Armv8/armv9 架构入门指南

Release v1.0

baron

Oct 06, 2022

CONTENTS


个人微信号
微信公众号
微信群-免费交流
ARM-Trustzone-TEE-ATF-SOC群7
微信临时群-下方课程咨询
《ARMv8-ARMv9架构学习》课程咨询
CSDN课程推荐
ARMv8/ARMv9架构学习系列课程
各位好，懂的人自然懂，不要在学习十年前(甚至十五六年前)的armv6了，不要学习七八年前的arm32了，不要再学习那一堆的过时的技术了（gicv2、arm几种模式、big.LITTLE架构），要学我们...
嵌入式
共56节 · 1075人已学习
61节课/22小时
¥799.0
免费试学

CHAPTER

ONE

1. 前言

1.1 简介

哈哈，TODO

1.2 推荐序

真好真香，同志们，卷起来。

1.3 说明

这是一篇由网友一起翻译的文档。原文是 `DEN0024A_v8_architecture_PG_1.0.pdf`。仅供大家学习使用，请勿传播，请勿用于商业用途。我们也不承担法律责任。

1.4 作者

参与翻译工作的作者简介（排名不分先后）：

1.5 Release

CHAPTER

TWO

2. ARMV8-A 架构和处理器

ARM 架构可以追溯到 1985 年，但它并没有停滞不前。相反，它从早期的 ARM 内核开始大规模发展，每一步都增加了特性和功能：

- ARMv4 及更早版本这些早期的处理器仅使用 ARM 32 位指令集。ARMv4T ARMv4T 架构将 Thumb 16 位指令集添加到 ARM 32 位指令集。这是第一个获得广泛许可的架构。它由 ARM7TDMI® 和 ARM9TDMI® 处理器实现
- ARMv5TE ARMv5TE 架构增加了对 DSP 类型操作的改进，饱和算法，以及用于 ARM 和 Thumb 互通。ARM926EJ-S® 实现了这个架构。
- ARMv6 ARMv6 进行了多项增强，包括支持未对齐的内存访问、对内存架构的重大更改以及对多处理器的支持。此外，还包括对在 32 位寄存器中的字节或半字上操作的 SIMD 操作的一些支持。ARM1136JF-S® 实现了这个架构。ARMv6 架构还提供了一些可选扩展，特别是 Thumb-2 和安全扩展 (TrustZone®)。Thumb-2 将 Thumb 扩展为混合长度的 16 位和 32 位指令集。
- ARMv7-A ARMv7-A 架构强制要求 Thumb-2 扩展，并添加了高级 SIMD 扩展 (NEON)。在 ARMv7 之前，所有内核都遵循基本相同的架构或功能集。为了帮助解决越来越多的不同应用程序，ARM 引入了一组架构配置文件：(1)ARMv7-A (2)ARMv7-M (3)ARMv7-R

2.1 2.1 ARMv8-A

ARMv8-A 架构是针对应用程序配置文件的最新一代 ARM 架构。ARMv8 这个名称用于描述整体架构，现在包括 32 位执行状态和 64 位执行状态。它引入了使用 64 位宽寄存器执行的能力，同时保持与现有 ARMv7 软件的向后兼容性。

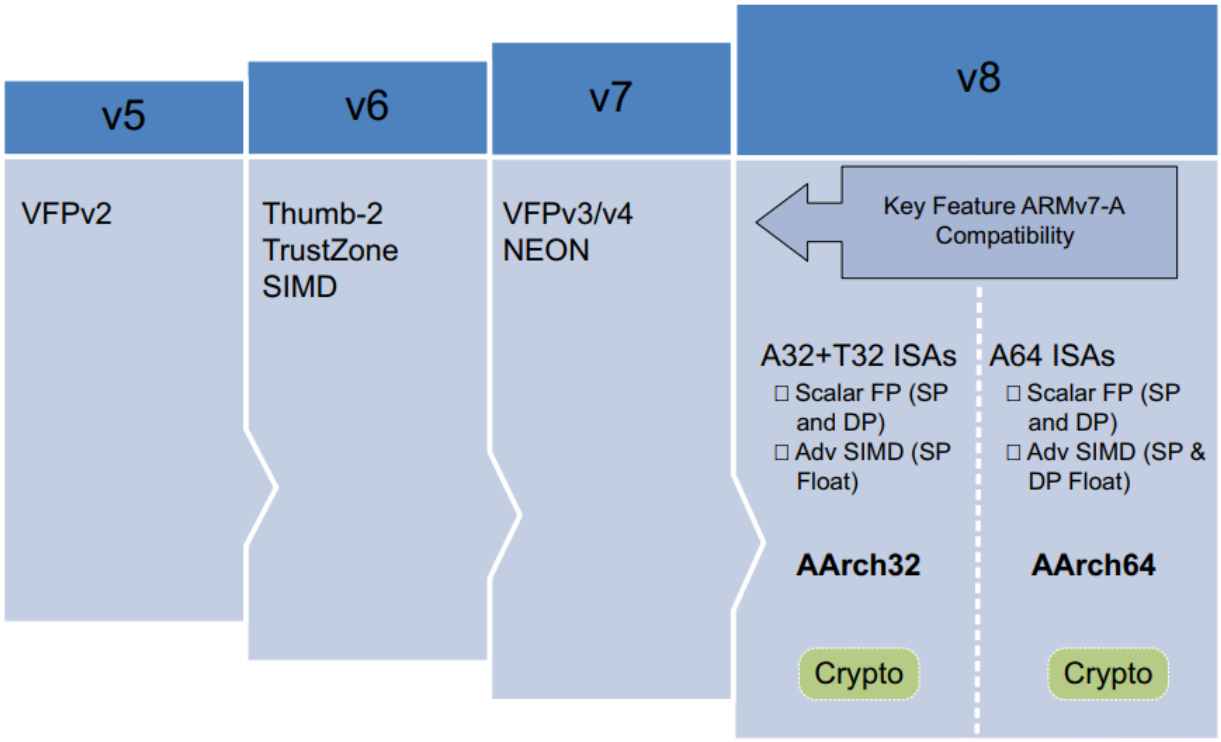

image-20220314221123659

ARMv8-A 架构引入了许多更改，可以设计出性能显着提高的处理器实现：

- 大物理地址 (Large physical address) 这使处理器能够访问超过 4GB 的物理内存。
- 64 位虚拟寻址 (64-bit virtual addressing) 这使虚拟内存超出 4GB 限制。这对于使用内存映射文件 I/O 或稀疏寻址的现代桌面和服务器软件很重要。
- 自动事件信号 (Automatic event signaling) 这可以实现节能、高性能的自旋锁
- 更大的寄存器文件 (Larger register files) 31 个 64 位通用寄存器可提高性能并减少堆栈使用。
- 高效的 64 位立即生成 (Efficient 64-bit immediate generation) 对文字池的需求较少
- 较大的 PC 相对寻址范围 (Large PC-relative addressing range) 一个 +/-4GB 的寻址范围，用于在共享库和与位置无关的可执行文件中进行有效的数据寻址。
- 额外的 16KB 和 64KB 翻译颗粒 (Additional 16KB and 64KB translation granules) 这降低了翻译后备缓冲区 (TLB) 未命中率和页面遍历深度。
- 新的异常模型 (New exception model) 这降低了操作系统和管理程序软件的复杂性。

- 高效的缓存管理理 Efficient cache management) 用户空间缓存操作提高了动态代码生成效率。使用数据缓存零指令快速清除数据缓存。
- 硬件加速密码学 (Hardware-accelerated cryptography) 提供 3 到 10 倍更好的软件加密性能。这对于小粒度解密和加密非常有用，因为太小而无法有效地卸载到硬件加速器，例如 https。
- 加载 - 获取、存储 - 释放指令 (Load-Acquire, Store-Release instructions) 专为 C++11、C11、Java 内存模型而设计。它们通过消除显式内存屏障指令来提高线程安全代码的性能。
- NEON 双精度浮点高级 SIMD(NEON double-precision floating-point advanced SIMD) 这使得 SIMD 矢量化能够应用于更广泛的算法集，例如科学计算、高性能计算 (HPC) 和超级计算机。

2.2 2.2 ARMv8-A 处理器属性

	Processor	
	Cortex-A53	**Cortex-A57**
Release date	July 2014	January 2015
Typical clock speed	2GHz on 28nm	1.5 to 2.5 GHz on 20nm
Execution order	In-order	Out of order, speculative issue, superscalar
Cores	1 to 4	1 to 4
Integer Peak throughput	2.3MIPS/MHz	4.1 to 4.76MIPS/MHz[a]
Floating-point Unit	Yes	Yes
Half-precision	Yes	Yes
Hardware Divide	Yes	Yes
Fused Multiply Accumulate	Yes	Yes
Pipeline stages	8	15+
Return stack entries	4	8
Generic Interrupt Controller	External	External
AMBA interface	64-bit I/F AMBA 4 (Supports AMBA 4 and AMBA 5)	128-bit I/F AMBA 4 (Supports AMBA 4 and AMBA 5)
L1 Cache size (Instruction)	8KB to 64 KB	48KB
L1 Cache structure (Instruction)	2-way set associative	3-way set associative
L1 Cache size (Data)	8KB to 64KB	32KB
L1 Cache structure (Data)	4-way set associative	2-way set associative
L2 Cache	Optional	Integrated
L2 Cache size	128KB to 2MB	512KB to 2MB
L2 Cache structure	16-way set associative	16-way set associative
Main TLB entries	512	1024
uTLB entries	10	48 I-side
		32 D-side

image-20220314222155065

2.2.1 2.2.1 ARMv8 处理器

Cortex-A53 处理器

Cortex-A53 处理器是一款中档、低功耗处理器，在单个集群中具有一到四个内核，每个内核都有一个 L1 缓存子系统、一个可选的集成 GICv3/4 接口和一个可选的 L2 缓存控制器。

Cortex-A53 处理器是一款非常节能的处理器，能够支持 32 位和 64 位代码。它提供了比非常成功的 Cortex-A7 处理器更高的性能。它能够部署为独立的应用处理器，或在 big.LITTLE 配置中与 Cortex-A57 处理器配对，以获得最佳性能、可扩展性和能源效率。

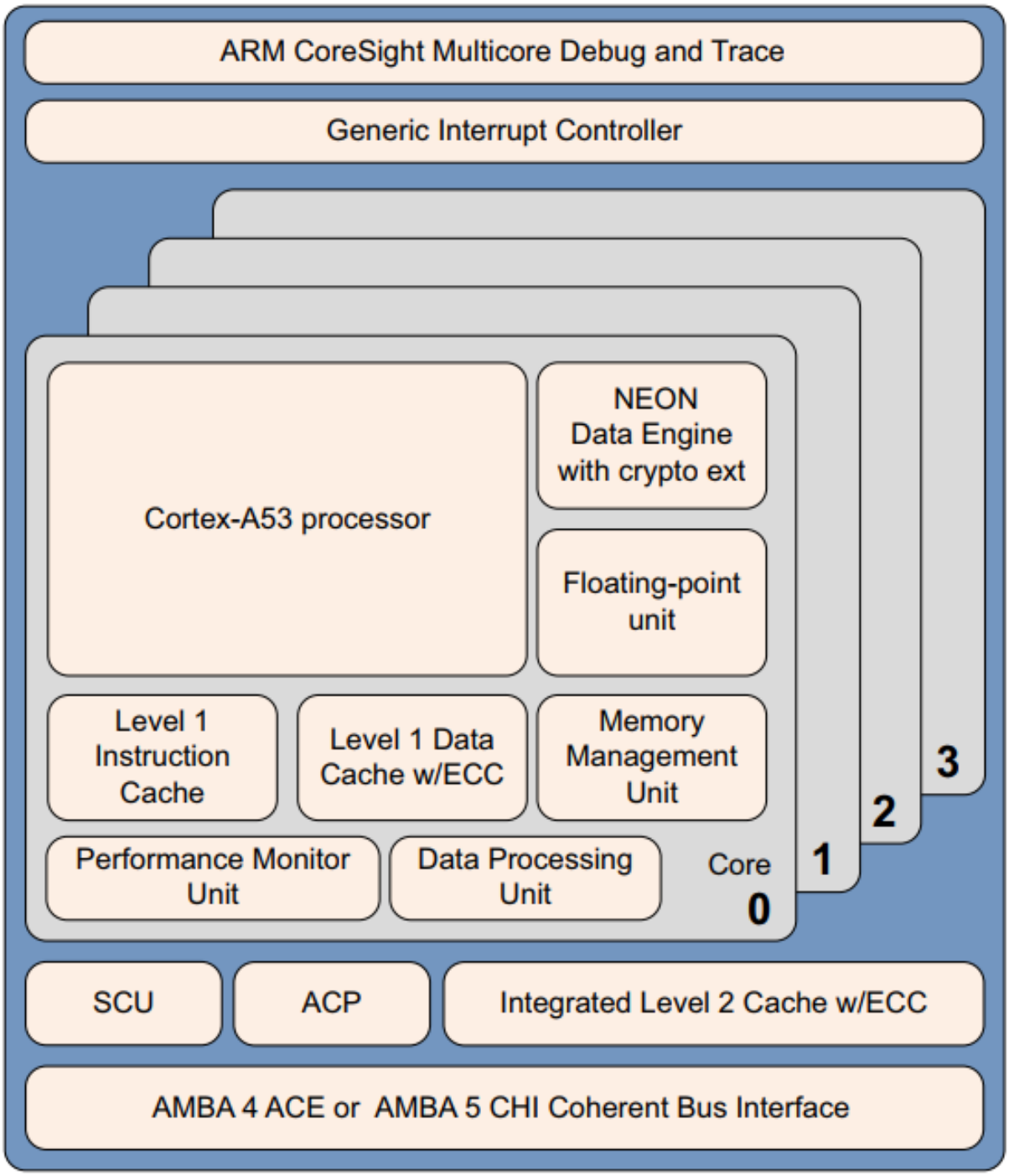

image-20220314222413650

Cortex-A53 处理器具有以下特性：

- 有序的八级流水线。
- 通过使用分层时钟门控、电源域和高级保留模式来降低功耗。
- 通过重复执行资源和双指令解码器提高双发能力。

- 功耗优化的二级缓存设计可提供更低的延迟并在性能与效率之间取得平衡。

Cortex-A57 处理器 Cortex-A57 处理器面向移动和企业计算应用，包括计算密集型 64 位应用，例如高端计算机、平板电脑和服务器产品。它可以与 Cortex-A53 处理器一起使用到 ARM big.LITTLE 配置中，以实现可扩展的性能和更高效的能源使用。

Cortex-A57 处理器具有与其他处理器的高速缓存一致性互操作性，包括用于 GPU 计算的 ARM Mali™ 系列图形处理单元 (GPU)，并为高性能企业应用程序提供可选的可靠性和可扩展性功能。它提供了比 ARMv7Cortex-A15 处理器更高的性能，并具有更高的能效水平。加密扩展的包含将加密算法的性能提高了 10 倍于上一代处理器。

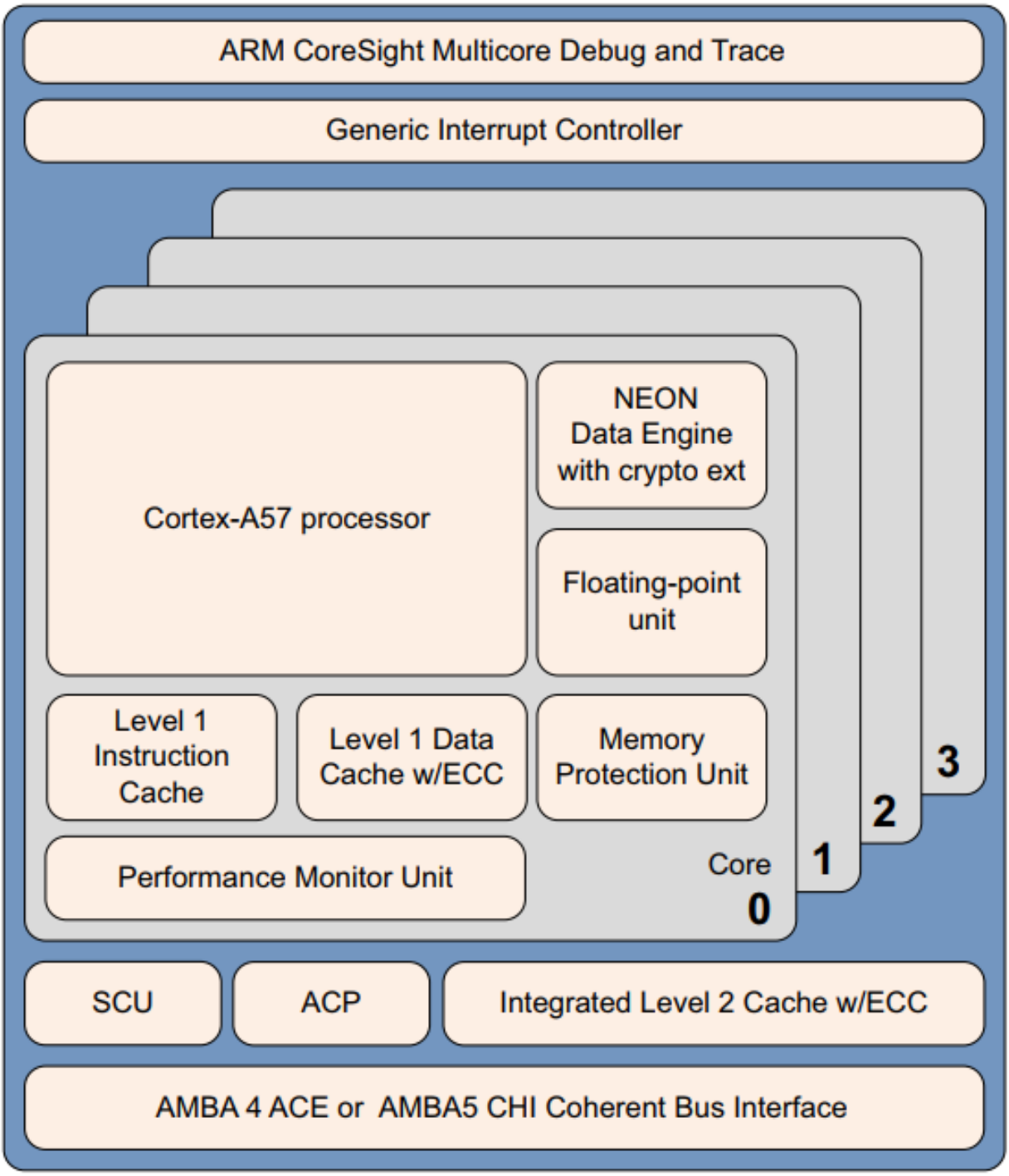

image-20220314222701135

Cortex-A57 处理器完全实现了 ARMv8-A 架构。它支持多核操作，在单个集群中具有一到四核多处理。通过 AMBA5 CHI 或 AMBA 4 ACE 技术，可以实现多个一致的 SMP 集群。可通过 CoreSight 技术进行调试和跟踪。

Cortex-A57 处理器具有以下特性：

- 乱序，15+ 阶段流水线。

- 省电功能包括路径预测、标记减少和缓存查找抑制。
- 通过重复执行资源增加峰值指令吞吐量。具有本地化解码、3 宽解码带宽的功率优化指令解码。
- 性能优化的 L2 缓存设计使集群中的多个核心可以同时访问 L2。

CHAPTER

THREE

3. ARMV8 基础知识

在 ARMv8 中，执行发生在四个异常级别之一。在 AArch64 中，异常级别决定了特权级别，类似于 ARMv7 中定义的特权级别。异常级别决定特权级别，因此在 ELn 执行对应于特权 PLn。类似地，具有比另一个更大的 n 值的异常级别处于更高的异常级别。一个数字比另一个小的异常级别被描述为处于较低的异常级别。

异常级别提供了适用于 ARMv8 架构的所有操作状态的软件执行权限的逻辑分离。它类似于并支持计算机科学中常见的分层保护域的概念。

- EL0 Normal user applications.
- EL1 Operating system kernel typically described as privileged.
- EL2 Hypervisor.
- EL3 Low-level firmware, including the Secure Monitor.

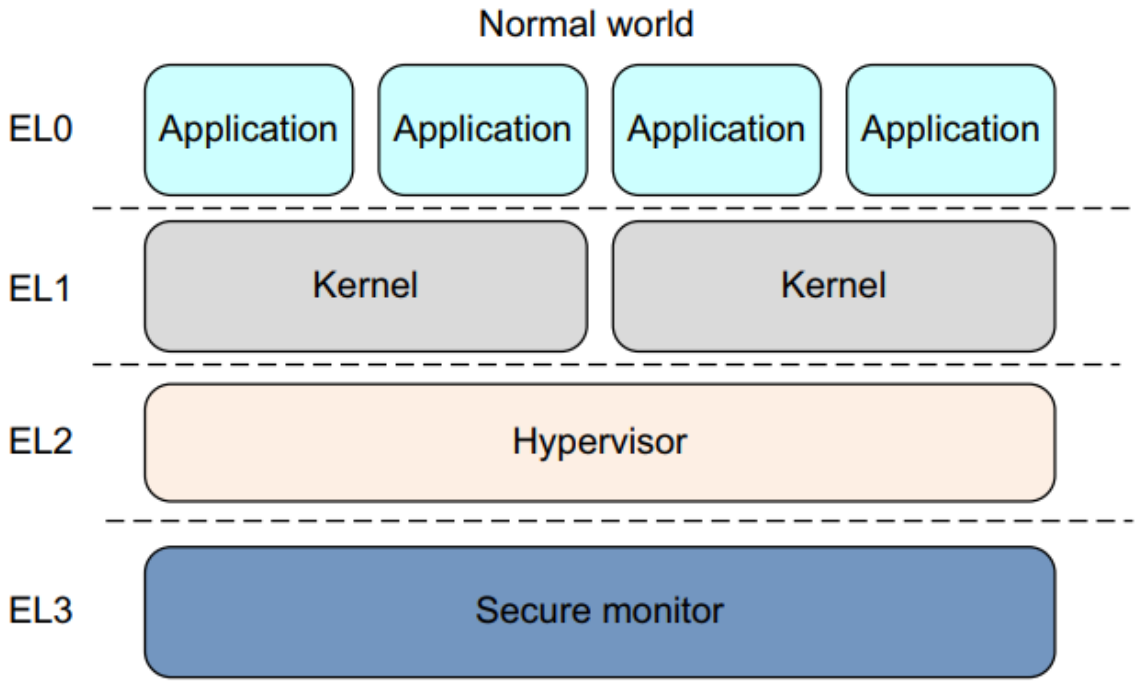

image-20220314223244988

通常，一个软件，例如应用程序、操作系统的内核或管理程序，占用一个异常级别。此规则的一个例外是内核中的虚拟机管理程序，例如 KVM，它在 EL2 和 EL1 上运行。

ARMv8-A 提供两种安全状态，安全和非安全。非安全状态也称为正常世界。这使操作系统 (OS) 能够与受信任的操作系统在同一硬件上并行运新 g，并提供针对某些软件攻击和硬件攻击的保护。ARM TrustZone 技术使系统能够在正常和安全世界之间进行分区。与 ARMv7-A 架构一样，安全监视器充当在正常和安全世界之间移动的网关。

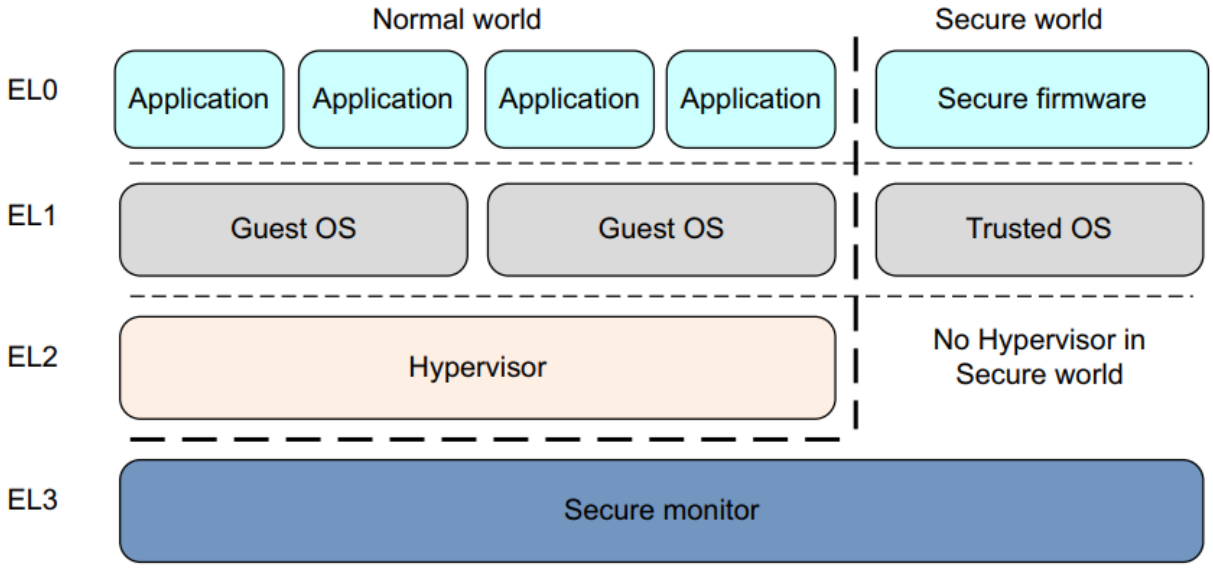

image-20220314223407631

ARMv8-A 还提供对虚拟化的支持，但仅在普通世界中。这意味着管理程序或虚拟机管理器 (VMM) 代码可以在系统上运行并托管多个客户操作系统。每个客户操作系统本质上都在虚拟机上运行。然后，每个操作系统都不会意识到它正在与其他客户操作系统共享系统上的时间。

正常世界（对应于非安全状态）具有以下特权组件：

- Guest OS kernels 此类内核包括在非安全 EL1 中运行的 Linux 或 Windows。在管理程序下运行时，丰富的操作系统内核可以作为来宾或主机运行，具体取决于管理程序模型。
- Hypervisor 这在 EL2 上运行，它始终是非安全的。虚拟机管理程序在存在并启用时，可为丰富的操作系统内核提供虚拟化服务。

安全世界具有以下特权组件：

- Secure firmware 在应用处理器上，这个固件必须是在启动时运行的第一件事。它提供了多种服务，包括平台初始化、可信操作系统的安装以及安全监视器调用的路由。
- Trusted OS 受信任的操作系统为普通世界提供安全服务，并为执行安全或受信任的应用程序提供运行时环境。

ARMv8 架构中的安全监视器处于更高的异常级别，并且比所有其他级别具有更高的特权。这提供了软件特权的逻辑模型。

3.1 3.1 执行状态

ARMv8 架构定义了两种执行状态，AArch64 和 AArch32。每个状态分别用于描述使用 64 位宽通用寄存器或 32 位宽通用寄存器的执行。虽然 ARMv8 AArch32 保留了 ARMv7 对特权的定义，但在 AArch64 中，特权级别由异常级别决定。因此，在 ELn 的执行对应于特权 PLn。

当处于 AArch64 状态时，处理器执行 A64 指令集。当处于 AArch32 状态时，处理器可以执行 A32（在早期版本的架构中称为 ARM）或 T32 (Thumb) 指令集。

下图显示了 AArch64 和 AArch32 中异常级别的组织。AArch64：

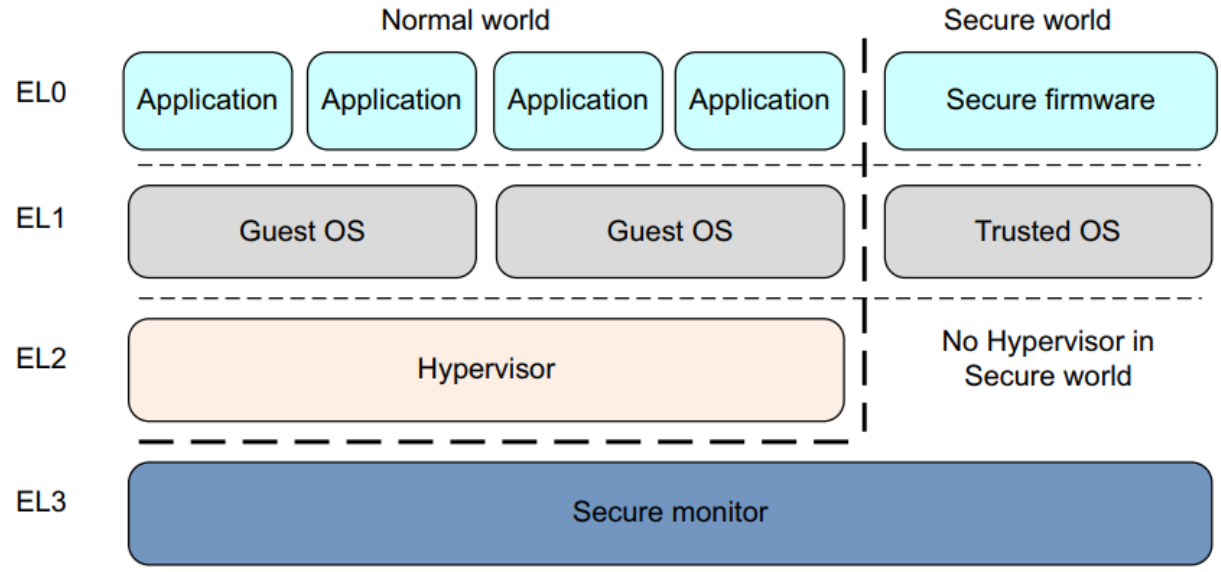

image-20220314223822814

AArch32：

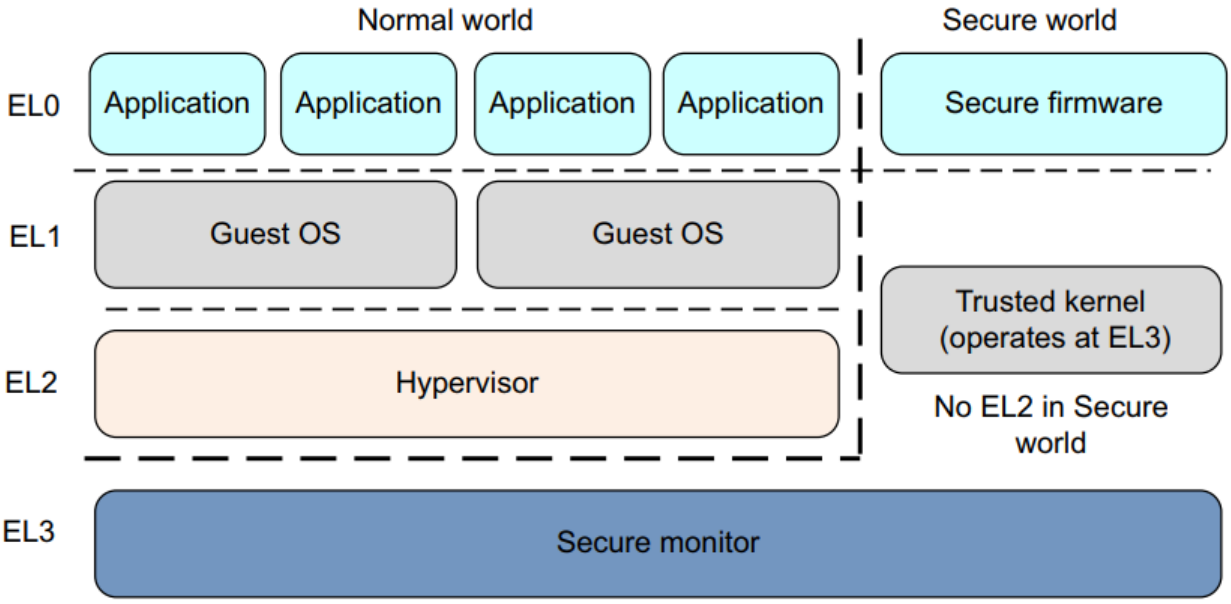

image-

20220314223838166

在 AArch32 状态下，Trusted OS 软件在 Secure EL3 中执行，而在 AArch64 状态下，它主要在 Secure EL1 中执行。

3.2 3.2 更改异常级别

在 ARMv7 架构中，处理器模式可以在特权软件控制下更改，也可以在发生异常时自动更改。当发生异常时，内核保存当前执行状态和返回地址，进入所需模式，并可能禁用硬件中断。

下表对此进行了总结。应用程序以最低级别的特权 PL0 运行，即以前的非特权模式。操作系统在 PL1 上运行，Hypervisor 在具有虚拟化扩展的系统中在 PL2 上运行。安全监视器作为在安全和非安全（正常）世界之间移动的网关，也在 PL1 上运行。

Mode	Function	Security state	Privilege level
User (USR)	Unprivileged mode in which most applications run	Both	PL0
FIQ	Entered on an FIQ interrupt exception	Both	PL1
IRQ	Entered on an IRQ interrupt exception	Both	PL1
Supervisor (SVC)	Entered on reset or when a Supervisor Call instruction (SVC) is executed	Both	PL1
Monitor (MON)	Entered when the SMC instruction (Secure Monitor Call) is executed or when the processor takes an exception which is configured for secure handling. Provided to support switching between Secure and Non-secure states.	Secure only	PL1
Abort (ABT)	Entered on a memory access exception	Both	PL1
Undef (UND)	Entered when an undefined instruction is executed	Both	PL1
System (SYS)	Privileged mode, sharing the register view with User mode	Both	PL1
Hyp (HYP)	Entered by the Hypervisor Call and Hyp Trap exceptions.	Non-secure only	PL2

image-20220314223946303

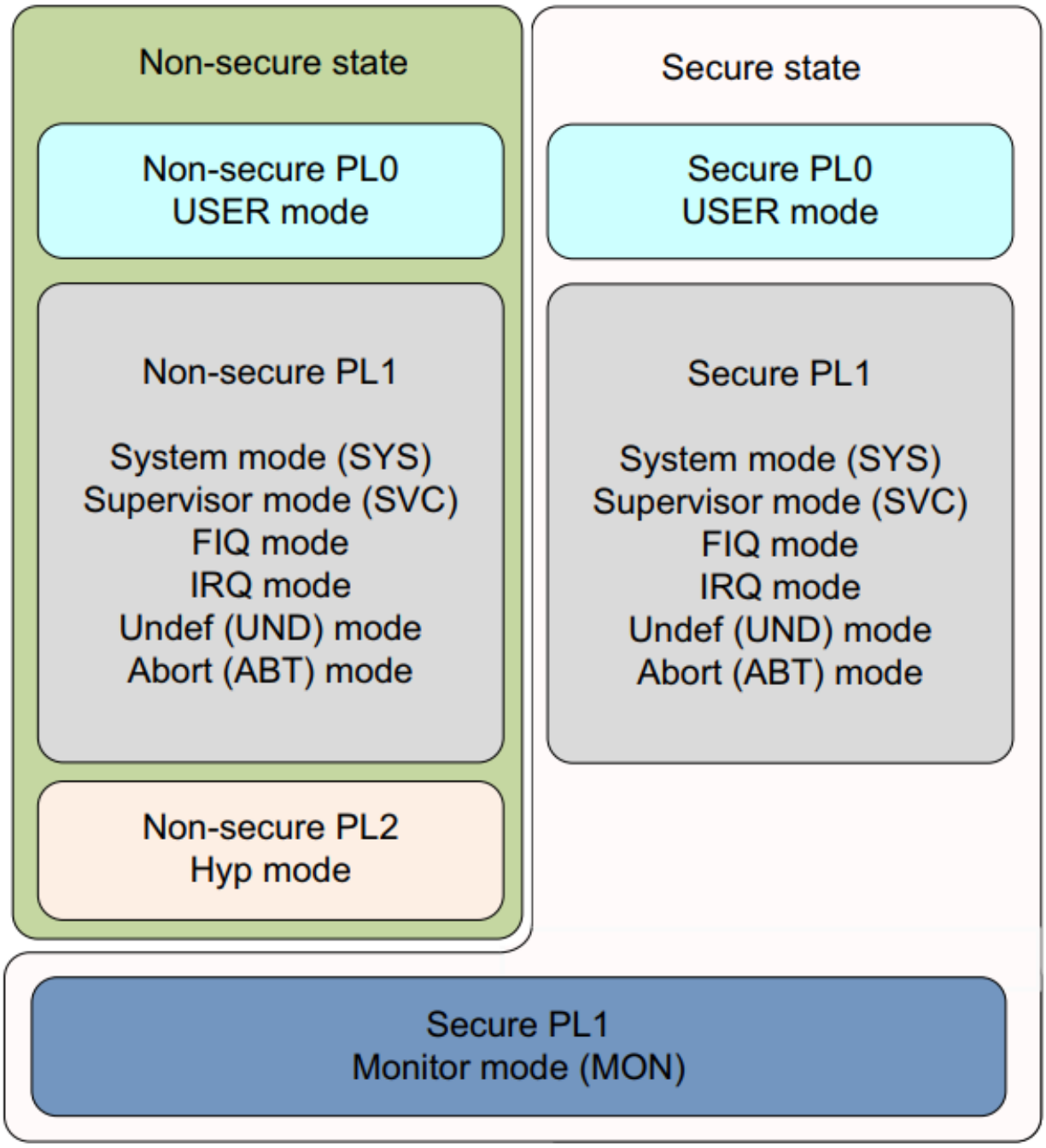

image-20220314224010456

在 AArch64 中，处理器模式映射到异常级别，如图 3-6 所示。与在 ARMv7 (AArch32) 中一样，当发生异常时，处理器将更改为支持处理异常的异常级别（模式）。

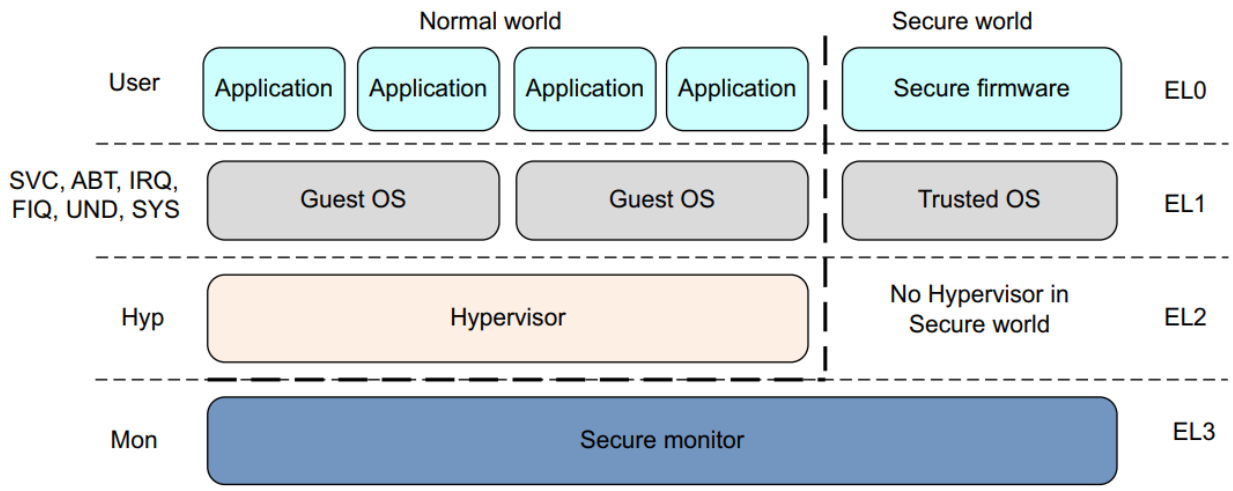

image-20220314224048313

异常级别之间的移动遵循以下规则：

- 移动到更高的异常级别，例如从 EL0 到 EL1，表示软件增加执行特权。
- 不能将异常处理到较低的异常级别。
- EL0 级别没有异常处理，必须在更高的异常级别处理异常
- 异常导致程序流程发生变化。异常处理程序的执行以高于 EL0 的异常级别从与所采取的异常相关的已定义向量开始。例外情况包括：IRQ 和 FIQ 等中断。内存系统中止。未定义的指令。系统调用。这些允许非特权软件对操作系统安全监视器或管理程序陷阱。
- 通过执行 ERET 指令来结束异常处理并返回到上一个异常级别。
- 从异常返回可以保持相同的异常级别或进入较低的异常级别。它不能移动到更高的异常级别。
- 安全状态确实会随着异常级别的变化而变化，除非从 EL3 重新调整到非安全状态

3.3 3.3 改变执行状态

有时您必须更改系统的执行状态。例如，如果您正在运行 64 位操作系统，并且希望在 EL0 上运行 32 位应用程序，则可能是这样。为此，系统必须更改为 AArch32。

当应用程序完成或执行返回操作系统时，系统可以切换回 AArch64。第 3-9 页上的图 3-7 表明您不能反过来做。AArch32 操作系统不能承载 64 位应用程序

要在相同的异常级别之间切换执行状态，您必须切换到更高的异常级别，然后返回到原始的异常级别。例如，您可能有 32 位和 64 位应用程序在 64 位操作系统下运行。在这种情况下，32 位应用程序可以执行并生成主管调用 (SVC) 指令，或接收中断，从而导致切换到 EL1 和 AArch64。（请参阅第 6-21 页的异常处理说明。）然后操作系统可以执行任务切换并返回到 AArch64 中的 EL0。实际上，这意味着您不能拥有混合的 32 位和 64 位应用程序，因为它们之间没有直接的调用方式。

您只能通过更改异常级别来更改执行状态。发生异常可能会从 AArch32 更改为 AArch64，从异常返回可能会从 AArch64 更改为 AArch32。

EL3 的代码无法将异常提升到更高的异常级别，因此无法更改执行状态，除非通过重置。

以下是在 AArch64 和 AArch64 和 AArch32 执行状态：

- AArch64 和 AArch32 执行状态都具有大致相似的异常级别，但安全和非安全操作之间存在一些差异。产生异常时处理器所处的执行状态可以限制其他执行状态可用的异常级别。
- 更改为 AArch32 需要从较高的异常级别转到较低的异常级别。这是通过执行 ERET 指令退出异常处理程序的结果。请参阅第 6-21 页的异常处理说明。
- 更改为 AArch64 需要从较低的异常级别转到较高的异常级别。异常可能是指令执行或外部信号的结果。
- 如果在发生异常或从异常返回时，异常级别保持不变，则执行状态不能改变。
- ARMv8 处理器在特定的 AArch32 执行状态下运行异常级别，它使用与 ARMv7 中相同的异常模型来处理该异常级别的异常。在 AArch64 执行状态下，它使用第 10 章 AArch64 异常处理中描述的异常处理模型。

因此，这两种状态之间的互通是在安全监视器、管理程序或操作系统的级别上执行的。在 AArch64 状态下执行的管理程序或操作系统可以支持较低权限级别的 AArch32 操作。这意味着在 AArch64 中运行的操作系统可以同时托管 AArch32 和 AArch64 应用程序。同样，AArch64 管理程序可以同时托管 AArch32 和 AArch64 客户操作系统。但是，32 位操作系统不能托管 64 位应用程序，32 位虚拟机管理程序不能托管 64 位客户操作系统。

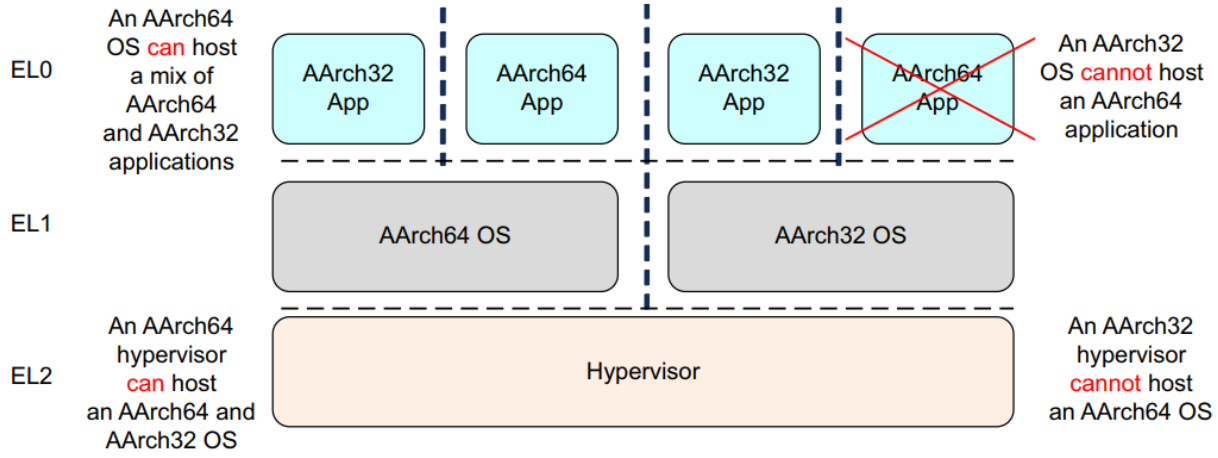

image-20220314224724121

对于最高实现的异常级别（Cortex-A53 和 Cortex-A57 处理器上的 EL3），在接受异常时用于每个异常级别的执行状态是固定的。异常级别只能通过重置处理器来更改。对于 EL2 和 EL1，它由第 4-7 页的系统寄存器控制。

CHAPTER

FOUR

4. ARMV8 寄存器

AArch64 执行状态提供了 32 个在任何时间任何特权级下都可访问的 64 位的通用寄存器。

每个寄存器都有 64 位宽，它们通常被称为寄存器 X0-X30。

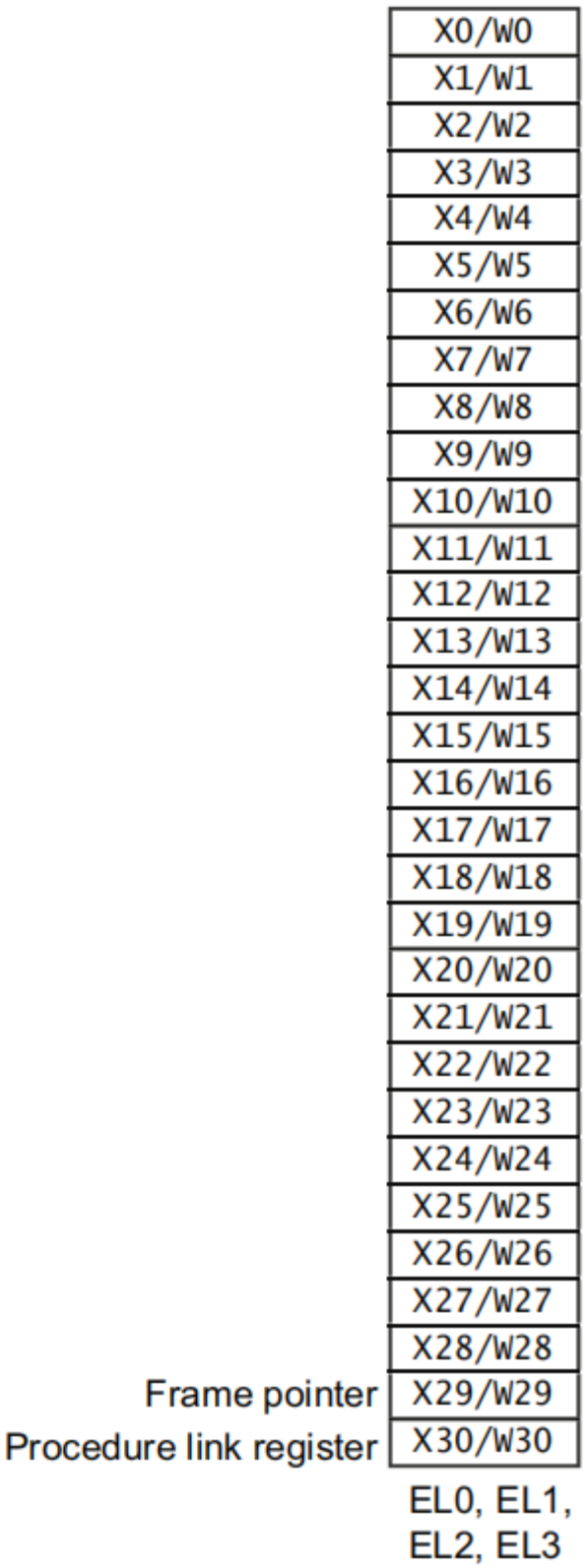

Figure 4-1 AArch64 general-purpose registers image-通用寄存器

每个 AArch64 64 位通用寄存器（X0-X30）也具有 32 位（W0-W30）形式。

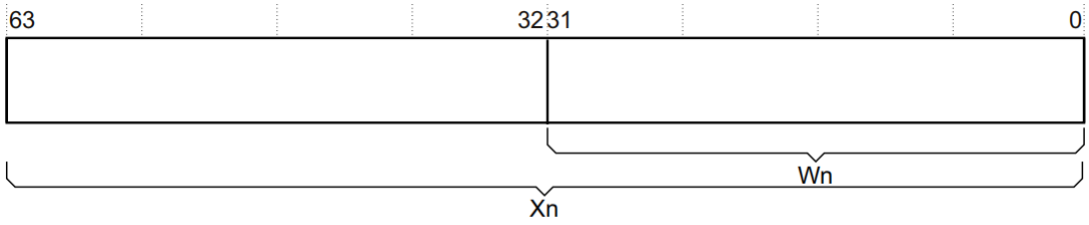

Figure 4-2 64-bit register with W and X access. image-通用寄存器

32 位 W 寄存器取自相应的 64 位 X 寄存器的低 32 位。也就是说，W0 映射到 X0 的低 32 位，W1 映射到 X1 的低 32 位。

从 W 寄存器读取时，忽略相应 X 寄存器高 32 位，并保持其它不变。写入 W 寄存器时，将 X 寄存器的高 32 位设置为零。也就是说，将 0xFFFFFFFF 写入 W0 会将 X0 设置为 0x00000000FFFFFFFF。

4.1 4.1 AArch64 特殊寄存器

除了 31 个核心寄存器外，还有几个特殊的寄存器。

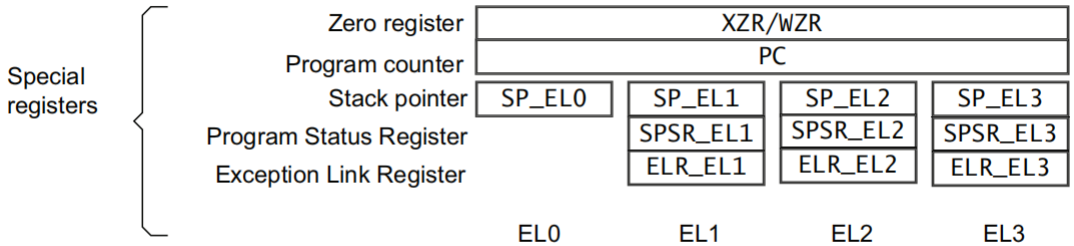

Figure 4-3 AArch64 special registers image-04_s_registers

> 注意：没有被称为 X31 或 W31 的寄存器。许多指令被编码，例如：31 代表零寄存器，ZR(WZR/XZR)。
> 还有一组受限制的指令，其中对一个或多个参数进行编码，使数字 31 表示堆栈指针 (SP)。

当访问零寄存器时，所有写操作都被忽略，所有读操作返回 0。请注意，64 位形式的 SP 寄存器不使用 X 前缀。

Table 4-1 Special registers in AArch64

Name	Size	Description
WZR	32 bits	Zero register
XZR	64 bits	Zero register
WSP	32 bits	Current stack pointer
SP	64 bits	Current stack pointer
PC	64 bits	Program counter

image-04_s_registers1

在 ARMv8 体系结构中，当 CPU 运行在 AArch64 状态时，异常返回状态保存在每个异常级别的以下专用寄存器中：

- *Exception Link Register* (ELR).
- *Saved Processor State Register* (SPSR).

每个异常级别都有一个专用的 SP 寄存器，但它不用于保存返回状态。

Table 4-2 Special registers by Exception level

	EL0	EL1	EL2	EL3
Stack Pointer (SP)	SP_EL0	SP_EL1	SP_EL2	SP_EL3
Exception Link Register (ELR)		ELR_EL1	ELR_EL2	ELR_EL3
Saved Process Status Register (SPSR)		SPSR_EL1	SPSR_EL2	SPSR_EL3

image-04_s_registers_el

4.1.1 4.1.1 零寄存器

零寄存器当用作源寄存器时读操作的结果为零，当用作目标寄存器时则将结果丢弃。你可以在大多数指令中但不是所有指令中使用零寄存器。

4.1.2 4.1.2 栈指针

在 ARMv8 体系结构中，要使用的栈指针的选择在一定程度上与异常级别是分开的。默认情况下，发生异常时会选择目标异常级别的 SP_ELn 作为栈指针。例如，当触发到 EL1 的异常时，就会选择 SP_EL1 作为栈指针。每个异常级别都有自己的栈指针，SP_EL0、SP_EL1、SP_EL2 和 SP_EL3。

当 AArch64 处于 EL0 以外的异常级别时，处理器可以使用：

- 与该异常级别相关联的一个专用的 64 位栈指针 (SP_ELn)
- 与 EL0 关联的栈指针 (SP_EL0)

EL0 永远只能访问 SP_EL0。

Table 4-3 AArch64 Stack pointer options

Exception level	Options
EL0	EL0*t*
EL1	EL1*t*, EL1*h*
EL2	EL2*t*, EL2*h*
EL3	EL3*t*, EL3*h*

image-04_registers_sp

t 后缀表示选择了 SP_EL0 栈指针。h 后缀表示选择了 SP_ELn 栈指针。

虽然大多数指令都无法使用 SP 寄存器。但是有一些形式的算术指令可以操作 SP，例如，ADD 指令可以读写当前的栈指针以调整函数中的栈指针。例如：

```
ADD SP, SP, #0x10 // Adjust SP to be 0x10 bytes before its current value
```

4.1.3 4.1.3 程序计数器

原来的 ARMv7 指令集的一个特性是 R15 作为程序计数器 (PC)，并作为一个通用寄存器使用。PC 寄存器的使用带来了一些编程技巧，但它为编译器和复杂的流水线的设计引入了复杂性。在 ARMv8 中删除了对 PC 的直接访问，使返回预测更容易，并简化了 ABI 规范。

PC 永远不能作为一个命名的寄存器来访问。但是，可以在某些指令中隐式的使用 PC，如 PC 相对加载和地址生成。PC 不能被指定为数据处理或加载指令的目的操作数。

4.1.4 4.1.4 异常链接寄存器（ELR）

异常链接寄存器保存异常返回地址。

4.1.5 4.1.5 程序状态保存寄存器（SPSR）

当异常发生时，CPSR 中的处理器状态将保存在相关的程序状态保存寄存器 (SPSR) 中，其方式类似于 ARMv7。SPSR 保存着异常发生之前的 PSTATE 的值，用于在异常返回时恢复 PSTATE 的值。

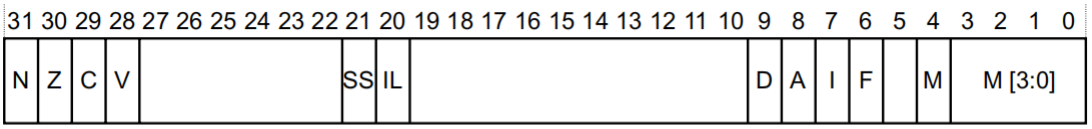

Figure 4-4 SPSR

image-04_registers_spsr

AArch64 下各 bit 的含义：

N 负数标志位，如果结果为负数，则 N=1；如果结果为非负数，则 N=0。

Z 零标志位，如果结果为零，Z=1，否则 Z=0。

C 进位标志位

V 溢出标志位

SS 软件步进标志位，表示当一个异常发生时，软件步进是否开启

IL 非法执行状态位

D 程序状态调试掩码，在异常发生时的异常级别下，来自监视点、断点和软件单步调试事件中的调试异常是否被屏蔽。

A SError（系统错误）掩码位

I IRQ 掩码位

F FIQ 掩码位

M[4] 异常发生时的执行状态，0 表示 AArch64

M[3:0] 异常发生时的 mode 或异常级别

在 ARMv8 中，写入的 SPSR 依赖于异常级别。如果异常发生在 EL1，则使用 SPSR_EL1。如果异常发生在 EL2，则使用 SPSR_EL2，如果异常发生在 EL3，则使用 SPSR_EL3。处理器核会在异常发生时填充 SPSR。

> 注意：与异常级别相关联的寄存器对 ELR_ELn 和 SPSR_ELn 保存着在较低异常级别执行期间的状态。

4.2 4.2 处理器状态

AArch64 没有直接与 ARMv7 当前程序状态寄存器 (CPSR) 等价的寄存器。在 AArch64 中，传统 CPSR 的组件作为可以独立访问的字段提供。这些状态被统称为处理器状态 (PSTATE)。

AArch64 的处理器状态或 PSTATE 字段有以下定义：

Table 4-4 PSTATE field definitions

Name	Description
N	Negative condition flag.
Z	Zero condition flag.
C	Carry condition flag.
V	oVerflow condition flag.
D	Debug mask bit.
A	SError mask bit.
I	IRQ mask bit.
F	FIQ mask bit.
SS	Software Step bit.
IL	Illegal execution state bit.
EL (2)	Exception level.
nRW	Execution state 0 = 64-bit 1 = 32-bit
SP	Stack Pointer selector. 0 = SP_EL0 1 = SP_EL*n*

image-04_pstate

在 AArch64 中，你可以通过执行 ERET 指令从一个异常中返回，这将导致 SPSR_ELn 被复制到 PSTATE 中。这将恢复 ALU 标志、执行状态、异常级别和处理器分支。从这里开始，将继续从 ELR_ELn 中的地址开始执行。

PSTATE.{N, Z, C, V} 字段可以在 EL0 级别访问。其他的字段可以在 EL1 或更高级别访问，但是这些字段在 EL0 级别未定义。

4.3 4.3 系统寄存器

在 AArch64 中，系统配置通过系统寄存器进行控制，并使用 MSR 和 MRS 指令进行访问。这与 ARMv7-A 形成了鲜明对比，在 ARMv7-A 中，这些寄存器通常通过协处理器 15(CP15) 操作来访问。寄存器的名称会告诉你可以访问它的最低异常级别。

例如：

- TTBR0_EL1 可以从 EL1、EL2 和 EL3 访问
- TTBR0_EL2 可以从 EL2 和 EL3 访问

具有后缀 _ELn 的系统寄存器在各异常级别中有一个独立的的副本，但通常不是 EL0，很少有系统寄存器可

以从 EL0 访问，尽管缓存类型寄存器 (CTR_EL0) 是一个可以从 EL0 访问的系统寄存器的例子。

可以采用以下形式来访问系统寄存器：

```
MRS x0, TTBR0_EL1 // Move TTBR0_EL1 into x0
MSR TTBR0_EL1, x0 // Move x0 into TTBR0_EL1
```

ARM 体系结构以前的版本使用协处理器来进行系统配置。但是，AArch64 并不包含对协处理器的支持。表 4-5 仅列出了本书中提到的系统寄存器。

有关完整的列表，请参见 ARM 体系结构参考手册-ARMv8 的附录 J，以了解 ARMv8-A 体系结构配置文件。

下表展示了具有独立副本的寄存器异常级别。例如，辅助控制寄存器 (ACTLRs) 在各异常级别作为 ACTLR_EL1、ACTLR_EL2 和 ACTLR_EL3 存在。

Table 4-5 System registers

Name	Register	Description	Allowed values of n
ACTLR_EL*n*	Auxiliary Control Register	Controls processor-specific features.	1, 2, 3
CCSIDR_EL*n*	Current Cache Size ID Register	Provides information about the architecture of the currently selected cache. See *Cache discovery* on page 11-18.	1
CLIDR_EL*n*	Cache Level ID Register	The type of cache, or caches, implemented at each level. The Level of Coherency and Level of Unification for the cache hierarchy. See *Cache maintenance* on page 11-13.	1, 2, 3
CNTFRQ_EL*n*	Counter-timer Frequency Register	Reports the frequency of the system timer. See *Timers* on page 14-5.	0
CNTPCT_EL*n*	Counter-timer Physical Count Register	Holds the 64-bit current count value. See *Timers* on page 14-5.	0
CNTKCTL_EL*n*	Counter-timer Kernel Control Register	Controls the generation of an event stream from the virtual counter. Also controls access from EL0 to the physical counter, virtual counter, EL1 physical timers, and the virtual timer. See *Timers* on page 14-5.	1

image-04_system_registers

Name	Register	Description	Allowed values of n
CNTP_CVAL_EL*n*	Counter-timer Physical Timer Compare Value Register	Holds the compare value for the EL1 physical timer. See *Timers* on page 14-5.	0
CPACR_EL*n*	Coprocessor Access Control Register	Controls access to Trace, floating-point, and NEON functionality. See *ISB in more detail* on page 13-9.	1
CSSELR_EL*n*	Cache Size Selection Register	Selects the current Cache Size ID Register, CCSIDR_EL1, by specifying the required cache level and the cache type, either instruction or data cache. See *Cache discovery* on page 11-18.	1
CNTP_CTL_EL*n*	Counter-timer Physical Control Register	Control register for the EL1 physical timer. See *Timers* on page 14-5.	0
CTR_EL*n*	Cache Type Register	Information about the architecture of the integrated caches. See *Cache discovery* on page 11-18.	0
DCZID_EL*n*	Data Cache Zero ID Register	Indicates the block size written with byte values of 0 by the *Data Cache Zero by Virtual Address* (DCZVA) system instruction. See *Cache discovery* on page 11-18.	0
ELR_EL*n*	Exception Link Register	Holds the address of the instruction which caused the exception.	1, 2, 3
ESR_EL*n*	Exception Syndrome Register	Includes information about the reasons for the exception. See *The Exception Syndrome Register* on page 10-9.	1, 2, 3
FAR_EL*n*	Fault Address Register	Holds the virtual faulting address. See *Handling synchronous exceptions* on page 10-7.	1, 2, 3
FPCR	Floating-point Control Register	Controls floating-point extension behavior. The fields in this register map to the equivalent fields in the AArch32 FPSCR. See *New features for NEON and Floating-point in AArch64* on page 7-2.	-
FPSR	Floating-point Status Register	Provides floating-point system status information. The fields in this register map to the equivalent fields in the AArch32 FPSCR. See *New features for NEON and Floating-point in AArch64* on page 7-2.	-
HCR_EL*n*	Hypervisor Configuration Register	Controls virtualization settings and trapping of exceptions to EL2. See *Exception handling on page 18-8*.	2
MAIR_EL*n*	Memory Attribute Indirection Register	Provides the memory attribute encodings corresponding to the possible values in a Long-descriptor format translation table entry for stage 1 translations at EL*n*. See *Memory types* on page 13-3.	1, 2, 3
MIDR_EL*n*	Main ID Register	The type of processor the code is running on (part number and revision).	1
MPIDR_EL*n*	Multiprocessor Affinity Register	The processor and cluster IDs, in multi-core or cluster systems. See *Determining which core the code is running on* on page 14-3.	1

image-04_system_register1

Name	Register	Description	Allowed values of n
SCR_EL*n*	Secure Configuration Register	Controls Secure state and trapping of exceptions to EL3. See *Handling synchronous exceptions* on page 10-7.	3
SCTLR_EL*n*	System Control Register	Controls architectural features, for example the MMU, caches and alignment checking.	0, 1, 2, 3
SPSR_EL*n*	Saved Program Status Register	Holds the saved processor state when an exception is taken to this mode or Exception level.	abt, fiq, irq, und, 1,2, 3
TCR_EL*n*	Translation Control Register	Determines which of the Translation Table Base Registers define the base address for a translation table walk required for the stage 1 translation of a memory access from EL*n*. Also controls the translation table format and holds cacheability and shareability information. See *Separation of kernel and application Virtual Address spaces* on page 12-7.	1, 2, 3
TPIDR_EL*n*	User Read/Write Thread ID Register	Provides a location where software executing at EL*n* can store thread identifying information, for OS management purposes. See *Context switching* on page 12-27.	0, 1, 2, 3
TPIDRRO_EL*n*	User Read-Only Thread ID Register	Provides a location where software executing at EL1 or higher can store thread identifying information. This informaton is visible to software executing at EL0, for OS management purposes. See *Context switching* on page 12-27.	0
TTBR0_EL*n*	Translation Table Base Register 0	Holds the base address of translation table 0, and information about the memory it occupies. This is one of the translation tables for the stage 1 translation of memory accesses at EL*n*. See *Separation of kernel and application Virtual Address spaces* on page 12-7.	1, 2, 3
TTBR1_EL*n*	Translation Table Base Register 1	Holds the base address of translation table 1, and information about the memory it occupies. This is one of the translation tables for the stage 1 translation of memory accesses at EL0 and EL1. See *Separation of kernel and application Virtual Address spaces* on page 12-7.	1
VBAR_EL*n*	Vector Based Address Register	Holds the exception base address for any exception that is taken to EL*n*. See *AArch64 exception table* on page 10-12.	1, 2, 3
VTCR_EL*n*	Virtualization Translation Control Register	Controls the translation table walks required for the stage 2 translation of memory accesses from Non-secure EL0 and EL1. Also holds cacheability and shareability information for the accesses. See *Translations at EL2 and EL3* on page 12-20.	2
VTTBR_EL*n*	Virtualization Translation Table Base Register	Holds the base address of the translation table for the stage 2 translation of memory accesses from Non-secure EL0 and EL1. See *Memory translation on page 18-3*.	2

image-04_system_register2

4.3.1 4.3.1 系统控制寄存器

系统控制寄存器 (SCTLR) 是一个用来控制标准内存、配置系统能力、提供处理器核状态信息的寄存器。

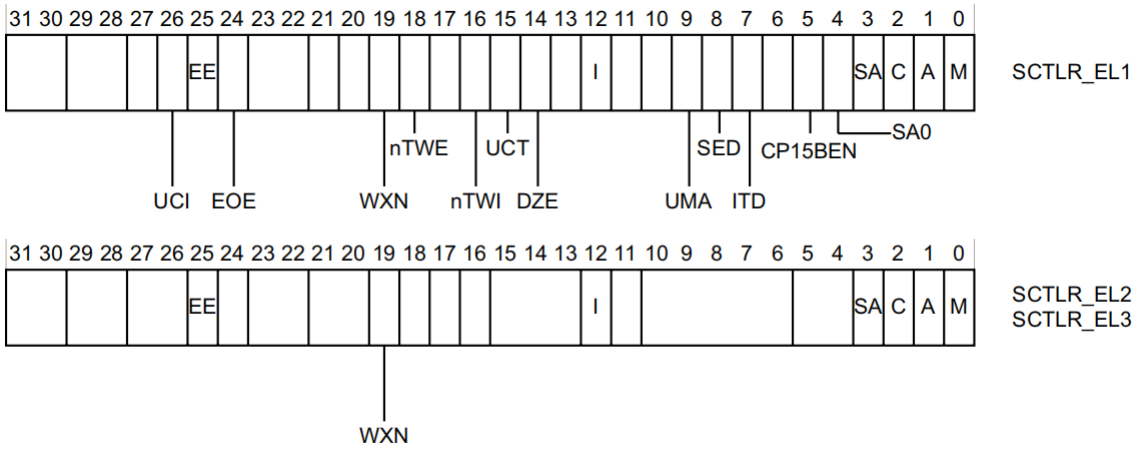

image-04_control_registers

并不是所有 bit 在 EL1 都可用，各 bit 的含义如下：

UCI 设置此位后，在 AArch64 中为 DC CVAU、DC CIVAC、DC CVAC 和 IC IVAU 指令启用 EL0 访问。

EE 异常字节顺序

0 小端

1 大端

EOE EL0 显式数据访问的字节序

0 小端

1 大端

WXN 写权限不可执行 XN (eXecute Never)。See *Access permissions* on page 12-23.

