 GB DDR3 SDRAM with the FPGA fabric using a high-bandwidth interconnect, also with 1 GB DDR3 SDRAM. It can be used for high-volume applications including protocol bridging, motor control drives, broadcast video converter and capture cards, and handheld devices.

B.5.6 Xilinx Zynq-7000 All Programmable SoC

The Xilinx Zynq-7000 contains a 1GHz, dual-core, hardened implementation of the ARM Cortex-A9 processor with 1GB of DDR3 component memory. The clusters communicate with on-chip memory, SDRAM and Flash memory controllers, and peripheral blocks through an ARM AMBA AXI-based interconnect. The board is ideal for motor control applications , video surveillance and machine vision, vending machines, as well as manufacturing, assembly and automation.

B.5.7 Freescale Vybrid

Vybrid F Series controller solutions range from a single Cortex-A5 core to a dual-core Cortex-A5 processor with a Cortex-M4 Memeory Control Unit and are designed for industrial applications that require critical safety and security, connectivity, rich HMI and real-time control.

Appendix C
Building Linux for ARM Systems

A working Linux system has two primary components, namely the kernel and the root filesystem. The kernel is the core of the operating system and acts as a resource manager. It is loaded into RAM by the bootloader and then executed during the boot process, as described in Chapter 13. The root filesystem contains system libraries, applications, tools and utilities, for example, command line interfaces or shells, graphical interfaces (such as the X window system), text editors (such as vi, emacs, gedit) and advanced applications like web browsers or office suites.

The root filesystem is often located in persistent storage, such as a hard disk or a flash device. However, the root filesystem can also reside in primary memory (RAM). Althoughwe do not cover RAM-based filesystems in great detail, we will cover the steps for building the kernel and then the root filesystem, before analyzing how these fit together to get the system running.

C.1 Building the Linux kernel

This section explains the steps required to build the kernel for an ARM processor based platform, either natively on the target platform or cross-compiled on, for example, an x86 PC. The platform we build the kernel for is called the *target* platform. The platform on which we build is called the *build* platform. We assume that the reader has some experience in using Linux, C compilers and has a machine (either x86 or ARM processor based) running a Linux distribution like Ubuntu – for other distributions some of the steps of the build procedure might differ slightly. We also assume that the build machine has a working internet connection. If this machine is ARM processor based, it is not necessary for the build platform to be exactly the same as the target platform.

The Linux kernel can be viewed as consisting two parts. One part is architecture independent and consists of components like process schedulers, system call interfaces architecture-independent device drivers and high-level network subsystems. The other part is closely related to the hardware platform for which the kernel is being built. This consists of board initialization code and drivers corresponding to a specific hardware platform. While building a kernel one has be sure of having the correct set of initialization code and drivers for the platform at hand.

The Linux kernel sources can be found at `http://www.kernel.org/`.

There are two ways to get the kernel source code. The first is to download a compressed tar file. The exact name of this file will naturally depend on the version of the kernel selected, for example 2.6.34. When downloaded, this file must be uncompressed using a command similar to the following.

```
tar xjvf  linux-2.6.34.tar.bz2
```

The other option for obtaining the kernel source is to obtain the source tree using GIT (git) commands. Linux is developed using the GIT version control system, which can be installed using a command similar to the one below for Ubuntu.

```
sudo apt-get install git-core
```

Obviously, a working internet connection is required for the above steps and those which follow.

The Linux source tree can be cloned with a command similar to the following:

```
git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux-2.6.git
```

This command will copy the official Linux kernel repository to the current directory, which can take some time. You can then check out a specific tag, which is a local operation and fairly quick.

```
git checkout -b  v2.6.34
```

Follow the instructions in *Prebuilt versions of GNU toolchains* on page B-8 in order to obtain and install a suitable toolchain. Also ensure that the toolchain is accessible on your path.

When the source tree is in place, the kernel has to be configured to match the hardware platform and desired kernel features. The standard method is to use a command such as:

```
make ARCH=arm realview_defconfig
```

which generates a default configuration file for the RealView platform file and stores it as `.config`.

There are several methods available for configuring the kernel. The most commonly used provides a text based interface based on the `ncurses` library. It can be invoked using the command:

```
make ARCH=arm menuconfig
```

This gives a configuration screen for selecting or omitting features of the kernel, as shown in Figure C-1.

