Migrating a software application from ARMv5 to ARMv7-A/R

Version: 1.0

Application Note 425

ARM®

ARM DAI0425 (ID080414)

Migrating a software application from ARMv5 to ARMv7-A/R
Application Note 425

Release Information

The following changes have been made to this application note.

Change history

Date	Issue	Confidentiality	Change
30 July 2014	A	Non-Confidential	First release

Proprietary Notice

Confidentiality Status

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

Product Status

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

Web Address

`http://www.arm.com`

Contents
Migrating a software application from ARMv5 to ARMv7-A/R Application Note 425

Chapter 1
Introduction

Read this chapter for information about the purpose of this application note and the benefit you can get from migrating to ARMv7.

It contains the following sections:

1.1 What this document contains

This document is intended to help you migrate software applications from ARMv5 to ARMv7.

It describes the differences between ARMv5 and ARMv7, and explains the issues involved in migrating an existing software application from ARMv5 to ARMv7.

Familiarity with ARMv5 is assumed. This application note also assumes that you have software development experience with ARMv5.

The main target platform is assumed to be built around an ARMv7-A processor. As ARMv7-A and ARMv7-R have many overlapping areas, part of this document also applies to the ARMv7-R processor. By default, ARMv7 refers to ARMv7-A and ARMv7-R in this application note.

This document contains the following chapters:

- Chapter 1 *Introduction* provides information about the migration benefits and the overall structure of this document.
- Chapter 2 *ARMv7 architecture overview* provides an overview of the ARMv7 architecture.
- Chapter 3 *ARMv5 and ARMv7 compared* describes the differences between ARMv5 and ARMv7.
- Chapter 4 *Migrating a software application from ARMv5 to ARMv7-A/R* provides information about how to migrate a software application from ARMv5 to ARMv7.
- Chapter 5 *Migration notes* provides some useful points when porting source code from ARMv5 to ARMv7.

1.2 Benefits of this migration

Compared with ARMv5, the ARMv7 architecture provides a number of new features and benefits:

- NEON™ technology provides flexible and powerful acceleration for consumer multimedia applications, and delivers a significantly enhanced user experience.
- TrustZone® technology provides security for a wide array of client and server computing platforms, including handsets, tablets, wearable devices, and enterprise systems.
- Thumb®-2 technology provides increased performance while maintaining the high code density of the original Thumb instruction set.
- A new set of features enhances multiprocessing functionality.
- *Strongly-ordered*, *Normal*, and *Device* memory types provide fine-grained control over memory accessing order.
- Exclusive access instructions replace the legacy `SWP` instruction to support nonblocking shared memory synchronization primitives.

1.3 References and further reading

This application note refers to the following books. You can download them directly from the ARM infocenter at `http://infocenter.arm.com`.

- *ARM® Architecture Reference Manual ARMv7-A and ARMv7-R Edition, ARM DDI 0406C*.
- *ARMv5 Architecture Reference Manual, ARM DUI 0100D*.
- *Introducing NEON™, ARM DHT0002*.
- *Cortex®-A Series Programmer's Guide, ARM DEN0013*.
- *NEON™ Programmer's Guide, ARM DEN0018*.
- *ARM® Security Technology Building a Secure System using TrustZone® Technology, PRD29-GENC-009492*.

1.4 Feedback on content

ARM welcomes feedback on this documentation.

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

- The title.
- The number, ARM DAI 0425.
- The page numbers to which your comments apply.
- A concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

Note

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

Chapter 2
ARMv7 architecture overview

Read this chapter for a brief introduction to the ARMv7 architecture.

It contains the following section:

2.1 ARMv7 architecture overview

ARMv7 is implemented in the Cortex range of processors, and is defined in the following three profiles:

- ARMv7-A is implemented in the Cortex®-A5, Cortex-A7, Cortex-A8, Cortex-A9, Cortex-A12, Cortex-A15, and Cortex-A17 processors.

  ARMv7-A supports fully featured application class devices running platform Operating Systems, such as Linux and Windows Mobile. It provides full virtual memory support, *Large Physical Address Extension* (LPAE), security and virtualization extensions, and optional media processing.

- ARMv7-R is implemented in the Cortex-R4, Cortex-R5, and Cortex-R7 processors.

  ARMv7-R is targeted at applications that require hard, predictable real-time performance. Devices incorporating a Cortex-R4 processor are used, for example, in engine management systems, hard disk drive controllers, and mobile baseband processors.

- ARMv7-M is used in microcontroller-type devices, principally those based around the Cortex-M3 and Cortex-M4 processors. For example, the SecurCore™ SC300™ processor is based on the ARM Cortex-M3 processor.

  ARMv7-M supports a subset of features in the ARMv7-A and ARMv7-R profiles, and enables devices that maximize power efficiency and minimize cost. An ARMv7-M processor incorporates many features common in the microcontroller world, such as hardware interrupt preemption.

The following optional extensions to the ARMv7-A architecture are available:

Security

The TrustZone® security extension was introduced in the v6K architecture and is an optional extension to the ARMv7-A profile. TrustZone introduces an additional operating mode, *Monitor* (Mon) mode, with associated banked registers and an additional “Secure” operating state.

Advanced SIMD and Floating Point

Both floating point (VFP) support and Advanced SIMD (NEON) are optional extensions to the ARMv7-A profile. They can be implemented together, in which case they share a common register bank and some common instructions. Almost all NEON implementations also include floating point support.

40-bit physical addressing

LPAE is an optional extension to the ARMv7-A profile. This extension to the VMSAv7 virtual memory architecture enables the generation of 40-bit physical addresses from 32-bit virtual addresses. LPAE is supported by the Cortex-A7, Cortex-A12, Cortex-A15, and Cortex-A17 processors.

Virtualization

The virtualization extension introduces an extra mode, Hypervisor mode, with associated banked registers. You can use a new Hyp exception to trap software accesses to hardware and configuration registers, thus enabling the implementation of an efficient hardware-assisted virtualization solution. This extension is supported by the Cortex-A7, Cortex-A12, Cortex-A15, and Cortex-A17 processors.

For more information about these extensions, see *ARM® Architecture Reference Manual ARMv7-A and ARMv7-R Edition, ARM DDI 0406C*.

Chapter 3
ARMv5 and ARMv7 compared

Read this chapter for a description of the differences between ARMv5 and ARMv7. This document does not intend to describe all the differences between ARMv5 and ARMv7. It only highlights relevant differences while migrating a software application from ARMv5 to ARMv7.

It contains the following sections:

3.1 Registers

Compared with ARMv5, ARMv7-A provides a number of extra features and modes, including some new registers and some changes to existing registers:

- *Monitor mode and Hypervisor mode registers*
- *System control coprocessor registers* on page 3-3
- *Program status register* on page 3-5
- *VFPv3 double-precision registers* on page 3-5

3.1.1 Monitor mode and Hypervisor mode registers

ARMv7-A introduces support for two more processor modes, *Monitor* (Mon) mode and *Hypervisor* (Hyp) mode, to support Security and Virtualization extensions respectively. Figure 3-1 shows the ARM register set in ARMv7-A.

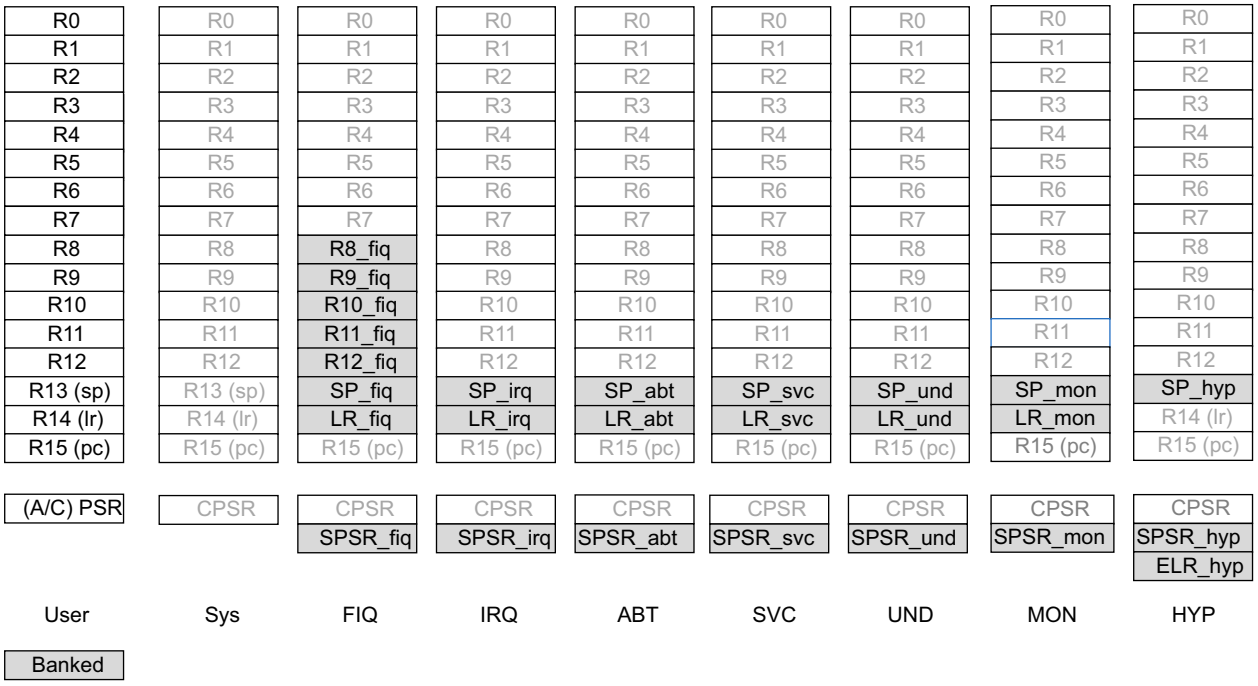

Figure 3-1 The ARM register set

As shown in Figure 3-1, the Mon mode and Hyp mode have the following banked registers, which do not exist in ARMv5:

- SP_mon.
- LR_mon.
- SPSR_mon.
- SP_hyp.
- SPSR_hyp.
- ELR_hyp.

Note

HYP mode shares the same LR with User or System mode.

3.1.2 System control coprocessor registers

In ARMv7-A, the following system control coprocessor registers are different from those in ARMv5:

- c0.
- c1.
- c13.

CP15 c0, Cache Type Register

In ARMv5, the format of the Cache Type Register is different from the ARMv7-A definition. However, the general properties described by the register, and the access rights for the register, remain unchanged.

Figure 3-2 shows the bit assignments of the Cache Type Register in ARMv5 and ARMv7-A.

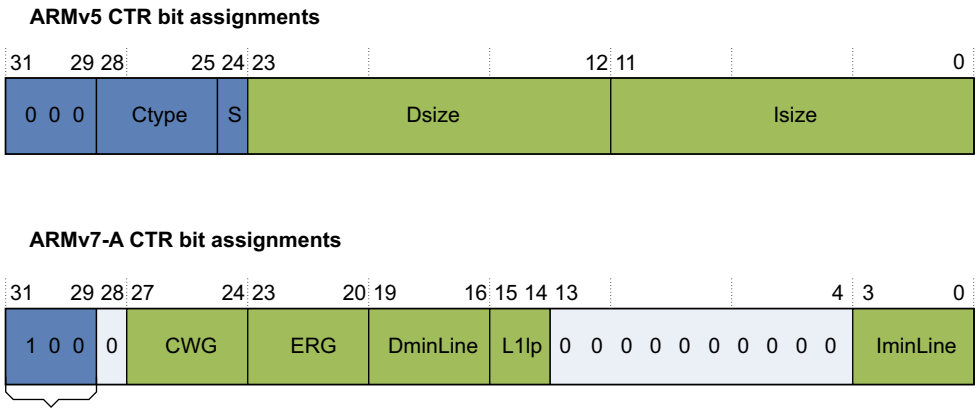

Figure 3-2 Bit assignments of Cache Type Register in ARMv5 and ARMv7-A

Bits[31:29] indicate the CTF format. In ARMv5, you can specify only one value, 0b000, which indicates the ARMv5 and ARMv6 format. In ARMv7-A, you can add another value, 0b100, to indicate the ARMv7 format.

The following points describe the descriptor fields for the Cache Type Register in an ARMv7 VMSA implementation:

Format, bits[31:29] Indicates the implemented CTR format.

Bit[28] RAZ.

CWG, bits[27:24] Cache Write-back Granule.

ERG, bits[23:20] Exclusives Reservation Granule.

DminLine, bits[19:16] Log2 of the number of words in the smallest cache line of all the data caches and unified caches that are controlled by the processor.

L1Ip, bits[15:14] Level 1 instruction cache policy. Indicates the indexing and tagging policy for the L1 instruction cache.

Bits[13:4] RAZ.

IminLine, bits[3:0] Log2 of the number of words in the smallest cache line of all the instruction caches that are controlled by the processor.

CP15 c1, System Control Register

Figure 3-3 shows the bit assignments of the System Control Register in ARMv5 and ARMv7-A.

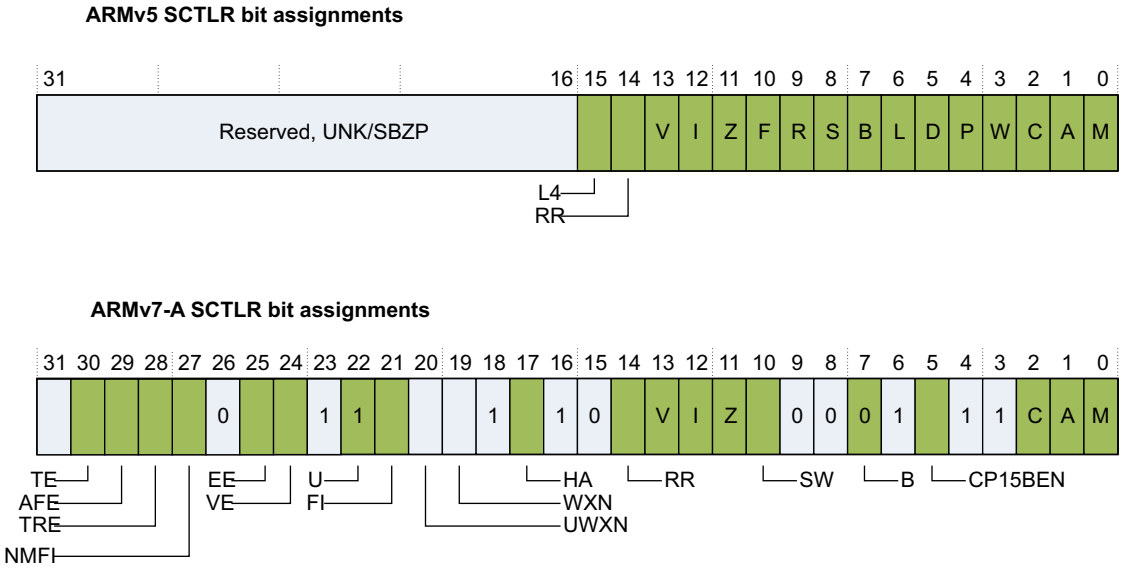

Figure 3-3 Bit assignments of System Control Register in ARMv5 and ARMv7

The following usage models in ARMv5 are not compatible with ARMv7-A:

- Bits[31:16] Reserved, UNK/SBZP.
- The following reserved bits in the SCTLR are allocated in certain circumstances:
  - Bits[19:16] can be associated with TCM support.
  - Bit[26], described as the L2 bit, can indicate level 2 cache support.

From ARMv6, ARM deprecates any use of the following features, and ARMv7-A does not support these features:

L4, bit [15] When set, this bit inhibits ARMv5T Thumb interworking behavior when set. It stops bit[0] updating the CPSR.T bit.

R, bit [9] ROM protection bit, supported for backwards compatibility.

S, bit[8] System protection bit, supported for backwards compatibility.

B, bit[7] This bit configures the ARM processor to the endianness of the memory system:

- 0 Little-endian memory system (LE).
- 1 Big-endian memory system (BE-32).

W, bit[3] It is the enable bit for the write buffer:

- 0 Write buffer disabled.
- 1 Write buffer enabled.

CP15 c13, VMSA FCSE support

The *Fast Context Switch Extension* (FCSE) is an implementation-defined option in ARMv5. The feature is supported by the FCSEIDR. ARMv7-A supports FCSEIDR.

The ARMv7-A Context ID and Software Thread ID registers are not supported in ARMv5.

3.1.3 Program status register

Figure 3-4 shows the bit assignments of the *Current Program Status Register* (CPSR) and *Saved Program Status Registers* (SPSR) registers in ARMv5 and ARMv7-A.

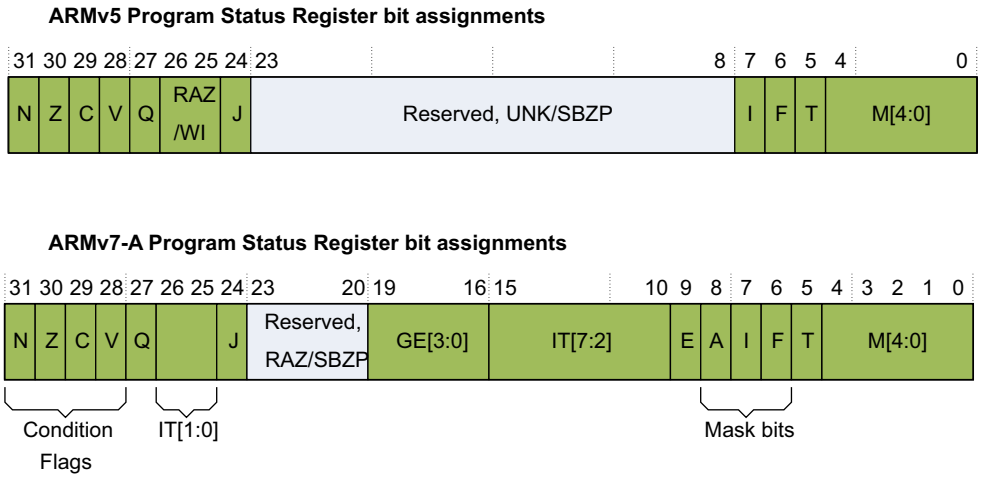

Figure 3-4 Program Status Register

In ARMv5, the definitions and general rules for the defined PSR bits are the same as in ARMv7-A, with the following exceptions:

- The IT[7:0], GE[3:0], E, and A bits are not implemented in ARMv5, but are defined in ARMv7.
- ARMv7-A introduces support for Mon mode and HYP mode, so the associated M[4:0] encoding is a reserved value in ARMv5, but is defined in ARMv7-A.
- Before ARMv5TEJ, the J bit is not implemented, but it is defined in ARMv7-A.

3.1.4 VFPv3 double-precision registers

In ARMv5, VFPv2 uses only 16 double-precision registers. In ARMv7, however, the number of double-precision registers depends on the implemented VFP architecture.

If VFPv3-D16 is implemented, there are 16 double-precision registers. If VFPv3-D32 is implemented, there are 32 double-precision registers.

3.2 ISA instructions

Compared with ARMv5, ARMv7 makes some changes to the instruction set, and adds some powerful new features and extra capabilities to the architecture. The changes can be classified into the following categories:

- *Thumb-2*.
- *VFPv3 and NEON*.
- *Alignment* on page 3-8.
- *Other ARMv7 new instructions* on page 3-7.

3.2.1 Thumb-2

ARMv7 supports Thumb-2 technology, which is a major enhancement to the 16-bit Thumb instruction set. It adds 32-bit instructions that can be freely intermixed with 16-bit instructions in a program. The additional 32-bit instructions enable Thumb-2 to cover most of the functionality of the ARM instruction set. The availability of 16-bit and 32-bit instructions combines the code density of earlier versions of Thumb with the performance of the ARM instruction set.

An important difference between the ARMv7-A Thumb instruction set and the ARMv7-A ARM instruction set is that most 32-bit Thumb instructions are unconditional, whereas most ARM instructions can be conditional. However, Thumb-2 introduces a conditional execution instruction `IT`, which is a logical *if-then-else* operation. You can apply the `IT` instruction to up to four subsequent instructions to make them conditional.