0 可写区域不设置不可执行权限（XN）

1 可写区域强制为不可执行 (XN)

nTWE 不陷入 WFE，此标志为 1 表示 WFE 作为普通指令执行

nTWI 不陷入 WFI, 此标志为 1 表示 WFI 作为普通指令执行

UCT 此标志为 1 时，开启 AArch64 的 EL0 下访问 CTR_EL0 寄存器

DNE EL0 下访问 DC AVA 指令，0 禁止执行，1 允许执行

I 开启指令缓存，这是在 EL0 和 EL1 下的指令缓存的启用位。对可缓存的正常内存的指令访问被缓存。

UMA 用户屏蔽访问。当 EL0 使用 AArch64，控制从 EL0 的中断屏蔽访问。

SED 禁止 SETEND。在 EL0 使用 AArch32 禁止 SETEND 指令。0 使能，1 禁止

ITD 禁止 IT 指令

0 IT 指令有效

1 IT 指令被当作 16 位指令。仅另外 16 位指令或 32 位指令的头 16 位可以使用，这依赖于实现

CP15BEN CP15 barrier 使能。如果实现了，它是 AArch32 CP15 DMB,DSB 和 ISB barrier 操作的使能位

SA0 EL0 的栈对齐检查使能位

**SA ** 栈对齐检查使能位

C 数据 cache 使能。EL0 和 EL1 的数据访问使能位。对 cacheable 普通内存的数据访问都被缓存

A 对齐检查使能位

M 使能 MMU

对 SCTLR 的访问

为访问 SCTLR_ELn，使用：

```
MRS <Xt>, SCTLR_ELn // Read SCTLR_ELn into Xt
MSR SCTLR_ELn, <Xt> // Write Xt to SCTLR_ELn
```

例如：

Example 4-1 Setting bits in the SCTLR

```
MRS X0, SCTLR_EL1          // Read System Control Register configuration data
ORR X0, X0, #(1 << 2)      // Set [C] bit and enable data caching
ORR X0, X0, #(1 << 12)     // Set [I] bit and enable instruction caching
MSR SCTLR_EL1, X0          // Write System Control Register configuration data
```

image-04_setting_sctlr

注意：处理器中各异常级别的 cache 必须在数据和指令的 cache 使能之前被无效化

4.4 4.4 字节序

在内存中查看字节有两种基本的方式，一种是小端 (LE)，另一种是大端 (BE)。在大端机器上，内存中对象中高字节存储在低地址（即接近于零的地址）。在小端机器上，低字节存储在低地址。术语 byte-ordering 也可以用来代替大小端。

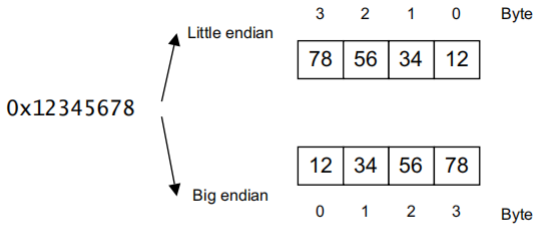

Figure 4-6 image-04_endianness

每个异常级别的数据的大小端都被单独控制。对于 EL3，EL2 和 EL1，通过 SCTLR_ELn.EE 设置大小端。EL1 中其他位，SCTLR_EL1.E0E 控制 EL0 的数据大小端的设置。在 AArch64 执行状态中，数据访问可以为 LE 或 BE，但指令的获取通常为 LE。

处理器是否支持 LE 和 BE 取决于处理器的实现。如果只支持小端，则 EE 位和 E0E 位始终为 0。类似地，如果仅支持大端，则 EE 位和 E0E 位被设置为 1。

当使用 AArch32 时，CPSR.E 位与相对应系统控制寄存器中 EE 位不同，该位在 EL1，EL2 和 EL3 被弃用。ARMv7 中 SETEND 指令也被弃用，它通过设置 SCTLR.SED 位在执行 SETEND 指令时可能导致未定义的异常。

4.5 4.5 改变执行状态（再次）

在第 3-8 页的更改执行状态时，我们从异常级别角度描述了 AArch64 和 AArch32 之间的变化。现在我们从寄存器的角度来考虑这个变化。

从 AArch32 的异常级别进入 AArch64 的异常级别时：

- 对于 AArch32 下访问任何异常级别的高 32 位寄存器的值的行为都是未知的。
- 在 AArch32 执行期间不可访问保留了它们在 AArch32 执行之前的状态的寄存器。
- 当 EL2 使用 AArch32 触发一个到 EL3 的异常时，ELR_EL2 的高 32 位是未知的。
- AArch32 不可以访问 AArch64 各异常级别的栈指针（SPs）和异常链接寄存器（ELRs），这些寄存器保存着它们在 AArch32 执行之前的状态: SP_EL0、SP_EL1 、SP_EL2 、ELR_EL1

通常，应用程序程序员可以在 AArch32 或 AArch64 下编写应用程序。只有操作系统必须考虑到这两个执行状态和它们之间的切换。

4.5.1 4.5.1 AArch32 下的寄存器

兼容 ARMv7 意味着 AArch32 必须匹配 ARMv7 特权级别。这也意味着 AArch32 只处理 ARMv7 32 位通用寄存器。因此，ARMv8 体系结构与 AArch32 执行状态提供的视图之间必须有一些对应关系。

请记住，在 ARMv7 体系结构中有 16 个 32 位通用寄存器 (R0-R15) 用于软件使用。其中 15 个（R0-R14）可用于通用数据存储。剩下的寄存器 R15 是程序计数器 (PC)，其值在处理器核心执行指令时被改变。软件还可以访问 CPSR，而从之前执行的模式中保存的 CPSR 是 SPSR。在发生异常时，CPSR 将被复制到发生异常的模式的 SPSR 中。

访问这些寄存器中的哪一个，取决于软件正在执行的处理器模式和寄存器本身，这称为 banking，第 4-14 页图 4-7 中的阴影寄存器称为 banked。它们使用不同的物理存储，通常程序只有在特定模式下执行时才能访问。

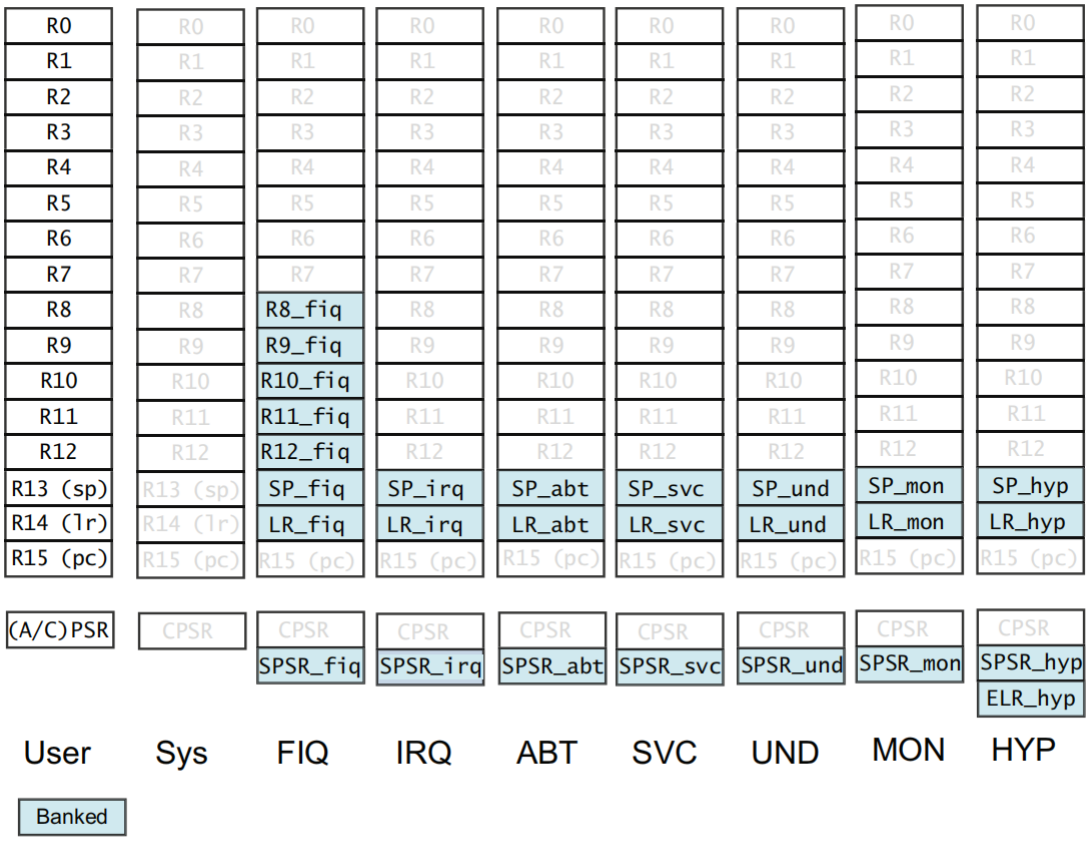

Figure 4-7 The ARMv7 register set showing banked registers image-04_banked_registers

在 ARMv7 中使用了 banking 来减少异常的延迟。然而，这也意味着在相当数量的寄存器中，任何时候可以使用的都不到一半。

相比之下，AArch64 执行状态有 31 个 64 位通用寄存器，并且可以在所有异常级别中随时访问。在 AArch64 和 AArch32 之间的执行状态的切换意味着 AArch64 寄存器必须映射到 AArch32(ARMv7) 寄存器集。此映射如图 4-8 所示。

当在 AArch32 状态下执行时，AArch64 寄存器的高 32 位是不可访问的。如果处理器在 AArch32 状态下工作，它将使用 32 位 W 寄存器，这相当于 32 位的 ARMv7 寄存器。

AArch32 需要将 banked 寄存器映射到 AArch64 寄存器，否则这些寄存器将无法访问。

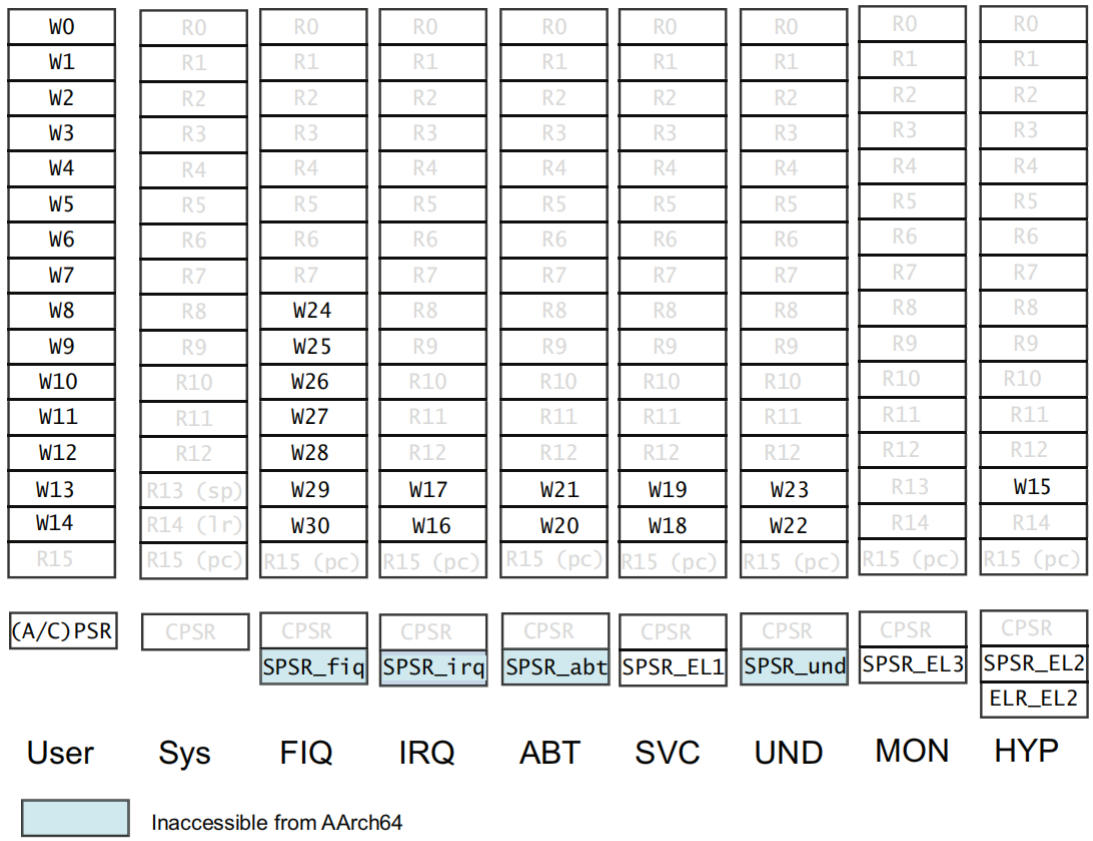

Figure 4-8 AArch64 to AArch32 register mapping image-c_aarch32_register

AArch32 中的 SPSR 和 ELR_Hyp 寄存器是只能被系统指令访问的附加寄存器。它们没有被映射到 AArch64 体系结构的通用寄存器空间中。下面这些寄存器在 AArch32 和 AArch64 之间进行映射：

- SPSR_svc 映射到 SPSR_EL1
- SPSR_hyp 映射到 SPSR_EL2
- ELR_hyp 映射到 ELR_EL2

以下寄存器仅在 AArch32 执行期间使用。由于在 EL1 上使用 AArch64 执行，所以，尽管在 AArch64 执行期间这些寄存器不可访问，但仍然保留了它们的状态。

- SPSR_abt
- SPSR_und
- SPSR_irq
- SPSR_fiq

SPSR 寄存器只能在 AArch64 状态用于上下文切换的高异常级别下可访问。

同样，如果在 AArch32 的异常级别触发一个到 AArch64 异常级别的异常，则 AArch64 ELR_ELn 的高 32 位都为零。

4.5.2 4.5.2 AArch32 下的 PSTATE

在 AArch64 中，传统 CPSR 的不同组件被表示为可以独立访问的处理器状态 (PSTATE) 字段。在 AArch32，有与 ARMv7 CPSR 位对应的附加字段。

提供只能在 AArch32 上访问的附加 PSTATE 位：

Table 4-6 PSTATE bit definitions

Name	Description
Q	Cumulative saturation (*sticky*) flag.
GE (4)	Greater than or Equal flags.
IT (8)	If-Then execution bits.
J	J bit.
T	T32 bit.
E	Endianness bit.
M	Mode field.

image-04_pstate_

4.6 4.6 NEON 和浮点寄存器

除了通用寄存器之外，ARMv8 还有 32 个 128 位浮点寄存器，标记为 V0-V31。32 个寄存器用于保存标量浮点指令的浮点操作数，以及 NEON 操作的标量操作数和向量操作数。NEON 和浮点寄存器也在第 7 章 *AArch64 Floating-point and NEON* 也会介绍。

4.6.1 4.6.1 AArch64 中浮点寄存器的组织

在对标量数据进行操作的 NEON 指令和浮点指令中，浮点和 NEON 寄存器的行为类似于主要的通用整数寄存器。因此，只访问较低的位，在读时忽略未使用的高位，在写时将其设置为零。标量浮点和 NEON 名称的限定名表示有效位的数目，如下所示，其中 n 是寄存器号 0-31。

Table 4-7 Operand name for differently sized floats

Precision	Size (bits)	Name
Half	16	Hn
Single	32	Sn
Double	64	Dn

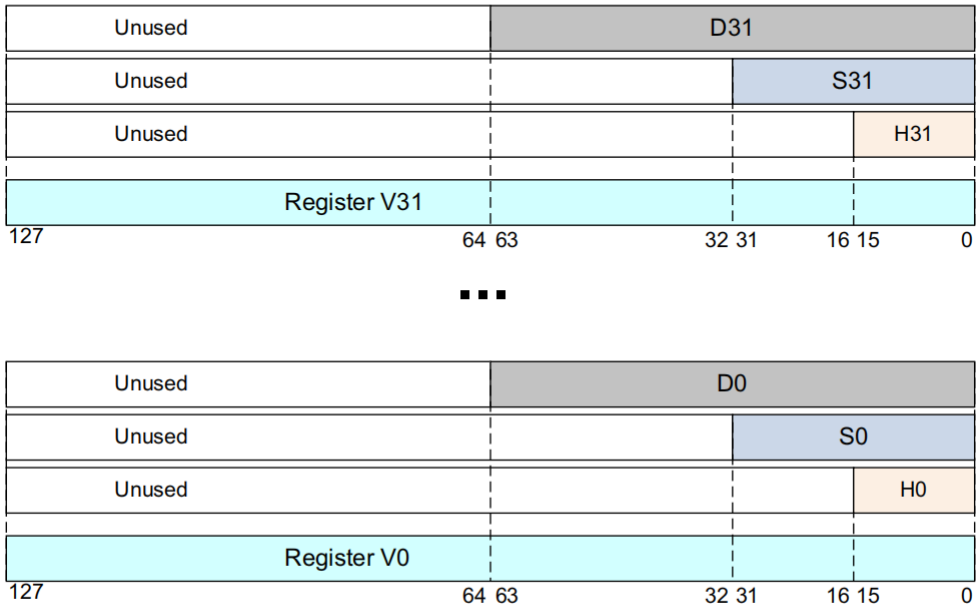

Figure 4-10 Arrangement of floating-point values image-04_floating_point

注意：支持 16 位浮点数，但仅作为要转换的格式。不支持数据处理操作。（**译注**：即不能直接对 16 位浮点数据进行操作）

前缀 F 和浮点数大小由浮点 ADD 指令指定:

```
FADD Sd, Sn, Sm // Single-precision
FADD Dd, Dn, Dm // Double-precision
```

半精度浮点指令用于在不同大小之间进行转换：

```
FCVT Sd, Hn // half-precision to single-precision
FCVT Dd, Hn // half-precision to double-precision
FCVT Hd, Sn // single-precision to half-precision
FCVT Hd, Dn // double-precision to half-precision
```

4.6.2 4.6.2 标量寄存器大小

在 AArch64 中，整数标量的映射已经从 ARMv7-A 中使用的映射更改为图 4-11 所示的映射

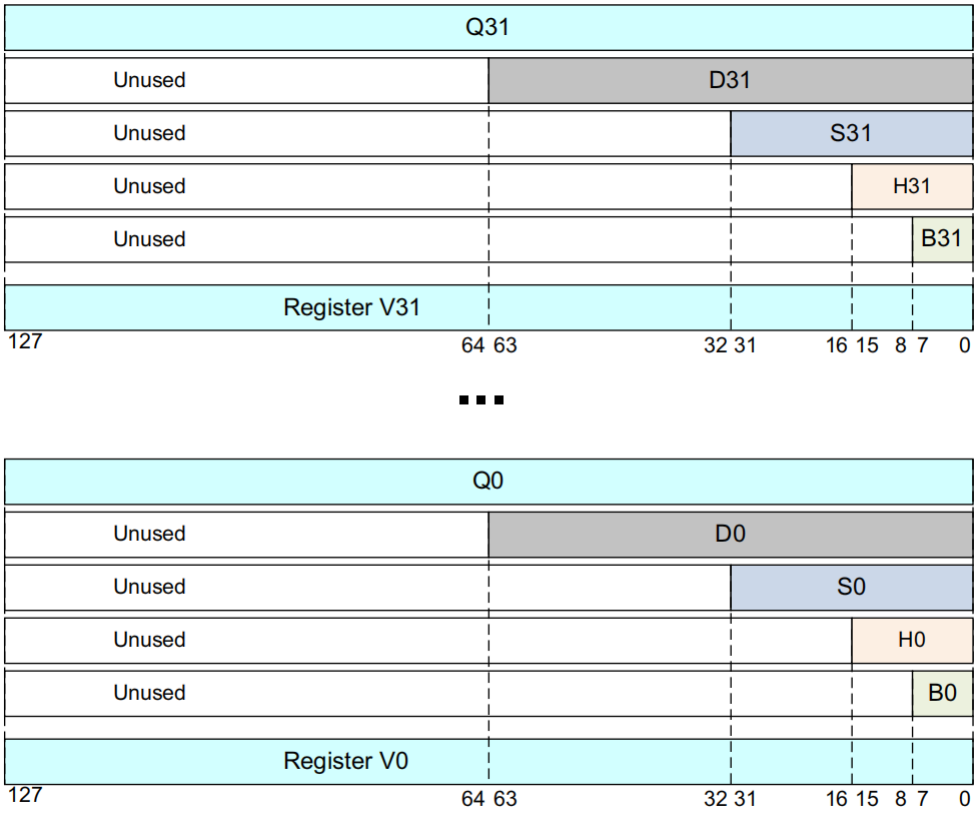

Figure 4-11 Arrangement of ARMv8 registers when holding scalar values image-04_scalar_register_size

图 4-11 S0 是 D0 的低半部分，D0 是 Q0 的低半部分。S1 是 D1 的低半部分，D1 同样是 Q1 的低半部分，以此类推。这消除了编译器在自动向量化高级代码时遇到的许多问题。

- 每个 Q 寄存器的低 64 位也可以看作是 D0-D31，32 个给浮点和 NEON 计算使用的 64 位宽寄存器。
- 每个 Q 寄存器的低 32 位也可以看作是 S0-S31，32 个给浮点和 NEON 计算使用的 32 位宽寄存器。
- 每个 S 寄存器的低 16 位也可以看作是 H0-H31，32 个给浮点和 NEON 计算使用的 16 位宽寄存器。
- 每个 H 寄存器的低 8 位也可以看作是 B0-B31, 32 个供 NEON 使用的 8 位宽寄存器。

  注意：在每种情况下只使用每个寄存器集的底部的位。读取时忽略寄存器空间的其余部分，写入时用 0 填充其余部分。

这种映射的结果是，如果在 AArch64 中执行的程序正在使用来自 AArch32 的 D 或 S 寄存器。那么，在使用 D 或 S 寄存器之前，程序必须将它们从 V 寄存器中解包。

对于标量 ADD 指令：

```
ADD Vd, Vn, Vm
```

例如，如果大小为 32 位，则指令为：

```
ADD Sd, Sn, Sm
```

Table 4-8 Operand name for differently sized scalars

Word size	Size (bits)	Name
Byte	8	Bn
Halfword	16	Hn
Word	32	Sn
Doubleword	64	Dn
Quadword	128	Qn

image-04_scalar_register_size1

4.6.3 4.6.3 向量寄存器大小

向量的大小为 64 位，有一个或多个元素。也可以为 128 位，有两个或多个元素。如图 4-12 所示:

对于向量 ADD 指令：

```
ADD Vd.T, Vn.T, Vm.T
```

这里对于 32 位向量，有 4 个 lanes（通道），指令变为：

```
ADD Vd.4S, Vn.4S, Vm.4S
```

Table 4-9 Operand names for different size vectors

Name	Shape
Vn.8B	8 lanes, each containing an 8-bit element
Vn.16B	16 lanes, each containing an 8-bit element
Vn.4H	4 lanes, each containing a 16-bit element
Vn.8H	8 lanes, each containing a 16-bit element
Vn.2S	2 lanes, each containing a 32-bit element
Vn.4S	4 lanes, each containing a 32-bit element
Vn.1D	1 lane containing a 64-bit element
Vn.2D	2 lanes, each containing a 64-bit element

image-04_operand_names

当这些寄存器以特定的指令形式使用时，必须进一步限定名称以指示数据形状。更具体地说，这意味着数据元素的大小和其中包含的元素或通道的数量。

4.6.4 4.6.4 AArch32 执行状态的 NEON

在 AArch32 中，较小的寄存器被打包成较大的寄存器 (比如 D0 和 D1 合并成 Q1)。这引入了一些复杂的循环迭代依赖关系，这会降低编译器向量化循环结构的能力。

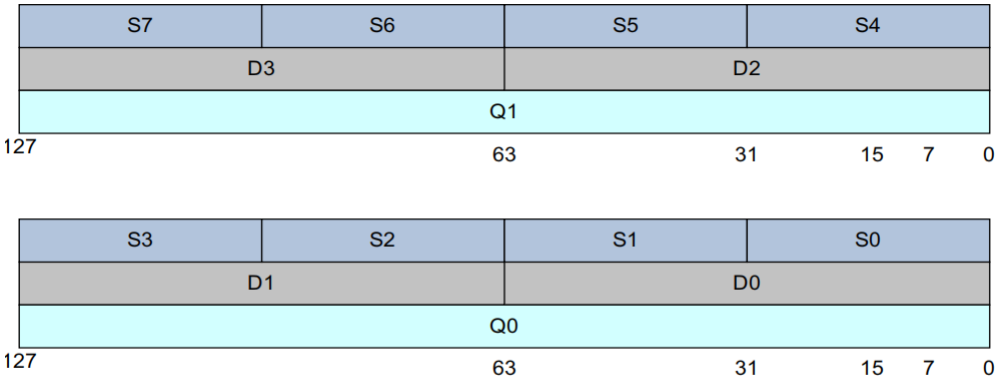

Figure 4-13 Arrangement of ARMv7 SIMD registers image-04_SIMD_registers

AArch32 中的浮点寄存器和高级 SIMD 寄存器被映射到 AArch64 FP 和 SIMD 寄存器。这样做是为了允许应用程序或虚拟机的浮点和 NEON 寄存器由更高级别的系统软件 (例如 OS 或 Hypervisor) 解释 (并在必要时修改)。

AArch64 V16-V31 FP 和 NEON 寄存器不能从 AArch32 访问。与通用寄存器一样，运行在 AArch32 中的异常级别时，这些寄存器保留了 AArch64 执行前的状态。

CHAPTER

FIVE

5. ARMV8 指令集简介

ARMv8 架构中引入的最重要的变化之一是增加了 64 位指令集。该指令集补充了现有的 32 位指令集架构。这一指令集提供了对 64 位宽整数寄存器和数据操作的访问，以及使用 64 位内存指针的能力。新的指令集被称为 **A64**，并且在 AArch64 状态下执行。ARMv8 架构还包括原始的 ARM 指令集（现称为 A32）和 Thumb（T32）指令集。A32 和 T32 都以 AArch32 状态执行，并且向后与 ARMv7 架构兼容。

虽然 ARMv8-A 向后兼容了 32 位 ARM 架构的特性，但 A64 指令集与旧的 ISA 指令是独立且不同的，而且他们的编码方式也不同。A64 增加了一些额外的功能，同时也删除了其他可能限制高性能实现速度或能效的功能。ARMv8 架构还包括对 32 位指令集（A32 和 T32）的一些增强性功能。然而，使用这些功能的代码与旧的 ARMv7 实现不兼容。然而，A64 指令集中的指令操作码长度仍然是 32 位，而不是 64 位。

更详细描述 A64 汇编语言的编程指南也可以参考 *ARM® Compiler arm asm Reference Guide v6.01*。

5.1 5.1 ARMv8 指令集

新的 A64 指令集与现有的 A32 指令集相似。指令宽度为 32 位，他们的语法十分相似。

指令集使用 ARMv8 架构中约定的通用命名，因此现在调用原始的 32 位状态指令集是 **A32** 和 **T32**。

- **A32** 在 AArch32 状态下，尽管指令集存在差异，但是在很大程度上与 ARMv7 兼容。请参阅 ARMv8-A 架构参考手册。它还提供了一些新说明，以与 A64 指令集中引入的一些功能保持一致。
- **T32** Thumb 指令集最初包含在 ARM7TDMI 处理器中，最初只包含 16 位指令。16 位指令以牺牲一些性能为代价提供了更小的程序体积。ARMv7 处理器包括 Cortex-A 系列处理器，支持 Thumb-2 技术，该技术扩展了 Thumb 指令集，以提供 16 位和 32 位指令的混合指令集。这提供了与 ARM 相似的性能，同时保留了缩小的代码体积。由于其大小和性能优势，编译或组合所有 32 位代码以利用 Thumb-2 技术越来越常见。

此外还引入了一个新的指令集，ARM 核心在 AArch64 状态下可以使用这个新的指令集。为了与命名约定保持一致，并反映 64 位操作，此指令集称为 **A64**。

- **A64** 提供了与 AArch32 或 ARMv7 中的 A32 和 T32 指令集类似的功能。新的 A64 指令集的设计进行了几项改进：
  - **一致的编码方案**

A32 中一些指令的延迟添加导致编码方案存在一些不一致。例如，对半字的 LDR 和 STR 支持与主流字节和单字传输指令的编码略有不同，这就会导致寻址模式略有不同。

– **种类繁多的常量**

A64 指令为常量提供了大量选项，每个选项都根据特定指令类型的要求进行量身定制。

* 算术指令通常接受 12 位立即数。

* 逻辑指令通常接受 32 位或 64 位常量，其编码存在一些约束。

* MOV 指令接受 16 位立即操作，可以移动到任何 16 位边界。

* 地址生成指令适用于与 4KB 页面大小对齐的地址。

A64 提供了灵活的常量，但对它们进行编码，甚至确定一个特定的常量是否可以在特定的环境下合法地进行编码，都可能是不太容易的。

– **数据类型处理更容易**

A64 可以自然地处理 64 位有符号和无符号数据类型，因为它提供了更简洁、更高效的 64 位整数操作方法。这对所有提供 64 位整数（如 C 或 Java）的语言都有好处。

– **长偏移量**

A64 指令通常为 PC 相关分支和偏移寻址提供了更长的偏移量。

分支范围的增加使其更容易管理节间跳转。动态生成的代码通常被放在堆上，所以在实践中，它可以位于任何地方。运行时系统发现，随着分支范围的增加，管理起来要容易得多，需要的修复工作也少了。

长期以来，对字词池（嵌入代码流中的字词数据块）的需求一直是 ARM 指令集的一个特点。这在 A64 指令集中仍然存在。然而，较大的 PC 相对加载偏移量对字词池的管理有很大帮助，使得每个编译单元可以使用一个字词池。这消除了在长代码序列中需要为多个池开辟位置的需要。

– **指针**

指针在 AArch64 中是 64 位的，这允许对更多的虚拟内存进行寻址，并为地址映射提供更多的自由。然而，使用 64 位指针确实会产生一些代价。同一段代码在使用 64 位指针运行时，通常比使用 32 位指针时使用更多的内存。每个指针存储在内存中，需要 8 个字节而不是 4 个字节。这听起来很微不足道，但加起来就会有很大的损失。此外，由于迁移到 64 位后对内存空间的使用增加，会导致访问缓存的次数减少。这种高速缓存命中率的下降会降低系统性能。有些语言可以用压缩指针来实现，如 Java，以此来规避出现的性能问题。

– **使用条件结构而不是 IT 块**

IT 块是 T32 一个很有用的特性，它可以实现高效的序列，避免在未执行的指令周围出现短的向前分支。然而，它们有时很难被硬件有效地处理。因此 A64 删除了这些块，用 CSEL（即条件选择）和 CINC（即条件增加）等条件指令来代替它们。这些条件结构更直接，更容易处理。

– **移位和旋转行为更直观**

A32 或 T32 的移位和旋转行为并不总是和高级语言预期的行为相对应。

ARMv7 提供了一个桶状移位器，可以作为数据处理指令的一部分使用。但是，指定移位的类型和数量需要一定数量的操作码位，这些操作码位可以用在其他地方。因此，A64 指令删除了很少使用的选项，而是增加了新的显式指令来执行更复杂的移位操作。

- **代码生成**

  在为常见的算术函数静态和动态生成代码时，A32 和 T32 通常需要不同的指令或指令序列。这是为了应对不同的数据类型。A64 中的这些操作更加一致，因此更容易为不同大小的数据类型的简单操作生成公共序列。

  例如，在 T32 中，相同的指令可以有不同的编码，具体取决于使用的寄存器（低寄存器或高寄存器）。

  A64 指令集编码更加规范和合理化。因此，A64 的汇编器通常需要比 T32 的汇编器更少的代码行。

- **固定长度指令**

  与 T32 不同的是，所有 A64 指令的长度都相同，T32 是一个可变长度的指令集。这使得管理和跟踪生成的代码序列更容易，特别是会影响动态代码生成器。

- **三操作数映射更好**

  一般来说，A32 为数据处理操作保留了真正的三操作数结构。另一方面，T32 包含了大量的双操作数指令格式，这使得它在生成代码时的灵活性稍差。A64 坚持采用一致的三操作数语法，这进一步促进了指令集的规则性和同质性，有利于编译器的使用。

5.1.1 5.1.1 区分 32 位和 64 位 A64 指令

A64 指令集中的大多数整数指令有两种形式，它们在 64 位通用寄存器文件中的 32 位或 64 位值上运行。

在查看指令使用的寄存器名称时：

- 如果寄存器名称以 X 开头，则寄存器为 64 位。
- 如果寄存器名称以 W 开头，则寄存器为 32 位。

选择 32 位指令表时，以下情况都成立：

- 在 31 位而不是 63 位向右移位和旋转注入。
- 由指令设置的条件标志是从较低的 32 位开始计算的。
- 将 X 寄存器的 W 寄存器的 Bit[63:32] 设置为零。

即使 32 位指令形式的结果与等效的 64 位指令形式计算的较低 32 位无法区分，这种区别也适用。例如，可以使用 64 位 ORR 执行 32 位按位 ORR，只需忽略结果的前 32 位。A64 指令集包括单独的 32 位和 64 位形式的 ORR 指令。

C 和 C++ 的 LP64 和 LLP64 数据模型预计将是 AArch64 上最常用的。它们都将经常使用的 int、short 和 char 类型定义为 32 位或更少。通过在指令集中维护这一语义信息，实现者可以利用这一信息。例如，避免耗费能量或周期来计算、转发和存储这种数据类型的未使用的前 32 位。实现者可以自由地以他们选择的任何方式利用这种自由来节省能量。

因此，新的 A64 指令集提供了不同的符号和零扩展指令。此外，A64 指令集意味着可以扩展和移动 ADD、SUB、CMN 或 CMP 指令的最终源寄存器以及 Load 或 Store 指令的索引寄存器。这使得可以高效地实现涉及 64 位数组指针和 32 位数组索引的数组索引计算。

5.1.2 5.1.2 地址

当处理器可以在单个寄存器中存储 64 位的值时，在程序中访问大量的内存就变得简单多了。一个在 32 位内核上执行的单线程，只能访问 4GB 的地址空间。该可寻址空间的大部分被保留给操作系统内核、库代码、外围设备等使用。因此，缺乏空间意味着程序在执行时可能需要将一些数据映射进或映射出内存。有一个更大的地址空间，有 64 位指针，就可以避免这个问题。这也使得诸如内存映射文件的技术更有吸引力，使用起来也更方便。文件内容被映射到线程的内存映射中，即使物理 RAM 可能没有大到足以容纳整个文件。

需要解决的其他改进包括：

- **独占访问**

  独占加载存储一个字节、半字、字和双字。对一对双字的独占访问允许对一对指针的原子更新，例如循环列表的插入。所有的独占访问必须自然

  对齐，独占对的访问必须对齐到数据大小的两倍，也就是说，一对 64 位的值要对齐到 128 位。

- **增加 PC 相对偏移地址**

  PC 相关的 load 指令偏移范围为 ±1MB。与 A32 中 PC 相关的 load 指令相比，这减少了字词池的数量，并增加了函数之间字词数据的共享。反过来，

  这也减少了 Icache 和 TLB 污染。

  大多数条件性分支的范围是 ±1MB，预计对发生在单个函数中的大多数条件性分支来说是足够的。

  无条件分支，包括分支和链接，其范围为 ±128MB，预计足以跨越大多数可执行加载模块和共享对象的静态代码段，而不需要链接器插入的单板。

  > 单板是由链接器自动插入的小段代码，例如，当它检测到一个分支目标超出范围时。单板成为原始分支的一个中间目标，单板本身就是通
  >
  > 往目标地址的一个分支。
  >
  > 链接器可以重复使用为前一次调用产生的单板，如果它在两次调用的范围内，则可以重复使用对同一函数的其他调用。偶尔，这种单板会
  >
  > 成为影响性能的因素。
  >
  > 如果你有一个通过单板调用多个函数的循环，你会得到许多管道刷新，从而得到次优的性能。将相关代码放在一起的内存中可以避免这种
  >
  > 情况。

  范围为 ±4GB 的 PC 相对 load 以及 store 和地址生成只能使用两个指令内联执行，即无需从字面池加载偏移量。

- **未对齐地址支持**

  除独占和有序访问外，所有 load 和 store 指令都支持在访问正常内存时使用未对齐的地址。这简化了向 A64 移植代码的工作。

- **批量传输**

  A64 中不存在 LDM、STM、PUSH 和 POP 指令。批量传输可以用 LDP 和 STP 指令来构建。这些指令从连续的内存位置加载和存储一对独立的寄存器。

  LDNP 和 STNP 指令提供了一个流式或非时间性的提示，即数据不需要保留在缓存中。

  PRFM，即预取内存指令可以将预取的目标锁定在一个特定的高速缓存级别。

- **加载/存储**

  所有的加载/存储指令现在支持一致的寻址模式。这使得从内存中加载和存储数量时，以同样的方式处理 char、short、int 和 long long 变得更加容易。浮点和 NEON 寄存器现在支持与核心寄存器相同的寻址模式，使得两个寄存器组的互换使用更加容易。

- **对齐检查**

  当在 AArch64 中执行程序时，在指令获取和使用堆栈指针的加载或存储时执行额外的对齐检查，使 PC 或当前 SP 的错误对齐检查成为可能。

  这种方法比强制对齐 PC 或 SP 更可取，因为 PC 或 SP 的错位通常表示软件错误，例如软件中的地址损坏。

  对齐检查有多种形式：

  – 当试图执行一条在 AArch64 中 PC 错位的指令时，程序计数器对齐检查会产生一个与指令获取相关的异常。

    错位的 PC 被定义为 PC 的位 [1:0] 不为 00 的情况。PC 错位在与目标异常级别相关的异常综合寄存器中被识别。

    当使用 AArch64 处理异常时，相关的异常链接寄存器持有整个 PC 的错位形式，就像故障地址寄存器 FAR_ELn 一样。在该异常级别中，

    PC 对齐检查会在 AArch64 中执行，在 AArch32 中作为数据终止异常处理的一部分。

  – 只要在 AArch64 中尝试使用堆栈指针作为基址的加载或存储，堆栈指针（SP）对齐检查就会产生一个与数据内存访问相关的异常。

    错位的堆栈指针是指堆栈指针的第 [3:0] 位，作为计算的基址，不是 0000。只要堆栈指针被用作基址，它就必须是 16 字节对齐的。

    堆栈指针对齐检查只在 AArch64 中执行，并且可以为每个异常级别独立启用。

    * EL0 和 EL1 由 SCTLR_EL1 中的两个单独的位控制。
    * EL2 在 SCTLR_EL2 中由位控制。
    * EL3 在 SCTLR_EL3 中由位控制。

5.1.3 5.1.3 寄存器

A64 64 位寄存器库有助于在大多数应用程序中降低寄存器压力。

A64 程序调用标准（PCS）在寄存器（X0-X7）中传递多达八个参数。相比之下，A32 和 T32 只在寄存器中传递四个参数，任何多余的参数都在堆栈中传递。

PCS 还定义了一个专用的帧指针（FP），通过可靠地解开堆栈，使调试和调用图分析变得更容易。有关更多信息，请参阅第 9 章 ARM 64 位架构的 ABI。

采用 64 位宽的整数寄存器的一个后果是，编程语言使用的变量的宽度各不相同。目前正在使用一些标准模型，它们主要在为整数、长和指针定义的大小上有所不同，下图是变量宽度表：

64 位的 Linux 实现了使用 LP64，这被 A64 程序调用标准所支持。其他 PCS 变体也被定义，可以被其他操作系统使用。

- **零寄存器**

  零寄存器（WZR/XZR）用于一些编码技巧。例如，没有普通的乘法编码，只有乘法加法。MUL W0、W1、W2 指令与使用零寄存器的 MADD W0、W1、W2、WZR 相同。并非所有指令都可以使用 XZR/WZR。正如我们在第四章中提到的，零寄存器与堆栈指针共享相同的编码。这意味着，对于某些参数数量以及非常有限的指令而言，WZR/XZR 不可用，而是使用 WSP/SP。

  A32 指令集：

```
mov r0, #0
str r0, [...]
```

  A64 指令集使用零寄存器：

```
str wzr, [...]
```

  不需要使用备用寄存器，或者使用以下方式写入 16 字节的零：

```
stp xzr, xzr, [...] etc
```

  零寄存器的一个方便的副作用是，有许多具有大直接字段的 NOP 指令。例如，ADR XZR，仅 # 即可在指令中为您提供 21 位数据，没有其他副作用。这对 JIT 编译器非常有用，可以在运行时修补代码。

- **栈指针**

  大多数指令无法引用堆栈指针（SP）。某些形式的算术指令可以读取或写入当前堆栈指针。这可以用于调整函数序言或结语中的堆栈指针。例如：

```
ADD SP, SP, #256 // SP = SP + 256
```

- **程序计数器**

  当前的程序计数器 (PC) 不能像一般寄存器文件的一部分那样用数字来表示，因此不能作为算术指令的源地址或目的地址，也不能作为加载和存储指令的基地址、索引或转移寄存器。

唯一读取 PC 的指令是那些计算 PC 相关地址的指令（ADR、ADRP、literal load 和 direct branches），以及在链接寄存器中存储返回地址的分支和链接指令（BL 和 BLR）。修改程序计数器的唯一方法是使用分支、异常产生和异常返回指令。

当 PC 被一条计算 PC 相关地址的指令所读取时，那么它的值就是该指令的地址。与 A32 和 T32 不同，PC 没有隐含的 4 或 8 字节的偏移。

- **FP 和 NEON 寄存器**

  NEON 寄存器最重要的更新是，NEON 现在有 32 个 16 字节寄存器，而不是之前的 16 个寄存器。浮点和 NEON 寄存器库中不同寄存器大小之间的更简单的映射方案使得这些寄存器更容易使用。对于编译器和优化器来说，这种映射更容易建模和分析。

- **寄存器索引地址**

  A64 指令集在 A32 的基础上提供了额外的寻址模式，允许 64 位的索引寄存器被添加到 64 位的基础寄存器中，并可以选择按访问大小对索引进行缩

  放。此外，它还提供了索引寄存器中 32 位数值的符号或零扩展，同样可以选择缩放。

5.2 5.2 C++ 内联汇编

在本节中，我们简要介绍了如何在 C 或 C++ 语言模块中包含汇编代码。

asm 关键字可以将内联 GCC 语法汇编代码合并到函数中。例如：

```
#include <stdio.h>

int add(int i, int j) {

        int res = 0;
  asm (
        "ADD %w[result], %w[input_i], %w[input_j]"//Use `%w[name]` to operate on W
   ␣
↪
↪/ registers (as in this case).
   ␣
↪
↪/ You can use `%x[name]` for X
   ␣
↪
↪/ registers too, but this is the
   ␣
↪
↪/ default.
        : [result] "=r" (res)
```

(continues on next page)

(continued from previous page)

```
        : [input_i] "r" (i), [input_j] "r" (j) );
        return res;
}

int main(void) {
        int a = 1; int b = 2; int c = 0;
        c = add(a,b)
        printf( “Result of %d + %d = %d\n, a, b, c);
}
```

Asm 内联汇编语句的一般形式是：

```
asm(code [: output_operand_list [: input_operand_list [: clobber_list]]]);
```

这里的代码是程序集代码。在我们的示例中，这是 “add%[result]，%[input_i]，%[input_j]”。

Output_operand_list 是一个可选的输出操作数列表，用逗号分隔。每个操作数由方括号中的符号名称、约束字符串和括号中的 C 表达式组成。在本例中，有一个输出操作数：[result] “=r”（res）。

Input_operand_list 是一个可选的输入操作数列表，用逗号分隔。输入操作数使用与输出操作数相同的语法。在本例中，有两个输入操作数：[input_i] “r”（i）和 [input_j] “r”（j）。

Clobber_list 是 crobbered 寄存器或其他值的可选列表。在我们的示例中，省略了这一点。

在 C/C++ 和程序集代码之间调用函数时，您必须遵循 AAPCS64 规则。

有关更多信息，请参阅：

https://gcc.gnu.org/onlinedocs/gcc/Using-Assembly-Language-with-C.html#Using-Assembly-L anguage-with-C

5.3 5.3 在指令集之间切换

不可能在单个应用程序中使用来自两个执行状态的代码。ARMv8 中 A64 和 A32 或 T32 指令集之间没有互操作，因为 A32 和 T32 指令集之间有互操作。用 A64 编写的 ARMv8 处理器代码无法在 ARMv7 Cortex-A 系列处理器上运行。然而，为 ARMv7-A 处理器编写的代码可以在 AArch32 执行状态下的 ARMv8 处理器上运行。下面的图总结了这一点。

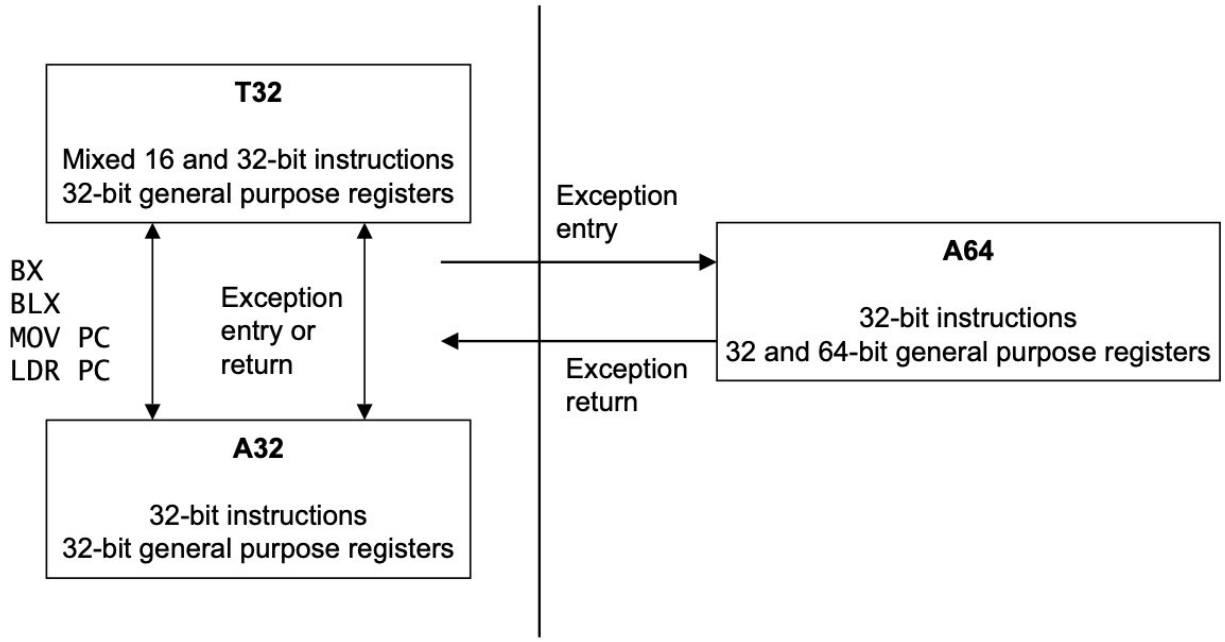
T32
Mixed 16 and 32-bit instructions
32-bit general purpose registers
BX
BLX
MOV PC
LDR PC
Exception
entry or
return
Exception
entry
A64
32-bit instructions
32 and 64-bit general purpose registers
Exception
return
A32
32-bit instructions
32-bit general purpose registers

image-05_01

CHAPTER

SIX

6. A64 指令集

许多程序员在应用层面上的写作不需要用汇编语言写代码。然而，在需要高度优化代码的情况下，汇编代码可能是有用的。当编写编译器时，或者需要使用 C 语言中不能直接使用的低级功能时，就是这种情况。在开发操作系统时，启动代码的一部分、设备驱动程序可能需要汇编代码。最后，在调试 C 语言时，能够阅读汇编代码是非常有用的，特别是了解汇编指令和 C 语句之间的映射关系。

6.1 6.1 指令助记符

A64 汇编语言重载了指令的记忆法，并根据操作数寄存器的名称来区分指令的不同形式。例如，下面的 ADD 指令都有不同的编码，但你只需记住一个助记符，汇编程序会根据操作数自动选择正确的编码。

```
ADD W0, W1, W2         // add 32-bit registers
ADD X0, X1, X2         // add 64-bit registers
ADD X0, X1, W2, SXTW         // add sign extended 32-bit register to 64-bit extended //
↪ register
ADD X0, X1, #42         // add immediate to 64-bit register
ADD V0.8H, V1.8H, V2.8H         // NEON 16-bit add, in each of 8 lanes
```

6.2 6.2 数据处理指令

这些指令是处理器的基本算术和逻辑操作，对通用寄存器中的数值或一个寄存器和一个即时数值进行操作。6-4 页的乘法和除法指令可以看作是这些指令的特例。数据处理指令大多使用一个目标寄存器和两个源操作数。一般的格式可以认为是指令，其次是操作数，如下所示。

```
Instruction Rd, Rn, Operand2
```

第二个操作数可能是一个寄存器，一个修改后的寄存器，或者一个立即数。R 的使用表明它可以是一个 X 或一个 W 寄存器。

数据处理操作包括：

- 算术和逻辑运算。

- 移动和位移操作。
- 符号和零扩展的指令。
- Bit 和 bitfield 操作。
- 有条件比较和数据处理。

6.2.1 6.2.1 算术和逻辑运算

下面的表格显示一些可用的算术和逻辑运算。

有些指令还有一个 S 的后缀，表示该指令设置了标志。在表 6-1 的指令中，包括 ADDS、SUBS、ADCS、SBCS、ANDS 和 BICS。还有其他设置标志的指令，特别是 CMP、CMN 和 TST，但这些指令没有 S 后缀。ADC 和 SBC 操作进行加法和减法，也使用携带条件标志作为输入。

```
ADC{S}: Rd = Rn + Rm + C SBC{S}
Rd = Rn - Rm - 1 + C
```

算术指令示例：

```
ADD W0, W1, W2, LSL #3 SUBS X0, X4, X3, ASR #2 MOV X0, X1        // W0 = W1 + (W2 <<␣
↪3)
CMP W3, W4         // X0 = X4 - (X3 >> 2), set flags // Copy X1 to X0
ADD W0, W5, #27         // Set flags based on W3 - W4
Example 6-1 Arithmetic instructions         // W0 = W5 + 27
```

逻辑操作本质上与在寄存器的单个位上运行的相应布尔运算符相同。

BIC（位元位清除）指令执行寄存器的 AND，这是目标寄存器之后的第一个，具有第二个操作数的倒置值。例如，要清除寄存器 X0 的位 [11]，请使用：

```
MOV X1, #0x800
BIC X0, X0, X1
```

ORN 和 EON 分别与第二操作数的位数-NOT 进行 OR 或 EOR。

比较指令只修改标志，没有其他作用。这些指令的即时值的范围是 12 位，这个值可以选择向左移动 12 位。

6.2.2 6.2.2 乘以和除法指令

乘法指令与 ARMv7-A 中的乘法指令大体相似，但能够在单个指令中执行 64 位乘法。汇编语言中的乘法操作如下图所示:

Opcode	Description
Multiply instructions	
MADD	Multiply add
MNEG	Multiply negate
MSUB	Multiply subtract
MUL	Multiply
SMADDL	Signed multiply-add long
SMNEGL	Signed multiply-negate long
SMSUBL	Signed multiply-subtract long
SMULH	Signed multiply returning high half
SMULL	Signed multiply long
UMADDL	Unsigned multiply-add long
UMNEGL	Unsigned multiply-negate long
UMSUBL	Unsigned multiply-subtract long
UMULH	Unsigned multiply returning high half
UMULL	Unsigned multiply long
Divide instructions	
SDIV	Signed divide
UDIV	Unsigned divide

image-06_01

有一些乘法指令对 32 位或 64 位的数值进行操作，并返回与操作数相同大小的结果。例如，用 MUL 指令可以将两个 64 位的寄存器相乘，产生一个 64 位的结果。

```
MUL X0, X1, X2 // X0 = X1 * X2
```

还可以用 MADD 或 MSUB 指令在第三个源寄存器中加减累加器的值。

例如，MNEG 指令可以用来对结果进行取反:

```
MNEG X0, X1, X2 // X0 = -(X1 * X2)
```

此外，还有一系列的乘法指令可以产生一个长的结果，即把两个 32 位的数字相乘，产生一个 64 位的结果。这些长线乘法有有符号和无符号的变体（UMULL, SMULL）。还有一些选项是将另一个寄存器中的数值累积起来（UMADDL, SMADDL）或进行取反（UMNEGL, SMNEGL）。

包括 32 位和 64 位的乘法，并可选择累加，其结果大小与操作数相同:

- 32±（32×32）得到 32 位结果。
- 64±（64×64）得到 64 位结果。
- ±（32×32）得到 32 位结果。
- ±（64×64）得到 64 位结果。

加宽乘法，即有符号和无符号的乘法，通过累加得到一个 64 位的结果:

- 64±（32×32）得到 64 位结果。
- ±（32×32）得到 64 位的结果。

一个 64×64 到 128 位的乘法运算需要两个指令序列来产生一对 64 位的结果寄存器。

- ±（64×64）得到结果 [63:0] 的较低 64 位。
- （64×64）得到结果的更高 64 位 [127:64]。

  该列表不包含 32×64 选项。您不能直接将 32 位 W 寄存器乘以 64 位 X 寄存器。

ARMv8-A 架构支持 32 位和 64 位大小值的有符号和无符号除法。比如说:

```
UDIV W0, W1, W2 // W0 = W1 / W2 (unsigned, 32-bit divide)
SDIV X0, X1, X2 // X0 = X1 / X2 (signed, 64-bit divide)
```

溢出和除零不会导致系统问题:

- 任何整数除以零都会返回零。
- 溢出只能在 SDIV 中发生:
  - INT_MIN / -1 返回 INT_MIN，其中 INT_MIN 是最小的负数，可以在用于操作的寄存器中进行编码。与大多数 C/C++ 方言一样，结果总是向零四舍五入。

6.2.3 6.2.3 位移操作

以下说明是专门针对位移的：

- 逻辑向左移位（LSL）。LSL 指令执行 2 的乘法。
- 逻辑向右移位（LSR）。LSR 指令以 2 的幂执行除法。
- 算术向右移（ASR）。ASR 指令执行 2 的幂的除法，保留符号位。
- 右旋转（ROR）。ROR 指令执行位旋转，将旋转的位从 LSB 包裹到 MSB。

位移操作如下表所示：

Table 6-3 Shift and move operations

Instruction	Description
Shift	
ASR	Arithmetic shift right
LSL	Logical shift left
LSR	Logical shift right
ROR	Rotate right
Move	
MOV	Move
MVN	Bitwise NOT

image-20220420181731308

位移操作示意图如下图所示：

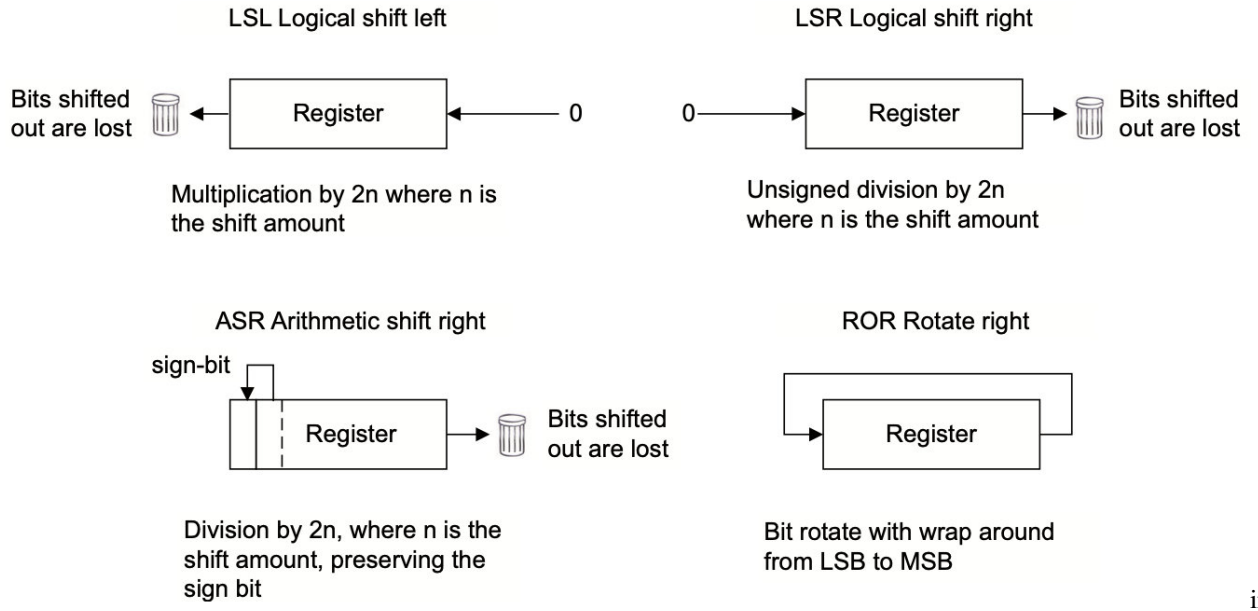

image-20220420181941575

为移位指定的寄存器可以是 32 位或 64 位。要移位的数量可以指定为一个即时值，即达到寄存器大小减 1，或者通过一个寄存器，其值只取自底部的 5 位（modulo-32）或 6 位（modulo-64）。

6.2.4 6.2.4 Bitfield 和字节操作指令

这些指令存在有符号（SXTB, SXTH, SXTW）和无符号（UXTB, UXTH）两种变体，是相应的 Bitfield 操作指令的别名。

这些指令的有符号和无符号变体都是将一个字节、半字或字（尽管只有 SXTW 在字上操作）扩展到寄存器大小。源寄存器总是一个 W 寄存器。目的寄存器可以是 X 或 W 寄存器，除了 SXTW 必须是 X 寄存器。

例如：

```
SXTB X0, W1 // 符号扩展了寄存器 W1 的最小有效字节，通过重复最左边的位，从 8 位到 64 位字节。
```

Bitfield(位域) 指令与 ARMv7 中的指令类似，包括位域插入（BFI）以及有符号和无符号位域提取（(S/U)BFX）。还有一些额外的位域指令，如 BFXIL（位域提取和插入低）、UBFIZ（无符号位域插入零）和 SBFIZ（有符号位域插入零）。

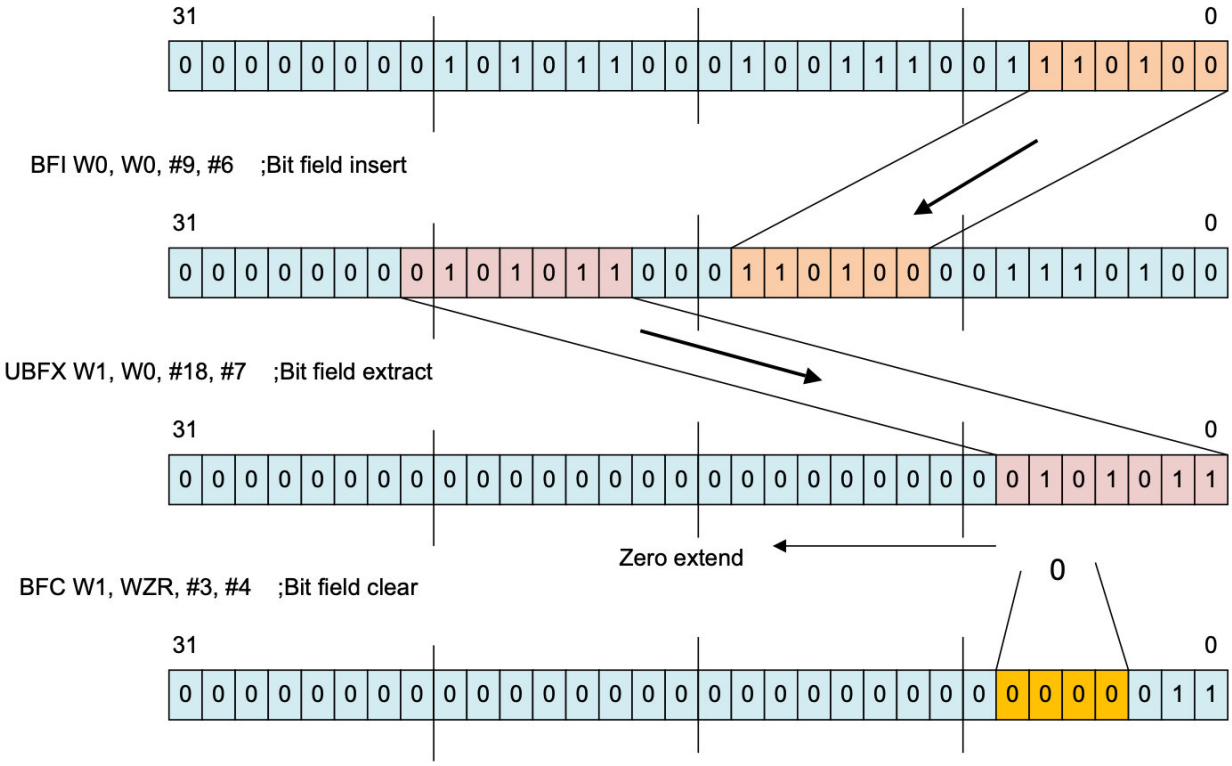

image-20220420182307696

还有 BFM、UBFM 和 SBFM 指令。这些是位域移动指令，是 ARMv8 的新指令。但是，不需要明确使用这些指令，因为为所有情况提供了别名。这些别名是已经描述过的位域操作。[SU]XT[BHWX]、ASR/LSL/LSR immediate、BFI、BFXIL、SBFIZ、SBFX、UBFIZ 和 UBFX。

如果你熟悉 ARMv7 架构，你可能会认出另一个位操作指令。

- CLZ 在寄存器中计数前导零位。

同样，相同的字节操作指令：

- RBIT 反转所有位。
- REV 反转寄存器的字节顺序。
- REV16 反转寄存器中每个半字的字节顺序。

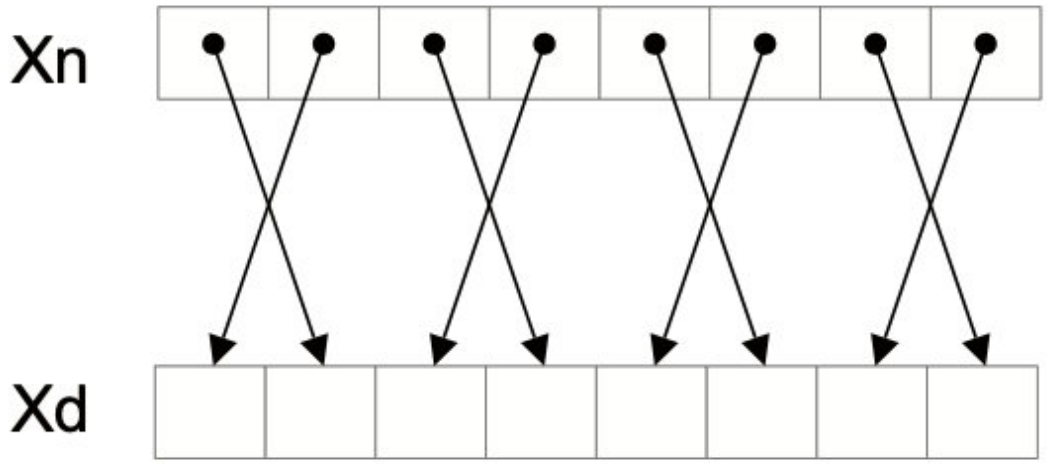

Figure 6-3 REV16 instruction image-20220420182506614

- REV32 反转寄存器中每个单词的字节顺序。

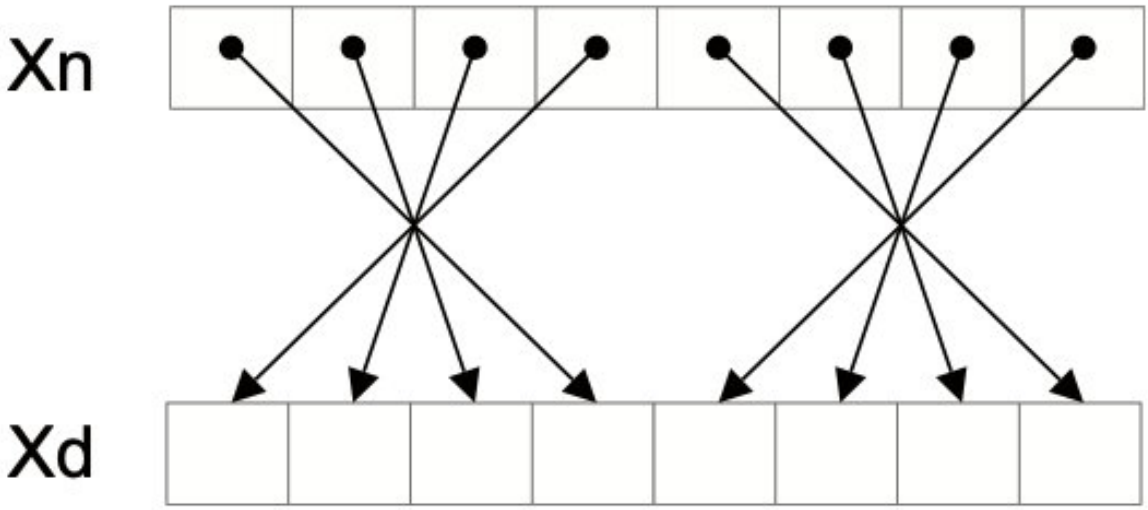

Figure 6-4 REV32 instruction image-20220420182548295

这些操作可以在 word（32 位）或双字（64 位）大小的寄存器上执行，但 REV32 除外，它仅适用于 64 位寄存器。

6.2.5 6.2.5 条件指令

A64 指令集并不支持每条指令的条件执行。预先执行的指令并没有提供足够的好处来证明其对操作码空间的大量使用。

第 4-6 页的处理器状态描述了四个状态标志，零（Z）、负（N）、Carry（C）和溢出（V）。下面的表格表示这些位的值，用于标志设置操作。

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

image-20220420182704887

如果一个无符号操作的结果溢出了结果寄存器，C 标志被设置。V 标志的操作方式与 C 标志相同，但用于签名操作。

> 条件标志（NZCV）和条件代码与 A32 和 T32 的相同。然而，A64 增加了 NV（0b1111），尽管它的行为与它的补码 AL（0b1110）相同。这与 A32 不同，A32 没有给 0b1111 赋予任何意义。

Code	Encoding	Meaning (when set by CMP)	Meaning (when set by FCMP)	Condition flags
EQ	0b0000	Equal to.	Equal to.	Z =1
NE	0b0001	Not equal to.	Unordered, or not equal to.	Z = 0
CS	0b0010	Carry set (identical to HS).	Greater than, equal to, or unordered (identical to HS).	C = 1
HS	0b0010	Greater than, equal to (unsigned) (identical to CS).	Greater than, equal to, or unordered (identical to CS).	C = 1
CC	0b0011	Carry clear (identical to LO).	Less than (identical to LO).	C = 0
LO	0b0011	Unsigned less than (identical to CC).	Less than (identical to CC).	C = 0
MI	0b0100	Minus, Negative.	Less than.	N = 1
PL	0b0101	Positive or zero.	Greater than, equal to, or unordered.	N = 0
VS	0b0110	Signed overflow.	Unordered. (At least one argument was NaN).	V = 1
VC	0b0111	No signed overflow.	Not unordered. (No argument was NaN).	V = 0
HI	0b1000	Greater than (unsigned).	Greater than or unordered.	(C = 1) && (Z = 0)
LS	0b1001	Less than or equal to (unsigned).	Less than or equal to.	(C = 0) || (Z = 1)
GE	0b1010	Greater than or equal to (signed).	Greater than or equal to.	N==V
LT	0b1011	Less than (signed).	Less than or unordered.	N!=V
GT	0b1100	Greater than (signed).	Greater than.	(Z==0) && (N==V)
LE	0b1101	Less than or equal to (signed).	Less than, equal to or unordered.	(Z==1) || (N!=V)
AL	0b1110	Always executed.	Default. Always executed.	Any
NV	0b1111	Always executed.	Always executed.	Any

image-20220420182839984

有一小部份的条件数据处理指令。这些指令是无条件执行的，但使用条件标志作为指令的额外输入。提供这组指令是为了取代 ARM 代码中条件执行的常见用法。

读取条件标志的指令类型有：

- **加/减**

  例如，用于多精度算术和校验和的传统 ARM 指令。

- **带有可选增量、否定或反转的条件选择**

  有条件地在一个源寄存器和第二个增量、否定、倒置或未修改的源寄存器之间进行选择。

  这些是 A32 和 T32 中单个条件指令最常见的用途。典型的用途包括有条件计数或计算有符号数量的绝对值。

- 条件操作

  A64 指令集只允许对程序流控制分支指令进行条件执行。这与 A32 和 T32 相反，后者的大多数指令都可以用条件代码来预测。这些可以总结为以

下几点：

- 有条件选择（移动）
  * CSEL 根据一个条件在两个寄存器之间进行选择。无条件指令，然后是条件选择，可以取代简短的条件序列。
  * CSINC 根据一个条件在两个寄存器之间进行选择。返回第一个源寄存器或第二个源寄存器增加一个。
  * CSINV 根据条件在两个寄存器之间进行选择。返回第一个源寄存器或倒置的第二个源寄存器。
  * CSNEG 根据条件在两个寄存器之间进行选择。返回第一个源寄存器或被否定的第二个源寄存器。
- 有条件设置

  有条件地在 0 和 1（CSET）或 0 和-1（CSETM）之间进行选择。例如，用于在一般寄存器中将条件标志设置为布尔值或掩码。
- 有条件比较

  （CMP 和 CMN）如果原始条件为真，则将条件标志设置为比较结果。如果不是真，条件标志将设置为指定的条件标志状态。条件比较指令对于表示嵌套或复合比较非常有用。

  使用 FCSEL 和 FCCMP 指令的浮点寄存器也可以使用条件选择和条件比较。

例如：

```
CSINC X0, X1, X0, NE // Set the return register X0 to X1 if Zero flag clear,else␣
↪increment X0
```

上面的代码提供了示例指令的一些别名，其中要么使用零寄存器，要么将同一寄存器用作指令的目标和两个源寄存器。

例如：

```
CINC X0, X0, LS          // If less than or same (LS) then X0 = X0 + 1
CSET W0, EQ                         // If the previous comparison was equal (Z=1) then␣
↪W0 = 1, // else W0 = 0
CSETM X0, NE         // If not equal then X0 = -1, else X0 = 0
```

这类指令为避免使用分支或有条件执行的指令提供了一个强有力的方法。编译器或汇编程序员可能采用一种技术，对 if-then-else 语句的两个分支进行操作。然后在最后选择正确的结果。

例如，考虑简单的 C 代码：

```
if (i == 0) r = r + 2; else r = r - 1;
```

这可能会产生类似于以下内容的代码：

```
CMP w0, #0        // if (i == 0)
SUB w2, w1, #1    // r = r - 1
ADD w1, w1, #2    // r = r + 2
CSEL w1, w1, w2, EQ  // select between the two results
```

6.3 6.3 内存访问指令

与之前的所有 ARM 处理器一样，ARMv8 架构是一个加载/存储架构。这意味着没有数据处理指令直接对内存中的数据进行操作。数据必须首先被加载到寄存器中，进行修改，然后存储到内存中。程序必须指定一个地址，要传输的数据大小，以及一个源寄存器或目标寄存器。还有一些额外的加载和存储指令，提供了更多的选择，如非时间性的加载/存储，加载/存储排他性，以及获取/释放。

内存指令可以以非对齐方式访问普通内存（见第 13 章内存排序）。这不支持独占访问、加载获取或存储释放变体。如果不需要非对齐访问，可以将其配置为故障。

6.3.1 6.3.1 加载指令格式

Load 指令的一般形式如下：

```
LDR Rt, <addr>
```

对于加载到整数寄存器中，你可以选择一个大小来加载。例如，要加载一个比指定的寄存器值小的尺寸，在 LDR 指令中加入以下后缀之一：

- LDRB (8-bit, zero extended).
- LDRSB (8-bit, sign extended).
- LDRH (16-bit, zero extended).
- LDRSH (16-bit, sign extended).
- LDRSW (32-bit, sign extended).

还有一些非比例偏移的形式，如 LDUR（参见第 6-14 页指定加载或存储指令的地址）。程序员通常不需要明确使用 LDUR 形式，因为大多数汇编程序可以根据使用的偏移量选择合适的版本。

你不需要指定一个零扩展的负载到 X 寄存器，因为写一个 W 寄存器有效的零扩展到整个寄存器的宽度。

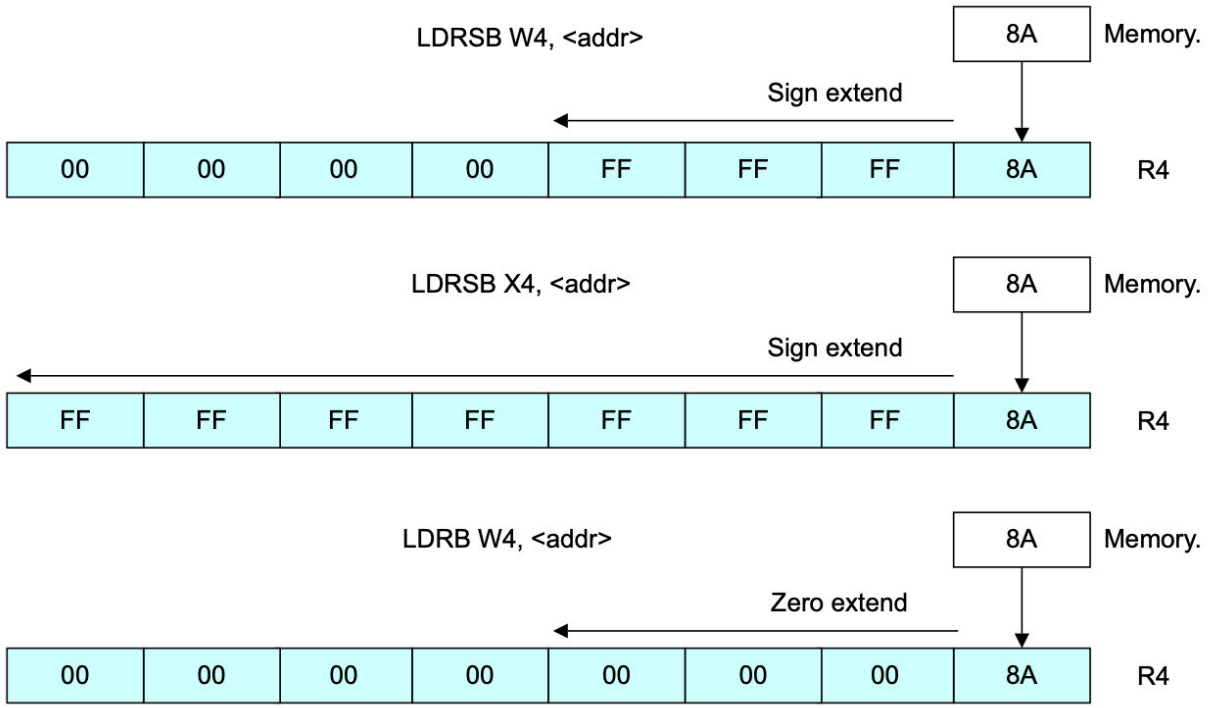

Figure 6-5 Load instructionsimage-20220420184129951

6.3.2 6.3.2 存储指令格式

同样，存储指令的一般形式如下：

```
STR Rn, <addr>
```

还有一些非比例偏移的形式，如 STUR（参见第 6-14 页的指定加载或存储指令的地址）。程序员通常不需要明确使用 STUR 形式，因为大多数汇编程序可以根据使用的偏移量选择合适的版本。

要存储的大小可能比寄存器小。你可以通过在 STR 中加入 B 或 H 的后缀来指定。在这种情况下，存储的总是寄存器中最小的有效部分。

6.3.3 6.3.3 浮点和 NEON 标量加载和存储

加载和存储指令也可以访问浮点/NEON 寄存器。在这里，大小只由被加载或存储的寄存器决定，它可以是 B、H、S、D 或 Q 寄存器中的任何一个。

加载指令总结如下表所示：

Table 6-6 Memory bits written by Load instructions

Load	Xt	Wt	Qt	Dt	St	Ht	Bt
LDR	64	32	128	64	32	16	9
LDP	128	64	256	128	64	-	-
LDRB	-	8	-	-	-	-	-
LDRH	-	16	-	-	-	-	-
LDRSB	8	8	-	-	-	-	-
LDRSH	16	16	-	-	-	-	-
LDRSW	32	-	-	-	-	-	-
LDPSW	-	-	-	-	-	-	-

20220420184437101

存储指令总结如下表所示：

Table 6-7 Memory bits read by Store instructions

Store	Xt	Wt	Qt	Dt	St	Ht	Bt
STR	64	32	126	64	32	16	8
STP	128	64	256	128	64	-	-
STRB	-	8	-	-	-	-	-
STRH	-	16	-	-	-	-	-

20220420184608448

对于加载到 FP/SIMD 寄存器中，没有符号扩展选项可用。这种加载的地址仍然使用通用寄存器来指定。

例如：

```
LDR D0, [X0, X1]
```

用 X0 加 X1 所指向的内存地址的双字加载寄存器 D0。

浮点和标量 NEON 负载和存储使用与整数负载和存储相同的寻址模式。

6.3.4 6.3.4 指定加载或存储指令的地址

A64 可用的寻址模式与 A32 和 T32 中的相似。有一些额外的限制以及一些新的功能，但是对于熟悉 A32 或 T32 的人来说，A64 可用的寻址模式并不令人惊讶。

在 A64 中，一个地址操作数的基寄存器必须总是一个 X 寄存器。然而，有几条指令支持零扩展或符号扩展，这样就可以提供一个 32 位的偏移作为一个 W 寄存器。

偏移模式

偏移寻址模式将一个立即数或一个可选择修改的寄存器值添加到一个 64 位的基础寄存器中以产生一个地址。

Table 6-8 Offset addressing modes

Example instruction	Description
LDR X0, [X1]	Load from the address in X1
LDR X0, [X1, #8]	Load from address X1 + 8
LDR X0, [X1, X2]	Load from address X1 + X2
LDR X0, [X1, X2, LSL, #3]	Load from address X1 + (X2 << 3)
LDR X0, [X1, W2, SXTW]	Load from address X1 + sign_extend(W2)
LDR X0, [X1, W2, SXTW, #3]	Load from address X1 + (sign_extend(W2) << 3)

image-20220420184852102

通常，在指定移位或扩展选项时，移位量可以是以字节为单位的访问大小的 0（默认值）或 log2（以便 Rn << 乘以访问大小）。这支持常见的数组索引操作。

```
// A C example showing accesses that a compiler is likely to generate.
void example_dup(int32_t a[], int32_t length) {
                int32_t first = a[0];        // LDR W3, [X0]
```

(continues on next page)

(continued from previous page)

```
                for (int32_t i = 1; i < length; i++) {
                                a[i] = first;                           // STR W3, [X0,␣
↪W2, SXTW, #2]
                }
}
```

索引模式

索引模式与偏移模式类似，但它们也更新基地址寄存器。这里的语法与 A32 和 T32 相同，但操作集的限制性更强。通常情况下，只能为索引模式提供即时偏移量。

有两种变体：在访问内存之前应用偏移量的预索引模式，以及在访问内存之后应用偏移量的后索引模式。

Table 6-9 Index addressing modes

Example instruction	Description
`LDR X0, [X1, #8]!`	Pre-index: Update X1 first (to X1 + #8), then load from the new address
`LDR X0, [X1], #8`	Post-index: Load from the unmodified address in X1 first, then update X1 (to X1 + #8)
`STP X0, X1, [SP, #-16]!`	Push X0 and X1 to the stack.
`LDP X0, X1, [SP], #16`	Pop X0 and X1 off the stack.

image-20220420185057121

这些选项准确地映射到一些常见的 C 操作上。:

```
// A C example showing accesses that a compiler is likely to generate.
void example_strcpy(char * dst, const char * src)
{
                char c;
                do {
                                c = *(src++);           // LDRB W2, [X1], #1
                                *(dst++) = c;           // STRB W2, [X0], #1
                } while (c != '\0');
}
```

PC-relative 模式（字面加载）

A64 增加了另一种专门用于访问文字池的寻址模式。文字池是在指令流中编码的数据块。这些池子不被执行，但是它们的数据可以使用 PC 相对的内存地址从周围的代码中访问。文字池经常被用来编码常量值，这些常量值不适合用简单的立即移动指令。

在 A32 和 T32 中，PC 可以像一个通用寄存器一样被读取，所以只需指定 PC 为基寄存器就可以访问一个字库。

在 A64 中，PC 普遍不能被访问，而是有一种特殊的寻址模式（仅用于加载指令）可以访问与 PC 相关的地址。这种特殊的寻址模式也比 A32 和 T32 中的 PC-relative 加载所能达到的范围要大得多。因此，文字池可以更稀疏地定位。

Table 6-10

Example instruction	Description
`LDR W0, <label>`	Load 4 bytes from <label> into W0
`LDR X0, <label>`	Load 8 bytes from <label> into X0
`LDRSW X0, <label>`	Load 4 bytes from <label> and sign-extend into X0
`LDR S0, <label>`	Load 4 bytes from <label> into S0
`LDR D0, <label>`	Load 8 bytes from <label> into D0
`LDR Q0, <label>`	Load 16 bytes from <label> into Q0

image-20220420185243855

6.3.5 6.3.5 访问多个内存位置

A64 不包括 A32 和 T32 代码可以使用的 Load Multiple(LDM) 或 Store Multiple(STM) 指令。

在 A64 代码中，有加载对（LDP）和存储对（STP）指令。与 A32 的 LDRD 和 STRD 指令不同，任何两个整数寄存器都可以被读取或写入。数据被读入或写入相邻的内存位置。为这些指令所提供的寻址模式选项比其他内存访问指令的限制性更强。LDP 和 STP 指令只能使用一个带有 7 位有符号即时值的基寄存器，并可选择预增加或后增加。与 32 位的 LDRD 和 STRD 不同，LDP 和 STP 可以进行无符号访问。

Table 6-11 Register Load/Store pair

Load and Store pair	Description
`LDP W3, W7, [X0]`	Loads word at address X0 into W3 and word at address X0 + 4 into W7. See Figure 6-6.
`LDP X8, X2, [X0, #0x10]!`	Loads doubleword at address X0 + 0x10 into X8 and the doubleword at address X0 + 0x10 + 8 into X2 and add 0x10 to X0. See Figure 6-7.
`LDPSW X3, X4, [X0]`	Loads word at address X0 into X3 and word at address X0 + 4 into X4, and sign extends both to doubleword size.
`LDP D8, D2, [X11], #0x10`	Loads doubleword at address X11 into D8 and the doubleword at address X11 + 8 into D2 and adds `0x10` to X11.
`STP X9, X8, [X4]`	Stores the doubleword in X9 to address X4 and stores the doubleword in X8 to address X4 + 8.

image-20220420185343789

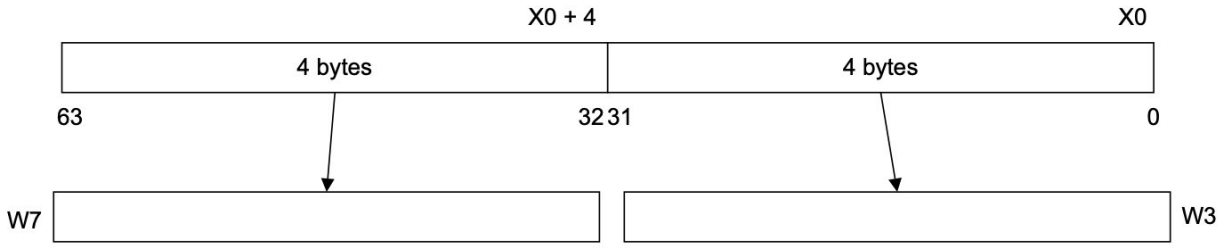

Figure 6-6 LDP W3, W7 [X0]

image-20220420185424932

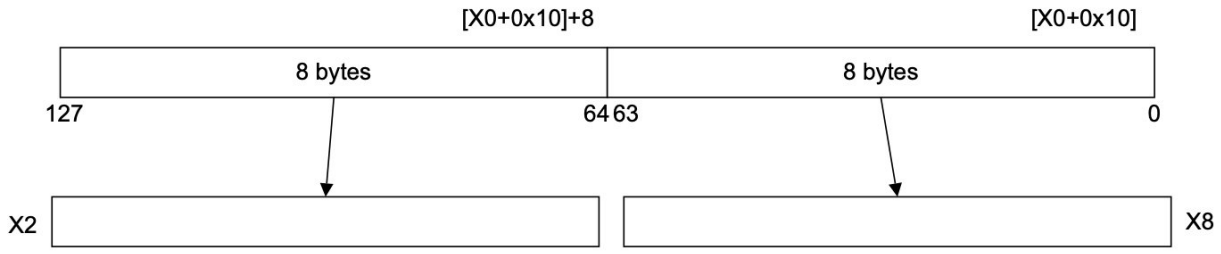

Figure 6-7 LDP X8, X2, [X0 + #0x10]!

image-20220420185446323

6.3.6 6.3.6 无特权访问

A64 的 LDTR 和 STTR 指令执行无特权的加载或存储（参见 ARMv8-A 架构参考手册中的 LDTR 和 STTR）：

- 在 EL0、EL2 或 EL3，它们表现为正常加载或存储。
- 当在 EL1 执行时，它们的行为就像是在 EL0 特权级别执行的一样。

这些指令相当于 A32 LDRT 和 STRT 指令。

6.3.7 6.3.7 预取内存

从内存中预取（PRFM）使代码能够向内存系统提供一个提示，即来自特定地址的数据将很快被程序使用。这个提示的效果由实施者决定，但通常情况下，它导致数据或指令被加载到一个缓存中。

指令语法是：

```
PRFM <prfop>, <addr> | label
```

其中 prfop 是以下选项的串联：

```
Type    PLD or PST (prefetch for load or store).
Target  L1, L2, or L3 (which cache to target).
Policy  KEEP or STRM (keep in cache, or streaming data).
```

例如，PLDL1KEEP。

这些指令与 A32 PLD 和 PLI 指令相似。

6.3.8 6.3.8 非时间性加载和存储对

ARMv8 中的一个新概念是非时间性的加载和存储。这些是 LDNP 和 STNP 指令，执行一对寄存器值的读或写。它们还向内存系统发出提示，说明缓存对该数据没有用处。这个提示并不禁止内存系统的活动，如地址的缓存、预加载或收集。然而，它表明缓存不太可能提高性能。一个典型的用例可能是流式数据，但要注意，有效使用这些指令需要针对微架构的方法。

非时间性的加载和存储放宽了内存排序的要求。在上面的例子中，LDNP 指令可能在前面的 LDR 指令之前被观察到，这可能导致从 X0 的不确定地址读取。

例如：

```
LDR X0, [X3]
LDNP X2, X1, [X0] // Xo may not be loaded when the instruction executes!
```

为了纠正上述问题，你需要一个明确的负载屏障：

```
LDR X0, [X3]
DMB nshld
LDNP X2, X1, [X0]
```

6.3.9 6.3.9 内存访问原子性

使用单一通用寄存器的对齐内存访问需要被保证为原子访问。一对通用寄存器的加载和存储指令，使用一个对齐的内存地址，保证显示为两个单独的原子访问。不对齐访问不是原子的，因为它们通常需要两个单独的访问。此外，浮点和 SIMD 内存访问不保证是原子的。

6.3.10 6.3.10 内存屏障和栅栏说明

ARMv7 和 ARMv8 都提供了对不同屏障操作的支持。这些将在第 13 章内存排序中详细描述：

- 数据内存屏障（DMB）。这迫使所有在程序中较早的内存访问在任何后续访问之前成为全局可见。
- 数据同步屏障（DSB）。所有悬而未决的加载和存储、缓存维护指令以及所有的 TLB 维护指令，在程序继续执行之前都会完成。DSB 的行为类似于 DMB，但有额外的属性。
- 指令同步屏障（ISB）。这条指令刷新了 CPU 流水线和预取缓冲区，导致 ISB 之后的指令被从缓存或内存中取走（或重新取走）。

ARMv8 引入了单边栅栏，它与释放一致性模型有关。这些被称为加载-获取（LDAR）和存储-释放（STLR），是基于地址的同步原语。(参见第 13-8 页的单向屏障。) 这两个操作可以配对形成一个完整的栅栏。这些指令只支持基寄存器寻址，不提供偏移量或其他类型的索引寻址。

6.3.11 6.3.11 原始同步

ARMv7-A 和 ARMv8-A 体系结构都提供对独占内存访问的支持。在 A64 中，这就是加载/存储独占（LDXR/STXR）对。

LDXR 指令从一个内存地址加载一个值，并试图默默地要求对该地址进行独占锁定。然后，只有在成功获得并保持锁的情况下，Store-Exclusive 指令才会向该位置写入一个新的值。LDXR/STXR 配对被用来构建标准的同步原语，如自旋锁。提供了一组成对的 LDXRP 和 STXRP 指令，以允许代码原子地更新一个跨越两个寄存器的位置。有字节、半字、字和双字选项。与加载获取/存储释放配对一样，只支持基本寄存器寻址，没有任何偏移。

CLREX 指令清除了监视器，但与 ARMv7 不同的是，异常进入或返回也会清除监视器。监控器也可能被虚假地清除，例如，通过缓存驱逐或其他与应用程序没有直接关系的原因。软件必须避免在成对的 LDXR 和 STXR 指令之间有任何显式内存访问、系统控制寄存器更新或高速缓存维护指令。

还有一对被称为 LDAXR 和 STLXR 的负载获取/存储释放指令的排他性。请看第 14-6 页的同步化。

6.4 6.4 流控

A64 指令集提供了许多不同种类的分支指令（见表 6-12）。对于简单的相对分支，即那些从当前地址开始的偏移量，使用 B 指令。无条件的简单相对分支可以从当前的程序计数器位置向后或向前分支，最多可达 128MB。有条件的简单相对分支，即在 B 指令上附加了一个条件代码，其范围较小，为 ±1MB。

对子程序的调用，如果需要将返回地址保存在链接寄存器（X30）中，则使用 BL 指令。这条指令没有条件版本。BL 的行为和 B 指令一样，有一个额外的效果，就是将返回地址，也就是 BL 后的指令地址，存储在寄存器 X30 中。

Table 6-12 Branch instructions

Branch instructions	
`B (offset)`	Program relative branch forward or back 128MB. A conditional version, for example B.EQ, has a 1MB range.
`BL (offset)`	As B but store the return address in X30, and hint to branch prediction logic that this is a function call.
`BR Xn`	Absolute branch to address in Xn.
`BLR Xn`	As BR but store the return address in X30, and hint to branch prediction logic that this is a function call.
`RET{Xn}`	As BR, but hint to branch prediction logic that this is a function return. Returns to the address in X30 by default, but a different register can be specified.
Conditional branch instructions	
`CBZ Rt, label`	Compare and branch if zero. If Rt is zero, branch forward or back up to 1MB.
`CBNZ Rt, label`	Compare and branch if non-zero. If Rt is not zero, branch forward or back up to 1MB.
`TBNZ Rt, bit, label`	Test and branch if zero. Branch forward or back up to 32kB.
`TBNZ Rt, bit, label`	Test and branch if non-zero. Branch forward or back up to 32kB.

image-20220420191124680

除了这些与 PC 有关的指令外，A64 指令集还包括两个绝对分支。BR Xn 指令执行一个绝对分支到 Xn 中的地址，而 BLR Xn 具有相同的效果，但也将返回地址存储在 X30 中（链接寄存器）。RET 指令的作用与 BR Xn 相似，但它提示分支预测逻辑它是一个函数返回。RET 的默认分支是指向 X30 中的地址，当然也可以指定其他寄存器。

A64 指令集包括一些特殊的条件性分支。这些分支在某些情况下可以提高代码密度，因为不需要明确的比较。

- CBZ Rt，标签//如果为零，则进行比较和分支

- CBNZ Rt，标签//比较和分支，如果不是零

这些指令将 32 位或 64 位的源寄存器与零进行比较，然后有条件地执行一个分支。分支偏移的范围是 ±1MB。这些指令不读取或写入条件代码标志（NZCV）。

有两种类似的测试和分支指令:

- TBZ Rt, bit, label. //测试和分支，如果 Rt 为零
- TBNZ Rt, bit, label //如果 Rt 不是零，则测试和分支

这些指令测试源寄存器中由立即指定的位位置上的位，并根据该位是否被设置或清除而有条件地进行分支。分支偏移的范围是 ±32kB。与 CBZ/CBNZ 一样，这些指令不读取或写入条件代码标志（NZCV）。

6.5 6.5 系统控制和其他指令

A64 指令集包含与以下内容相关的说明:

- 异常处理。
- 系统注册访问。
- 调试。
- 提示指令，在许多系统中都有电源管理应用程序。

6.5.1 6.5.1 异常处理指令

有三条异常处理指令，其目的是导致异常的发生。这些指令用于调用运行在操作系统（EL1）、管理程序（EL2）或安全监控器（EL3）中更高的异常级别的代码:

- SVC #imm16 //主管调用，允许应用程序调用内核//（EL1）。
- HVC #imm16 //虚拟机管理程序调用，允许操作系统代码调用虚拟机管理程序（EL2）。
- SMC #imm16 //安全监视器调用，允许操作系统或虚拟机管理程序调用安全//监视器（EL3）。

异常综合寄存器中的处理程序可以获得立即数。这是对 ARMv7 的改变，在 ARMv7 中，立即数必须通过读取调用指令的操作码来确定。更多信息请参见第 10 章 AArch64 异常处理。

要从异常中返回，请使用 ERET 指令。这条指令通过将 SPSR_ELn 复制到 PSTATE 来恢复处理器的状态，并分支到 ELR_ELn 中保存的返回地址。

6.5.2 6.5.2 系统寄存器访问

系统寄存器访问提供了两项说明：

- MRS Xt,

例如：MRS X4, ELR_EL1 // Copies ELR_EL1 to X4

- MSR , Xt

  例如：MSR SPSR_EL1, X0 // Copies X0 to SPSR_EL1

PSTATE 的个别字段也可以用 MSR 或 MRS 访问。例如，要选择与 EL0 相关的堆栈指针或当前的异常级别。

- MSR SPSel, #imm // A value of 0 or 1 in this register is used to select // between using EL0 stack pointer or the current exception

  // level stack pointer

这些指令有特殊形式可用于清除或设置单个异常掩码位（见第 4-5 页的保存进程状态寄存器）：

- MSR DAIFClr, #imm4
- MSR DAIFSet, #imm4

请参阅第 4-7 页的系统寄存器。

6.5.3 6.5.3 调试指令

有两个与调试相关的说明：

- BRK #imm16 // 进入监视器模式调试，其中有片上调试监视器代码
- HLT #imm16 // 进入停止模式调试，其中连接外部调试硬件

有关调试的信息，请参阅第 18 章调试。

6.5.4 6.5.4 提示指令

HINT 指令在规则上可以被当作 NOP 处理，但是它们可以产生特定的实现效果：

- NOP // No operation - not guaranteed to take time to execute
- YIELD // Hint that the current thread is performing a task that // can be swapped out
- WFE // Wait for Event
- WFI // Wait for interrupt
- SEV // Send Event
- SEVL // Send Event Local

这些概念也包含在第 14 章多核处理器和第 15 章电源管理中。

6.5.5 6.5.5 NEON 指令

NEON 指令集也有一些增强功能，其中一些是相当重要的。第 7 章 AArch64 浮点和 NEON 更详细地描述了这些。

A64 中 NEON 的变化包括:

- 支持双精度浮点，使得用双精度浮点的 C 代码能够向量化。
- 对存储在 NEON 寄存器中的标量数据进行操作的新指令。
- 插入和提取矢量元素的新指令。
- 用于类型转换和饱和整数运算的新指令。
- 浮点值规范化的新指令。
- 新的跨线指令用于向量减少、求和、取最小或最大值。
- 执行诸如比较、加法、查找绝对值和否定等操作的指令已被扩展到能够对 64 位整数元素进行操作。

6.5.6 6.5.6 浮点指令

A64 提供了一套类似于 ARMv7-A VFPv4 扩展的浮点指令，它提供了对标量浮点值的单精度和双精度数学运算。有一些变化和新功能:

- 浮点比较直接设置条件标志（NZCV）。在 A64 中，无需显式地将比较结果从浮点传输到整数标志。
- 已经添加了有关 IEEE754-2008 标准的说明，例如计算一对数字的最小值和最大值。
- 在从整数到浮点格式的转换中，现在可以明确指定舍入模式。当需要在特定的舍入模式下进行简单转换时，不再需要设置全局 FPCR 标志。其中一些选项也适用于 ARMv8 A32 和 T32。
- 添加了支持 64 位整数和浮点格式之间转换的说明。
- 在 A64 中，涉及整数类型的浮点操作直接在整数寄存器上工作。不需要手动在浮点和整数寄存器之间转移整数值进行转换操作。

6.5.7 6.5.7 加密指令

ARMv8 的可选扩展增加了加密指令，大大改善了 AES 加密以及 SHA1 和 SHA256 散列等任务的性能。

CHAPTER

SEVEN

7 AARCH64 浮点数和 NEON

我们把与 ARM 系列高级处理器 SIMD 架构相关实现和其对软件的支持通常被称为 NEON 技术。实际上对于 AArch32（相当于 ARMv7 NEON 指令）和 AArch64 均有 NEON 指令集。两者均可用于处理大量数据中重复操作的情形，且具有显著的效果，因此可以用在多媒体领域。

AArch64 的 NEON 架构使用 32 × 128 位寄存器，是 ARMv7 的两倍。这些寄存器与浮点指令使用的寄存器相同。所有编译过的代码和子例程都符合 EABI，它指定在特定的子例程中哪些寄存器的内容可以修改，哪些寄存器必须保留。

所有标准 ARMv8 实现都需要浮点数和 NEON。然而，针对特定市场，目前浮点数,SMID 和 NEON 有以下几种组合：

• 没有 NEON 或浮点。

• 支持完整的异常捕获浮点数和 SIMD。

• 支持完整的浮点数和 SIMD，无异常捕获

7.1 7.1 AArch64 中 NEON 和浮点数的新功能

AArch64 NEON 在现有 AArch32 NEON 的基础上进行了以下改动:

> 现在有 32 个 128 位寄存器，而在 ARMv7 中只具有 16 个

> 较小的寄存器不再被打包到较大的寄存器中，而是一对一地映射到 128 位寄存器的低位。单精度浮点数使用低 32 位，而双精度浮点数使用 128 位寄存器的低 64 位。详细内容请参考本章第二小节的内容.