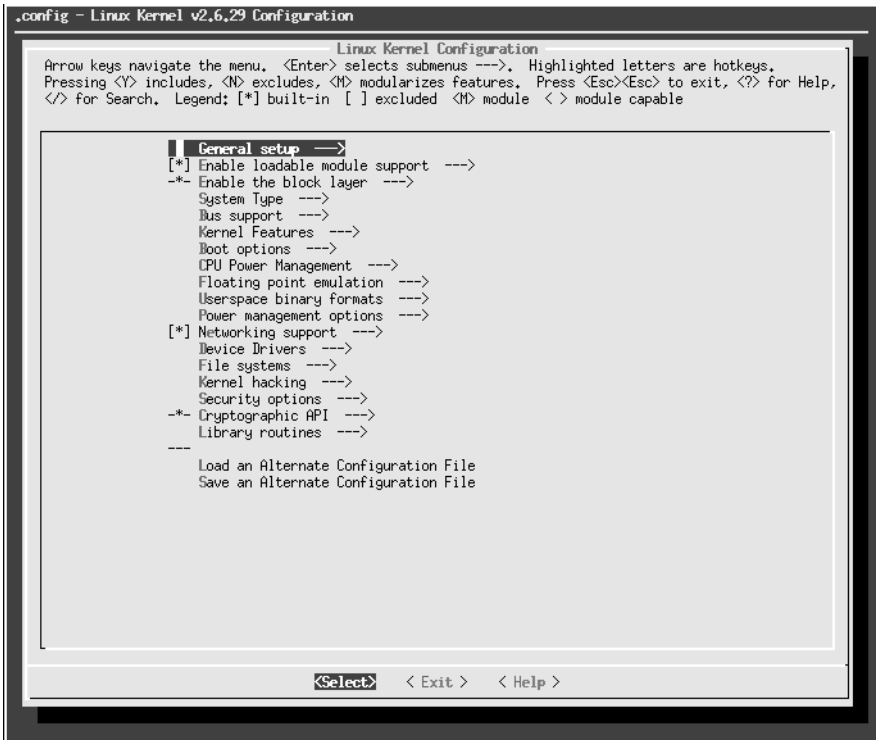


Figure C-1 Kernel Configuration screen using make menuconfig

If this command fails, and the configuration menu does not appear, this could be because the `ncurses` header files are not installed on your build host. You can install them by executing:

```
sudo apt-get install libncurses5-dev
```

The other alternative is to use a graphical Xconfig tool; this uses the Qt GUI library and can be invoked using the command:

```
make ARCH=arm xconfig
```

Figure C-2 on page C-4 shows the Kernel Configuration screen from this tool.