3.2.2 VFPv3 and NEON

VFPv2 is an extension to the ARM instruction set in ARMv5, while VFPv3 is an extension to the ARM, Thumb, and ThumbEE instruction sets in ARMv7. VFPv3 is backward-compatible with VFPv2, unless:

- All supported VFP instructions are implemented in hardware.
- Floating point exceptions are not trapped.
- Software support code is not required.
- Vector operation support is deprecated.

VFPv3 instructions can be executed conditionally with the `IT` instruction in Thumb state. In VFPv3, new instructions are added to perform the following operations:

- Placing floating point constant in a register.
- Converting between floating point and fixed point.

ARMv7 introduces support for NEON instructions, which are suitable for handling coding or code generation by the ARM vectoring compiler. All mnemonics for NEON instructions begin with the letter “V”. NEON instructions can operate on different data types.

NEON instructions can be classified into the following categories according to their functions:

- Data processing instructions:
  - Simple mathematical operations.
  - Basic logical operations.
  - Other basic and miscellaneous instructions.
- Memory access instructions:
  - Register Load and Store instructions to load NEON registers.
  - Element Load and Store instructions to load vector elements inside registers.

3.2.3 Other ARMv7 new instructions

ARMv7 also includes other new instructions that can be classified into the following categories:

- Exception-generating instructions:
  - `SMC` – Requests a Secure Monitor function, causing the processor to enter Mon mode.
  - `HVC` – Requests a Hypervisor function, causing the processor to enter Hyp mode. For more information.
- New exclusive instructions:
  - `LDREX` – Loads a word from a location and marks it for exclusive access.
  - `LDREXB` – Loads a byte from a location and marks it for exclusive access.
  - `LDREXD` – Loads a 64-bit doubleword from a location and marks it for exclusive access.
  - `LDREXH` – Loads a halfword from a location and marks it for exclusive access.
  - `STREX` – Stores a word to a location marked for exclusive access.
  - `STREXB` – Stores a byte to a location marked for exclusive access.
  - `STREXD` – Stores a 64-bit doubleword to a location marked for exclusive access.
  - `STREXH` – Stores a halfword to a location marked for exclusive access.
  - `CLREX` – Clears any exclusive access tag marked for a location.
- New Move Wide variants:
  - `MOVT` – Loads 16-bit immediate constant into top half of register.
  - `MOVW` – Loads 16-bit immediate constant into bottom half of register.
- New “T” variants for `LDR`, `SDR` subword instructions:
  - `LDRHT` – Loads register halfword unprivileged.
  - `LDRSBT` – Loads register signed byte unprivileged.
  - `LDRSHT` – Loads register signed halfword unprivileged.
  - `STRHT` – Stores register halfword unprivileged.
- Hint instructions:
  - `WFE` – Waits for an event.
  - `SEV` – Sends an event.
  - `WFI` – Waits for an Interrupt.
- Bit manipulation instructions:
  - `BFI` – Bit Field Insert.
  - `UBFX` – Bit Field Extract.
  - `BFC` – Bit Field Clear.
  - `RBIT` – Reverse Bit Order.
- Ordering and synchronization instructions:
  - `DMB` – Data memory barrier.
  - `DSB` – Data synchronization barrier.
  - `ISB` – Instruction Synchronization barrier.

3.2.4 Alignment

ARMv5 requires the addresses for data load and store instructions to be aligned to their natural boundary. However, ARMv7 introduces hardware support for unaligned accesses by some load and store instructions. Support for unaligned accesses is limited to the following load and store instructions:

- `LDRH, LDRHT, LDRSH, LDRSHT, STRH, STRHT, TBH`.
- `LDR, LDRT, STR, STRT`.
- `VLD1, VLD2, VLD3, VLD4, VST1, VST2, VST3, VST4`.

Unaligned accesses are only permitted to regions of the Normal memory type. Software can set the SCTLR.A bit to control whether a misaligned access to Normal memory by one of these instructions causes an Alignment fault Data Abort exception.

3.3 Exception model

The exception model in ARMv7 differs from that in ARMv5 in the following areas:

- *Entry into Supervisor mode*.
- *Mon mode and Hyp mode*.
- *Secure state and Non-secure state*.
- *Privilege levels* on page 3-10.

3.3.1 Entry into Supervisor mode

In ARMv5, you enter Supervisor mode by executing a *Software Interrupt instruction* (`SWI`). In ARMv7, however, Software Interrupt is called Supervisor Call (`SVC`). The `SVC` instruction and the `SWI` instruction are the same instruction, only with different names.

In ARMv7, you can use the `SVC` instruction to request a Supervisor function, causing the processor to enter Supervisor mode. Typically, you can use an `SVC` instruction to request an operating system function.

3.3.2 Mon mode and Hyp mode

In ARMv7-A, the following instructions are added to support two additional processor modes, Mon mode and Hyp mode:

- Secure Monitor Call (`SMC`) instruction – Requests a Secure Monitor function, causing the processor to enter Mon mode.
- Hypervisor call (`HVC`) instruction – Requests a Hypervisor function, causing the processor to enter Hyp mode.

3.3.3 Secure state and Non-secure state

ARMv7-A defines the Secure state and Non-secure state. In this document and many others, the Secure state is also known as *Secure world* while the Non-secure state is also known as *Normal world*.

In ARMv5, processor modes do not have Secure state and Non-secure state. In ARMv7-A, the introduction of the TrustZone Security Extension creates two security states for all processor modes, except Mon mode and Hyp mode, as shown in Table 3-1.

Table 3-1 Processor modes and security states

Modes	Security states
User (USR)	Both
FIQ	Both
IRQ	Both
Supervisor (SVC)	Both
Monitor (MON)	Secure only
Abort (ABT)	Both
Hypervisor (HYP)	Non-secure only
Undef (UND)	Both
System (SYS)	Both

3.3.4 Privilege levels

In ARMv5, User modes are known as unprivileged modes, while the following modes are known as privileged modes:

- FIQ.
- IRQ.
- Supervisor.
- Abort.
- Undefined.
- System.

In ARMv7-A with the Virtualization implementation, the privileged model is different from that in ARMv5. In Non-secure state, ARMv7-A has 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.

PL1 modes refer to all modes other than User mode and Hyp mode.

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

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

If Virtualization Extensions are implemented, a Hypervisor executes at PL2 (Hyp mode). A Hypervisor controls and enables multiple Operating Systems to co-exist and execute on the same processor system.

These privilege levels are separate from the TrustZone Secure and Non-secure settings. *Privilege levels and security* on page 3-34 shows processor modes and their corresponding privilege levels.

Note

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

3.4 Memory model

The memory model in ARMv7 differs from that in ARMv5 in the following areas:

- ARMv7 introduces new memory attributes, which are described in *Memory management*.
- ARMv7 supports different page sizes and has different descriptor formats, which are described in *Page size support* on page 3-12.
- ARMv7-A introduces many new or changed MMU registers, which are described in *MMU registers* on page 3-15.
- ARMv7 deprecates the use of *Instruction Memory Barriers* (`IMB`s), and introduces new barriers such as `DMB`, `DSB`, and `IMB`. For more information, see *Barriers* on page 3-18.
- ARMv7 deprecates the use of the synchronization instructions `SWP` and `SWPB`, and introduces new synchronization instructions `LDREX` and `STREX`. For more information, see *Synchronization* on page 3-19.

3.4.1 Memory management

In ARMv5, you can specify the memory access behavior of pages by configuring whether the cache and write buffer can be used for that location. This scheme is inadequate for more complex systems and processors. Therefore, ARMv7 adds the following mutually exclusive memory types:

- Strongly-ordered.
- Device.
- Normal.

Table 3-2 summarizes the memory attributes in ARMv7.

Table 3-2 Memory attributes summary

Memory type	Shareable/ Non-shareable	Cacheable	Description
Normal	Shareable	Yes	Designed to handle normal memory that is shared among 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. 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.

In addition, ARMv7 supports the following memory attributes:

- Secure attribute.

  Memory can be defined in Secure state or Non-secure state.
- Shared attribute.

  Memory can be Shareable or non-Shareable.

Page size support

In ARMv5, the following page sizes are supported:

- 1KB.
- 4KB.
- 64KB.

In ARMv7-A without the LPAE implementation, the following page sizes are supported:

- 4KB.
- 64KB.

In ARMv7-A with the LPAE implementation, the following page sizes are supported:

- 4KB.
- 64KB.
- 1MB.
- 2MB.
- 1GB.

Note

The 1KB page size is no longer supported in ARMv7.

Short-descriptor format

In ARMv7-A, *Short-descriptor format* is the only format supported on implementations that do not include the LPAE extension.

The Short-descriptor format uses 32-bit descriptor entries in the translation tables, and includes the following features:

- Up to two levels of address lookup.
- 32-bit input addresses.
- Support for *Physical Addresses* (PAs) of more than 32 bits by use of supersections, with 16MB granularity.
- Support for No access, Client, and Manager domains.
- 32-bit table entries.

Each entry in the first-level table describes the mapping of the associated 1MB MVA range. Figure 3-5 on page 3-13 shows the possible first-level descriptor formats:

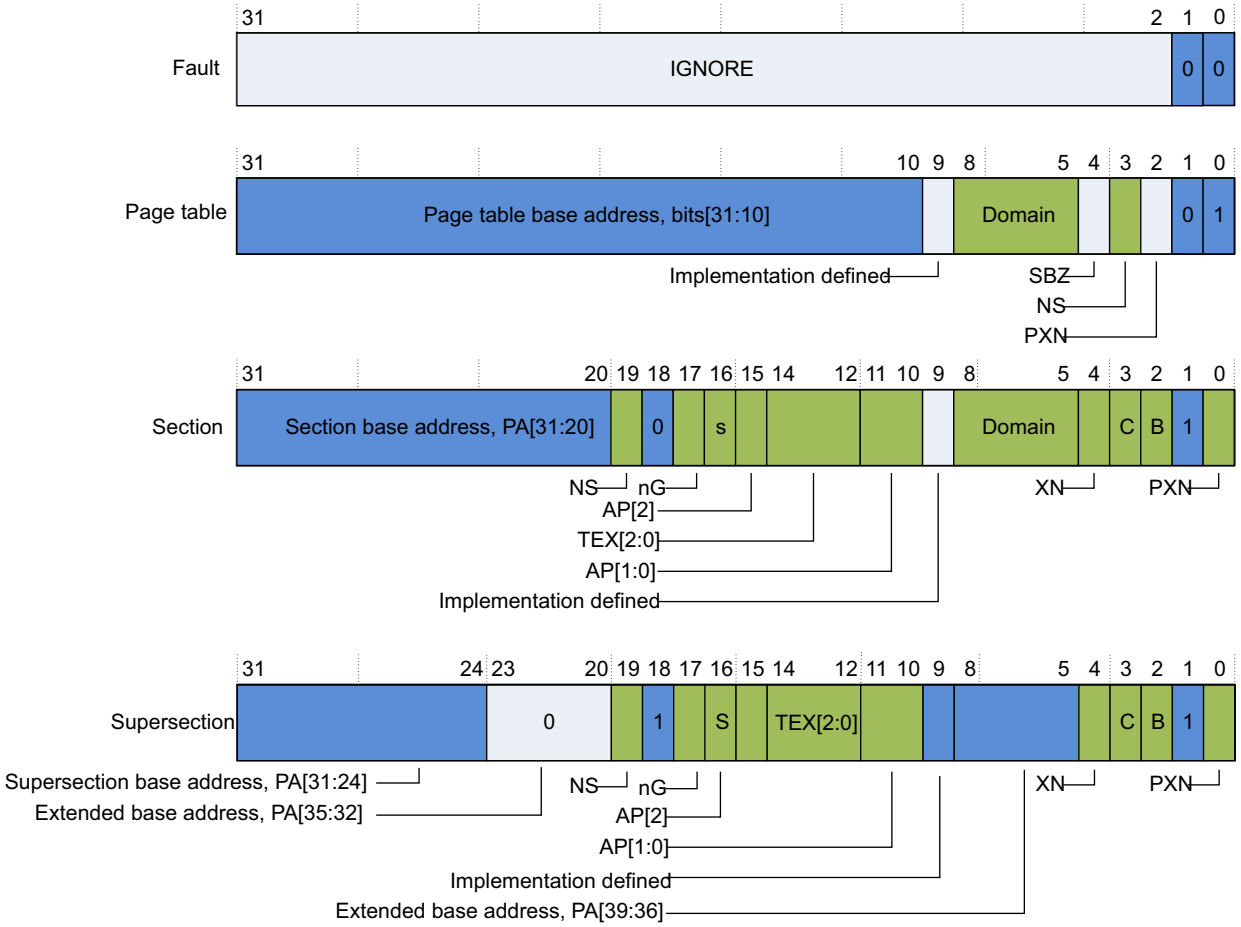

Figure 3-5 Short-descriptor first-level descriptor formats

Figure 3-6 shows the possible formats of a second-level descriptor:

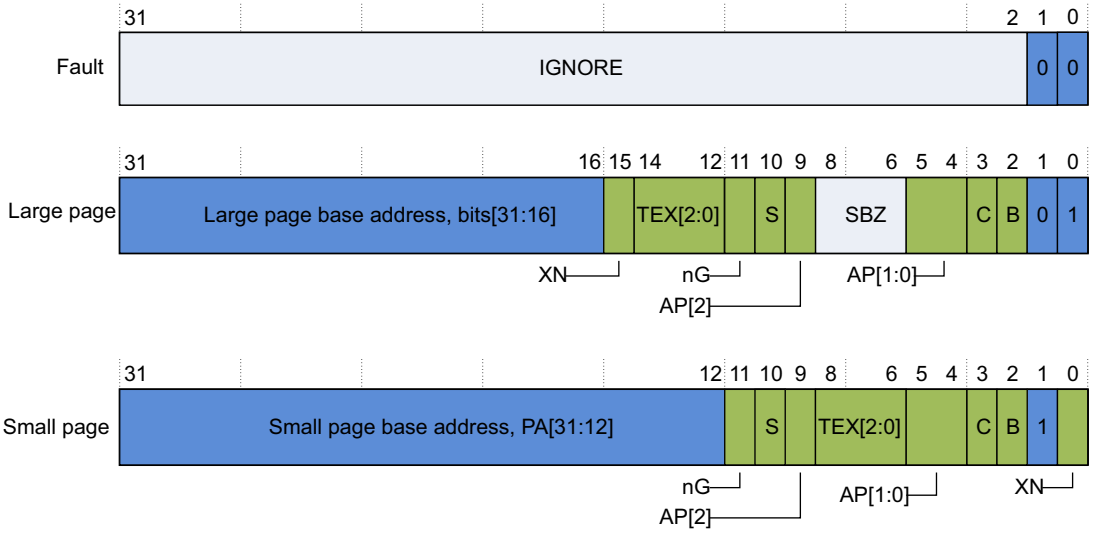

Figure 3-6 Short-descriptor second-level descriptor formats

The following points describe the descriptor fields, with the exception of the descriptor type field and the address field:

TEX[2:0], C, B Memory region attribute bits.

These bits are not present in a Page table entry.

XN bit The Execute-never bit, which determines whether the processor can execute software from the addressed region.

This bit is not present in a Page table entry.

PXN bit The Privileged Execute-never bit, when supported, determines whether the processor can execute software from the region when executing at PL1.

NS bit Non-secure bit, which specifies whether the translated PA is in the Secure or Non-secure address map.

Note

This bit is not present in second-level descriptors, and it is only applicable when CPU is in Secure state.

Domain Domain field.

This field is not present in a Supersection entry. This bit is not present in second-level descriptors.

AP[2], AP[1:0] Access Permissions bits.

These bits are not present in a Page table entry.

S bit The Shareable bit, which determines whether the addressed region is Shareable memory.

This bit is not present in a Page table entry.

nG bit The Not Global bit, which determines how the translation is marked in the TLB.

This bit is not present in a Page table entry.

Bit[18], when bits[1:0] indicate a Section or Supersection descriptor

0 – Descriptor is for a Section.

1 – Descriptor is for a Supersection.

Large Physical Address Extension

ARMv7-A introduces the LPAE extension. The LPAE provides an address translation system supporting physical addresses of up to 40 bits at a fine grain of translation. To do this, LPAE uses the *Long-descriptor* format.

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, as shown in Figure 3-19 on page 3-34. The second stage of this translation process is configured and controlled by the Hypervisor. In addition to an *Address Space ID* (ASID), TLB entries can also have an associated *Virtual Machine ID* (VMID). You can also disable the stage 2 MMU and have a flat mapping from IPA to PA.

The Long-descriptor format uses 64-bit descriptor entries in the translation tables, and includes the following features:

- Up to three levels of address lookup.

- Input addresses of up to 40 bits, when used for stage 2 translations.
- Output addresses of up to 40 bits.
- 4KB assignment granularity across the entire PA range.
- No support for domains, all memory regions are treated as in a Client domain.
- 64-bit table entries.
- Fixed 4KB table size, unless truncated by the size of the input address space.

In the Long-descriptor translation tables, the formats of the first-level and second-level descriptors differ only in the size of the block of memory addressed by the block descriptor. Figure 3-7 shows the Long-descriptor first-level and second-level descriptor formats:

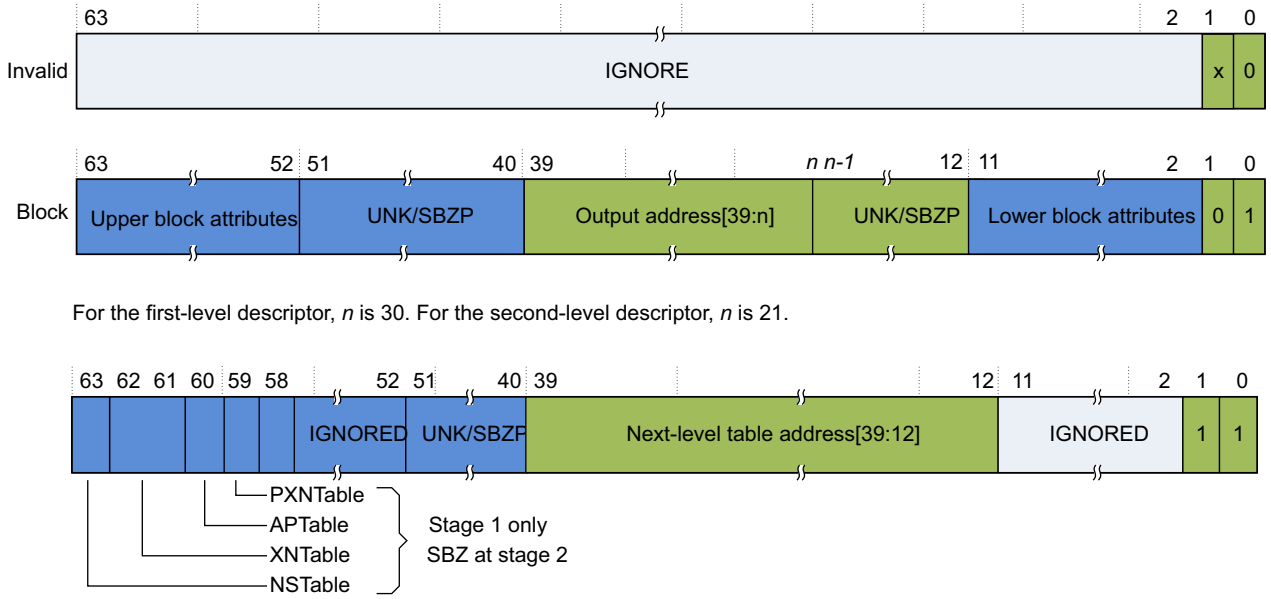

Figure 3-7 Long-descriptor first-level and second-level descriptor formats

Each entry in a third-level table describes the mapping of the associated 4KB input address range. Figure 3-8 shows the Long-descriptor third-level descriptor formats:

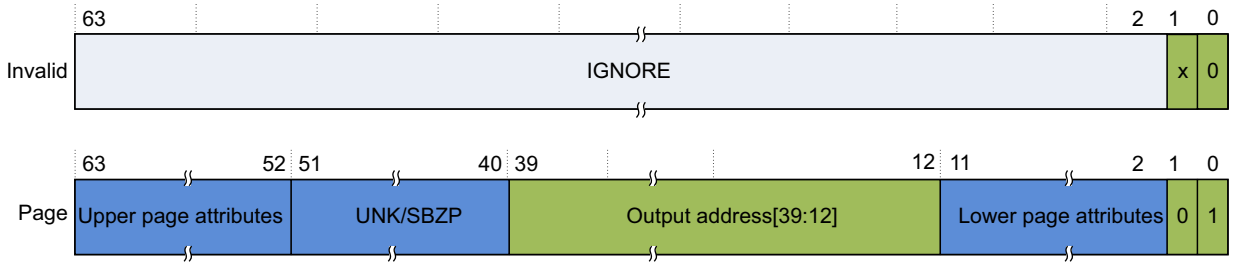

Figure 3-8 Long-descriptor third-level descriptor formats

MMU registers

Compared with ARMv5, ARMv7-A has much larger MMU registers. Table 3-3 on page 3-16 shows an overview of all new or changed MMU registers, in coprocessor register number order.