>ARMv7-A NEON 指令中的前缀 V 被移除。

> 向向量寄存器写入 64 位或更少位的值会导致高位被清零。

> 在 AArch64 中，无法采用通用寄存器上执行 SIMD 或饱和算术指令，该类操作需要使用 NEON 寄存器。

> 增加了新的通道插入和提取指令，用来支持新的寄存器打包方案。

> 提供了用于生成或使用 128 位向量寄存器的高 64 位的附加指令。数据处理指令已经被分割成单独的指令，他会生成一个以上的结果寄存器 (扩大到 256 位向量)，或消耗两个源 (缩小到一个 128 位向量)，。

> 一组新的向量规约操作提供了跨通道和 (across-lane sum)、最小值和最大值

> 一些现有的指令已经扩展到支持 64 位整数值。如比较、加法、绝对值和取反指令，包括饱和版本。

> 饱和指令进行了扩展，把无符号累加包含到了有符号中，将有符号累加包含到了无符号累加中

>AArch64 NEON 支持双精度浮点和全精度浮点操作，包括舍入模式、非规范化数字和 NaN 处理。

通过以下更改，AArch64 中的浮点功能得到了增强：

> 将 ARMv7-A 浮点指令中的前缀 V 替换为 F

> 支持 IEEE 754 浮点标准定义的单精度（32 位）和双精度（64 位）浮点向量数据类型和算法，支持 FPCR 舍入模式字段（FPCR Rounding Mode field）、默认 NaN 控件、刷新到零控件（Flush-to-Zero）和异常陷阱启用位（在实现方案支持的情况下）

>FP/NEON 寄存器的加载/存储寻址模式与整数加载/存储相同，包括加载或存储一对浮点寄存器的能力

> 支持浮点操作中的条件选择和比较指令，其等效于整数操作中的条件选择指令 CSEL 和比较指令 CCMP。

浮点指令 FCMP、FCMPE、FCCMP 和 FCCMP 会根据浮点指令的结果来设置标志 PSTATE.{N，Z，C，V}，但不会修改浮点状态寄存器（FPSR）中的条件标志，如同 ARMv7 中那样。

> 所有浮点乘加（Multiply-Add）和乘减 (Multiply-Subtrac) 指令都是融合的。

融合乘法是在 VFPv4 中引入的，这意味着乘法的结果在用于加法之前不会四舍五入。在早期的 ARM 浮点体系架构中，乘法累加操作将会对执行中间结果和最终结果的四舍五入操作，这种操作可能会导致损失精度的。

> 提供了额外的转换操作，例如，64 位整数和浮点之间以及半精度和双精度之间的转换操作。

将浮点转换为整数的指令（FCVTxU、FCVTxS）将直接采有向舍入的方式进行编码：

—趋近于 0

—趋近于正无穷

—趋近于负无穷

—与偶数关系最近

—距离最近

> 添加了具有把浮点数四舍五入到最近整数的指令 FRINTx

> 一种新的双精度到单精度的向下转换指令，其不精确取整到奇数，可以通过 FCVTXN 来进行向下转化到半精度。

> 添加了 FMINNM 和 FMAXNM 指令，这两个指令用来实现 IEEE754-2008 中的操作 minNum () 和 maxNum ()。如果其中一个操作数是静态 NaN，则返回数值

> 增加了加速浮点向量规范化的指令（FRECPX、FMULX）

7.2 7.2 NEON 和浮点架构

NEON 寄存器的内容是具有相同数据类型元素的向量。向量被划分为通道，每个通道包含一个称为元素的数据值。

NEON 向量中的通道数取决于向量的大小及其向量中的数据元素。

通常，每个 NEON 指令有 n 个并行操作，其中 n 就是输入向量被划分的通道数。不能从一个通道进位或者溢出到另一个通道的。向量中元素的顺序是从最低有效位开始的。这意味着元素 0 使用寄存器的低有效位。

NEON 和浮点指令可以对以下类型的元素进行操作：

>32 位单精度和 64 位双精度浮点数。

注意：16 位浮点数也是支持的，但它仅作为转换自/到的数据格式，不支持数据处理操作。

>8 位、16 位、32 位或 64 位无符号和有符号整数。

>8 位和 16 位多项式。

多项式类型用于代码, 比如进行错误纠正，使用的是有限域的 2 的幂或者 {0,1} 上的简单多项式，普通 ARM 整数代码通常使用查找表来进行有限域运算。AArch64 NEON 提供了使用大型查找表的指令。

多项式运算很难从其他运算中合成出来，因此有一个基本的乘法运算是非常有必要的，我们可以从这个乘法运算合成其他更复杂的运算。

NEON 将寄存器看作:

32 个 128 位的四字寄存器 V0-V31，如下图 7-1 所示:

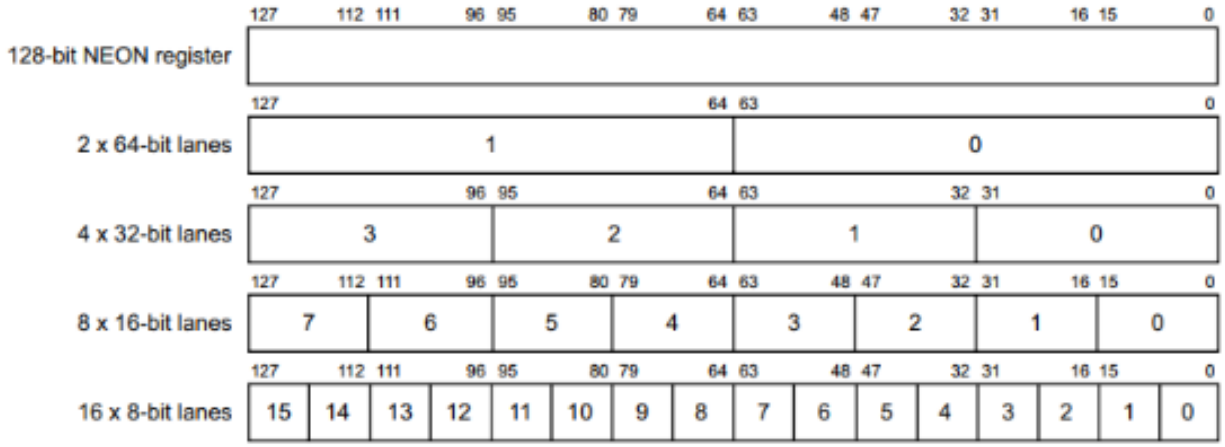

图 1 V 类寄存器划分

32 个 64 位 D 或双字寄存器 D0-D31，每个寄存器如图 7-2 所示：

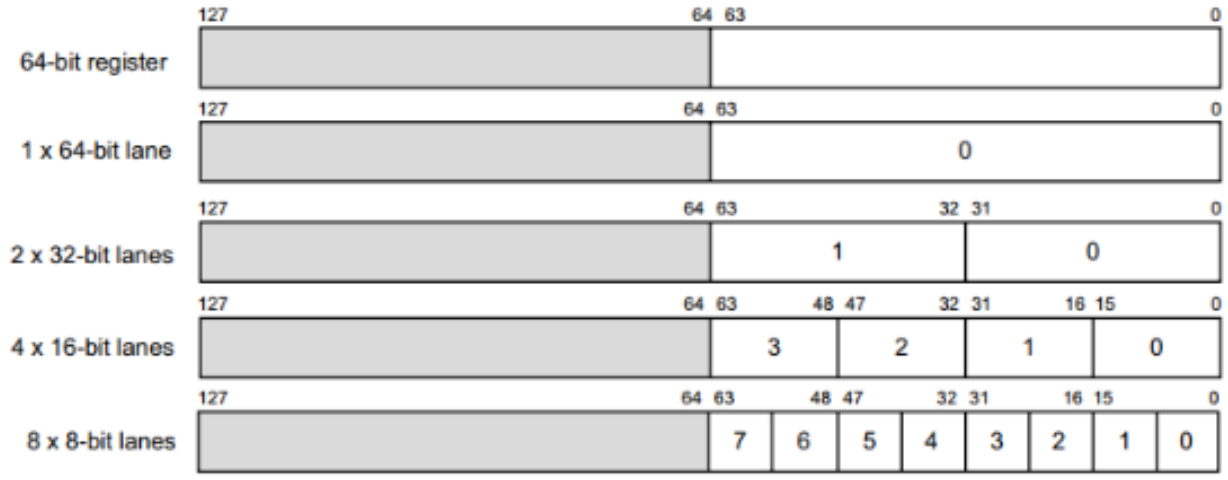

图 2 D 类寄存器划分

所有这些寄存器都可以在任何时候访问。软件不需要显式地在它们之间切换，因为使用的指令决定了适当的视图。

7.2.1 7.2.1 浮点

在 AArch64 中，浮点单元将 NEON 寄存器看作:

>32 个 64 位寄存器 D0-D31。D 寄存器称为双精度寄存器，其寄存器中数据是双精度浮点数。

>32 个 32 位寄存器 S0-S31。S 寄存器称为单精度寄存器，其寄存器中数据是单精度浮点数。

>32 个 16 位寄存器 H0-H31。H 寄存器称为半精度寄存器，其寄存器中数据是半精度浮点数。

> 上述 3 种寄存器的组合。

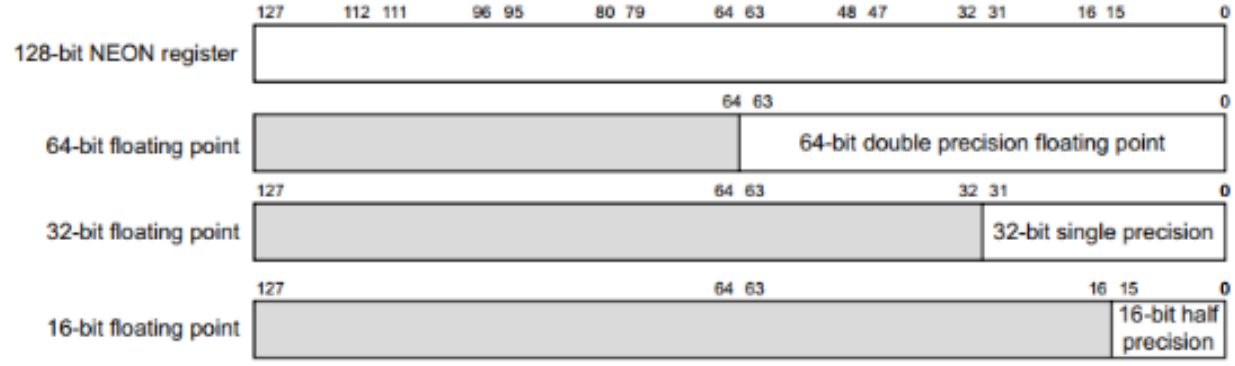

图 3 浮点寄存器划分

7.2.2 7.2.2 标量数据和 NEON

标量数据指的是单个值，而不是包含多个值的向量。有些 NEON 指令使用标量操作数。寄存器内的标量是通过向量的索引值来访问。

访问向量中单个元素的通用的数组表示法是:

Vd.Ts[index1], Vn.Ts[index2]

其中:

Vd 是目标寄存器。

Vn 是第一个源寄存器。

Ts 是元素的大小说明符。

Index 是元素的索引。

如下面的例子:

INS V0.S[1], V1.S[0]

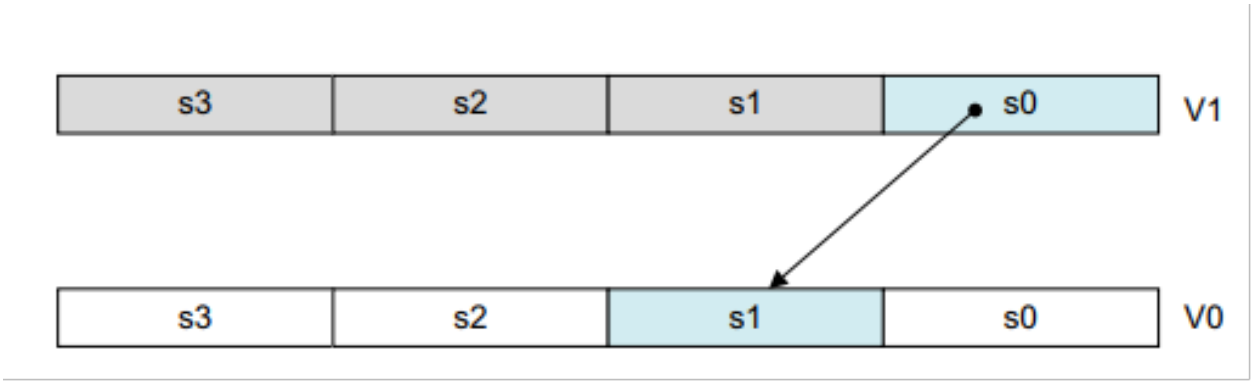

图 4 插入一个元素到向量中 INS V0.S[1], V1.S[0]

在指令 MOV V0.B[3], W0 中，将寄存器 W0 的最低字节复制到寄存器 V0 的第四个字节。

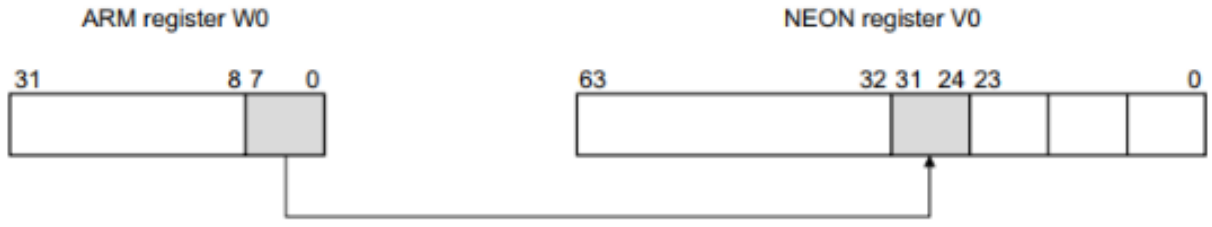

图 5 移动一个标量到通道中 MOV V0.B[3], W0

NEON 标量可以是 8 位、16 位、32 位和 64 位。除了乘法指令，访问标量的指令可以访问寄存器中的任何元素。

乘指令只允许 16 位或 32 位标量，并且只能访问寄存器中的前 128 位的标量:

>16 位标量被限制在寄存器 Vn.H[x] 中，其中 0≤n≤15。

>32 位标量被限制在寄存器 Vn.S[x] 中。

7.2.3 7.2.3 浮点参数

浮点参数是通过浮点寄存器传递给函数 (或从函数返回)。整数寄存器 (通用寄存器) 和浮点寄存器都可以同时使用。这意味着浮点参数在浮点 H、S 或 D 寄存器中传递，而其他参数在整数 X 或 W 寄存器中传递。根据 AArch64 调用标准，调用时会强制在需要浮点运算的地方采用浮点硬件单元，因此在 AArch64 状态下没有软件浮点链接方式。

在 ARMv8-A 架构参考手册中给出了详细的指令列表，在此处仅列出了主要的浮点数据处理操作，用来说明其可以完成的功能:

7.3 7.3 AArch64 NEON 指令格式

为了同 AArch64 的整数和标量浮点指令集的语法一致，NEON 和浮点指令的语法做过不少改动。其指令助记符基于 ARMv7。

> ARMv7 NEON 指令中的前缀 V 已被移除

一些助记符被重命名，因为移除前缀 V 导致了与 ARM 指令集助记符的冲突。

这意味着，存在一些指令具有相同的名字，做相同的事情，它们可以是 ARM 核心指令，NEON，或浮点指令，这取决于指令的语法，例如:

ADD W0, W1, W2{, shift #amount}}

和

ADD X0, X1, X2{, shift #amount}}

是 A64 基本指令。

ADD D0, D1, D2

是标量浮点指令，

```
ADD V0.4H, V1.4H, V2.4H
```

是 NEON 向量指令

> 添加了 S、U、F 或 P 前缀来表示有符号、无符号、浮点或多项式 (仅其中一种) 数据类型。该助记符指出了操作的数据类型。例如:PMULL V0.8B, V1.8B, V2.8B

> 向量结构 (元素的大小和通道的数量) 由寄存器限定符描述。例如:

ADD Vd.T, Vn.T, Vm.T

其中 Vd, Vn 和 Vm 是寄存器名，T 是要使用的寄存器的细分。对本例中，T 是排列说明符，可以是 8B、16B、4H、8H、2S、4S 或 2D 中的一个。根据使用的是 64 位、32 位、16 位还是 8 位的数据，以及使用的是 64 位还是 128 位的寄存器。

要添加 2 × 64 位通道，可以使用 ADD V0.2D, V1.2D, V2.2D

> 在 ARMv7 中，一些 NEON 数据处理指令存在 Normal, Long，wide，narrow 和饱和版本。其 Long, Wide 和 Narrow 的版本用后缀表示：

普通指令可以对任何向量类型进行操作，并生成与操作数向量大小相同、类型通常相同的结果向量。

长指令或加长指令对双字向量操作数进行操作，并生成四字向量结果。结果元素的宽度是操作数宽度的两倍。长指令是在指令后面附加 L 来指定的。例如：SADDL V0.4S, V1.4H, V2.4H

图 7-6 给了长指令的演示，操作数在操作之前被提升。

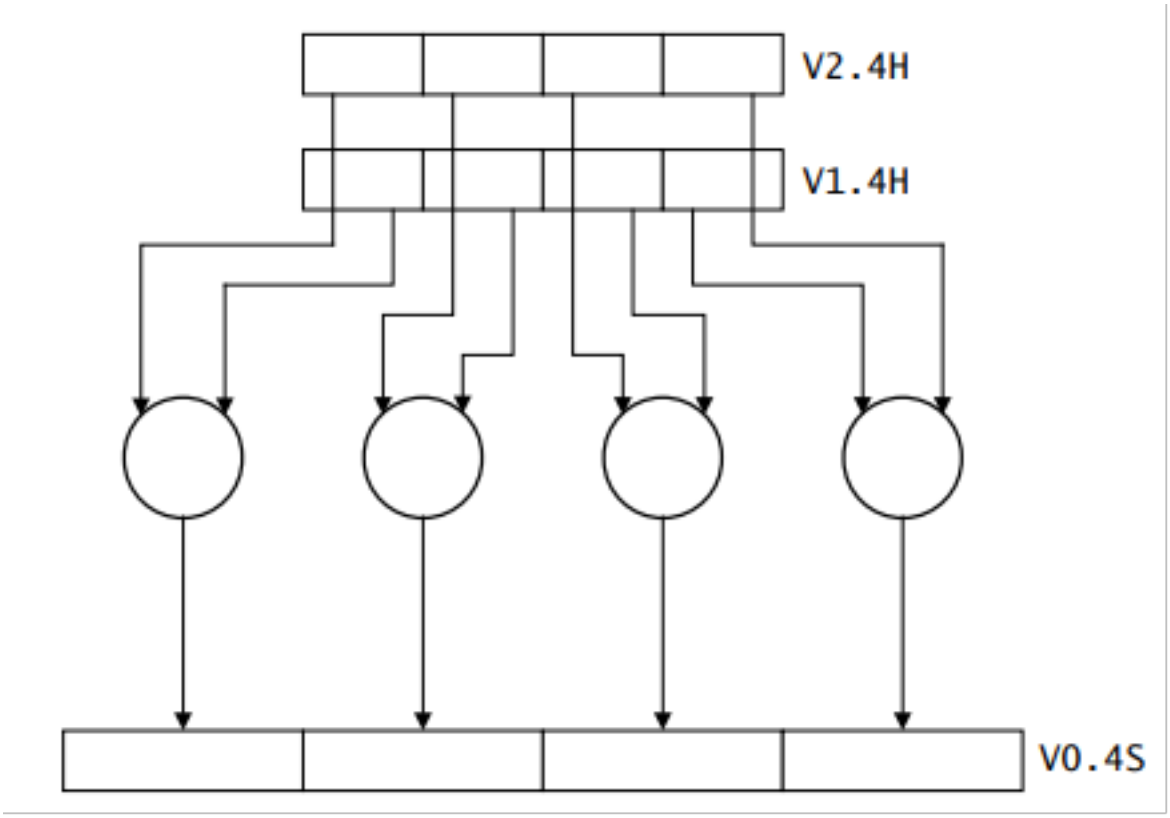

图 6 NEON 长指令

宽指令或加宽指令 (Wide or Widening) 对双字向量操作数和四字向量操作数进行操作，生成的是四字向量结果。结果元素和第一个操作数的宽度是第二个操作数元素宽度的两倍。宽指令是在指令后面附件 W 来表示的。例如：SADDW V0.4S, V1.4H, V2.4S

图 7-7 给了宽指令的演示，输入的双字操作数在操作之前被提升。

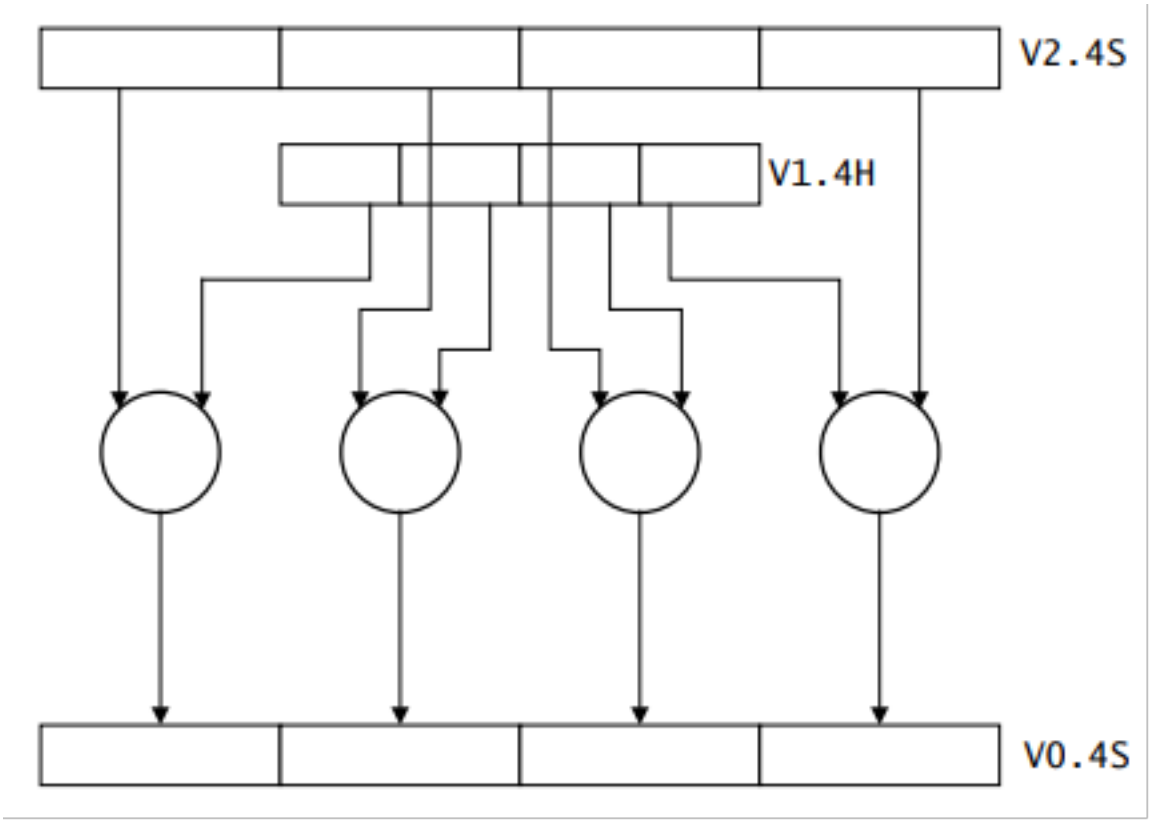

图 7 NENO 宽指令

窄化或窄化 (Narrow or Narrowing) 指令对四字向量操作数进行操作，并生成双字向量结果。结果元素通常是操作数元素宽度的一半。窄指令是在指令后面附件 N 来表示的。例如：SUBHN V0.4H, V1.4S, V2.4S。

图 7-8 给了窄指令的演示，输入的双字操作数在操作之前被降级。

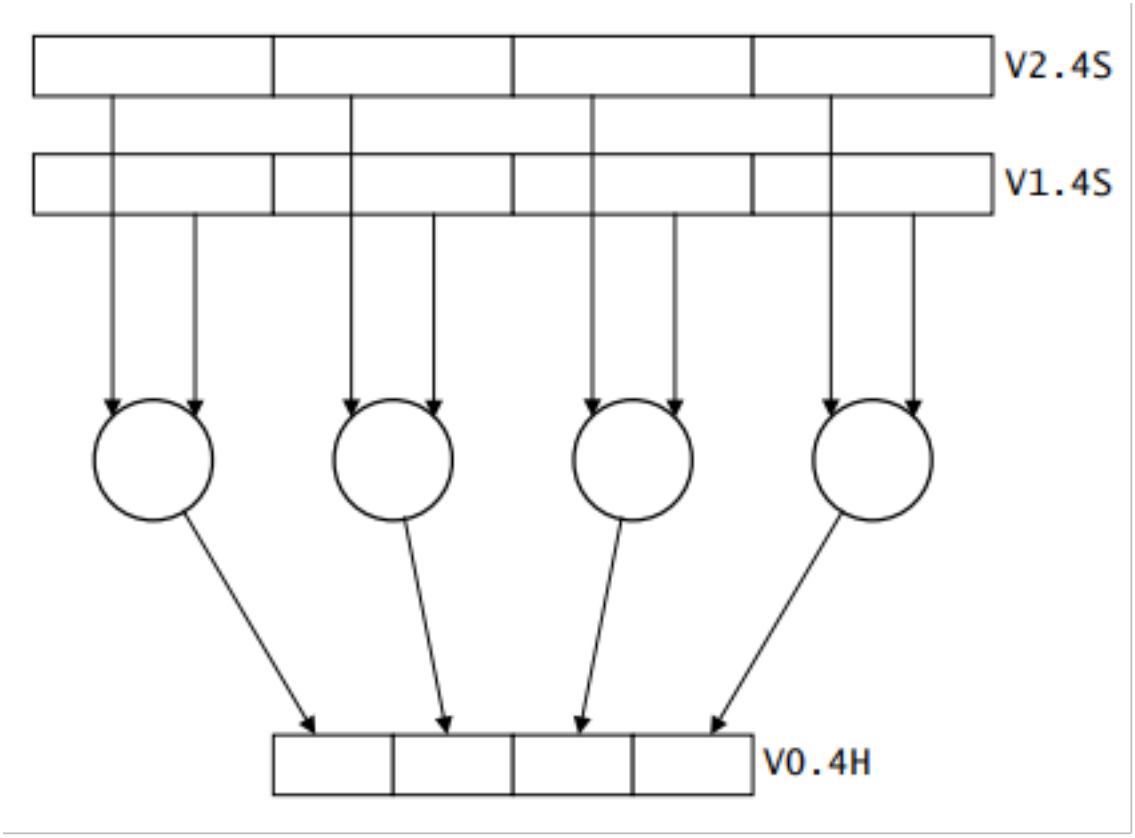

图 8 NEON 窄指令

> 有符号和无符号饱和变量 (由前缀 SQ 或 UQ 标识) 可用于许多指令，如 SQADD 和 UQADD。如果结果超过数据类型的最大值或最小值，饱和指令将返回该最大值或最小值。饱和限制取决于指令的数据类型。

> 在 ARMv7 中成对操作的前缀是 P，而在 ARMv8 中，P 是后缀，例如在 ADDP 中，成对指令对相邻的双字或四字操作数对进行操作。例如: ADDP V0.4S, V1.4S, V2.4S

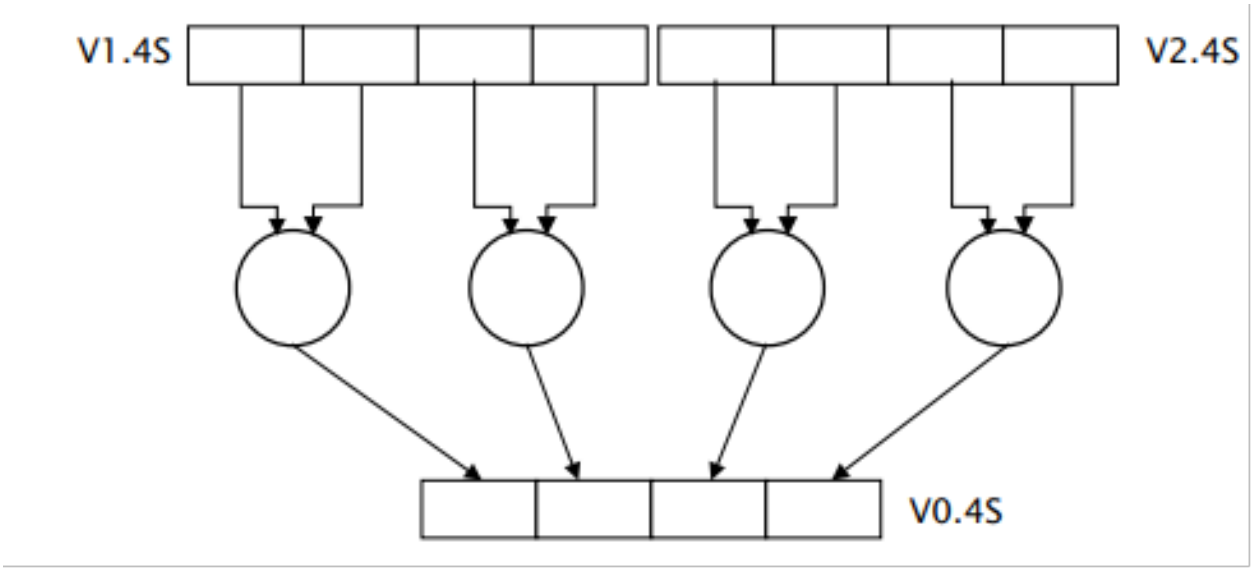

图 9 对操作

> 添加后缀 V 用于跨通道 (整个寄存器) 操作，例如 ADDV S0, V1.4S

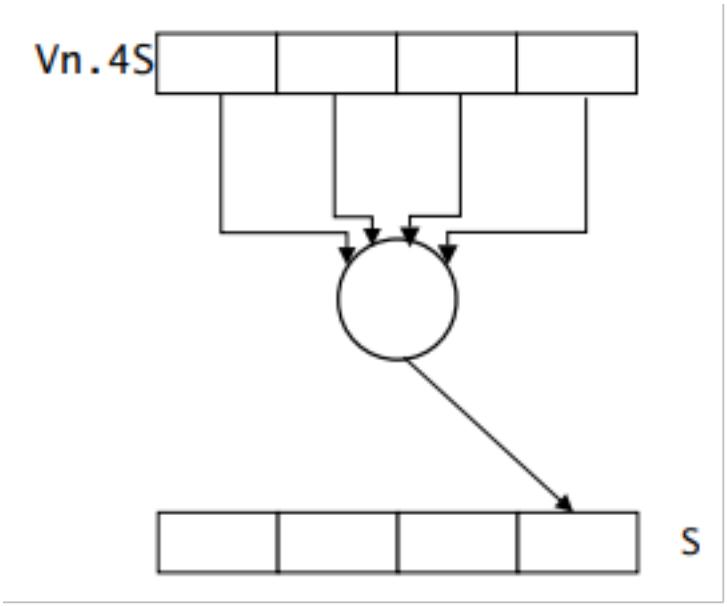

图 10 跨通道操作

> 后缀 2，称为第二和上半说明符，添加这个后缀用来扩大、缩小或加长第二部分指令。如果后缀 2 存在，它会导致在存放较窄元素寄存器的高 64 位上执行操作:

后缀为 2 的加宽指令从包含较窄值的向量高编号通道获取输入数据，并将扩展后的结果写入 128 位的目的地址。例如: SADDW2 V0.2D, V1.2D, V2.4S

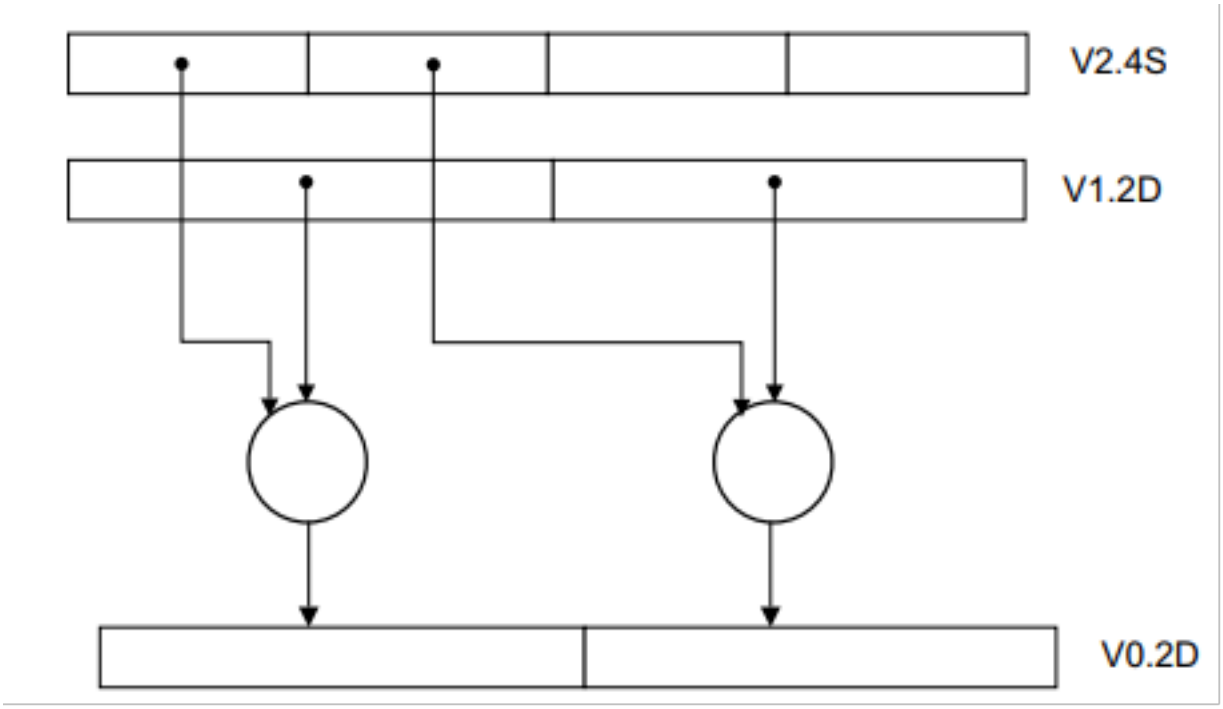

图 11 SADDW2

后缀为 2 的窄指令从 128 位源操作数获取其输入数据，并将其窄化结果插入 128 位的目的地，并保持低通道不变。例如: XTN2 V0.4S, V1.2D

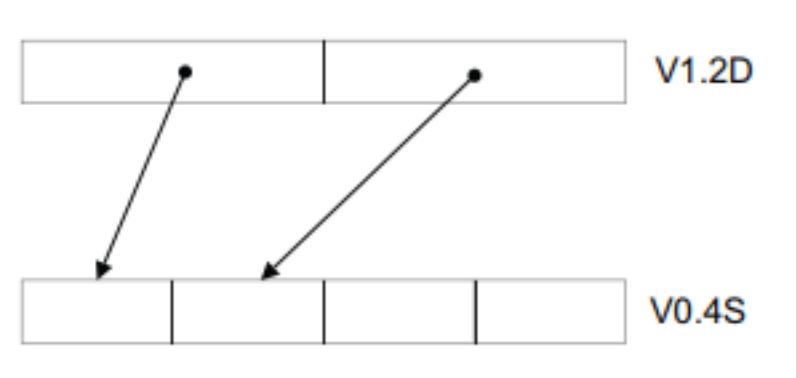

图 12 XTN2

后缀为 2 的长指令从 128 位源向量的高编号通道获取输入数据，并将加长后的结果写入 128 位的目的地。例如: SADDL2 V0.2D, V1.4S, V2.4S

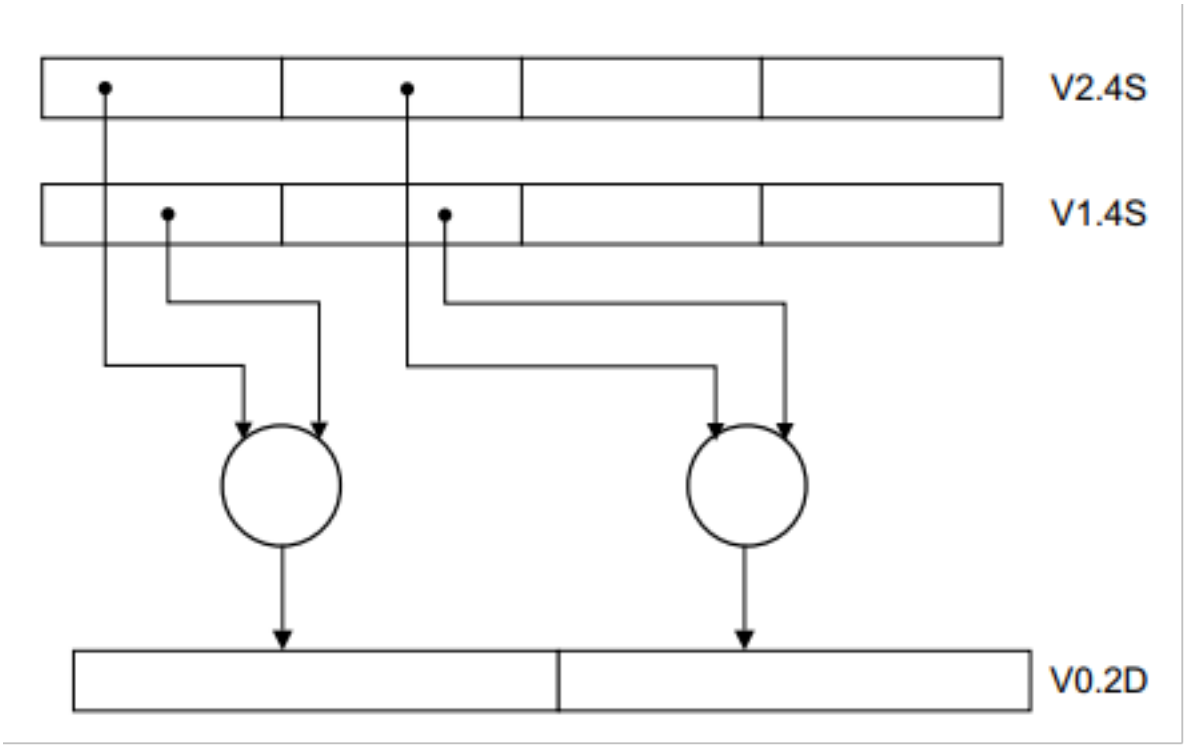

图 13 SADDL2

> 比较指令使用条件代码名来指示条件是什么以及条件是有符号的还是无符号的（如果适用），例如，CMGT 和 CMHI、CMGE 和 CMHS。

7.4 7.4 NEON 编程的选择

NEON 的代码有许多种写法。此处我简要列出了几种方法 (您可以参阅 ARM NEON Programmers Guide 获取更多细节)。这包括使用内部函数（intrinsic）, C 代码的自动向量化，库的使用，采用汇编语言等方法。

内部函数是 C 或 C++ 伪函数调用，编译器将替换为对应的 NEON 指令。因此我们可以采用 NEON 中可用的数据类型和操作，同时允许编译器处理指令调度和寄存器分配。这些原语在 ARM 扩展语言文档中都有定义。

自动矢量化由 ARM 编译器 6 中的-fvectorize 选项进行控制，但当采用更高的优化级别（-O2 及以上）时会自动启用。即使指定了-fvectorize，自动矢量化也会在级别-O0 时禁用。因此，我们可以采用下面的命令在优化级别是-O1 时启用自动矢量化：

armclang –target=armv8a-arm-none-eabi -fvectorize -O1 -c file.c

有各种库可以使用 NEON 代码。但这些库会随着时间的推移而变化，因此本文档中不包括当前这方面的内容。

尽管从技术角度考虑，NEON 汇编是可以进行手动优化的，但因为管道和内存访问时序是具有复杂的相互依赖性，因此 NEON 优化是相当困难的。ARM 强烈建议使用内部函数 (intrinsic)，而不是直接使用汇编：

- 使用 instrinsic 编写代码会比使用汇编助记符更容易
- instrinsic 为跨平台开发提供了良好的可移植性
- 无需担心管道和内存访问
- 通常情况下，其性能较好

如果您不是一个擅长使用汇编编写代码的程序员，那同汇编相比，采用 instrinsic 通常能获得更好的性能。Instrinsic 提供了几乎与汇编语言一样多的功能，且编译器会复杂寄存器的分配，这样您就可以专注于代码算法部分，同汇编语言相比，这将会更有利于您对源代码的维护。

CHAPTER

EIGHT

8. 移植到 A64

本章不打算作为一个详细的为所有系统编写可移植代码的指南，但是，覆盖了工程师应该知道的在 ARM 处理器上移植代码的主要领域。在将代码从 A32 和 T32 指令集移动到 AArch64 中的 A64 指令集时，您应该注意一些显着差异：

- A32 指令集中的大多数指令都可以有条件地执行。也就是说，可以将条件代码附加到指令，并根据先前标志设置指令的结果执行（或不执行）指令。尽管这使编程技巧能够减少代码大小和周期数，但这会使具有乱序执行的高性能处理器的设计变得非常复杂。

  在操作码字段中保留操作码（predication）的必要位可以有用地用于其他目的（例如，为从更大的通用目的寄存器池中进行选择提供空间-译注：不明白）。因此，在 A64 代码中，只有一小部分指令可以有条件地执行，而一些比较和选择指令取决于执行条件。请参阅第 6-8 页的条件指令。

- 许多 A64 指令可以对源寄存器或仅受操作数大小限制的寄存器应用任意常量移位。此外，A64 提供了非常有用的扩展寄存器形式。需要明确的指令来处理更复杂的情况，例如变量移位。T32 也比 A32 更具约束性，因此在某些方面 A64 是相同原则的延续。A32 的 Operand2 在 A64 中并不存在，但各自有类似的指令。

- 加载和存储指令的可用寻址模式有一些变化。A32 和 T32 中的偏移量、前置-索引和后置-索引形式在 A64 中仍然可用。A64 引入一种新的 PC 相对寻址模式，因为不能使用与通用寄存器相同的方式访问 PC。~~A64 加载指令可以以内联方式替换寄存器 ~~（尽管不如 A32 灵活），并且可以使用一些扩展模式（例如，可以使用 32 位数组索引）

  A64 从以前的 ARM 架构中删除了所有复合内存访问指令（加载或存储多个-译注：LDM/STM），这些指令能够从内存中读取或写入任意寄存器列表。现在 A64 应使用可对任意两个寄存器进行操作的加载 (LDP) 和存储 (STP) 指令。PUSH 和 POP 也已被删除。

- ARMv8 添加了包括单向内存栅栏（barrier）的加载和存储指令：加载-获取（load-acquire）和存储-释放（store-release）。这些在 ARMv8 A32 和 T32 以及 A64 中可用。加载-获取指令要求任何后续内存访问（编程顺序）仅在加载获取之后可见。store-release 确保在 store-release 变得可见之前所有早期的内存访问都是可见的。请参阅第 6-18 页的内存屏障和栅栏说明。

  译注：Acquire：Acquire 之后的所有内存操作将发生在 Acquire 操作之后。Release：Release 确保所有先于 Release 的内存操作将先于 Release 操作发生

- AArch64 不支持协处理器的概念，包括 CP15。新的系统指令（todo: 列举一下指令）允许访问通过 AArch32 中的 CP15 协处理器指令访问的寄存器。
- AArch64 中没有 CPSR 这个寄存器。PSTATE（NZCV 等标志）可以通过特殊寄存器、操作码访问访问。

  译注：

```
指令： DAIFSet DAIFClr  特殊寄存器： NZCV DAIF CurrentEL
```

- 对于许多应用程序，将代码从旧版本的 ARM 架构或其他处理器架构移植到 A64 意味着只需重新编译源代码。但是，在许多领域 C 代码不能完全移植。A64 和 A32/T32 之间的相似性在以下示例中说明。下面的三个序列显示了一个简单的 C 函数和第一个 T32 和 A64 中的输出代码。两者的对应关系很容易看出来。

```
// C code
int foo(int val)
{
    int newval = bar(val);
    return val + newval;
}
```

```
//␣
↪T32                                                                    /
↪/ A64
foo:                                                                   foo:
SUB SP, SP,
↪#8                                                          SUB SP,␣
↪SP #16
STRD R4, R14,␣
↪[SP]                                                     STP X19, X30,␣
↪[SP]
MOV R4,␣
↪R0                                                                   MOV␣
↪W19, W0
BL␣
↪bar                                                                          BL␣
↪bar
ADD R0, R0,␣
↪R4                                                          ADD W0,␣
↪W0, W19
LDRD R4, R14,␣
↪[SP]                                                     ADD SP, SP, #16
ADD SP, SP,
↪#8                                                          RET
BX LR
```

A64 提供的通用功能是从 A32 和 T32 的功能中演化而来，因此在这两者之间移植代码比较简单。将 A32 汇编代码转换为 A64 的也很简单。大多数指令在这些指令集之间很容易映射，许多（执行）序列在 A64 中变得更简单。

8.1 8.1 字节对其

数据和代码必须满足适当的内存对其方式。访问对齐会影响 ARM 内核的性能，并且在将代码从早期架构迁移到 ARMv8-A 时可能会出现可移植性问题。出于性能原因，或者在移植对指针或 32 位和 64 位整数变量做出假设的代码时，需要注意对齐问题。

以前版本的 ARM 编译器语法程序集提供了 ALIGN n 指令，其中 n 以字节为单位指定对齐边界。例如，指令 ALIGN 128 将地址对齐到 128 字节边界。

GNU 汇编语法（ARM Complier 6 语法）提供了.balign n 指令，它使用与 ALIGN 相同的格式。

注意：

> GNU 语法汇编还提供了.align n 指令。但是，n 的格式因系统而异。.balign 指令提供与.align 相同的对齐功能，并在所有架构中具有一致的行为
>
> > 译注：.balign, b 感觉应为 boundary 的意思，边界对其。译注：gnu-gcc 汇编.align 2 是 2^2，4 字节对其的意思。

在从旧编译器迁移到 ARM Compiler 6 时，您应该将 ALIGN n 的所有实例转换为.balign n。

8.2 8.2 数据类型

在 64 位机器上的 C 和 C 派生语言的许多编程环境中，int 变量仍然是 32 位宽，但长整数和指针是 64 位宽。这些被描述为具有 LP64 数据模型（译注：lp64 指一系列数据类型，见下表格）。本章假设为 LP64，尽管其他数据模型可用，请参见第 5-7 页的表 5-1。ARM ABI 为 LP64 定义了许多基本数据类型。其中一些可能因架构而异，如下所示：

Table 8-1 Basic data types

Type	A32	A64	Description
int/long	32-bit	32-bit	integer
short	16-bit	16-bit	integer
char	8-bit	8-bit	byte
long long	64-bit	64-bit	integer
float	32-bit	32-bit	single-precision IEEE floating-point
double	64-bit	64-bit	double-precision IEEE floating-point
bool	8-bit	8-bit	Boolean
wchar_t[a]	16-bit unsigned	16-bit unsigned	short (compiler dependent)
	32-bit unsigned	32-bit unsigned	int (compiler dependent)
void* pointer	32-bit	64-bit	addresses to data or code
enumerated types	32-bit	32-bit[b]	signed or unsigned integer
bit fields	not larger than their natural container size		
ABI defined extension types			
__int128/__uint128	128-bit	128-bit	signed/unsigned quadword
__f16	16-bit	16-bit	half precision

a. Environment-dependent. In GNU-based systems (such as Linux) this type is always 32-bit.

b. If the set of values in an enumerated type cannot be represented using either int or unsigned int as a container type, and the language permits extended enumeration sets, then a long long or unsigned long long container may be used.

image-ch8.2-table8-1

译注：a. 对于一些 linux 环境，wchar_t 总是为 32bit

b. 如果定义枚举类型时没有显示的指定变量为 int 或者 unsigned（使用 0x0 赋值某个枚举类型），而且没有指定-fshort-enums, 则枚举类型的长度有可能是 unsigned long long 或者 long long

在将 AArch64 与以前版本的 ARM 架构进行比较时，由于通用寄存器和操作都是 64 位，通常可以更有效地处理 64 位数据类型。int 仍然是 32 位的，可以通过通用寄存器（W 寄存器）有效地处理它。然而，指针是指向数据或代码的 64 位地址。ARM ABI 将 char 默认定义为无符号。对于以前版本的架构也是如此。如果您的代码不以不可移植的方式操作指针，例如转换为非指针类型或从非指针类型转换或执行指针算术的情况，移植将变的简单。这意味着你从未将指针存储在 int 变量中（可能有 intptr_t 和 uintptr_t 例外），并且从未将指针转换为 int。有关这方面的更多信息，请参阅将代码从 32 位环境移植到 64 位环境时遇到的问题（第 8-8 页）。

除其他影响外，这会更改大小，并可能更改结构和参数列表的对齐方式。当存储大小很重要的情

况下，可以使用 stdint.h 中的 int32_t 和 int64_t 类型。注意 size_t 和 ssize_t 在 AAPCS64-LP64 中都是 64 位的。 出于性能原因，编译器尝试在自然大小（译注：参考 linux 对其方式）边界上对齐数据。大多数编译器都试图优化编译模块中全局数据的布局。

AArch64 支持 16、32、64 和 128 位数据非对齐访问，其中使用的地址不是要加载或存储的数量的倍数。但是，独占加载（loaded) 或存储 (stored) 以及加载-获取 (load-acquire) 或存储-释放 (store-release) 指令只能访问对齐的地址 (译注：比如 ldp，stp 指令始终要满足 64bit 对其)。这意味着用于构造信号量（sem、mutex）和其他锁（lock）机制的变量通常必须对齐（译注：还有压栈、出栈等情况）。

注意

在正常情况下，所有变量都应该对齐。在大多数情况下，非对齐访问的平均效率仍然低于对齐访问。对于系统中的其他 CPU 或总线控制，未对齐的访问永远不能保证是原子的。此规则的唯一主要例外是访问 packed 的数据结构——在通过文件或网络连接等方式将数据进行组织（译注：打包数据），为发送或接收外部数据节省大量精力。

与对齐访问相比，未对齐访问可能会对性能产生影响。在自然大小边界上对齐的数据可以更有效地访问，而未对齐的访问可能会花费额外的总线或缓存周期。应该使用 packed 属性 (*__attribute__*((packed,aligned(1))) 警告编译器潜在的未对齐访问，例如在手动转换指向不同数据类型的指针时。

译注：这里我理解的是，packed 取消默认的自然对其方式，采用实际大小进行对其，~~~ TODO：举例不明白想表达什么意思。 ~~~

8.2.1 8.2.1 汇编代码

许多 A32 汇编指令可以很容易地替换为类似的 A64 指令。不幸的是目前没有自动化机制。然而，很多东西可以相当简单地转换。下表显示了 A32/T32 和 A64 指令集在许多方面的紧密匹配。

Table 8-2 Instructions that are similar for A32 and A64

A32	A64
ADD Rd,Rn,#7	ADD Wd,Wn,#7
ADDS Rd,Rn,Rm,LSL #2	ADDS Wd,Wn,Wm,LSL #2
B label	B label
BFI Rd,Rn,#lsb,#wid	BFI Wd,Wn,#lsb,#wid
BL label	BL label
CBZ Rn,label	CBZ Wn,label
CLZ Rd,Rm	CLZ Wd,Wm
LDR Rt,[Rn,#imm]	LDR Wt,[Xn,#imm]
LDR Rt,[Rn,#imm]!	LDR Wt,[Xn,#imm]!
MOV Rd,#imm	MOV Wd,#imm
MUL Rd,Rn,Rm	MUL Wd,Wn,Wm
RBIT Rd,Rm	RBIT Wd,Wm

image-ch8.2-table8-2

但是，在许多有重写需求的地方存在差异。下表显示了其中的一些。

image-

MOVT	MOVK
MOVW	MOVZ
BX lr	
MOV pc, lr	RET
BX <reg>	BR <reg>
MLA	MADD
STMIA r0, {r1, r2}	STP X1, X2, [X0], #8
LDMIA r0, {r1, r2}	LDP X1, X2, [X0], #8
POP {r0-r1}	

ch8.2-table8-3

**Note **

64-bit APCS 需要栈按照 128-bit(16 byte) 对其

表 8-4 展示了如何通过 PSTATE 各个字段的别名来替代 CPSR 寄存器

Table 8-4 Use of named fields

	A32	A64
CPSR is replaced with a set of separate registers and fields		
Disable IRQ	MRS R0, CPSR ORR R0, R0, #IRQ_Bit MSR CPSR_c, R0 CPSID i	MSR DAIFSET, #IRQ_bit
ALU Flags	MRS R0, CPSR MSR CPSR_f, R0	MRS X0, NZCV MSR NZCV, X0
Set Endianness	SETEND BE	SCTLR_ELn.EE controls EL*n* data endianness SCTLR_EL1.E0E controls EL0 data endianness MRS X0, SCTLR_EL1 ORR X0, X0, #EE_bit MSR SCTLR_EL1, X0 See *Endianness* on page 4-12.

imge-ch8.2-table8-4

T32 条件执行编译为第 8-6 页、表 8-4 的 A32 列中所示的序列。在 A64 中，它使用了新的条件选择指令，如 A64 列所示。以下示例说明了两个指令集（T32 和 A64）中条件执行之间的区别：

```
// C code
int gcd(int a, int b)
{
    while (a != b) {
        if (a > b) {
            a = a - b;
        } else {
            b = b - a;
```

(continues on next page)

(continued from previous page)

```
        }
    }
    return a;
}
```

```
//A32                                                    // A64
gcd:                                                   gcd:
CMP R0, R1                                                   SUBS W2, W0, W1
ITE                                                                CSEL W0, W2, W0, gt
SUBGT R0, R0, R1                                       CSNEG W1, W1, W2, gt
SUBLE R1, R1, R0                                       BNE gcd
BNE gcd                                                          RET
BX lr
```

8.3 8.3 移植 32-bit 代码到 64-bit 时遇到的问题

移植 C 代码到 64 位环境中运行时可能会出现一些常见问题。这些问题不只是在 ARM 架构中出现。

- 需要注意指针和整数变量，因为它们的大小可能不同。ARM 建议使用 stdint.h 中的 uintptr_t 或 intptr_t 将指针类型作为整数值处理。指针运算中使用的偏移量应声明为 ptrdiff_t，因为使用 int 可能会产生不正确的结果。
- 64 位系统具有更大的潜在的内存访问，32 位 int 可能不足以索引数组中的所有条目。
- C 表达式中的隐式类型转换可能会产生一些意想不到的效果。注意确保使用的任何常量值与掩码本身具有相同的类型。
- 使用不同长度或符号的数据类型执行操作时要小心。例如，当无符号和有符号 32 位整数在表达式中混合并将结果分配给有符号长整数时，可能需要将其中一个操作数显式转换为其 64 位类型。这会导致所有其他操作数也被提升为 64 位。请注意，long 通常是 A64 (LP64) 上的 64 位类型。

8.3.1 8.3.1 代码重写和再编译

任何移植都不可避免地需要重新编译和重写代码。大多数情况下的目标是尽可能使用前者而不是后者。

好消息是很多代码只需要重新编译。但是，请谨慎行事，因为许多基本数据类型的大小将发生变化。尽管编写良好的 C 代码不应该对单个类型的大小有太多依赖，但你很可能会遇到一些。

因此，最佳实践必须是在重新编译时启用所有警告和错误，并确保您注意到编译器发出的任何警告，即使代码看起来编译时没有错误。密切注意代码中的任何显式类型转换，因为当底层类型的

大小发生变化时，这些通常是错误的根源。

8.3.2 8.3.2 ARMv8-A 使用 ARM Compiler 6 的一些选项

向编译器提供正确的选项以生成代码或为 ARMv8-A 生成目标代码. 以下是可用的选项：

使用 –**target** 为指定的目标生成代码。

–**target** 选项是强制性的，没有默认值。您必须始终指定目标体系结构。

Syntax

–target=triple

triple 为 architecture-vendor-OS-abi 的架构

已经支持的 target 选项如下：

aarch64-arm-none-eabi : ARMv8-A 架构的 AArch64 模式。

armv8a-arm-none-eabi : ARMv8-A 架构的 AArch32 模式。

armv7a-arm-none-eabi： ARMv7-A 架构

例如: –target=armv8a-arm-none-eabi

Note

–target 选项是 armclang 选项。对于所有其他工具，例如 armasm 和 armlink，使用–cpu 和–fpu 选项来指定目标处理器和架构。

译注：对于 gun gcc 还有两个参数-march -mtune 可以参考

使用–mcpu 选项为特定 ARM 处理器生成代码。

如下：

-mcpu=+(no)crc - 使能或者禁止 crc 指令

-mcpu=+(no)crypto - 使能或者禁止加解密引擎

-mcpu=+(no)fp - 使能或者禁止 floating point 扩展

-mcpu=+(no)simd - 使能或者禁止 NEON 指令

processor 可以是 cortex-a53 或者 cortex-a57，（译注：感觉 a55 也是可以的）

为 AArch32 架构编译代码时会产生与 ARMv7-A 非常相似的代码。虽然 AArch32 有一些新指令（例如 Load-Acquire 和 Store-Release），并且 SWP 指令已被删除，但这些指令通常不是由编译器生成的。使用 +nosimd 选项进行编译可避免使用任何 NEON/Floating-point 指令或寄存器。这对于 NEON 单元未使能的系统或特定代码段（例如重置代码和异常处理程序）可能很有用，确保其中不使用 NEON/Floating-point 非常重要。默认编译代码时为无加密扩展，但使用 NEON。

8.4 8.4 C 代码的一些建议

- 使用 sizeof() 代替常量，例如：

```
(void **) calloc(4, 100);
```

替换为：

```
(void **) calloc(sizeof(void*), 100);
```

或者最好使用：

```
void *a; (void **) calloc(sizeof(a), 100);
```

- 如果需要显示的（explicit）定义数据类型，最后使用 stdin.h
- 如果需要将指针强制转换为整数，请使用保证能够容纳该指针的类型，例如 uintptr_t。

  如果你不关心实际的指针代表的类型，类型 *bob；bob++ 仍然是首要使用的方式，指针运算更适用于潜在的（基础）数据类型。（**译注：不明白他想说什么**）
- 当使用结构体成员时（结构体）需要注意，此时数据大小和布局很重要，如下：

  ```
  struct {void *a; int b; int c} bob;
  ```

  不建议使用

  ```
  struct {int b; void *a; int c;}
  ```

  这种与 AAPCS64 中规定的一样，元素 a 在它之前插入了 32 位填充以保持 64 位对齐。
- 适当使用 size_t (译注：当表示数据大小的类型时)
- 当假定数据类型大小时，应该谨慎使用 limits.h
- 恰当的使用函数、宏、built-ins 应用在数据类型

  例如：使用 atol(char*) 替换 atoi（char*)
- 使用原子操作时，使用正确的 64 位的函数操作 64 位数据类型。
- 不要假设对同一结构中不同位域的操作是独立处理的——在 64 位平台上比在 32 位平台上可以读取和写入的位更多。
- 对 Long 数据类型使用‘L’时，32-bit 架构位 32bit，64bit 架构位 64bit。如下表达式需要选择正确的类型：

```
long value = 1L << (超过了 32 位);
```

对于编译 32 位和编译 64 位指令时，当数据长度需要求是 64 位时，必须使用 LL 或者 ULL 后缀。

- 或者，你可以使用 C99 中 stdint.h 提供的宏，可以通过这些宏使用字面数值（比如：1，2，3）

  ```
  size_t value = UINT64_C(1) << SOMANY;
  ```

8.4.1 8.4.1 显示和隐示转换

当不同长度和不同符号的数据类型混合在表达式中时，C/C++ 中的内部提升和类型转换可能会导致一些意想不到的问题。特别是，要了解在表达式计算时进行转换的时间点。

例如：

int + long => long

unsigned int + signed int => unsigned int

int64_t + uint32_t => int64_t

如果在提升到 long 之前执行了丢失了符号位的转换，那么当分配给带符号的 long 时，结果可能不正确。

如果在表达式中混合了无符号和有符号 32 位整数并将结果分配给有符号 64 位整数，则将非 64 位操作数转换为其 64 位类型。这会导致其他操作数提升为 64 位，并且在表达式中不需要进一步的转换。

另一种解决方案是强制转换整个表达式，以便在赋值时发生符号扩展。但是，对于这些问题，没有一种万能的解决方案。在实践中，修复它们的最佳方法是了解代码试图做什么。

在如下例子中，你希望 a 的结果位-1：

```
long a;
int b;
unsigned  int c;
b = -2;
c = 1;
a = b + c;
```

当 long 为 32 位长时 a = -1（表示为 0xFFFFFFFF），对于 64 位长 a = 0x00000000FFFFFFFF（或十进制的 4 294 967 295）。

显然这是一个意想不到的非常错误的结果！这是因为 b 在加法之前被转换为 unsigned int （以匹配 c），所以加法的结果是一个无符号整数。

一种可能的解决方案是在执行加法之前强制转换为更长的数据类型。

```
long a;
int b;
unsigned int c;
b = -2;
c = 1;
a = (long)b + c;
```

这样得到了二进制补码的 -1（或 0xFFFFFFFFFFFFFFFF）的预期的结果。计算是按照 64bit 算数运算执行，并转换为有符号数输出正确结果。

译注：无符号和有符号数运输时，会将有符号数转换为无符号数。

8.4.2 8.4.2 位操作

注意确保位掩码的宽度正确，C 表达式中的隐式类型转换可能会产生一些意想不到的效果。考虑以下用于在 64 位变量中设置指定位的函数：

```
long SetBitN(long value, unsigned bitNum)
{
    long mask;
    mask = 1 << bitNum;
    return value | mask;
}
```

32bit 的环境中这个函数没有问题，并且可以设置【31：0】bit。要将其移植到 64 位系统，您可能认为更改掩码类型以允许设置 [63:0] 位就足够了，如下所示：

```
long long SetBitN(long value, unsigned bitNum)
{
    long long mask;
    mask = 1 << bitNum;
    return value | mask;
}
```

同样，这不能正常工作，因为数字 1 为 int 类型。确切的行为取决于各个编译器的配置和假设。要使代码正确运行，您需要赋予常量与掩码相同的类型：

```
long long SetBitN(long value, unsigned bitNum)
{
    long long mask;
    mask = 1LL << bitNum;
    return value | mask;
}
```

如果您需要一个特定大小的整数，请使用 stdint.h 中的 uint32_t 和 UINT32_C 宏类型。

8.4.3 8.4.3 下标索引

在 64 位环境中使用大型数组或对象时，请注意 int 可能不再大到足以索引所有条目。特别是，在使用 int 下标遍历数组时要小心。

```
static char array[BIG_NUMBER];
for (unsigned int index = 0; index != BIG_NUMBER; index++) ...
```

由于 size_t 是 64 位类型，而 unsigned int 是 32 位类型，因此可能错误定义了一类数据（译注：BIG_NUMBER > 2^32) 使得循环永不终止。

CHAPTER

NINE

9. ARM 64 位架构的 ABI

ARM 体系结构的应用程序二进制接口 (ABI) 指定了所有可执行本机代码模块必须遵守的基本规则，以便它们可以正确地协同工作。这些基本规则由特定编程语言（例如，C++）的附加规则补充。除了 ARM ABI 指定的规则之外，个别操作系统或执行环境（例如 Linux）可能会指定额外的规则来满足他们自己的特定要求。

AArch64 架构的 ABI 有许多组件：

Executable and Linkable Format (ELF)

ARM 64 位架构 (AArch64) 的 ELF 指定目标（object）和可执行格式。

Procedure Call Standard (PCS)

ARM 64 位架构 (AArch64) ABI 版本的过程调用标准 (PCS) 指定子例程单独编写、编译和汇编之间如何协同工作。它指定调用例程和被调用者之间或例程与其执行环境之间的约定，例如当调用例程或堆栈布局时的约定。

DWARF

这是一种广泛使用的标准化调试数据格式。AArch64 DWARF 基于 DWARF 3.0，但有一些附加规则。有关详细信息，请参阅 ARM 64 位架构 (AArch64) 的 DWARF。

C and C++ libraries

参考 *ARM Compiler ARM C and C++ Libraries and Floating-Point Support User Guide* 。

C++ ABI

参考 *C++ Application Binary Interface Standard for the ARM 64-bit Architecture*。

9.1 9.1 AArch64 内部寄存器在函数调用中的传递标准

了解寄存器使用的标准会很有用。了解如何传递参数可以帮助你：

- 编写更高效的 C 代码。
- 了解反汇编代码。
- 编写汇编代码。
- 调用用不同语言编写的函数。

9.1.1 9.1.1 通用目的寄存器作为参数

出于函数调用的目的，通用寄存器分为四组：

Argument registers (X0-X7)：参数寄存器 x0-x7

这些寄存器用于将参数传递给函数并返回结果。它们可以用作临时寄存器或调用者保存的寄存器变量（译注：传入参数），可以在函数内保存中间值，或者函数调用之间传递参数。与 AArch32 相比，8 个寄存器可用于参数传递，这样减少了因参数过多放入堆栈的需要。

Caller-saved temporary registers (X9-X15)（调用者）

如果调用者需要在调用另一个函数时保留一些寄存器中的任何值，则调用者必须将受影响的寄存器保存在自己的堆栈帧中。也可以修改子例程，（译注：使用 x9-x15 实现相同功能，）而不需要在返回调用函数时保存值到堆栈和从堆栈中恢复它们。

Callee-saved registers (X19-X29)（被调用）

这些寄存器保存在被调用者内部。只要在返回前保存并恢复，就可以在被调用的子程序中修改。

Registers with a special purpose (X8, X16-X18, X29, X30) 特殊目的寄存器

- X8 是间接结果寄存器。这用于传递间接结果的地址位置，例如，函数返回大型结构的位置。（译注：不只是返回，函数内部有大的结构体也会使用）
- X16 和 X17 分别是 IP0 和 IP1，intra-procedure-call 临时寄存器