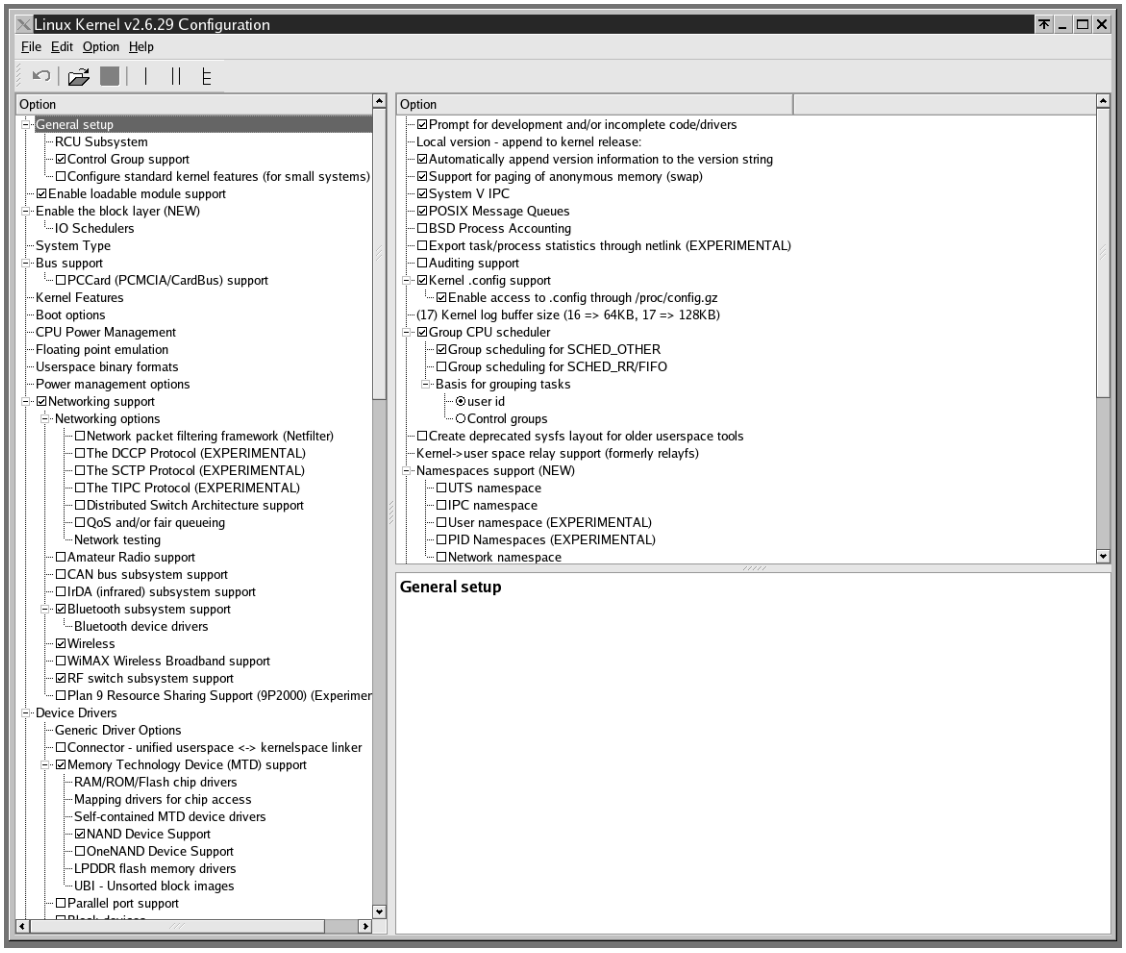


Figure C-2 Kernel Configuration screen using make xconfig

When the kernel is configured correctly then it can be built using a simple make command as below – exact details can differ slightly depending on the bootloader used:

```
make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-
```

The `CROSS_COMPILE` value must be set to the toolchain cross-compilation prefix (*Prebuilt versions of GNU toolchains* on page B-8) or must be completely left out for native compilation.

The output of the compilation would be in the form of a compressed kernel zImage. This can usually be found in the path `<source root>/arch/arm/boot` as a file named zImage.

When compiling natively on an ARM processor based system, the `CROSS_COMPILE`... parameter can be left out. If a different cross compilation toolchain than the codesourcery Linux EABI toolchain is used, `arm-none-linux-eabi-` might have to be modified to reflect the name of the toolchain executables. The above also assumes that the cross compilation toolchain executable directory is listed in your `PATH` environment variable.

This operation requires the `mkimage` utility to be installed. The Ubuntu package name for this tool is `uboot-mkimage`.

If the hardware platform uses the U-Boot bootloader, the kernel image must be converted into a form accepted by U-boot. For example, in Ubuntu the `uboot-mkimage` package can be installed followed by the command below, instead of the simple make, to create an “ubootified” image.

```
make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- uImage
```

The built kernel image would be found as the file named `uImage`, again in the path `<source root>/arch/arm/boot`.

Now that the kernel is built, the filesystem must be created in order to have a working ARM Linux system. The next section deals with creating the filesystem.

C.2 Creating the Linux filesystem

Creating a filesystem for an ARM processor based platform is relatively straightforward, using a Ubuntu distribution. The procedure described here does not actually compile the filesystem, rather it downloads pre-built packages to create the filesystem. A full compilation of the filesystem would take many hours of compilation time and can be technically challenging.

For carrying out the process of putting together the filesystem the rootstock package must be installed, which can be done using the command below:

```
sudo apt-get install rootstock
```

When the package is installed, navigate to the directory to be used for creating and saving the filesystem. The filesystem can be created in the form of a compressed tarball using the command below.

```
sudo rootstock --fqdn ubuntu --login ubuntu --password ubuntu --imagesize 3G --seed
ubuntu-desktop
```

In this case, the system would be based on the `ubuntu-desktop` seed which includes a desktop windowing system. Other *seeds* could be used instead, for example `xubuntu-desktop` (for a smaller, more lightweight desktop system) or `build-essential` (for a text based interface).

Note

This step might take a significant amount of time, depending on the speed of the internet connection.

The following steps describe the process of preparing a disk drive and transferring the filesystem to it. This disk would then be used as the root filesystem on the ARM processor based platform. The term disk drive is used here in a loose sense and refers to a variety of secondary storage devices, for example, hard disks, compact-flash drives, or USB drives. After plugging in the disk drive to the computer which was used to create the filesystem, the following command can be used to check the connected drives

```
sudo fdisk  -l
```

Note

The disk drive being used for storing the root filesystem of the ARM processor based platform will be formatted by the following steps. The process of formatting the disk will destroy any data that might exist on the disk. You must ensure that the disk does not contain any useful data prior to formatting, as this data will be permanently erased. It is also easy to lose the data elsewhere on the system if the wrong device is specified here.