For detailed information about these registers, see section *B3.17 Organization of the CP15 registers in a VMSA implementation* in the ARM® Architecture Reference Manual ARMv7-A and ARMv7-R Edition, ARM DDI 0406C.

Table 3-3 New or changed MMU registers

Name	New or Changed	Width	Description
SCTLR	Changed	32-bit	System Control Register
ACTLR	New	32-bit	IMPLEMENTATION DEFINED Auxiliary Control Register
CPACR	New	32-bit	Coprocessor Access Control Register
SCR	New	32-bit	Secure Configuration Register
SDER	New	32-bit	Secure Debug Enable Register
NSACR	New	32-bit	Non-secure Access Control Register
HSCTLR	New	32-bit	Hypervisor System Control Register
HACTLR	New	32-bit	Hypervisor Auxiliary Control Register
HCR	New	32-bit	Hypervisor Configuration Register
HDCR	New	32-bit	Hypervisor Debug Configuration Register
HCPTR	New	32-bit	Hypervisor Coprocessor Trap Register
HSTR	New	32-bit	Hypervisor System Trap Register
HACR	New	32-bit	Hypervisor Auxiliary Configuration Register
TTBR0	New	32-bit	Translation Table Base Register 0
TTBR0	New	64-bit	Translation Table Base Register 0
TTBR1	New	32-bit	Translation Table Base Register 1
TTBR1	New	64-bit	Translation Table Base Register 1
TTBCR	New	32-bit	Translation Table Base Control Register
HTCR	New	32-bit	Hypervisor Translation Control Register
VTCR	New	32-bit	Virtualization Translation Control Register
HTTBR	New	32-bit	Hypervisor Translation Table Base Register
VTTBR	New	32-bit	Virtualization Translation Table Base Register
DFSR	New	32-bit	Data Fault Status Register
IFSR	New	32-bit	Instruction Fault Status Register
ADFSR	New	32-bit	Auxiliary Data Fault Status Register
AIFSR	New	32-bit	Auxiliary Instruction Fault Status Register
HADFSR	New	32-bit	Hypervisor Auxiliary Data Fault Syndrome Register
HAIFSR	New	32-bit	Hypervisor Auxiliary Instruction Fault Syndrome Register
HSR	New	32-bit	Hypervisor Syndrome Register

Table 3-3 New or changed MMU registers (continued)

Name	New or Changed	Width	Description
DFAR	New	32-bit	Data Fault Address Register
IFAR	New	32-bit	Instruction Fault Address Register
HDFAR	New	32-bit	Hypervisor Data Fault Address Register
HIFAR	New	32-bit	Hypervisor Instruction Fault Address Register
HPFAR	New	32-bit	Hypervisor IPA Fault Address Register
PAR	New	32-bit	Physical Address Register
PAR	New	64-bit	Physical Address Register
PRRR	New	32-bit	Primary Region Remap Register
MAIR0	New	32-bit	Memory Attribute Indirection Register 0
NMRR	New	32-bit	Normal Memory Remap Register
MAIR1	New	32-bit	Memory Attribute Indirection Register 1
AMAIR0	New	32-bit	Auxiliary Memory Attribute Indirection Register 0
AMAIR1	New	32-bit	Auxiliary Memory Attribute Indirection Register 1
HMAIR0	New	32-bit	Hypervisor Memory Attribute Indirection Register 0
HMAIR1	New	32-bit	Hypervisor Memory Attribute Indirection Register 1
HAMAIR0	New	32-bit	Hypervisor Auxiliary Memory Attribute Indirection Register 0
HAMAIR1	New	32-bit	Hypervisor Auxiliary Memory Attribute Indirection Register 1
VBAR	New	32-bit	Vector Base Address Register
MVBAR	New	32-bit	Monitor Vector Base Address Register
ISR	New	32-bit	Interrupt Status Register
HVBAR	New	32-bit	Hypervisor Vector Base Address Register
FCSEIDR	New	32-bit	FCSE Process ID Register
CONTEXTIDR	New	32-bit	Context ID Register
TPIDRURW	New	32-bit	User Read/Write Thread ID Register
TPIDRURO	New	32-bit	User Read-Only Thread ID Register
TPIDRPRW	New	32-bit	PL1 only Thread ID Register
HTPIDR	New	32-bit	Hypervisor Software Thread ID Register

3.4.2 Barriers

In ARMv5, you can use `IMB` to ensure consistency between data and instruction streams. You can use `IMB` when treating code as data, for example when using self-modifying code, or when loading code into memory.

In ARMv7, `IMB` is deprecated, and the following memory barriers are introduced:

- `DSB`
- `DMB`
- `ISB`

For more information about the `DSB`, `DMB`, and `ISB` memory barriers, see *Replacing ARMv5 barriers with equivalent ARMv7 barriers* on page 4-8.

3.5 Synchronization

Mutual exclusion is often the preferred method for protecting shared resources. The section of code that is being executed by a core while accessing such a shared resource is known as the *critical section*.

In ARMv5, you can use the `SWP` and `SWPB` instructions for synchronization. `SWP` and `SWPB` provide a method for software synchronization that does not require disabling interrupts. You can achieve this by performing a special type of memory access, that is, reading a value into a processor register and writing a value to a memory location as an atomic operation.

In ARMv7, the `SWP` and `SWPB` instructions are deprecated because they have the following limitations:

- If an interrupt triggers while a swap operation is taking place, the processor must complete both the load and the store part of the instruction before taking the interrupt. This increases interrupt latency.
- In a multi-core system, preventing accesses to main memory for all processors for the duration of a swap instruction can reduce overall system performance. This is especially true in a multi-core system where processors operate at different frequencies but share the same main memory.

ARMv7 introduces Load-Exclusive and Store-Exclusive synchronization primitives, `LDREX` and `STREX`, to provide flexible and scalable synchronization. For more information about the new synchronization primitives, see *Replacing ARMv5 synchronization primitives with equivalent ARMv7 synchronization primitives* on page 4-11.

3.6 Multiprocessing

The Multiprocessing extension provides a set of features that enhance multiprocessing functionality. ARMv7-A supports Multiprocessing while ARMv5 does not.

3.6.1 Cache

The cache differences between ARMv5 and ARMv7-A are in the following areas:

- *Data cache type*.
- *Data cache clean on context switches*.
- *Cache behavior at reset*.
- *L2 cache support*.

Data cache type

In ARMv5, data cache is organized as a *Virtually-indexed, Virtually-tagged* (VIVT) cache, in which both the index and the tag are based on the virtual address. This caching method can require the cache to be flushed when virtual address to physical address mappings are changed.

In ARMv7-A, data cache is organized as *Physically-indexed, Physically-tagged* (PIPT), in which both the index and the tag are based on the physical address. PIPT cache is simple and avoids problems with aliasing.

Data cache clean on context switches

In ARMv5, you might need to perform a data cache clean on context switches. In ARMv7-A, there is no need to perform a data cache clean on context switches.

Note

A context switch means that the scheduler transfers 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.

Cache behavior at reset

In ARMv5, cache is invalidated by hardware at reset. In ARMv7-A, cache invalidation by hardware is implementation-defined.

L2 cache support

Existing ARMv5 processors do not support the L2 cache, while ARMv7-A can support the L2 cache.

3.6.2 Translation Lookaside Buffers

Full translation table lookup is performed automatically by hardware, and has a significant execution-time cost. To reduce the average cost of a memory access, the results of translation table walks are cached in one or more structures known as *Translation Lookaside Buffers* (TLBs).

In ARMv5, you must invalidate the entire TLBs on context switches. However, ARMv7-A introduces support for *Address Space IDs* (ASIDs), and does not need to invalidate the entire TLBs on context switches. For more information about ASIDs, see *Address Space Identifiers* on page 3-23.

3.6.3 Generic Interrupt Controller

Generic Interrupt Controller (GIC) provides a set of hardware resources for managing interrupts in a single-core or multi-core system, as shown in Figure 3-9.

The GIC provides memory-mapped registers that can be used to manage interrupt sources and, in multi-core systems, for routing interrupts to individual cores.

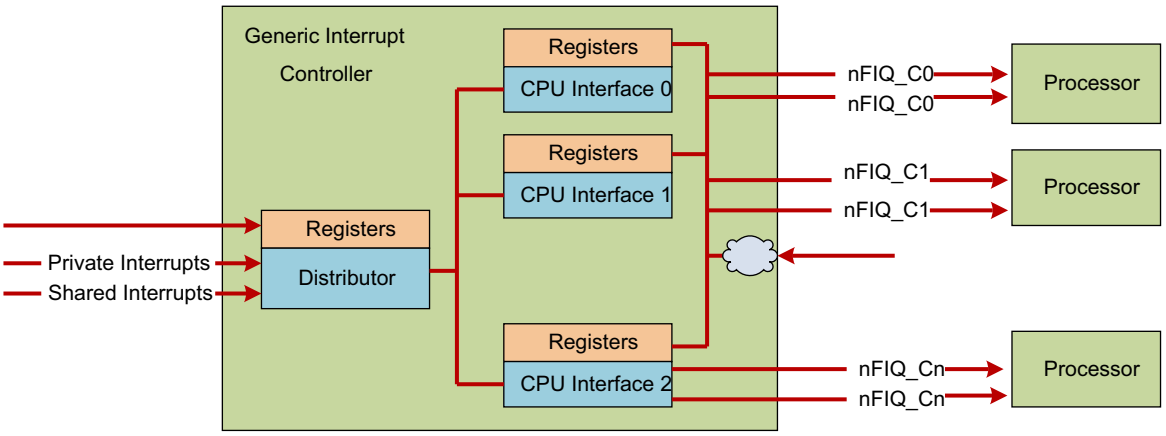

Figure 3-9 Generic Interrupt Controller

The GIC enables software to perform the following tasks:

- Masking, enabling, and disabling interrupts from individual sources.
- Prioritizing individual sources in hardware.
- Generating software interrupts.

Interrupts can be of the following types:

***Software Generated Interrupt* (SGI)**

This interrupt is generated explicitly by software by writing to a dedicated distributor register, the Software Generated Interrupt Register. It is most commonly used for inter-core communication. SGIs can be targeted at all, or at a selected group of cores in the system. Interrupt numbers 0-15 are reserved for this. The software manages the exact interrupt number used for communication.

***Private Peripheral Interrupt* (PPI)**

This interrupt is generated by a peripheral that is private to an individual core. Interrupt numbers 16-31 are reserved for this. PPIs identify interrupt sources private to the core, and are independent of the same source on another core, for example, per-core timer.

***Shared Peripheral Interrupt* (SPI)**

This interrupt 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 system.

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, and 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 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 mask bits in the CPSR.

You can disable the entire interrupt mechanism in the system by disabling the distributor block. You can disable interrupt delivery to an individual core by disabling its CPU interface block, or by setting mask bits in the CPSR of that core. You can also disable or enable individual interrupts 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 must be cleared.

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.

The read returns the interrupt ID, and causes the interrupt to be marked as active in the distributor. When the interrupt ID is known, the top-level handler can 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 the Interrupt register in the CPU Interface block, indicating the end of interrupt processing.

Apart from removing the active status, which makes the final interrupt status either Inactive or Pending, if the state was Active and Pending, this enables the CPU Interface to forward more pending interrupts to the core. This concludes the processing of a single interrupt.

There can be more than one interrupt waiting to be serviced on the same core. However, 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 reads the spurious interrupt ID, it can complete its execution, and prepare the core to resume the task it was doing before taking the interrupt.

3.7 Operating system support

Operating system (OS) support in ARMv7 differs from that in ARMv5 in the following areas:

- *Dual Translation Tables*.
- *Context saving*.
- *Address Space Identifiers*.
- *Generic Timer*.
- *Symmetric Multi-Processing* on page 3-24.

3.7.1 Dual Translation Tables

In ARMv7-A, you can split the 32-bit virtual address space into two regions, one for user application and the other one for kernel mapping. The application space changes on each context switch, but the kernel space is likely to remain static.

On a task switch, you must update the page tables to those for the newly switched task. You can do this by rewriting the existing page table, or by changing the TTBR to point to a different table. However, rewriting the table takes time, and changing to a different table would duplicate the OS portion of the table each time.

To avoid these problems, you use *Virtual Memory System Architecture* (VMSA), which split the virtual address space across two L1 tables. The TTBR1 register points to one table containing the kernel-related translations, and this table remains constant across task switches. The other table contains the application-specific space. Each process has its own table, with TTBR0 redirected to the appropriate table on a task switch.

3.7.2 Context saving

In ARMv7-A, the kernel must save or restore NEON registers and Thread ID registers on context switches. ARMv5 does not have these registers.

3.7.3 Address Space Identifiers

ARMv7-A introduces support for ASID, which is a number assigned by the OS to each individual task. This value is in the range of 0-255, and the value for the current task is written in the ASID register.

When the TLB is updated and the entry is marked as non-global, the ASID value is stored in the TLB entry in addition to the normal translation information. Subsequent TLB look-ups only match an entry if the current ASID matches with the ASID 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.

3.7.4 Generic Timer

ARMv7 introduces support for the Generic Timer, which can trigger events after a period has elapsed. It provides:

- A physical counter that contains the count value of the system counter.
- A virtual counter that indicates virtual time.
- Generation of timer events as interrupt outputs.
- Generation of event streams.
- Support for Virtualization Extensions.

With the Generic Timer extension, ARMv7 does not need an external timer to tick the kernel as ARMv5 does.

3.7.5 Symmetric Multi-Processing

Symmetric Multi-Processing (SMP) extensions are supported in ARMv7-A, but not in ARMv5. 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 shared hardware. Any application, process, or task, can run on any core. The operating system scheduler can dynamically migrate tasks among cores to achieve optimal system load.

3.8 Floating-Point Unit

ARM *Floating Point architecture* (VFP) provides hardware support for floating point operations in half-, single-, and double-precision floating point arithmetic.

3.8.1 ARM VFP Architecture versions

The following versions of the architecture have been implemented:

- VFPv1. This version is now obsolete.
- VFPv2. This version is an optional extension to the ARM instruction set in the ARMv5TE, ARMv5TEJ, and ARM6 architectures.
- VFPv3. This version is an optional extension to the ARM, Thumb, and ThumbEE instruction sets in ARMv7-A and ARMv7-R profiles.

  VFPv3 is backward-compatible with VFPv2, except that VFPv3 cannot trap floating-point exceptions and therefore requires no software support code.

  A VFPv3 implementation can be either 32 or 16 double word registers. VFPv3-D16 is a VFPv3 implementation that provides 16 double-precision registers. VFPv3-D32 is a VFPv3 implementation that provides 32 double-precision registers.

3.8.2 VFP comparison between ARM926 and Cortex-A9

To show the VFP architecture differences between ARMv5 and ARMv7, ARM926 and Cortex-A9 are compared below.

On ARM926, the optional VFP9-S coprocessor is a VFPv2 implementation. The VFP9-S coprocessor is optimized for the following purposes:

- Fast hardware execution of a high percentage of operations on normalized data.
- Divide and square root operations in parallel with other arithmetic operations to reduce the impact of long-latency operations.

On Cortex-A9, the FPU is a VFPv3-D16 implementation of the ARMv7 floating-point architecture. It supports all addressing modes and operations described in the *ARM® Architecture Reference Manual ARMv7-A and ARMv7-R Edition, ARM DDI 0406C*:

- 16, 64-bit Double Precision floating point registers.
- High data transfer bandwidth through 64-bit split load and store buses.

The FPU has the following features:

- Support for single-precision and double-precision floating-point formats.
- Support for conversion between half-precision and single-precision.
- Support for out-of-order completion of load transfers.
- Support for handling of normalized and denormalized data in hardware.
- Support for trapless operation to enable fast execution.
- Support for speculative execution.

The use of VFP vector mode is deprecated in ARMv7. Vector operations are not supported in the hardware. If you use vectors, a support code is required.

3.9 NEON

ARMv7 architecture introduces the advanced *Single Instruction Multiple Data* (SIMD) extension as an optional extension to the ARMv7-A and ARMv7-R profiles. It extends the SIMD concept by defining groups of instructions operating on vectors stored in 64-bit D, doubleword registers and 128-bit Q, quadword vector registers. The implementation of the Advanced SIMD extension used in ARM processors is called NEON, and the NEON technology is implemented in all current ARM Cortex-A series processors.

The NEON technology can accelerate multimedia and signal processing algorithms, such as video encode and decode, 2D and 3D graphics, gaming, audio and speech processing, image processing, telephony, and sound synthesis by at least three times the performance of ARMv5.

NEON is designed as an additional load and store architecture to provide vectorizing compiler support from languages, such as C and C++. NEON instructions operate on wide 64-bit and 128-bit vector registers, and form part of the normal ARM or Thumb code. Therefore, NEON instructions are easy to understand, and make hand-coding easy for applications that require the highest performance.

The NEON architecture uses a 32 × 64-bit register file. They are actually the same registers used by the floating-point unit, VFPv3. The compiler can use any NEON or VFP registers for floating-point values or NEON data at any point in the code. NEON differs from VFP primarily in the following aspects:

- NEON only works on vectors and does not support double-precision floating point.
- NEON does not support certain complex operations, such as square root and division.

For example, the `VADD.I16 Q0, Q1, Q2` instruction performs a parallel addition of eight lanes of 16-bit elements from vectors in Q1 and Q2, and stores the result in Q0, as shown in Figure 3-10.

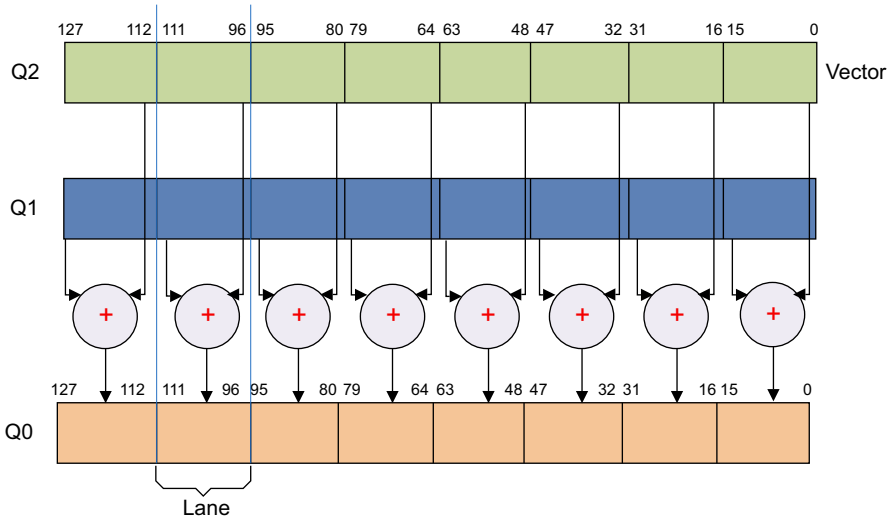

Figure 3-10 8-lane 16-bit integer add operation

3.9.1 Supported 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.

3.9.2 NEON registers

The NEON register bank consists of 32 × 64-bit registers. If both NEON and VFPv3 are implemented, they share this register bank. In this case, VFPv3 is implemented in the VFPv3-D32 form that supports 32 double-precision floating-point registers.