  这些可以通过 veneers 或者类似代码使用，或者作为子程序调用之间的中间值的临时寄存器。它们可以被函数破坏。veneers 代码是链接器自动插入的一小段代码，例如当分支（跳转）目标超出分支指令（支持的）范围时。
- X18 是平台寄存器，保留供平台 ABI 使用。这是平台上的一个附加临时寄存器，没有为其分配特殊含义。
- X29 是 Frame Pointer 寄存器（FP）
- X30 是链接寄存器 (LR)。

第 9-4 页的图 9-1 显示了 64 位 X 寄存器。有关寄存器的更多信息，请参阅第 4 章。有关浮点参数的信息，请参阅第 7-7 页的浮点参数。

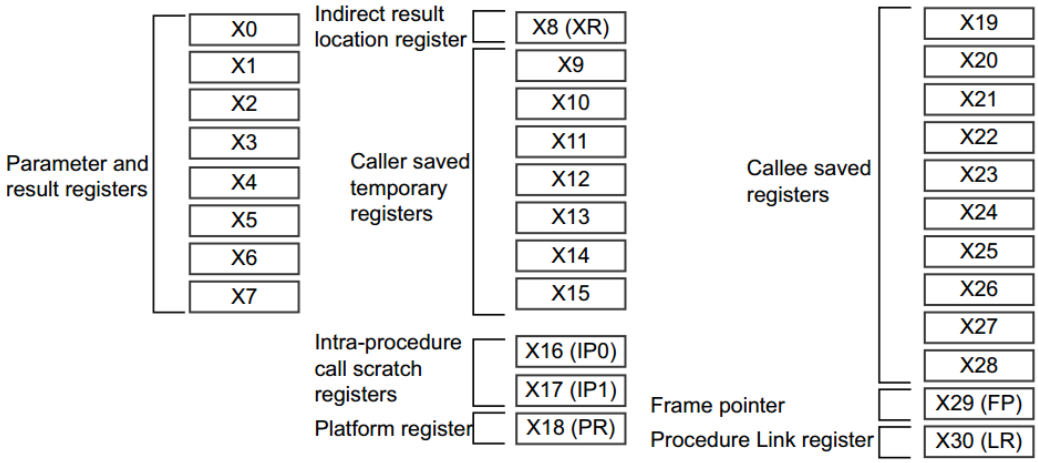

Figure 9-1 General-purpose register use in the ABI image-9-1

9.1.2 9.1.2 返回值的传递（Indirect result location）

再提一下，X8（XR）用于传递间接结果的位置，类似如下代码：

```
// test.c//
struct struct_A
{
    int i0;
    int i1;
    double d0;
    double d1;
}AA;
struct struct_A foo(int i0, int i1, double d0, double d1)
{
    struct struct_A A1;
    A1.i0 = i0;
    A1.i1 = i1;
    A1.d0 = d0;
    A1.d1 = d1;
    return A1;
}
void bar()
{
    AA = foo(0, 1, 1.0, 2.0);
}
```

可以用如下方法编译：

armclang -target aarch64-arm-none-eabi -c test.c fromelf-c test.o

注意

此代码未经优化编译以演示所涉及的机制和原理。通过优化，编译器可能会删除所有这些。

```
foo:
    SUB SP, SP, #0x30
    STR W0, [SP, #0x2C]
    STR W1, [SP, #0x28]
    STR D0, [SP, #0x20]
    STR D1, [SP, #0x18]
    LDR W0, [SP, #0x2C]
    STR W0, [SP, #0]
    LDR W0, [SP, #0x28]
    STR W0, [SP, #4]
    LDR W0, [SP, #0x20]
    STR W0, [SP, #8]
    LDR W0, [SP, #0x18]
    STR W0, [SP, #10]
    LDR X9, [SP, #0x0]
    STR X9, [X8, #0]
    LDR X9, [SP, #8]
    STR X9, [X8, #8]
    LDR X9, [SP, #0x10]
    STR X9, [X8, #0x10]
    ADD SP, SP, #0x30
    RET
bar:
    STP X29, X30, [SP, #0x10]!
    MOV X29, SP
    SUB SP, SP, #0x20
    ADD X8, SP, #8
    MOV W0, WZR
    ORR W1, WZR, #1
    FMOV D0, #1.00000000
    FMOV D1, #2.00000000
    BL foo
    ADRP X8, {PC}, 0x78                // 译注： x8 存放 AA 地址，bar 使用 sp 中的
返回值，复制给 AA
    ADD X8, X8, #0
    LDR X9, [SP, #8]
```

(continues on next page)

(continued from previous page)

```
    STR X9, [X8, #0]
    LDR X9, [SP, #0x10]
    STR X9, [X8, #8]
    LDR X9, [SP, #0x18]
    STR X9, [X8, #0x10]
    MOV SP, X29
    LDP X20, X30, [SP], #0x10 // 译注： 原文这里好像写错了应该是 X29 寄存器
    RET
```

在此示例中，该结构包含超过 16 个字节。根据 AArch64 的 AAPCS，返回的对象被写入 XR 指向的内存。

通过生成的代码可以看到：

- W0、W1、D0 和 D1 用于传递整数和双精度参数。
- bar() 在堆栈上为 foo() 的返回结构值腾出空间，并将 sp 放入 X8。
- bar() 将 X8 与 W0、W1、D0 和 D1 中的参数一起传递给 foo()，然后 foo() 获取地址以进行进一步的操作。
- foo() 可能会损坏 X8，因此 bar() 使用 SP 访问返回结构。

使用 X8 (XR) 的优点是它不会降低用于传递函数参数的寄存器的可用性。

AAPC64 栈帧如图 9-2 所示。帧指针 (X29) 应该指向保存在栈上的前一个帧指针，保存的 LR (X30) 存储在它之后。链中的最后一个帧指针应设置为 0。堆栈指针必须始终在 16 字节边界上对齐。堆栈帧的确切布局可能存在一些变化，特别是在可变参数或无帧函数的情况下。有关详细信息，请参阅 AAPCS64 文档。

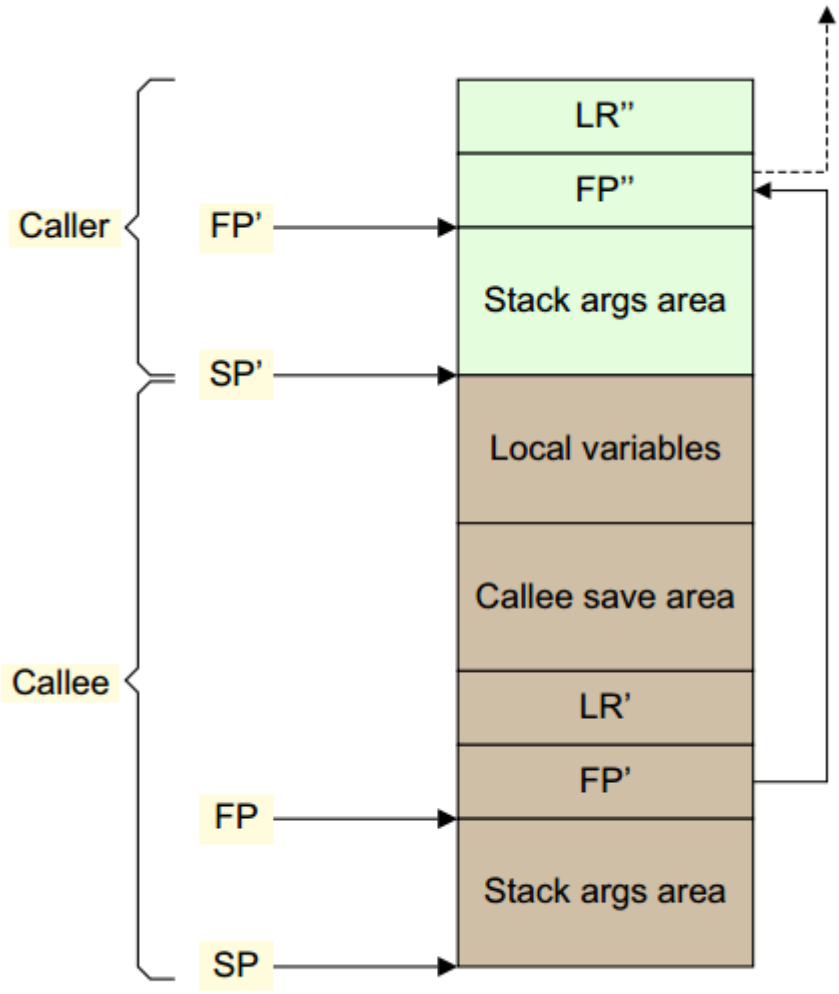

Figure 9-2 Stack frame image-ch9.1-figure9-2

注意

AAPCS 仅指定 FP、LR 块布局以及这些块如何链接在一起。图 9-2 中的其他所有内容（包括两个函数栈帧之间边界的精确位置）都未指定，编译器可以自由选择。

图 9-2 说明了一个使用两个被调用者保存的寄存器（X19 和 X20）和一个临时变量的帧，其布局如下（左边的数字是从 FP 偏移的字节数）：

```
40: <padding>
32: temp
24: x20
16: x19
```

(continues on next page)

(continued from previous page)

```
8: LR'
0: FP'
```

填充是维持栈指针的 16 字节对齐所必需的。

```
function:
    STP X29, X30, [SP, #-48]!        // Push down stack pointer and store FP␣
↪and LR
    MOV X29, SP                                          // Set the frame␣
↪pointer to the bottom of the new frame
    STP X19, X20, [X29, #16]         // Save X19 and X20
    : :
    Main body of code
    : :
    LDP X19, X20, [X29, #16]         // Restore X19 and X29
    LDP X29, X30, [SP], #48                 // Restore FP' and LR' before␣
↪setting the stack
                                                              // pointer to␣
↪its original position
    RET                                                   // Return to caller
```

9.1.3 9.1.3 NEON 和 Floating-Point 寄存器作为参数

ARM 64 位架构也有 32 个寄存器，v0-v31，可供 NEON 和浮点运算使用。通过改变指代寄存器的名称更改访问的大小。

注意

与 AArch32 不同，在 AArch64 中，NEON 和浮点寄存器的 128 位和 64 位的视图在狭义视图中不存在多个寄存器叠加（组成一个寄存器的情况），因此 q1、d1 和 s1 都指的是寄存器组中相同入口。

译注：AArch32 中 Q0[127:0] = S3[31:0] + S2[31:0] + S1[31:0] + S0[31:0]

Q1[127:0] = S7[31:0] + S6[31:0] + S5[31:0] + S4[31:0]

AArch64 中 Q0[127:0] 中的 [31:0] 与 S0[31:0] 对应

Q1[127:0] 中的 [31:0] 与 S1[31:0] 对应

所以 AArch64 入口相同，长度不同，而 AArch32 由 4 个 S 组成一个 Q，多个寄存器叠加而生成一个寄存器。

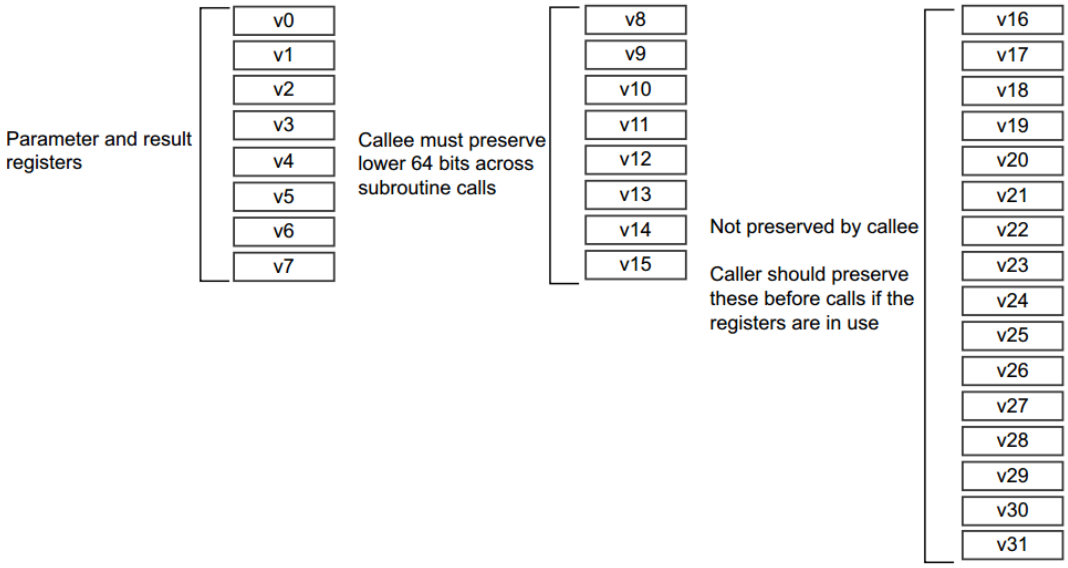

Figure 9-3 SIMD and floating-point registers in the ABI image-ch9.1.3-figure9-3

- V0-V7 用于将参数值传递给子例程并从函数返回结果值。它们也可用于在例程中保存中间值（但通常仅在子例程调用之间）。
- 经由子例程调用的被调用例程（callee routine）中必须保存 V8-V15。只需要保存 V8-V15 中存储的每个值的低 64 位。
- V16-V31 不需要保留（或应该由调用者保留）。

CHAPTER

TEN

10 AARCH64 异常处理

严格来说，中断是说软件执行流程的东西，但是，在 arm 术语中，统称为异常。异常是需要特权软件（异常处理程序）执行某些操作以确保系统顺利运行的条件或系统事件。每种异常类型都有一个异常处理程序。一旦处理完异常，特权软件就会让内核准备好恢复它在处理异常之前所做的任何事情。下面介绍了几种异常：

Interrupt：

一般有两种，分为 irq 和 fiq。fiq 的优先级高于 IRQ，这两种异常通常都与内核上的输入引脚相关。假设中断未被禁用，外部硬件断言了一个中断请求并在当前指令完成执行时触发相应的异常类型（irq or fiq),fiq 和 irq 对 core 来说都是物理信号，当被断言时，如果内核当前已启用，则会发生相应的异常。在几乎所有系统上，各种中断源都使用中断控制器连接。中断控制器对中断进行仲裁并确定其优先级，进而提供串行化的单个中断信号，然后将其连接到内核的 FIQ 或 IRQ 信号。有关详细信息，请参阅通用中断控制器。由于 IRQ 和 FIQ 中断的发生在任何时间，与内核正在执行的软件没有直接关系，因此它们被归类为异步异常。

Aborts（中止）：

在指令获取失败（指令中止）或数据访问失败（数据中止）时生成中止。它可来源于内存访问错误时的外部内存系统（可能表明指定的地址与系统中的实际内存不对应）。也可来源于 core 内的内存管理单元 (MMU) 生成中止。操作系统可以使用 MMU 中止来为应用程序动态分配内存。

当一条指令被提取时，它在流水线中被标记为 abort。仅当内核随后尝试执行指令时，才会发生指令中止异常。异常发生在指令执行之前。如果在中止指令到达流水线执行阶段之前清除了流水线，则不会发生中止异常。数据中止异常是由于加载或存储指令而发生的，并且被认为是在数据读取或写入之后发生的。

如果 abort 是由于执行或试图执行指令流而生成的，并且返回的地址提供了引起中止指令的详细信息，则该中止被描述为同步 abort。

异步 abort 不是通过执行指令生成的，而返回地址可能并不总是会提供引起 abort 原因的详细信息。在 ARMv8-A 中，指令和数据 abort 是同步的。异步异常包括 `IRQ/FIQ` 和 `System errors (SError)`。参考 *Synchronous and asynchronous exceptions*。

Reset：

复位被视为最高异常级别的特殊向量。

这是 ARM 处理器在触发异常时的指令跳转位置。这个向量使用 IMPLEMENTATION DEFINED 地址。RVBAR_ELn 包含此复位向量地址，其中 n 是实现的最高异常级别的编号。所有内核都有一个复位输入，

并在复位后立即发生复位异常。它是最高优先级的异常，不能被屏蔽。此异常用于在上电后在内核上执行代码以对其进行初始化。

生成异常的指令

执行某些指令会产生异常。通常执行此类指令可以从运行更高权限级别的软件请求服务：

- Supervisor Call (SVC) 指令可以使用户模式程序能够请求操作系统（OS）服务。
- Hypervisor Call (HVC) 指令使客户操作系统（guest OS）能够请求管理程序服务。
- Secure monitor Call（安全监控 SMC) 指令使非安全世界请求安全世界服务。

如果异常是由于在 EL0 处取指令而产生的，则将其视为 EL1 的异常，除非 HCR_EL2.TGE 位设置为非安全状态，在这种情况下将其视为 EL2。如果异常是由于任何其他异常级别的指令提取而生成的，则异常级别保持不变。

在本书的前面部分，我们看到 ARMv8-A 架构有四个异常级别。处理器执行只能通过获取异常或从异常返回从而在异常级别之间移动。当处理器从较高的异常级别移动到较低的异常级别时，执行状态可以保持不变，也可以从 AArch64 切换到 AArch32。相反，当从较低的异常级别移动到较高的异常级别时，执行状态可以保持不变或从 AArch32 切换到 AArch64。

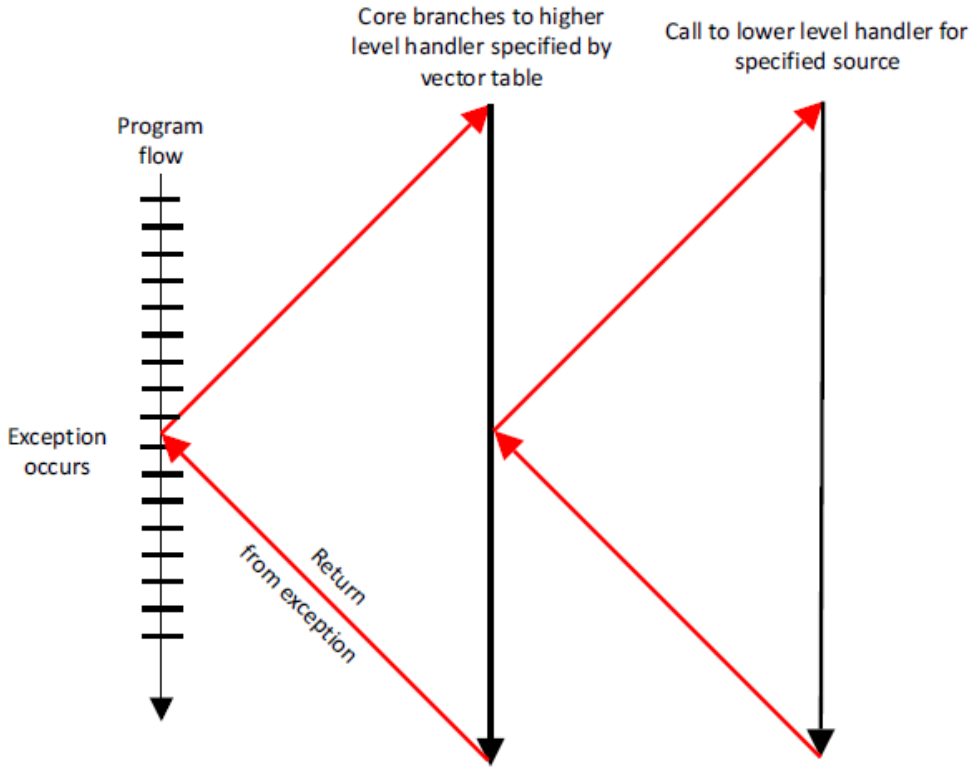

Figure 10-1 Exception flow Untitled

图 10-1 示意性地显示了运行应用程序时发生的异常程序流程。异常发生后，处理器分支到一个向量表，其中包含每个异常类型的条目。向量表包含一个调度代码，该代码通常识别异常的原因，选择并调用适当的函数

来处理它。此代码完成执行，然后返回到高级处理程序，然后执行 ERET 指令来返回到应用程序。

10.1 10.1 异常处理寄存器

第 4 章描述了处理器的当前状态是如何存储在单独的 PSTATE 字段中。如果发生异常，则 PSTATE 信息被保存在 Saved Program Status Register 中。(SPSR_ELn) 存在于 SPSR_EL3、SPSR_EL2 和 SPSR_EL1 中

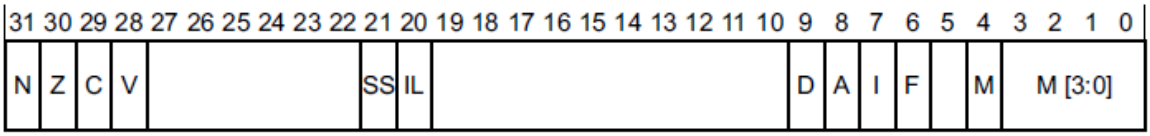

Figure 10-2 When exceptions are taken from AArch64

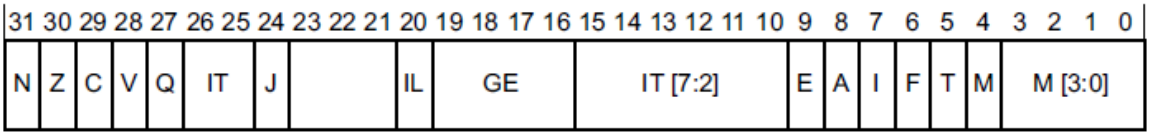

Figure 10-3 When exceptions are taken from AArch32

Untitled

SPRSR.M 字段（第 4 位）用于记录执行状态（0 表示 AArch64，1 表示 AArch32）。

Table 10-1 PSTATE fields

PSTATE fields	Description
NZCV	Condition flags
Q	Cumulative saturation bit
DAIF	Exception mask bits
SPSel	SP selection (EL0 or EL*n*), not applicable to EL0
E	Data endianness (AArch32 only)
IL	Illegal flag
SS	Software stepping bit

Untitled

异常屏蔽位 (DAIF) 允许屏蔽异常事件。该位置 1 时不发生异常。**D：**debug 异常 mask。**A：**SError 中断 mask，例如异步外部中止。**I：** IRQ 中断 mask。**F：**FIQ 中断 mask。

SPSel 字段选择是否应使用当前异常级别堆栈指针或 SP_EL0。这可以在任何异常级别设置，除了 EL0。(**译注:** 本来就是为了选择 EL0 和非 EL0 的 SP，在 EL0 就不用选）这将在本章后面讨论。

IL 字段在设置时后，会导致执行的下一条指令触发异常。被使用做非法异常返回，例如配置为 AArch32 时，试图以 AArch64 返回 EL2。

第 18 章调试中介绍了软件步进 (SS) 位。调试器使用它来执行单条指令，然后在下一条指令上发生调试异常。

其中一些分散的 field（CurrentEL、DAIF、NZCV 等）在发生异常时被合并在一起复制到 SPSR_ELn 寄存器中（返回时则相反）。

当导致异常的事件发生时，处理器硬件会自动执行某些动作。SPSR_ELn 被更新（其中 n 是发生异常的异常级别），SPSR_ELn 用来存储在异常结束时需要返回的正确的 PSTATE 信息。PSTATE 被更新以反映新的处理器状态（这可能意味着异常级别被提高，或者它可能保持不变）。在异常结束时使用的返回地址存储在 ELR_ELn 中。

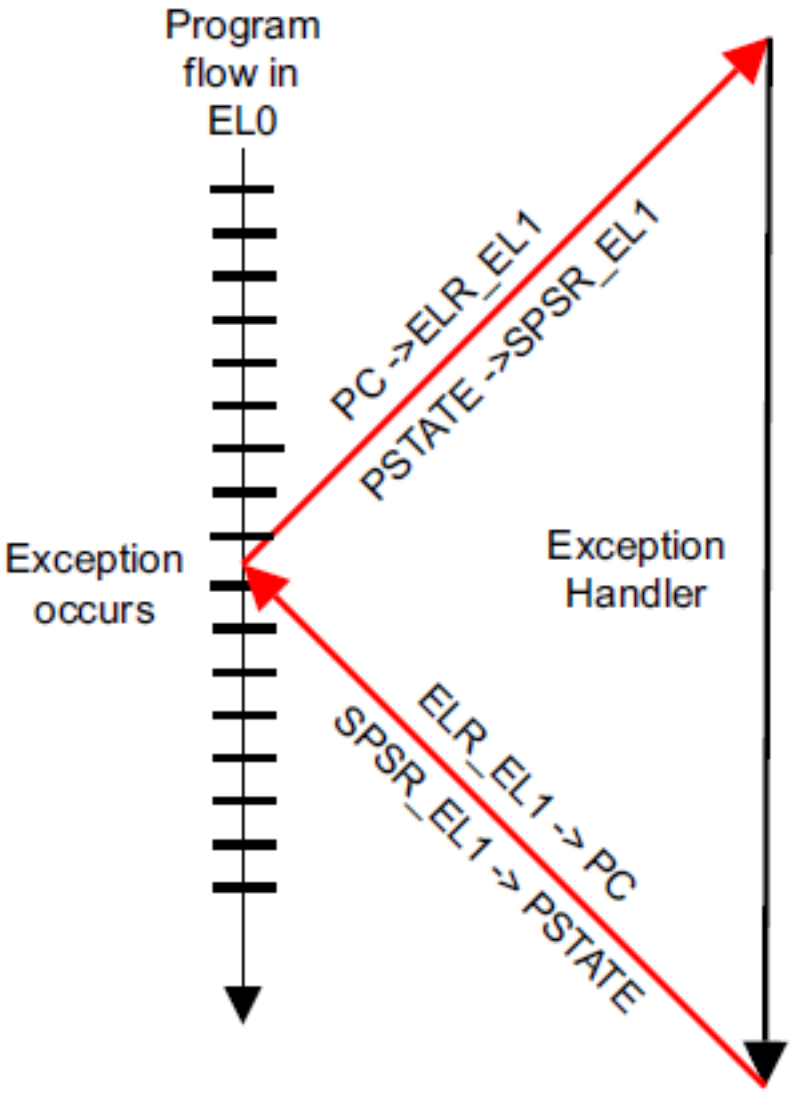

Figure 10-4 Exception handling

Untitled

需要注意的的是，寄存器后缀 _ELn 表示这些寄存器存在多个副本，这些不同的副本用于不同的异常级别，例如，SPSR_EL1 和 SPSR_EL2 是不同的物理寄存器，除此之外，在同步或 SError 发生的时候，ESR_ELn 寄存器会更新存储着异常原因的值

处理器何时从中断中返回，是通过执行 ERET 指令来完成，执行此指令将从 SPSR_ELn 恢复预异常 PSTATE，并通过从 ELR_ELn 恢复 PC 将程序执行返回到原始位置。

我们已经了解了 SPSR 寄存器如何为异常返回保存必要的状态信息。下面。我们将学习用于存储程序地址信息的链接寄存器，此架构为函数调用和异常返回提供了单独的链接寄存器。

我们在第六章 A64 指令集中看到，寄存器 X30 用于（于 RET 指令一起）从子程序中返回，每次我们使用链接指令（BL 或 BLR）执行分支时，X30 里面的值就更新为需要返回的指令地址。

ELR_ELn 寄存器存储着异常的返回地址，该寄存器的值在进入异常程序时会自动写入。并且此值会被写到 PC 里面，这是因为执行异常返回指令 ERET 所影响。

如果在异常返回时，SPSR 寄存器与系统寄存器不一致会报 error

对于一些特殊的异常，ELR_ELn 寄存器保存着异常返回地址，对于一些异常，ELR_ELn 保存着异常产生之后下一条待执行指令的地址。例如，当执行 SVC(系统调用）指令时，我们只是希望返回到应用程序中的以下指令。在其他情况下，我们可能希望重新执行产生异常的指令。

对于异步异常，ELR_ELn 寄存器保存着由于分支到中断处理而导致未来得及处理或者已经处理过的指令地址。例如，如果程序在一个异步中断之后必须要返回到一个指令，这时允许程序代码修改 ELR_En 寄存器。ARMv8-A 模型比 ARMv7-A 中使用的模型要简单得多，因为向后兼容的原因，当从某些类型的异常返回时，必须从 Link 寄存器值中减去 4 或 8。

除了 SPSR 和 ELR 寄存器外，每个异常级别都有自己的专用堆栈指针寄存器。这些被命名为 SP_EL0、SP_EL1、SP_EL2 和 SP_EL3。这些寄存器用于指向专用堆栈，例如，该堆栈可用于存储被异常处理程序破坏的寄存器，以便在返回原始代码之前将它们恢复为原始值。

处理程序代码可以从使用 SP_ELn 切换到 SP_EL0。例如，SP_EL1 可能指向一块内存，其中包含一个内核可以保证始终有效的小堆栈。SP_EL0 可能指向更大的内核任务堆栈，但不能保证不会溢出。这种切换是通过写入 [SPSel] 位来控制的，如下

```
MSR SPSel, #0 // switch to SP_EL0
MSR SPSel, #1 // switch to SP_ELn
```

10.2 10.2 同步和异步中断

在 AArch 中，异常可以是同步的，也可以是异步的。如果异常是由于执行或尝试执行指令而生成，并且返回的地址提供了指令的详细信息，则将其看作为同步异常。一个异步异常不会通过执行指令产生，并且返回的地址中，也许不会提供引起此异常的细节信息。

异步异常的来源是 IRQ（正常优先级中断）、FIQ（快速中断）或 SError（系统错误）。 系统错误有许多可能的原因，最常见的是异步数据中止（例如，将错误数据从高速缓存行写回外部存储器触发的中止）。

同步异常有多种来源：

- 指令从 MMU 中止。例如，通过从标记为从不执行的内存位置读取指令。
- 从 MMU 中止数据。例如，权限失败或对齐检查。
- SP 和 PC 对齐检查。

- 同步外部中止。例如，读取翻译表时发生的 abort。
- 未分配指令。
- 调试异常。

10.2.1 10.2.1 同步中断

同步中断可能通过多种可能的原因产生

- 产生于 MMU. 例如权限失效或者访问标志错误的内存区域
- SP 和 PC 对齐检查
- 未定义的指令
- 服务呼叫（SVC，SMC，HVC）

  此类异常可能是操作系统正常操作的一部分。例如，在 Linux 中，当一个任务希望请求分配一个新的内存页面时，这是通过 MMU 中止机制来处理的。

  在 ARMv7-A 架构中，prefetch abort、Data Abort 和 undef 异常是独立的项目。在 AArch64 中，所有这些事件都会生成同步中止。然后，异常处理程序可以读取状态和 FAR 寄存器以获得区分它们的必要信息（稍后将更详细地描述。）

10.2.2 10.2.2 处理同步异常

寄存器会提供处理同步异常的信息。异常状态寄存器（ESR_ELn）提供异常的原因信息，故障地址寄存器（FAR_
↪ELn）保存所有同步指令和数据中止以及对齐故障的故障虚拟地址。

异常链接寄存器（ELR_ELn）保存导致中止数据访问的指令的地址（对于数据中止）。 这通常在内存故障后更新，但在其他情况下被配置，例如，分支到了一个未对齐的地址。

如果异常从 AArch32 的异常级别转移到 AArch64 的异常级别，并且该异常写入了与目标异常级别相关联的故障地址寄存器（FAR_ELn），则 FAR_ELn 的前 32 位全部设置为零。

对于实现 EL2（Hypervisor）或 EL3（Secure Kernel）的系统，同步异常通常在当前或更高的异常级别上进行。异步异常可以（如果需要）路由到更高的异常级别，以由 Hypervisor 或安全内核处理。`SCR_EL3`

寄存器指定哪些异常将被路由到 EL3，类似地，HCR_EL2 指定将哪些异常路由到 EL2。有单独的位允许单独控制 IRQ、FIQ 和 SError 的路由。（**译注**：比如在 EL2 上运行 Hypervisor 时，来自设备的中断需要路由到 EL2，再由 Hypervisor 转发到 EL1 中的虚拟机中，这个时候就要设置 HCR_EL2 寄存器中 IRQ 相关的位）。

10.2.3 10.2.3 系统响应

某些指令或系统功能只能在特定的异常级别下执行。如果运行在较低异常级别的代码需要执行特权操作，例如，当应用程序代码向内核请求功能时。一种方法是使用 SVC 指令。这允许应用程序生成异常。可以传入参数寄存器，或在系统调用中编码。

10.2.4 10.2.4 系统响应到 EL2/EL3

前面我们看到了如何使用 SVC 从 EL0 的用户应用程序调用 EL1 的内核。HVC 和 SMC 系统调用指令以与 EL2 和 EL3 类似的方式移动处理器。当处理器在 EL0（应用程序）执行时，它不能直接调用管理程序（EL2）或安全监视器（EL3）。这只能从 EL1 及以上进行。因此，应用程序必须使用 SVC 调用内核并允许内核代表它们调用更高的异常级别。

从 OS 内核 (EL1)，软件可以使用 HVC 指令调用管理程序 (EL2)，或者使用 SMC 指令调用安全监视器 (EL3)。如果处理器使用 EL3 实现，则提供了让 EL2 捕获来自 EL1 的 SMC 指令的能力。如果没有 EL3，则 SMC 未分配并在当前 Exception 级别触发。同样，从管理程序代码 (EL2) 中，程序可以使用 SMC 指令调用安全监视器 (EL3)。如果您在 EL2 或 EL3 中进行 SVC 调用，它仍然会导致同一异常级别的同步异常，并且该异常级别的处理程序可以决定如何响应。

10.2.5 10.2.5 未分配的指令

未分配的的指令会导致 AArch64 产生同步中断，当处理器执行下面的操作之一时会生成此异常类型：

- 未分配的指令操作码
- 需要比当前异常级别更高级别特权的指令。
- 已禁用的指令。
- PSTATE.IL 域被设置时的任何指令。

10.2.6 10.2.6 异常状态寄存器

异常状态寄存器（Exception Syndrome Register, `ESR_ELn`）包含允许异常处理程序确定异常原因的信息。它只对针对同步异常和 `SError` 做更新，不为 `IRQ` 或 `FIQ` 更新，因为这些中断处理程序通常从通用中断控制器 (`GIC`) 的寄存器中获取状态信息。寄存器的位编码为:

- `ESR_ELn` 的 Bits[31:26] 表示异常类，它允许处理程序区分各种可能的异常原因 (如未分配的指令，源自 MCR/MRC 的到 CP15 的异常，FP 操作的异常，执行了 `SVC`，`HVC` 或 `SMC`，数据中止和对齐异常)。
- 位 [25] 表示陷入的指令长度 (0 表示 16 位指令，1 表示 32 位指令)，并且也为某些异常类设置。
- 位 [24:0] 形成指令特定症状 (`ISS`) 字段，该字段包含特定于该异常类型的信息。例如，当执行系统调用指令 (`SVC`、`HVC` 或 `SMC`) 时，该字段包含与操作码相关的**立即值**，例如对于 `SVC 0x123456` 的 0x123456。

10.3 10.3 异常导致的执行状态和异常级别的变化

当发生异常时，处理器可能会更改执行状态（从 AArch64 到 AArch32）或保持相同的执行状态。例如，外部源可能会在执行以 AArch32 模式运行的应用程序时生成 IRQ（中断）异常，然后在以 AArch64 模式运行的 OS 内核中执行 IRQ 处理程序。

SPSR 寄存器包括了执行状态和要返回的异常级别。当发生异常时，这由处理器自动设置。但是，每个异常级别中异常的执行状态控制如下：

- 最高异常级别（不一定是 EL3）的复位执行状态通常由硬件配置输入确定。但这不是固定的，因为我们有 RMR_ELn 寄存器来更改运行时最高异常级别的执行状态（寄存器宽度）（导致软复位）。请记住，EL3 与安全监视器代码相关联。监视器是一小段受信任的代码，始终以特定状态运行
- 对于 EL2 和 EL1，执行状态由 SCR_EL3.RW 和 HCR_EL2.RW 位控制。SCR_EL3.RW 位在 EL3（安全监视器）中编程并设置下一个较低级别（EL2）的状态。HCR_EL2.RW 位可在 EL2 或 EL3 中编程，并设置 EL1/0 的状态。
- 你永远不会在 EL0 中处理异常，（记住 EL0 是最低优先级，用于应用程序代码）。

考虑一个在 EL0 中运行的应用程序，它被一个 IRQ 中断，如图 10-5 所示。内核 IRQ 处理程序在 EL1 上运行。处理器在收到 IRQ 异常时确定要设置的执行状态。它通过查看控制寄存器的 RW 位来处理异常级别之上的异常级别。因此，在示例中，异常发生在 EL1 中，它是 HCR_EL2.RW 控制处理程序的执行状态。

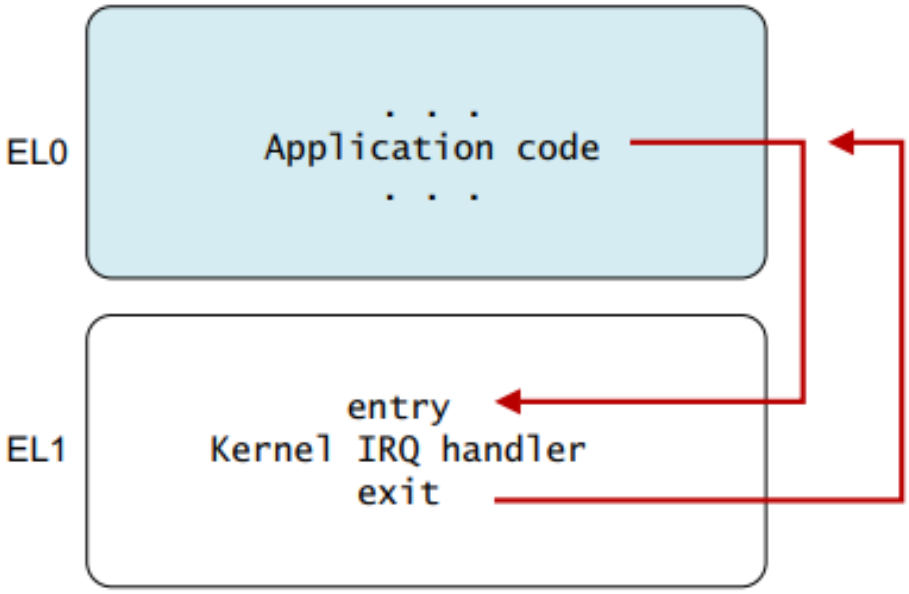

Figure 10-5 Exception to EL1

Untitled

现在我们必须考虑异常处于什么异常级别。同样，当接受异常时，异常级别可能保持不变，也可能变得更高。正如我们已经看到的，在 EL0 中从不接受异常。

同步异常通常在当前或更高的异常级别中被接受。而异步异常可以路由到一个更高的异常级别。对于安全代码，SCR_EL3 指定将哪些异常路由到 EL3。对于 hypervisor 代码，HCR_EL2 指定要路由到 EL2 的异常。

在这两种情况下，都有单独的位来控制 IRQ、FIQ 和 SError 的路由。处理器仅将异常引入到被它路由到的异常级别。异常级别永远不会因异常而下降。中断总是在中断发生的异常级别被屏蔽。

从 AArch32 到 AArch64 的异常处理时，有一些特殊的注意事项。AArch64 处理程序代码可能需要访问 AArch32 寄存器，因此架构定义了映射关系以允许访问 AArch32 寄存器。

AArch32 寄存器 R0 到 R12 作为 X0 到 X12 访问。AArch32 模式下的 SP 和 LR 的分组版本通过 X13 到 X23 访问，而分组的 R8 到 R12 FIQ 寄存器作为 X24 到 X29 访问。这些寄存器的位 [63:32] 在 AArch32 状态下不可用，包含 0 或最后写入 AArch64 的值。没有关于它的价值的架构保证。因此，通常将寄存器访问为 W 寄存器。

10.4 10.4 AArch64 异常向量表

当异常发生时，处理器必须执行与异常对应的处理程序代码。 存储处理程序的内存位置称为异常向量。 在 ARM 体系结构中，异常向量存储在一个表中，称为异常向量表。

每个异常级别都有自己的向量表，即 EL3、EL2 和 EL1 各有一个。 该表包含要执行的指令，而不是一组地址。 个别异常的向量位于表开头的固定偏移量处。 每个表基的虚拟地址由基于向量的地址寄存器 VBAR_EL3、VBAR_EL2和 VBAR_EL1 设置。

向量表中的每个条目有 16 条指令长。 与每个条目为 4 个字节的 ARMv7 相比，这本身就是一个重大变化。ARMv7 向量表的这种间距意味着每个条目几乎总是某种形式的分支，指向内存中其他地方的实际异常处理程序。在 AArch64 中，向量的间距更宽，因此顶层处理程序可以直接写入向量表中。

表 10-2 展示了其中一个向量表。基址由 VBAR_ELn 给出，然后每一项与这个基址有一个定义好了的偏移量。每个表有 16 项，每项 128 字节 (32 条指令) 大小。该表实际上由 4 组 4 项组成。使用哪一项取决于几个因素:

- 异常类型（SError、FIQ、IRQ 或同步）
- 如果在相同的异常级别处理异常，则要使用的堆栈指针（SP0 或 SPx）
- 如果异常是在较低的异常级别上被接受的，则降低一个异常的执行状态 (AArch64 或 AArch32)

Table 10-2 Vector table offsets from vector table base address

Address	Exception type	Description
VBAR_ELn + 0x000	Synchronous	Current EL with SP0
+ 0x080	IRQ/vIRQ	
+ 0x100	FIQ/vFIQ	
+ 0x180	SError/vSError	
+ 0x200	Synchronous	Current EL with SPx
+ 0x280	IRQ/vIRQ	
+ 0x300	FIQ/vFIQ	
+ 0x380	SError/vSError	
+ 0x400	Synchronous	Lower EL using AArch64
+ 0x480	IRQ/vIRQ	
+ 0x500	FIQ/vFIQ	
+ 0x580	SError/vSError	

Untitled

Table 10-2 Vector table offsets from vector table base address (continued)

Address	Exception type	Description
+ 0x600	Synchronous	Lower EL using AArch32
+ 0x680	IRQ/vIRQ	
+ 0x700	FIQ/vFIQ	
+ 0x780	SError/vSError	

Untitled

考虑一个例子可能会使这更容易理解。如果内核代码在 EL1 处执行并且产生 IRQ 中断信号，发生了 IRQ 异常。此特定中断与管理程序或安全环境无关，它也在内核中处理，在 SP_EL1 处设置了 SPSel 位，因此使用的是 SP_EL1。因此从地址 VBAR_EL1 + 0x280 开始执行。

> 在 ARMv8-A 架构中没有 `LDR PC, [PC, #offset]`在这样的指令情况下，必须使用更多的指令来从多个寄存器中读取目标。向量间距（128 字节）的选择是为了避免未被使用的向量对典型大小的指令缓存行造成缓存污染。复位地址是一个完全独立的地址，是处理器的实现来定义的，并且通常由核心内的硬连接配置设置。这个地址在 `RVBAR_EL1/2/3`寄存器中可见。
>
> 对于每个异常，无论是来自当前异常级别还是来自较低异常级别，都有一个单独的异常向量，这为操作系统或管理程序提供了确定较低异常级别的 AArch64 和 AArch32 状态的灵活性。 SP_ELn 用于从较低级别生成的异常。 但是，软件可以切换到在处理程序中使用 SP_EL0。 使用此机制时，它有助于从线程中的线程访问值处理程序。

10.5 10.5 中断处理

ARM 通常使用中断来表示中断信号。在 ARM A-profile 和 R-profile 处理器上，这意味着外部 IRQ 或 FIQ 中断信号。该架构没有指定如何使用这些信号。FIQ 通常保留用于安全中断源。在早些时候架构版本中，FIQ 和 IRQ 用于表示高优先级和标准中优先级的中断，但在 ARMv8-A 中并非如此。

当处理器接受一个异常到 AArch64 执行状态时，所有的 PSTATE 中断掩码都会自动设置。这意味着后续的异常将被禁用。如果软件要支持异常嵌套，例如，允许高优先级的中断去打断对低优先级源的处理，那么软件需要显式地重新启用中断。

对于下面的指令:

```
MSR DAIFClr, #imm
```

这个立即数值实际上是一个 4 位字段，因为也有如下掩码:

- PSTATE.A (for SError)
- PSTATE.D (for Debug)

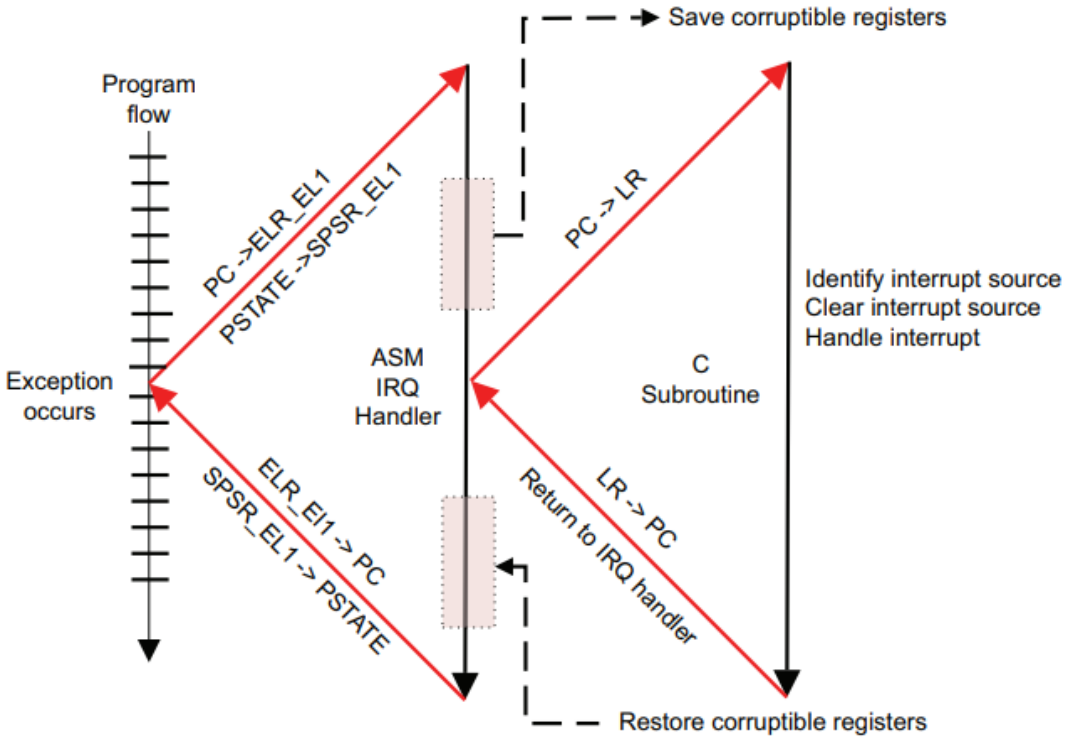

Figure 10-6 Interrupt handler in C code

Untitled

```
IRQ_Handler
// 压栈寄存器
STP X0, X1, [SP, #-16]!// SP = SP -16
...
STP X2, X3, [SP, #-16]!// SP = SP - 16
// 不像 ARMv7 没有 STM 指令，这里需要多条 STP 指令
BL read_irq_source// 一个函数计算出为什么收到一个中断
// 清中断
BL C_irq_handler// C 语言的中断处理函数
// 从栈中恢复寄存器
LDP X2, X3, [SP], #16// S = SP + 16, X2,X3 出栈
LDP X0, X1, [SP], #16// S = SP + 16, X0,X1 出栈
…
ERET// 异常返回
```

但是，从性能的角度来看，下面的处理顺序更优

```
IRQ_Handler
SUB SP, SP, #<frame_size>// SP = SP - <frame_size>
STP X0, X1, [SP]// 在底层帧中加载 X0 和 X1
```

(continues on next page)

(continued from previous page)

```
STP X2, X3, [SP]// 在底层帧 +16byte 处加载 X2 和 X3
...// 更多寄存器的保存
...
// 中断处理
BL read_irq_source// 一个函数计算出为什么收到一个中断
// 清中断
BL C_irq_handler// C 语言的中断处理函数
// 从栈中恢复寄存器
LDP X0, X1, [SP]// 在底层帧中加载 X0 和 X1
LDP X2, X3, [SP]// 在底层帧 +16byte 处加载 X2 和 X3
...// 更多的寄存器加载
ADD SP, SP, #<frame_size>// 将 SP 恢复为原始值
…
ERET// 异常返回
```

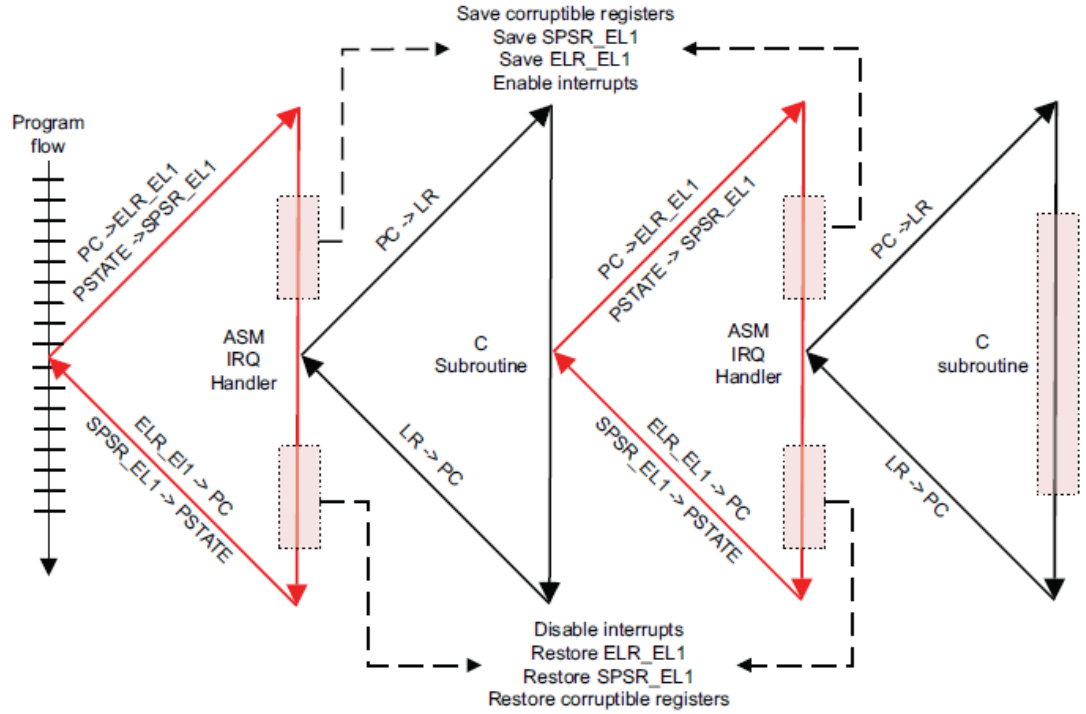

Figure 10-7 Handling nested interrupts

Untitled

嵌套处理程序需要一些额外的代码。它必须在堆栈上保存 SPSR_EL1 和 ELR_EL1 的内容。在确定（并清除）中断源之后，我们还必须重新启用 IRQ。但是（与 ARMv7-A 不同），由于子程序调用的链接寄存器与异常的链接寄存器不同，因此我们不需要对 LR 或者 mode 做特殊的处理。

10.6 10.6 通用中断处理

ARM 提供了用于 ARMv8-A 架构的标准中断控制器。该中断控制器的可编程接口在 GIC 架构中定义。GIC 架构规范有多个版本。本文档主要介绍版本 (GICv2)。ARMv8-A 处理器通常连接到 GIC，例如 GIC-400 或 GIC-500。通用中断控制器 (GIC) 支持在多核系统中的内核之间路由软件（SPI）生成、私有 (PPI) 和共享的外围中断 (SPI)。

GIC 架构提供的寄存器可用于管理中断源和中断行为，以及在多核系统中将中断路由到各个内核。以使软件能够屏蔽、启用和禁用来自单个源的中断，对（在硬件中）单个源进行优先级排序并生成软件中断。GIC 接受在系统级别断言的中断，并可以向它所连接的每个内核发送信号，从而可能导致发生 IRQ 或 FIQ 异常。

从软件的角度来看，GIC 主要有两个主要的功能块

Distributor：系统中的所有中断源都连接到 distributor。Distributor 提供寄存器来控制各个中断的属性，例如优先级、状态、安全性、路由信息和启用状态。distributor 通过附加的 CPU 接口确定将哪个中断转发到内核。

CPU Interface：cpu 通过它接收中断。CPU 接口提供寄存器来屏蔽、识别和控制转发到该内核的中断状态。系统中的每个内核都有一个单独的 CPU 接口。

中断在软件中通过一个编号来标识，称为中断 ID。一个中断 ID 对应一个中断源。软件可以使用中断 ID 来识别中断源并调用相应的处理程序来服务中断。提供给软件的确切中断 ID 由系统设计决定，中断可以有多种不同的类型：

** 软件生成中断 (SGI)：** 这是由软件通过写入 distributor 寄存器生成的中断，软件生成的中断寄存器（GICD_SGIR）。通常用于内核间通信。SGI 可以完全只对系统中选定的一组 core 起效。为 SGI 保留中断 ID 是 0-15。**专用外设中断 (PPI)**：这是 distributor 可以路由到指定的 cpu 上的中断。为 PPI 保留的中断 ID 是 16-31。这些表示中断 core 私有的中断源，并且独立于另一个 core 上的相同源，例如，每个 core 上的计时器产生的 PPI。** 共享外设中断 (SPI)：** 这是 GIC 可以路由到多个 core 的外设中断。中断号 32-1020 用于 SPI。SPI 用于发出中断信号从整个系统可访问的各种外围设备。

局部特定外设中断 (LPI)：这些是路由到特定 core 的基于消息的中断。LPI 是 GICv2 或 GICv1 不支持。

中断可以是边沿触发的（当 GIC 检测到相关输入的上升沿时被认为是有效的，并且在清除之前保持有效）或电平敏感的（认为仅当 GIC 的相关输入为高电平时才有效）。

中断可以处于多种不同的状态：

• 活动——这意味着该中断已被内核确认并且正在目前正在服务中。• 活动和未决——这描述了内核正在服务中断的情况并且 GIC 也有来自同一源的未决中断。

- Inactive——这意味着当前中断没有断言。
- Pending——这意味着中断源已被断言，但正在等待由核心处理。待处理的中断是被转发到 CPU 接口的候选然后再到 CPU
- active——这意味着该中断已被 cpu 接收并且正在目前正在处理中。
- active and pending——这描述了内核正在处理该中断并且 GIC 判断该中断还一直送过来。

中断传递的优先级和目标 core 都在 distributor 中配置。外围设备 distributor 发送的中断都处于待处理状态（如果它已经处于活动状态，则处于活动和待处理状态）。distributor 确定最高优先级的待处理的中断，可以传递到 core 并将其转发到 CPU 接口，在 CPU 接口处，中断依次发送给 core，此时 core 接受 FIQ 或 IRQ 异常。core 执行异常处理程序作为响应。处理程序从 CPU 接口寄存器中查询中断 ID 并开始执行中断服务程序。完成后，处理程序必须写入 CPU 接口寄存器以报告处理结束。

对于给定的中断，典型的顺序是：

- 非活动 -> 待处理：当外设断言中断时。
- 待定 -> 活动：当处理程序确认中断时
- 活动 -> 非活动：当处理完中断时。

Distributor 提供了报告不同中断 ID 的当前状态的寄存器。在多核/多处理器系统中，单个 GIC 可以由多个内核共享（在 GICv2 中最多 8 个）。GIC 提供寄存器来控制 SPI 所对应的内核。这个机制使操作系统能够跨内核共享和分发中断，并且协调工作。有关 GIC 行为的更多详细信息，请参阅 `ARM Generic Interrupt Controller Architecture specification`。

10.6.1 10.6.1 配置

GIC 的寄存器实现都是外部 memory_map 形式。所有的核都可以访问公共的 Distributor，但是 CPU 接口是 banked 的，即每个核使用相同的地址访问自己的私有 CPU 接口。一个内核不可能访问另一个内核的 CPU 接口。

Distributor 包含许多寄存器，您可以使用它们来配置各个中断的属性。这些可配置的属性有：

- 中断优先级（GICD_IPRIORITY），distributor 使用它来确定哪个中断接下来被转发到 CPU 接口。
- 中断配置（GICD_ICFGR）。这决定了中断是电平还是边缘敏感。不适用于 SGI。
- 一个中断目标 core（GICD_ITARGETSR）。这确定了中断可以路由到哪些 core。仅适用于 SPI。
- 中断启用或禁用状态（GICD_ISENABLERGICD_ICENABLER）。只有那些在 distributor 中启用的中断。当它们处于待处理状态时有资格被路由到 cpu 接口。
- 中断安全 (GICD_IGROUPR) 确定中断是否分配给安全或非安全。
- 中断状态。

Distributor 还提供了优先级屏蔽，通过它可以防止低于某个优先级的中断到达内核。distributor 在确定是否可以将挂起的中断转发到特定内核时使用它每个内核上的 CPU 接口有助于微调中断控制和处理核。

10.6.2 10.6.2 初始化

distributor 和 CPU interface 在复位时都被禁用。GIC 必须在复位后初始化，然后才能向内核提供中断。

在 Distributor 中，软件必须配置优先级、目标 core、安全性并启用各个中断。随后必须通过其控制寄存器 (GICD_CTLR) 启用 distributor。对于每个 CPU interface，软件必须对优先级掩码和优先级抢占进行设置。每个 CPU interface 本身必须通过控制寄存器 (GICD_CTLR) 启用。在 cpu 处理中断之前，软件通过设置使 cpu 准备好接受中断向量表中的有效中断向量，并清除 PSTATE 中的中断屏蔽位，并设置路由控制.. 可以通过禁用 Distributor 来禁用系统中的整个中断机制。也可以通过禁用其 CPU 接口来禁用对单个内核的中断传递。也可以在分配器中禁用（或启用）各个中断。对于一个中断到达核心，单个中断、分配器和 CPU 接口必须全部启用。中断也需要有足够的优先级，即高于内核的优先掩码。

10.6.3 10.6.3 中断处理

当 core 接受了中断，就跳转到顶层中断向量表并且开始执行。顶层的中断处理程序从 CPU interface 读取寄存器以获得中断 ID。

读取寄存器除了返回中断 ID 外，还会导致中断在 distributor 中被标记为活 active。一旦知道中断 ID（识别中断源），顶层处理程序就可以调度特定于设备的中断处理程序来服务中断。

当中断处理程序完成执行时，顶层处理程序将相同的中断 ID 写入 CPU 接口块中的中断结束 (EoI) 寄存器，指示中断处理结束。

除了移除活动状态，使最终的中断状态为 Inactive 或 pending（如果状态为 active and pending），这使 CPU 接口能够将更多挂起的中断转发到 core。这结束了单个中断的处理。

在同一个内核上可能有多个中断等待服务，但是 CPU 接口一次只能发出一个中断信号。顶层中断处理程序可以重复上述顺序，直到读取到特殊中断 ID 值 1023，表示有在这个核心上没有更多的中断挂起。这个特殊的中断 ID 被称为虚假的中断标识。虚假中断 ID 是一个保留值，不能分配给系统。当顶层处理程序读取了虚假中断 ID 后，它可以完成它的执行，并准备内核在中断之前恢复它正在执行的任务。通用中断控制器 (GIC) 通常管理来自多个中断源的输入。并将它们分发给 IRQ 或 FIQ 。

CHAPTER

ELEVEN

11. 缓存 CACHE

ARM 架构最初开发时，处理器的时钟速度和内存的访问速度大致相似。今天的处理器内核要复杂得多，并且时钟频率可以提高几个数量级。然而，外部总线和存储设备的频率并没有达到同样的程度。可以实现能够以与内核相同的速度运行的片上 SRAM 小块，但与容量可以增加数千倍的标准 DRAM 块相比，这种 RAM 非常昂贵。在许多基于 ARM 处理器的系统中，访问外部存储器需要数十甚至数百个内核周期。

高速缓存是位于核心内存和主内存之间的小而快的内存块。它在主存储器中保存项目的副本。对高速缓冲存储器的访问比对主存储器的访问要快得多。每当核心读取或写入特定地址时，它首先在缓存中查找它。如果它在缓存中找到地址，它会使用缓存中的数据，而不是执行对主存储器的访问。通过减少缓慢的外部存储器访问时间的影响，这显着提高了系统的潜在性能。通过避免驱动外部信号，它还降低了系统的功耗。

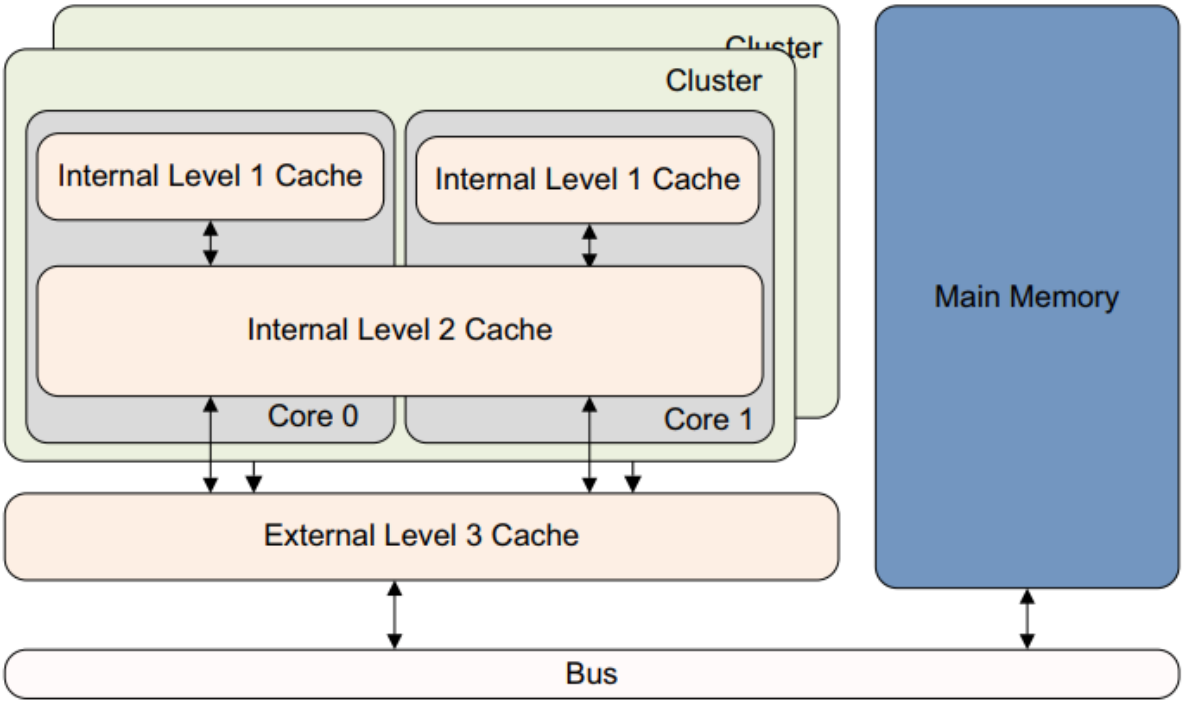

image-20220315073226390

实现 ARMv8-A 架构的处理器通常使用两级或多级缓存来实现。这通常意味着处理器的每个内核都有小型 L1 指令和数据缓存。Cortex-A53 和 Cortex-A57 处理器通常使用两级或多级缓存实现，即小型 L1 指令和数据缓存和更大的统一 L2 缓存，在集群中的多个内核之间共享。此外，可以有一个外部 L3 缓存作为外部硬件块，

在集群之间共享。

将数据提供给缓存的初始访问并不比正常速度快。对缓存值的任何后续访问都会更快，而性能提升正是由此而来。核心硬件检查缓存中的所有指令获取和数据读取或写入，但您必须将内存的某些部分（例如包含外围设备的部分）标记为不可缓存。因为缓存只保存主内存的一个子集，所以您需要一种方法来快速确定您要查找的地址是否在缓存中。

有时，缓存中的数据和指令与外部存储器中的数据可能不一样；这是因为处理器可以更新尚未写回主存的缓存内容。或者，代理可能会在核心获取自己的副本后更新主内存。这是第 14 章中描述的一致性问题。当您有多个内核或内存代理（例如外部 DMA 控制器）时，这可能是一个特殊问题。

11.1 11.1 缓存术语

在冯诺依曼架构中，单个缓存用于指令和数据（统一缓存）。修改后的哈佛架构具有独立的指令和数据总线，因此有两个高速缓存，一个指令高速缓存（I-cache）和一个数据高速缓存（D-cache）。在 ARMv8 处理器中，有不同的指令和数据 L1 缓存，由统一的 L2 缓存支持。

缓存需要保存地址、一些数据和一些状态信息。

以下是一些使用的术语的简要总结和说明缓存基本结构的图表：

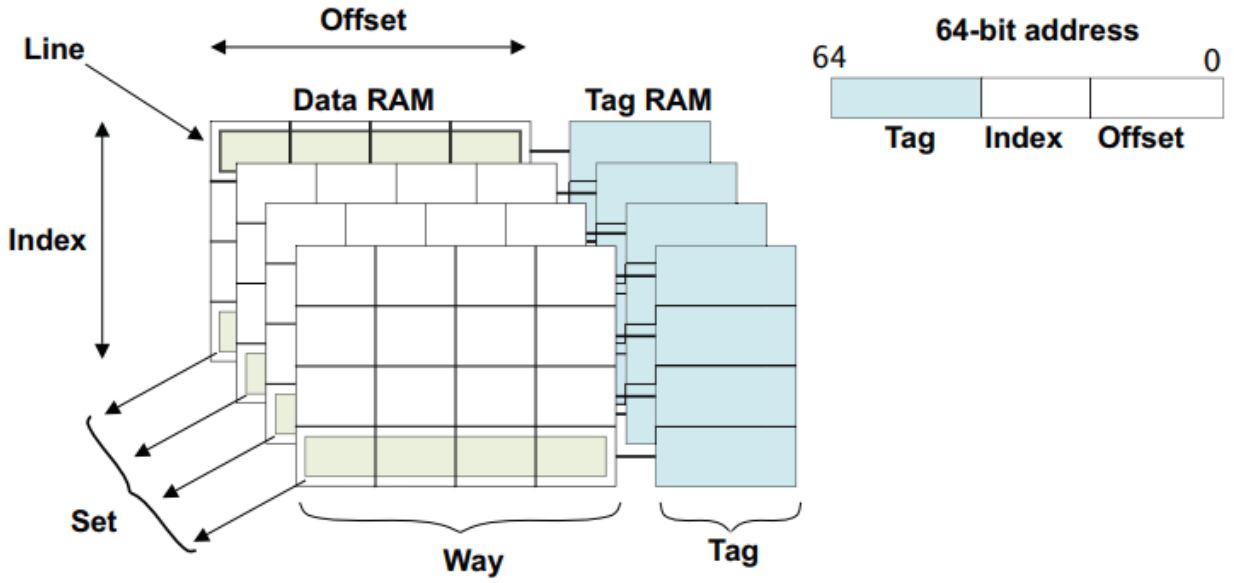

image-20220315073355337

- 标签是存储在高速缓存中的内存地址的一部分，用于标识与数据行相关的主内存地址。
- 64 位地址的高位告诉缓存信息在主存中的来源，称为标记。总缓存大小是它可以保存的数据量的度量，尽管用于保存标记值的 RAM 不包括在计算中。但是，标签确实占用了缓存中的物理空间
- 为每个标签地址保存一个数据字是低效的，因此通常将多个位置组合在同一个标签下。这个逻辑块通常称为高速缓存行，是指高速缓存的最小可加载单元，即来自主存储器的连续字块。当缓存行包含缓存的数据或指令时，它被认为是有效的，如果不包含缓存行则称为无效。与每一行数据相关的是一个

或多个状态位。通常，您有一个有效位，将行标记为包含可以使用的数据。这意味着地址标签代表了一些真实的价值。在数据缓存中，您可能还具有一个或多个脏位，用于标记缓存行（或其一部分）是否保存与主内存内容不同（更新）的数据。

- 索引是内存地址的一部分，它确定可以在高速缓存的哪些行中找到该地址。地址或索引的中间位标识行。索引用作缓存 RAM 的地址，不需要存储作为标记的一部分。本章稍后将对此进行更详细的介绍。
- 路是高速缓存的细分，每个路的大小相同并以相同的方式索引。一组由共享特定索引的所有方式的高速缓存行组成
- 这意味着地址的底部几位（称为偏移量）不需要存储在标签中。您需要整行的地址，而不是行中每个字节的地址，因此五个或六个最低有效位始终为 0。

11.1.1 11.1.1 设置关联缓存和方式

ARM 内核的主缓存始终使用一组关联缓存来实现。这显着降低了直接映射缓存出现缓存抖动的可能性，提高了程序执行速度并提供了更具确定性的执行。它以增加硬件复杂性和略微增加功率为代价，因为每个周期都会比较多个标签。

使用这种缓存组织方式，缓存被分成许多大小相等的块，称为路。然后，内存位置可以映射到一条路而不是一条线。地址的索引字段继续用于选择特定行，但现在它以每种方式指向单独的行。通常，L1 数据缓存有两种或四种方式。Cortex-A57 具有 3 路 L1 指令高速缓存。L2 缓存通常有 16 种方式。

外部 L3 缓存实现，例如 ARM CCN-504 缓存一致性网络（请参阅第 14-18 页的计算子系统和移动应用程序），可以有更多的方式，即更高的关联性，因为它们的大小要大得多。具有相同索引值的缓存行被称为属于一个集合。要检查命中，您必须查看集合中的每个标签。

在图 11-3 中，显示了一个 2 路缓存。来自地址 0x00、0x40 或 0x80 的数据可能会在任一缓存方式的第 0 行中找到，但不能同时在两种缓存方式中找到。

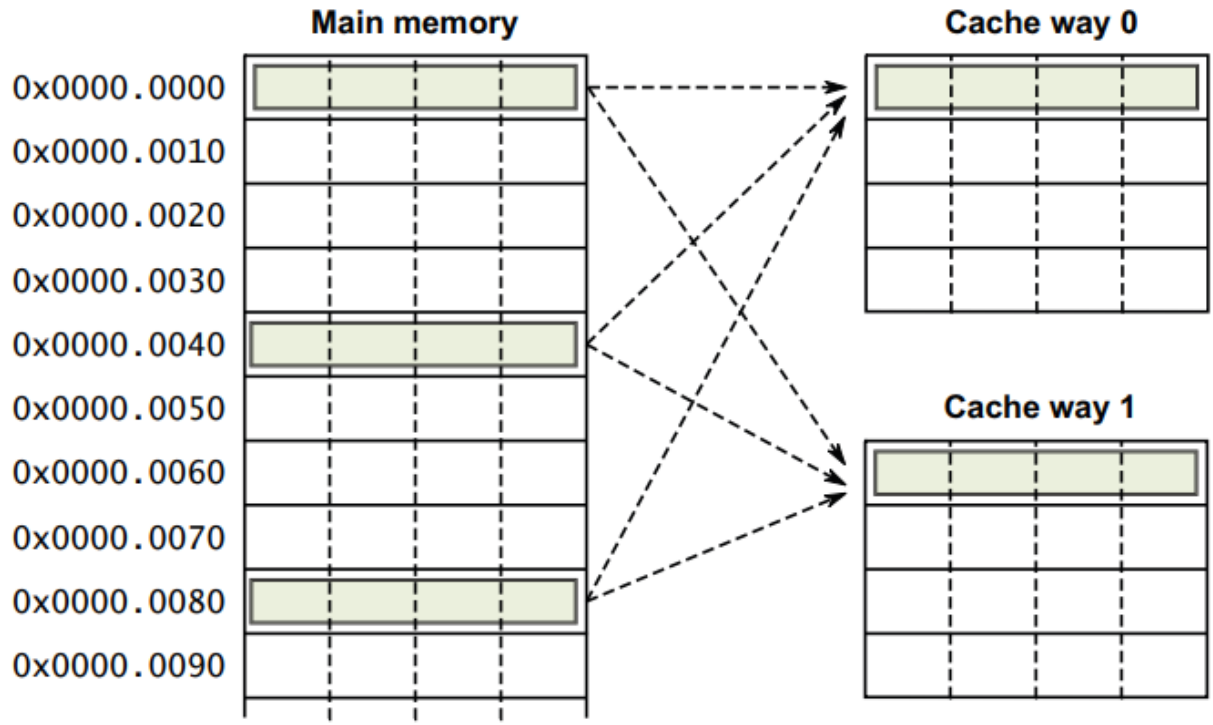

image-20220315073652616

增加缓存的关联性会降低抖动的可能性。理想的情况是完全关联的缓存，其中任何主内存位置都可以映射到缓存中的任何位置。但是，除了非常小的缓存（例如与 MMU TLB 相关的缓存）之外，构建这样的缓存是不切实际的。在实践中，8 路以上的性能改进是最小的，而 16 路关联性对于较大的 L2 缓存更有用。

11.1.2 11.1.2 缓存标签和物理地址

每条线都有一个与之关联的标签，该标签记录了与该线关联的外部存储器中的物理地址。高速缓存行的大小由实现定义。但是，由于互连，所有内核都应具有相同的高速缓存行大小。

访问的物理地址用于确定数据在缓存中的位置。最低有效位用于选择高速缓存行中的相关项。中间位用作索引以选择高速缓存集中的特定行。最高有效位标识地址的其余部分，并用于与该行的存储标签进行比较。在 ARMv8 中，数据缓存通常是物理索引、物理标记 (PIPT)，但也可以是非混叠虚拟索引、物理标记 (VIPT)。

缓存中的每一行包括：

- 来自相关物理地址的标签值。
- 有效位表示该行是否存在于缓存中，即标签是否有效。如果高速缓存跨多个内核是一致的，则有效位也可以是 MESI 状态的状态位。
- 脏数据位指示缓存行中的数据是否与外部存储器不一致。

ARM 缓存设置为关联的。这意味着对于任何给定地址，都有多个可能的缓存位置或方式。一组关联缓存显着降低了缓存抖动的可能性，从而提高了程序执行速度，但代价是增加了硬件复杂性和略微增加了功率。

一个简化的四路组相联 32KB L1 缓存（如 Cortex-A57 处理器的数据缓存），缓存线长度为 16 字（64 字节），如图所示（32KB 4-way set associative data cache ）：

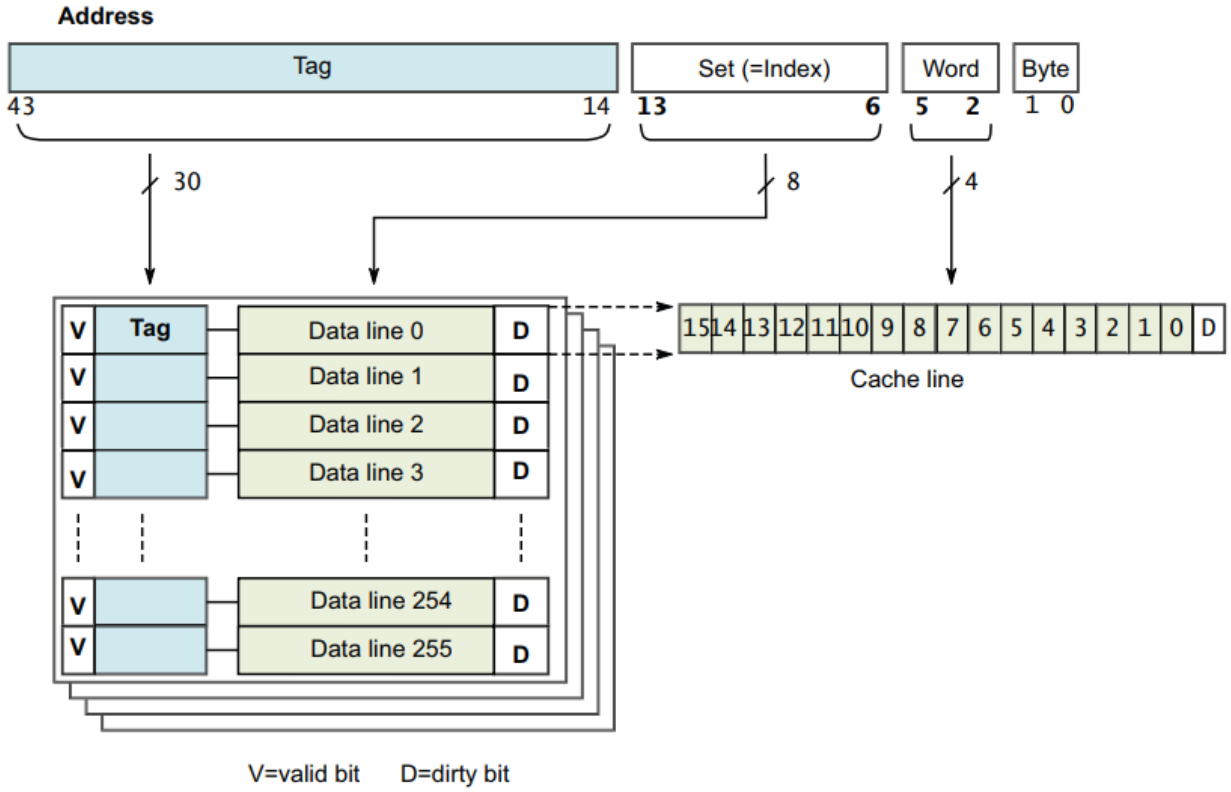

image-20220501091015149

11.1.3 11.1.3 Inclusive 和 exclusive 的 cache

考虑一个简单的内存读取，例如，单核处理器中的 LDR X0, [X1]。

- 如果 X1 指向内存中标记为可缓存的位置，则在 L1 数据缓存中进行缓存查找。
- 如果在 L1 缓存中找到该地址，则从 L1 缓存中读取数据并返回给内核。

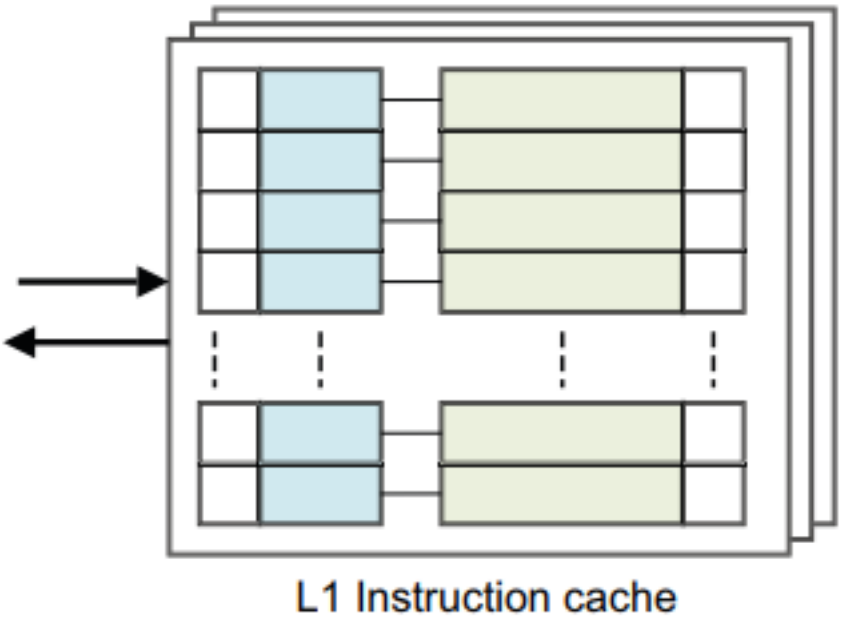

image-20220501091546127

- 如果在 L1 缓存中没有找到地址，但在 L2 缓存中，则将缓存行从 L2 缓存加载到 L1 缓存中，并将数据返回给核心。这可能会导致从 L1 中逐出一条线以腾出空间，但它可能仍存在于较大的 L2 缓存中。