The disk drive to be used must be identified correctly in the list given by the command above (for example, `/dev/sdb`). Assuming that `/dev/sdb` is the correct device or disk drive, the following command is required to partition the drive correctly.

```
sudo fdisk /dev/sdb
```

On entering the above command, the `fdisk` prompt will be displayed. Now a sequence of `fdisk` command must be entered which are in the form of single characters.

Type `m` to display the help for the possible set of `fdisk` commands. Use `d` to delete any existing partitions on the disk. You can then create a new partition using `n`. The character `w` is used to write the changes to the disk and exit. You can check that the partition has been written correctly by starting `fdisk` again using the above command, followed by a `p` for printing the partition table.

When the partition table has been written correctly, the disk partition that has been created can be formatted. Usually the first partition created on the disk identified by (for example) `/dev/sdb` is denoted as `/dev/sdb1`. The command for formatting this partition as an ext-3 filesystem is as below.

```
sudo mkfs.ext3 /dev/sdb1
```

Now, the previously created tarball for the Linux filesystem must be decompressed into this disk partition. To do this, the disk partition must be *mounted* which either can be done manually with the `mount` command, or automatically by unplugging and re-plugging in the drive. In the automatic mounting case, the disk is usually mounted at the location `/media/<disk_directory>`. Navigate to the directory at which the disk is mounted and uncompress the file-stem tarball into it using the command below:

```
sudo tar zxvf  <path_to_tarred_file_system/file.tgz>
```

Now that both the kernel and filesystem have been created, the two must be brought together to have a working Linux system.

Note

It is also possible to store the root filesystem in primary memory, using a RAMDISK. However, this alternative will not be described here and is left to advanced users.

C.3 Putting it together

The steps for creating the kernel and the filesystem are generic and common to many different boards or platforms. However, the steps for having the kernel programmed onto the platform can vary between boards. For most boards, the kernel must be transferred to some form of secondary memory that either exists on the board or is connected to it. For example, on ARM Versatile boards, the kernel must be copied on to a flash device on the board. There can be two different kinds of procedures for doing this depending on the board.

Use the documentation for your development board to find the appropriate method for getting your kernel installed, and follow the instructions. You must also ensure that the root filesystem which was created earlier is connected correctly or copied onto the target board.

In order to obtain a list of possible U-Boot commands type `help` at the U-Boot prompt.

The U-Boot bootloader is popular on many ARM processor based platforms. It is free and open-source. Other than U-Boot, there are also other bootloaders which might be proprietary for certain ARM processor based platforms. In order to interact with the bootloader, it might be necessary to connect a serial interface between the board or platform and a personal computer running a serial terminal. Again, refer to the board's user manual for more details.

The bootloader can pass a set of parameters to the kernel during the boot process. Among these is the kernel command line, known in U-Boot as *bootargs* (short for boot arguments).These parameters are used by the kernel for some initialization configurations. An example command for setting up the bootargs on ARM Versatile boards is shown below. This must be entered at the U-Boot prompt.

```
  setenv bootargs root=/dev/sda1 mem=512M@0x20000000 ip=dhcp console=ttyAMA0 clcd=xvga
rootwait
```

The bootargs in this case specifies the following:

- The root filesystem is in `/dev/sda1`.
- The memory region to be used by the operating system in the form of a size (512MB) and a starting address location. This isn't required if the bootloader passes the correct ATAG_MEM atag.
- The IP address (for example, 192.168.0.7) or the mechanism for obtaining it (in this case DHCP).
- The display interface details.
- The delay required before trying to mount the root filesystem which can be required for devices to be recognized (for example, USB disks).

The command above can be followed by a `saveenv` command in U-Boot, to save the changes into flash and make them permanent. In order to obtain a list of possible U-Boot commands type help at the U-Boot prompt. Type `help <command>` at the U-Boot prompt, to get more details regarding a particular command.

The bootloader also requires a boot command to start the boot process automatically. In the simplest case the bootcmd can be set as follows, where `0x41000000` is the location where the kernel is stored for example in flash memory.

```
  setenv bootcmd bootm 0x41000000
```

If the kernel image exists at a different location, or on the network, a `cp` or `tftp` command can precede the `bootm` command. U-Boot commands on the same line must be separated by a ";", for example:

```
setenv bootcmd cp 0x65000000 0x41000000 0x8000000; bootm 0x41000000
```

Again, a `saveenv` command would be required, to ensure that changes are saved to secondary storage. Now that the bootcmd is in place, the system must be restarted so that the bootcmd is used automatically and the system starts running. The sequence of boot commands can also be entered manually at the U-Boot prompt, to try different options without saving them permanently.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Z