The NEON unit can view the same register bank as either one of the following register groups:

- 16 × 128-bit quadword registers, Q0-Q15.
- 32 × 64-bit doubleword registers, D0-D31.

3.9.3 NEON instructions

NEON instructions provide data processing and load and store operations only, and are integrated into ARM and Thumb instruction sets.

NEON instructions are executed as part of the ARM or Thumb instruction stream. This simplifies software development, debugging, and integration compared to using an external accelerator. Traditional ARM or Thumb instructions manage all program flow and synchronization.

NEON instructions can perform the following operations:

- Memory access.
- Data copying between NEON and general-purpose registers.
- Data type conversion.
- Data processing.

3.9.4 Example: Swapping color channels

The following example shows you the syntax and usage of NEON instructions.

This example involves a 24-bit RGB image. The red (R), green (G), and blue (B) pixels are arranged in memory in the sequence R0, G0, B0, R1, G2, B2, and so on. R0 is the first red pixel, R1 is the second red pixel, and so on.

This example shows you how to swap red and blue channels so that the sequence in memory becomes B0, G0, R0, B1, G1, R1, and so on. This is a simple signal processing operation, which NEON instructions can perform efficiently.

Figure 3-11 shows a normal load that pulls consecutive R, G, and B data from memory into registers. Code to swap channels based on this input requires mask, shift, and combine operations. It is not elegant and is unlikely to be efficient.

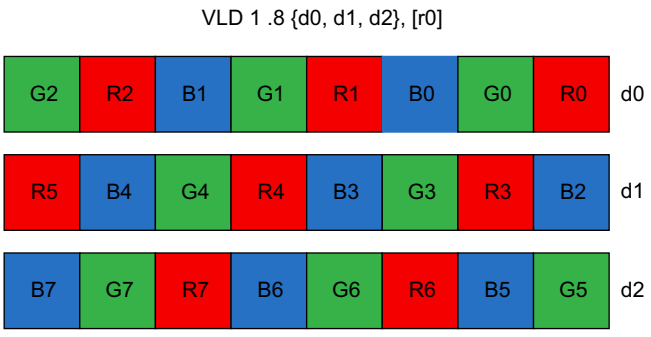

Figure 3-11 Loading RGB data with a linear load

NEON provides structure load and store instructions to help in these situations, as shown in Figure 3-12. These instructions pull in data from memory and simultaneously separate values into different registers. For this example, you can use `VLD3` to split up red, green, and blue when they are loaded.

Structure load instructions read data from memory into 64-bit NEON registers, with optional de-interleaving. Structure store instructions work similarly, reinterleaving data from registers before writing it to memory.

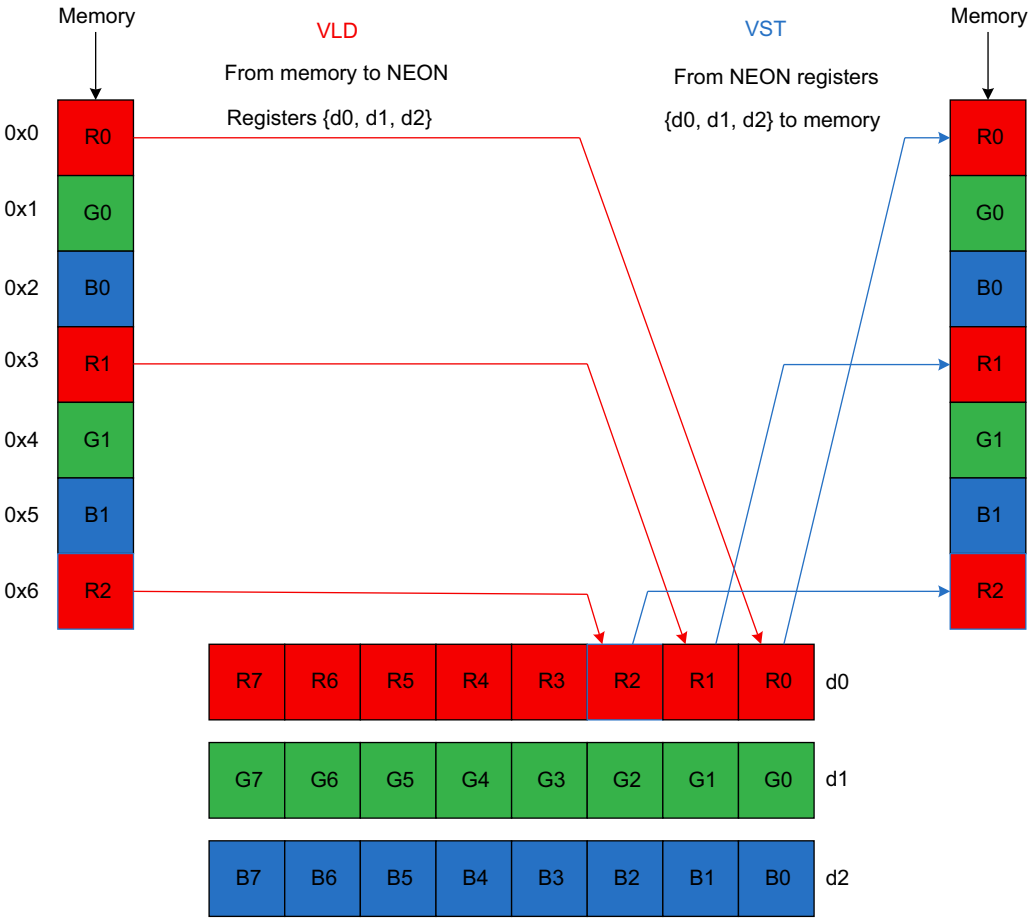

Figure 3-12 NEON structure loads and stores

Syntax

Figure 3-13 on page 3-29 shows the syntax of the structure load and store instructions. The syntax consists of five parts.

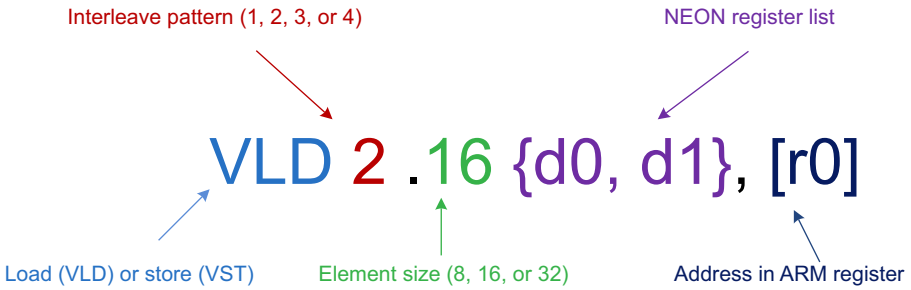

Figure 3-13 The structure load and store syntax

- The load instructions start with `VLD`, and the store instructions start with `VST`.
- A numeric *interleave pattern* is the number of registers to interleave. This is also the gap between corresponding elements in each structure.
- An *element size* specifies the number of bits in the accessed elements.
- A list of 64-bit *NEON registers* to load from or store to memory. The list can contain up to four registers, depending on the interleave pattern.
- An ARM *address register* contains the location to be accessed in memory to store to or load from. It is possible to update the address after the access.

Interleave pattern

Instructions can load, store, and de-interleave structures that contain from one to four equally sized elements. These elements usually have NEON-supported widths of 8, 16, or 32 bits.

- `VLD1` is the simplest form. It loads one to four registers of data from memory, with no de-interleaving. Use it when processing an array of non-interleaved data.
- `VLD2` loads two or four registers of data, and de-interleaves even and odd elements into those registers. For example, use it to separate stereo audio data into left and right channels.
- `VLD3` loads three registers and de-interleaves. For example, use it to split RGB pixels into channels.
- `VLD4` loads four registers and de-interleaves. For example, use it to process RGB image data.

Store instructions support the same options, but interleave the data from registers before writing them to memory.

Element sizes

Load and store instructions interleave elements based on the size specified to the instruction. For example, loading two NEON registers with `VLD2.16` results in four 16-bit elements in the first register and four 16-bit elements in the second, with even and odd elements in adjacent pairs separated to each register, as shown in Figure 3-14 on page 3-30.

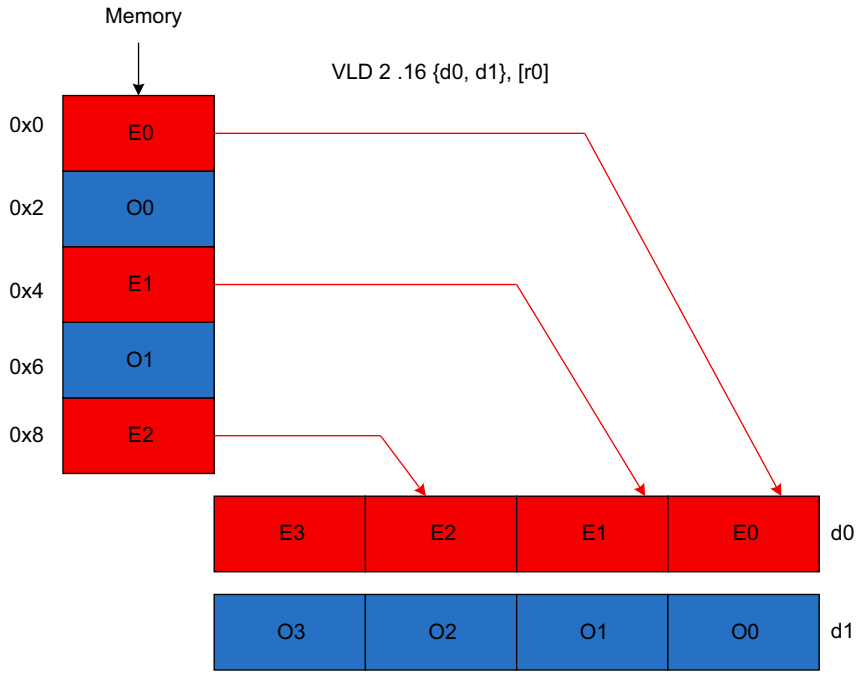

Figure 3-14 Loading and de-interleaving 16-bit data

Changing the element size to 32 bits causes the same amount of data to be loaded. However, only two elements make up each vector, because each element is 32 bits rather than 16 bits. This still separates the even and odd elements from memory into separate registers, as shown in Figure 3-15.

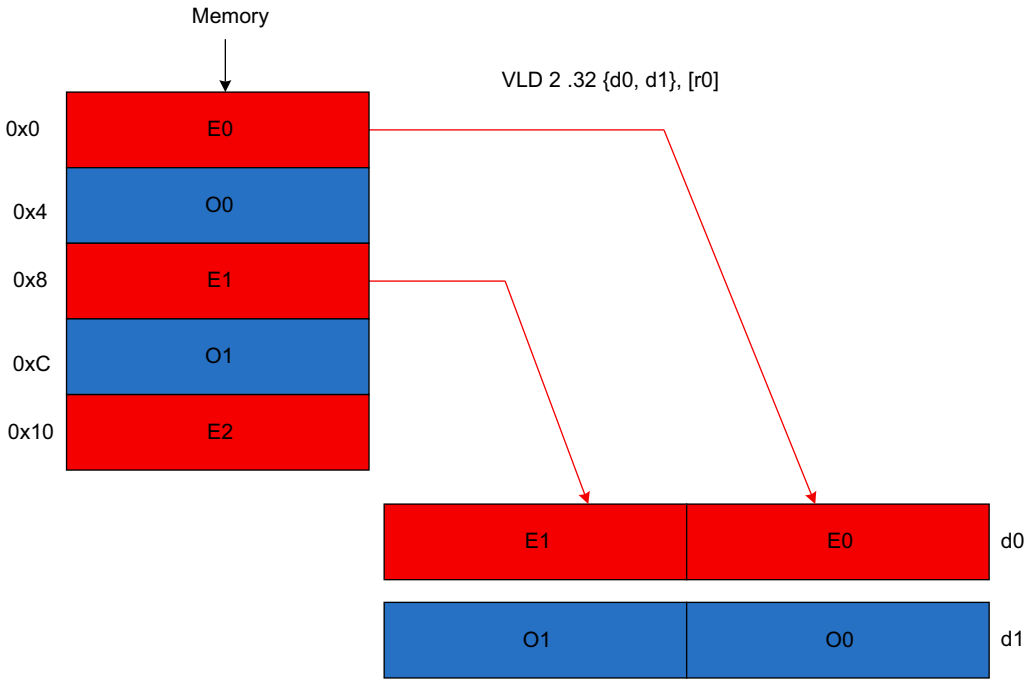

Figure 3-15 Loading and de-interleaving 32-bit data

Element size also affects endianness handling. In general, if you specify the correct element size in load and store instructions, bytes are read from memory in an appropriate order, and the same code works on little- and big-endian systems.

Finally, element size has an impact on pointer alignment. Alignment to the element size generally gives better performance. It might be a requirement of your target operating system. For example, when loading 32-bit elements, align the address of the first element to at least 32 bits.

Single or multiple elements

Structure load instructions de-interleave from memory. In each NEON register, they support the following loading methods:

- Load multiple lanes with different elements, as shown in Figure 3-12 on page 3-28.
- Load multiple lanes with the same element, as shown in Figure 3-16.
- Load a single lane with a single element and leave other lanes unaffected, as shown in Figure 3-17.

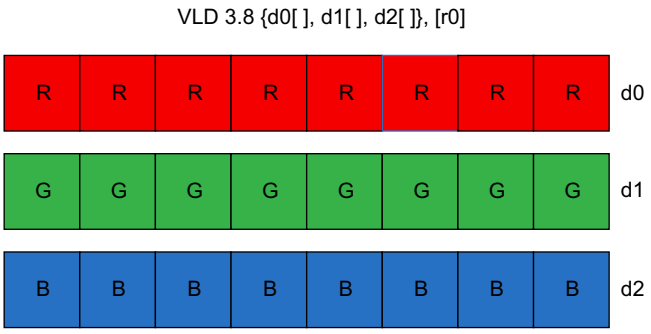

Figure 3-16 Loading and de-interleaving to all vector lanes

When you construct a vector from data scattered in memory, it is efficient to load a single lane with a single element and leave other lanes unaffected.

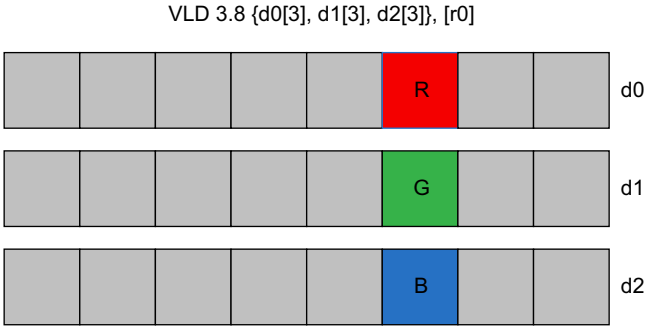

Figure 3-17 Loading and de-interleaving to a single vector lane

Stores are similar, and they provide support for writing single or multiple elements with interleaving.

Other loads and stores

In addition to structure loads and stores, NEON provides the following load and store instructions:

- `VLDR` – Loads a single register as a 64-bit value.
- `VSTR` – Stores a single register as a 64-bit value.
- `VLDM` – Loads multiple registers as 64-bit values.
- `VSTM` – Stores multiple registers as 64-bit values.

`VLDM` and `VSTM` are useful for storing and retrieving registers from the stack.

3.10 Virtualization

Virtualization is a concept whereby more than one Operating System is enabled to co-exist and operate on the same system. The ARM Virtualization Extensions make it possible to run multiple Operating Systems on the same system, while offering each Operating System an illusion of sole ownership.

ARMv7-A supports Virtualization extensions, but ARMv5 does not. 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. It might be necessary to separate software applications from their environments. This might be for reasons of robustness, or as a result of different requirements for real-time behavior, or it might simply be to isolate them from one another.

Hyp mode, with its associated banked registers, supports Virtualization extensions. Figure 3-18 shows that Hyp mode is added to existing privileged modes. This PL2 mode is even more privileged than PL1 mode. 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 Non-secure state.

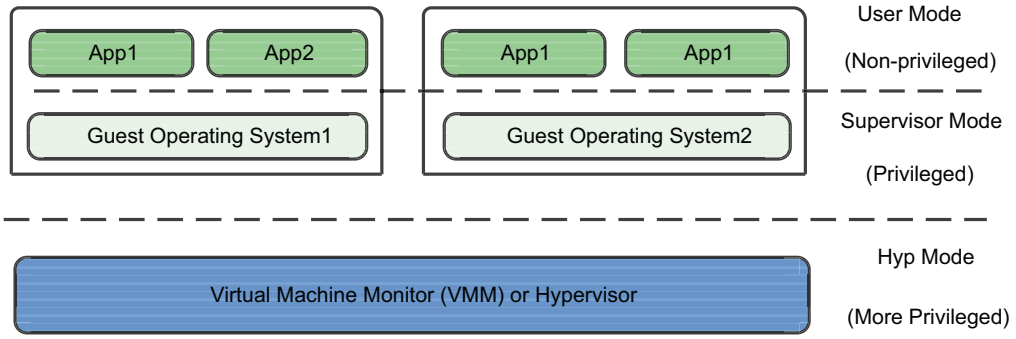

Figure 3-18 Hyp mode and privileged levels

Figure 3-19 on page 3-34 introduces an additional memory translation called Stage 2. Previously, the PL1 and PL0 MMU translated the *Virtual Address* (VA) to *Physical Address* (PA). This translation is now known as Stage 1, and the old Physical Address is now called *Intermediate Physical Address* (IPA). The IPA is subject to another level of translation in Stage 2 to obtain the final PA that corresponds to the VA.

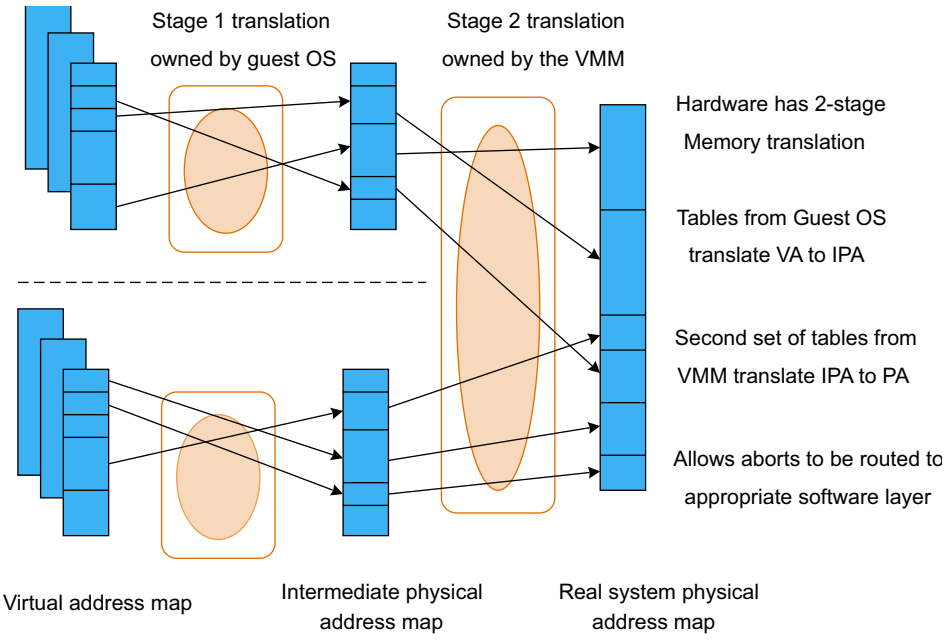

Figure 3-19 Two-stage memory translation

You can configure interrupts to be taken to the Hypervisor. The Hypervisor then handles delivery of interrupts to the appropriate guest.