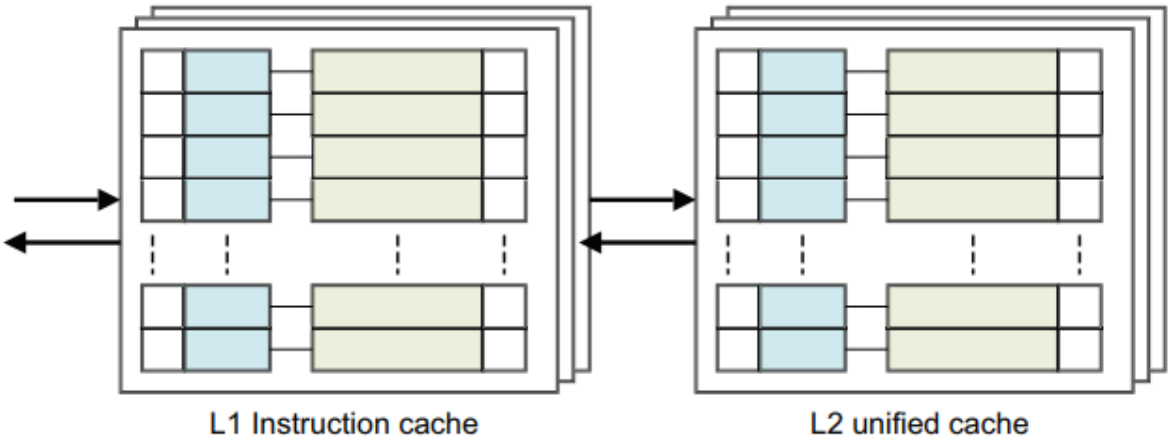

image-20220501091607789

- 如果地址不在 L1 或 L2 缓存中，则数据从外部存储器加载到 L1 和 L2 缓存中并提供给内核。这可能会导致线路被驱逐。

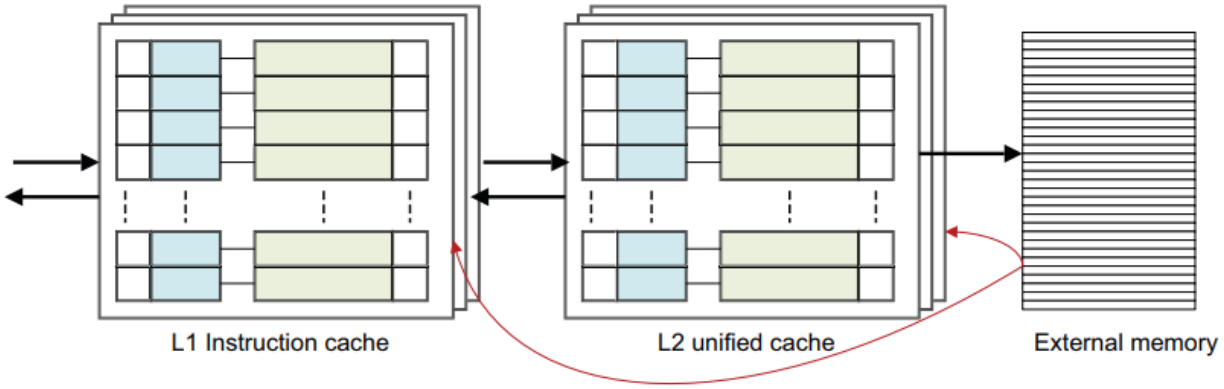

image-20220501091623776

这是一个比较简单的观点。对于多核和多集群系统，在从外部存储器执行加载之前，也可能会检查集群内或其他集群的内核的 L2 或 L1 缓存的缓存。此外，此时没有考虑 L3 或系统缓存。

这是一种包容性缓存模型，其中相同的数据可以同时存在于 L1 和 L2 缓存中。在独占缓存中，数据只能存在于一个缓存中，并且不能同时在 L1 和 L2 缓存中找到一个地址。

11.2 11.2 缓存控制器

高速缓存控制器是负责管理高速缓存内存的硬件块，其方式对程序来说在很大程度上是不可见的。它自动将代码或数据从主存写入缓存。它从内核接收读取和写入内存请求，并对高速缓存或外部存储器执行必要的操作。

当它收到来自核心的请求时，它必须检查是否要在缓存中找到所请求的地址。这称为缓存查找。它通过将请求的地址位的子集与与缓存中的行关联的标记值进行比较来做到这一点。如果存在匹配，称为命中，并且该行被标记为有效，则使用高速缓存进行读取或写入。

当核心从特定地址请求指令或数据，但与缓存标签不匹配或标签无效时，会导致缓存未命中，请求必须传递到内存层次结构的下一层，即 L2 缓存或外部存储器。它还可能导致缓存行填充。缓存行填充会导致将一块主内存的内容复制到缓存中。同时，请求的数据或指令被流式传输到内核。这个过程是透明地发生的，软件开发人员不能直接看到。在使用数据之前，核心不需要等待 linefill 完成。高速缓存控制器通常首先访问高速缓存行内的关键字。例如，如果您执行的加载指令在缓存中未命中并触发缓存行填充，则内核首先检索缓存行中包含所请求数据的那部分。这些关键数据被提供给核心流水线，而缓存硬件和外部总线接口则在后台读取缓存线的其余部分。

11.3 11.3 缓存策略

缓存策略使我们能够描述何时应将一行分配给数据缓存，以及当执行存储指令时会发生什么，该指令命中数据缓存。

缓存分配策略是：

- **Write allocation (WA)** 在写未命中时分配高速缓存行。这意味着在处理器上执行存储指令可能会导致发生突发读取。在执行写入之前，有一个 linefill 来获取缓存行的数据。高速缓存包含整行，这是其最小的可加载单元，即使您只写入该行中的单个字节。
- **Read allocation (RA)** 在读取未命中时分配高速缓存行。

缓存更新策略是：

- **Write-back (WB)** 写入仅更新缓存并将缓存行标记为脏。仅当线路被逐出或明确清除时，才会更新外部存储器。

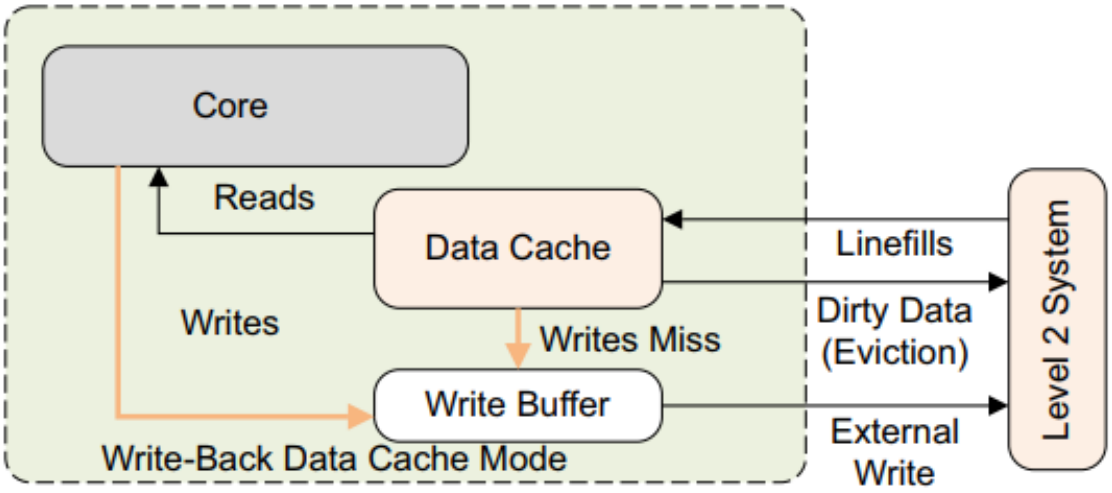

image-20220501092328170

- **Write-through (WT)** 写入更新缓存和外部存储器系统。这不会将高速缓存行标记为 dirty。

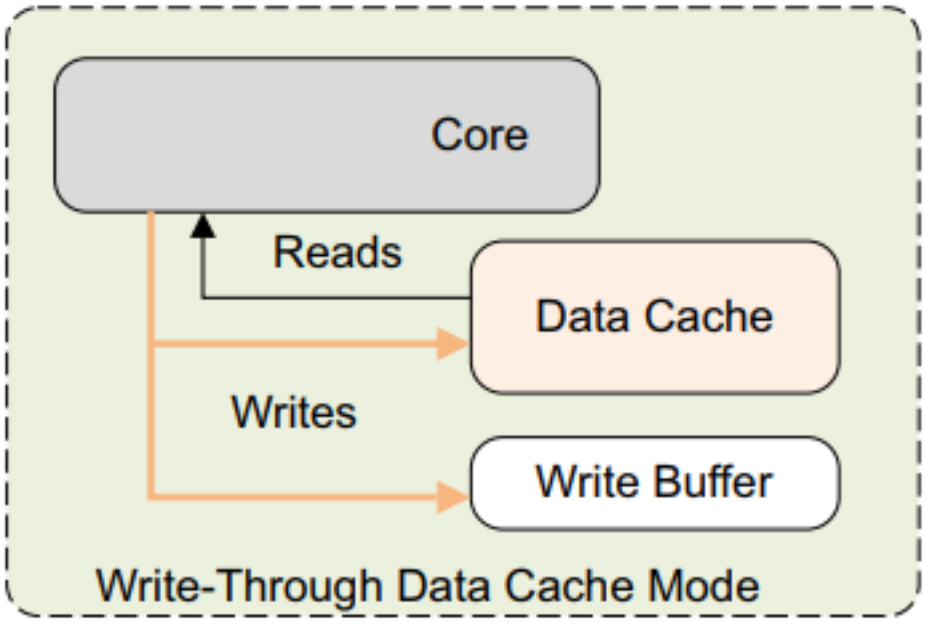

image-20220501092346467

在 WT 和 WB 缓存模式下，命中缓存的数据读取行为相同。

普通内存的可缓存属性分别指定为内部和外部属性。内部和外部之间的划分由实现定义并在第 13 章中详细介绍。通常，内部属性由集成缓存使用，外部属性在处理器内存总线上可供外部缓存使用。

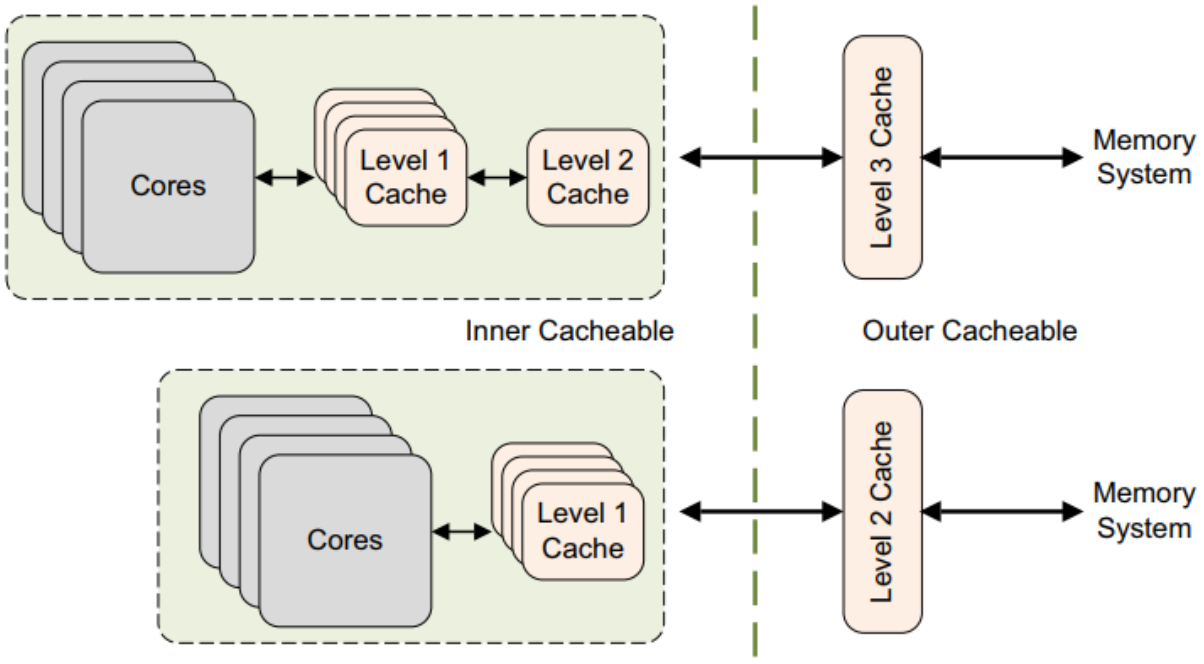

image-20220501092610027

处理器可以推测性地访问普通内存，这意味着它可以潜在地自动将数据加载到缓存中，而无需程序员明确请求特定地址。这在第 13 章内存排序中有更详细的介绍。但是，程序员也可以向内核指示将来使用哪些数据。ARMv8-A 提供预加载提示指令。缓存是否支持推测和预加载由实现定义。可以使用以下说明：

- AArch64：`PRFM PLDL1KEEP, [Xm, #imm]`；这表示从 “Xm + offset” 加载到 L1 缓存中作为临时预取的预取，这意味着数据可能会被多次使用。
- AArch32: `PLD Rm` // 将数据从 Rm 中的地址预加载到缓存中

  通常，预取内存的 A64 指令具有以下形式：

  - `PRFM <prfop>, addr`
  -
  - `<prfop> <type><target><policy> | #uimm5`
  - `<type>` PLD for prefetch for load PST for prefetch for store
  - `<target>` L1 for L1 cache, L2 for L2 cache, L3 for L3 cache
  - `<policy>` KEEP 用于保留或临时预取意味着通常在缓存中分配 STRM 用于流式或非临时预取意味着内存仅使用一次
  - `uimm5` 将提示编码表示为 5 位立即数。这些是可选的.

11.4 11.4 一致性和统一点

对于基于集合和基于方式的清理和无效，操作是在特定级别的缓存上执行的。对于使用虚拟地址的操作，架构定义了两点：

- Point of Coherency (PoC) 。对于特定地址，PoC 是所有观察者（例如，可以访问内存的内核、DSP 或 DMA 引擎）确保看到内存位置的相同副本的点。通常，这是主要的外部系统存储器

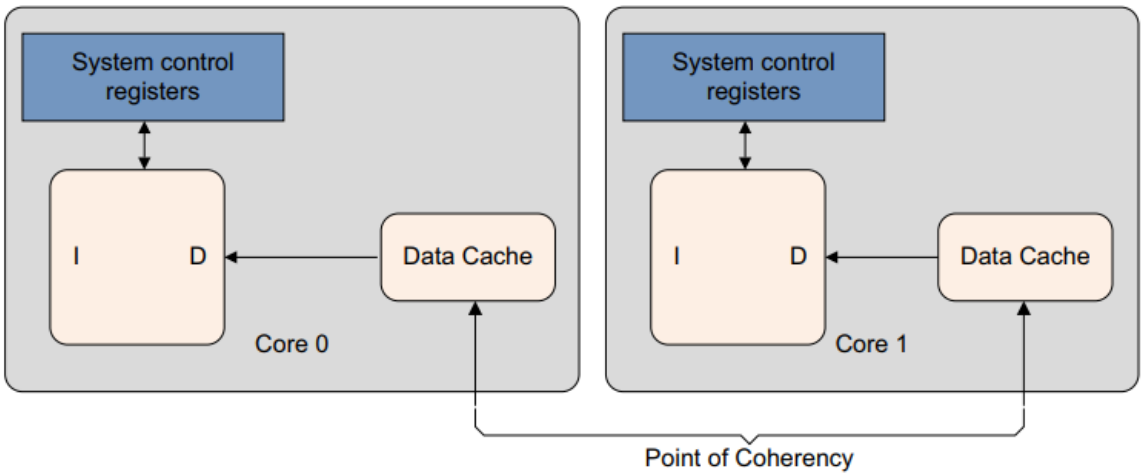

image-20220501104006294

- Point of Unification (PoU) 。内核的 PoU 是保证内核的指令和数据缓存以及转换表遍历可以看到内存位置的相同副本的点。例如，统一的 2 级缓存将成为系统中的统一点，该系统具有哈佛 1 级缓存和用于缓存转换表条目的 TLB。如果不存在外部缓存，则主存储器将是统一点。

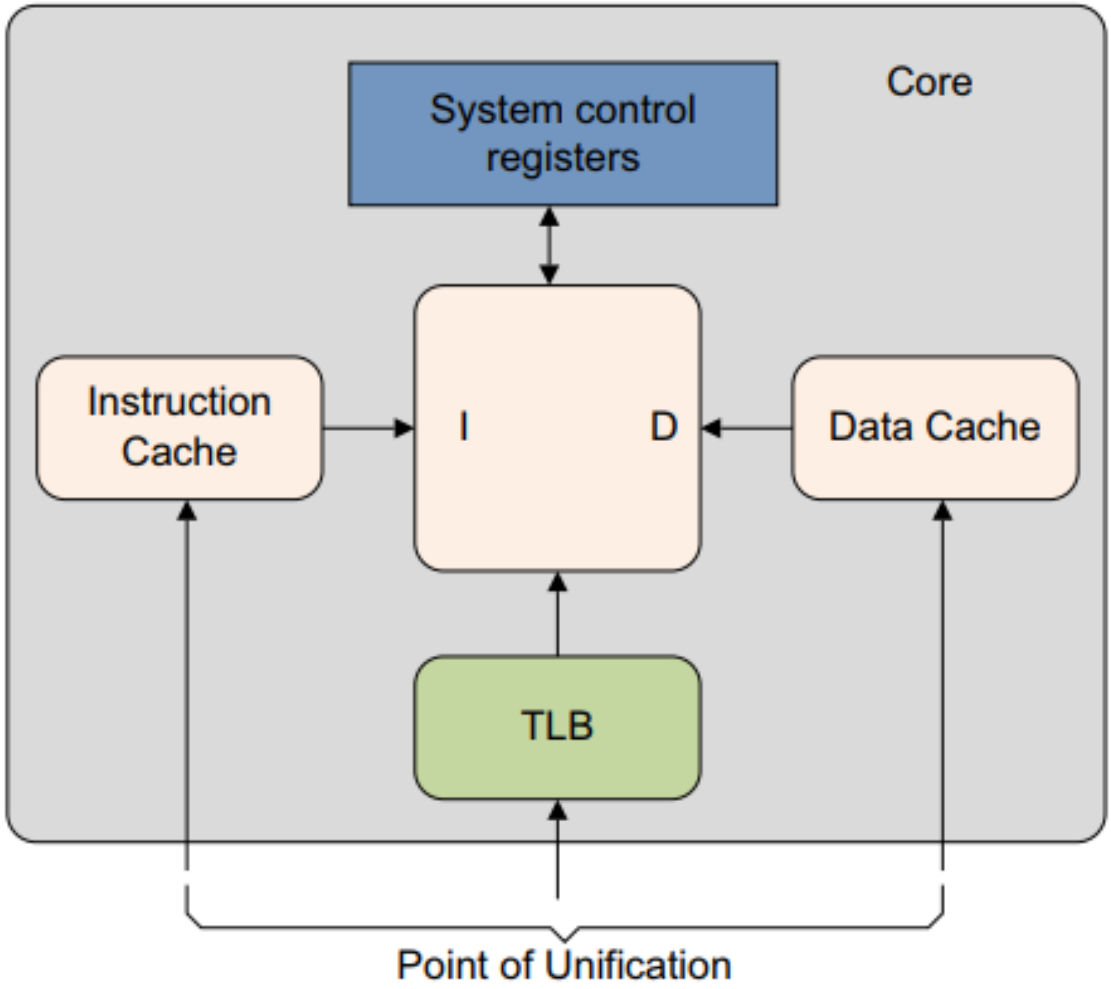

image-20220501104029490

对 PoU 的了解使代码能够自我修改，以确保从修改后的代码版本中正确获取未来的指令。他们可以通过使用两个阶段的过程来做到这一点：

- 按地址清理相关数据缓存条目。
- 按地址使指令缓存条目无效。

ARM 架构不需要硬件来确保指令缓存和内存之间的一致性，即使对于共享内存的位置也是如此。

11.5 11.5 缓存维护

软件有时需要清理缓存或使缓存失效。当外部存储器的内容已更改并且需要从缓存中删除陈旧数据时，可能需要这样做。在与 MMU 相关的活动（例如更改访问权限、缓存策略或虚拟到物理地址映射）之后，或者当 I 和 D 缓存必须为动态生成的代码（例如 JIT 编译器和动态库加载器）同步时，也可能需要它。

- 高速缓存或高速缓存行的无效意味着通过清除一个或多个高速缓存行的有效位来清除其中的数据。缓存在重置后必须始终无效，因为它的内容是未定义的。这也可以被视为一种在缓存外的内存域中对缓

存用户可见的更改的方式。

- 清理高速缓存或高速缓存行意味着将标记为脏的高速缓存行的内容写入下一级高速缓存或主存储器，并清除高速缓存行中的脏位。这使得高速缓存行的内容与高速缓存或内存系统的下一级保持一致。这仅适用于使用回写策略的数据缓存。这也是一种使缓存中的更改对外部内存域的用户可见的方法，但仅适用于数据缓存。
- 零。这会将缓存中的内存块归零，而无需首先从外部域读取其内容。这仅适用于数据缓存。

对于这些操作中的每一个，您可以选择操作应应用于哪些条目：

- all，表示整个缓存，不可用于数据或统一缓存
- 修改后的虚拟地址 (MVA)，VA 的另一个名称，是包含特定虚拟地址的高速缓存行
- Set 或 Way 是由其在缓存结构中的位置选择的特定缓存行

AArch64 缓存维护操作使用具有以下一般形式的指令执行：`< 缓存> < 操作>{, <Xt>}`

Cache	Operation	Description	AArch32 Equivalent
DC	CISW	Clean and invalidate by Set/Way	DCCISW
	CIVAC	Clean and Invalidate by Virtual Address to Point of Coherency	DCCIMVAC
	CSW	Clean by Set/Way	DCCSW
	CVAC	Clean by Virtual Address to Point of Coherency	DCCMVAC
	CVAU	Clean by Virtual Address to Point of Unification	DCCMVAU
	ISW	Invalidate by Set/Way	DCISW
	IVAC	Invalidate by Virtual Address, to Point of Coherency	DCIMVAC
DC	ZVA	Cache zero by Virtual Address	-
IC	IALLUIS	Invalidate all, to Point of Unification, Inner Sharable	ICIALLUIS
	IALLU	Invalidate all, to Point of Unification, Inner Shareable	ICIALLU
	IVAU	Invalidate by Virtual Address to Point of Unification	ICIMVAU

image-20220501101948798

那些接受地址参数的指令采用一个 64 位寄存器，该寄存器保存要维护的虚拟地址。此地址没有对齐限制。采用 Set/Way/Level 参数的指令采用 64 位寄存器，其低 32 位遵循 ARMv7 架构中描述的格式。AArch64 数据缓存按地址无效指令 DC IVAC 需要写权限，否则会生成权限错误。

所有指令高速缓存维护指令可以相对于其他指令高速缓存维护指令、数据高速缓存维护指令以及加载和存储以任何顺序执行，除非在指令之间执行 DSB。

除了 `DC ZVA` 之外，指定地址的数据缓存操作只有在指定相同地址时才能保证以相对于彼此的程序顺序执

行。那些指定地址的操作相对于所有未指定地址的维护操作按程序顺序执行。

考虑以下代码序列。

```
IC IVAU, X0 // Instruction Cache Invalidate by address to Point of Unification
DC CVAC, X0 // Data Cache Clean by address to Point of Coherency
IC IVAU, X1 // Might be out of order relative to the previous operations if x0 and x1␣
↪differ
```

前两条指令按顺序执行，因为它们引用相同的地址。但是，最终指令可能会相对于先前的操作重新排序，因为它引用了不同的地址

```
IC IVAU, X0 // I cache Invalidate by address to Point of Unification
IC IALLU // I cache Invalidate All to Point of Unification Operations execute in order
```

这仅适用于发出指令。只有在 DSB 指令之后才能保证完成。

使用 DC ZVA 指令预加载数据缓存为零值的能力是 ARMv8-A 中的新功能。处理器的运行速度明显快于外部存储器系统，有时从存储器加载高速缓存行可能需要很长时间。

高速缓存行归零的行为与预取类似，因为它是向处理器暗示将来可能使用某些地址的一种方式。但是，归零操作可以更快，因为无需等待外部存储器访问完成。

不是从内存中读取实际数据到缓存中，而是在缓存行中填充零。它可以向处理器提示代码完全覆盖了缓存行的内容，因此不需要初始读取。

考虑需要大型临时存储缓冲区或正在初始化新结构的情况。您可以让代码简单地开始使用内存，或者您可以编写代码在使用它之前预取它。两者都将使用大量周期和内存带宽来将初始内容读取到缓存中。通过使用缓存零选项，您可能会节省这种浪费的带宽并更快地执行代码。

可以根据指令是通过 VA 操作还是通过 Set/Way 操作来定义发生高速缓存维护指令的点。

可以选择范围，可以是 PoC，也可以是 PoU，可以广播的操作，参见第 14 章多核处理器，可以选择 Shareability。以下示例代码说明了将整个数据或统一缓存清理到 PoC 的通用机制。

```
MRS X0, CLIDR_EL1
AND W3, W0, #0x07000000 // Get 2 x Level of Coherence
LSR W3, W3, #23
CBZ W3, Finished
MOV W10, #0                                  // W10 = 2 x cache level
MOV W8, #1                                  // W8 = constant 0b1
Loop1: ADD W2, W10, W10, LSR #1 // Calculate 3 x cache level
LSR W1, W0, W2                                  // extract 3-bit cache type for this␣
↪level
AND W1, W1, #0x7
CMP W1, #2
B.LT Skip                                     // No data or unified cache at this␣
↪level
```

(continues on next page)

(continued from previous page)

```
MSR CSSELR_EL1, X10                 // Select this cache level
ISB                                                 // Synchronize change of CSSELR
MRS X1, CCSIDR_EL1 // Read CCSIDR
AND W2, W1, #7 // W2 = log2(linelen)-4
ADD W2, W2, #4 // W2 = log2(linelen)
UBFX W4, W1, #3, #10 // W4 = max way number, right aligned
CLZ W5, W4 /* W5 = 32-log2(ways), bit position of way in DC operand */
LSL W9, W4, W5 /* W9 = max way number, aligned to position in DC operand */
LSL W16, W8, W5 // W16 = amount to decrement way number per iteration
Loop2: UBFX W7, W1, #13, #15 // W7 = max set number, right aligned
LSL W7, W7, W2 /* W7 = max set number, aligned to position in DC operand */
LSL W17, W8, W2 // W17 = amount to decrement set number per iteration
Loop3: ORR W11, W10, W9 // W11 = combine way number and cache number...
ORR W11, W11, W7 // ... and set number for DC operand
DC CSW, X11 // Do data cache clean by set and way
SUBS W7, W7, W17 // Decrement set number
B.GE Loop3
SUBS X9, X9, X16 // Decrement way number
B.GE Loop2
Skip: ADD W10, W10, #2 // Increment 2 x cache level
CMP W3, W10
DSB /* Ensure completion of previous cache maintenance operation */
B.GT Loop1
```

需要注意的几点：

- 在正常情况下，清理或使整个缓存无效是只有固件应该做的事情，作为内核加电或断电序列的一部分。它也可能需要大量时间，L2 缓存中的行数可能非常大，并且有必要逐个循环它们。因此，这种清洁绝对只适用于特殊场合！
- 缓存维护操作（例如 DC CSW）在缓存维护（第 11-13 页）中进行了描述。
- 缓存必须在序列开始时禁用，以防止在序列中间分配新行。如果缓存是独占的，则一条线可以在级别之间迁移。
- 在 SMP 系统中，另一个内核可能能够从缓存中间序列中获取脏缓存行，从而阻止它们到达 PoC。Cortex-A53 和 Cortex-A7 处理器都可以做到这一点。
- 如果存在 EL3，则缓存必须从安全世界中失效，因为某些条目可能是无法从普通世界中失效的“安全脏”数据。如果保持不变，“安全脏”数据可能会在由于安全或正常世界中的正常缓存使用而被驱逐时破坏内存系统。

如果软件需要指令执行和内存之间的一致性，它必须使用 ISB 和 DSB 内存屏障和缓存维护指令来管理这种一致性。如下示例中所示的代码序列可用于此目的。

```
/* Coherency example for data and instruction accesses within the same Inner
Shareable domain. Enter this code with <Wt> containing a new 32-bit instruction,
to be held in Cacheable space at a location pointed to by Xn. */

STR Wt, [Xn]
DC CVAU, Xn // Clean data cache by VA to point of unification (PoU)
DSB ISH // Ensure visibility of the data cleaned from cache
IC IVAU, Xn // Invalidate instruction cache by VA to PoU
DSB ISH // Ensure completion of the invalidations
ISB // Synchronize the fetched instruction stream
```

此代码序列仅对适合单个 I 或 D-cache 行的指令序列有效。

该代码通过虚拟地址清除和使数据和指令缓存无效，该区域从 x0 中给出的基地址和 x1 中给出的长度开始。

```
//
// X0 = base address
// X1 = length (we assume the length is not 0)
//
// Calculate end of the region
ADD x1, x1, x0 // Base Address + Length
//
// Clean the data cache by MVA
//
MRS X2, CTR_EL0 // Read Cache Type Register
// Get the minimun data cache line
//
UBFX X4, X2, #16, #4 // Extract DminLine (log2 of the cache line)
MOV X3, #4 // Dminline iss the number of words (4 bytes)
LSL X3, X3, X4 // X3 should contain the cache line
SUB X4, X3, #1 // get the mask for the cache line
BIC X4, X0, X4 // Aligned the base address of the region
clean data cache:
DC CVAU, X4 // Clean data cache line by VA to PoU
ADD X4, X4, X3 // Next cache line
CMP X4, X1 // Is X4 (current cache line) smaller than the end
// of the region
B.LT clean_data_cache // while (address < end_address)
DSB ISH // Ensure visibility of the data cleaned from cache
//
//Clean the instruction cache by VA
//
// Get the minimum instruction cache line (X2 contains ctr_el0)
AND X2, X2, #0xF // Extract IminLine (log2 of the cache line)
MOV X3, #4 // IminLine is the number of words (4 bytes)
```

(continues on next page)

(continued from previous page)

```
LSL X3, X3, X2 // X3 should contain the cache line
SUB x4, x3, #1 // Get the mask for the cache line
BIC X4, X0, X4 // Aligned the base address of the region
clean_instruction_cache:
IC IVAU, X4 // Clean instruction cache line by VA to PoU
ADD X4, X4, X3 // Next cache line
CMP X4, X1 // Is X4 (current cache line) smaller than the end
// of the region
B.LT clean_instruction_cache // while (address < end_address)
DSB ISH // Ensure completion of the invalidations
ISB // Synchronize the fetched instruction stream
```

11.6 11.6 缓存发现

缓存维护操作可以通过缓存集、方式或虚拟地址来执行。与平台无关的代码可能需要知道缓存的大小、缓存行的大小、集合和路径的数量以及系统中有多少级缓存。重置后缓存失效和零操作最有可能出现此要求。架构缓存上的所有其他操作都可能在 PoC 或 PoU 的基础上进行。

有许多包含此信息的系统控制寄存器：

- 缓存级别的数量可以通过软件读取缓存级别 ID 寄存器 (CLIDR_EL1) 来确定。
- 缓存行大小在缓存类型寄存器 (CTR_EL0) 中给出。
- 如果这需要由运行在执行级别 EL0 的用户代码访问，这可以通过设置系统控制寄存器 (SCTLR/SCTLR_EL1) 的 UCT 位来完成。

需要对两个单独的寄存器进行异常级别访问，以确定高速缓存中的组数和路数。

- (1)。代码必须首先写入缓存大小选择寄存器 (CSSELR_EL1) 以选择您想要获取信息的缓存。
- (2)。然后代码读取缓存大小 ID 寄存器 (CCSIDR/CCSIDR_EL1)。
- (3)。数据缓存零 ID 寄存器 (DCZID_EL0) 包含要为零操作归零的块大小。
- (4)。SCTLR/SCTLR_EL1 的 [DZE] 位和 Hypervisor 配置寄存器 (HCR/HCR_EL2) 中的 [TDZ] 位控制哪些执行级别和哪些世界可以访问 DCZID_EL0。CLIDR_EL1、CSSELR_EL1 和 CCSIDR_EL1 只能通过特权代码访问，即 AArch32 中的 PL1 或更高版本，或 AArch64 中的 EL1 或更高版本。
- (5)。如果在异常级别禁止通过虚拟地址执行数据高速缓存零 (DC ZVA) 指令，EL0 由 SCTLR_EL1.DZE 位控制，EL1 和 EL0 中的非安全执行由 HCR_EL2 控制.TDZ 位，然后读取该寄存器返回一个值，指示该指令不受支持。
- (6)。CLIDR 寄存器只知道处理器本身集成了多少级缓存。它不能提供有关外部存储系统中任何缓存的信息。例如，如果只集成了 L1 和 L2，CLIDR/CLIDR_EL1 标识了两个级别的缓存，处理器不知道任何外部 L3 缓存。在执行缓存维护或与集成缓存保持一致性的代码时，可能需要考虑非集成缓存。

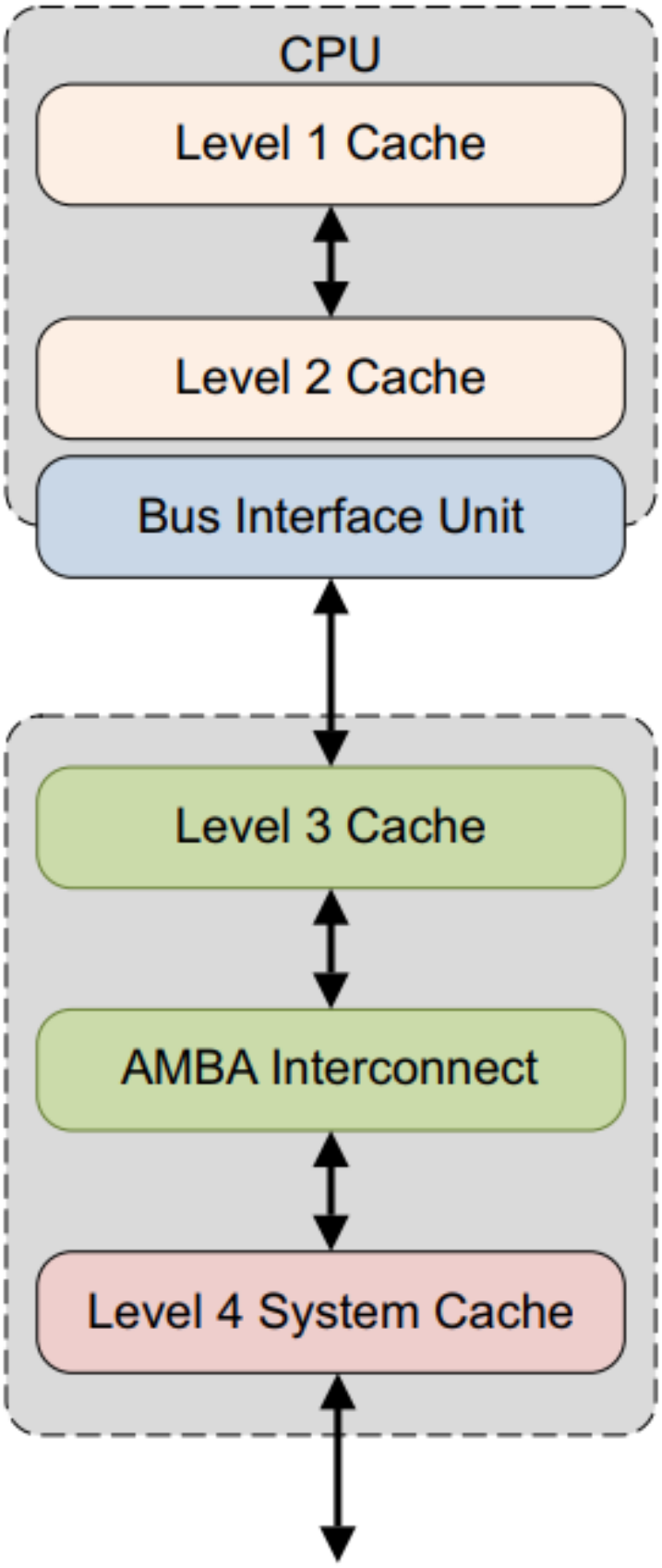
CPU
Level 1 Cache
Level 2 Cache
Bus Interface Unit
Level 3 Cache
AMBA Interconnect
Level 4 System Cache

image-

20220501094054765

此外，在 big.LITTLE 系统中，所描述的缓存层次结构可能因内核而异，例如，Cortex-A53 和 Cortex-A57 处理器具有不同的“CTR.L1IP”字段。

CHAPTER

TWELVE

12. 内存管理单元

内存管理单元（MMU）的一个重要功能是使系统能够运行多个任务，作为独立的程序运行在他们自己的私有虚拟内存空间。它们不需要了解系统的物理内存图，即硬件实际使用的地址，也不需要了解可能在同一时间执行的其他程序。

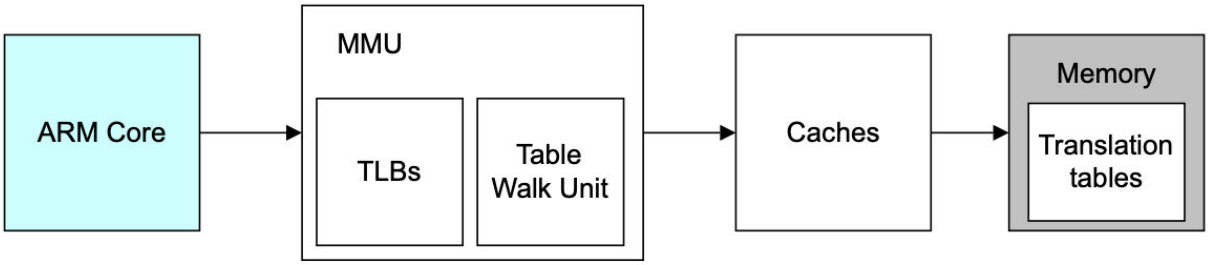

image-20220421130622557

你可以为每个程序使用相同的虚拟内存地址空间。你也可以使用一个连续的虚拟内存地图，即使物理内存是碎片化的。这个虚拟地址空间与系统中的实际物理内存地图是分开的。你可以编写、编译和链接应用程序以在虚拟内存空间中运行。

如下图所示的内存虚拟和物理视图的系统实例，一个系统中的不同处理器和设备可能有不同的虚拟和物理地址图。操作系统对 MMU 进行编程，在这两个内存视图之间进行转换。

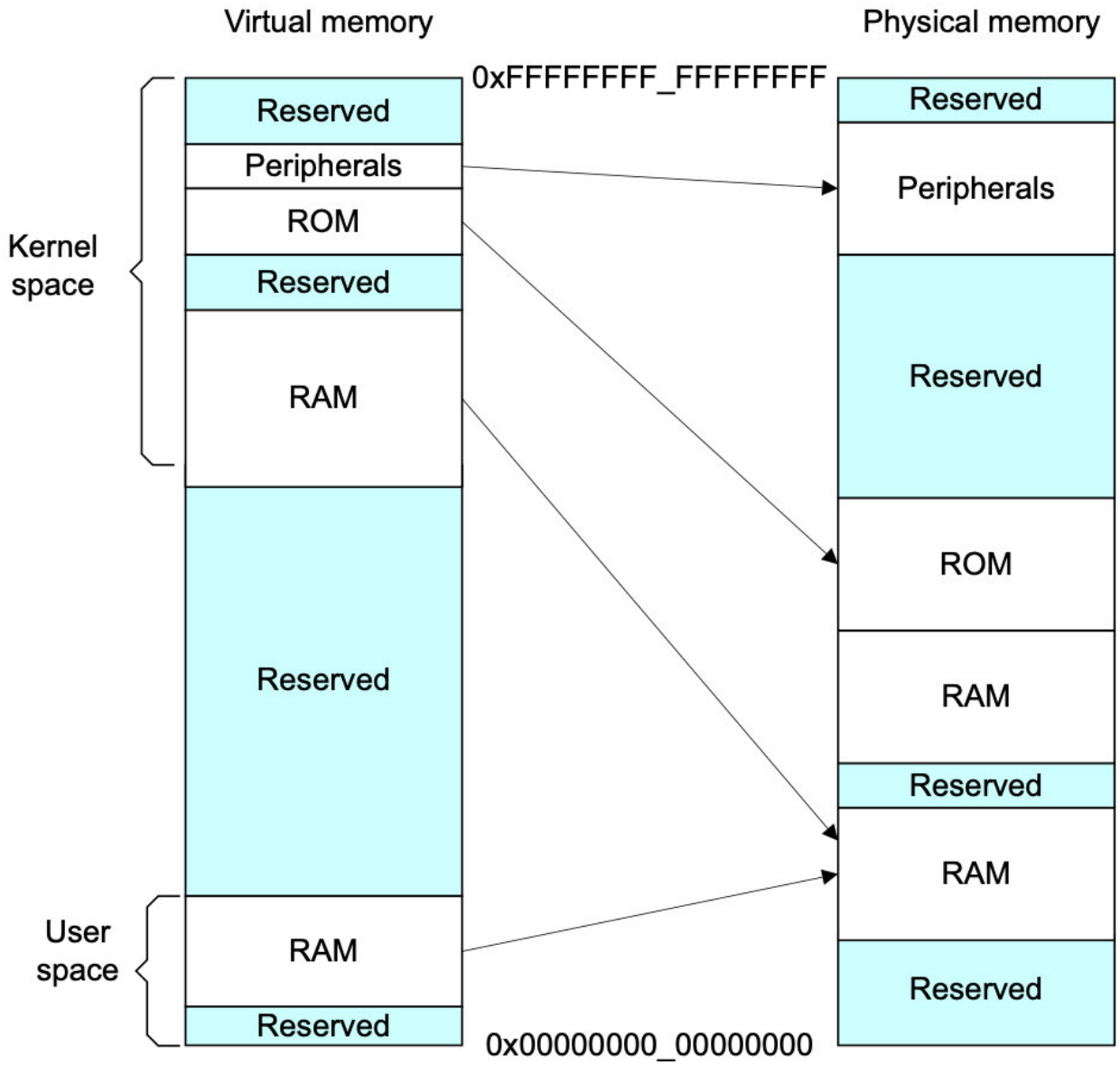

image-20220421130817155

要做到这一点，虚拟内存系统中的硬件必须提供地址转换，即把处理器发出的虚拟地址转换为主内存中的物理地址。

虚拟地址是你、编译器和链接器在内存中放置代码时使用的地址。物理地址是由实际的硬件系统使用的。

MMU 使用虚拟地址的最重要的位来索引映射表中的条目，并确定哪个块被访问。MMU 将代码和数据的虚拟地址映射成实际系统中的物理地址。这种转换是在硬件中自动进行的，对应用程序是透明的。除了地址转换外，MMU 还控制内存访问权限、内存排序和每个区域内存的缓存策略。

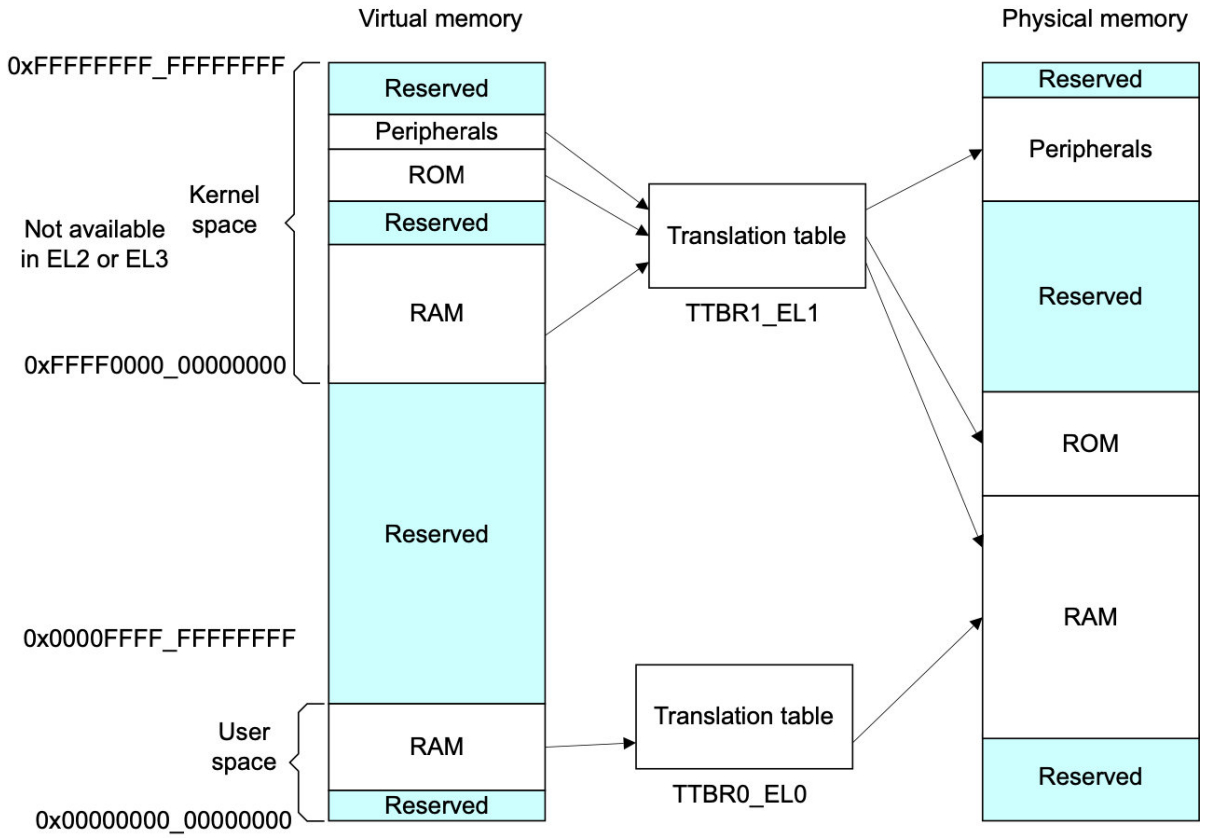

image-20220421131001911

MMU 使任务或应用程序的编写方式要求它们对系统的物理内存图或可能同时运行的其他程序一无所知。这使你可以为每个程序使用相同的虚拟内存地址空间。

它还允许你使用一个连续的虚拟内存地图，即使物理内存是碎片化的。这个虚拟地址空间与系统中的实际物理内存地图是分开的。应用程序被编写、编译和链接以在虚拟内存空间中运行。

12.1 12.1 转址旁路缓存（TLB）

转址旁路缓存（TLB）是 MMU 中最近访问的页面映射的缓冲区。对于处理器执行的每个内存访问，MMU 检查映射是否被缓存在 TLB 中。如果请求的地址翻译在 TLB 中引起了命中，那么该地址的映射就为立即可用。

每个 TLB 条目通常不仅包含物理地址和虚拟地址，还包含诸如内存类型、缓存策略、访问权限、地址空间 ID（ASID）和虚拟机 ID（VMID）等属性。如果 TLB 不包含处理器发出的虚拟地址的有效映射，称为 TLB 缺失，则执行外部映射表漫游或查找。MMU 内的专用硬件使其能够读取内存中的映射表。然后，新加载的映射可以被缓存在 TLB 中，以便在映射表行走没有导致页面故障的情况下进行重复使用。TLB 的确切结构在 ARM 处理器的实现之间有所不同。

如果操作系统修改了可能已缓存在 TLB 中的映射条目，那么操作系统就有责任使这些陈旧的 TLB 条目失效。

执行 A64 代码时，有一个 TLBI，这是一个 TLB 无效指令。

```
TLBI <type><level>{IS} {, <Xt>}
```

下面的列表给出了类型字段的一些比较常见的选择。完整的列表在表 12-1 中给出。

- **ALL** 所有 TLB 条目。
- **VMALL** 所有 TLB 条目。这是当前访客操作系统的第一阶段。
- **VMALLS12** 所有 TLB 条目。这是当前访客操作系统的第 1 和第 2 阶段。
- **ASID** 与 Xt 中的 ASID 匹配的条目
- **VA** Xt 中指定的虚拟地址和 ASID 条目。
- **VAA** 在 Xt 中指定的虚拟地址条目，具有任何 ASID。

每个异常级别，即 EL3、EL2 或 EL1，都有自己的虚拟地址空间，该操作适用于该空间。IS 字段指定这仅适用于内部可共享条目。

有关 ASID 和映射表配置的信息，请参阅上下文切换，以了解有关可共享性概念的更多信息。

字段只是指定操作应适用的异常级虚拟地址空间（可以是 3、2 或 1）。

IS 字段指定这仅适用于内部可共享条目。

Table 12-1 TLB configuration instructions

TLB invalidate	Variant	Description
TLBI	ALLEn	TLB invalidate All, EL*n*.
	ALLEnIS	TLB invalidate All, EL*n*, Inner Shareable.
	ASIDE1	TLB invalidate by ASID, EL1.
	ASIDE1IS	TLB invalidate by ASID, EL1, Inner Shareable.
	IPAS2E1	TLB invalidate by IPA, Stage 2, EL1.
	IPAS2E1IS	TLB invalidate by IPA, Stage 2, EL1, Inner Shareable.
	IPAS2LE1IS	TLB invalidate by IPA, Stage 2, Last level, EL1, Inner Shareable.
	VAAE1	TLB invalidate by VA, All ASID, EL1.
	VAAE1IS	TLB invalidate by VA, All ASID, EL1, Inner Shareable.
	VAALE1IS	TLB invalidate for the Last level, by VA, All ASID, EL1, Inner Shareable.
	VAEn	TLB invalidate by VA, EL*n*.
	VAEnIS	TLB invalidate by VA, EL*n*, Inner Shareable.
	VALEn	TLB invalidate by VA, Last level, EL*n*.
	VALEnIS	TLB invalidate by VA, Last level, EL*n*, Inner Shareable.
	VMALLE1	TLB invalidate by VMID, All at stage 1, EL1.
	VMALLE1IS	TLB invalidate by VMID, EL1, Inner Shareable.
	VMALLS12E1	TLB invalidate by VMID, All at Stage 1 and 2, EL1.
	VMALLS12E1	TLB invalidate by VMID, All at Stage 1 and 2, EL1.
	VMALLS12E1IS	TLB invalidate by VMID, All at Stage 1 and 2, EL1 Inner Shareable.
	VMALLS12E1IS	TLB invalidate by VMID, All at Stage 1 and 2, EL1 Inner Shareable.

image-20220421132016485

下面的代码例子显示了对由内部可共享内存支持的映射表的写入序列。

```
<< Writes to Translation Tables >>
DSB ISHST // ensure write has completed
TLBI ALLE1 // invalidate all TLB entries
DSB ISH // ensure completion of TLB invalidation
ISB // synchronize context and ensure that no instructions are
                // fetched using the old translation
```

有关示例中显示的 DSB 和 ISB 屏障指令的更多信息，请参阅第 13-6 页。

例如，要更改单个条目，请使用以下指令：

```
TLBI VAE1, X0
```

这将使与寄存器 X0 中指定的地址关联的条目无效。

TLB 可以容纳固定数量的条目。你可以通过最大限度地减少由映射表遍历引起的外部内存访问数量并获得高 TLB 命中率来实现最佳性能。ARMv8-A 架构提供了一个称为连续块条目的功能，以有效地使用 TLB 空间。映射表块条目每个都包含一个连续的位。当设置时，该位向 TLB 发出信号，它可以缓存一个涵盖多个块的映射的单一条目。一个查找可以索引到一个连续块所覆盖的地址范围的任何地方。因此，TLB 可以为定义的地址范围缓存一个条目，使得在 TLB 中存储更大范围的虚拟地址成为可能。

要使用一个连续的位，连续的块必须是相邻的，也就是说它们必须对应于一个连续的虚拟地址范围。它们必须从一个对齐的边界开始，具有一致的属性，并指向同一级别转换的连续输出地址范围。要求的对齐方式是，4KB 颗粒的 VA[20:16] 或 64KB 颗粒的 VA[28:21]，对所有地址都是一样的。需要以下数量的连续块：

- 16×4KB 相邻块，带有 4KB 颗粒的 64KB 块条目。
- 32×32MB 相邻块，为 L2 描述符提供 1GB 条目，128×16KB 为使用 16KB 颗粒时，L3 描述符的 2MB 条目。
- 32×64Kb 相邻块，带有 64KB 颗粒的 2MB 块条目。

如果不满足这些条件，则会发生编程错误，这可能会导致 TLB 中止或查找损坏。这种错误的可能例子包括：

- 一个或多个表条目没有连续的位集。
- 其中一个条目的输出点在对齐范围之外。

对于 ARMv8 架构，不正确的使用不允许逃避 EL0 和 EL1 有效地址空间之外的权限检查，或者错误地提供对 EL3 空间的访问。

12.2 12.2 内核和应用程序虚拟地址空间的分离

操作系统通常有许多应用程序或任务在同时运行。每个任务都有自己独特的转换表，内核在一个任务和另一个任务之间切换时，会从一个转换表切换到另一个。然而，大部分内存系统仅由内核使用，并且具有固定的虚拟到物理地址的映射，其中转换表项很少改变。ARMv8 架构提供了许多功能来有效地处理这一要求。

映射表基地址在映射基础寄存器（TTBR0_EL1）和（TTBR1_EL1）中指定。当 VA 的上面几位都是 0 时，TTBR0 所指向的映射表被选中，当 VA 的上面几位都被设置为 1 时，TTBR1 被选中。你可以启用 VA 标记，将前八位排除在检查之外。

来自指令获取或数据访问的处理器的虚拟地址是 64 位。然而，你必须在一个 48 位的物理地址内存映射中映射上面定义的两个区域。。

EL2 和 EL3 有一个 TTBR0，但没有 TTBR1。这意味着：

- 如果 EL2 使用的是 AArch64，它只能使用范围在 0x0 到 0x0000FFFF_FFFFFFFF。
- 如果 EL3 使用的是 AArch64，它只能使用范围在 0x0 到 0x0000FFFF_FFFFFFFF。

下面的图显示了内核空间如何被映射到内存的最重要区域，以及与每个应用程序相关的虚拟地址空间如何被映射到内存的最不重要区域。然而，这两者都被映射到一个小得多的物理地址空间。

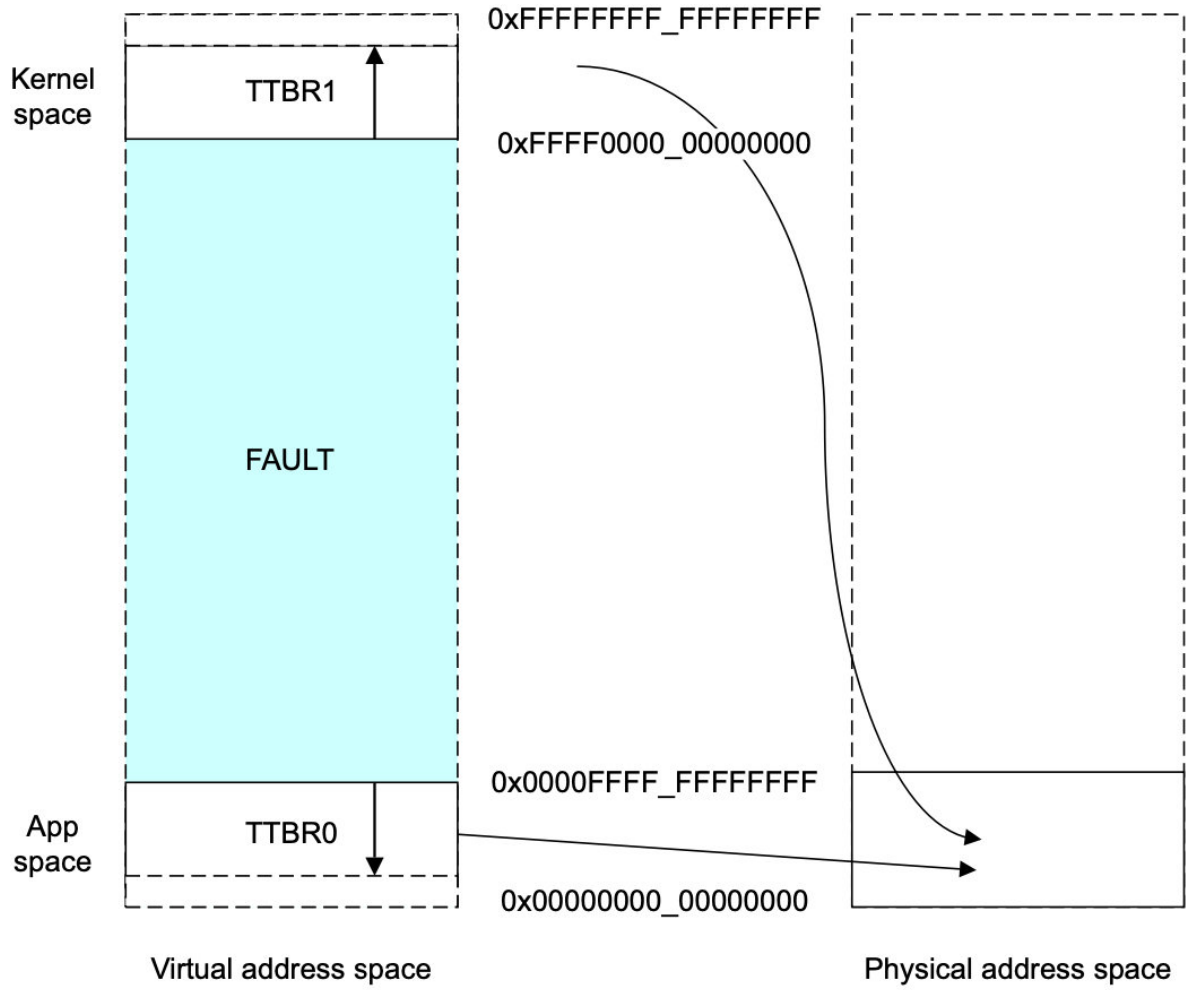

image-20220421135316499

映射控制寄存器 TCR_EL1 定义了被检查的最有效位的确切数量。TCR_EL1 包含大小字段 T0SZ[5:0] 和 T1SZ[5:0]。该字段中的整数给出了必须全部为 0 或全部为 1 的最有意义位的数量。这些字段有指定的最小值和最大值，它们随颗粒大小和起始表级别而变化。因此，你必须始终使用这两个空间，在所有系统中至少需要两个映射表。一个没有操作系统的简单裸机系统仍然需要一个只包含故障项的小的上层映射表。

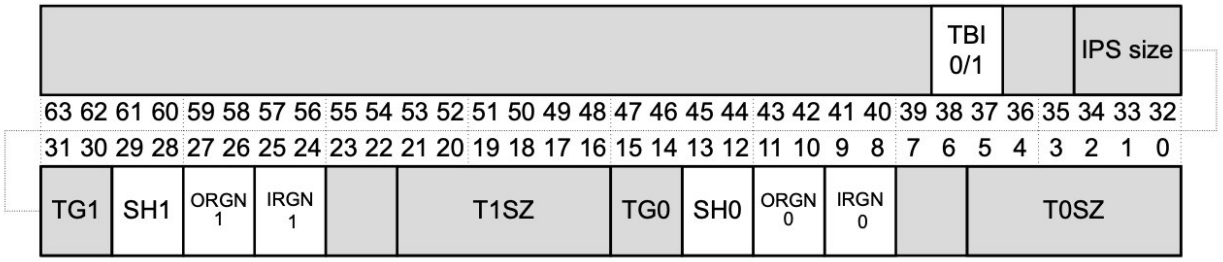

Figure 12-5 Translation table control configuration

image-20220421142717196

TCR_EL1 控制 EL1 和 EL0 的其他内存管理功能。图 12-5 仅显示了控制地址范围和颗粒大小的字段。

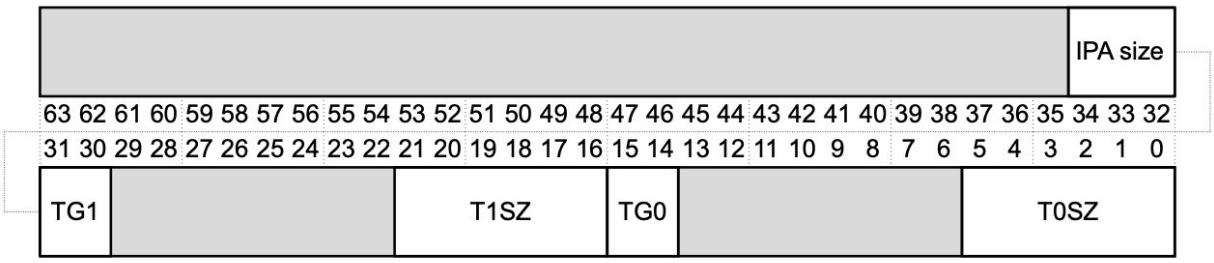

Figure 12-6 Translation table control register image-20220421142808439

中级物理地址大小（IPS）字段控制最大输出地址大小。如果映射指定了超出此范围的输出地址，则访问失败，000=32 位物理地址，101=48 位。两位映射颗粒（TG）TG1 和 TG0 字段分别给出内核或用户空间的颗粒大小，00=4KB，01=16KB，11=64KB。

你可以配置用于第一次查找的映射表级别。完整的映射过程可能需要三到四级的映射表。你不需要实现所有等级的映射表。第一级的查找实际上是由颗粒大小和 TCR_ELn.TxSZ 字段决定的。您可以为 TTBR0_EL1 和 TTBR1_EL1 单独配置。

12.3 12.3 将虚拟地址转换为物理地址

当处理器为指令获取或数据访问发出一个 64 位的虚拟地址时，MMU 硬件会将虚拟地址转换为相应的物理地址。对于一个虚拟地址来说，前 16 位 [63:47] 必须全部是 0 或 1，否则地址会触发一个故障。

然后，最不重要的位被用来提供所选部分的偏移量，这样 MMU 就把块表项的物理地址位和原始地址的最不重要的位结合起来，产生最终地址。

该架构还支持标记地址。这个地址中最重要的 8 位容易被忽略（被视为不是地址的一部分）。这意味着这些位可以用于其他方面，例如，记录关于一个指针的信息。

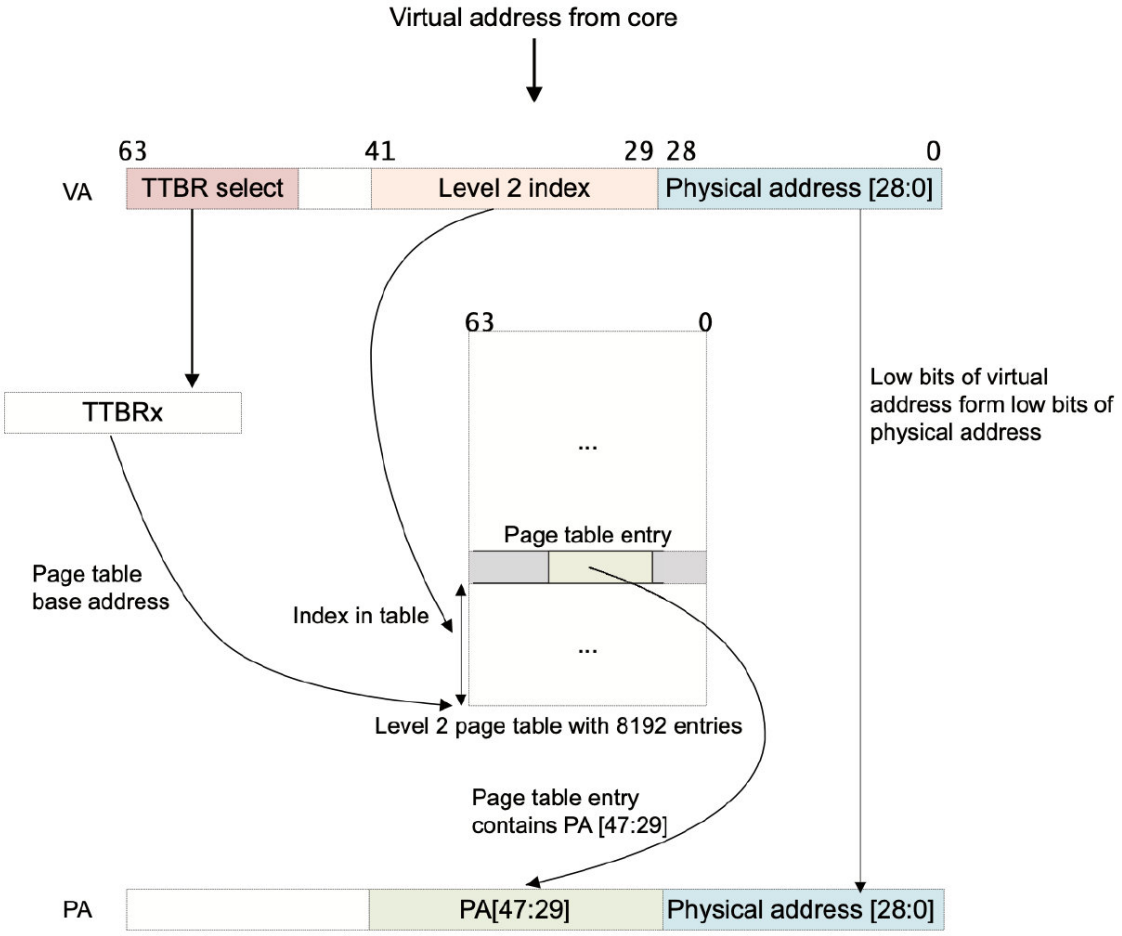

Figure 12-7 Virtual to Physical Address translation for a 512MB block image-20220421142915864

在一个简单的地址转换中，只涉及一个层次的查找。假设我们使用的是一个 64KB 的颗粒，有一个 42 位的虚拟地址。MMU 映射虚拟地址的方法如下:

1. 如果 VA[63:42] = 1，则 TTBR1 用于第一页表的基地址。当 VA[63:42] = 0 时，TTBR0 用于第一页表的基地址。
2. 页表包含 8192 个 64 位页表条目，并使用 VA[41:29] 进行索引。MMU 从表中读取相关的 2 级表格条目。
3. MMU 检查页面表条目的有效性，以及是否允许请求的内存访问。假设它有效，则允许内存访问。
4. 在图 12-7 中，页表条目指的是 512MB 的页面（它是一个块描述符）。
5. Bit[47:29] 取自此页面表条目，并形成物理地址的 Bit[47:29]。
6. 由于我们有一个 512MB 的页面，VA 的 Bit[28:0] 被取为 PA[28:0]。请参阅第 12-15 页颗粒大小对映射表的影响。
7. 返回完整的 PA[47:0]，以及页面表条目中的其他信息。

在实践中，如此简单的映射过程严重限制了您如何细微地划分地址空间。一级表条目可以指向二级页面表，而不是只使用此一级映射表。

通过这种方式，操作系统可以进一步将很大一部分虚拟内存划分为更小的页面。对于二级表，一级描述符包含二级页面表的物理基地址。与处理器请求的虚拟地址相对应的物理地址可以在二级描述符中找到。

下面的图图显示了一个 64 位颗粒从第 1 阶段第 2 级开始的正常 64KB 页面的转换实例。

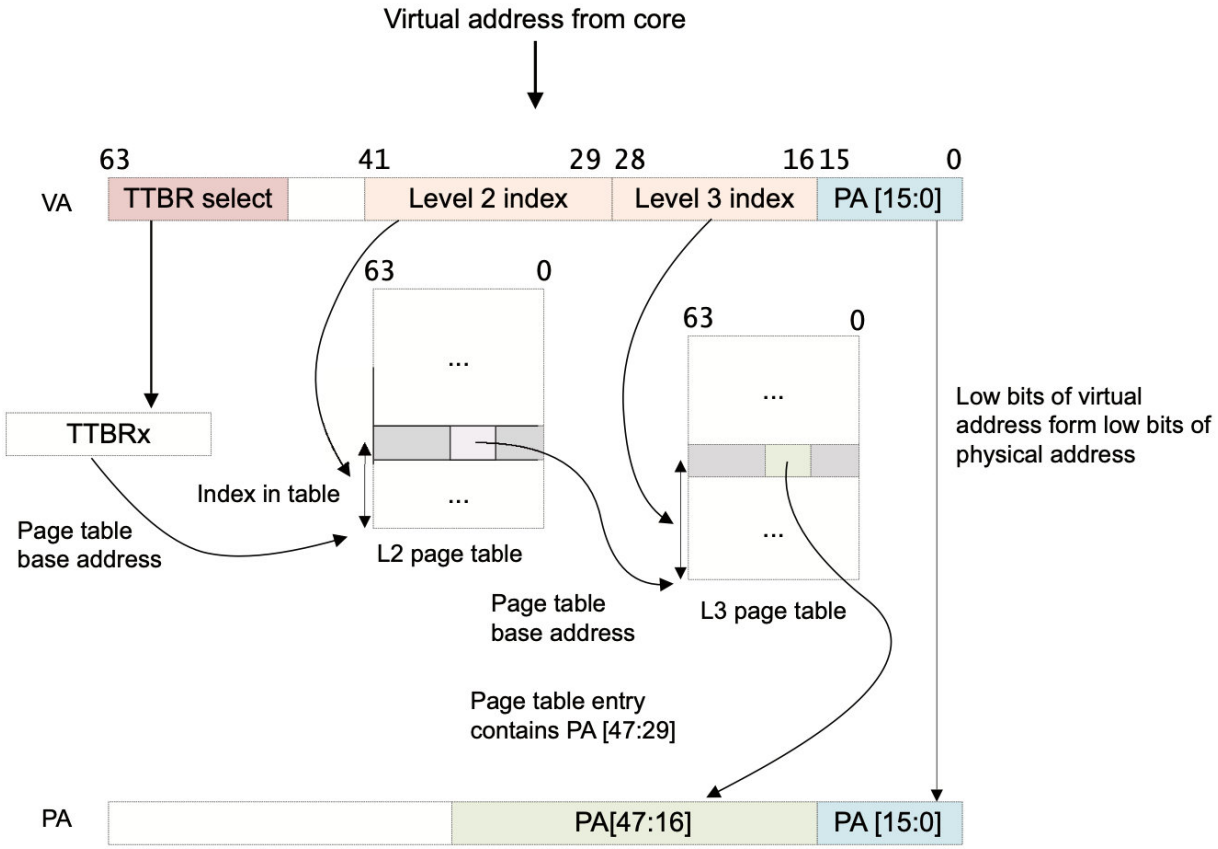

Figure 12-8 Virtual to Physical Address translation for a 64KB page image-20220421193056756

每个二级表都与一个或多个一级条目相关。你可以有多个一级描述符指向同一个二级表，这意味着你可以将多个虚拟地址别名到同一个物理地址。

图 12-8 描述了一种有两层查找的情况。同样，这也假设了一个 64KB 的颗粒和 42 位的虚拟地址空间。

1. 如果 VA[63:42] = 1，则 TTBR1 用于第一页表的基地址。当 VA[63:42] = 0 时，TTBR0 用于第一页表的基地址。
2. 页面表包含 8192 个 64 位页面表条目，并通过 VA[41:29] 进行索引。MMU 从表中读取相关的二级表格条目。
3. MMU 检查二级页面表条目的有效性，以及是否允许请求的内存访问。假设它有效，则允许内存访问
4. 在图 12-8 中，二级页表条目是指三级页表的地址（它是一个表描述符）。
5. Bit[47:16] 取自二级页表条目，形成三级页表的基地址。
6. VA 的 Bit[28:16] 用于索引 3 级页面表条目。MMU 从表格中读取相关的 3 级表格条目。

7. MMU 检查三级页面表条目的有效性，以及是否允许请求的内存访问。假设它有效，则允许内存访问。
8. 在图 12-8 中，三级页面表条目指的是 64KB 页面（它是一个页面描述符）。
9. Bit[47:16] 取自三级页面表条目，用于形成 PA[47:16]。
10. 由于我们有一个 64KB 页面，VA[15:0] 被取为 PA[15:0]。
11. 返回完整的 PA[47:0]，以及页面表条目中的其他信息。

12.3.1 12.3.1 安全和非安全地址

理论上，安全和非安全物理地址空间是相互独立的，并行存在。一个系统可以设计为有两个完全独立的内存系统。然而，大多数真实系统将安全和非安全视为访问控制的属性。正常（非安全）世界只能访问不安全的物理地址空间。安全世界可以访问两个物理地址空间。同样，这是通过映射表控制的。

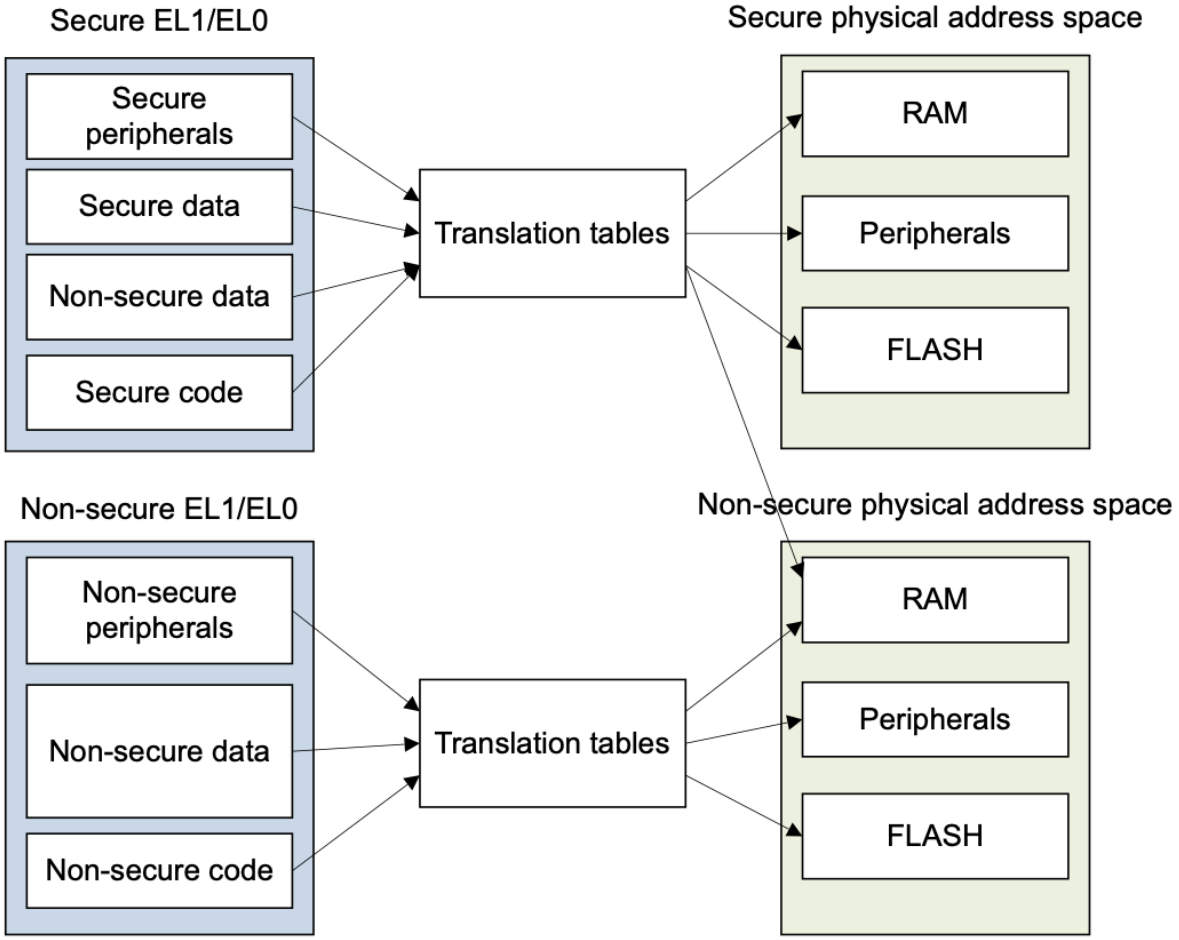

Figure 12-9 Physical Address spaces image-20220421193836859

这也会对缓存的一致性产生影响。例如，由于安全的 0x8000 和非安全的 0x8000 从技术上讲是不同的物理地

址，它们可以同时出现在高速缓存中。

在一个安全内存和非安全内存位于不同位置的系统中，不会有问题。更有可能的是，它们会在同一个位置。理想情况下，内存系统会阻止对非安全内存的安全访问和对安全内存的非安全访问。在实践中，大多数只阻止对安全内存的非安全访问。同样，这意味着你可能会在缓存中出现两次相同的物理内存，安全和非安全的。这始终是一个编程错误。为了避免这种情况，安全世界必须始终使用非安全的方式来访问非安全的内存。