You can use a `HVC` instruction 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.

3.10.1 Relationship between Virtualization and Security Extensions

Figure 3-20 shows how in the Non-secure state, you have PL0 for User mode, PL1 for exception modes, and PL2 for the Hypervisor, while in the Secure state, you have only PL0 and PL1, with Mon mode at PL1.

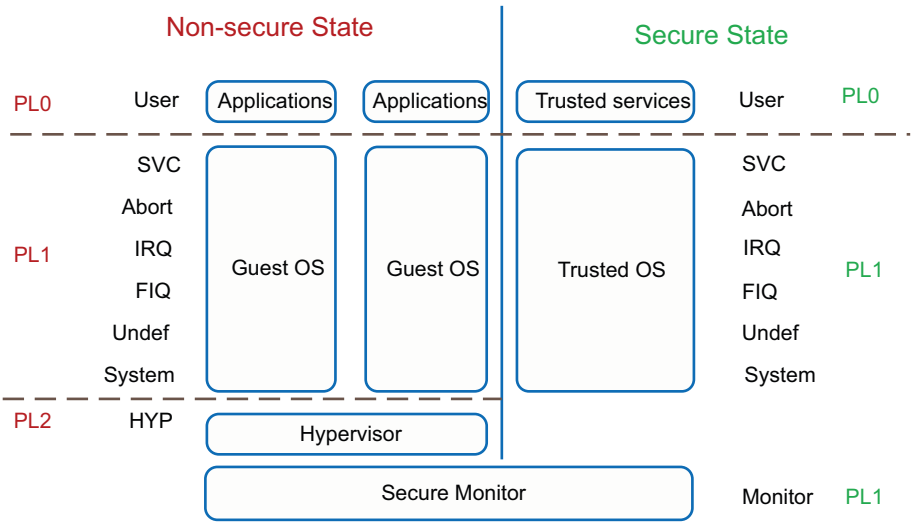

Figure 3-20 Privilege levels and security

Note

Secure Monitor can access Secure and Non-secure state. Otherwise, code executing in either Secure or Non-secure state cannot use system registers associated with the other security state. For example, a Hypervisor executing at PL2 cannot access any of the Secure system registers.

3.11 TrustZone®

ARMv7-A supports TrustZone, which is a hardware extension to enable construction of secure systems. TrustZone is not supported by ARMv5.

3.11.1 TrustZone hardware architecture overview

The TrustZone hardware architecture provides a security framework that enables a device to counter specific threats. Instead of providing a fixed one-size-fits-all security solution, TrustZone technology provides the infrastructure foundations that enable a *System on Chip* (SoC) designer to choose from a range of components that can fulfill specific functions within the security environment.

The primary security objective of the architecture is to enable the construction of a programmable environment that protects the confidentiality and integrity of assets from specific attacks. A platform with these characteristics is suited to building a wide-ranging set of security solutions that would not be cost-effective with traditional methods.

The security of the system is achieved by partitioning all of the SoC hardware and software resources so that they exist in one of two states:

- Secure state for the security subsystem.
- Non-secure state for everything else.

TrustZone enables a single physical processor core to execute code safely and efficiently from both the Non-secure state and the Secure state. This removes the need for a dedicated security processor core. Therefore, it saves silicon area and power, and enables high performance security software to run alongside the Non-secure state operating environment.

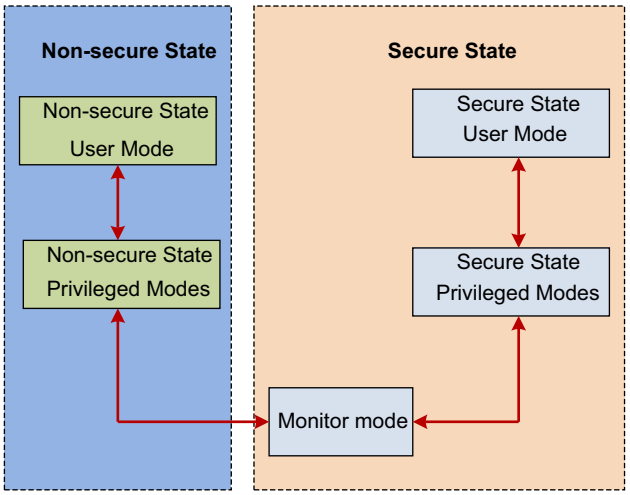

Figure 3-21 Non-secure and Secure states

Figure 3-21 shows the two virtual processors that perform context switches via a new processor mode called Mon mode, when changing the currently running virtual processor.

The mechanisms by which the physical processor can enter Mon mode from the Non-secure state are tightly controlled, and are all viewed as exceptions to the Mon mode software. Software can trigger an entry into Mon mode, by executing the `SMC` instruction, or by a subset of hardware exception mechanisms. Configuration of the IRQ, FIQ, external Data Abort, and external Prefetch Abort exceptions can cause the processor to switch into Mon mode.

The software that executes in Mon mode is implementation-defined. However, it saves the current state and restores the state at the location to which it switches. It then performs a return-from-exception to restart processing in the restored state.

Broad SoC security is achieved via the security state that TrustZone aware processors propagate into ARM® AMBA® AXI bus fabric. This ensures that Non-secure state components cannot access Secure state resources, and constructs a strong perimeter boundary between the two. If a design places sensitive resources in the Secure state and implements robust software running on the Secure processor cores, it can protect assets against many possible attacks. For example, it can protect the passwords you enter using a keyboard or touchscreen. Normally, these assets are hard to secure. By separating security sensitive peripherals through hardware, a designer can limit the number of subsystems that must go through security evaluation. Therefore, it saves costs when submitting a device for security certification.

The final aspect of the TrustZone hardware architecture is a security-aware debug infrastructure that can enable control over accesses to Secure state debug, without impairing debug visibility of the Non-secure state.

3.12 MP-core cache coherency

Cache coherency ensures that all processors or bus masters in the system have the same view of memory. It means that changes to data in the cache of one core are visible to other cores, making it impossible for cores to see stale copies of the data.

In ARMv7-A, there are three mechanisms to maintain cache coherency:

- Disable caching

  This is the simplest mechanism, but might cost significant core performance. To achieve the highest performance, processors are pipelined to run fast, and to run from caches that offer a low latency. Caching of data that is accessed multiple times increases performance significantly and reduces DRAM accesses and power. Marking data as *non-cached* would affect performance and power.

- Managed coherency through software

  Software-managed coherency is a traditional solution to the data sharing problem. 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. If there are high rates of sharing between requesters, the cost of software cache maintenance can be significant, and can limit performance.

- Managed coherency through hardware

  Hardware-managed coherency is the most efficient solution. Any data marked *shared* in a hardware coherent system is always up-to-date. All cores and bus masters in that sharing domain see the exact same value. Although hardware-managed coherency might add some complexity to the interconnect and clusters, it greatly simplifies the software and enables applications that would be impossible with software-managed coherency.

3.12.1 MESI and MOESI protocols

Cache coherency schemes operate in a number of standard ways. Most ARM processors use the *Modified Owner Exclusive Shared Invalid* (MOESI) protocol, while Cortex-A9 uses the *Modified Exclusive Shared Invalid* (MESI) protocol.

Based on the protocol in use, the *Snoop Control Unit* (SCU) marks each line in the cache with one of the following attributes:

- *Modified* (M) – Cache line has been modified. It is different from main memory, and is the only cached copy.
- *Owned* (O) – Cache line is dirty and is possibly in 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. Other cores can hold the data in the Shared state.
- *Exclusive* (E) – Cache line is the same as main memory and is the only cached copy.
- *Shared* (S) – Same as main memory but copies might exist in other caches.
- *Invalid* (I) – Line data is not valid as in simple cache.

Figure 3-22 on page 3-39 shows the use of the MESI protocol in memory and cache.

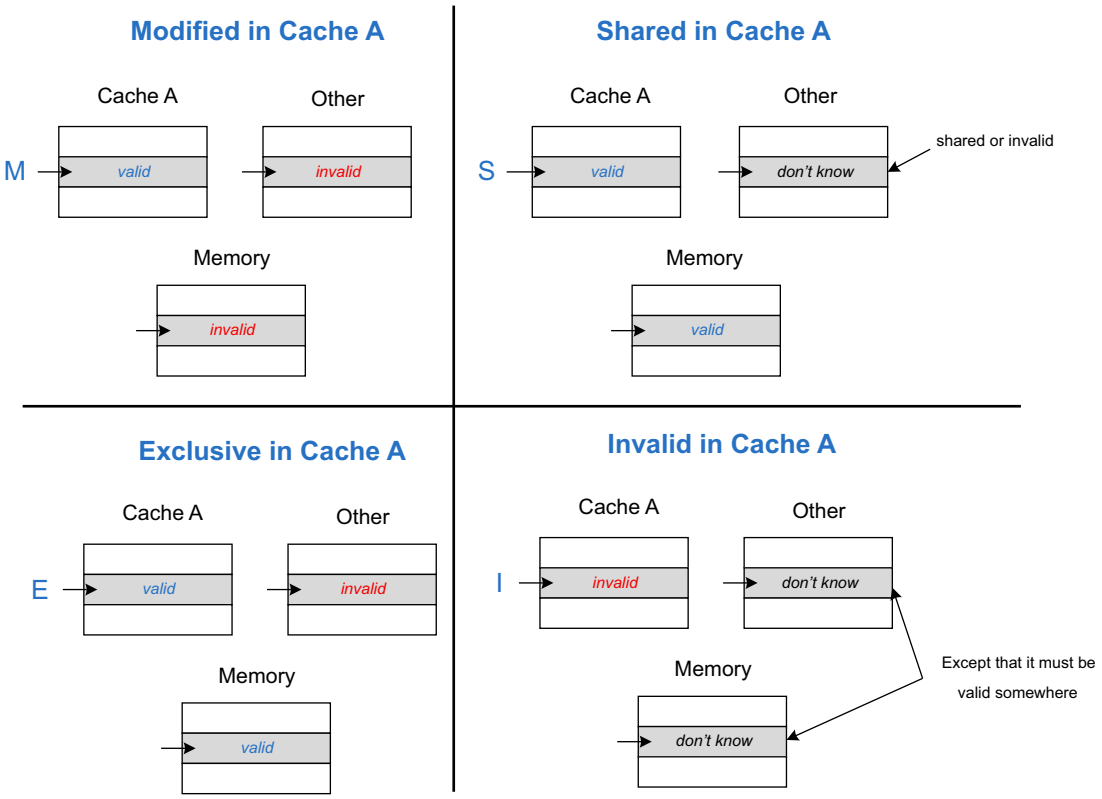

Figure 3-22 MESI state in memory and cache

3.12.2 Snoop Control Unit

Figure 3-23 shows how the *Snoop Control Unit* (SCU) maintains coherency among the L1 data cache of each core. The SCU arbitrates accesses to L2 AXI master interfaces, for both instructions and data. Duplicated Tag RAMs track what data is allocated in each CPU cache.

Because 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.

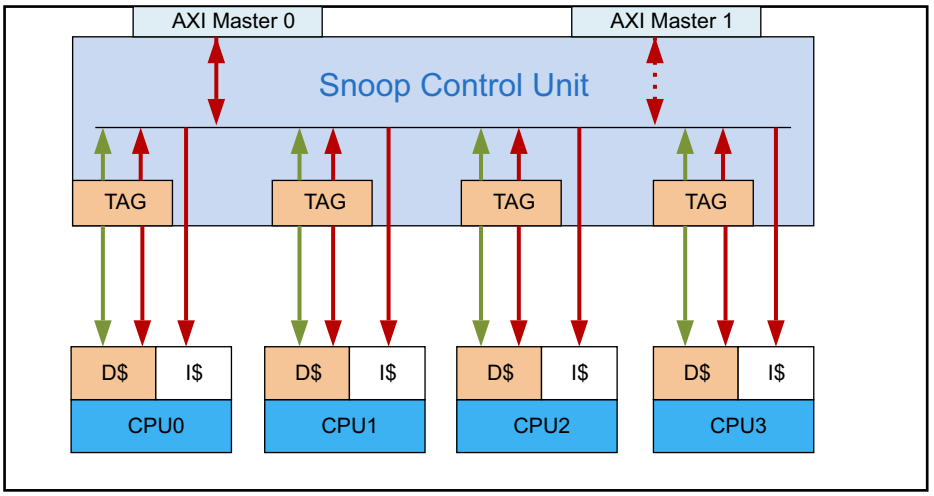

Figure 3-23 Snoop Control Unit

To enable coherency management for a memory access, all of the following conditions must be true:

- The SCU must be enabled through its control register in the private memory region. The SCU has configurable access control, restricting which processors can configure it.
- The core performing the access must be configured to participate in the Inner Shareable domain. You can configure it using the operating system at boot time, by setting the SMP bit in the CP15: ACTLR, Auxiliary Control Register.

  The following code example sets the SMP bit in either a Cortex-A7 or a 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 must be enabled.
- The page being accessed must be marked as Normal Shareable, with a cache policy of write-back, write-allocate. Device and Strongly-ordered memory, however, is not cacheable, and write-through caches behave like uncached memory from the perspective of the core.

The SCU can only maintain coherency within a single cluster.

3.12.3 Cache Coherent Interface

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 AXI protocol. The full ACE interface enables hardware coherency among clusters and enables an SMP operating system to extend to more cores.

The ACE protocol adds three coherency channels 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 coherent system masters such as 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 processor data caches.

The CoreLink™ CCI-400 is one of the first implementations of AMBA 4 ACE. It supports up to two ACE processor clusters, enabling up to eight cores to see the same view of memory and run an SMP OS.

Figure 3-24 on page 3-41 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 the Cortex-A15 cluster cache.

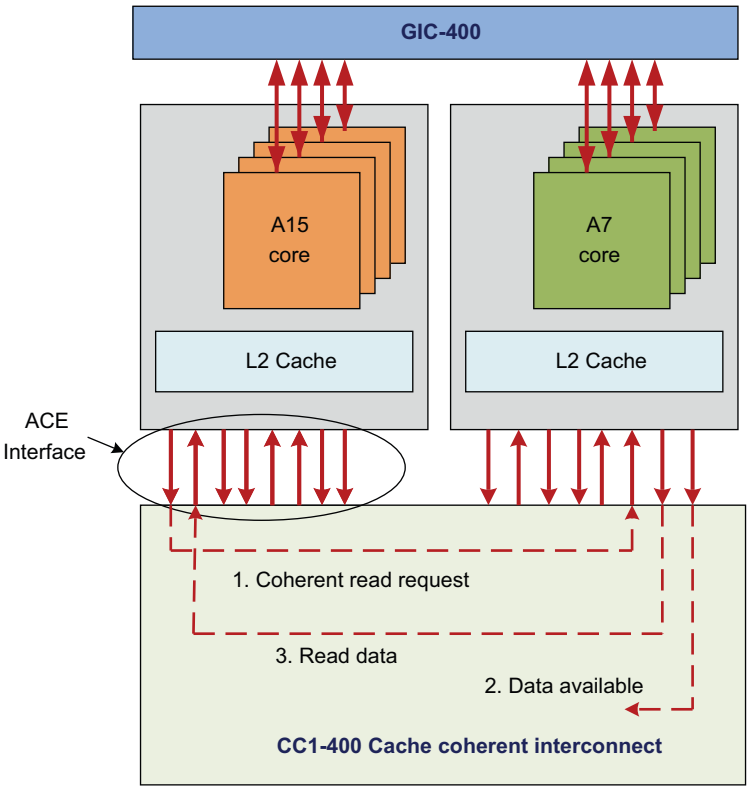

Figure 3-24 Cache coherency in a multi-cluster system

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.

Chapter 4
Migrating a software application from ARMv5 to ARMv7-A/R

Read this chapter for detailed information about how to migrate a software application from ARMv5 to ARMv7.

To perform the migration, take the following steps:

1. *Changing startup code and set up MMU cache* on page 4-2.
2. *Modifying exception-handling code* on page 4-5.
3. *Replacing ARMv5 barriers with equivalent ARMv7 barriers* on page 4-8.
4. *Replacing ARMv5 synchronization primitives with equivalent ARMv7 synchronization primitives* on page 4-11.
5. *[Optional] Implementing TrustZone to provide a robust security solution* on page 4-13.
6. *[Optional] Using NEON to improve application performance* on page 4-16.
7. *[Optional] Using Symmetric Multi-Processing to deliver higher performance* on page 4-23.
8. *Choosing the right software development tools and debug adaptors* on page 4-33.
9. *[Optional] Enabling FPU to improve application performance* on page 4-36.

4.1 Changing startup code and set up MMU cache

You must change your startup code and set up MMU Cache in ARMv7-A.

4.1.1 V7 Cache and MMU setup code

Example 4-1 shows you how to set up the caches, the MMU, and branch predictors. You can 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 or the L2 caches, or both, at reset, but others require manual invalidation. Check the *Technical Reference Manual* (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, and it can improve performance.

Example 4-1 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, 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 the 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.
```

The following table shows the code you must use to create your translation tables. Use the variable `ttb_address` to denote the address 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.

Example 4-2 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.

    @ 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 first entry in the table, You can 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, do not 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.
```

4.1.2 Memory type mapping

In ARMv7, the memory attributes are different from those in ARMv5. Table 4-1 shows the memory type mapping between ARMv7 and ARMv5.

Table 4-1 Memory type mapping between ARMv7 and ARMv5

ARMv7 attributes	ARMv5 attributes
Strongly-ordered	Non-cacheable, Non-bufferable (NCNB)
Device	Non-cacheable, Bufferable (NCB)
Non-shareable Normal, Write-Through Cacheable	Write-Through Cacheable, Bufferable
Non-shareable Normal, Write-Back Cacheable	Write-Back Cacheable, Bufferable

4.2 Modifying exception-handling code

The approach to handle exception is changed in ARMv7, so you must modify exception-handling code accordingly. Exception handling in ARMv7 differs from that in ARMv5 in the following areas.

- *Vector Tables*
- *Exception state and endianness* on page 4-6
- *Exception return* on page 4-6
- *Abort exception handling* on page 4-6
- *Unaligned access support* on page 4-6
- *Fault handling* on page 4-7

4.2.1 Vector Tables

ARMv5 systems have a single vector table. It is a table of instructions that the ARM core jumps to when an exception is raised.

In ARMv5, there is only one vector table. In ARMv7-A, however, there can be up to four vector tables, as shown in Figure 4-1.

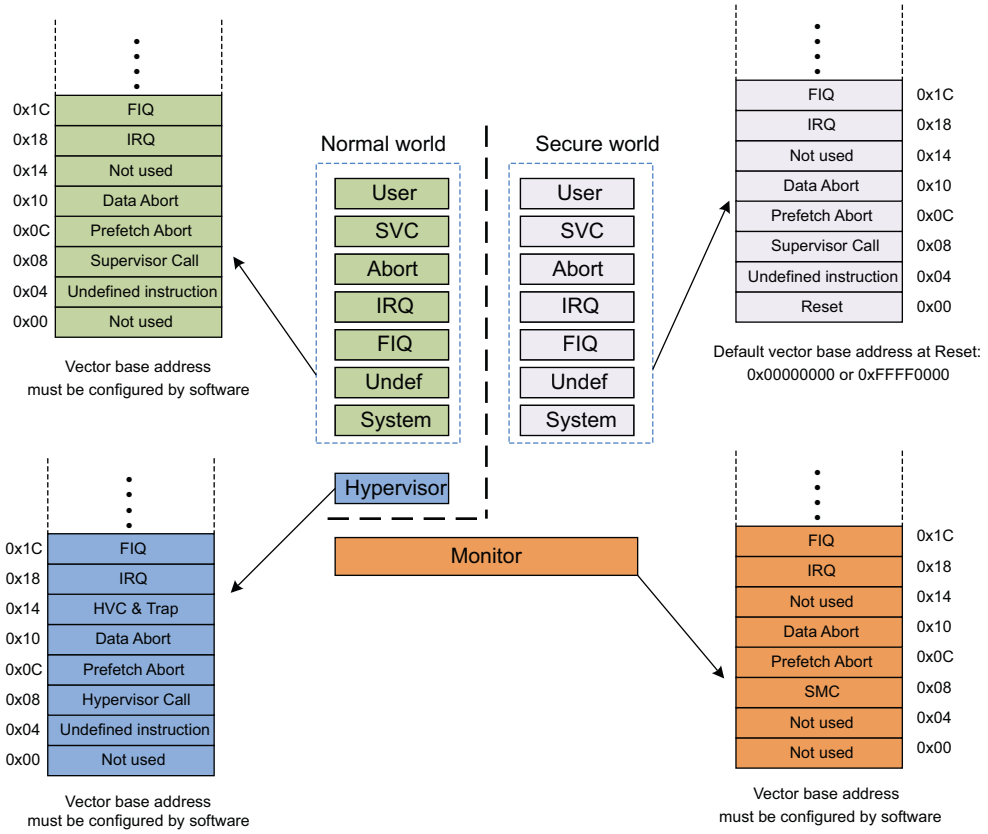

Figure 4-1 Vector tables in ARMv7

In ARMv5, a vector table can only be at `0xFFFF0000` or `0x00000000`. In ARMv7-A, the vector table in Secure state, not the one owned by Mon mode, has the default vector base address at reset: `0xFFFF0000` or `0x00000000`. However, this base address can be relocated to other addresses by software. The other three vector tables do not have a defined location at reset time, so the boot code must initialize them. You can use software to configure their vector base addresses.

4.2.2 Exception state and endianness

In ARMv5, when an exception is entered, the first instruction to execute in the vector table must be an ARM instruction.

In ARMv7, however, when an exception is entered, the first instruction to execute in the vector table can be an ARM instruction or a Thumb-2 instruction. You can use the SCTLR.TE bit to control whether exceptions are taken in ARM or Thumb state.

In ARMv5, when an exception is entered, the exception is handled in whatever endian state the processor was already in. This means that the whole system must use the same endianness.

In ARMv7, however, when an exception is entered, a piece of code can use different endianness from the exception handlers. You can use the SCTLR.EE bit to control exception endianness. The SCTLR.EE bit defines the value of the CPSR.E bit on entry to an exception.

4.2.3 Exception return

ARMv5 uses the following instructions to return from exceptions:

- Data processing instructions, for example, `MOVS PC, LR`.
- LDM instructions, for example, `LDM SP!, {…, PC}^`.

ARMv7 still uses these instructions to return from exceptions. In addition, ARMv7 introduces the following new instructions to return from exceptions:

- `SRS`.
- `RFE`.

4.2.4 Abort exception handling

Aborts can be classified into *precise aborts* and *imprecise aborts*:

Precise aborts

A precise abort, also known as a *synchronous abort*, is one for which the exception is guaranteed to be taken on the instruction that generated the aborting memory access. The abort handler can use the value in the Link Register (r14_abt) to determine which instruction generated the abort, and the value in the Saved Program Status Register (SPSR_abt) to determine the state of the processor when the abort occurred.

Imprecise aborts

An imprecise abort, also known as an *asynchronous abort*, is one for which the exception is taken on a later instruction to the instruction that generated the aborting memory access. The abort handler cannot determine which instruction generated the abort, or the state of the processor when the abort occurred. Therefore, imprecise aborts are normally fatal.

ARMv5 cannot differentiate precise aborts and imprecise aborts. In some ARMv5 systems, such as ARM926, imprecise aborts are completely ignored by the CPU.

However, ARMv7 can detect whether an abort is precise or imprecise. In ARMv7, you can use the CPSR.A bit to mask an imprecise abort.

4.2.5 Unaligned access support

In ARMv5, unaligned memory accesses can cause an alignment exception.

ARMv7, however, introduces hardware support for unaligned accesses by some load and store instructions. You can use the SCTLR.A bit to control whether alignment checking is enabled or disabled. If enabled, all unaligned memory accesses cause an alignment exception. If disabled, unaligned accesses are permitted for the following instructions:

- `LDR`, `LDRH`, `LDRSH`, `LDRT`, `LDRSHT`, `LDRHT`.
- `STR`, `STRH`, `STRT`, `STRHT`.
- `TBH`.

4.2.6 Fault handling

In ARMv5, the registers *Fault Status Register* (FSR) and *Fault Address Register* (FAR) investigate the cause of aborts resulting from data accesses, known as *Data Aborts*. It is implementation-defined whether the FSR and FAR are updated for an abort arising from an instruction fetch, and if so, what useful information they contain about the fault.

Aborts are further classified into *data aborts* and *prefetch aborts*. ARMv7 can differentiate whether an abort is generated either on failed instruction fetches, known as *prefetch abort*s, or on failed data accesses, known as *data aborts*. Therefore, the following registers are provided to indicate the cause of aborts:

DFAR The DFAR holds the virtual address of the faulting address that caused a synchronous Data Abort exception.

DFSR The DFSR holds status information about the last data fault.

IFAR The IFAR holds the address of the access that caused a synchronous Prefetch Abort exception.

IFSR The IFSR holds status information about the last instruction fault.

The encodings of FAR and FSR in ARMv5 are different from those of DFAR and DFSR, or IFAR and IFSR, in ARMv7.

For more information about the encodings of DFAR, FDSR, IFAR, and IFSR, see the VMSA System control registers section in the *ARM® Architecture Reference Manual ARMv7-A and ARMv7-R Edition, ARM DDI 0406C*.

4.3 Replacing ARMv5 barriers with equivalent ARMv7 barriers

In ARMv7, the `IMB` barrier is deprecated, and you must replace it with equivalent ARMv7 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 also refers to a compiler mechanism that prevents the compiler from scheduling data access instructions across the barrier when performing optimizations. In GCC, for example, you can use the inline assembler memory clobber, to indicate that the instruction changes memory and therefore the optimizer cannot reorder memory accesses across the barrier. The syntax is as follows:

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

ARM Compiler, armcc, includes a similar intrinsic, called `__schedule_barrier()`.

However, this document focuses on hardware memory barriers, provided through dedicated ARM assembly language instructions. Core optimizations can result in memory operations occurring in a different order from that specified in the executing code.

Normally, this reordering is invisible to you, and you do not have to worry about memory barriers. However, there are cases where you must take care of such ordering issues. For example, you must consider these issues in device drivers or when you have multiple observers of the data that must be synchronized.

The ARM architecture provides 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.

***Data Synchronization Barrier* (DSB)**

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

***Data Memory Barrier* (DMB)**

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

***Instruction Synchronization Barrier* (ISB)**

This instruction ensures that the effects of all context-altering operations preceding the ISB are recognized by subsequent instructions. This results in a flushing of the instruction pipeline, with the instruction following the ISB being refetched.

To provide the type of access and the shareability domain it applies to, the following options can be specified with the `DMB` or `DSB` instructions.

SY The barrier applies to the full system, including all cores and peripherals. This is the default.

ST The barrier only waits for stores to complete.

ISH The barrier only applies to the Inner Shareable domain.

ISHST The barrier combines ST and ISH. That is, it only stores to the Inner Shareable domain.

NSH The barrier only applies to the *Point of Unification* (PoU).

NSHST The barrier only waits for stores to complete and only out to the point of unification.

OSH The barrier operation only applies to the Outer Shareable domain.

OSHST The barrier operation only waits for stores to complete, and only to the Outer Shareable domain.

The `DMB` instruction enforces memory access ordering within a shareable domain. All processors within the shareable domain are guaranteed to observe all explicit memory accesses preceding the `DMB` instruction, before they observe any of the explicit memory accesses following the `DMB` instruction.

The `DSB` instruction has the same effect as the `DMB`. In addition, 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 stops until all outstanding explicit memory accesses are completed. When all outstanding reads have completed and the write buffer is drained, execution resumes as normal.

It might be easier to understand 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 ensures that all data memory accesses in the program order preceding the barrier complete, before any explicit memory accesses that appear in the program order after the barrier. This barrier ensures that all cores within the cluster 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 cannot be guaranteed for external observers, such as DMA controllers or DSPs.

4.3.1 Example of using barriers

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 4-3:

Example 4-3 Code example showing memory ordering issues

Core A:

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

Core B:

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

There is no ordering requirement. The transactions might occur in different orders. The addresses `Addr1` and `Addr2` are independent. 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 four possible legal outcomes of this piece of code:

- 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 involve a third core C, there is no requirement that it observes either of the stores in the same order as either of the other cores. It is possible 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 set by A and then reads data from A. You might have code similar to that in Example 4-4:

Example 4-4 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.
```

This program might not behave in an expected way. It is possible that core B performs the read from `[Msg]` before the read from `[Flag]`. This is normal for weakly-ordered memory. 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 ensure that they occur in the order you originally specified.

Core B requires a `DMB` before the `LDR R0, [Msg]`, to ensure that the message is read-only after the flag is set.

You can rewrite the code as Example 4-5.

Example 4-5 Using barriers to ensure the correct order for postbox

Core A:

```
STR R0, [Msg]       @ write some new data into postbox
DMB
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
      DMB
      LDR R0, [Msg] @ read new data.
```

4.4 Replacing ARMv5 synchronization primitives with equivalent ARMv7 synchronization primitives

In ARMv7, the SWP and SWPB instructions are deprecated, you must replace them with equivalent ARMv7 synchronization primitives in source code.

ARMv7 provides the following instructions relating to exclusive access. Variants of these instructions are also provided to operate on byte, halfword, word, or doubleword sized data. The instructions rely on the ability of the core or memory system to tag particular addresses to be monitored for exclusive accesses by that core, using an exclusive access monitor.

- LDREX performs a load of memory, but also tags the physical address to be monitored for exclusive access by that core. For example, LDREX R1, [R0] performs a Load-Exclusive from the address in R0, places the value into R1, and updates the exclusive monitors.
- STREX performs a conditional store to memory. The store succeeds 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. For example, STREX R2, R1, [R0] performs a Store-Exclusive operation to the address in R0, conditionally storing the value from R1 and indicating success or failure in R2.
- CLREX clears any exclusive access tag for that core.

Only perform Load-Exclusive and Store-Exclusive operations on Normal memory. The operations have slightly different effects, depending on whether the memory is marked as Shareable. If the core reads from Shareable memory with an LDREX, the load occurs 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, any attempt to write to the tagged address by the one that tagged it causes the tag to be cleared. If the core performs 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 that 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 used in single-core systems, to implement synchronization operations among 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. However, it can do so if the processor is implemented 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 you use exclusive accesses to synchronize with external masters outside the core, or to regions marked as Shareable even between cores in the same cluster, you must 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 and STREX sequence performed to a memory location that has no suitable exclusive access monitor fails, with the STREX instruction always returning 1.

Example 4-6 shows the use of the LDREX and STREX instructions.

Example 4-6 Spin-lock

```
    ; void lock(lock_t* pAddr)
lock
    ; Is locked?
    LDREX   r1, [r0]            ; Check if locked.
    CMP     r1, #LOCKED         ; Compare with “locked“.
    BEQ     lock                ; If LOCKED, try again.

    ; Attempt to lock
    MOV     r1, #LOCKED
    STREX   r2, r1, [r0]        ; Attempt to lock.
    CMP     r2, #0x0            ; Check whether store completed.
    BNE     lock                ; If store failed, try again.
    DMB
    BX      lr
```

4.5 [Optional] Implementing TrustZone to provide a robust security solution

In ARMv7-A, there are many possible software architectures that a Secure state software stack on a TrustZone-enabled processor core can implement. The most complex is a dedicated Secure state operating system; the simplest is a synchronous library of code placed in a Secure state. There are many intermediate options between these two extremes.

4.5.1 Secure operating system

A dedicated operating system in a Secure state is a complex, but powerful, design. It can simulate concurrent execution of multiple independent Secure state applications, run-time download of new security applications, and Secure state tasks that are completely independent of a Non-secure state environment.

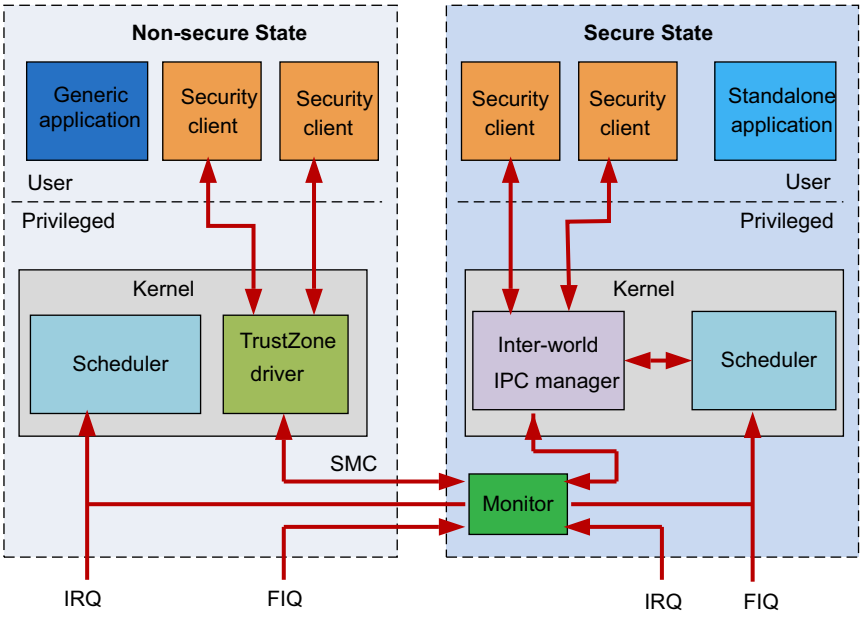

Figure 4-2 A possible architecture with an independent Secure state OS

One advantage of a design based on operating system principles, is the use of the processor MMU to separate a Secure state memory space into multiple user-space sandboxes. If the Secure state kernel software is correctly implemented, Secure tasks from independent stakeholders can execute at the same time, without needing to trust each other. The kernel design can enforce the logical isolation of Secure tasks from each other, and prevents one Secure task from tampering with the memory space of another.

4.5.2 Synchronous library

Many use cases do not need the complexity of a Secure state operating system. A simple library of code in a Secure state can handle one task at a time, and is sufficient for many applications. This code library is entirely scheduled and managed using software calls from the Non-secure state operating system.

The Secure state in these systems is a slave to the Non-secure state and cannot operate independently. However, it can have a much lower level of complexity.

4.5.3 Intermediate options

There is a range of options between these two extremes. For example, you can design a Secure state multi-tasking operating system to have no dedicated interrupt source, and you can provide it with a virtual interrupt by a Non-secure state. This design is vulnerable to a denial-of-service attack if the Non-secure state operating system stops providing the virtual interrupt. Alternatively, you can use the MMU to statically separate different components of an otherwise synchronous Secure state library.

4.5.4 Booting a secure system

A TrustZone-enabled processor starts in the Secure state when you power on the system. This enables any sensitive security checks to run before the Non-secure state software has an opportunity to modify any aspect of the system.

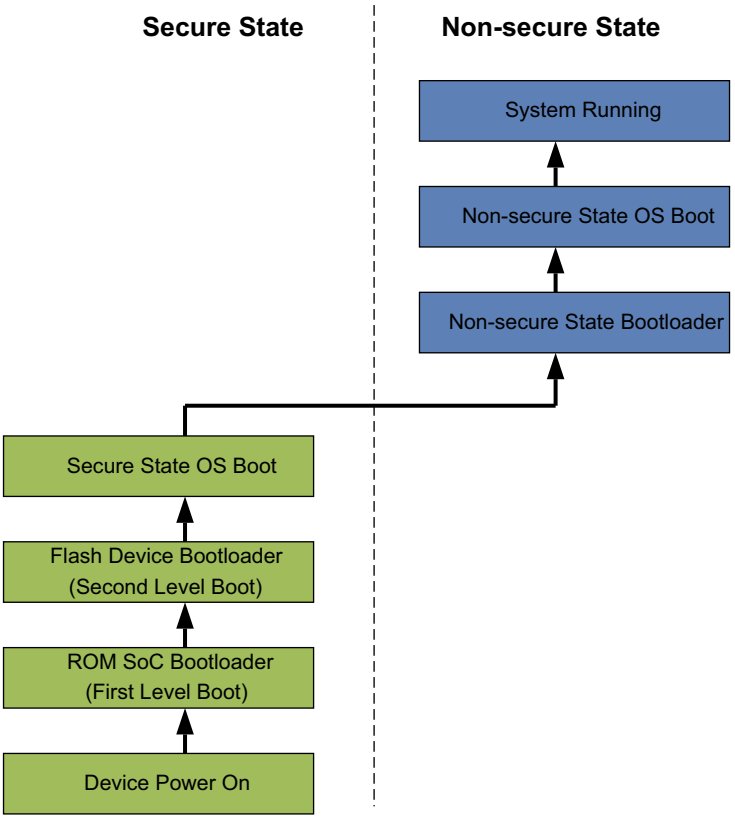

Figure 4-3 A typical boot sequence of a TrustZone-enabled processor

After power-on, most SoC designs start executing a ROM-based bootloader before switching to a device bootloader located in external non-volatile storage, such as flash memory. A ROM-based bootloader initializes critical peripherals, such as memory controllers. The boot sequence then progresses through the Secure state operating environment initialization stages, before passing control to the Non-secure state bootloader. This starts the Non-secure state operating system, at which point the system can be considered running.

4.5.5 Booting Linux in a Non-secure state

Booting Linux in a Non-secure state requires a number of additional configuration steps. In a system, the boot code performs these steps in a Secure state.

To boot Linux in a Non-secure state, you must configure the system to allow Non-secure (NS) access to the following peripherals and memory used by Linux:

- *Interrupt controller*.
- *Memory Map*.
- *CPU Specifics*.

Interrupt controller

If you are using a GIC, you must configure it while still in a Secure state.

- Interrupt Security Register, ICDISR{n}, used to set which interrupt IDs are available in a Non-secure state.
- Interrupt Priority Mask Register, ICCPMR, shared between a Non-secure and Secure state. Before the Non-secure state can access the register, the Secure state must write a value greater than `0x80` to it.

  For more information, refer to section 4.2.1 *Non-secure access to register fields for Secure interrupt priorities* of the *ARM® Generic Interrupt Controller Architecture Specification (ARM IHI 0048A)*.

Memory Map

Most TrustZone-enabled systems include a *TrustZone Protection Controller* (TZPC) or something similar. You can use it to set which address regions are accessible in the Non-secure state. It must be configured before entering the Non-secure state.

The *Real-Time System Models* (RTSMs) do not include a TZPC, so this step can be skipped.

CPU Specifics

CP15 includes three TrustZone configuration registers:

- *Non-secure Access Control Register* (NSACR).

  This register controls Non-secure state access to coprocessors such as VFP and NEON, and TLB lock down. On the Cortex-A8, it also controls access to the PLE and L2 cache lock down. On Cortex-A5 and Cortex-A9, it controls access to the ACTLR.SMP bit.

  To boot Linux, you must enable access to at least the VFP and NEON if Linux is built to use them.

- *Secure Debug Enable Register* (SDER).

  This register controls debug access. You do not have to change the default value to boot Linux on the RTSMs.

- *Secure Configuration Register* (SCR).

  This register controls exception behavior and which state the core is currently in. The NS bit must be set to switch the core into the Non-secure state. This step must be performed when you have completed all the other configuration steps.

The core is now in a Non-secure state, with the GIC configured to allow Non-secure state interrupts.

4.6 [Optional] Using NEON to improve application performance

If your code contains multimedia and signal processing algorithms, such as video encode or decode, 2D or 3D graphics, gaming, audio and speech processing, image processing, telephony, or sound synthesis, you can use NEON to improve application performance.

Code that targets NEON hardware can be written in C, C++, or assembly language. There is a range of tools and libraries available to support NEON. The following sections show examples of how you can use NEON to improve application performance.

4.6.1 Enabling NEON

The NEON data engine is disabled at reset. You must enable NEON in software before use. Complete the following steps to enable NEON:

- Enable access to coprocessors 10 and 11 and permit NEON instructions:

```
MRC   p15, 0x0, r0, c1, c0, 2   ; Read CP15 CPACR
ORR   r0, r0, #(0x0f << 20)     ; Full access rights
MCR   p15, 0x0, r0, c1, c0, 2   ; Write CP15 CPACR
```

- Enable NEON and VFP

```
MOV r0, #0x40000000             ; set bit 30
VMSR FPEXC, r0                  ; write r0 to Floating Point Exception Register
```

4.6.2 Assembler

You can program the NEON unit directly in Assembly language. The consistent design of the NEON instruction set makes this less complex than you might expect. The *Gnu's Not Unix* (GNU) and ARM assemblers use the same instruction format, but other syntax differs.

Example 4-7 Simple NEON assembler example for GAS

```
    .text
    .arm
    .global double_elements
double_elements:
    vadd.i32 q0,q0,q0
    bx lr
    .end
```

To assemble the code in Example 4-7 using GNU Assembler, add `-mfpu=neon` to the assembler command line. It specifies that NEON instructions are permitted. For example:

```
arm-none-linux-gnueabi-as -mfpu=neon asm.s
```

Example 4-8 Simple NEON assembler example for ARM Compiler

```
    AREA RO, CODE, READONLY
    ARM
    EXPORT double_elements
double_elements

VADD.I32 Q0, Q0, Q0
BX LR
```

```
END
```

To assemble the code in Example 4-8 using ARM Compiler, you must specify a target processor that supports NEON instructions. For example:

```
armasm --cpu=Cortex-A8 asm.s
```

4.6.3 Compiler intrinsics

Compiler intrinsics provide similar functionality to inline assembly, and additional features such as type checking and automatic register allocation. An intrinsic function appears as a function call in C or C++, but is replaced during compilation by a sequence of low-level instructions. You can express low-level architectural behavior in a high-level language.

In addition to giving direct access to instructions that do not normally map well onto high-level language statements, using intrinsics means that the compiler can optimize the operation to improve performance. When using intrinsics, you do not have to consider register allocation and interlock issues. The compiler handles these issues.

GCC and ARM compilers support the same NEON intrinsic syntax, making C or C++ code portable between the toolchains. To add support for NEON intrinsics, include the header file `arm_neon.h`.

The header file `arm_neon.h` defines a set of vector data types of different sizes. For example:

- `int16x4_t` – vector of four 16-bit short int values (held in a D register).
- `float32x4_t` – vector of four 32-bit float values (held in a Q register).

The intrinsics in Example 4-9 are equivalent to the NEON instruction `VADD.I32 result, a, b`.

Example 4-9 Intrinsics basic usage

```
int32x2_t result, a, b;
result = vadd_s32(a,b)
```

4.6.4 Automatic vectorization

ARM Compiler and GCC can also perform automatic vectorization on C or C++ source code. This gives access to high NEON performance, without writing assembly code or using intrinsics. In this way, source code remains portable among different tools and target platforms.

Because the C language does not specify parallelizing behavior, you can indicate to the compiler where it is safe and optimal. You can do this without compromising the portability of the source code among different platforms or toolchains.

Example 4-10 shows a small function that the compiler can safely and optimally vectorize. This is possible if you use the `__restrict` keyword to ensure that the pointers `pa` and `pb` do not address overlapping regions of memory. You can also force the `for` loop to always execute a multiple of four times by masking off the bottom two bits of `n` for the limit test. This extra information makes it safe for the compiler to vectorize this function into NEON load and store operations.

Example 4-10 NEON vectorization

```
void add_ints(int * __restrict pa, int * __restrict pb, unsigned int n, int x)
{
```

```
    unsigned int i;
    for(i = 0; i < (n & ~3); i++)
     pa[i] = pb[i] + x;
}
```

Compiling the example

Although ARM Compiler and GNU development tools support the same source syntax, the command-line syntax differs significantly between the two compilers.

Automatic vectorization with ARM Compiler

To enable automatic vectorization, specify a target processor that includes NEON technology, compile for optimization level `-O2` or higher, and add `-Otime` and `--vectorize` to the command line. For example:

```
armcc --cpu=Cortex-A9 -O3 -Otime --vectorize -c vectorized.c
```

Note

When you specify `--vectorize`, you must also specify `-Otime` and an optimization level of `-O2` or `-O3` to enable automatic vectorization.

Because parallel accumulations of floating-point values can reduce the precision gained by sorting input data, these are disabled unless you specify `--fpmode=fast` on the command line.

You can request more verbose compiler output by adding `--remarks` to the command line. This provides additional information about many aspects of the compilation taking place. For NEON vectorization, this includes the following information:

- Code that the compiler has vectorized.
- Code that could not be vectorized, and hints of why this was not done.

This information can be used to modify the code into a format that the compiler can vectorize.

Automatic vectorization with GCC

To enable automatic vectorization, you must add `-mfpu=neon` and `-ftree-vectorize` to the GCC command line. For example:

```
arm-none-linux-gnueabi-gcc -mfpu=neon -ftree-vectorize -c vectorized.c
```

Depending on your toolchain, you might also have to add `-mfloat-abi=softfp` to indicate that NEON variables must be passed in general-purpose registers.

You can request more verbose compiler output by adding `-ftree-vectorizer-verbose=1` to the command line. This gives the following compiler output:

- Code that it has vectorized.
- Code that it could not vectorize, and hints of why this was not done.

You can use this information to modify the code into a format that the compiler can vectorize. Some versions of GCC support verbosity values higher than 1, providing even more details about vectorization.

4.6.5 Matrix multiplication example

You can use NEON to improve the performance of matrix multiplication.

What is matrix multiplication

In mathematics, matrix multiplication is a binary operation that takes a pair of matrices, and produces another matrix. Mathematically, if C is a matrix resulting from the multiplication of two matrices, A and B, then the elements cij of C are given by the following equation:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

Where k = 1, 2,…, n is the number of columns in A and the number of rows in B.

In other words, you multiply each of the elements of a row in the left-hand matrix by the corresponding elements of a column in the right-hand matrix, and then sum the resulting n products to obtain one element in the product matrix.

Matrix multiplication implementation using the C code

Suppose you want to calculate the product matrix C of multiplying the following two matrices A and B:

$$A = \begin{bmatrix} a11 & a12 & a13 & a14 \\ a21 & a22 & a23 & a24 \\ a31 & a32 & a33 & a34 \\ a41 & a42 & a43 & a44 \end{bmatrix} \qquad B = \begin{bmatrix} b11 & b12 & b13 & b14 \\ b21 & b22 & b23 & b24 \\ b31 & b32 & b33 & b34 \\ b41 & b42 & b43 & b44 \end{bmatrix}$$

To use the C code to calculate the product matrix, you can store matrix elements into arrays, and use loops to calculate each element of the product matrix. The C code implementation can be as follows:

```
f32 *a, *b, *c;
f32 sum;

for (i = 0; i < 4; i++)
{
    for (k = 0; k < 4; k++)
    {
        sum = 0.0f;

        for (j = 0; j < 4; j++)
        {
            sum += a[i*4+j] * b[j*4 + k]
        }

        // Product matrix elements are calculated and stored into array c.
        c[i*4 + k] = sum;

    }
}
```

However, the C code implementation is not efficient.

Matrix multiplication implementation using NEON

To achieve improved performance, you can use NEON instructions to multiply the four-by-four matrices. Assume that the matrices are stored in memory in column-major order. To identify suboperations that can be implemented using NEON instructions, expand the matrix multiply operation in detail, as shown in Figure 4-4 on page 4-20.

$$\begin{bmatrix} a11 & a12 & a13 & a14 \\ a21 & a22 & a23 & a24 \\ a31 & a32 & a33 & a34 \\ a41 & a42 & a43 & a44 \end{bmatrix} \bullet \begin{bmatrix} b11 & b12 & b13 & b14 \\ b21 & b22 & b23 & b24 \\ b31 & b32 & b33 & b34 \\ b41 & b42 & b43 & b44 \end{bmatrix} = \begin{bmatrix} a11 \bullet b11 + a12 \bullet b21 + a13 \bullet b31 + a14 \bullet b41 & \cdots & \cdots & \cdots \\ a21 \bullet b11 + a22 \bullet b21 + a23 \bullet b31 + a24 \bullet b41 & \cdots & \cdots & \cdots \\ a31 \bullet b11 + a32 \bullet b21 + a33 \bullet b31 + a34 \bullet b41 & \cdots & \cdots & \cdots \\ a41 \bullet b11 + a42 \bullet b21 + a43 \bullet b31 + a44 \bullet b41 & \cdots & \cdots & \cdots \end{bmatrix}$$

Figure 4-4 Matrix multiplication showing one column of results

In Figure 4-4, each column of the first matrix in red is multiplied by a corresponding single value in the second matrix in blue. Then, the individual products are added to give a column of values for the results matrix. This operation is repeated for the remaining columns in the results matrix.

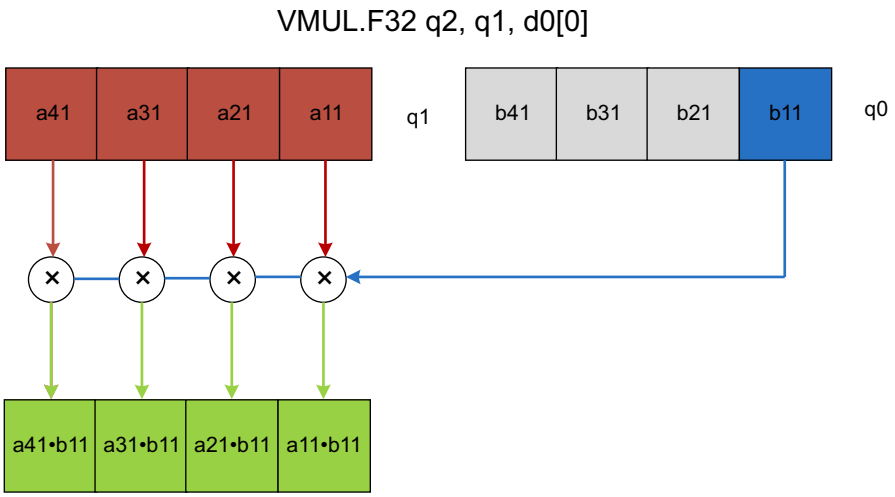

Figure 4-5 NEON vector-by-scalar multiplication

If each column is a vector in a NEON register, the vector-by-vector multiply instruction efficiently calculates each column in the results matrix. You can implement the suboperation highlighted in Figure 4-5 by using this instruction. Use the accumulating version of the multiply instruction to accumulate the individual products from each column of the first matrix.

This example operates on the columns of the first matrix, and produces a column of results. Therefore, reading and writing elements to and from memory is a linear operation, and does not require interleaving load or store instructions.

Suppose this is an implementation that multiplies single precision floating-point matrices. Each matrix contains sixteen 32-bit floating-point values.

You can begin by loading the matrices from memory into NEON registers. The matrices are stored in column-major order, so columns of the matrix are stored linearly in memory. A column can be loaded into NEON registers using `VLD1`, and written back to memory using `VST1`. You can load two matrices into NEON registers by using the following instructions:

```
vld1.32  {d16-d19}, [r1]!          @ load the first eight elements of matrix 0
vld1.32  {d20-d23}, [r1]!          @ load the second eight elements of matrix 0
vld1.32  {d0-d3}, [r2]!            @ load the first eight elements of matrix 1
vld1.32  {d4-d7}, [r2]!            @ load the second eight elements of matrix 1
```

Because NEON has 32 × 64-bit registers, you can load all of the elements from both input matrices into registers, and still have registers left for use as accumulators. You can use d16 to d23 hold 16 elements from the first matrix, and d0 to d7 hold 16 elements from the second.

In this example, you must handle columns of four 32-bit floating point numbers, which fit into a single 128-bit Q register. Therefore, you can use Q registers frequently. You can calculate a column of results by using only four NEON multiply instructions:

```
vmul.f32    q12, q8, d0[0] @ multiply col element 0 by matrix col 0
vmla.f32    q12, q9, d0[1] @ multiply-acc col element 1 by matrix col 1
vmla.f32    q12, q10, d1[0]@ multiply-acc col element 2 by matrix col 2
vmla.f32    q12, q11, d1[1]@ multiply-acc col element 3 by matrix col 3
```

The first instruction, vmul.f32, implements the operation highlighted in *NEON vector-by-scalar multiplication* on page 4-20 - a11, a21, a31, and a41 in register q8 are each multiplied by b11 , which is element 0 in register d0, and then stored in q12.

Subsequent instructions operate on other columns of the first matrix, and multiply by corresponding elements of the first column of the second matrix. Results are accumulated into q12 to give the first column of values for the results matrix.

Note

The scalar used in the multiply instructions refers to D registers. Although q0[3] is the same value as d1[1], the GNU assembler might not accept the use of Q register to refer to a scalar. Therefore, specify a scalar using a D register.

If you only need to calculate a matrix-by-vector multiplication, which is a common operation in 3D graphics, the operation can be considered complete, and the result vector can be stored to memory. However, to complete the matrix-by-matrix multiplication, you must execute three more iterations:

- The second iteration uses values b12 - b42 in register q1.
- The third iteration uses values b13 - b43 in register q2.
- The fourth iteration uses values b14 - b44 in register q3.

To simplify your code, you can create a macro for the instructions above.

```
.macro mul_col_f32 res_q, col0_d, col1_d
vmul.f32    \res_q, q8, \col0_d[0] @ multiply col element 0 by matrix col 0
vmla.f32    \res_q, q9, \col0_d[1] @ multiply-acc col element 1 by matrix col 1
vmla.f32    \res_q, q10, \col1_d[0] @ multiply-acc col element 2 by matrix col 2
vmla.f32    \res_q, q11, \col1_d[1] @ multiply-acc col element 3 by matrix col 3
.endm
```

The implementation of a four-by-four floating point matrix multiplication now looks like this:

```
vld1.32  {d16-d19}, [r1]! @ load first eight elements of matrix 0
vld1.32  {d20-d23}, [r1]! @ load second eight elements of matrix 0
vld1.32  {d0-d3}, [r2]! @ load first eight elements of matrix 1.
vld1.32  {d4-d7}, [r2]! @ load second eight elements of matrix 1.

mul_col_f32 q12, d0, d1 @ matrix 0 * matrix 1 col 0
mul_col_f32 q13, d2, d3 @ matrix 0 * matrix 1 col 1
mul_col_f32 q14, d4, d5 @ matrix 0 * matrix 1 col 2
mul_col_f32 q15, d6, d7 @ matrix 0 * matrix 1 col 3

vst1.32  {d24-d27}, [r0]! @ store first eight elements of result.
vst1.32  {d28-d31}, [r0]! @ store second eight elements of result.
```

Performance comparison between C and NEON

The two different implementations of the matrix multiplication have significant performance differences. Based on the performance tests on Cortex-A8, the application using NEON is 4x times the performance of the application using the C code.

4.7 [Optional] Using Symmetric Multi-Processing to deliver higher performance

You can use SMP to deliver higher performance if performance is a concern. ARMv7-A introduces support for SMP, which allows multiple identical processing subsystems on a single chip, all running the same instruction set and with equal access to memory, I/Os, and external interrupts.

By using multiple cores, the CPUs of today can complete more work faster, and at lower power, than their single core predecessors. SMP enables mobile processors to not only deliver higher performance, but also meet peak performance demands while staying within mobile power budgets.

4.7.1 Booting

Initialization of the external system might have to be synchronized among cores. Typically, only one of the cores in the system must run code that initializes the memory system and peripherals. Similarly, the initialization of an SMP operating system 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 secondary cores is lost.

If all of the cores come out of reset at the same time, they normally all start executing from the same reset vector. The boot code then reads the cluster ID to determine which core is the primary. The primary core performs the initialization and then signals 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 method requires hardware support to coordinate the reset.

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, depending 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 ARMv7, 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 eight bits identifying individual blocks at each level, that is, Affinity Level 0, 1, and 2.

This information can also be useful to an operating system scheduler, as it indicates the magnitude order of the cost of migrating a process to a different core, processor, or cluster.

The format of the register is 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.

SMP boot in Linux

The boot process comprises the following main stages:

1. The kernel decompresses itself.
2. The processor-dependent kernel code is executed to initialize the CPU and memory.
3. The processor-independent kernel code is executed to start the ARM Linux SMP kernel by booting up all cores and initialize all the kernel components and data structures.

Figure 4-6 shows the boot process of the ARM Linux kernel:

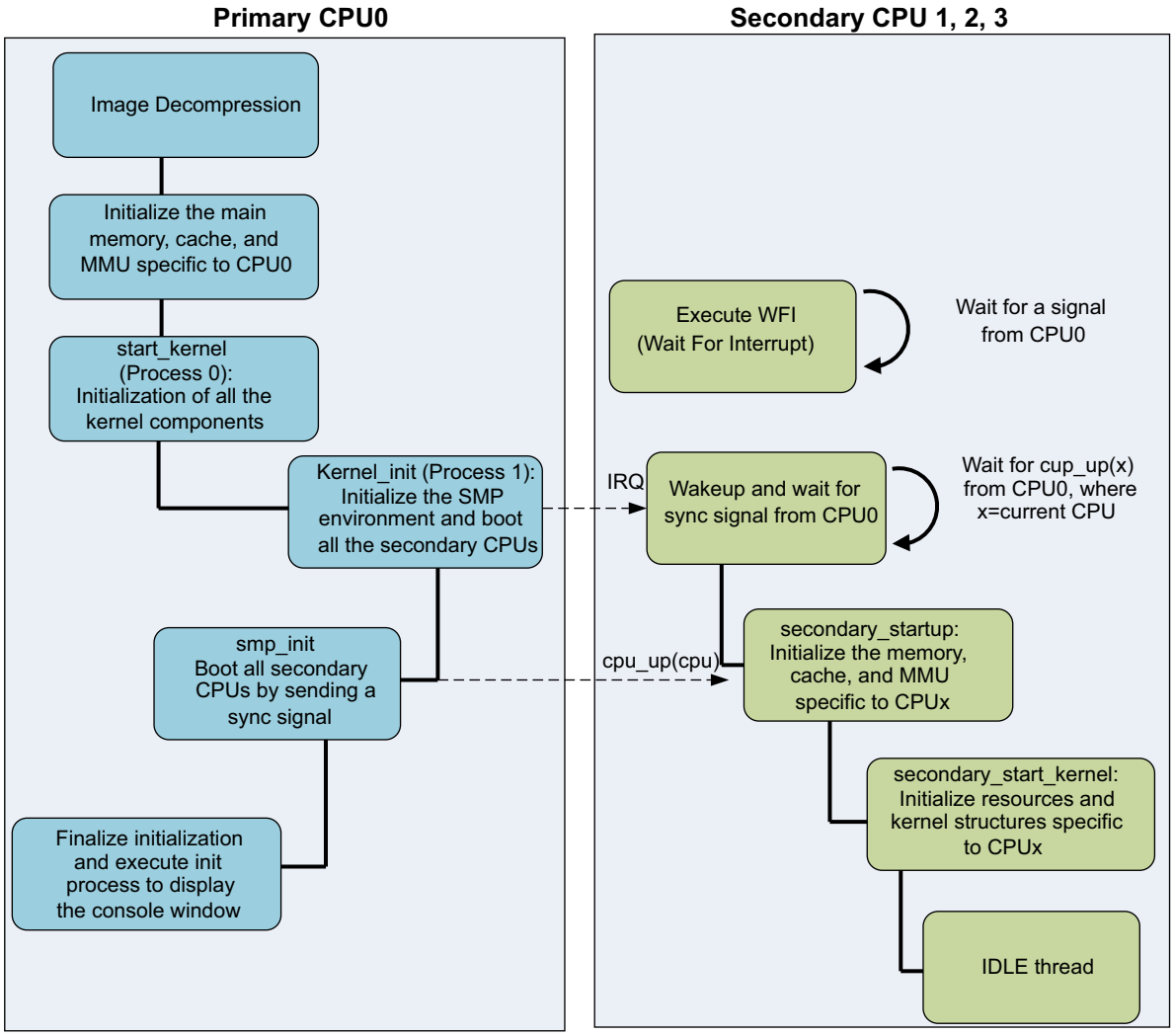

Figure 4-6 Booting flowchart for the ARM Linux Kernel

To boot the primary core, the kernel allocates a 4KB page as the vector page. It maps this using `devicemaps_init()` in the file `arch/arm/mm/mmu.c` to the location of the exception vectors, virtual address `0xFFFF0000` or `0x00000000`. This step is invoked very early in the ARM system boot.

When this step is complete, `trap_init` in `arch/arm/kernel/traps.c` copies the exception vector table, exception stubs, and kuser helpers into the vector page. You must copy the exception vector table to the start of the vector page. Use a series of `memcpy()` operations to copy the exception stubs to address `0x200`, and copy kuser helpers to the top of the page, at `0x1000 - kuser_sz`.

The method for booting the secondary cores can differ somewhat, depending on SoC. The primary core invokes the boot_secondary() method to get a secondary core booted into the operating system. You must implement this method 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 are held in a standby state, using the `WFI` instruction. The primary core provides a startup address for the secondary cores and wakes them using an *Inter-Processor Interrupt* (IPI), an SGI signaled through the GIC. 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 the 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 checks the value of `pen_release` against their own ID value by using the MPIDR register.

Booting of the secondary core proceeds in a way similar to the primary core. The secondary core enables the MMU. 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 can detect the setting of the appropriate bit in `cpu_online_map` and set `pen_release` to the next secondary core.

4.7.2 Programming

An SMP system enables you to run multiple threads efficiently and concurrently across multiple cores. However, it is insufficient in many cases. You must rewrite code to improve application performance by exploiting the benefit of parallelization.

The operating system cannot automatically parallelize an application. It can only treat the application as a single scheduling unit. In such cases, you must split the application into multiple smaller tasks. Each of these tasks can 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.

Decomposition methods

It is good practice to decompose your application into smaller tasks capable of parallel execution. The best way to do it depends on the characteristics of the original application. You can break down large data-processing algorithms into smaller pieces, with a number of similar threads that execute in parallel on smaller portions of a dataset. This method is known as data decomposition.

A different approach is task decomposition. You can identify areas of code that are independent of each other and capable of being executed concurrently. This is more difficult because you must consider the discrete operations being carried out and the interactions among them.

For algorithms that you cannot handle through data or task decomposition, you must analyze the program to identify 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, but do not have a corresponding dependency on time.

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.

Threading models

There are two widely used threading models, the fork-join model and workers pool model.

In the fork-join model, individual threads have explicit start and end conditions. There is an overhead associated with managing their creation, destruction, and latencies associated with the synchronization point. Therefore, threads must be sufficiently long-lived to justify these costs.

If some execution threads are repeatedly required to consume input data, you can use the workers pool threading model. You can 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, dispatches the task to the first available worker thread. Alternatively, the workers pool can contain several different data processing operators, and data items are tagged to show which worker can consume the data.

In each of these models, 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, data dependencies can cause different execution times for similar threads. There can be some synchronization overhead. For example, a parent thread must wait for all spawned threads to return in the fork-join model; a pool of workers must complete data consumption before execution can be resumed.

Threading libraries

You can use a threading library to modify your source code, and make your target application capable of concurrent execution. Multi-threading support is available in the OS. When modifying existing code, you must 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 is allowed 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 ensure interoperability and portability of code among 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. Take OpenMP for example, it 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, compiler directives, and environment variables.

The Pthreads standard provides a set of C primitives that enable you to create, manage, and terminate threads and to control thread synchronization and scheduling attributes. You can use Pthreads to build multi-threaded software to run on our SMP system.

Pthreads provides 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 argument is a pointer to a `pthread_t`, which is where you want to 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 that the new thread starts by executing. The thread is 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 can terminate the thread. A thread can be detached, using `pthread_detach()`. A detached thread automatically has its associated data structures released on exit.

For a thread that has not been detached, this resource cleanup occurs 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. Use this function with caution because 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.

`The function 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 it is no longer required with the `pthread_mutex_destroy()` function. Semaphores are created in a similar way, using `sem_init()`. However, you must specify the initial value of the semaphore. The functions `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.

The following example shows how to use the Pthreads library.

Example 4-11 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;
}
```