12.3.2 12.3.2 配置和启用 MMU

对控制 MMU 的系统寄存器的写入是切换上下文的事件，它们之间不存在排序要求。这些事件的结果不能保证在上下文同步事件之前被看到（见第 13-6 页的屏障）。

```
MSR TTBR0_EL1, X0 // Set TTBR0
MSR TTBR1_EL1, X1 // Set TTBR1
MSR TCR_EL1, X2 // Set TCR
ISB        // The ISB forces these changes to be seen before the MMU is enabled.
MRS X0, SCTLR_EL1 // Read System Control Register configuration data
ORR X0, X0, #1 //Set [M] bit and enable the MMU.
MSR SCTLR_EL1, X0 // Write System Control Register configuration data
ISB         // The ISB forces these changes to be seen by the next instruction
```

这与扁平映射的要求是不同的，扁平映射是为了确保我们知道在写入 SCTL_EL1.M 后直接执行哪条指令。如果我们看到写入的结果，那就是使用新的转换机制的 VA+4 的指令。如果我们没有看到结果，它仍然是在 VA+4 的指令，但是 VA=PA。ISB 在这里没有帮助，因为我们不能保证它是下一条执行的指令，除非我们使用扁平映射。

12.3.3 12.3.3 停用内存管理单元时的操作

当阶段 1 MMU 被禁用时，对于不安全的 EL0 和 EL1，当 HCR_EL2 时访问。直流位设置为启用数据缓存，默认内存类型为正常不可共享，内部回写读写分配，外部回写读写分配。

12.4 12.4 ARMv8-A 中的转址表

ARMv8-A 架构支持三组不同的转址表形式：

- ARMv8-A AArch64 长描述符格式。
- ARMv7-A 长描述符格式，例如 ARM Cortex-A15 处理器中的 ARMv7-A 架构的大物理地址扩展（LPAE）。
- ARMv7-A 短描述符格式。

在 AArch32 状态下，你可以使用现有的 ARMv7-A 长描述符和短描述符格式来运行现有的客户操作系统和现有的应用程序代码，无需修改。ARMv7-A 短描述符只能在 EL0 和 EL1 阶段 1 映射中使用。因此，它们不能被管理程序或安全监控代码使用。

在 AArch64 执行状态下始终使用 ARMv8-A 长描述符格式。这与带有大型物理地址扩展的 ARMv7-A 长描述符格式非常相似。它使用相同的 64 位长描述符格式，但有一些变化。它引入了一个新的 0 级表索引，它使用与 1 级表相同的描述符格式。增加了对多达 48 位输入和输出地址的支持。输入虚拟地址现在来自一个 64 位寄存器。然而，由于该架构不支持完整的 64 位寻址，地址的第 63:48 位必须全部相同，即全部为 0 或全部为 1，或者前 8 位可用于 VA 标记。

AArch64 支持映射三种不同的颗粒。这些定义了映射表最底层的块大小，并控制使用中的映射表的大小。较大的颗粒大小减少了所需的页表层数，这在使用管理程序提供虚拟化的系统中可能成为一个重要的考虑因素。

支持的颗粒大小为 4KB、16KB 和 64KB，支持这三种颗粒的大小是由具体设计方案决定的。创建页表的代码能够读取系统寄存 ID_AA64MMFR0_EL1，以了解哪些是支持的大小。Cortex-A53 处理器支持所有三种尺寸，但一些早期版本的处理器不是这样的，比如 Cortex-A57，它不支持 16K 颗粒尺寸。在映射控制寄存器（TCR_EL1）中，每个映射表的大小都可以配置。

12.4.1 12.4.1 AArch64 描述符格式

你可以在所有级别的表中使用描述符格式，从 0 级到 3 级。0 级描述符只能输出 1 级表的地址。三级描述符不能指向另一个表，只能输出块地址。因此，三级表的格式略有不同。

下面的图显示了表描述符类型由条目的 Bit[1:0] 标识，可以参考以下任一个：

- 下一级表的地址，在这种情况下，内存可以进一步细分为较小的块。
- 可变大小的内存块的地址。
- 表条目，可以标记为 Fault 或 Invalid。

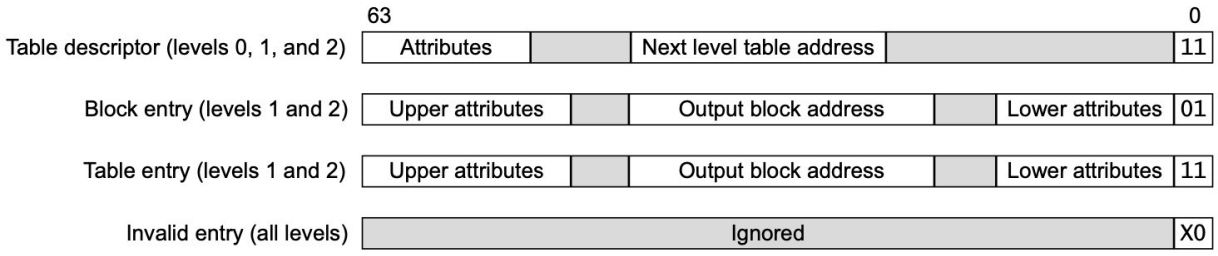

Figure 12-10 A64 Table descriptor type image-20220421194527603

为了清晰起见，此图表没有指定位字段的宽度。

12.4.2 12.4.2 颗粒尺寸对映射表的影响

三种不同的颗粒尺寸可能会影响所需映射表的数量和大小。

在所有情况下，如果 VA 输入范围限制为 42 位，则可以省略一级映射表。

根据可能的 VA 范围的大小，级数可能会更少。例如，对于 4KB 颗粒，如果 TTBCR 设置为低地址仅跨度 1GB，则不需要级别 0 和 1，映射从 2 级开始，4KB 页面从 3 级开始。

- **4KB**

  当你使用 4kB 的颗粒大小时，硬件可以使用 4 级查找过程。48 位地址每一级有 9 个地址位被翻译，即每一级有 512 个条目，最后的 12 位在 4kB 内选择一个字节，直接来自原始地址。

  虚拟地址的第 47:39 位索引到 512 个条目的 L0 表。每个表项都跨越 512GB 范围，并指向一个 L1 表。在这个 512 条目的 L1 表中，第 38:30 位被用作索引来选择一个条目，每个条目都指向一个 1GB 块或一个 L2 表。位 29:21 索引到一个 512 条目的 L2 表，每个条目指向一个 2MB 块或下一级表。在最后一级，第 20:12 位索引到一个 512 条目的 L2 表，每个条目指向一个 4kB 的块。

VA bits [47:39]	VA bits [38:30]	VA bits [29:21]	VA bits [20:12]	VA bits [11:0]
Level 0 Table Index Each entry contains: Pointer to L1 table (No block entry)	Level 1 Table Index Each entry contains: Pointer to L2 table Base address of 1GB block (IPA)	Level 2 Table Index Each entry contains: Pointer to L3 table Base address of 2MB block (IPA)	Level 3 Table Index Each entry contains: Base address off 4KB block (IPA)	Block offset and PA [11:0]

Figure 12-11 4KB Granule image-20220421194732495

- **16KB**

  当你使用 16KB 的颗粒大小时，硬件可以使用 4 级查找过程。48 位的地址每一级映射有 11 个地址位，即每一级有 2048 个条目，最后的 14 位在 4KB 内选择一个字节，直接来自原始地址。0 级表只包含两个条目。虚拟地址的第 47 位从两个条目的 L0 表中选择一个描述符。这些表项中的每一个都跨越了 128TB 的范围，并指向一个 L1 表。在这个 2048 个条目的 L1 表内，第 46:36 位被用作索引来选择一个条目，每个条目指向一个 L2 表。第 35:25 位索引到一个 2048 个条目的 L2 表，每个条目指向一个 32MB 的块或下一个表层。在最后的映射阶段，Bit[24:14] 索引到一个 2048 个条目的 L2 表，每个条目指向一个 16KB 的块。

VA bit [47]	VA bits [46:36]	VA bits [35:25]	VA bits [24:14]	VA bits [13:0]
Level 0 Table Index Each entry contains: Pointer to L1 table (No block entry)	Level 1 Table Index Each entry contains: Pointer to L2 table	Level 2 Table Index Each entry contains: Pointer to L3 table Base address of 32MB block (IPA)	Level 3 Table Index Each entry contains: Base address off 16KB block (IPA)	Block offset and PA [13:0]

Figure 12-12 16KB Granule image-20220421194900253

- **64KB**

当你使用 64kB 的颗粒大小时，硬件可以使用 3 级查找过程。第 1 级表只包含 64 个条目。

虚拟地址的第 47:42 位从 64 个条目的 L1 表中选择一个描述符。这些表项中的每一个都跨越了 4TB 的范围，并指向一个 L2 表。在这个 8192 个条目的 L2 表内，第 41:29 位被用作索引来选择一个条目，每个条目指向一个 512MB 的块或一个 L2 表。在最后的第 28:16 位索引到一个 8192 条目的 L3 表，每个条目都指向一个 64kB 的块。

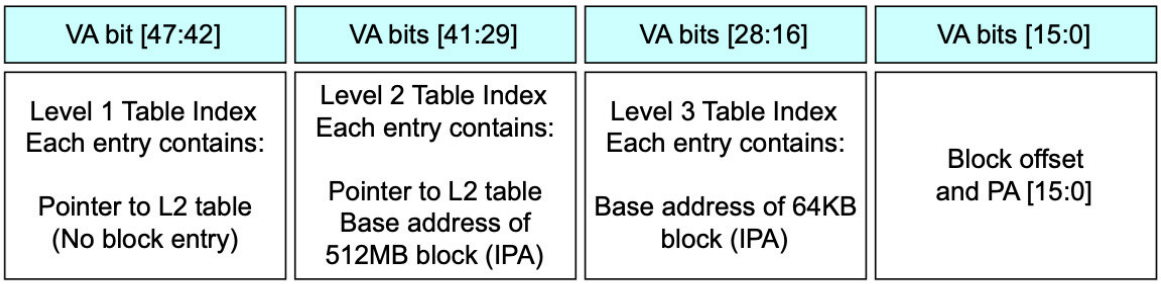

Figure 12-13 64KB Granule image-20220421195005584

12.4.3 12.4.3 缓存配置

MMU 使用映射表和映射寄存器来控制哪些内存位置是可缓存的。MMU 控制缓存策略、内存属性和访问权限，并提供虚拟到物理地址的映射。

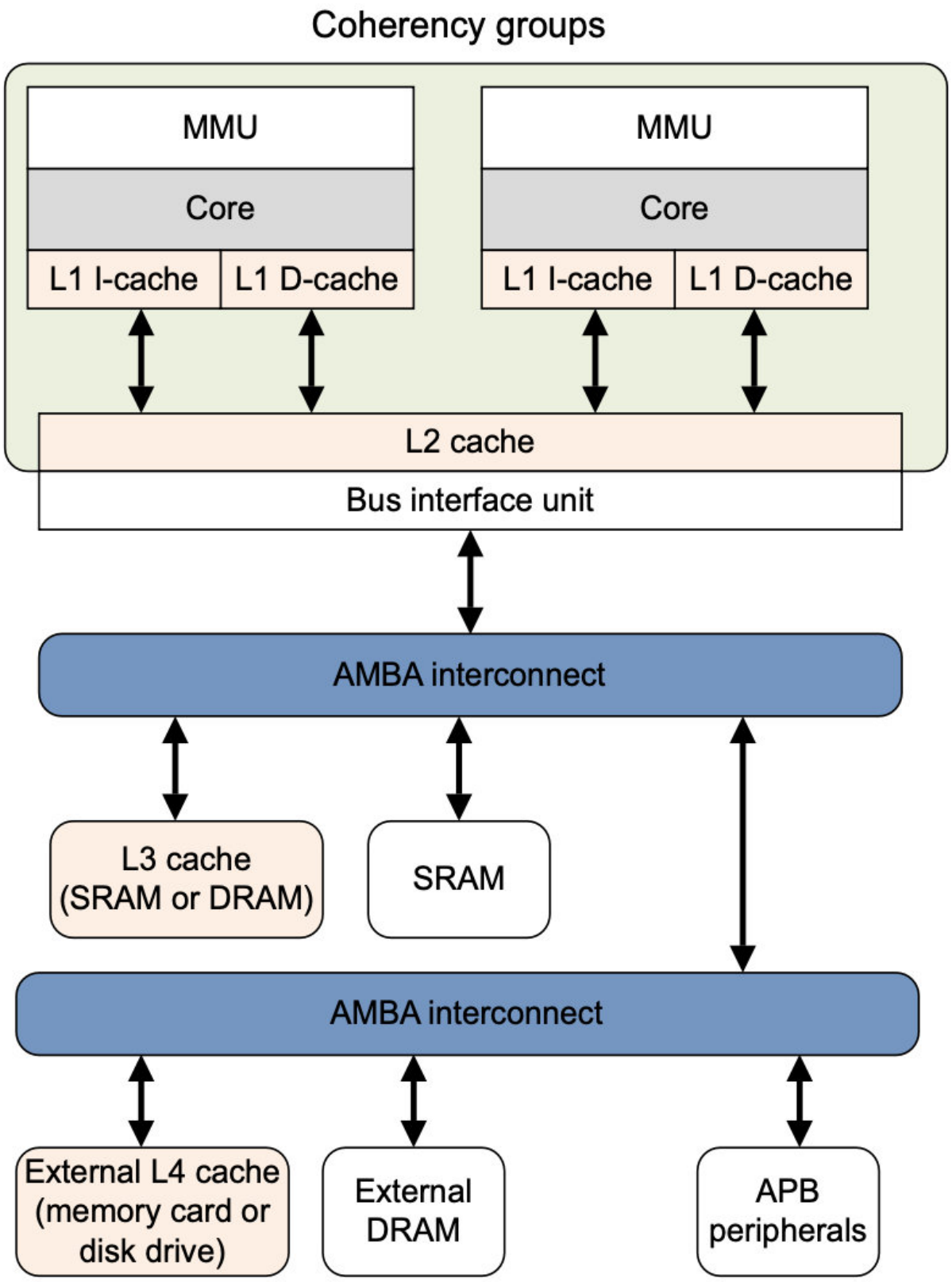

Figure 12-14 Memory busses and caches image-

20220421195158817

软件配置由系统寄存器执行（其中一些列在第 4 章 ARMv8 寄存器中）。

在一些设计中，外部存储器系统可能包含进一步的特定实现的外部存储器的缓存。

12.4.4 12.4.4 缓存策略

MMU 映射表也定义了内存系统中每个块的缓存策略。被定义为 Normal 的内存区域可能被标记为可缓存或不可缓存。映射表项的位 [4:2] 指的是内存属性定向寄存器（MAIR）中的八个内存属性编码之一。内存属性编码指定了访问该内存时要使用的缓存策略。这些都是对处理器的提示，在一个特定的实现中是否支持所有的缓存策略，以及哪些缓存数据被认为是连贯的，这都是实施定义的。一个内存区域可以用它的可共享属性来定义。

12.5 12.5 转址表配置

除了在 TLB 中存储单个映射关系外，你可以配置 MMU 将映射表存储在可缓存的内存中。这通常比总是从外部存储器读取映射表要快得多。TCR_EL1 有额外的字段来控制这个。

这些附加字段指定了 TTBR0 和 TTBR1 的映射表的可缓存性和可共享性。相关的字段被称为 SH0/1 Shareability，IRGN0/1 Inner Cacheable，和 ORGN0/1 Outer Cacheable。表 12-2 显示了可缓存性的允许设置。

Table 12-2 Cacheability settings

IRGN/ORGN bits for TTBR0/TTBR1	Cacheable Property
00	Normal memory, Inner Non-cacheable
01	Normal memory, Inner Write-Back Write-Allocate Cacheable
10	Normal memory, Inner Write-Through Cacheable
11	Normal memory, Inner Write-Back no Write-Allocate Cacheable

image-20220421195349059

内存的可共享性的对应表与映射表的行走有关。对于一个设备或强排序的内存区域，该值被忽略。

Table 12-3 Memory shareability

SH0 bits[13:12]	Shareability
00	Non-shareable
01	UNPREDICTABLE
10	Outer shareable
11	Inner shareable

image-20220421195423269

在 TCR_EL1 中指定的属性必须与为存储映射表的虚拟内存区域指定的属性相同。缓存映射表是正常的默认行为。

12.5.1 12.5.1 虚拟地址标记

映射控制寄存器 TCR_ELn 有一个额外的字段，称为顶部字节忽略（TBI），提供标记寻址支持。通用寄存器是 64 位宽，但地址中最重要的 16 位必须全部是 0xFFFF 或 0x0000。任何试图使用不同位值的行为都会触发故障。

当标签寻址支持被启用时，前八位，也就是虚拟地址的 [63:56] 被处理器忽略了。它在内部设置位 [55]，将地址扩展为 64 位格式。然后，虚拟地址的前八位可以用来传递数据。这些位在寻址和映射故障中被忽略。TCR_EL1 对 EL0 和 EL1 有单独的启用位。ARM 没有规定或授权标记寻址的具体使用情况。

一个用例可能是支持面向对象的编程语言。除了拥有一个指向对象的指针外，可能还有必要保留一个引用计数，以跟踪指向对象的引用或指针或句柄的数量，例如，这样，自动垃圾收集代码就可以取消分配不再被引用的对象了。这个引用计数可以作为标签地址的一部分来存储，而不是在一个单独的表中，这样可以加快创建或销毁对象的过程。

12.6 12.6 EL2 和 EL3 的转址

ARMv8-A 架构的虚拟化扩展引入了第二个映射阶段。当系统中存在一个管理程序时，可能会出现一个或多个客户操作系统。这些系统继续使用 TTBRn_EL1，如前所述，MMU 操作看起来没有变化。

管理程序必须在一个两阶段的过程中执行一些额外的转换步骤，以便在不同的客户操作系统之间共享物理内存系统。在第一阶段，一个虚拟地址（VA）被映射成中间物理地址（IPA）。这通常是在操作系统的控制之下。第二阶段，由管理程序控制，然后执行 IPA 到最终物理地址（PA）的翻译。管理程序和安全监控器也有他们自己的代码和数据的第一阶段映射表，直接从 VA 到 PA 进行映射。

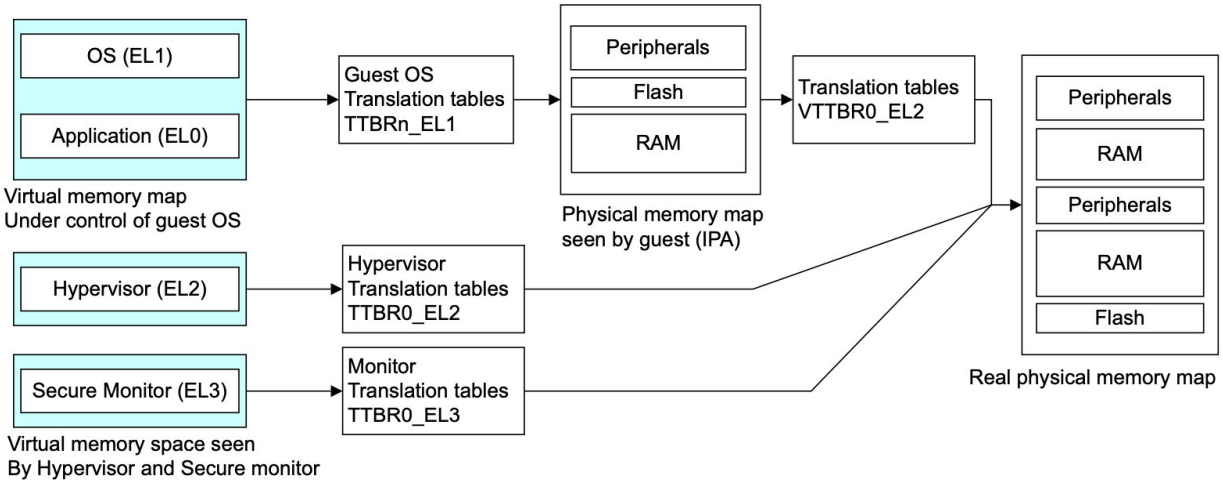

image-20220421195641972

第二阶段的转换，将中间物理地址转换为物理地址，使用一套额外的表，由管理程序控制。这些必须通过写入管理程序配置寄存器 HCR_EL2 来明确启用。这个过程只适用于非安全的 EL1/0 访问。

这个第二阶段映射表的基础地址在虚拟化映射表基础寄存器 VTTBR0_EL2 中指定。它指定了内存底部的一个连续的地址空间。支持的地址空间的大小在虚拟化映射控制寄存器 VTCR_EL2 的 TSZ[5:0] 字段中指定。

该寄存器的 TG 字段指定了颗粒大小，而 SL0 字段控制了第一级表的查找。任何超出定义的地址范围的访问都会导致映射故障。

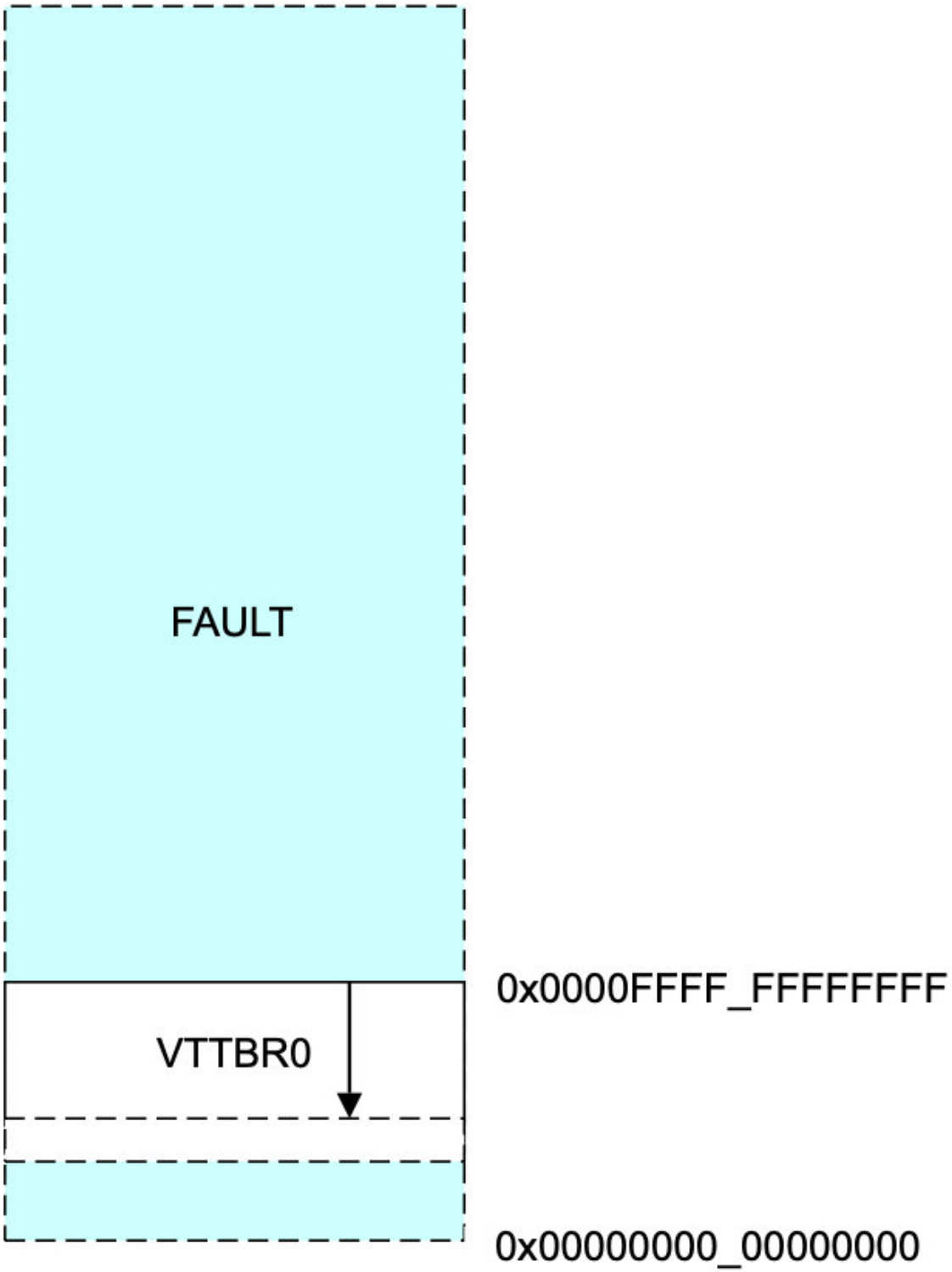

Figure 12-16 Maximum IPA space

image-20220421195726919

管理程序 EL2 和安全监控器 EL3 有自己的一级表，直接从虚拟空间映射到物理地址空间。表的基址分别在 TTBR0_EL2 和 TTBR0_EL3 中指定，使内存底部的单一连续地址空间的大小可变。TG 字段指定颗粒大小，SL0 字段控制表查找的第一级。任何超出定义的地址范围的访问都会导致转换错误。

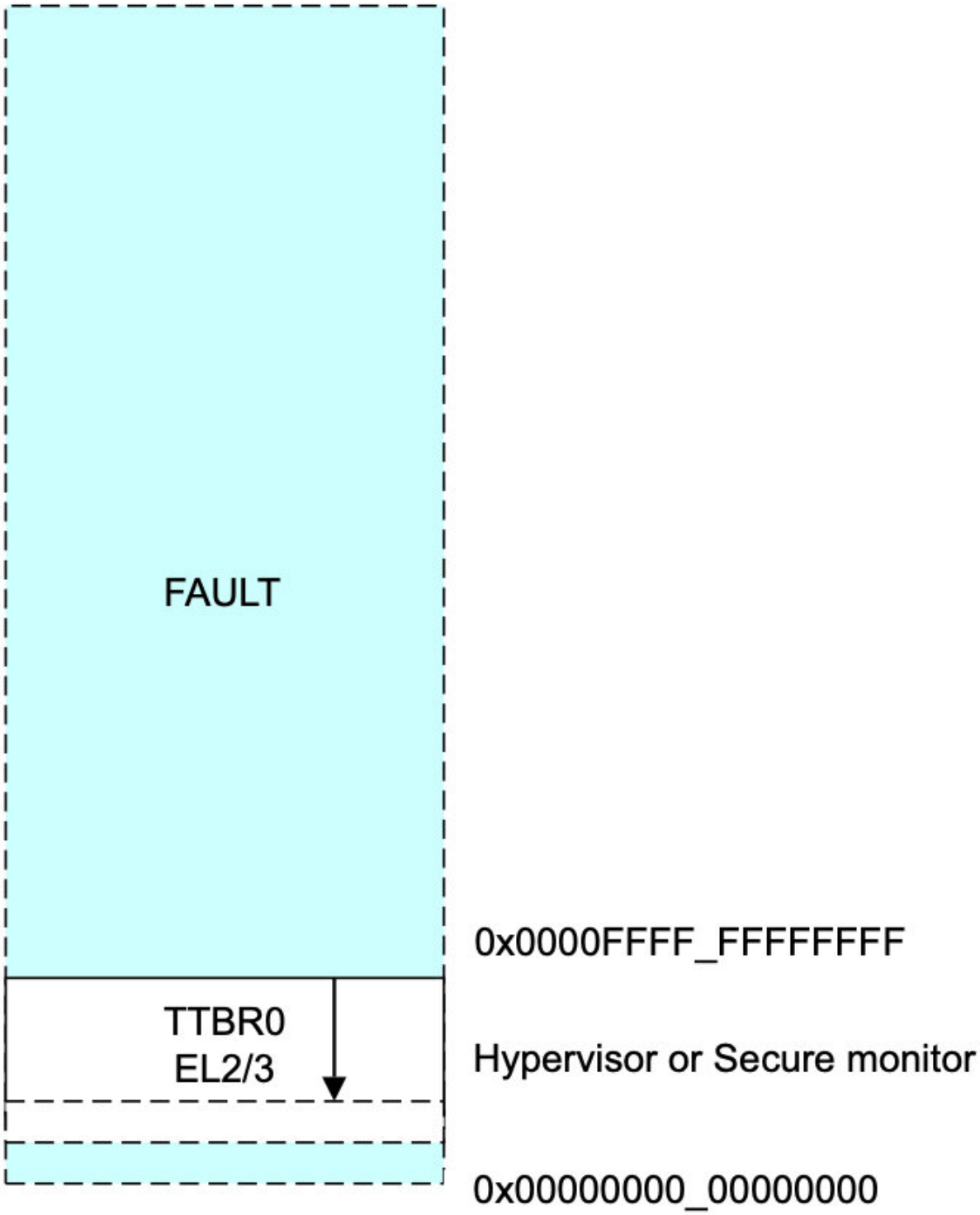

Figure 12-17 Maximum Virtual Address space image-20220421195803080

安全监控器 EL3 有自己的专用映射表。表的基址在 TTBR0_EL3 中指定，并通过 TCR_EL3 进行配置。映射表能够访问安全和非安全的物理地址。TTBR0_EL3 只在安全监控 EL3 模式下使用，而不是由可信的内核本身使用。当过渡到安全世界时，受信任的内核使用 EL1 映射，也就是由 TTBR0_EL1 和 TTBR1_EL1 指向的映射表。由于这些寄存器在 AArch64 中没有入库，安全监控代码必须为安全世界配置新的表，并保存和恢复 TTBR0_EL1 和 TTBR1_EL1 的副本。

与非安全状态下的正常操作相比，EL1 转换机制在安全状态下的行为是不同的。第二阶段的映射被禁用，EL1 翻译系统现在可以指向安全或非安全的物理地址。在安全状态下没有虚拟化，因此 IPA 总是与最终的 PA 相同。

TLB 中的条目被标记为安全或非安全，因此，当你在安全和正常世界之间转换时，不需要维护 TLB。

12.7 12.7 访问权限

访问权限是通过转换表项控制的。访问权限控制一个区域是可读还是可写，或者两者都是，可以分别设置为非特权的 EL0 和特权访问的 EL1、EL2、EL3，如表 12-4 所示。

Table 12-4 Access permissions

AP	Unprivileged (EL0)	Privileged (EL1/2/3)
00	No access	Read and write
01	Read and write	Read and write
10	No access	Read-only
11	Read-only	Read-only

image-20220421200308803

操作系统的内核运行在执行层 EL1。它定义了映射表的映射，这些映射被内核本身和运行在 EL0 的应用程序所使用。由于内核为自己的代码和应用程序指定了不同的权限，所以需要区分非特权和特权访问权限。在执行层 EL2 运行的管理程序和安全监控器 EL3 只有自己使用的翻译方案，因此没有必要在权限上进行特权和非特权的划分。

另一种访问权限是可执行属性。区块可以被标记为可执行或不可执行（从不执行（XN））。你可以分别设置从不执行特权（Unprivileged Execute Never, UXN）和从不执行特权（PXN）的属性，并以此来防止，例如，以内核权限运行的应用程序代码，或在非特权状态下试图执行内核代码。设置这些属性可以防止处理器对内存位置进行推测性指令获取，并确保推测性指令获取不会意外地访问可能被这种访问扰乱的位置，例如，先进先出（FIFO）页面替换队列。因此，设备区域必须始终被标记为永不执行。

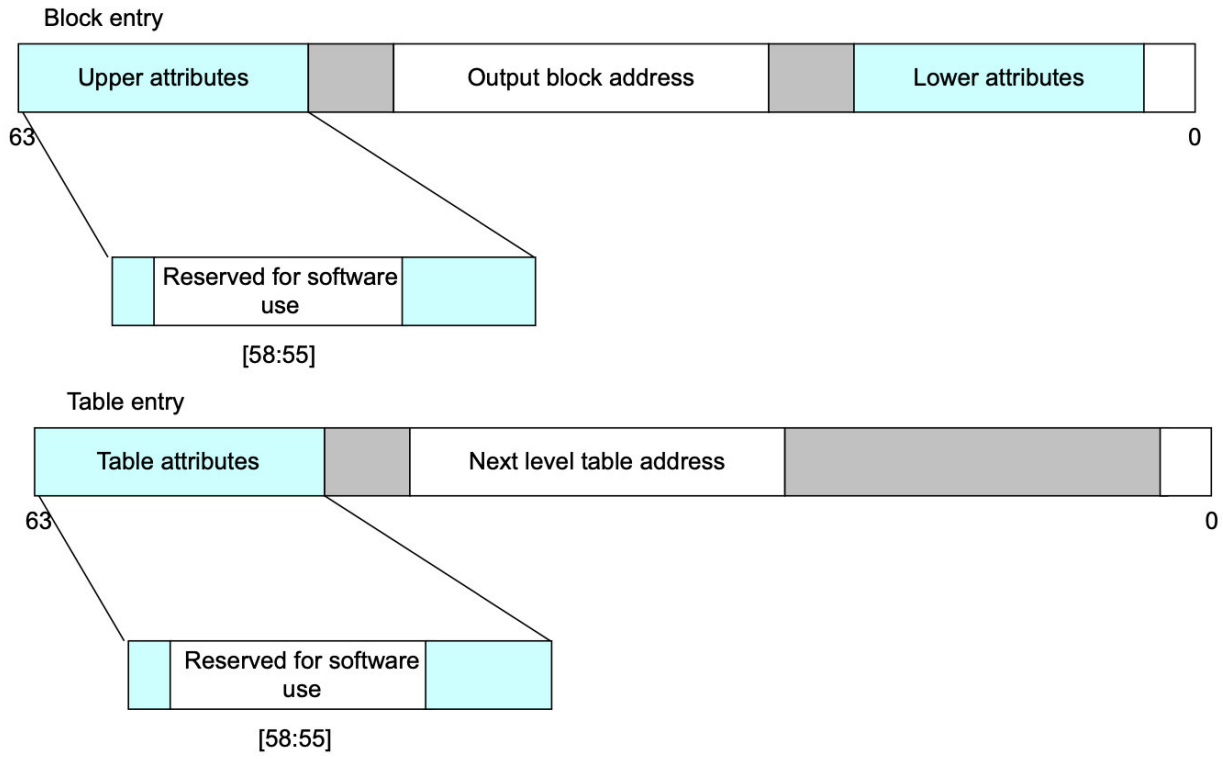

Figure 12-19 Translation table descriptors image-20220421200356230

你可以使用 SCTLR 寄存器中的以下位来配置处理器，使其将可写区域视为永不执行：

- SCTLR_EL1。WXN。在 EL0 下可写的区域在 EL0 和 EL1 处被视为 XN。在 EL1 下可写的区域在 EL1 处被视为 XN。
- SCTLR_EL2 和 3。WXN。在 ELn 可以写的区域在 ELn 被视为 XN。
- SCTLR.UWXN。在 EL0 下可写的区域在 EL1 处被视为 XN。这仅适用于 AArch32。

SCTLR_ELn 位可以被缓存在 TLB 条目中。因此，改变 SCTLR 中的位可能不会影响已经在 TLB 中的条目。当修改这些位时，需要进行 TLB 的无效化和 ISB 序列。关于 ISB 屏障的信息，请参见第 13-6 页的屏障。

12.8 12.8 操作系统对映射表描述符的使用

描述符中的另一个内存属性位，访问标志（AF），指示一个块条目何时被首次使用。

- AF = 0：此块条目尚未使用。
- AF = 1：已使用此块条目。

操作系统使用一个访问标志位来跟踪哪些页面正在被使用。软件管理着这个标志。当页面第一次被创建时，它的条目 AF 被设置为 0。当代码第一次访问该页面时，如果它的 AF 为 0，就会触发一个 MMU 故障。页面故障处理程序记录这个页面现在正在被使用，并手动设置表项中的 AF 位。例如，Linux 内核在 ARM64 上使

用 PTE_AF 的 [AF] 位（Linux 内核对 AArch64 的称呼），用于检查一个页面是否曾经被访问过。这影响了一些内核内存管理的选择。例如，当一个页面必须被换出内存时，它不太可能换出正在被使用的页面。

描述符的 [58:55] 位被标记为保留给软件使用，可以用来在转换表中记录操作系统的特定信息。例如，Linux 内核使用这些位中的一个来标记一个条目为干净或肮脏。脏状态记录了该页是否被写入。如果该页后来被换出内存，一个干净的页可以简单地被丢弃，但一个脏页必须先保存其内容。

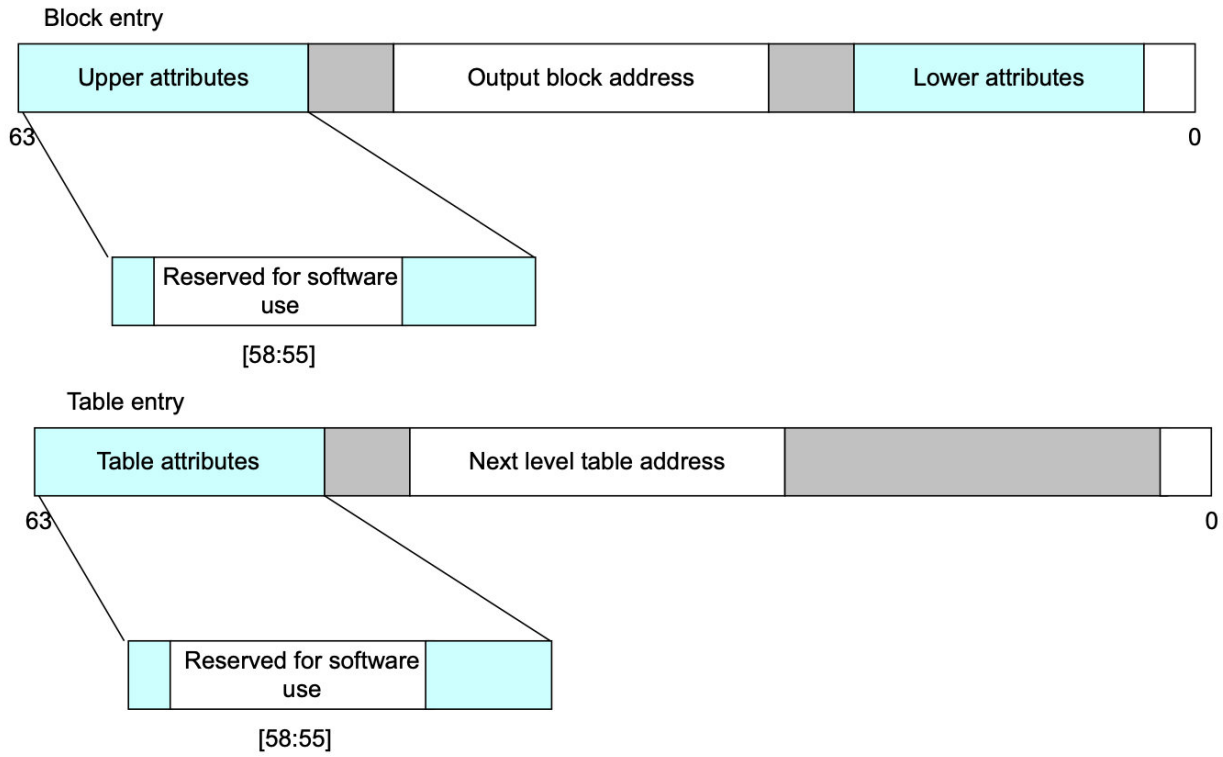

Figure 12-19 Translation table descriptors image-20220421200629384

有关指定内存类型及其可缓存性和可共享性属性的其他内存属性的信息，请参阅第 13 章内存排序。

12.9 12.9 安全和 MMU

ARMv8-A 架构定义了两种安全状态：安全和非安全。它还定义了两个物理地址空间。因此，正常世界只能访问非安全物理地址空间。安全世界可以同时访问安全和非安全物理地址空间。

在非安全状态下，映射表中的 NS 位和 NS Table 位被忽略。只有非安全的内存可以被访问。在安全状态下，NS 位和 NSTable 位控制虚拟地址是否转换为安全或非安全的物理地址。你可以使用 SCR_EL3.CIF 来防止安全世界从任何转换到非安全物理地址的虚拟地址中执行。此外，当在安全世界中，你可以使用 SCR.CIF 位来控制安全指令的获取是否可以在非安全的物理内存中进行。

12.10 12.10 内容切换

实现 ARMv8-A 架构的处理器通常用于运行复杂操作系统的系统，该系统具有许多并发运行的应用程序或任务。每个进程都有自己独特的映射表驻留在物理内存中。当应用程序启动时，操作系统为其分配一组映射表项，这映射表项将应用程序使用的代码和数据映射到物理内存。这些表随后可以被内核修改，例如，映射到额外的空间，并在应用程序不再运行时被删除。

因此，在内存系统中可能有多个任务存在。内核调度器定期地将执行从一个任务转移到另一个任务。这被称为上下文切换，要求内核保存所有与进程相关的执行状态，并恢复接下来要运行的进程的状态。内核还将映射表的条目切换到下一个要运行的进程的条目。当前没有运行的任务的内存完全被保护起来，不受正在运行的任务的影响。

确切地说，需要保存和恢复的内容在不同的操作系统中有所不同，但通常情况下，进程上下文切换包括保存或恢复以下一些或全部元素。

- 通用寄存器 X0-X30。
- 高级 SIMD 和浮点寄存器 V0-V31。
- 一些状态寄存器。
- TTBR0_EL1 和 TTBR0。
- 线程进程 ID（TPIDxxx）注册。
- 地址空间 ID (ASID)。

对于 EL0 和 EL1，有两个映射表。TTBR0_EL1 为虚拟地址空间的底部提供映射，这通常是应用程序空间，而 TTBR1_EL1 覆盖虚拟地址空间的顶部，通常是内核空间。这种分割意味着操作系统的映射不必在每个任务的映射表中进行复制。

映射表项包含一个非全局（nG）位。如果一个特定的页面的 nG 位被设置，它就与一个特定的任务或应用程序相关。如果该位被标记为 0，那么该条目是全局的，适用于所有任务。

对于非全局条目，当 TLB 被更新并且该条目被标记为非全局时，除了正常的映射信息外，还有一个值被存储在 TLB 条目中。这个值被称为地址空间 ID（ASID），它是由操作系统分配给每个单独任务的数字。后续的 TLB 查询只有在当前的 ASID 与存储在条目中的 ASID 相匹配时才会在该条目上进行匹配。这就允许在一个被标记为非全局的特定页面上存在多个有效的 TLB 条目，但其 ASID 值不同。换句话说，当我们进行上下文切换时，我们不一定需要刷新 TLB。

在 AArch64 中，这个 ASID 值可以被指定为 8 位或 16 位的值，由 TCR_EL1.AS 位控制。当前的 ASID 值被指定在 TTBR0_EL1 或 TTBR1_EL1 中。TCR_EL1 控制哪个 TTBR 持有 ASID，但通常是 TTBR0_EL1，因为它对应的是应用空间。

> 将 ASID 的当前值存储在翻译表寄存器中意味着您可以在单个指令中原子地修改翻译表和 ASID。与 ARMv7-A 架构相比，这简化了更改表和 ASID 的过程。

此外，ARMv8-A 架构提供线程 ID 寄存器供操作系统软件使用。这些寄存器没有硬件意义，通常由线程库作为每个线程数据的基本指针使用。这通常被称为线程本地存储（TLS）。例如，pthreads 库使用这一功能，包括以下寄存器：

- 用户读写线程 ID 寄存器（TPIDR_EL0）。
- 用户只读线程 ID 寄存器（TPIDRRO_EL0）。
- 线程 ID 寄存器，仅限特权访问（TPIDR_EL1）。

12.11 12.11 用户权限的内核访问

有一些指令允许在 EL1 执行的代码（例如，操作系统）以 EL0 或应用程序的权限执行内存访问。例如，这可以用来取消对系统调用提供的指针的引用，并使操作系统能够检查是否只有应用程序可访问的数据被访问。这可以通过 LDTR 或 STTR 指令来实现。当在 EL1 执行时，这些指令执行加载或存储，就像在 EL0 执行一样。在所有其他的异常级别，LDTR 和 STTR 的行为与普通的 LDR 或 STR 指令一样。有通常的大小和有符号和无符号的变体作为正常的加载和存储指令，但有一个较小的偏移量和受限制的索引选项。

CHAPTER

THIRTEEN

13. 内存排序

如果您的代码直接与硬件或在其他内核上执行的代码交互，或者如果它直接加载或写入要执行的指令，或修改页表，您需要注意内存排序问题。

如果你是应用程序开发者，硬件交互可能是通过设备驱动，与其他内核的交互是通过 Pthreads 或其他多线程 API，而与分页内存系统的交互是通过操作系统。在所有这些情况下，相关代码都会为您处理内存排序问题。但是，如果您正在编写操作系统内核或设备驱动程序，或者实现管理程序、JIT 编译器或多线程库，则必须对 ARM 体系结构的内存排序规则有一个很好的理解。您必须确保在您的代码需要显式内存访问顺序的地方，您能够通过正确使用屏障来实现这一点。

ARMv8 架构采用弱排序的内存模型。一般来说，这意味着内存访问的顺序不需要与加载和存储操作的程序顺序相同。处理器能够相对于彼此重新排序内存读取操作。写入也可以重新排序（例如，写入组合）。因此，硬件优化（例如缓存和写入缓冲区的使用）以提高处理器性能的方式起作用，这意味着所需的带宽可以减少处理器和外部存储器之间的延迟，并且隐藏与此类外部存储器访问相关的长延迟。

对普通内存的读取和写入可以由硬件重新排序，仅受数据依赖性和显式内存屏障指令的影响。某些情况需要更严格的排序规则。您可以通过描述该内存的转换表条目的内存类型属性向核心提供有关此的信息。

非常高性能的系统可能支持诸如推测性内存读取、多次发出指令或乱序执行等技术，这些技术与其他技术一起，为内存访问的硬件重新排序提供了进一步的可能性：

- 多 pipeline 发射指令 (Multiple issue of instructions) 一个处理器可能在每个周期发出并执行多条指令，以便可以同时执行按程序顺序依次执行的指令。
- 乱序执行 (Out-of-order execution) 许多处理器支持非相关指令的乱序执行。每当一条指令在等待前一条指令的结果时停止，处理器就可以执行没有依赖关系的后续指令。
- 推测当处理器遇到条件指令 (Speculation) （例如分支）时，它可以推测性地在确定是否必须执行该特定指令之前开始执行指令。因此，如果条件解决表明推测是正确的，则可以更快地获得结果。
- 推测负载 (Speculative loads) 如果推测性地执行从可缓存位置读取的加载指令，则可能导致缓存行填充和现有缓存行的潜在驱逐。
- 加载和存储优化 (Load and store optimizations) 由于对外部存储器的读取和写入可能具有较长的延迟，因此处理器可以通过例如将多个存储合并到一个更大的事务中来减少传输次数。
- 外部存储系统 (External memory systems) 在许多复杂的片上系统 (SoC) 设备中，有许多代理能够启动传输和多条路由到读取或写入的从设备。其中一些设备，例如 DRAM 控制器，可能能够同时接受来自不

同主机的请求。事务可以由互连缓冲或重新排序。这意味着来自不同主设备的访问可能因此需要不同数量的周期才能完成，并且可能会相互超越。

- 缓存一致的多核处理 (Cache coherent multi-core processing) 在多核处理器中，硬件缓存一致性可以在内核之间迁移缓存线。因此，不同的内核可能会以彼此不同的顺序看到对缓存内存位置的更新。
- 优化编译器 (Optimizing compilers) 优化编译器可以重新排序指令以隐藏延迟或充分利用硬件功能。它通常可以向前移动内存访问，使其更早，并在需要值之前给它更多时间来完成。

在单核系统中，这种重新排序的效果通常对程序员是透明的，因为单个处理器可以检查危险并确保尊重数据依赖性。但是，如果您有多个内核通过共享内存进行通信，或者以其他方式共享数据，则内存排序考虑变得更加重要。本章讨论了与多处理 (MP) 操作和多个执行线程的同步相关的几个主题。它还讨论了架构定义的内存类型和规则以及如何控制这些。

13.1 13.1 内存类型

ARMv8 架构定义了两种互斥的内存类型。所有内存区域都配置为这两种类型中的一种，即 Normal 和 Device。第三种内存类型，Strongly Ordered，是 ARMv7 架构的一部分。这种类型和设备内存之间的差异很小，因此现在在 ARMv8 中省略了。（请参阅第 13-4 页的设备内存。）

除了内存类型之外，属性还提供对可缓存性、可共享性、访问和执行权限的控制。可共享和缓存属性仅适用于普通内存。设备区域始终被视为不可缓存且可外部共享。对于可缓存位置，您可以使用属性向处理器指示缓存分配策略。

内存类型不直接编码在转换表条目中。相反，每个块条目都为内存类型表指定一个 3 位索引。该表存储在内存属性间接寄存器 MAIR_ELn 中。该表有 8 个条目，每个条目有 8 位，如图 13-1 所示。

虽然翻译表块条目本身并不直接包含内存类型编码，但处理器内部的 TLB 条目通常为特定条目存储此信息。

因此，在 ISB 指令屏障和 TLB 无效操作之后，可能不会观察到对 MAIR_ELn 的更改。

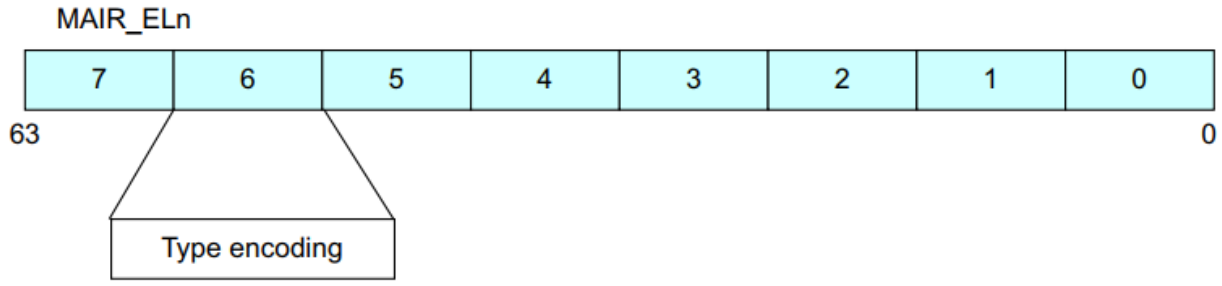

image-20220314230729157

13.1.1 13.1.1 普通内存

您可以将普通内存用于所有代码和内存中的大多数数据区域。普通内存的示例包括物理内存中的 RAM、闪存或 ROM 区域。这种内存提供了最高的处理器性能，因为它是弱排序的并且对处理器的限制较少。处理器可以重新排序、重复和合并对普通内存的访问。

此外，处理器可以推测性地访问标记为正常的地址位置，以便可以从内存中读取数据或指令，而无需在程序中显式引用，或者在显式引用的实际执行之前。这种推测性访问可能是由于分支预测、推测性高速缓存行填充、无序数据加载或其他硬件优化而发生的。

为获得最佳性能，请始终将应用程序代码和数据标记为 Normal，并且在需要强制内存排序的情况下，您可以通过使用显式屏障操作来实现。普通内存实现了弱排序的内存模式。相对于其他正常访问或设备访问，正常访问不需要按顺序完成。

但是，处理器必须始终处理由地址依赖性引起的危险。

例如，考虑以下简单的代码序列：

```
STR X0, [X2]
LDR X1, [X2]
```

处理器始终确保 X1 中的值是写入 X2 中存储的地址的值。

这当然适用于更复杂的依赖关系。考虑以下代码：

```
ADD X4, X3, #3
ADD X5, X3, #2
STR X0, [X3]
STRB W1, [X4]
LDRH W2, [X5]
```

在这种情况下，访问发生在彼此重叠的地址上。处理器必须确保内存的更新就像 STR 和 STRB 按顺序发生一样，以便 LDRH 返回最新的值。处理器将 STR 和 STRB 合并到包含要写入的最新、正确数据的单个访问中仍然有效。

13.2 13.1.2 设备内存

您可以将设备内存用于访问可能会产生副作用的所有内存区域。例如，对 FIFO 位置或定时器的读取是不可重复的，因为它为每次读取返回不同的值。对控制寄存器的写入可能会触发中断。它通常仅用于系统中的外围设备。设备内存类型对内核施加了更多限制。

推测性数据访问不能对标记为设备的内存区域执行。对此有一个不常见的例外。如果使用 NEON 操作从设备内存中读取字节，则处理器可能会读取未明确引用的字节（如果它们位于包含一个或多个明确引用的字节的对齐 16 字节块内）。

尝试从标记为设备的区域执行代码通常是不可预测的。该实现可能会像处理具有 Normal 不可缓存属性的内存位置一样处理指令提取，或者可能会出现权限错误。

有四种不同类型的设备内存，适用不同的规则。

- Device-nGnRnE 限制性最强（相当于 ARMv7 架构中的强有序内存）
- Device-nGnRE
- Device-nGRE
- Device-GRE least restrictive

The letter suffixes refer to the following three properties:

- Gathering or non Gathering (G or nG) 此属性确定是否可以将多个访问合并到此内存区域的单个总线事务中。如果地址被标记为非聚会 (nG)，那么内存总线上对该位置执行的访问次数和大小必须与代码中显式访问的次数和大小完全匹配。如果地址被标记为 Gathering (G)，则处理器可以例如将两个字节写入合并为单个半字写入。对于标记为 Gathering 的区域，还可以合并对同一内存位置的多个内存访问。例如，如果程序读取同一个位置两次，核心只需要执行一次读取，就可以返回两条指令的结果相同。对于从标记为非 Gathering 的区域读取，数据值必须来自终端设备。不能从写缓冲区或其他位置窥探它。
- Re-ordering (R or nR) 这决定了对同一设备的访问是否可以相互重新排序。如果地址被标记为非重新排序 (nR)，则同一块内的访问始终按程序顺序出现在总线上。这个块的大小是实现定义的。如果这个块的大小很大，它可以跨越几个表条目。在这种情况下，对于也标记为 nR 的任何其他访问，都遵守排序规则。
- Early Write Acknowledgement (E or nE) 这确定是否允许处理器和正被访问的从设备之间的中间写缓冲区发送写完成的确认。如果地址被标记为非早期写确认 (nE)，则写响应必须来自外设。如果地址被标记为早期写入确认 (E)，则允许互连逻辑中的缓冲区在终端设备实际接收到写入之前发出写入接受信号。这本质上是给外部存储系统的信息。

13.3 13.2 内存屏障

ARM 体系结构包括屏障指令，用于在特定点强制访问排序和访问完成。在某些架构中，类似的指令被称为栅栏。

如果您正在编写顺序很重要的代码，请参阅 ARM 体系结构参考手册 - ARMv8 中的附录 J7 Barrier LitmusTests ，了解 ARMv8-A 体系结构配置文件和 ARM 体系结构参考手册 ARMv7-A/R 版中的附录 G 障碍石蕊测试，其中包括许多工作示例。

ARM 体系结构参考手册定义了某些关键词，特别是术语遵守和必须遵守。在典型系统中，这定义了主设备的总线接口（例如内核或 GPU 和互连）必须如何处理总线事务。只有主人能够观察传输。所有总线事务均由主机发起。主设备执行事务的顺序不一定与此类事务在从设备上完成的顺序相同，因为除非明确强制执行某些排序，否则事务可能由互连重新排序。

描述可观察性的一种简单方法是说“当我可以阅读您所写的内容时，我观察了您的写作，并且当我无法再更改您阅读的值时，我已经观察了您的阅读”，我和您都指核心或其他系统中的主人。

该架构提供了三种屏障指令：

- 指令同步屏障 (ISB) 这用于保证再次获取任何后续指令，以便使用当前 MMU 配置检查特权和访问。它用于确保任何先前执行的上下文更改操作，例如写入系统控制寄存器，在 ISB 完成时已经完成。例如，在硬件方面，这可能意味着指令流水线被刷新。它的典型用途是内存管理、缓存控制和上下文切换代码，或者代码在内存中移动的地方。
- 数据存储屏障 (DMB) 这防止了跨屏障指令的数据访问指令的重新排序。该处理器在 DMB 之前执行的所有数据访问，即加载或存储，但不是指令提取，在 DMB 之后的任何数据访问之前，对指定可共享域内的所有其他主控器都是可见的。例如：

```
LDR x0, [x1] // Must be seen by the memory system before the STR below.
DMB ISHLD
ADD x2, #1 // May be executed before or after the memory system sees
LDR.
STR x3, [x4] // Must be seen by the memory system after the LDR above.
```

它还确保在执行任何后续数据访问之前已完成任何显式的先前数据或统一缓存维护操作。

```
DC CSW, x5 // Data clean by Set/way
LDR x0, [x1] // Effect of data cache clean might not be seen by this
// instruction
DMB ISH
LDR x2, [x3] // Effect of data cache clean will be seen by this instruction
```

- 数据同步屏障 (DSB) 这强制执行与数据存储器屏障相同的顺序，但具有阻止执行任何进一步指令的额外效果，而不仅仅是加载或存储，或两者兼而有之，直到同步完成。这可用于防止执行 SEV 指令，例如，该指令将向其他内核发出事件发生的信号。它一直等到此处理器发出的所有高速缓存、TLB 和分支预测器维护操作都已针对指定的可共享域完成。例如：

```
DC ISW, x5 // operation must have completed before DSB can complete
STR x0, [x1] // Access must have completed before DSB can complete
DSB ISH
ADD x2, x2, #3 // Cannot be executed until DSB completes
```

从上面的示例中可以看出，DMB 和 DSB 指令采用一个参数，该参数指定屏障操作之前或之后的访问类型，以及它适用的可共享域。

<option>	Ordered Accesses (before – after)	Shareability Domain
OSHLD	Load – Load, Load – Store	Outer shareable
OSHST	Store – Store	
OSH	Any – Any	
NSHLD	Load – Load, Load – Store	Non-shareable
NSHST	Store – Store	
NSH	Any – Any	
ISHLD	Load –Load, Load – Store	Inner shareable
ISHST	Store – Store	
ISH	Any – Any	
LD	Load –Load, Load – Store	Full system
ST	Store – Store	
SY	Any – Any	

image-20220315071201901

有序访问字段指定屏障操作的访问类别。有三个选项。

- Load - Load/Store 这意味着屏障需要在屏障之前完成所有加载，但不需要存储来完成。在程序顺序中出现在屏障之后的加载和存储都必须等待屏障完成。
- Store - Store 这意味着屏障仅影响存储访问，并且负载仍然可以围绕屏障自由重新排序。
- Any - Any 这意味着加载和存储都必须在屏障之前完成。在程序顺序中出现在屏障之后的加载和存储都必须等待屏障完成。

屏障用于防止发生不安全的优化并强制执行特定的内存排序。因此，使用不必要的屏障指令会降低软件性能。仔细考虑在特定情况下是否需要使用屏障，如果需要，使用哪种屏障才是正确的。

排序规则的一个更微妙的影响是核心的指令接口、数据接口和 MMU table walker 被视为单独的观察者。这意味着您可能需要，例如，使用 DSB 指令来确保访问一个接口保证在另一个接口上是可观察的。

如果执行数据缓存清理和无效指令，例如 DCCVAU、X0，则必须在此之后插入 DSB 指令，以确保后续页表遍历、对转换表条目的修改、指令提取或对内存中指令的更新，都可以看到新的值。

例如，考虑转换表的更新：

```
STR X0, [X1] // update a translation table entry
DSB ISHST // ensure write has completed
TLBI VAE1IS, X2 // invalidate the TLB entry for the entry that changes
DSB ISH // ensure TLB invalidation is complete
ISB // synchronize context on this processor
```

需要一个 DSB 来确保维护操作的完成，并且需要一个 ISB 来确保这些操作的效果可以通过下面的说明看到。

处理器可能随时推测性地访问标记为正常的地址。因此，在考虑是否需要屏障时，不要只考虑加载或存储指令生成的显式访问。

13.3.1 13.2.1 单项内存屏障 (One-way barriers)

AArch64 添加了具有隐式屏障语义的新加载和存储指令。这些要求按照程序顺序观察隐式屏障之前或之后的所有加载和存储。

- Load-Acquire (LDAR) 所有按程序顺序在 LDAR 之后的加载和存储，并且与目标地址的可共享域匹配，都必须在 LDAR 之后观察。
- Store-Release (STLR) 在 STLR 之前，与目标地址的可共享域匹配的所有加载和存储都必须在 STLR 之前观察。

以上的命令，也有 exclusive 版本，如 LDAXR and STLXR

与使用限定符来控制哪些可共享域看到屏障效果的数据屏障指令不同，LDAR 和 STLR 指令使用所访问地址的属性。

LDAR 指令保证 LDAR 之后的任何内存访问指令仅在加载获取之后可见。存储释放保证在存储释放变得可见之前所有早期的内存访问都是可见的，并且存储对于能够同时存储缓存数据的系统的所有部分都是可见的。

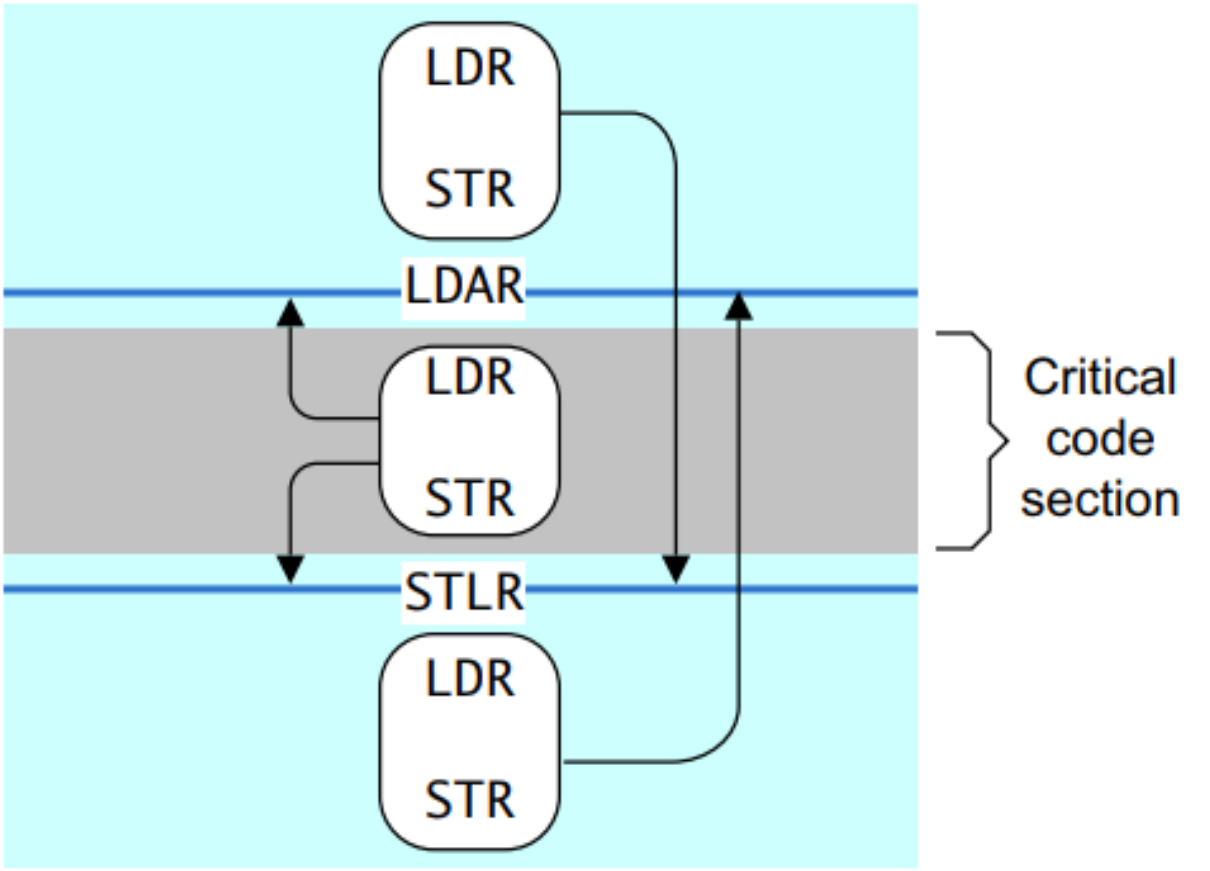

image-20220315071950470

该图显示了访问如何在一个方向上越过单向障碍，但在另一个方向上却不能

13.3.2 13.2.2 ISB 详细介绍

ARMv8 架构将上下文定义为系统寄存器的状态，将上下文更改操作定义为缓存、TLB 和分支预测器维护操作，或对系统控制寄存器的更改，例如 SCTLR_EL1、TCR_EL1 和 TTBRn_EL1。只有在上下文同步事件之后才能保证看到这种上下文改变操作的效果。

上下文同步事件分为三种：

- Taking an exception.
- Returning from an exception.
- Instruction Synchronization Barrier (ISB)

ISB 刷新流水线，并从缓存或内存中重新获取指令，并确保在 ISB 之前完成的任何上下文更改操作的效果对 ISB 之后的任何指令都是可见的。它还确保在 ISB 之后的任何上下文更改操作

指令只在 ISB 执行后生效，在 ISB 之前的指令看不到。这并不意味着在修改处理器寄存器的每条指令之后都需要 ISB 。例如，对 PSTATE 字段、ELR、SP 和 SPSR 的读取或写入以相对于其他指令的程序顺序发生。

此示例显示如何启用浮点单元和 NEON，您可以在 AArch64 中通过写入 CPACR_EL1 寄存器的位 [20] 来执行此操作。ISB 是一个上下文同步事件，可确保在执行任何后续指令或 NEON 指令之前完成启用。

```
MRS X1, CPACR_EL1
ORR X1, X1, #(0x3 << 20)
MSR CPACR_EL1, X1
ISB
```

13.3.3 13.2.3 C 语言中的内存屏障使用

C11 和 C++11 语言具有良好的独立于平台的内存模型，如果可能的话，它比内部函数更可取。

所有版本的 C 和 C++ 都有序列点，但 C11 和 C++11 也提供内存模型。序列点仅防止编译器重新排序 C++ 源代码。没有什么可以阻止处理器对生成的目标代码中的指令重新排序，也没有什么可以阻止读取和写入缓冲区重新排序将数据传输发送到缓存的顺序。换句话说，它们只与单线程代码相关。对于多线程代码，要么使用 C11 / C++11 的内存模型特性，要么使用操作系统提供的其他同步机制，例如互斥锁。通常，编译器无法跨序列点重新排列语句并限制编译器可以进行的优化。

代码中的序列点示例包括函数调用和对 volatile 变量的访问。C 语言规范对序列点的定义如下："在被称为序列点的执行序列中的某些指定点，之前评估的所有副作用都应该是完整的，并且后续评估的副作用应该没有发生。"

Linux 中的内存屏障：Linux 内核包括许多独立于平台的屏障函数。请参阅 memory-barriers.txt 文件中的 Linux 内核文档：https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/了解更多详情。

13.3.4 13.2.4 LDNP 和 STNP

ARMv8 中的一个新概念是非临时加载和存储。这些是 LDNP 和 STNP

执行读取或写入一对寄存器值的指令。它们还向内存系统提示缓存对这些数据没有用处。该提示不会禁止内存系统活动，例如地址缓存、预加载或收集，而只是表明缓存不太可能提高性能。一个典型的用例可能是流数据，但您应该注意，有效使用这些指令需要一种特定于微架构的方法。

非临时加载和存储放宽了内存排序要求。在上述情况下，可能会在前面的 LDR 指令之前观察到 LDNP 指令，这可能导致从 X0 中不可预知的地址读取。例如：

```
LDR X0, [X3]
LDNP X2, X1, [X0]
```

要纠正上述问题，您需要一个明确的负载屏障：

```
LDR X0, [X3]
DMB NSHLD
LDNP X2, X1, [X0]
```

13.4 13.3 内存属性

系统的内存映射被划分为多个区域。每个区域可能需要不同的内存属性，例如访问权限，包括针对不同特权级别、内存类型和缓存策略的读取和写入权限。代码和数据的功能片段通常在内存映射中组合在一起，并且这些区域中的每一个的属性单独控制。此功能由内存管理单元执行。转换表条目使 MMU 硬件能够将虚拟地址转换为物理地址。此外，它们指定了与每个页面相关的许多属性。

图 13-3 显示了如何在阶段 1 块条目中指定内存属性。转换表中的块条目定义了每个内存区域的属性。第 2 阶段的参赛作品有不同的布局。

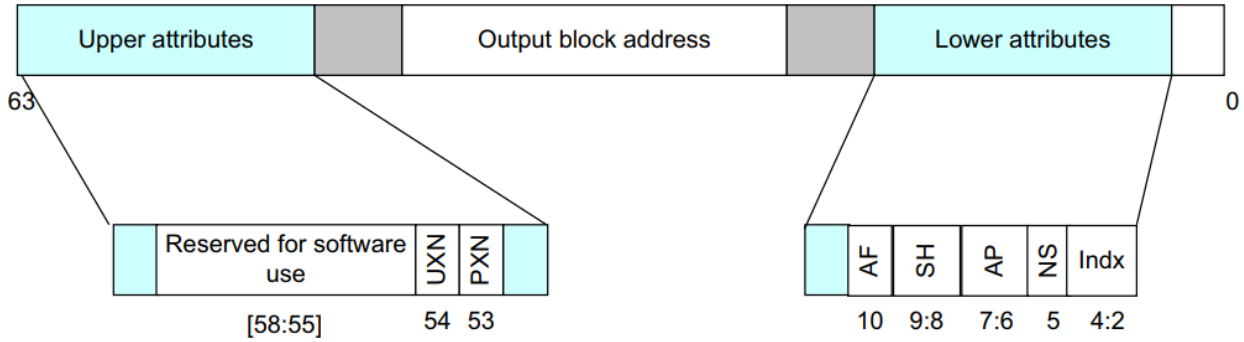

image-20220315072544728

- UXN and PXN are execution permissions
- AF is the access flag
- SH is the shareable attribute
- AP is the access permission
- NS is the security bit, but only at EL3 and Secure EL1
- Indx is the index into the Memory Attribute Indirection Register MAIR_ELn

描述符格式提供了对分层属性的支持，以便在一个级别设置的属性可以被较低级别继承。这意味着 L0、L1 或 L2 表中的表条目可以覆盖它指向的表中指定的一个或多个属性。这可用于访问权限、安全性和执行权限。例如，L1 表中具有 NSTable=1 的条目意味着它指向的 L2 和 L3 表中的 NS 位被忽略，并且所有条目都被视为具有 NS=1。此功能仅限制同一翻译阶段的后续查找级别。

13.4.1 13.3.1 可缓存和可共享的内存属性

标记为正常的内存区域可以指定为缓存或非缓存。有关可缓存内存的更多信息，请参阅第 14 章多核处理器。内存缓存可以通过内部和外部属性分别控制，用于多级缓存。内部和外部之间的划分是实现定义的，但通常是集合

内部属性的一部分由集成到处理器中的缓存使用，而外部属性从处理器导出到外部存储器总线，因此可能由内核或集群外部的缓存硬件使用。

shareable 属性用于定义一个位置是否与多个内核共享。将区域标记为不可共享意味着它仅由该内核使用，而将其标记为内部可共享或外部可共享，或两者兼而有之，意味着该位置与其他观察者共享，例如，GPU 或 DMA 设备可能是被认为是另一个观察者。同样，内部和外部之间的划分是实现定义的。这些属性的架构定义是，它们使我们能够定义观察者集，这些观察者的可共享性属性使数据或统一缓存对数据访问透明。这意味着系统提供硬件一致性管理，以便内部可共享域中的两个核心必须看到标记为内部可共享的位置的一致副本。如果系统中的处理器或其他主机不支持一致性，那么它必须将可共享区域视为不可缓存。

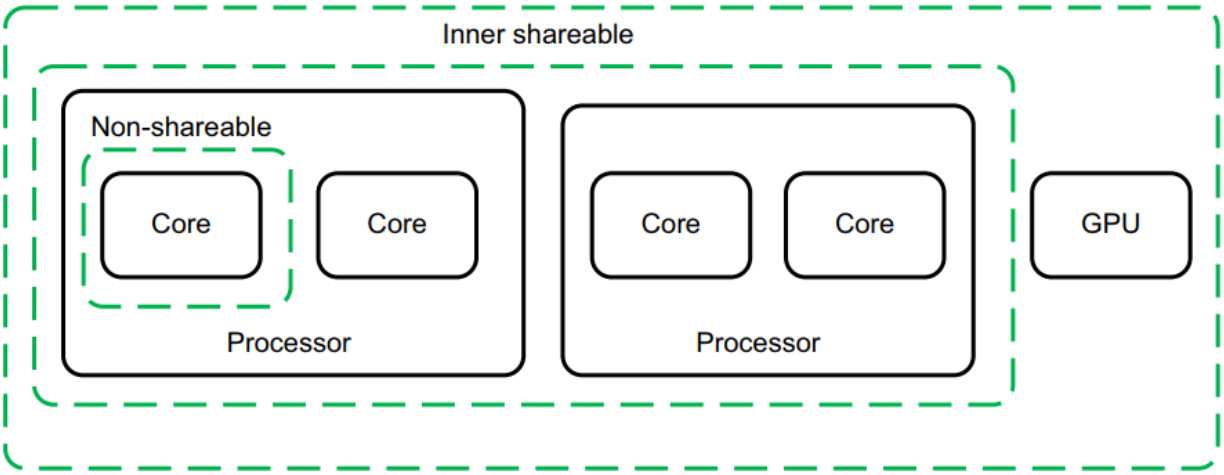

image-20220315072816685

缓存一致性硬件存在一定的开销。与其他方式相比，数据存储器访问可能需要更长的时间并消耗更多的功率。通过保持较少数量的主设备之间的一致性并确保它们在硅片中物理上靠近在一起，可以最大限度地减少这种开销。出于这个原因，该架构将系统拆分为多个域，从而可以将开销限制在需要一致性的那些位置。

系统中以下共享域的解释:

- Non-shareable 这表示内存只能由单个处理器或其他代理访问，因此内存访问永远不需要与其他处理器同步。此域通常不用于 SMP 系统。
- Inner shareable 这表示可以由多个处理器共享的可共享域，但不一定是系统中的所有代理。一个系统可能有多个内部共享域。影响一个 Inner Shareable 域的操作不会影响系统中的其他 Inner Shareable 域。此类域的一个示例可能是四核 Cortex-A57 集群
- Outer shareable 外部可共享 (OSH) 域重新排序由多个代理共享，并且可以由一个或多个内部可共享域组成。影响外部可共享域的操作也会隐式影响其内部的所有内部可共享域。但是，它不会以其他方式表现为内部可共享操作。
- Full system 对整个系统 (SY) 的操作会影响系统中的所有观察者。

CHAPTER

FOURTEEN

14. 多核处理器

ARMv8-A 架构为包含多个处理元素的系统提供了重要级别的支持。ARM 多核处理器（例如 Cortex-A57MPCore 和 Cortex-A53MPCore 处理器）可以包含一到四个内核。使用 Cortex-A57 或 Cortex-A53 处理器的系统几乎总是以这种方式实现。多核处理器可能包含多个能够独立执行指令的内核，这些内核可以被视为单个单元或集群。ARM 多核技术使集群中的四个组件内核中的任何一个都可以在不使用时关闭以节省电力，例如当设备负载较轻或处于待机模式时。当需要更高的性能时，每个处理器都在使用以满足需求，同时仍分担工作负载以保持尽可能低的功耗。

多处理可以定义为在包含两个或多个内核的单个设备中同时运行两个或多个指令序列。现在，在用于通用应用处理器的系统和更传统地定义为嵌入式系统的领域中，它都是一种广泛采用的技术。

多核系统的整体能耗可以显着低于基于单处理器内核的系统。多个内核可以使执行更快地完成，因此系统的某些元素可能会在更长的时间内完全断电。或者，具有多个内核的系统可能能够以低于单个处理器所需的频率运行以实现相同的吞吐量。较低功率的硅工艺或较低的电源电压可以降低功耗并减少能源使用。大多数当前系统不允许独立更改内核频率。但是，每个内核都可以进行动态时钟门控，从而节省更多功率和能源。

拥有多个可供我们使用的内核还可以为系统配置提供更多选择。例如，您的系统可能使用单独的内核，一个用于处理硬实时要求，另一个用于需要高性能、不间断性能的应用程序。这些可以合并到一个单一的多处理器系统中。

多核设备也可能比单核设备响应更快。当中断分布在内核之间时，有多个内核可用于响应中断，并且每个内核要服务的中断更少。多核还使重要的后台进程能够与重要但不相关的前台进程同时进行

14.1 14.1 多处理器系统

我们可以区分包含以下内容的系统：• 包含单个内核的单个处理器。• 多核处理器，例如 Cortex-A53，具有多个能够独立执行指令的内核，并且可以由系统设计人员或可以抽象底层的操作系统在外部视为单个单元或集群来自应用层的资源。• 多个集群，其中每个集群包含多个核心。

下面对多处理系统的描述定义了本书中使用的术语。在其他操作系统上，它们可能具有不同的含义。

14.1.1 14.1.1 代码在哪个内核上运行

一些软件操作取决于代码在哪个内核上运行。例如，全局初始化通常由在单个内核上运行的代码执行，然后在所有内核上进行本地初始化。

多处理器关联寄存器 (MPIDR_EL1) 使软件能够确定它正在哪个内核上执行，无论是在集群内还是在具有多个集群的系统中，它都可以确定在哪个内核上以及在哪个集群中执行。

某些处理器配置中的 U 位指示这是单核还是多核集群。亲和性字段给出了核心相对于其他核心的位置的分层描述。通常，Affinity 0 是集群内的核心 ID，而 Affinity 1 是集群 ID。

> 在 EL1 上运行的软件可能在由管理程序管理的虚拟机内运行。为了配置虚拟机，EL2 或 EL3 可以在运行时将 MPIDR_EL1 设置为不同的值，以便特定虚拟机看到每个虚拟内核的一致、唯一值。虚拟内核和物理内核之间的关系由管理程序控制，并且可能随时间而变化。
>
> MIPDR_EL3 包含每个物理内核的不可更改 ID。没有两个内核共享相同的 MPIDR_EL3 值。

14.1.2 14.1.2 对称多处理

对称多处理 (SMP) 是一种动态确定各个内核角色的软件架构。集群中的每个核心都具有相同的内存和共享硬件视图。任何应用程序、进程或任务都可以在任何内核上运行，并且操作系统调度程序可以在内核之间动态迁移任务以实现最佳系统负载。多线程应用程序可以同时在多个内核上运行。操作系统可以隐藏应用程序的大部分复杂性。

在本指南中，操作系统下应用程序的每个运行实例称为一个进程。应用程序通过调用系统库来执行许多操作，该系统库从库代码中提供某些功能，但也充当对内核操作的系统调用的包装器。各个进程具有关联的资源，包括堆栈、堆和常量数据区域，以及调度优先级设置等属性。进程的内核视图称为任务。进程是共享某些公共资源的任务的集合。其他操作系统可能有不同的定义。

在描述 SMP 操作时，我们使用术语内核来表示包含异常处理程序、设备驱动程序以及其他资源和进程管理代码的操作系统部分。我们还假设存在通常使用定时器中断调用的任务调度程序。调度程序负责在多个任务之间对内核上的可用周期进行时间切片，动态确定各个任务的优先级，并决定接下来运行哪个任务。

线程是在同一进程空间内执行的独立任务，使应用程序的不同部分能够在不同的内核上并行执行。它们还允许应用程序的一部分在另一部分等待资源时继续执行。

通常，进程中的所有线程共享多个全局资源（包括相同的内存映射以及对任何打开文件和资源句柄的访问）。线程也有自己的本地资源，包括它们自己的堆栈和寄存器使用情况，这些资源由内核在上下文切换时保存和恢复。但是，这些资源是本地的这一事实并不意味着可以保证任何线程的本地资源都不会受到其他线程的错误访问。线程是单独调度的，即使在单个进程中也可以具有不同的优先级。

支持 SMP 的操作系统为应用程序提供可用核心资源的抽象视图。多个应用程序可以在 SMP 系统中同时运行，无需重新编译或更改源代码。传统的多任务操作系统使系统能够在单核或多核处理器中同时执行多个任务或活动。在多核系统中，我们可以实现真正的并发，其中多个任务实际上在不同的核上同时并行运行。管理这些任务在可用内核上的分布的角色由操作系统执行。通常，操作系统任务调度程序可以在系统中的可用内核之间分配任务。此功能称为负载平衡，旨在获得更好的性能或节能，甚至两者兼而有之。例如，对于某些类

型的工作负载，如果将构成工作负载的任务安排在更少的内核上，则可以实现节能。这将允许更多资源闲置更长的时间，从而节省能源。

在其他情况下，如果任务分布在更多内核上，则可以提高工作负载的性能。与在更少的内核上运行相比，这些任务可以更快地向前推进，而不会相互干扰。

在另一种情况下，与在较高频率下较少内核相比，以较低频率在更多内核上运行任务可能是值得的。这样做可以在节能和性能之间提供更好的权衡。

SMP 系统中的调度程序可以动态地重新确定任务的优先级。这种动态任务优先级允许其他任务在当前任务休眠时运行。例如，在 Linux 中，性能受处理器活动限制的任务可以降低其优先级，以支持性能受 I/O 活动限制的任务。I/O-bound 进程中断计算-bound 进程，因此它可以启动其 I/O 操作然后返回睡眠，并且处理器可以在 I/O 操作完成时执行计算-bound 代码。

中断处理也可以跨内核进行负载平衡。这有助于提高性能或节省能源。跨内核平衡中断或为特定类型的中断保留内核可以减少中断延迟。这也可能导致缓存使用减少，从而有助于提高性能。

使用更少的内核来处理中断可能会导致更多的资源闲置更长的时间，从而以降低性能为代价来节省能源

14.1.3 14.1.3 定时器

支持 SMP 操作的操作系统内核通常有一个任务调度程序，它负责在多个任务之间对内核上的可用周期进行时间切片。它动态确定各个任务的优先级，并决定接下来在每个内核上运行哪个任务。通常需要一个计时器，以使每个内核上的活动任务的执行能够定期中断，从而使调度程序有机会选择不同的任务进行处理。

当所有内核竞争相同的关键资源时，可能会出现问题。每个核心运行调度程序来决定它应该执行哪个任务，并且这会以固定的时间间隔发生。内核调度程序代码需要使用一些共享数据，例如任务列表，这些数据可以通过排除（由互斥锁提供）来防止并发访问。互斥体在任何时候只允许一个核心有效地运行调度程序。

系统定时器架构描述了一个通用系统计数器，每个内核最多可提供四个定时器通道。该系统计数器应处于固定时钟频率。有安全和非安全物理计时器以及两个用于虚拟化目的的计时器。每个通道都有一个比较器，它与系统范围的 64 位计数进行比较，该计数从零开始计数。您可以配置定时器，以便在计数大于或等于编程的比较器值时产生中断。

尽管系统计时器必须具有固定频率，通常以 MHz 为单位，但允许变化的更新粒度。这意味着，您可以在每 10 或 100 个周期以相应降低的速率将计时器增加一个较大的量，例如 10 或 100，而不是在每个时钟滴答上增加计数。这给出了相同的有效频率，但更新粒度降低了。这对于实现低功耗状态很有用。CNTFRQ_EL0 寄存器报告系统定时器的频率。

一个常见的误解是 CNTFRQ_EL0 由所有内核共享。只有每个内核的寄存器，然后仅从固件的角度来看：所有其他软件应该看到该寄存器已经初始化为所有内核上的正确公共值。然而，计数器频率是全局的，并且对于所有内核都是固定的。CNTFRQ_EL0 为引导 ROM 或固件提供了一种方便的方式来告诉其他软件全局计数器频率是多少，但不控制硬件行为的任何方面。

CNTPCT_EL0 寄存器报告当前计数值。CNTKCTL_EL1 控制 EL0 是否可以访问系统定时器。

要配置计时器，请完成以下步骤：

- (1) 将比较器值写入 64 位寄存器 CNTP_CVAL_EL0。

- (2) 在 CNTP_CTL_EL0 中启用计数器和中断生成。
- (3) 轮询 CTP_CTL_EL0 以报告 EL0 定时器中断的原始状态。

您可以将系统计时器用作倒数计时器。在这种情况下，所需的计数被写入 32 位 CNTP_TVAL_EL0 寄存器。硬件会为您计算正确的 CNTP_CVAL_EL0 值。

14.1.4 14.1.4 同步

在 SMP 系统中，必须经常在任何特定时间将数据访问限制为一个修饰符。这对于外围设备是正确的，对于由多个线程或进程访问的全局变量和数据结构也是如此。保护此类共享资源通常是通过一种称为互斥的方法。在多核系统中，您可以使用自旋锁（实际上是具有原子不可分割机制的共享标志）来测试和设置其值。

ARM 体系结构提供了三个与独占访问相关的指令，以及这些指令的变体，它们对字节、半字、字或双字大小的数据进行操作。

这些指令依赖于内核或内存系统标记特定地址以供该内核使用独占访问监视器进行独占访问监视的能力。这些指令的使用在多核系统中很常见，但也出现在单核系统中，以实现在同一核上运行的线程之间的同步操作。

A64 指令集有实现此类同步功能的指令：

- 负载独占 (LDXR)：LDXR W|Xt, [Xn]
- Store Exclusive (STXR): STXR Ws, W|Xt, [Xn] 其中 Ws 表示存储是否成功完成。0 = 成功。
- 清除独占访问监视器（CLREX）这用于清除本地独占监视器的状态。

LDXR 执行内存加载，但也标记要监视的物理地址以供该内核独占访问。STXR 对内存执行条件存储，仅当目标位置被标记为被该核心监控以进行独占访问时才会成功。如果存储不成功，则该指令在通用寄存器 Ws 中返回非零值，如果存储成功，则返回值 0。在汇编语法中，它总是被指定为 W 寄存器，即，不是 X 寄存器。此外，STXR 清除了独占标签。

加载独占和存储独占操作仅保证对映射有以下所有属性的普通内存起作用：

- Inner or Outer Shareable.
- Inner Write-Back.
- Outer Write-Back.
- Read and Write allocate hints.
- Not transient.

互斥锁或自旋锁可用于控制对外围设备的访问。锁定位置将在普通 RAM 中。您不使用加载或存储独占来访问外围设备本身。

每个内核只能标记一个地址。独占监视器不会阻止另一个内核或线程读取或写入被监视的位置，而只是监视该位置自 LDXR 以来是否已被写入。

尽管体系结构和硬件支持独占访问的实现，但它们依赖于程序员强制执行正确的软件行为。互斥体只是一个标志，独占访问机制使该标志能够以原子方式访问。任何访问该标志的线程或程序都可以知道它设置正确。

但是，互斥锁控制的实际资源仍然可以由行为不正确的软件直接访问。同样，用于存储互斥体的内存也没有特殊属性。当独占访问序列完成时，它只是内存中的另一条数据。此外，在编写使用互斥锁进行资源保护的代码时，了解弱序内存模型至关重要。例如，如果没有正确使用屏障和其他内存排序注意事项，推测可能意味着在授予互斥锁之前已加载数据，或者在更新关键资源之前释放互斥锁。

14.1.5 14.1.5 非对称多处理

非对称多处理 (AMP) 系统使您能够将各个角色静态分配给集群内的一个核心，这样实际上您就拥有单独的核心，每个核心在每个集群内执行单独的作业。这称为功能分发软件架构，通常意味着您在各个内核上运行单独的操作系统。该系统在您看来是一个带有用于某些关键系统服务的专用加速器的单核系统。AMP 并不是指任务或中断与特定内核相关联的系统。

在 AMP 系统中，每个任务都可以有不同的内存视图，并且高负载的内核无法将工作传递给负载较轻的内核。在此类系统中不需要硬件缓存一致性，尽管通常存在通过共享资源在内核之间进行通信的机制，可能需要专用硬件。第 14-10 页的缓存一致性中描述的系统可以帮助减少与系统之间共享数据相关的开销。

使用多核处理器实现 AMP 系统的原因可能包括安全性、保证满足实时期限的要求，或者因为单个内核专用于执行特定任务。

有几类系统同时具有 SMP 和 AMP 功能。这意味着有两个或更多内核运行 SMP 操作系统，但系统具有不作为 SMP 系统的一部分运行的其他元素。SMP 子系统可以看作是 AMP 系统中的一个元素。缓存一致性在 SMP 核心之间实现，但不一定在 SMP 核心和系统内的 AMP 元素之间实现。这样，可以在同一个集群内实现独立的子系统。

完全有可能（并且正常）构建单个内核运行不同操作系统（这些称为多操作系统系统）的 AMP 系统

> 在这些单独的内核之间需要同步的地方，可以通过消息传递通信协议来提供，例如多核通信协会 API (MCAPI)。这些可以通过使用共享内存来传递数据包和使用软件触发的中断来实现所谓的门铃机制来实现。

14.1.6 14.1.6 异构多处理

术语异构多处理 (HMP) 在许多不同的环境中都有应用。它通常与 AMP 混为一谈，用来描述由不同类型的处理器组成的系统，例如多核 ARM 应用处理器和专用处理器（例如基带控制器芯片或音频编解码器芯片）。

ARM 使用 HMP 来表示由应用处理器集群组成的系统，这些应用处理器在指令集架构上 100% 相同，但在微架构上却大不相同。所有处理器都是完全缓存一致的，并且是同一一致域的一部分。

最好使用称为 big.LITTLE 的 HMP 技术的 ARM 实现来解释这一点。在 big.LITTLE 系统中，节能的 LITTLE 内核与高性能的大内核相干耦合，形成一个系统，可以以最节能的方式完成高强度和低强度任务。

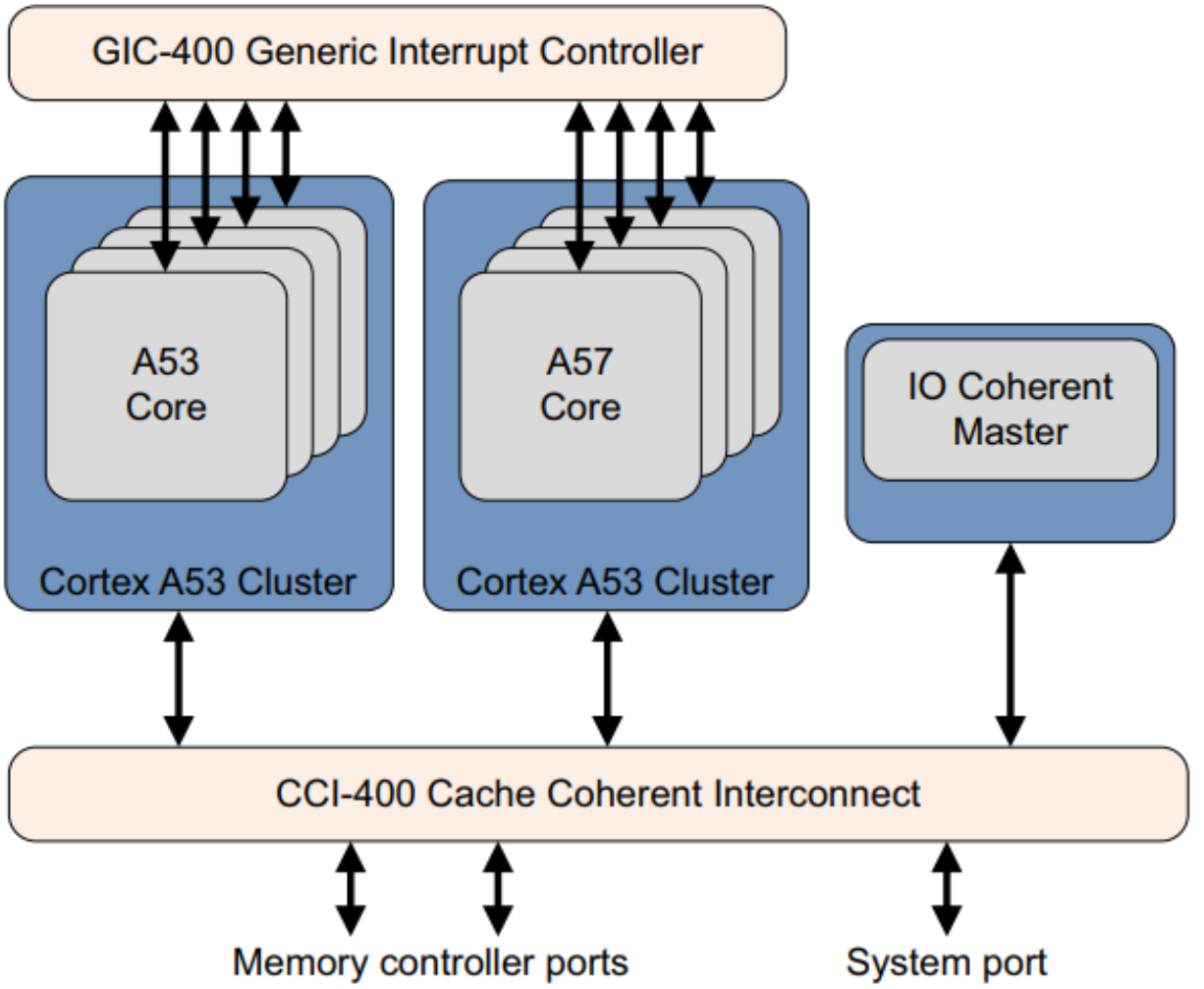

image-20220430190452892

big.LITTLE 的核心原则是应用软件无需修改即可在任何一种处理器上运行

14.1.7 14.1.7 独占监视器

典型的多核系统可能包括多个独占监视器。每个核心都有自己的本地监视器，并且有一个或多个全局监视器。与用于独占加载或存储指令的位置相关的转换表条目的可共享和可缓存属性确定使用哪个独占监视器。

在硬件中，核心包括一个名为本地监视器的设备。该监视器观察核心。当核心执行独占加载访问时，它会在本地监视器中记录该事实。当它执行独占存储时，它会检查是否执行了先前的独占加载，如果不是，则使独占存储失败。该体系结构使各个实现能够确定监视器执行的检查级别。内核一次只能标记一个物理地址。

每次异常返回时，即执行 ERET 指令时，本地独占监视器都会被清除。在 Linux 内核中，多个任务在 EL1 的内核上下文中运行，并且可以在没有异常返回的情况下进行上下文切换。只有当我们在其关联的内核任务的上下文中返回用户空间线程时，我们才会执行异常返回。这与 ARMv7 架构不同，其中内核任务调度程序必须明确清除每个任务切换上的独占访问监视器。本地独占监视器的重置是否也会重置全局独占监视器，这是由实现定义的。

当用于独占访问的位置标记为不可共享时使用本地监视器，即仅在同一内核上运行的线程。本地监视器还可以处理将访问标记为内部可共享的情况，例如，互斥锁保护在可共享域内的任何核心上运行的 SMP 线程之间共享的资源。对于在不同的、非一致的内核上运行的线程，互斥锁位置被标记为正常、不可缓存，并且需要系统中的全局访问监视器。

系统可能不包括全局监视器，或者全局监视器可能仅适用于某些地址区域。如果对系统中不存在合适的监视器的位置执行独占访问，会发生什么情况，这是由实现定义的。以下是一些允许的选项：

- 该指令生成外部中止。
- 该指令产生一个 MMU 故障。
- 该指令被视为 NOP。
- 独占指令被视为标准 LDR/STR 指令，存储独占指令的结果寄存器中保存的值变为 UNKNOWN。

Exclusives Reservation Granule (ERG) 是独占监视器的粒度。它的大小是实现定义的，但通常是一个高速缓存行。它为监视器提供了地址之间的最小间距以区分它们。在单个 ERG 中放置两个互斥锁可能会导致误报，其中对任一互斥锁执行 STXR 指令会清除两者的独占标签。这不会阻止架构正确的软件正常运行，但它可能效率较低。可以从缓存类型寄存器 CTR_EL0 中读取特定内核上独占监视器的 ERG 大小。

14.2 14.2 缓存一致性

前面章节的缓存仅考虑单个处理器中缓存的影响。Cortex-A53 和 Cortex-A57 处理器支持集群中不同内核之间的一致性管理。这需要用正确的可共享属性标记地址区域。这些处理器允许构建包含多核集群的系统，其中可以保持集群之间共享数据的一致性。这种系统级一致性需要高速缓存一致性互连，例如实现 AMBA 4 ACE 总线规范的 ARM CCI-400。请参下图.

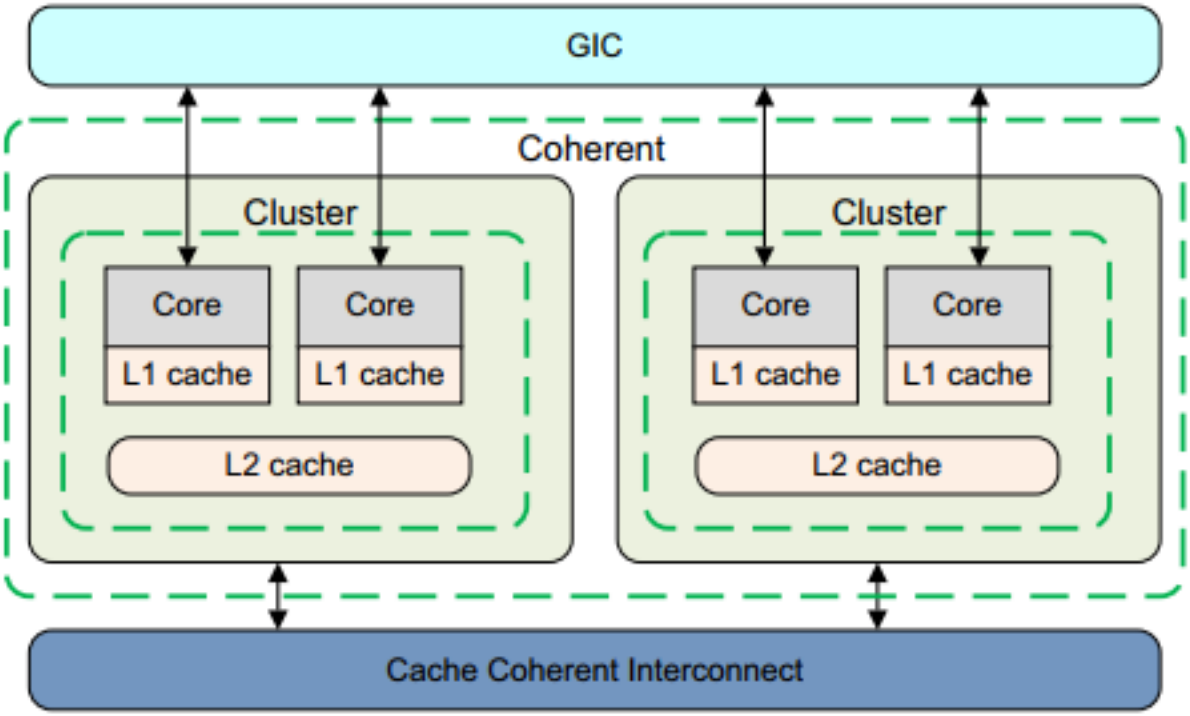

image-20220430190908866

系统中的一致性支持取决于硬件设计决策，并且存在许多可能的配置。例如，一致性只能在单个集群内得到支持。双集群 big.LITTLE 系统是可能的，其中内部域包括两个集群的核心，或者内部域包括集群而外部域包括其他集群的多集群系统。有关 big.LITTLE 系统的更多信息，请参阅第 16 章 big.LITTLE 技术。

除了维护缓存之间数据一致性的硬件之外，您还必须能够将运行在一个内核上的代码执行的缓存维护活动广播到系统的其他部分。有硬件配置信号，在复位时采样，控制是否广播内部或外部缓存维护操作，以及是否广播系统屏障指令。AMBA 4 ACE 协议允许向其他主设备发送障碍信号，从而维持维护和一致性操作的顺序。互连逻辑可能需要通过引导代码进行初始化。

软件必须通过创建适当的转换表条目来定义哪个地址区域将由哪个主控组使用，即哪些其他主控共享该地址。对于普通可缓存区域，这意味着将可共享属性设置为不可共享、内部可共享或外部可共享之一。对于不可缓存的区域，shareable 属性被忽略。

在多核系统中，不可能知道特定核是否有一条线覆盖其缓存之一中的特定地址（尤其是在互连具有缓存的情况下，例如 CCN-50x）。

维护可能需要广播到互连。这意味着一个内核上的软件可以向当前可能存储在保存该地址的不同内核的数据缓存中的地址发出缓存清理或无效操作。当如下图所示广播维护操作时，该操作由特定共享域中的所有核执行。

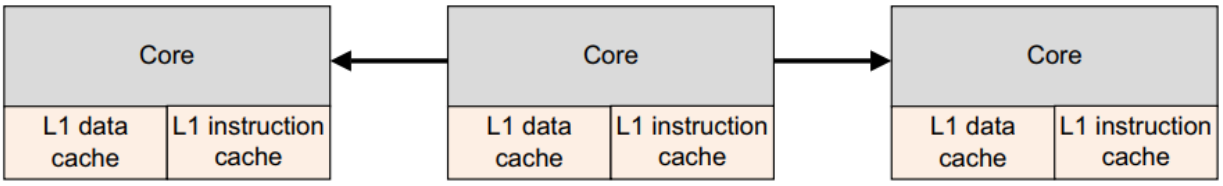

image-

20220430191251643

SMP 操作系统通常依赖于能够广播缓存和 TLB 维护操作。考虑外部 DMA 引擎能够修改外部存储器内容的情况。

运行在特定内核上的 SMP 操作系统不知道哪个内核拥有哪些数据。它只需要在集群中的任何位置使地址范围无效。如果操作未广播，则操作系统必须在每个内核上本地发出清理或无效操作。DSB 屏障指令使核心等待它发出的广播操作完成。屏障不会强制广播接收到的操作完成。有关屏障指令的更多信息，请参阅第 13 章内存排序。

下表缓存维护操作以及是否广播：

Instructions	Description	Broadcast?
IC IALLUIS	I-cache invalidate all to Point of Unification, Inner Shareable	Yes (inner only)
IC IALLU	I-cache invalidate all to Point of Unification	No[a]
IC IVAU, Xt	I-cache invalidate by address to Point of Unification	Maybe[b]
DC ZVA, Xt	D-cache zero by address	No
DC IVAC, Xt	D-cache invalidate by address to Point of Coherency	Yes
DC ISW, Xt	D-cache invalidate by Set/Way	No
DC CVAC, Xt	D-cache clean by address to Point of Coherency	Maybe[b]
DC CSW, Xt	D-cache clean by Set/Way	No
DC CVAU, Xt	D-cache clean by address to Point of Unification	Maybe[b]
DC CIVAC, Xt	D-cache clean and invalidate by address to Point of Coherency	Yes
DC CISW, Xt	D-cache clean and invalidate by Set/Way	No

image-20220430191319455

- a 如果设置了 HCR/HCR_EL2 FB 位，则在非安全 EL1 中广播，覆盖正常行为。当从 Non-secure 执行时，该位会导致以下指令在内部共享域内广播 EL1: TLBI VMALLE1, TLBI VAE1, TLBI ASIDE1, TLBI VAAE1, TLBI VALE1, TLBI VAALE1, IC IALLU
- b 广播由内存区域的 shareability 属性决定

对于 IC 指令，即指令缓存维护操作，IS 表示该功能适用于 Inner Shareable 域内的所有指令缓存。

14.3 14.3 集群内的多核缓存一致性

一致性意味着确保系统内的所有处理器或总线主控器具有相同的共享内存视图。这意味着对保存在一个内核缓存中的数据的更改对其他内核可见，从而使内核无法看到过时或旧的数据副本。这可以通过简单地不缓存来处理，即禁用共享内存位置的缓存，但这通常具有很高的性能成本。

软件管理的一致性

软件管理的一致性是处理数据共享的一种更常见的方式。数据被缓存，但软件（通常是设备驱动程序）必须清除脏数据或使缓存中的旧数据无效。这需要时间，增加软件复杂性，并且在共享率很高时会降低性能

硬件管理的一致性

硬件保持集群内 1 级数据缓存之间的一致性。内核在上电时自动参与一致性方案，启用其 D-cache 和 MMU，并且地址被标记为一致性。然而，这种高速缓存一致性逻辑并不保持数据和指令高速缓存之间的一致性。

在 ARMv8-A 架构和相关实现中，可能存在硬件管理的一致性方案。这些确保在硬件一致系统中标记为可共享的任何数据都具有该可共享域中的所有内核和总线主控器看到的相同值。这为互连和集群增加了一些硬件复杂性，但极大地简化了软件并启用了仅使用软件一致性无法实现的应用程序。

缓存一致性方案可以通过多种标准方式运行。ARMv8 处理器使用 MOESI 协议。ARMv8 处理器也可以连接到 AMBA 5 CHI 互连，其缓存一致性协议类似于（但不完全相同）MOESI。

根据使用的协议，SCU 用以下属性之一标记缓存中的每一行：M（修改）、O（拥有）、E（独占）、S（共享）或 I（无效）。这些描述如下：

Modified 缓存行的最新版本在此缓存中。其他缓存中不存在内存位置的其他副本。高速缓存行的内容不再与主存一致。

**Owned ** 这描述了一条脏的并且可能在多个缓存中的行。拥有状态的高速缓存行保存最新的正确数据副本。只有一个核心可以保存拥有状态的数据。其他核心可以将数据保持在共享状态。

Exclusive 高速缓存行存在于该高速缓存中并且与主存储器一致。其他缓存中不存在内存位置的其他副本。

Shared 高速缓存行存在于此高速缓存中，并且不一定与内存保持一致，因为 Owned 的定义允许将脏行复制到共享行中。但是，它将具有最新版本的数据。它的副本也可以存在于一致性方案中的其他缓存中。

Invalid 缓存行无效。

以下规则适用于协议的标准实现：

- 只有在高速缓存行处于修改或独占状态时才能执行写入。如果处于 Shared 状态，则必须首先使所有其他缓存的副本失效。写入将行移动到修改状态。
- 缓存可以随时丢弃共享行，将其更改为无效状态。修改后的行首先被写回。
- 如果一个高速缓存保持一行处于修改状态，则从系统中的其他高速缓存读取从高速缓存接收更新的数据。传统上，这是通过首先将数据写入主存储器，然后在执行读取之前将高速缓存行更改为共享状态来实现的。
- 当另一个缓存读取该行时，具有处于独占状态的行的缓存必须将该行移动到共享状态。

- 共享状态可能不准确。如果一个缓存丢弃了共享行，另一个缓存可能不知道它现在可以将该行移动到独占状态。

处理器集群包含一个窥探控制单元 (SCU)，其中包含存储在各个 L1 数据缓存中的标签的重复副本。因此缓存一致性逻辑：

- 保持 L1 数据缓存之间的一致性。
- 仲裁对 L2 接口的访问，包括指令和数据。
- 具有重复的标记 RAM 以跟踪每个内核数据中分配的数据。

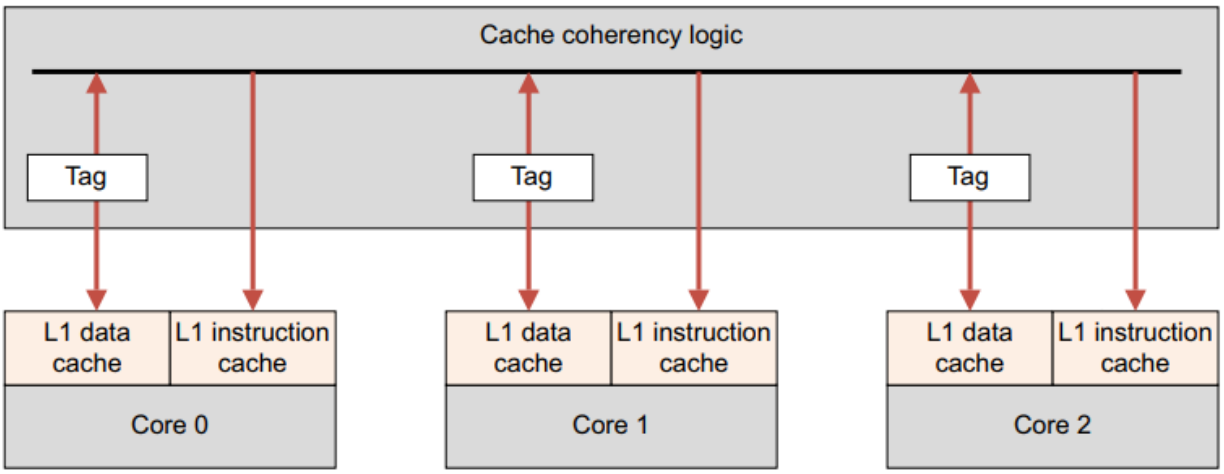

image-20220501085310374

图中的每个内核都有自己的数据和指令缓存。缓存一致性逻辑包含来自 D 缓存的标签的本地副本。但是，指令缓存不参与一致性。数据缓存和一致性逻辑之间有 2 路通信。ARM 多核处理器还实现了优化，可以直接在参与的 L1 缓存之间复制干净数据和移动脏数据，而无需访问和等待外部存储器。此活动由 SCU 在多核系统中处理。

14.3.1 14.3.1 探听控制单元 (Snoop Control Unit)

Snoop Control Unit (SCU) 维护每个内核的 L1 数据缓存之间的一致性，并负责管理以下互连操作：

- Arbitration.
- Communication.
- Cache-2-cache and system memory transfers.

该处理器还将这些功能提供给其他系统加速器和非缓存 DMA 驱动的外设，以提高性能并降低系统范围的功耗。这种系统一致性还降低了在每个操作系统驱动程序内维护软件一致性时所涉及的软件复杂性。

每个核心都可以单独配置为参与或不参与数据缓存一致性管理方案。处理器内部的 SCU 设备自动维护集群内内核之间的 1 级数据缓存一致性。有关详细信息，请参阅第 14-10 页的缓存一致性和第 14-13 页的集群内的多核缓存一致性。

由于可执行代码的更改频率要低得多，因此此功能不会扩展到 L1 指令缓存。一致性管理是使用基于 MOESI 的协议实现的，经过优化以减少外部存储器访问的数量。为了使一致性管理对内存访问有效，以下所有条件都必须为真：

- SCU 通过位于私有内存区域的控制寄存器启用。SCU 具有可配置的访问控制，限制哪些处理器可以对其进行配置。
- MMU 已启用。
- 被访问的页面被标记为 Normal Shareable，缓存策略为 write-back, write-allocate。然而，设备和强排序内存不可缓存，从内核的角度来看，直写缓存的行为类似于未缓存的内存。

SCU 只能在单个集群内保持一致性。如果系统中有额外的处理器或其他总线主控器，当它们与 MP 块共享内存时，需要明确的软件同步

14.3.2 14.3.2 加速器一致性端口 (ACP：Accelerator coherency port)

SCU 上的此 AMBA 4 AXI 兼容从接口为直接与 ARMv8 处理器接口的主接口提供互连点：

- 该接口支持所有标准读写事务，无需额外的一致性要求。但是，任何对内存连贯区域的读取事务都会与 SCU 交互，以测试信息是否已存储在 L1 缓存中。
- SCU 在将写入转发到内存系统之前强制执行写入一致性，并可能分配到 L2 缓存中，从而消除直接写入片外内存的功率和性能影响。

14.3.3 14.3.3 集群之间的缓存一致性

集群内的多核缓存一致性显示了硬件如何保持在同一集群中的多个处理器缓存之间共享的数据之间的一致性。系统还可以包含硬件，通过处理可共享的数据事务和广播屏障和维护操作来保持集群之间的一致性。可以动态地从一致性管理中添加或删除集群，例如，当整个集群（包括 L2 缓存）断电时。操作系统可以通过内置的性能监控单元 (PMU) 监控一致互连上的活动

14.3.4 14.3.4 Domains

在 ARMv8-A 体系结构中，术语域用于指代一组主总线主控器。域确定哪些主服务器被窥探，以进行连贯事务。窥探是检查主缓存以查看请求的位置是否存储在那里。有四种定义的域类型：

- Non-shareable.
- Inner Shareable.
- Outer Shareable.
- System.

典型的系统使用是运行在同一个操作系统下的主服务器在同一个 Inner Shareable 域中。共享可缓存数据但耦合度不高的主节点位于同一个外部可共享域中。同一个内域的 master 也必须在同一个外域。通过页表中的条目控制内存访问的域选择

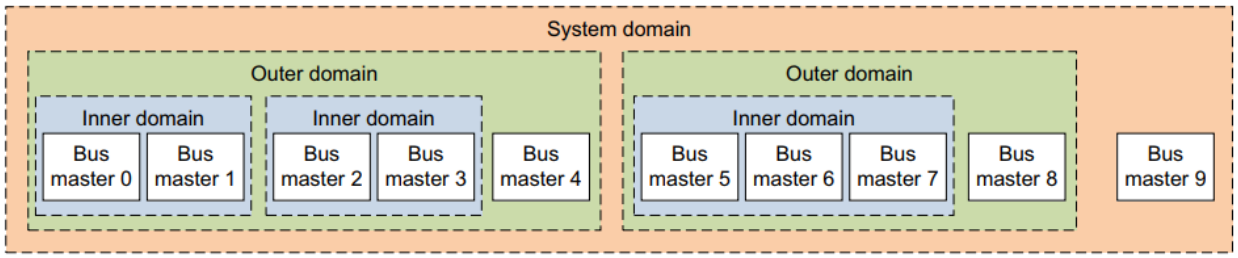

image-20220501090031179

14.4 14.4 总线协议和缓存一致性互连

将硬件一致性扩展到多集群系统需要一致的总线协议。AMBA 4 ACE 规范包括 AXI 一致性扩展 (ACE)。完整的 ACE 接口可实现集群之间的硬件一致性，并使 SMP 操作系统能够在多个内核上运行。

如果您有多个集群，则一个集群中对内存的任何共享访问都可以窥探其他集群的缓存，以查看数据是否存在，或者是否必须从外部内存加载。AMBA 4 ACE-Lite 接口是完整接口的子集，专为 DMA 引擎、网络接口和 GPU 等单向 IO 相干系统主机而设计。

这些设备可能没有自己的缓存，但可以从 ACE 处理器读取共享数据。非核心主机的缓存通常不会与核心缓存保持一致。例如，在许多系统中，核心无法窥探从端口上 GPU 的缓存内部。但反过来并不总是正确的。

ACE-Lite 允许其他主节点窥探其他集群的缓存。这意味着对于可共享的位置，必要时从一致的缓存中完成读取，并且可共享的写入与从一致的缓存行中强制清除和无效合并。ACE 规范允许将 TLB 和 I-Cache 维护操作广播到所有能够接收它们的设备。数据屏障被发送到从接口以确保它们以编程方式完成。

CoreLink CCI-400 Cache Coherent Interface 是 AMBA 4 ACE 的首批实现之一，支持多达两个 ACE 集群，使多达 8 个内核能够看到相同的内存视图并运行 SMP 操作系统，例如 big. 小组合，例如 Cortex-A57 处理器和 Cortex-A53 处理器，如图所示。

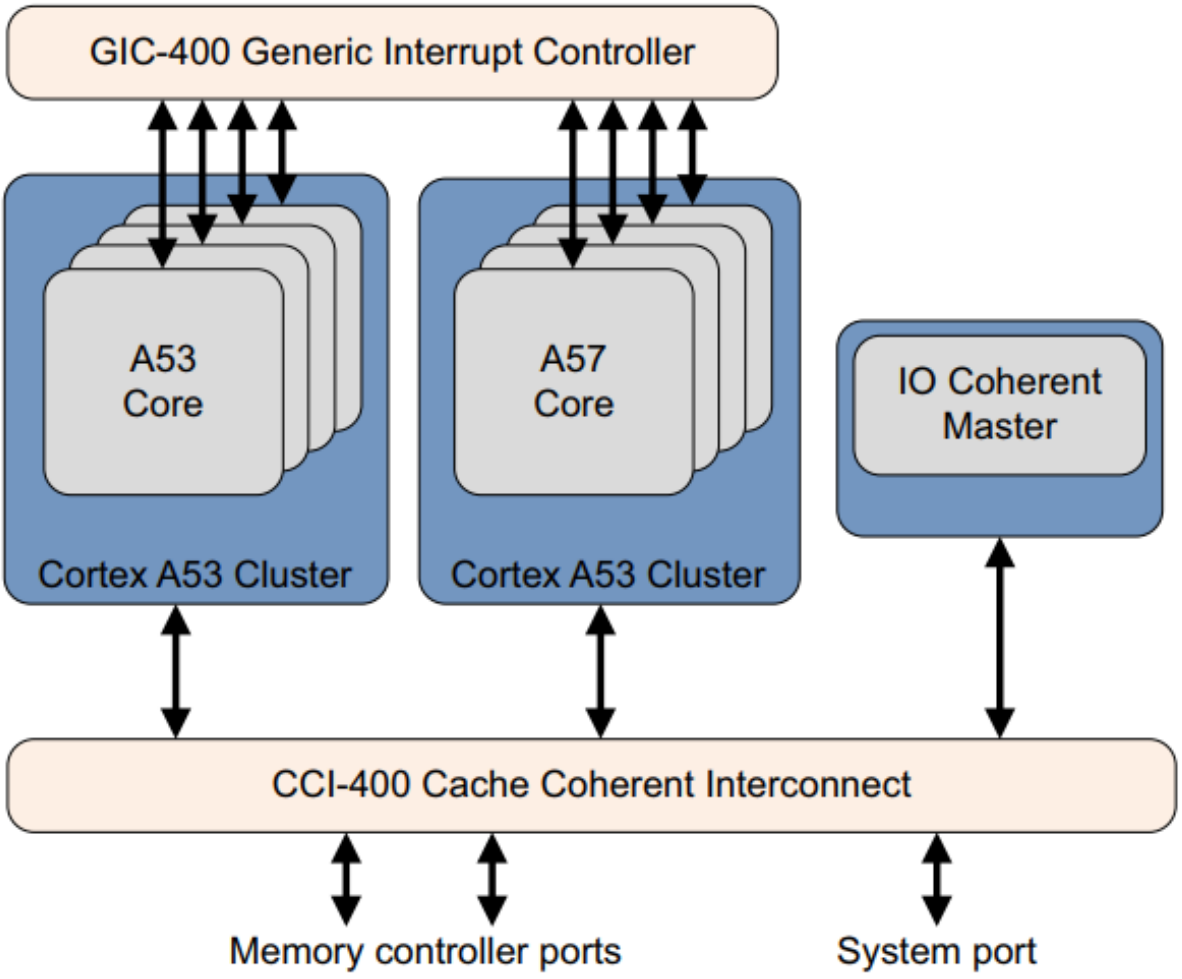

image-20220430191847275

它还具有三个 ACE-lite 相干接口，可供例如 DMA 控制器或 GPU 使用。

下图显示了从 Cortex-A53 集群读取到 Cortex-A57 集群的一致数据：

- (1)。Cortex-A53 集群发出一个一致的读取请求。
- (2)。CCI-400 将请求传递给 Cortex-A53 处理器以窥探 Cortex-A57 集群缓存。
- (3)。收到请求后，Cortex-A57 集群会检查其数据缓存的可用性并以所需信息进行响应。
- (4)。如果请求的数据在缓存中，CCI-400 将数据从 Cortex-A57 集群移动到 Cortex-A53 集群，从而导致 Cortex-A53 集群中的缓存行填充。

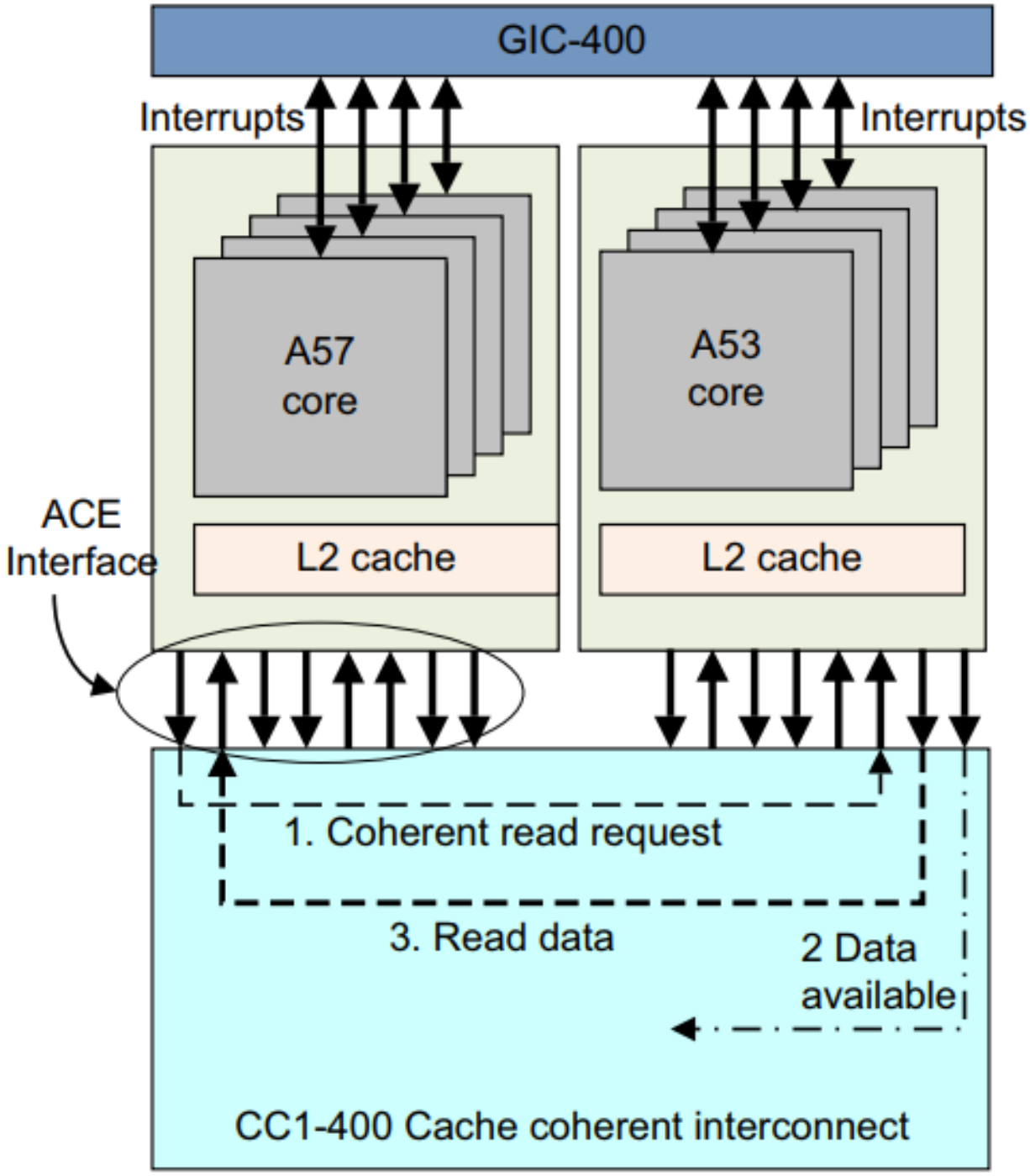

image-20220430192046860

CCI-400 和 ACE 协议可实现 Cortex-A57 和 Cortex-A53 集群之间的完全一致性，从而无需外部存储器事务即可实现数据共享。

ARM CoreLink 互连和内存控制器系统 IP 解决了在 Cortex-A 系列处理器、高性能媒体处理器和动态内存之间高效移动和存储数据的关键挑战，以优化系统级芯片的系统性能和功耗（SoC）。CoreLink 系统 IP 使 SoC 设计人员能够最大限度地利用系统内存带宽并减少静态和动态延迟。

14.4.1 14.4.1 计算子系统和移动应用程序

下图显示了具有 Cortex-A57 和 Cortex-A53 系列处理器、CoreLink MMU-500 系统 MMU 和一系列 CoreLink 400 系统 IP 的移动应用处理器示例。

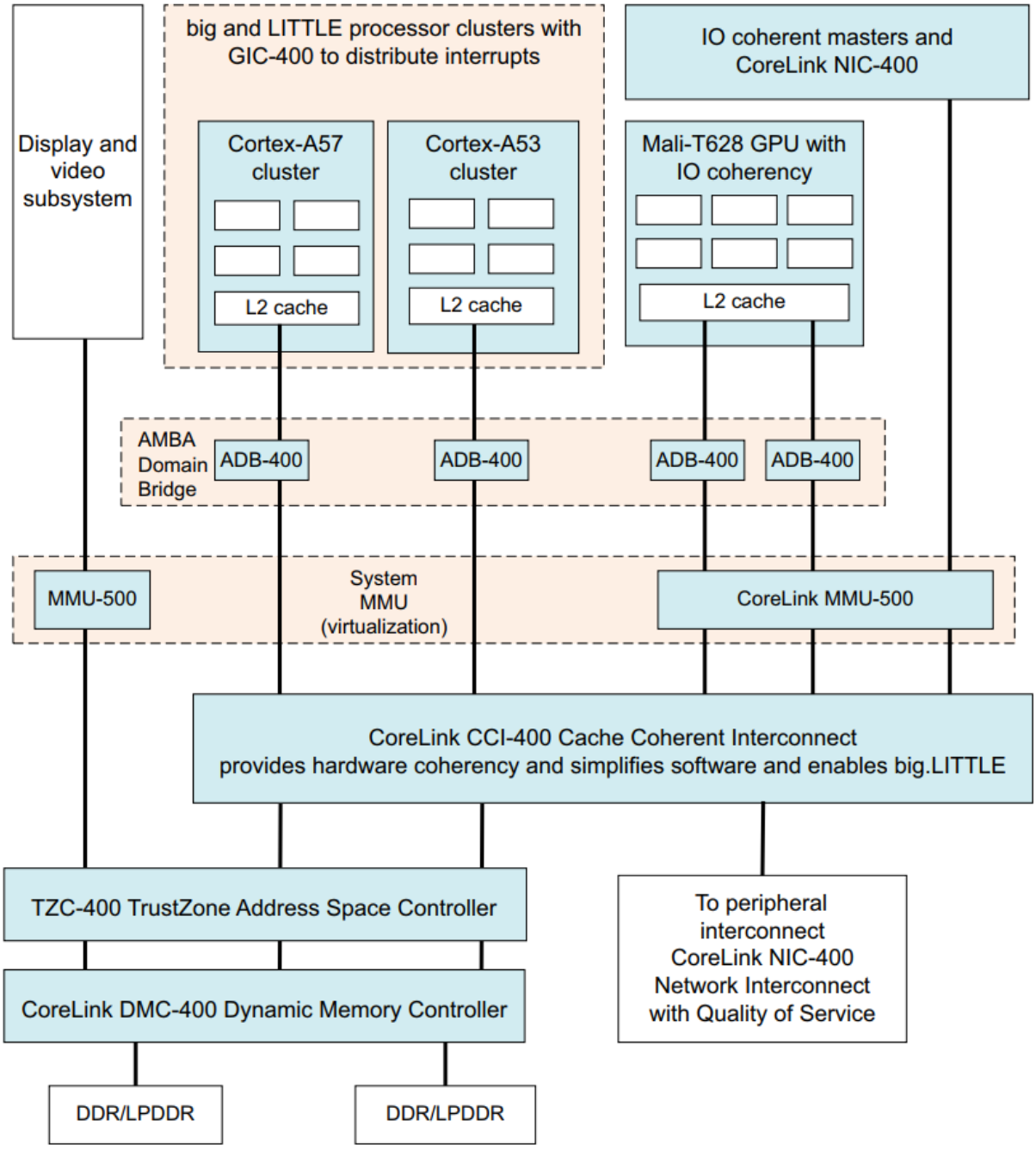

image-20220430192308808

cortex-A53 处理器提供 big.LITTLE 集群组合，并通过 AMBA 4 ACE 连接到 CCI-400，以提供完整的硬件一致性。ARM Mali®-T628 GPU 和 IO 一致性主控通过 AMBA 4 ACE-Lite 接口连接到 CCI-400。

ARM 提供了不同的互连选项来维护跨集群的一致性：

- **CoreLink CCI-400 Cache Coherent Interconnect** 这支持两个多核集群并使用 AMBA 4 和 AMBA Coherency Extensions 或 ACE。ACE 使用 MOESI 状态机实现跨集群一致性。
- **CoreLink CCN-504 Cache Coherent Network** 这支持多达四个多核集群，并包括集成的 L3 缓存和两个通道 72 位 DDR。ARM CoreLink CCN-504 Cache Coherent Network 提供最佳的系统带宽和延迟。CCN 系列互连设计用于使用 AMBA 5 CHI 连接的内核，尽管有一些 AMBA 4 ACE 支持。特别是 CCN-504 提供了符合 AMBA 4 AXI 一致性扩展 (ACE) 的端口，以实现多个 Cortex-A 系列处理器之间的完全一致性，更好地利用缓存并简化软件开发。此功能对于高带宽应用至关重要，包括需要一致的单核和多核处理器集群的游戏服务器和网络。结合 ARM CoreLink 网络互连和内存控制器 IP，CCN 提高了系统性能和电源效率。
- **CoreLink CCN-508 Cache Coherent Network** 这支持多达 8 个多核集群、32 个内核，并包括集成的 L3 缓存和四通道 72 位 DDR
- **CoreLink MMU-500 System MMU** 这为系统组件提供地址转换，请参阅第 12 章内存管理单元。
- **CoreLink TZC-400 TrustZone Address Space Controller** 这将对内存或外围设备的事务执行安全检查，并允许将内存区域标记为安全。
- **CoreLink DMC-400 Dynamic Memory Controller** 这提供了动态内存调度和与外部 DDR2/3 或 LPDDR2 内存的接口。

CHAPTER

FIFTEEN

15. 电源管理

许多 ARM 系统是手机设备和电池供电的设备。在这些系统中，功率使用和总能量使用的优化是一个关键的设计约束条件。程序员通常会花费大量时间来尝试节省此类系统中的电池寿命。

即使在不使用电池的系统中，节电也可能是一个问题。例如，由于环境的原因，您可能希望尽量减少能源使用以降低消费者的电费或者尽量减少设备产生的能量。

ARM 内核中内置了许多旨在降低功耗的设计方法。

能源使用可分为两部分：

静态的只要内核逻辑或 RAM 模块通电，就会发生静态功耗，通常也称为泄露。一般来说，漏电流与总硅面积成正比，这意味着芯片越大，漏电流越高。当您转向更小的制造几何形状时，来自泄漏的功耗比例会显着提高。

动态的动态功耗是由于晶体管开关而发生的，并且是内核时钟速度和每个周期改变状态的晶体管数量的函数。显然，更高的时钟速度和更复杂的内核会消耗更多的功率

电源管理感知操作系统动态改变内核的电源状态，平衡当前工作负载的可用计算容量，同时尝试使用最少的电源。其中一些技术动态地打开和关闭内核，或者将它们置于静止状态，它们不再执行计算。这意味着它们消耗的电源非常少。使用这些技术的主要例子是：

- 空闲管理第 15-3 页
- 动态电压和频率调整第 15-6 页

15.1 15.1 空闲管理

当内核空闲时，操作系统电源管理 (OSPM) 将其转换为低功耗状态。通常状态选择是有用的，具有不同的进入和退出延迟，以及与每个状态相关的不同级别的功耗。使用的状态通常取决于再次需要内核的速度。任何时候都可以使用的电源状态也可能取决于 SoC 中除内核之外的其他组件的活动。每个状态由一组组件定义，这些组件在进入该状态时是时钟门控或电源门控。

从低功耗状态移到运行状态所需的时间（称为唤醒延迟）在更深的睡眠状态下更长。尽管空闲电源源管理是由内核上的线程行为驱动的，但 OSPM 可以将平台置于影响内核本身之外的许多其他组件的状态。如果集群中的最后一个内核变得空闲，OSPM 可以针对影响整个集群的电源状态。同样，如果 SoC 中的最后一个内核

空闲，OSPM 可以针对影响整个 SoC 的电源状态。选择还取决于系统中其他组件的使用。一个典型的例子是当所有内核和任何其他总线主控器都空闲时，将内存置于自刷新状态。

OSPM 必须提供必要的电源管理软件基础架构来确定正确的状态选择。在空闲管理中，当内核或集群进入低功耗状态时，它可以随时通过内核唤醒事件重新激活。也就是说，一个事件可以将内核从低功耗状态唤醒，例如中断。OSPM 不需要明确的命令来使内核或集群恢复运行。OSPM 认为受影响的一个或多个内核始终可用，即使它们当前处于低功耗状态。

15.1.1 15.1.1 电源和时钟

一种可以减少能源使用的方法是去除电源，它可以去除动态和静态电流（有时称为电源门控），或者停止内核的时钟，它只去除动态功耗，可以称为时钟门控。

ARM 内核通常支持多个级别的电源管理，如下所示：

- 待机
- 保留
- 掉电
- 休眠模式
- 热插拔

对于某些操作，需要保存和恢复断电前后的状态。执行保存和恢复所花费的时间以及这项额外工作所消耗的功率都是选择适当电源管理活动的软件的重要因素。

包含内核的 SoC 设备可以具有额外的低功耗状态，例如 Stop 和 Deep Sleep。这些是指硬件锁相环 (PLL) 和稳压器由电源管理软件控制的能力。

15.1.2 15.1.2 待机

在待机操作模式下，内核保持上电状态，但其大部分时钟停止或时钟门控。这意味着内核的几乎所有部分都处于静态状态，唯一消耗的功率是由于泄漏电流和少逻辑的时钟，用于寻找唤醒条件。

使用 WFI（等待中断）或 WFE（等待事件）指令进入此待机模式。ARM 建议在 WFI 或 WFE 之前使用数据同步屏障 (DSB) 指令，以确保待处理的内存事务在更改状态之前完成。

如果调试通道处于活动状态，它将保持活动状态。内核停止执行命令直到检测到唤醒事件。唤醒条件取决于进入指令。对于 WFI，中断或外部调试请求会唤醒内核。对于 WFE，存在许多指定的事件，包括集群中其他一个内核执行 SEV 指令。

来自 Snoop 控制单元 (SCU) 的请求还可以唤醒时钟用来多核系统中进行高速缓存一致性操作。这意味着处于待机状态的内核的缓存继续与其他内核的缓存保持一致（但处于待机状态的内核不一定执行下一条指令）。内核复位总是强制内核退出待机状态。

各种形式的动态时钟门控也可以在硬件中实现。例如，当检测到空闲条件时，SCU、GIC、定时器、指令流水线或 NEON 模块可以自动进行时钟门控，以节省电源。

可以快速进入和退出待机模式（通常在两个时钟周期内）。因此，它对内核的延迟和响应能力几乎可以忽略不计。

对于 OSPM，待机状态与保留状态几乎没有区别。这种差异对于外部调试器和硬件实现是显而易见的，但对于操作系统的空闲管理子系统来说并不明显。

15.1.3 15.1.3 保留

内核状态（包括调试设置）保存在低功耗结构中，使内核能够至少部分关闭。运行操作不需要复位内核。保存的内核状态在从低功耗保持状态更改为运行操作时恢复。从操作系统的角度来看，保留状态和待机状态之间没有区别，除了进入方法、延迟和使用相关的限制。但是，从外部调试器的角度来看，状态不同，因为外部调试请求调试事件保持未决状态，并且无法访问内核电源域中的调试寄存器。

15.1.4 15.1.4 掉电

在这种状态下，内核断电。设备上的软件必须保存所有内核状态，以便在掉电时保存。从断电变为运行操作必须包括：

- 在电源电平恢复后复位内核
- 恢复保存的内核状态

断电状态的定义特征是它们对上下文具有破坏性。这会影响在给定状态下关闭的所有组件，包括内核，以及在更深的状态下系统的其他组件，例如 GIC 或特定于平台的 IP。根据调试和跟踪电源域的组织方式，调试和跟踪上下文之一或两者可能在某些掉电状态下丢失。必须提供机制以使操作系统能够为每个给定状态执行相关的上下文保存和恢复。恢复执行从复位向量开始，之后每个操作系统都必须恢复其上下文。

15.1.5 15.1.5 休眠模式

休眠模式是一种断电状态的实现。在休眠模式下，内核逻辑断电，但高速缓存 RAM 保持通电状态。RAM 通常保持在低功耗保持状态，因此在该状态下它们保留他们的内容，但在其他方面不起作用。这提供了比完全关闭更快的重新启动机制，因为实时数据和代码保留在缓存中。同样，在多核系统中，可以将单个核置于休眠模式。

在允许集群中的各个内核进入休眠模式的多核系统中，在内核断电时，没有保持一致性的余地。因此，这样的内核必须首先将自己与相干域隔离开来。在执行此操作之前，它们会清除所有脏数据，并且通常使用另一个内核向外部逻辑发出信号以重新供电以将其唤醒。

唤醒的内核必须在重新加入一致性域之前恢复原始内核状态。因为当内核处于休眠模式时内存状态可能已经改变，所以无论如何它可能不得不使缓存无效。因此，休眠模式更有可能在单核环境中而不是在集群中有用。这是因为离开和重新加入一致性域的额外费用。在集群中，休眠模式通常可能仅在其他内核已关闭时仅由最后一个内核使用。

15.1.6 15.1.6 热插拔

CPU 热插拔是一种可以动态打开或关闭内核的技术。OSPM 可以使用热插拔来根据当前的计算要求更改可用的计算容量。出于可靠性原因，有时也使用热插拔。热插拔和空闲状态下使用断电状态之间存在许多差异：

1. 当一个内核被上电断开时，监控软件停止对该内核的所有使用中断和线程处理。调用操作系统不再认为内核可用。

2.OSPM 必须发出显式命令以使内核重新联机，即热插拔内核。对应的监控软件仅在该命令之后开始调度或启用对该内核的中断。

操作系统通常在一个主内核上执行大部分内核启动过程，在稍后阶段使辅助内核联机。二次引导的行为与将内核热插入系统非常相似。两种情况下的操作几乎相同。

15.2 15.2 动态电压和频率调整

许多系统在其工作负载可变的条件下运行。因此，能够降低或提高内核性能以匹配预期的内核工作负载是很有用的。更缓慢地为内核提供时钟可以降低动态功耗。

动态电压和频率缩放 (DVFS) 是一种节能技术，它利用：

- 功耗与工作频率之间的线性关系。
- 功耗与工作电压之间的二次方关系。

这种关系如下：$$ $P = C \times V^2 \times f$ $$ 其中：

P 是动态功耗

C 是逻辑电路的开关电容

V 是工作电压

f 是工作频率

通过调整内核时钟的频率来实现节约功耗。

在较低频率下，内核也可以在较低电压下运行。降低电源电压的好处是它可以降低动态和静态功耗。

给定电路的工作电压与电路可以安全工作的频率范围之间存在特定于实现的关系。给定的操作频率及其相应的操作电压表示为元组，称为操作性能点 (OPP)。对于给定的系统，可达到的 OPP 范围统称为系统 DVFS 曲线。

操作系统使用 DVFS 来节省能源，并在必要时保持在热限制范围内。操作系统提供 DVFS 策略来管理功耗和所需的性能。以高性能为目标的策略选择更高的频率并使用更多的能量。旨在节省能源的政策选择较低的频率，因此会导致较低的性能。

15.3 15.3 电源相关的汇编语言指令

ARM 汇编语言包括可用于将内核置于低功耗状态的指令。该架构将这些指令定义为提示，这意味着内核在执行它们时不需要采取任何特定操作。然而，在 Cortex-A 处理器系列中，这些指令的实现方式是关闭几乎所有内核部分的时钟。这意味着内核的功耗显着降低，仅消耗静态漏电流，没有动态功耗产生。

WFI 指令具有暂停执行的效果，直到内核被以下条件之一唤醒：

- 一个 IRQ 中断，即使得 PSTATE I 位被置位
- 一个 FIQ 中断，即使得 PSTATE F 位被置位
- 一个异步中止

如果在相关 PSTATE 中断标志被禁用时内核被中断唤醒，内核将执行 WFI 之后的下一条指令。

WFI 指令广泛用于电池供电的系统。例如，手机可以每秒多次将内核置于待机模式，同时等待您按下按钮。

WFE 类似于 WFI。它暂停执行直到发生事件。这可以是列出的事件条件之一，也可以是集群中另一个内核发出的事件信号。其他内核可以通过执行 SEV 指令来发出事件信号。SEV 向所有内核发送一个事件信号。还可以对通用定时器进行编程为周期性事件因此通过 WFE 来唤醒内核。

15.4 15.4 电源状态协调接口

电源状态协调接口 (PSCI) 提供了一种与操作系统无关的方法，用于实现可以启动或关闭内核的电源管理用例。这包括：

- 内核空闲管理
- 动态的添加和移除内核（热插拔）以及副核引导启动
- 大小端数据迁移
- 系统关闭和复位

使用此接口发送的消息被所有相关的执行级别接收到。也就是说，如果实现了 EL2 和 EL3，则在 guest 中执行的 Rich OS 发送的消息必须被 hypervisor 接收。如果 hypervisor 发送它，则消息必须由安全固件接收，然后与受信任的操作系统协调。这允许每个操作系统确定是否需要保存上下文。

有关详细信息，请参阅电源状态协调接口 (PSCI) 规范。

CHAPTER

SIXTEEN

16. BIG.LITTLE 技术

现代软件堆栈对移动系统提出了相互矛盾的要求。一方面是对游戏等任务的高性能需求，另一方面是对音频播放等低强度应用的节俭能源储备的持续要求。

传统上，不可能有一个单一的处理器设计能够同时具备高峰值性能和高能效。这意味着大量能量被浪费了，因为高性能内核将用于低强度任务，从而导致电池寿命缩短。性能本身会受到内核可以持续运行的热限制的影响。

ARM 的 big.LITTLE 技术通过将高能效的 LITTLE 内核与高性能的 big 内核结合在一起解决了这个问题。big.LITTLE 是异构处理系统的一个示例。此类系统通常包括具有不同微架构的几种不同处理器类型，例如通用处理器和专用 ASIC。

big.LITTLE 将异构性更进一步，因为它包括通用处理器，这些处理器的微架构不同，但指令集架构兼容。经常与此类系统一起使用的术语是异构多处理（HMP）（14-8 页异构多核处理）。HMP 与非对称多处理 (AMP)（第 14-7 页的非对称处理）的不同之处在于，HMP 系统中的所有处理器都是完全一致的，并且运行相同的操作系统映像。

根据性能要求，软件可以在大型或小型处理器（或两者）上运行。当需要最高性能时，可以将软件移至仅在大型处理器上运行。对于普通任务，软件可以在 LITTLE 处理器上完美运行。通过这种组合，big.LITTLE 提供了一种解决方案，能够在系统的热范围内提供最新移动设备所需的峰值性能，并具有最大的能源效率。

16.1 16.1 big.LITTLE 的系统结构

big.LITTLE 系统中的两种类型的内核都是完全缓存一致的，并且共享相同的指令集架构 (ISA)。相同的应用程序二进制文件在任何一个上都未经修改即可运行。处理器内部微架构的差异使它们能够提供不同的功率和性能特征，这些特征是 big.LITTLE 概念的基础。这些通常由操作系统管理。

big.LITTLE 软件模型需要在 big 和 LITTLE 集群之间透明且高效地传输数据。硬件一致性实现了这一点，对软件透明。集群之间的一致性由缓存一致性互连提供，例如第 14 章中描述的 ARM CoreLink CCI-400。如果没有硬件一致性，大核和小核之间的数据传输将始终通过主内存进行，但这会很慢而且不会省电。此外，它还需要复杂的缓存管理软件，以实现大集群和小集群之间的数据一致性。

此外，这样的系统还需要一个共享的中断控制器，例如 GIC-400，使中断能够在集群中的任何内核之间迁移。所有内核都可以使用分布式中断控制器（例如 CoreLink GIC-400）相互发送信号。任务切换通常完全在 OS

调度程序中处理，对应用软件不可见。下图显示了一个示例系统。

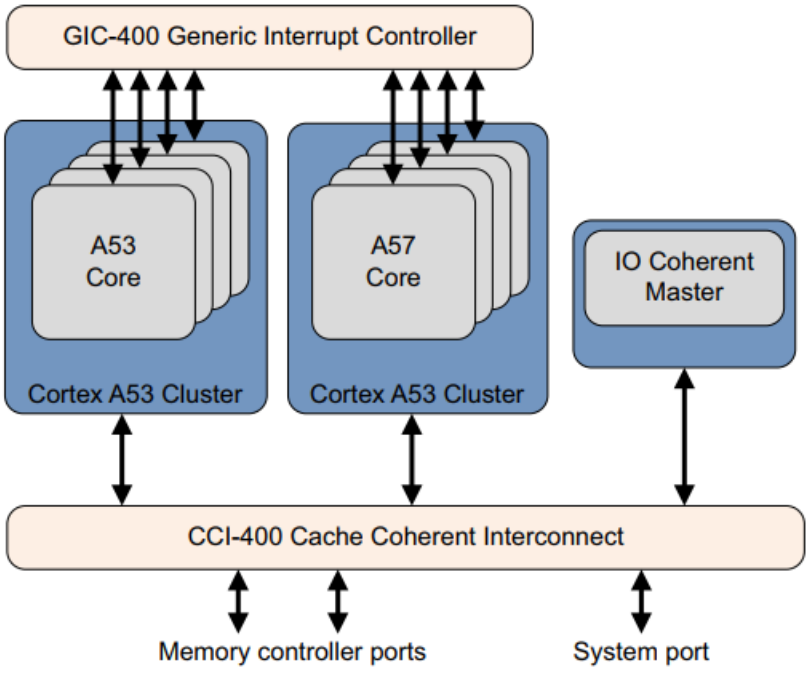

Figure 16-1 Typical big.LITTLE system image-20220502003122009

16.1.1 16.1.1 big.LITTLE 配置

许多种 big.LITTLE 配置都是可能的，上使用 Cortex-A57 内核作为大集群，使用 Cortex-A53 内核作为小集群，但也可以使用其他配置。

小集群能够处理大多数低强度任务，例如音频播放、网页滚动、操作系统事件和其他始终在线、始终连接的任务。因此，小集群很可能是软件堆栈所在的位置，例如运行游戏或视频处理等密集型任务。

大集群可用于繁重的工作负载，例如某些高性能游戏图形。网页渲染是另一个常见的例子。这两种集群类型的结合提供了节省能源和满足移动设备中应用程序堆栈日益增长的性能需求的机会。

16.2 16.2 big.LTTLE 中的软件执行模型

big.LITTLE 有两种主要的执行模型

迁移

迁移模型是对 DVFS 等电源性能管理技术的自然扩展（参考 15-6 章节动态电压和频率调整）

迁移模型有两种类型：

- 集群迁移
- CPU 迁移

迁移操作类似于 DVFS 操作点转换。响应负载变化，遍历内核的 DVFS 曲线上的工作点。当当前内核（或集群）达到最高运行点时，如果软件堆栈需要更高的性能，则执行内核（或集群）迁移操作。然后在另一个内核（或集群）上继续执行，遍历该内核（或集群）上的操作点。当不需要性能时，执行可以切换回来。

全局任务调度

在全局任务调度（请参阅第 16-5 章节的全局任务调度）中，操作系统任务调度程序了解大核和小核之间计算能力的差异。调度程序跟踪每个单独的软件线程的性能要求，并使用该信息来决定为每个软件线程使用哪种类型的内核。可以关闭未使用的内核。与迁移模型相比，这种方法具有许多优点。

16.2.1 16.2.1 集群迁移

如果只有一个集群，无论是大集群还是小集群，在任何时候都处于活动状态，除非在集群上下文切换到另一个集群期间非常短暂。为了获得最佳的功率和性能效率，软件堆栈主要在节能的 LITTLE 集群上运行，而在大集群上只运行很短的时间。此模型要求两个集群中的内核数量相同。

该模型不能很好地处理不平衡的软件工作负载，即在集群内的内核上放置显着不同负载的工作负载。在这种情况下，集群迁移会导致完全切换到大集群，即使并非所有内核都需要该级别的性能。出于这个原因，集群迁移不如其他方法受欢迎。

16.2.2 16.2.2 CPU 迁移

在这个模型中，每个大内核都与一个小内核配对。在任何时候，每对中只有一个内核处于活动状态，不活动的内核被断电。该对中的有源磁芯是根据当前负载条件选择的。使用第 16-5 页图 16-2 中的示例，操作系统可以看到四个逻辑内核。每个逻辑内核在物理上可以是大内核或小内核。这种选择是由动态电压和频率调整 (DVFS) 驱动的。此模型要求两个集群中的内核数量相同。

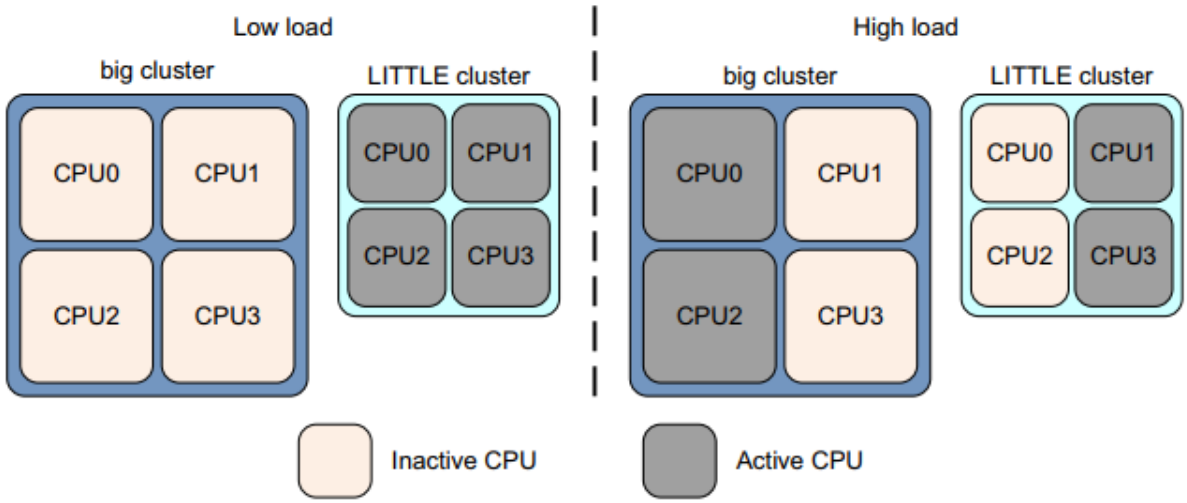

Figure 16-2 CPU migration image-20220501233951309

系统主动监控每个内核上的负载。高负载使得执行上下文被移动到大内核，反之，当负载低时，执行被移动到小内核。在任何时候，配对中只有一个内核可以处于活动状态。当负载从出站内核（负载离开的内核）移动到入站内核（它到达的内核）时，前者被关闭。该模型允许在任何时候混合使用大核和小核。

16.2.3 16.2.3 全局任务调度

通过 big.LITTLE 技术的发展，ARM 已经将软件模型从各种迁移模型发展到全局任务调度 (GTS)，它构成了 big.LITTLE 技术所有未来发展的基础。GTS 的 ARM 实现称为 big.LITTLE 多核处理系统 (MP)。

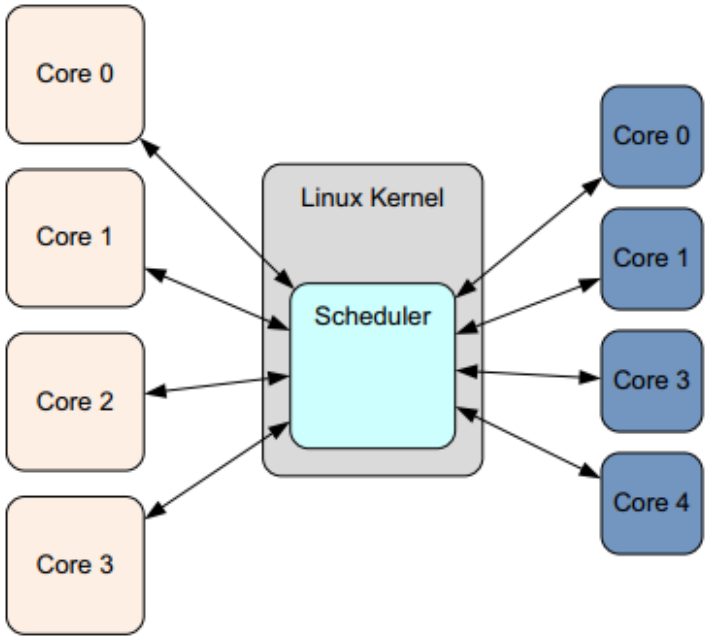

Figure 16-3 Global Task Scheduling image-20220501234416320

在这个模型中，操作系统任务调度程序知道大核和小核之间计算能力的差异。使用统计数据，调度程序跟踪每个单独软件线程的性能要求，并使用该信息来决定每个使用哪种类型的内核。该模型可以在任何集群中具有任意数量内核的 big.LITTLE 系统上运行。这显示在第 16-5 页的图 16-3 中。与迁移模型相比，这种方法具有许多优点，例如：

- 系统可以有不同数量的大核和小核。
- 与迁移模型不同，任意数量的内核可以在任何时候处于活动状态。如果需要峰值性能，这可以增加可用的最大计算容量。
- 可以隔离大集群，专供密集线程使用，而轻线程运行在 LITTLE 集群上。这使得繁重的计算任务能够更快地完成，因为没有额外的后台线程。
- 可以将中断单独定位到大内核或小内核。