Inter-thread communications

Threads use semaphores to signal to another thread. For example, where one thread produces a buffer containing shared data, it can 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.

Threaded performance

There are a few general points to consider when writing a multi-threaded application:

- Each thread has its own stack space. You must be careful with its size if large numbers of threads are in use.
- Multiple threads contending for the same mutex or semaphore result in contention and wasted core cycles.

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 uses only the selected cores 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 can run on any core in an SMP system.

ARM DS-5 Streamline can reveal the affinity of a thread by using a display mode called Core map. You can use this mode to visualize how tasks are divided up by the kernel and shared among several cores.

Thread safety and reentrancy

Functions that can be used concurrently by more than one thread must be both thread-safe and reentrant. This is important for device drivers and for library functions.

For a function to be reentrant, it must meet 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 you must change the implementation by adding synchronization blocks to protect concurrent accesses to shared resources, from different threads. Reentrancy is a stronger property; it means that not every thread-safe function is reentrant.

There are 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.

Performance issues

There are several multi-core specific issues related to performance of threads:

- Bandwidth.

  The connection to external memory is shared among all cores within a cluster. 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 resides within the same coherent cache lines, there can be cache line migration overhead even if the actual variables are not shared.

Bandwidth concerns

Bandwidth issues can be optimized in a number of ways. The code itself must be optimized to minimize cache misses, and therefore reduce the bandwidth utilization.

Another option is to control thread allocation. The kernel scheduler does not monitor data usage by threads. Instead, it uses priority to decide which threads to run. You can provide hints that enable more efficient scheduling by using thread affinity.

Thread dependencies

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 then execute to completion, without preemption. Later, tasks can rely on earlier tasks having completed. This can cause problems if such software is moved to a multi-core system without checking for such assumptions.

A lower-priority task can run at the same time as a higher-priority task, but the expected execution order of the original single-core system is no longer guaranteed. There are several ways to resolve this problem. 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 by using the kernel synchronization mechanisms that give you explicit control over the execution flow and better SMP performance. However, this approach requires the legacy code to be modified.

Cache thrashing

Cortex-A series processors use physically tagged caches that remove the requirement for flushing caches on context switch.

In an SMP system, tasks can migrate among 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 later 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 in the core cache.

Memory-intensive tasks that quickly fill the data cache might thrash each other's cached data. This results in poor performance, because of the higher number of cache misses; this also increases system energy usage, because of additional interaction with external memory.

Multi-core optimizations for cache line migration mitigate the effects of cache thrashing. In addition, the OS scheduler can try to reduce the problem by keeping tasks on the same core. You can also do this by setting core affinity to threads and processes.

False sharing

This is a problem of systems with shared coherent caches and is 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 is altered by another core. The MESI protocol can end up migrating data that is not truly shared among 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 reloaded 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.

Deadlock and livelock

When writing code that includes critical sections, you must know that the following common problems can lead to correct execution of the program:

- Deadlock is the situation where two or more threads are waiting for each other to release a resource. Such threads are blocked, waiting for a lock that can never be released.
- Livelock occurs when multiple threads can execute, without blocking indefinitely as in the deadlock case. However, the system as a whole cannot proceed, because of a repeated pattern of resource contention.

Both deadlocks and livelocks can be avoided either by correct software design, or by the use of lock-free software techniques.

4.7.3 Synchronization primitives, locks, and semaphore

When porting software from a single core environment to run on multi-core cluster, you might need to modify code to perform the following operations:

- Enforce a particular order of execution.
- 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 can improve software performance. Serialization and multiple threads contending for a resource can reduce the performance benefit provided by the multiple cores. In all cases, minimizing the size of the critical section provides best performance.

Completions

Completions are a feature provided by the Linux kernel. You can use them to serialize task execution. They provide a lightweight mechanism that 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)`. The task that is sending the signal typically uses either of the following:

- `complete (struct completion *comp)`, which wakes up one waiting process
- `complete_all (struct completion *comp)`, which 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.

Spinlocks

A spinlock provides a simple binary locking mechanism to protect 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 preemption.

Semaphores

Semaphores are a widely used method to control accesses to shared resources. You can use them to achieve serialization of execution. They provide a counting locking mechanism that can cope with multiple threads attempting to lock.

They can be used to protect 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 is 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 tasks can run. Semaphores can be binary (in which case they are also mutexes) or counting.

Lock-free synchronization

If you have multiple readers and writers to a shared resource, using a mutex might not be efficient. A mutex would prevent concurrent read access to the shared resource because only a single thread is permitted inside the critical section.

The use of lock-free data structures, such as circular buffers, can avoid the overheads associated with spinlocks or semaphores. The Linux kernel also provides the following synchronization mechanisms that are lock-free:

- *Read-Copy-Update*.
- *Seqlocks* on page 4-32.

Read-Copy-Update

Read-Copy-Update (RCU) 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, and access to shared resources must be granted 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 ensures 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.

Seqlocks

Seqlocks provide quick access to shared resources, without using locks. They are optimized for short critical sections. Readers can access the shared resource with no overhead, but must explicitly check and retry if there is a conflict with a write. Writes 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. Using a seqlock, instead of a mutex, enables many readers to share access, without locking out the writer from accessing the critical section.

4.8 Choosing the right software development tools and debug adaptors

You must choose the right software development tools and debug adaptors for the migration. This section shows you an overview of the ARM software development toolchain, GCC toolchain, and debug adaptors, and gives you suggestions about tool selection.

ARM provides a number of toolchains for software development, including:

- *Software Development Toolkit* (SDT).
- *ARM Developer Suite* (ADS).
- *RealView Development Suite* (RVDS).
- *ARM Development Studio 5* (DS™-5).

You can also use the GCC toolchain supplied by CodeSourcery and Linaro. There are also other third-party toolchains, which are not covered in this document. Table 4-2 shows an overview of the ARM software development toolchain:

Table 4-2 ARM software development toolchain

Tool suits	Introduction
SDT	• This tool is obsolete. • SDT provides support for processors up to ARMv4T. • SDT has been superseded by ADS.
ADS	• This tool is obsolete. • ADS consists of a suite of applications that enable you to write and debug applications for the ARM family of RISC processors. • ADS provides support for ARMv5TEJ, including ARM9E and ARM10. • ADS has been superseded by RVDS.
RVDS	• RVDS is no longer maintained. • The RVDS is designed for SoC, FPGA, and ASIC users that create complex embedded applications or interfaces to platform OS components. • RVDS provides support for the following processors: — ARM7, ARM9, and ARM11 processor families. — ARM11 MPCore multicore processor. — Cortex family of processors.
DS-5	• DS-5 is a professional software development solution for Linux-based systems and bare-metal embedded systems, covering all stages in development from boot code and kernel porting to application and bare-metal debugging including performance analysis. • DS-5 provides support for the following processors: — ARM11. — ARM9. — ARM7. — Cortex-A. — Cortex-R. — Cortex-M.

You can also use the GCC toolchain supplied by CodeSourcery and Linaro. Table 4-3 shows an overview of the GCC toolchain:

Table 4-3 GCC software development tool chain

Suppliers	Introduction
CodeSourcery	• Sourcery CodeBench Lite Edition includes: — GNU C and C++ compilers. — GNU assembler and linker. — C and C++ runtime libraries. — GNU debugger. • For further information on the GNU toolchain supplied by CodeSourcery, see the website at `http://www.codesourcery.com/gnu_toolchains/arm/`
Linaro	• Linaro is a non-profit organization that 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 among a number of technology companies to provide engineering help and resources to the open source community. • Linaro provides support for the ARM Cortex-A processors family including Cortex-A5, Cortex-A7, Cortex-A8, Cortex-A9, and Cortex-A15.

Table 4-4 shows an overview of the debug adaptors:

Table 4-4 Debug adaptors

Debug adaptors	Introduction
Multi-ICE	• Multi-ICE is legacy tool, and is no longer in production. • Multi-ICE is the EmbeddedICE logic debug solution from ARM. It enables you to debug software running on ARM processor cores that include the EmbeddedICE logic. • Multi-ICE provides support for the following processors: — ARM7. — ARM9. — ARM10.
RVI & RVT2	• *RealView ICE* (RVI) and *RealView Trace 2* (RVT2) are ARM legacy debug adapters, and are no longer in production. • RVI & RVT2 provides support for the following processors: — ARM7 — ARM9 — ARM11 — Cortex-A — Cortex-R — Cortex-M
DSTREAM	• DSTREAM is an ARM debug and trace hardware unit. It enables you to connect a software debugger to an ARM processor-based target using a hardware interface such as JTAG or *Serial Wire Debug* (SWD). It also enables the collection of trace from the device for non-intrusive debug and code optimization. • DSTREAM provides support for all processors from ARMv4 to Armv8.

4.8.1 Tools selection

If you are building the Linux Kernel, ARM recommends you use the GCC toolchains. The Linux kernel has a large amount of assembly code that is written in GNU assembler syntax. The ARM assembler does not support the GNU assembler syntax, and therefore cannot be used to build the Linux kernel.

If you are building bare-metal applications, ARM recommends you use the ARM toolchains, which can be beneficial to application performance:

- If you are using the legacy ADS and Multi-ICE, ARM recommends you upgrade to DS-5 and DSTREAM. ADS and Multi-ICE do not support ARMv7 targets.
- If you are using RVDS with RVI and RVT2, you can continue to use them.

  Note

  These legacy tools are no longer maintained.

4.9 [Optional] Enabling FPU to improve application performance

If your source code contains floating-point arithmetic, you can enable FPU to improve the application performance. To enable your ARM Floating Point architecture correctly, you must select a number of compiler options in the ARM and GCC compilers.

4.9.1 Enabling Vector Floating-Point

The VFP extension is disabled at reset. Any attempt to execute a VFP instruction results in an Undefined Instruction exception being taken. To enable software to access VFP features, the following conditions must be satisfied:

- Enable access to CP10 and CP11 for the appropriate privilege level.
- If you require Non-secure access to the VFP features is required, set the access flags for CP10 and CP11 in the NSACR to 1.

In addition, software must set the FPEXC.EN bit to 1 to enable most VFP operations.

When VFP operation is disabled because FPEXC.EN is 0, all VFP instructions are treated as undefined instructions, except for execution of the following in privileged modes:

- A VMSR to the FPEXC or FPSID register.
- A VMRS from the FPEXC, FPSID, MVFR0, or MVFR1 registers.

To use the FPU in Secure state only

To use the FPU in Secure state only, complete the following steps:

1. Set the CPACR for access to CP10 and CP11, the FPU coprocessors:

```
LDR r0, =(0xF << 20)
MCR p15, 0, r0, c1, c0, 2
```

2. Set the FPEXC.EN bit to enable the FPU:

```
MOV r3, #0x4000000
VMSR FPEXC, r3
```

To use the FPU in Secure state and Non-secure state

To use the FPU in Secure state and Non-secure state, complete the following steps:

1. Set bits [11:10] of the NSACR for access to CP10 and CP11 from both Secure and Non-secure states:

```
MRC p15, 0, r0, c1, c1, 2
ORR r0, r0, #2_11<<10 ; enable fpu
MCR p15, 0, r0, c1, c1, 2
```

2. Set the CPACR for access to CP10 and CP11:

```
LDR r0, =(0xF << 20)
MCR p15, 0, r0, c1, c0, 2
```

3. Set the FPEXC EN bit to enable the FPU:

```
MOV r3, #0x40000000
VMSR FPEXC, r3
```

4.9.2 Vector Floating-Point support in the ARM Compiler

VFP is fully supported by the ARM compiler. However, you can configure some builds to assume no VFP support by default, in which case floating-point calculations use library code.

To enable VFP support, use the following option:

- `--fpu=name`, which specifies the target floating-point hardware.

To specify support for a specific VFP implementation on an ARM Cortex-A series processor, use the following options:

- `--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).

You can use these options for code that only runs on these VFP implementations, and does not require backward compatibility with older VFP implementations. Use `--fpu=list` to see the full list of FPUs supported.

Use the following options 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 the explicit or implicit use of `--fpu`.

To achieve better application performance, recompile source files for a target with an FPU and hardware floating-point linkage. If it is impossible to recompile existing code, use options, such as `--fpu=softvfp+vfpv4`, to maintain compatibility with existing software.

4.9.3 Vector Floating-Point support in GCC

VFP is fully supported by GCC. However, you can configure some builds to default to assume no VFP support by default, in which case floating-point calculations use library code.

To enable VFP support, use the following option:

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

To specify support for a specific VFP implementation on an ARM Cortex-A series processor, use the following options:

- `-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).

You can use these options for code that only runs on these VFP implementations, and does not require backward compatibility with older VFP implementations.

To specify which ABI to use to enable VFP, use the following options:

- `-mfloat-abi=softfp`

  `softfp` uses a Procedure Call Standard compatible with software floating-point, so it provides binary compatibility with legacy code. This permits the running of older soft float code with new libraries that support hardware floating-point, but still uses hardware floating-point registers between function calls.

- `-mfloat-abi=hard`

  `hard` has floating-point values passed in floating-point registers. This is more efficient but is not backward-compatible with the `softfp` ABI variant.

Note

There can be a significant function call overhead when using `-mfloat-abi=softfp`, if many floating-point values are being passed.

Chapter 5
Migration notes

Read this chapter for some useful points when porting source code from ARMv5 to ARMv7.

It contains the following section:

5.1 Migration notes

If you are porting source code from ARMv5 to ARMv7, there are a number of points that you must consider, depending on your source code and configuration.

- If your source code covers MMU or Cache maintenance, consult *Memory model* on page 3-11 and *Changing startup code and set up MMU cache* on page 4-2 to see the changes in ARMv7.
- If your source code contains floating-point arithmetic, you can enable FPU to improve the application performance.

  For information about FPU differences between ARMv5 and ARMv7, see *Floating-Point Unit* on page 3-25

  For information about how to enable and use FPU, see *[Optional] Enabling FPU to improve application performance* on page 4-36.
- If your code contains multimedia and signal processing algorithms, you can use NEON to improve the application performance.

  For an introduction to NEON, see *NEON* on page 3-26.

  For information about how to use NEON, see *[Optional] Using NEON to improve application performance* on page 4-16.
- If your code has memory ordering issue, see *Synchronization* on page 3-19 and *Replacing ARMv5 barriers with equivalent ARMv7 barriers* on page 4-8.
- If power saving is a concern, change your power management. As ARMv7 supports more features, it consumes more power than ARMv5.

  In ARMv5, the wait for interrupt instruction is `MCR p15,0,<Rd>,c7,c0,4`, and this instruction is replaced by `WFI` in ARMv7.

  If you want to support static power management, you must consult the section that describes power management in the corresponding TRM, because each processor has different power-down and power-up sequences.
- If you want to know which is the correct toolchain and debug adaptor to use for ARMv7, see *Choosing the right software development tools and debug adaptors* on page 4-33.
- If you want to know how to initialize interrupts and modify interrupt handlers, see *Exception model* on page 3-9 and *Modifying exception-handling code* on page 4-5.
- If you want to know memory type differences and memory type mapping between ARMv5 and ARMv7, see *Memory management* on page 3-11 and *Memory type mapping* on page 4-4.

Index

M

N

O

P

R

S

T

U

V

W