16.3 16.3 big.LITTLE 多核处理

对于 Linux 内核上的 big.LITTLE MP，基本要求是调度程序决定软件线程何时可以在 LITTLE 内核或大内核上运行。调度程序通过将跟踪的软件线程负载与可调负载阈值、向上迁移阈值和向下迁移阈值进行比较来实现这一点，如图 16-4 所示。

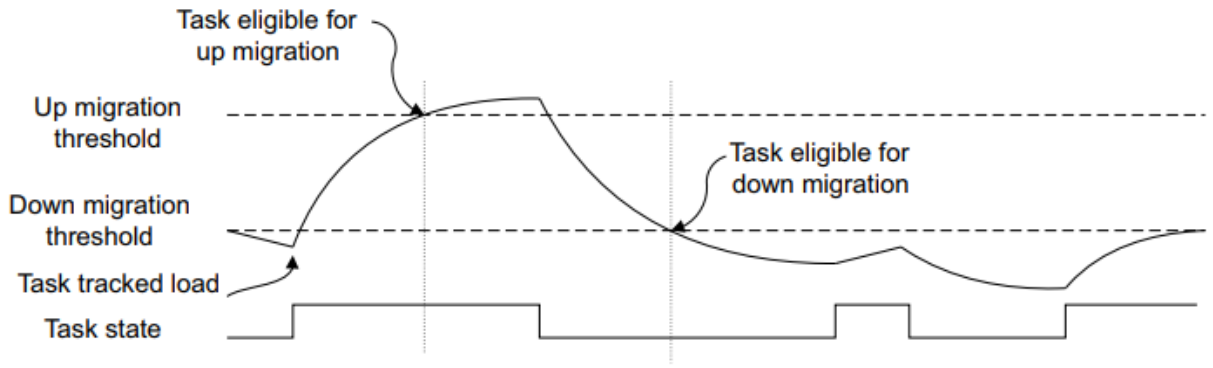

Figure 16-4 Migration thresholds image-20220502001058658

当当前分配给小内核的线程的跟踪负载平均值超过向上迁移阈值时，该线程被认为符合迁移到大内核的条件。相反，当当前分配给大内核的线程的平均负载低于向下迁移阈值时，它被认为有资格迁移到小内核。在 big.LITTLE MP 中，这些基本规则管理大核和小核之间的任务迁移。在集群内，标准 Linux 调度程序负载平衡适用。这试图在一个集群中的所有内核之间保持负载平衡。

通过基于内核的当前频率调整跟踪的负载指标来完善模型。当内核以半速运行时正在运行的任务，以比内核全速运行时的一半速率累积跟踪负载。这使 big.LITTLE MP 和 DVFS 管理能够和谐地协同工作。

big.LITTLE MP 使用多种机制来确定何时在大核和小核之间迁移任务：

16.3.1 16.3.1 分叉迁移

这在使用 fork 系统调用创建新的软件线程时运行。此时，显然没有可用的历史负载信息。系统默认为新线程使用大内核，假设轻线程由于唤醒迁移而快速迁移到小内核。

分叉迁移使要求苛刻的任务受益，而且成本不高。低强度和持久性的线程，如 Android 系统服务，仅在创建时移动到大内核，之后迅速移动到更合适的小内核。整个过程中明显要求高的线程不会因为首先在 LITTLE 内核上启动而受到惩罚。线程偶尔运行但往往需要性能的线程受益于在大型集群上启动并根据需要继续在那里运行。

16.3.2 16.3.2 唤醒迁移

当之前空闲的任务准备好运行时，调度程序必须决定哪个集群执行该任务。要在 big 和 LITTLE 之间进行选择，big.LITTLE MP 使用任务的跟踪负载历史记录。通常，假设任务在相同的集群上恢复前。不会为处于休眠状态的任务更新负载指标。因此，当调度程序在唤醒时检查任务的负载指标时，在选择执行它的集群之前，该指标具有任务上次运行时的值。此属性意味着足够繁忙的任务总是倾向于在大内核上唤醒。例如，音频播放任务会周期性地忙碌。但这通常是一项要求不高的任务，因此整体负载可能会舒适地放在一个小内核上。任务必须实际修改其行为以更改集群。

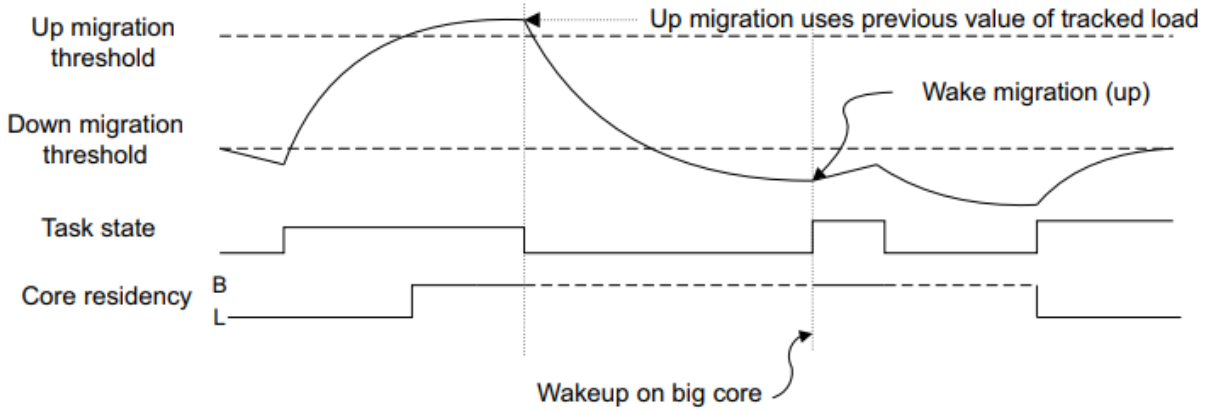

Figure 16-5 Wake migration on a big core image-20220502001655399

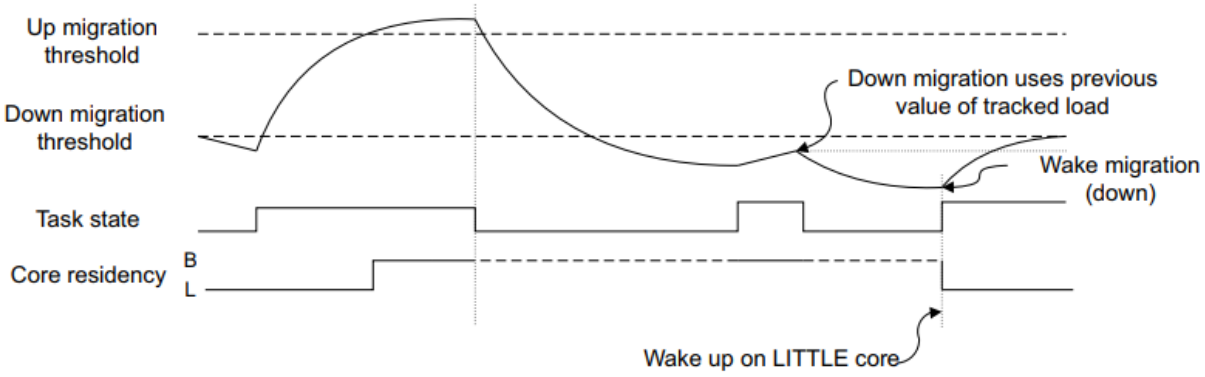

Figure 16-6 Wake migration on a LITTLE core image-20220502001712651

如果任务修改了其行为，并且负载指标已超过向上或向下迁移阈值，则可以将任务分配给不同的集群。图 16-5 和图 16-6 说明了这个过程。定义的规则确保大核通常只运行一个密集线程并运行到完成，因此向上迁移只发生在空闲的大核上。向下迁移时，此规则不适用，可以将多个软件线程分配给一个小内核。

16.3.3 16.3.3 强制迁移

强制迁移处理长时间运行的软件线程不休眠或不经常休眠的问题。调度程序定期检查每个 LITTLE 内核上运行的当前线程。如果跟踪的负载超过向上迁移阈值，则任务将转移到大内核，如第 16-9 页的图 16-7 中所示。

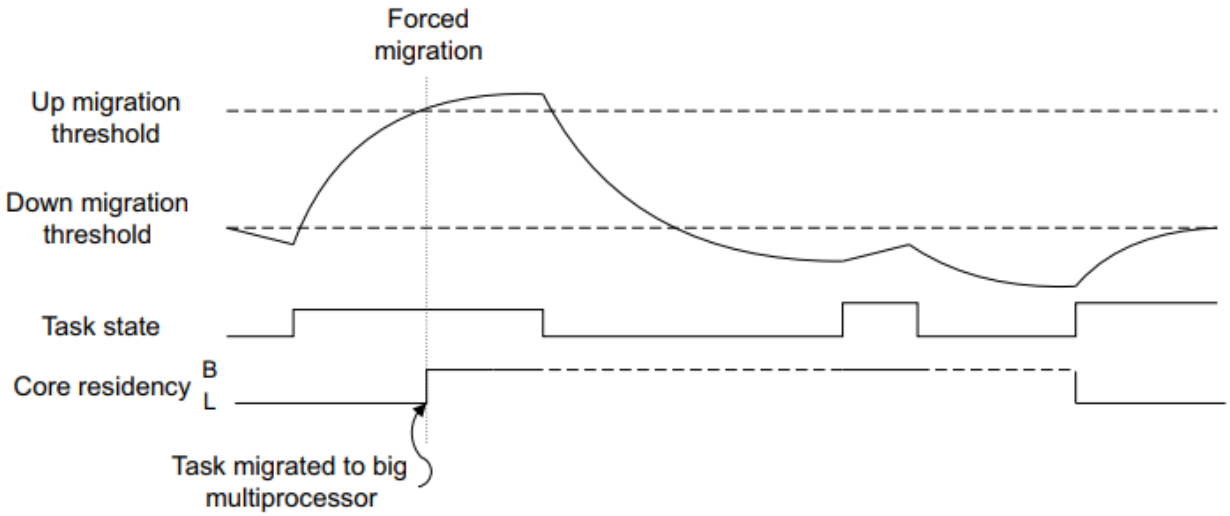

Figure 16-7 Forced migration image-20220502001844416

16.3.4 16.3.4 空闲拉取迁移

空闲拉取迁移旨在充分利用活跃的大内核。当大内核没有任务要运行时，会检查所有小内核，以查看小内核上当前正在运行的任务是否具有高于向上迁移阈值的负载指标。然后可以立即将这样的任务迁移到空闲的大内核。如果没有找到合适的任务，则可以将大内核断电。这种技术可确保大内核在运行时始终执行系统中最密集的任务并运行它们以完成。

16.3.5 16.3.5 卸载迁移

卸载迁移需要禁用正常的调度程序负载平衡。这样做的缺点是长时间运行的线程可以集中在大内核上，而让小内核闲置且未得到充分利用。在这种情况下，通过利用所有内核可以明显提高整体系统性能。

卸载迁移用于定期将线程向下迁移到 LITTLE 内核，以利用未使用的计算容量。以这种方式向下迁移的线程如果在下一次调度机会超过阈值，则仍然是向上迁移的候选者。

CHAPTER

SEVENTEEN

17. 安全

提供一定级别安全性的系统，即受信任的系统，是一种保护资产（例如密码和加密密钥或信用卡详细信息）免受一系列似是而非的攻击的系统，以防止它们被复制、损坏或制造不可用。

安全性通常由机密性、完整性和可用性原则定义。机密性是密码和加密密钥等资产的关键安全问题。防止修改和真实性证明对于用于安全的软件和片上机密至关重要。可信系统的示例可能包括移动支付的密码输入、数字版权管理（DRM）和电子票务。在开放系统的世界中，安全性更难实现，在开放系统中，您可以将各种软件下载到一个平台上，无意中也会下载恶意或不可信的代码，这些代码可能会篡改您的系统。

移动设备可用于观看视频、听音乐、玩游戏或浏览 Web 和访问金融服务。这需要用户和银行或服务提供商都信任该设备。该设备运行具有高度连接性的复杂操作系统，并且可能容易受到恶意软件的攻击。您可以通过软件系统设计实现一定程度的安全性，但您可以通过 CPU 和系统级内存分区获得更高级别的保护。

ARM 处理器包括特定的硬件扩展，以支持构建受信任的系统。编写可信操作系统或可信执行环境 (*Trusted Execution Environment*, TEE) 系统超出了本书的范围。但是，如果您设置安全部分字段以实现 ARMv7 安全扩展，请注意这会对操作系统和非特权代码（即不属于受信任系统的代码）施加一些限制。

软件和硬件攻击可以分为以下几类：

软件攻击

恶意软件的攻击通常不需要对设备进行物理访问，并且可以利用操作系统或应用程序中的漏洞。

简单的硬件攻击

这些通常是被动的、主要是非破坏性的攻击，需要访问设备并接触电子设备，并使用逻辑探针和 JTAG 运行控制单元等常用工具。

实验室硬件攻击

这种攻击需要复杂且昂贵的工具，例如聚焦离子束 (FIB) 技术或功率分析技术，并且更常用于针对智能卡设备。

TrustZone 技术旨在防止软件和简单硬件攻击。

17.1 17.1 TrustZone 硬件架构

TrustZone 体系结构为系统设计人员提供了一种方法，可以使用 TrustZone 安全扩展和安全外设来帮助保护系统。即使不使用安全特性，底层程序员也必须理解 TrustZone 体系结构对系统施加的限制。

ARM 安全模型划分了设备硬件和软件资源，因此它们要么存在于安全子系统的安全世界中，要么存在于其他一切的正常世界中。系统硬件确保无法从普通世界访问任何安全世界资产。安全设计将所有敏感资源置于安全世界中，理想情况下具有运行稳健的软件，可以保护资产免受各种可能的软件攻击。

ARM 架构参考手册（*ARM Architecture Reference Manual* ）使用术语“安全”和“不安全”来表示系统安全状态。“不安全”状态并不意味着安全漏洞，而是普通操作，因此与“普通”世界相同。正常情况下，非安全世界和安全世界之间存在一种主从关系，因此只有当普通世界执行安全监视器调用 (*Secure Monitor Call*，*SMC*) 时，安全世界才会被执行，(参考 *SMC instruction in ARMv8-A Architecture Reference Manual*)。“世界”这个词不仅用来描述执行状态，还用来描述只有在该状态下才能访问的所有内存和外设。

支持 TrustZone 的体系结构使单个物理核心可以以时间分片的方式同时执行普通世界和安全世界的代码，尽管这取决于产生中断的外设的可用性，这些外设可以被配置为仅供安全世界访问。例如，可以使用一个安全定时器中断来为安全世界保证一些执行时间，其方式类似于抢占式多任务处理。此类外围设备可能可用也可能不可用，具体取决于平台设计人员打算支持的安全级别和用例。

或者，可以使用更接近协作多任务的执行模型。在这种情况下，虽然安全世界在每个世界可以访问的资源方面独立于普通世界，但执行时间的调度通常在两个世界之间相互依赖。

与固件或任何其他系统软件一样，安全世界中的软件必须小心谨慎，将其对系统其他部分的影响降到最低。例如，通常避免消耗大量的执行时间，除非执行普通世界请求的某些操作，并且应该尽快向普通世界发出不安全中断的信号。这有助于确保普通世界软件的良好性能和响应性，而不需要进行大量的移植。

存储系统是通过伴随外设和存储器地址的附加位来划分的。该位称为 NS 位，表示访问是安全的还是非安全的。该位被添加到所有内存系统事务中，包括缓存标签以及对系统内存和外围设备的访问。这个额外的地址位为安全世界提供了一个物理地址空间，为普通世界提供了一个完全独立的物理地址空间。在普通世界中运行的软件只能对内存进行非安全访问，因为在普通世界生成的任何内存事务中，ARM Core 总是将 NS 位设置为 1。运行在安全世界中的软件通常只进行安全内存访问，但也可以使用页表项中的 NS 和 NSTable 标记对特定的内存映射进行非安全访问。

尝试对标记为安全的缓存数据执行非安全访问会导致缓存未命中。尝试对标记为安全的外部存储器执行非安全访问会导致存储器系统拒绝该请求，并且从设备返回错误响应。没有向非安全系统指示错误是由尝试访问安全内存引起的（无法区分这个错误是访问安全内存导致的还是其他原因导致的）。

在 AArch64 中，EL3 有自己的翻译表，由寄存器 TTBR0_EL3 和 TCR_EL3 管理。在安全世界中只允许第一阶段的翻译，并且没有 TTBR1_EL3。AArch64 EL1 转换表寄存器不在安全状态之间存储，因此在安全监视器的上下文切换时必须为每个世界保存和恢复 TTBR0_EL1、TTBR1_EL1 和 TCR_EL1 的值。这使每个世界都可以拥有一组本地转换表，其中安全世界映射对普通世界隐藏并受到保护。安全世界转换表中的条目包含 NS 和 NSTable 属性位，它们决定特定访问是否可以访问安全或非安全物理地址空间。

安全项和非安全项可以在缓存和转换后备缓冲区 (*Translation Lookaside Buffers*，*TLBs*) 中共存。在两个世界之间切换时，不需要使缓存数据无效。普通世界只能生成非安全访问，所以只能命中标记为非安全的缓存行，而安全世界可以生成安全的和非安全的访问。TLB 中的条目记录了哪个世界生成了一个特定的条目，虽然非

安全状态永远不能对安全进行操作，但是安全世界可以将 NS 行分配到缓存中。此外，对于每个异常级别，缓存是分别启用和禁用的。缓存控制是独立于两个世界的，但不是独立于所有的异常级别，因此 EL0 永远不能直接启用或禁用缓存，而 EL2 可以覆盖非安全的 EL1 的行为。

17.2 17.2 通过中断切换安全世界

当内核执行来自两个世界的代码时，它们之间的上下文切换通过执行安全监视器 (*SMC*) 指令或硬件异常机制（如中断）发生。ARM 处理器有两种中断类型，FIQ 和 IRQ。

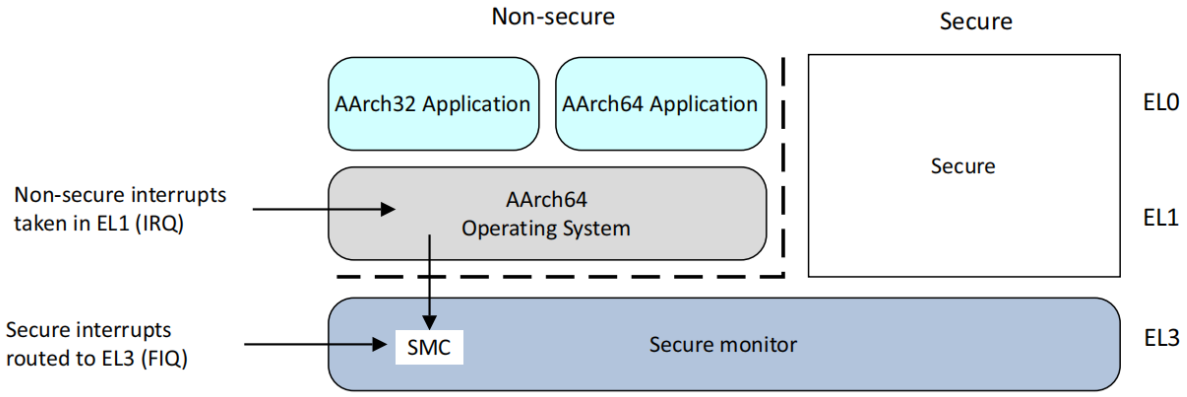

Figure 17-1 Non-secure interrupts image-20220426184356293

对于安全中断有明确的支持，将异常和中断重定向到 EL3，而不依赖于当前的 DAIF。然而，这些控制只区分主要的中断类型:IRQ、FIQ 和异步中止。更细粒度的控制要求将中断过滤到安全组和非安全组。要想有效地做到这一点，需要得到 GIC 的支持，GIC 对此有明确的设施。

一个典型的用例是将 FIQ 用作安全中断，方法是将安全中断源映射为中断控制器中的 FIQ。相关外设和中断控制器寄存器必须标记为仅安全访问，以防止正常世界重新配置这些中断。

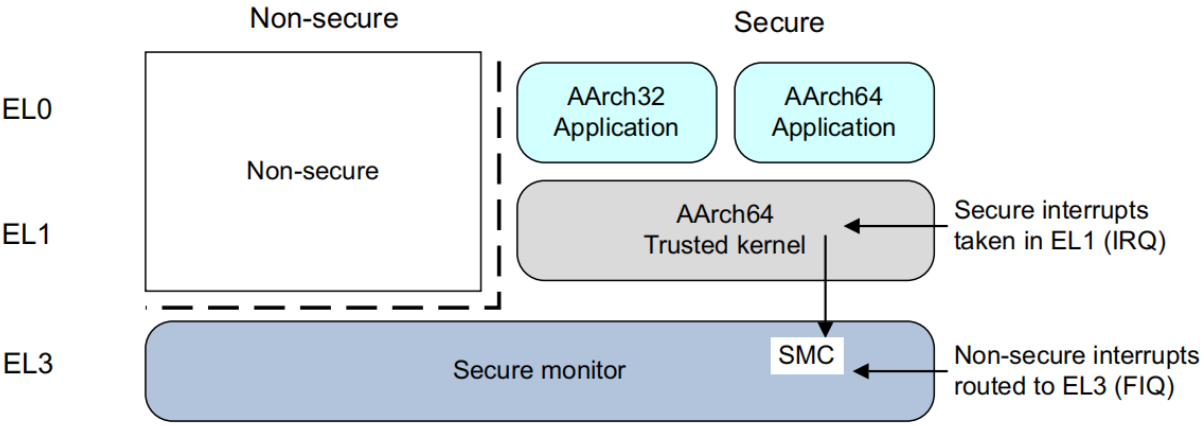

Figure 17-2 Secure interrupts image-

20220426171144670

这些安全 FIQ 中断必须路由到处于安全执行状态的处理程序。

使用安全扩展的实现通常有一个轻量级的可信内核，在安全世界中托管安全服务（比如加密）。一个完整的操作系统运行在普通世界，并能够使用 SMC 指令访问安全服务。通过这种方式，普通世界可以访问服务功能，而不会有将安全资产（如关键材料或其他受保护的数据）暴露给在普通世界中执行的任意代码的风险。

17.3 17.3 多核系统中的安全

多核系统中的每个核都具有本章所述的相同的安全特性。集群中的任意数量的核都可以在任何时间点在安全世界中执行，并且核能够在彼此独立的世界之间转换。附加的寄存器控制普通世界代码是否可以修改窥探控制单元（*Snoop Control Unit*，*SCU*）的设置。类似地，必须对跨多核簇分配优先级的中断的 GIC 进行配置，以注意安全问题。

17.3.1 17.3.1 普通世界与安全世界互动

如果您在包含一些安全服务的系统中编写代码，那么了解如何使用这些服务会很有用。典型系统具有轻量级内核或 TEE 托管服务，例如安全世界中的加密。这与正常世界中的完整操作系统交互，该操作系统可以使用 SMC 调用访问安全服务。通过这种方式，普通世界可以访问服务功能，而不会将密钥暴露在风险中。

通常，应用程序开发人员不会直接与安全扩展、TEE 或可信服务交互。相反，它们使用由普通世界的库提供的高级 API，例如 authenticate()。该库由与可信服务相同的厂商提供。例如，为信用卡公司提供服务，并处理低级别的交互。图 17-3 以流的形式显示了这种交互，用户应用程序调用 OS 适当的 API，该 API 然后传递给驱动程序代码，然后通过安全监视器将执行传递给 TEE。

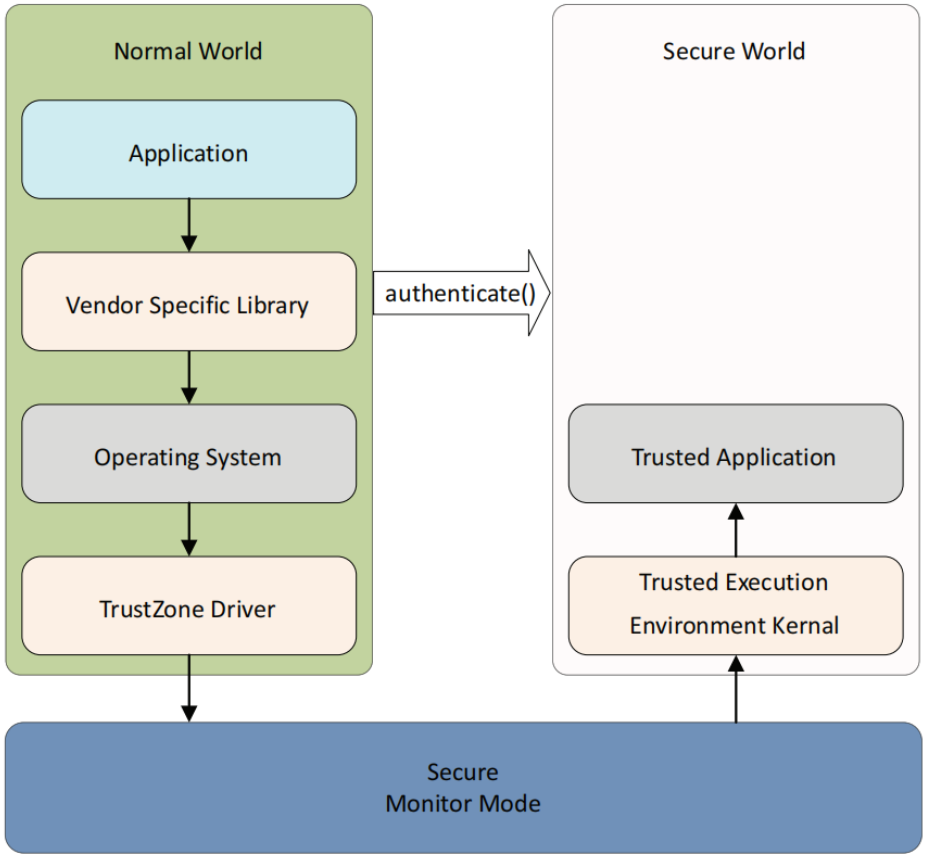

Figure 17-3 Interaction with Security Extension image-20220426174333761

在安全世界和普通世界之间传递数据是很常见的。例如，在安全的世界中，你可能有一个签名检查器。普通世界可以使用 SMC 调用请求安全世界验证下载更新的签名。安全世界需要访问普通世界使用的内存，安全世界可以在它的转换表描述符中使用 NS 位，以确保它使用非安全访问来读取数据。这非常重要，因为数据可能已经在缓存中，由普通世界中执行的访问把地址标记为非安全的。安全属性可以看作是一个额外的地址位。如果核心使用安全内存访问来尝试读取数据，它不会命中已经在缓存中的非安全数据。

如果你是普通世界的程序员，一般来说，可以忽略安全世界中发生的事情，因为它的操作对你是隐藏的。一个副作用是，如果在安全环境中发生中断，中断延迟可能会稍微增加，但与典型操作系统的总体延迟相比，这种增加是很小的。请注意，这种类型的服务质量问题取决于安全世界操作系统的良好设计和实现。

创建安全世界操作系统和应用程序的细节超出了本书的范围.

17.3.2 17.3.1 安全调试

安全系统还控制调试提供的可用性。可以通过完整的 JTAG 调试和跟踪控制为普通和安全软件世界配置单独的硬件，这样就不会泄露有关受信任系统的信息。可以通过安全外围设备控制硬件配置选项，也可以硬连线并使用以下信号控制它们：

- 安全特权侵入式调试启用 (*Secure Privileged Invasive Debug Enable, SPIDEN*)：JTAG 调试。
- 安全特权非侵入调试启用 (*Secure Privileged Non-Invasive Debug Enable, SPNIDEN*) : 跟踪和性能监视器。

17.4 17.4 安全状态与非安全状态的切换

使用 ARMv7 安全扩展，软件使用监控模式在安全和非安全状态之间切换。这种模式与其他在安全状态中的特权模式对等。

对于 ARMv8 架构，当 EL3 使用 AArch32 时，系统的行为与 ARMv7 相同，以确保完全兼容，因此安全状态下的所有特权模式都被视为处于 EL3。

AArch32 的安全模型如图 17-4 所示。在这种情况下，AArch32 使用 EL3 来提供安全操作系统和监视器。

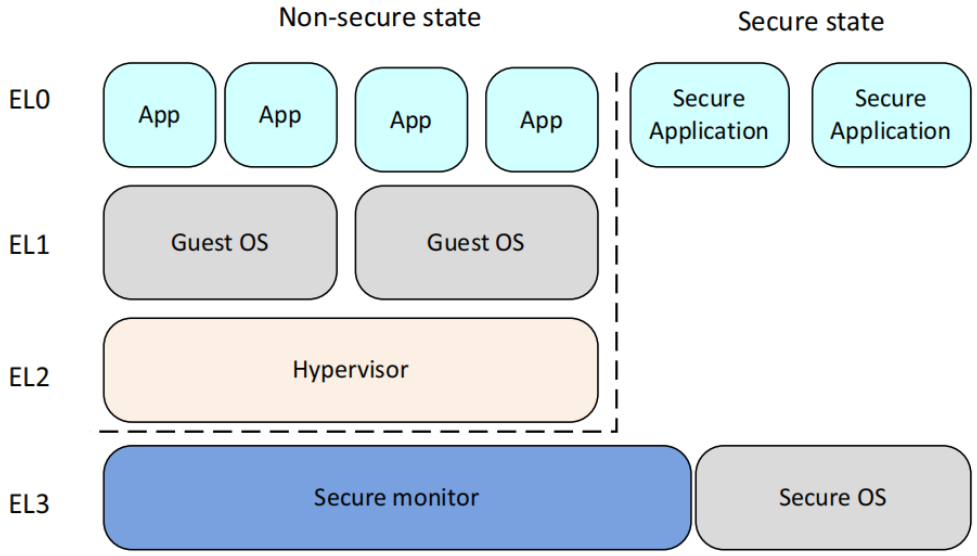

Figure 17-4 Security model when EL3 is using AArch32 image-20220426180934516

为了与 ARMv7 架构保持一致，安全状态 EL1 和 EL0 具有与非安全状态 EL1 和 EL0 不同的虚拟地址空间。这允许来自 ARMv7 32 位架构的安全端代码用于具有 64 位操作系统或在非安全端运行的管理程序的系统中。

图 17-5 显示了 AArch64 使用 EL3 的安全模型来提供一个安全监视器。对于 AArch32 来说，EL3 状态是不可用的，但是 EL1 可以用于安全操作系统。当 EL3 使用 AArch64 时，EL3 级别用于执行负责在非安全状态和安全状态之间切换的代码。

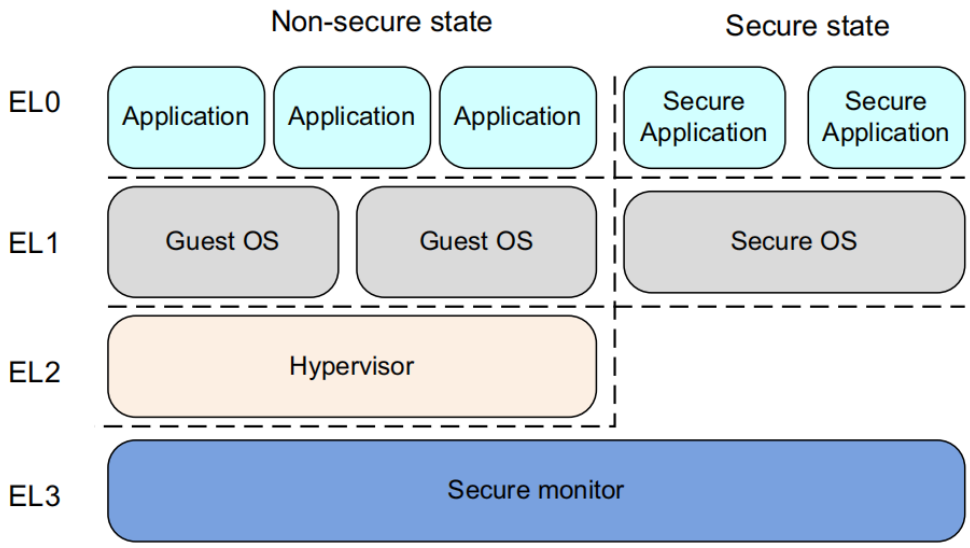

Figure 17-5 Security model when EL3 is using AArch64 image-20220426181731802

CHAPTER

EIGHTEEN

18. 调试

调试是软件开发的关键部分，通常被认为是过程中最耗时，因此也是最昂贵的部分。它使软件开发人员能够创建满足高性能、低功耗和可靠性三个关键标准的应用程序、中间件和平台软件。然而，bug 可能难以检测、重现和修复，也很难预测解决 bug 所需的时间长度。当产品交付给客户时，解决问题的成本显著增加。在很多情况下，当产品的销售时间窗口较小时，如果产品推迟，就会错失市场机会。因此，系统提供的调试工具对于任何开发人员来说都是至关重要的考虑因素。

许多使用 ARM 处理器的嵌入式系统具有有限的输入/输出设施，意味着传统的桌面调试方法（例如使用 printf()）可能不合适。在过去的系统中，开发人员可能会使用昂贵的硬件工具（如逻辑分析仪或示波器）来观察程序的行为。本书中描述的处理器是包含内存、缓存和许多其他模块的复杂片上系统 (SoC) 的一部分。可能没有在芯片外可见的处理器信号，因此无法通过连接逻辑分析仪（或类似设备）来监控行为。出于这个原因，ARM 系统通常包括专用硬件，以提供广泛的控制和观察设施以进行调试。

外部调试特性最初是在 ARMv4 架构处理器上引入的，以支持使用嵌入式和深度嵌入式处理器的开发人员，现在已经发展成为广泛的调试和跟踪特性组合。对丰富应用程序软件平台的支持，特别是对自托管调试和性能分析的支持，是最近在 ARMv6 和 ARMv7-A 架构中添加的。

ARMv8 处理器提供的硬件特性使调试工具能够对核心活动提供重大级别的控制，并以非侵入性的方式收集有关程序执行的大量数据。硬件特性有两大类，侵入性 (*invasive*) 和非侵入性 (*non-invasive*)

18.1 18.1 ARM 调试硬件

侵入式调试提供了使您能够停止程序并逐行逐行执行它们的工具，无论是在 C 源代码级别，还是单步执行汇编语言指令。这可以通过使用芯片 JTAG 引脚连接到内核的外部设备来实现，也可以通过调试监控代码来实现。

Note

JTAG 是联合测试行动组 (Joint Test Action Group) 的缩写，指的是 IEEE-1149.1 规范，最初设计用于标准化板上电子设备的测试，但现在被广泛重新用于核心调试连接。

18.1.1 18.1.1 概述

调试器提供了控制程序执行的能力，使您能够将代码运行到某个点、暂停内核、单步执行代码并恢复执行。可以在特定指令上设置断点，使调试器在内核到达该指令时进行控制。这些工作使用两种不同方法之一。软件断点通过将指令替换为 HLT 或 BRK 指令的操作码来工作。

如果连接了外部调试器并且相关的安全权限允许进入调试状态，则 HLT 指令使内核进入调试状态。AArch64 中的 BRK 指令会产生同步调试异常，但不会导致内核进入调试状态。有关调试状态的详细信息，请参阅第 18-4 页的调试事件。

显然，这些只能用于存储在 RAM 中的代码，但它们的优点是可以大量使用。调试软件必须跟踪它放置软件断点的位置，以及最初位于这些地址的操作码是什么，这样当您想要执行断点指令时，它就可以把正确的代码放回去。

硬件断点使用内核内置的比较器，并在执行到达指定地址时停止执行。这些可以在内存中的任何地方使用，因为它们不需要更改代码，但硬件提供有限数量的硬件断点单元。

调试工具可以支持更复杂的断点，例如，在地址范围内的任何指令上停止，或者只在特定的事件序列发生或硬件处于特定状态时停止。当读取或写入特定的数据地址或地址范围时，数据观察点给予调试器提供控制。这些也可以称为数据断点。例如，Cortex-A57 处理器在硬件资源中有 6 个硬件断点和 4 个观察点。请参阅调试 ID 寄存器 (*DBGDIDR*) 获取给定实现的这些值。

单步是指调试器在一段代码中移动的能力，一次一条指令。Step-In 和 Step-Over 之间的区别可以参考函数调用来解释。如果 Step-Over 执行函数调用，则整个函数将作为一个步骤执行，使能够在不想单步执行的函数之后继续。Step-In 意味着您只需单步执行该功能。

当遇到一个断点，或者单步执行时，您可以检查和更改 ARM 寄存器和内存的内容。更改内存的一种特殊情况是代码下载。调试工具通常使您能够更改代码、重新编译，然后将新映像下载到系统中。

18.1.2 18.1.2 停止或自托管调试

侵入式调试可以分为停止调试（也称为外部调试）和监控调试（也称为自托管调试）。在任何一种情况下，内核的调试逻辑都会生成一个调试事件以响应某些情况，例如命中断点。该调试事件的处理是监控调试与停止调试的区别。

在停止调试时，调试事件会导致内核进入调试状态。在调试状态下，内核停止，这意味着它不再获取指令。相反，内核在调试器的指导下执行指令，该调试器在通过 JTAG 或另一个外部接口连接的不同主机上运行。

在监视器调试中，调试事件会引发调试异常。异常必须由运行在同一内核上的专用调试监控软件处理。监视器调试以软件支持为前提。

18.1.3 18.1.3 调试事件

处理器的调试逻辑负责生成调试事件。调试事件是正在调试的进程的一部分，它导致系统通知调试器。调试事件包括断点单元等事件，该断点单元将指令的地址与存储在其寄存器中的地址进行匹配。它们可以是同步或异步的。断点、BRK 和 HLT 指令以及观察点都是同步调试事件。处理器将调试事件转换为一系列操作中的一个，即：

- 调试异常
  - 调试异常是自托管调试模型的基础
- 进入特殊的调试状态
  - 调试状态是外部调试模型的基础
- 忽略调试事件
- 挂起调试事件并稍后将其转换为操作
- 根据外部调试状态和控制寄存器 (*EDSCR*) 的设置，进入两种调试模式之一
  - 监控调试模式
  - 停止调试模式

调试事件转换为异常或进入调试状态取决于调试逻辑的配置和调试事件的类型。例如，一些调试事件永远不会导致进入调试状态，而其他事件永远不会导致调试异常。调试事件永远不会同时转换为调试异常和调试状态的入口。

有时，尽管配置了调试逻辑，处理器仍无法将调试事件转换为这些操作之一。这是因为这样做会破坏处理器的安全模型。如果处理器在安全状态下执行并且连接到它的外部调试器不受信任，则处理器不允许进入调试状态。

软件调试事件

软件调试事件是：

- 断点调试事件
- 观察点调试事件
- 软件步骤调试事件
- 软件断点指令调试事件
- Vector 捕获调试事件

除了下面介绍的外部调试使用断点和观察点的情况外，断点和观察点调试事件：

- 如果启用了当前安全状态和异常级别，则为调试异常目标生成一个调试异常。
- 只有当启用调试异常时才会生成软件步骤调试事件
- 软件断点指令调试事件总是产生调试异常

断点调试事件

地址断点通过将系统寄存器中保存的值与指令地址进行比较来生成调试事件

有些断点是上下文感知的，可以将其编程为上下文断点，与上下文 ID 或 (在非安全状态下) 虚拟机标识符 (*Virtual Machine Identifier, VMID*) 的值进行比较。

可以对断点进行编程，使其只在某些模式、异常级别和安全状态下匹配。地址中断点可以链接到上下文中断点。

处理器中的断点数目是由实现定义的 (*IMPLEMENTATION DEFINED*)。

观察点调试事件

地址观察点通过将系统寄存器中保存的值与加载和存储指令生成的数据地址进行比较来生成调试

事件。

可以对观察点进行编程，使其只在某些模式、异常级别和安全状态下匹配。地址观察点可以链接到上下文断点。

观察点也可以通过编程来匹配访问类型; 也就是说，只匹配加载，只匹配存储，或者同时匹配加载和存储。观察点与指令读取不匹配。

处理器中观察点的数量由实现定义。

软件步骤调试事件

软件步骤调试事件用于单步执行一条指令，即执行一条指令，然后将控制权返回给调试器。单步执行指令:

1. 调试器软件启用软件步骤
2. 调试器软件将 PC 设置为要执行的指令
3. 处理器执行那条指令
4. 在下一条指令上执行软件步骤异常

然而，当指令被单步执行时，可能会产生另一个同步异常。

软件断点指令调试事件

A64 指令集定义了一个软件断点指令。

```
BRK #<immediate>
```

A32 和 T32 指令集定义了软件断点指令。

```
BKPT #<immediate>
```

软件断点指令生成无法屏蔽的同步调试异常。

Vector 捕获调试事件

Vector 捕获调试事件仅在 AArch32 阶段 1 转换机制中生成，并且仅生成调试异常。Vector 捕获异常仅从 AArch32 状态生成。

18.1.4 18.1.4 调试事件

在任何标准接口可用于调试之前，复杂系统需要其大部分硬件和软件能够正常工作。在不依赖被调试系统的情况下对系统进行调试非常重要。为此，需要可靠的外部调试，即硬件辅助、运行控制调试和跟踪功能。所有这些都可以在不需要在平台上运行软件的情况下进行控制，但通常在产品设计周期的早期就需要这样做。

自托管工具通常需要软件支持层，这使得调试软件的某些部分变得困难，或者使得调试对于诊断某些类型的 bug 来说过于侵入性。低成本的外部调试接口，如串行线调试 (SWD)，还有助于扩展具有外部调试吸引力的应用程序的范围。更多信息请参阅第 18-9 页的 CoreSight。

18.1.5 18.1.5 停止调试模式

在停止调试模式下，调试事件会导致核进入调试状态，核停止并与系统的其余部分隔离。这意味着调试器显示核所看到的内存，并且内存管理和缓存操作的效果变得可见。在调试状态下，核从程序计数器指示的位置停止执行指令，而是通过外部调试接口进行控制。这使得外部代理 (如调试器) 能够查询核心上下文并控制所有后续指令的执行。可以修改核状态和系统状态。因为核已经停止，所以在调试器重新启动执行之前不会处理任何中断。

停止调试的基本原则与 ARMv7-A 保持不变。那是：

- 当编程为停止调试时，调试事件会导致进入一个特殊的调试状态。
- 在调试状态下，核不会从内存中获取指令，而是从特殊的指令传输寄存器中获取指令。
- 数据传输寄存器用于在主机和目标之间移动寄存器和内存内容。

18.1.6 18.1.6 自托管调试

我们已经看到了 ARM 架构如何为外部调试器提供了广泛的特性。在核 (驻留在目标系统上的调试监视器) 上运行的软件也可以使用其中的许多功能。监视系统可能很便宜，因为它们可能不需要任何额外的硬件。但是，它们会占用系统中的内存空间，并且只有在目标系统本身实际运行时才能使用。对于一个至少不能正确引导的系统来说，它们没有什么价值。

为了帮助开发人员创建应用程序，平台需要经常 (至少部分地) 在应用程序处理器本身上运行的开发工具，而不是需要昂贵的接口硬件来连接第二台主机。ARMv8-A 体系结构对这种自托管调试形式的体系结构支持进行了改进。在现有的桌面平台上，自托管是软件开发的流行方法。

18.1.7 18.1.7 调试 Linux 程序

Linux 是一个多任务操作系统，其中每个进程都有自己的进程地址空间，并配有私有的转换表映射。这使得调试某些类型的问题相当棘手。

广义地说，在 Linux 系统中有两种不同的调试方法。

Linux 应用程序通常使用运行在目标机上的 GDB 调试服务器进行调试，通常通过以太网与主机进行通信。在调试会话发生时，内核继续正常运行。此调试方法不提供对内置硬件调试工具的访问。目标系统永久处于运行状态。服务器接收来自主机调试器的连接请求，然后接收命令并将数据返回给主机。

主调试器向 GDB 服务器发送加载请求，GDB 服务器通过启动一个新进程来运行被调试的应用程序来进行响应。在执行开始之前，它使用系统调用 ptrace() 来控制应用程序进程。来自这个进程的所有信号都被转发到 GDB 服务器。发送到应用程序的信号会发送到 GDB 服务器，GDB 服务器可以处理该信号，或者将其转发给正在调试的应用程序。

要设置断点，GDB 服务器会在代码中的所需位置插入生成 SIGTRAP 信号的代码。执行此操作时，将调用 GDB 服务器，然后可以执行经典的调试器任务，例如检查调用堆栈信息、变量或寄存器内容。

18.1.8 18.1.8 调试 Linux 内核

对于内核调试，使用了一个基于 jtag 的调试器。当执行断点时，系统将停止。这是检查诸如设备驱动程序加载、错误操作或内核引导失败等问题的最简单方法。另一个常用方法是通过 printk() 函数调用。strace 工具显示有关用户系统调用的信息。

Kgdb 是 Linux 内核的源代码级调试器，可在单独的机器上与 GDB 一起使用，并支持检查堆栈跟踪和内核状态（例如 PC 值、计时器内容和内存）视图。device/dev/kmem 启用对内核内存的运行时访问。

当然，可以使用支持 Linux 的 JTAG 调试器来调试线程，通常只能停止所有进程；不能停止单个线程或进程而让其他线程或进程继续运行。可以为所有线程设置断点，也可以仅在特定线程上设置断点。

由于内存映射取决于哪个进程处于活动状态，因此通常只能在映射特定进程时设置软件断点。ARM DS-5 调试器能够使用 gdbserver 调试 Linux 应用程序并使用调试 Linux 内核和 Linux 内核模块 JTAG。DS-5 调试器的调试和跟踪功能将在下一节中描述。

18.1.9 18.1.9 调用栈

应用程序代码使用调用堆栈来传递参数、存储本地数据和存储返回地址。每个函数压入堆栈的数据被组织成一个堆栈帧。当调试器停止内核时，它可能能够分析堆栈上的数据，为您提供调用堆栈，即导致当前情况的函数调用列表。这在调试时非常有用，因为它使您能够确定应用程序达到特定状态的原因。

要重建调用堆栈，调试器必须能够确定堆栈上的哪些条目包含返回地址信息。如果代码构建时包含了调试器信息 (*DWARF* 调试表)，或者遵循应用程序压入堆栈的帧指针链，则该信息可能包含在调试器信息 (DWARF 调试表) 中。为此，必须构建代码来使用框架指针。如果这两种类型的信息都不存在，则无法构造调用堆栈。

在多线程应用程序中，每个线程都有自己的堆栈。因此，调用堆栈信息只与正在检查的特定线程相关。

18.1.10 18.1.10 半主机调试

半主机是一种机制，它使运行在 ARM 目标上的代码能够使用运行调试器的主机上提供的工具。

例如，键盘输入、屏幕输出和磁盘 I/O。例如，您可以使用这种机制来启用 C 库函数，例如 printf() 和 scanf()，以使用主机的屏幕和键盘。开发硬件通常没有完整的输入和输出设施，但是半主机使主机计算机能够提供这些设施。

半主机是通过一组定义好的软件指令来实现的，这些指令会产生一个异常。应用程序调用适当的半主机调用，然后调试代理处理异常。调试代理程序提供与主机所需的通信。

对于 ARMv8 处理器，半主机使用的规范与实现 ARMv7 的处理器不同。DS-5 调试器通过拦截 AArch64 中的 HLT 0xF000 来处理半主机。

当然，在开发环境之外，运行在主机上的调试器通常不会连接到系统。因此，开发人员有必要重新定位任何使用半主机的 C 库函数，例如，通过使用 fputc()。这将涉及将使用 SVC 调用的库代码替换为可以输出字符的代码。

18.2 18.2 ARM 跟踪硬件

非侵入式调试，可以在执行时观察核行为。尽管有不同类型的非侵入式调试，但本节将特别介绍跟踪和跟踪硬件。可以记录执行的内存访问（包括地址和数据值）并生成程序的实时跟踪，查看外设访问、堆栈和堆访问以及对变量的更改。对于许多实时系统，不可能使用侵入式调试方法。例如，考虑一个引擎管理系统，虽然您可以在特定点停止核，但引擎仍在移动，您将无法进行有用的调试。即使在实时要求不那么繁重的系统中，跟踪也非常有用。

跟踪通常由连接到内核的内部硬件块提供。这称为嵌入式跟踪宏单元 (*Embedded Trace Macrocell, ETM*)，是大多数基于 ARM 处理器的系统的一部分。在某些情况下，每个内核有一个 ETM。片上系统设计人员可以从他们的硅片中省略这个模块以降低成本。这些模块观察但不影响核心行为，并且能够监控指令执行和数据访问。

捕获跟踪有两个主要问题。首先是在当前非常高的核时钟速度下，即使是几秒钟的操作也可能意味着数万亿个执行周期。显然，要理解这些信息量是极其困难的。

第二个相关问题是当前的核每个周期可能执行一个或多个 64 位高速缓存访问，并且记录数据地址和数据值可能需要很大的带宽。这带来了一个问题，通常，芯片上可能只提供几个引脚，并且这些输出可以以比核时钟频率低得多的速率切换。如果核以 1GHz 的速度每个周期产生 100bit 的信息，而芯片以 200MHz 的速度只能输出 4bit 的 trace，那就有问题了。

为了解决后一个问题，跟踪宏单元试图压缩信息以减少所需的带宽。然而，处理这些问题的主要方法是控制跟踪块，以便只收集选定的跟踪信息。例如，您可能只跟踪执行，而不记录数据值。

此外，通常将跟踪信息存储在片上内存缓冲区 (嵌入式跟踪缓冲区 (*Embedded*

Trace Buffer, ETB)) 中。这减轻了快速获取芯片外信息的问题，但在硅面积 (因此芯片的价格) 方面有额外的成本，并对可捕获的跟踪量提供了固定的限制。

ETB 以循环方式存储压缩的跟踪信息，不断捕获跟踪信息直到停止。ETB 的大小因芯片实现而异，但 8 或 16KB 的缓冲区通常足以容纳数千行程序跟踪。当程序失败时，如果启用跟踪缓冲区，您可以看到一部分程序历史记录。使用此程序历史记录，可以更轻松地回顾您的程序以查看故障点之前发生的情况。这对于调查通过需要停止和启动内核的传统调试方法难以识别的间歇性和实时故障特别有用。使用硬件跟踪可以显着减少查找这些故障所需的时间，因为跟踪准确地显示了执行了什么、时间是什么以及发生了哪些数据访问。

18.2.1 18.2.1 CoreSight

ARM CoreSight™ 技术扩展了 ETM 提供的功能。同样，它在特定系统中的存在和能力由系统设计者定义。CoreSight 提供了许多极其强大的调试工具。它支持调试多核系统（非对称和 SMP），可以共享调试访问和跟踪引脚，并具有完全控制权哪些核在哪些时间被追踪。嵌入式交叉触发机制使工具能够以同步方式控制多个核，例如，当一个核遇到断点时，所有其他内核也将停止。

分析工具可以使用这些数据来显示程序在哪里花费了时间，以及存在哪些性能瓶颈。代码覆盖工具可以使用跟踪数据来提供调用图的探索。操作系统感知的调试器可以使用跟踪，在某些情况下，还可以使用附加的代码插装来提供高级的系统上下文信息。下面是对一些可用的 CoreSight 组件的简要描述

调试访问端口 (DAP)

DAP 是 ARM CoreSight 系统的可选部分。并非每个设备都包含 DAP。它使外部调试器能够直接访问系统的内存空间，而无需将内核置于调试状态。在没有 DAP 的情况下读取或写入内存可能需要调试器停止内核并让它执行加载或存储指令。DAP 使外部调试工具可以访问系统中的所有 JTAG 扫描链，从而调试和跟踪可用内核和其他组件的配置寄存器。

嵌入式交叉触发 (ECT)

ECT 模块是一个 CoreSight 组件，可以包含在 CoreSight 系统中。它的目的是将系统中多个设备的调试能力链接在一起。例如，可以有两个相互独立运行的核心。当您在一个核上运行的程序上设置断点时，如果能够指定当该核心断点处停止时，另一个核也必须停止 (无论它当前执行的是什么指令)，这将是很有用的。ECT 内部的交叉触发矩阵和接口使调试状态和控制信息能够在核和跟踪宏单元之间传播。

在 ARMv8 处理器系统中始终需要交叉触发块，因为它提供了在处理器进入暂停模式后重新启动处理器执行的唯一方法。

CoreSight 串行线

CoreSight 串行线调试提供了一个使用调试访问端口 (*Debug*

Access Port, DAP) 的 2 针连接，在功能上相当于一个 5 针 JTAG 接口。

系统跟踪宏单元 (STM)

这为多个核 (和进程) 提供了一种执行 printf() 样式调试的方法。运行在系统中任何主机上的软件都能够访问 STM 通道，而无需知道其他人使用了它，只需使用非常简单的代码片段。这使得内核和用户空间代码的带时间戳的软件检测成为可能。时间戳信息给出了与以前事件相关的增量，可能非常有用。

跟踪内存控制器 (TMC)

如前所述，在已封装的 IC 上添加额外的引脚可以显著增加其成本。在一个设备上有多个核 (或能够生成跟踪信息的其他块) 的情况下，很可能因为经济原因而无法提供多个跟踪端口。CoreSight 跟踪内存控制器是一个跟踪漏斗，能够将多个跟踪源合并到系统内存中的单个总线中。

提供的控件用于在这些多个输入源之间启用、排序和选择。可以使用专用的跟踪端口，通过 JTAG 或串行线接口，或通过重用 SoC 的 I/O 端口，将跟踪信息导出芯片外。跟踪信息可以存储在 ETB 或系统内存中。

18.3 18.3 DS-5 调试和跟踪

DS-5 调试器为使用基于 ARM 架构的处理器在硬件目标和模型上调试应用程序提供了一个强大的工具。您可以完全控制执行流程，以便快速隔离和纠正错误。

DS-5 调试器提供了广泛的调试功能，例如：

- 加载镜像和符号
- 运行镜像
- 断点和观察点
- 源和指令级步进
- 控制变量和寄存器值
- 查看调用堆栈
- 支持处理异常和 Linux 信号
- 多线程 Linux 和 Android 应用程序的调试
- Linux、内核和 Android 模块的调试，引导代码和内核移植
- 应用程序倒带，允许您通过 Linux 和 Android 应用程序向后和向前调试

调试器支持一组全面的 DS-5 调试器命令，这些命令可以在 Eclipse IDE、脚本文件或命令行控制台中执行。此外，有一小部分 CMM 风格的命令足以运行目标初始化脚本。

DS-5 调试器支持使用 JTAG 进行裸机调试、使用 gdbserver 进行 Linux 应用程序调试、使用 JTAG 进行 Linux 内核调试和内核模块调试。对裸机 SMP 系统的调试和跟踪支持，包括交叉触发和依赖于内核的视图和断点、PTM 跟踪以及高达 4 GB 的 DSTREAM 跟踪。此支持将在以下部分中描述。

此外，DS-5 调试器支持 ARM CoreSight ETM, PTM, ETB 和 STM，以提供非侵入性的程序跟踪，使您能够审查指令 (和相关的源代码)，因为他们已经发生。它还提供了调试时间敏感问题的能力，否则传统的侵入式步进技术将无法解决这些问题。

18.3.1 18.3.1 使用 DS-5 调试 Linux 或 Android 应用程序

调试 Linux 或 Android 应用程序需要在目标上安装和运行诸如 gdbserver 之类的调试服务器。您可以使用 TCP 或串行连接到运行操作系统的目标，它可以是真正的目标硬件或软件模型。

DS-5 调试器负责下载并连接到调试服务器。开发人员只需指定平台和 IP 地址。这将使用多个应用程序和一个终端的复杂任务减少到 IDE 中的几个步骤。

18.3.2 18.3.2 调试 Linux 内核模块

Linux 内核模块提供了一种扩展内核功能的方法，通常用于设备和文件系统驱动程序等。模块既可以内置到内核中，也可以编译为可加载模块，然后在开发过程中动态地从正在运行的内核中插入和删除，而无需频繁地重新编译内核。

但是，有些模块必须内置到内核中，不适合动态加载。内置模块的一个示例是在内核引导期间必需的，并且必须在安装根文件系统之前可用。

您可以使用 DS-5 调试器在模块中设置源代码级断点，前提是调试信息已加载到调试器中。在模块插入内核之前尝试在模块中设置断点会导致断点被挂起。

调试模块时，必须确保目标上的模块与主机上的模块相同。代码布局必须相同，但目标上的模块不必包含调试信息。

内置模块

要使用 DS-5 调试器调试已内置到内核中的模块，过程与调试内核本身相同

1. 将内核与模块一起编译
2. 将内核镜像加载到目标上
3. 将带有调试信息的相关内核镜像加载到调试器中
4. 像调试任何其他内核代码一样调试模块

可加载模块

调试可加载内核模块的过程更为复杂。在 Linux 终端 shell 中，可以使用 insmod 和 rmmod 命令插入和删除模块。内核和可加载模块的调试信息都必须加载到调试器中。当您插入和删除模块时，DS-5 调试器会自动为调试信息和现有断点解析内存位置。

为此，DS-5 调试器拦截内核内的调用，以插入和删除模块。当调试器停止内核来查询各种数据结构时，每个操作都会有一个小的延迟。

18.3.3 18.3.3 使用 DS-5 调试 Linux 内核

要调试 Linux 内核模块，可以使用 DSTREAM 等调试硬件代理连接主机工作站和运行的目标。为了能够在源代码级别调试内核，需要将包含调试符号的 vmlinux 文件加载到调试器中。

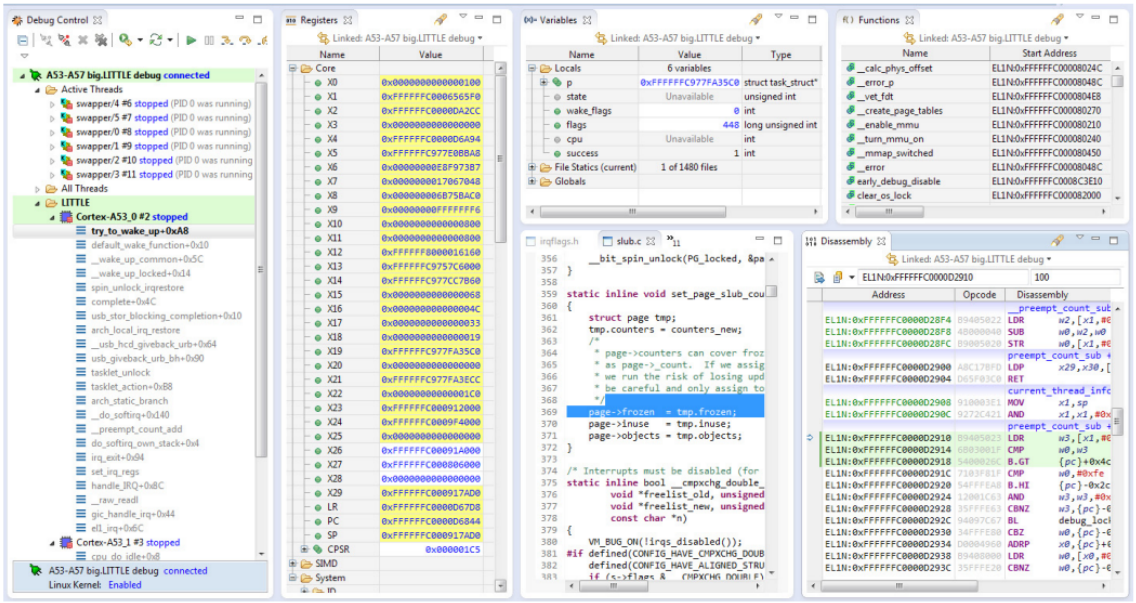

Figure 18-1 Debugging a kernel using DS-5 image-20220427225645243

18.3.4 18.3.4 使用 DS-5 调试多线程应用程序

DS-5 调试器使用调试器变量 $thread 跟踪当前线程。可以在打印命令或表达式中使用此变量。线程显示在调试控制视图中，具有调试器使用的唯一 ID 和操作系统的唯一 ID。例如：

```
Thread 1 (OS ID 1036)
```

其中线程 1 是调试器使用的 ID，操作系统 ID 1036 是操作系统的 ID

为每个线程维护单独的调用堆栈，所选堆栈帧以粗体文本显示。DS-5 Debug 透视图中的所有视图都与所选的堆栈帧相关联，当选择另一个堆栈帧时，这些视图将被更新。

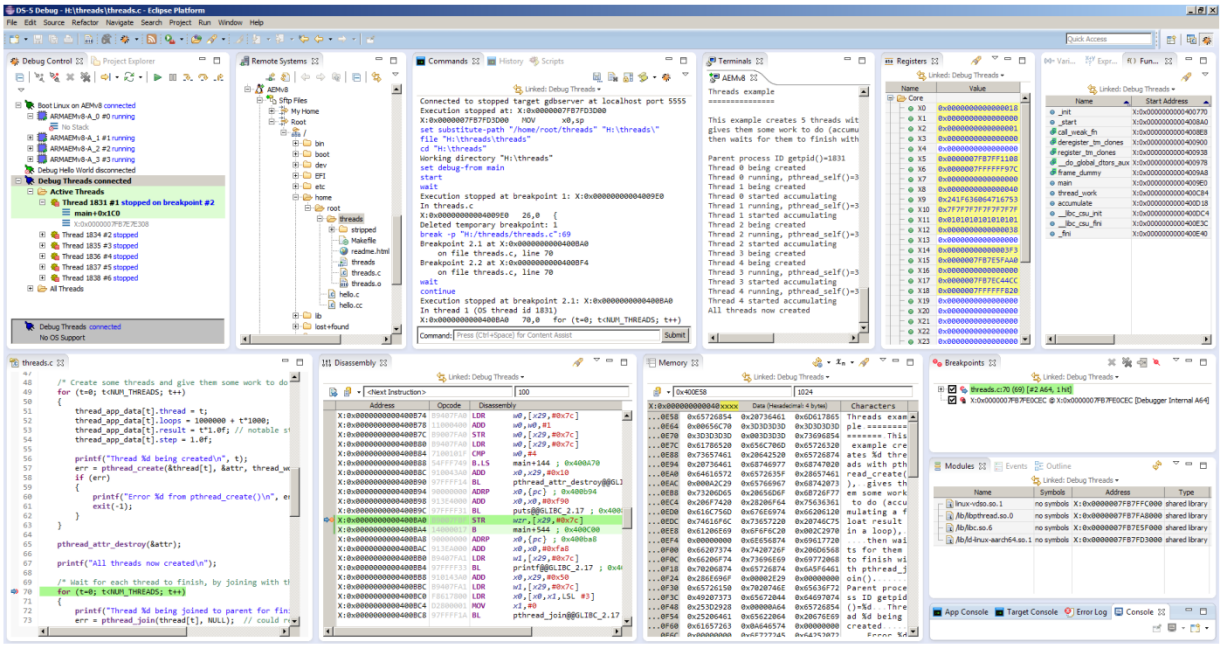

Figure 18-2 Threading call stacks in the DS-5 Debug Control view image-20220427230110146

18.3.5 18.3.5 调试共享库

共享库使您的应用程序的某些部分能够在运行时动态加载。您必须确保目标上的共享库与主机上的共享库相同。代码布局必须相同，但目标上的共享库不必包含调试信息。

您可以在共享库中设置标准执行断点，但必须在应用程序加载该库并将调试信息加载到调试器中之后才能设置。但是，挂起断点使您能够在应用程序加载共享库之前在共享库中设置执行断点。

当加载一个新的共享库时，DS-5 调试器会重新评估所有挂起的断点，那些具有它可以解析的地址的断点会被设置为标准执行断点。未解析的地址仍然作为待处理的断点。

当应用程序卸载库时，调试器自动将共享库中的任何断点更改为挂起的断点。

18.3.6 18.3.6 DS-5 中的跟踪支持

DS-5 使您能够对您的应用程序或系统执行跟踪。您可以实时捕获历史的、非侵入性的指令跟踪。跟踪是一个强大的工具，使您能够在系统全速运行时调查问题。这些问题可能是间歇性的，并且很难通过需要启动和停止内核的传统调试方法来识别。在尝试识别潜在瓶颈或改进应用程序的性能关键区域时，跟踪也很有用。

跟踪视图

当跟踪被捕获时，调试器从跟踪流中提取信息并将其解压缩以提供已执行代码的完整反汇编和符号。

此视图显示了一个图形化导航图，其中显示了带有导航时间线的函数执行情况。此外，反汇编跟踪显示带有关联地址的函数调用，如果选择，还显示指令。在图表中单击特定的时间可以同步拆卸视图

在图表的左栏中，显示了总跟踪的每个函数的百分比。例如，如果总共执行了 1000 条指令，其中 300 条与 myFunction() 关联，那么这个函数显示 30%。

在导航时间轴中，颜色编码是一个热图，显示执行的指令以及每个函数在每个时间轴中执行的指令数量。深红色表示指令更多，浅黄色表示指令更少。然而，在 1:1 的比例下，颜色方案将内存访问指令显示为深红色，分支指令显示为中等橙色，所有其他指令显示为浅绿色。

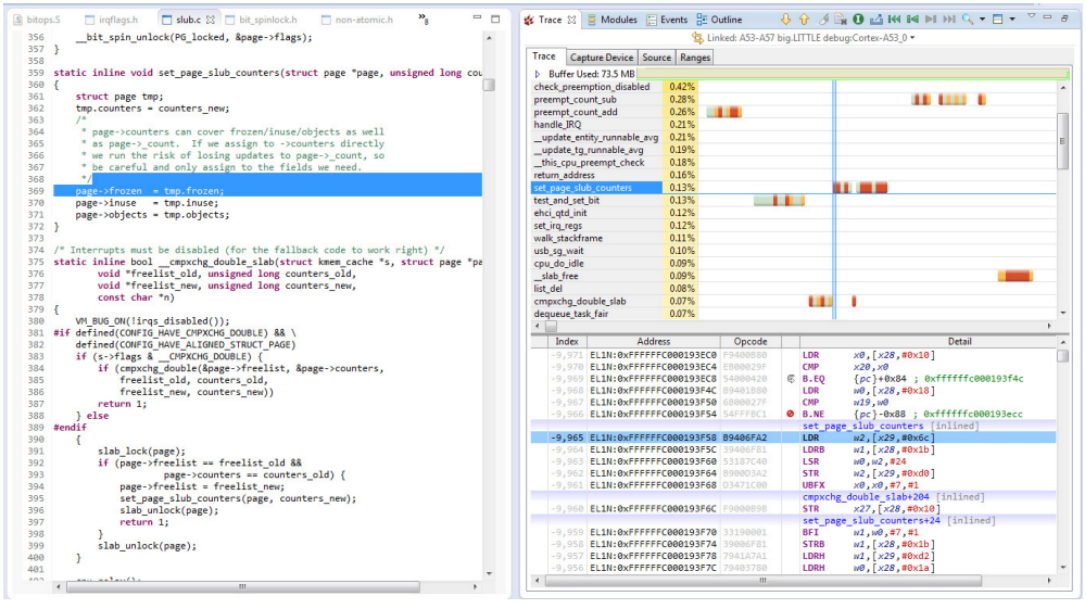

Figure 18-3 DS-5 Debugger Trace view image-20220427231703166

基于跟踪的分析

根据从目标接收到的跟踪数据，DS-5 调试器可以生成带有信息的时间线图表，以帮助开发人员快速了解他们的软件如何在目标上执行以及哪些功能使用内核最多。时间线提供各种缩放级别，并且可以根据每个时间单位的指令数量显示热图，或者在最高分辨率下，提供按每组指令的典型延迟进行颜色编码的每条指令可视化。

CHAPTER

NINETEEN

19 ARMV8 模型

平台模型，例如本章中描述的模型，可以在不需要实际硬件的情况下进行软件开发。软件模型从程序员的角度提供处理器和设备的模型。模型的功能行为等同于真实硬件。

为了获得快速的模拟执行速度，牺牲了绝对时序精度。这意味着您可以使用 PV 模型来确认软件功能，但您不能依赖循环计数、低级组件交互或其他硬件特定行为的准确性。

ARMv8-A 基础平台中的处理器并非基于任何现有的处理器设计，但仍符合 ARMv8-A 架构规范。ARMv8-A 基础平台使用 ARM 快速模型技术，是 ARM 处理器建模解决方案套件的一部分。这些建模解决方案可在通过 ARM 快速模型交付的模型组合中使用。

在基本平台 FVP 中建模的处理器可以配置为类似于 Cortex-A53 和 Cortex-A57 处理器。

19.1 19.1 ARM 快速模型

Fast Models 是一个用于创建以高仿真速度执行的虚拟平台模型的环境。它们提供对基于 ARM 的系统的访问，这些系统适用于在芯片可用之前进行早期软件开发。与 ARM Development Studio 5 (DS-5) 结合使用，FastModels 可以帮助开发人员在整个开发周期中调试、分析和优化他们的应用程序。

这些虚拟平台可以方便地分发给软件开发人员进行早期软件开发，而无需昂贵的开发板。他们：

- 每秒执行多达 2.5 亿条 ARM 指令，与实际硬件相当
- 具有针对应用程序和固件以及早期驱动程序开发量身定制的性能和准确性
- 快速启动 Linux 和 Android 等操作系统
- 提供基于 ARM 处理器的子系统的 SystemC Transaction Level Messaging (TLM) 2.0 导出
- 拥有功能准确的 ARM 指令集模型，针对 ARM 处理器设计进行了全面验证
- 对先进的 ARM 技术进行建模，例如缓存、MMU、LPAE、虚拟化、TrustZone 和 VFP
- 模拟外围设备，例如以太网、LCD、键盘和鼠标

生成的平台配备了组件架构调试接口 (CADI)，可以独立运行或从合适的调试器运行。Fast Models 自动为独立平台和集成平台生成所需的接口。

快速模型可用于许多 ARM 处理器和系统控制器，以及经典 ARM 处理器和 CoreLink 系统控制器。

快速模型只关注处理器上运行的程序的准确性。他们没有尝试准确地模拟总线事务，也没有准确地模拟指令时序。整个模拟具有非常准确的时序概念，但代码转换 (CT) 处理器并未声称以类似设备的时序调度指令。

快速模型试图对硬件进行准确建模，但在执行速度、准确性和其他标准之间存在折衷。在某些情况下，处理器型号可能与硬件不匹配。

快速模型可以：

- 准确地模拟指令
- 正确执行架构上正确的代码

但是，快速模型不能：

- 验证硬件
- 对所有架构上不可预测的行为进行建模
- 模型周期计数
- 对时序敏感的行为进行建模
- 模型 SMP 指令调度
- 测量软件性能
- 模拟 bus 线路

快速模型旨在准确地反映系统程序员的观点。软件能够检测硬件和模型之间的差异，但这些差异通常取决于未精确指定的行为。例如，可以检测指令和总线事务的准确时序、推测性预取的影响和缓存牺牲品选择的差异。某些类别的行为被指定为 UNPREDICTABLE 并且这些情况可以被软件检测到。依赖于这种行为的程序，即使是无意的，也不能保证在任何设备或快速模型上可靠地工作。利用此行为的程序可能会在硬件和模型之间以不同的方式执行。

通常，处理器在模拟时间的同一点发出一组指令（一个量程），然后在执行下一个量程之前等待一段时间。时序安排使得处理器平均每个时钟节拍一条指令。

这样做的结果是：

- 在模型上运行的软件的感知性能不同于现实世界的软件。特别是，内存访问和算术运算都需要大量时间
- 程序可能能够检测处理器的量化执行行为，例如通过高分辨率计时器进行轮询

19.1.1 19.1.1 哪里获取到 ARM 快速模型

有关如何购买 ARM 快速模型的详细信息，请访问 http://www.arm.com/fastmodels

19.2 19.2 ARMv8-A 基础平台

ARMv8-A 基础平台是 ARMv8-A 架构的测试平台。它是一个简单的平台模型，能够运行裸机半托管应用程序并启动一个完整的操作系统。

它作为平台模型提供，从命令行配置仿真并使用平台中的外围设备进行控制。

基础平台有：

- 包含 1-4 个内核的 ARMv8-A 处理器集群模型，实现：
- 所有异常级别的 AArch64
- EL0 和 EL1 支持 AArch32
- 所有异常级别的小端和大端
- 通用计时器
- 自托管调试
- GICv2 和可选的 GICv3 内存映射处理器接口和分配器
- 8GB 内存
  - 该平台可模拟高达 8GB 的 RAM
  - 要模拟具有 4GB RAM 的系统，您需要至少具有 8GB RAM 的主机
  - 要模拟具有 8GB RAM 的系统，您需要一台至少具有 12GB RAM 的主机
- 四个 PL011 UART 连接到 xterms
- 平台外设包括实时时钟、看门狗定时器、实时定时器和电源控制器
- 安全外围设备，包括受信任的看门狗、随机数生成器、非易失性计数器和根密钥存储
- 连接到主机网络资源的网络设备模型
- 在主机上实现为文件的块存储设备
- 带有 LED 和开关的小型系统寄存器块，使用 Web 服务器可见
- 一个简单的网络界面来指示模型的状态。请参阅网页页面第 19-13 页
- 主机文件系统访问实现为计划 9 文件系统

缓存被建模为无状态并且没有写缓冲区。这在数据端产生了完美的内存一致性效果。指令端有一个可变大小的预取缓冲区，因此需要在目标代码中使用正确的屏障才能正确运行。

除非集群中的所有内核都处于等待中断 (WFI) 状态，否则平台会尽可能快地运行或等待异常 (WFE)，在这种情况下，平台空闲，直到发生中断或外部事件发生。

基础平台已经过修订，以支持 ARM 可信基础系统架构 (TBSA) 和服务器基础系统架构 (SBSA)。添加了许多外围设备，并对内存映射进行了相应的更改。它也被更新以更紧密地与 Versatile Express 基板和 ARM 快速模

型中的外设保持一致。在适当的情况下，原始 Foundation 模型现在称为 Foundation v1 或 Foundation v2 ，而 Foundation Platform 是 Foundation v9.1。

针对以前版本的平台编写的软件将使用默认的 --no-gicv3 配置选项在平台上未经修改地工作。只有使用早期 RAM 块的软件可能需要一些调整。

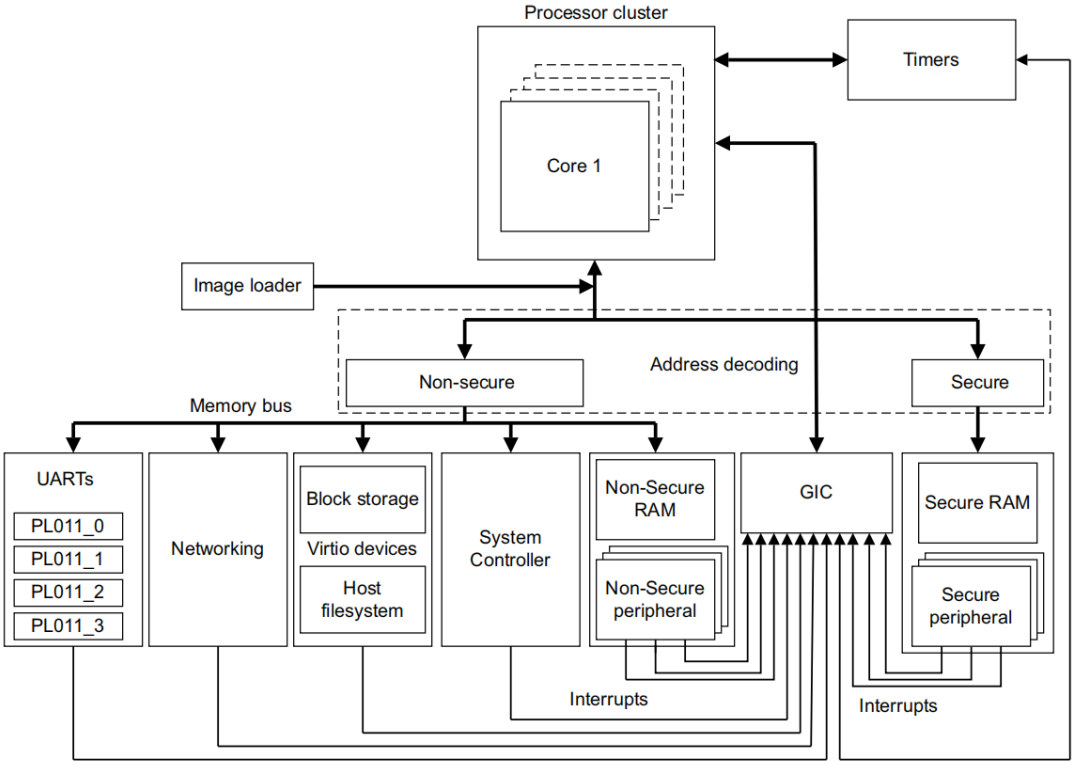

image-19-1-1

Figure 19-1 Block diagram of ARMv8-A Foundation Platform

地址解码块的行为取决于 --secure-memory 是否使用命令行选项。

该平台提供以下类型的网络支持：

基于 NAT,IPV4

- 基于 IPv4 的网络通过使用针对以前版本的平台编写的软件将使用默认的 --no-gicv3 配置选项在平台上未经修改地工作。只有使用早期 RAM 块的软件可能需要一些调整。

桥接

- 桥接网络需要设置以太网桥接设备，以在主机上的以太网端口和平台提供的网络接口之间进行桥接。这通常需要管理员权限。有关更多信息，请参阅 Linux bridge-utils 包中的文档。

19.2.1 19.2.1 基金会平台的局限性

以下限制适用于 ARMv8-A 基础平台：

- 写缓冲区没有建模
- 并非在每个指令边界处都发生中断
- 缓存被建模为无状态
- 没有组件架构调试接口 (CADI)、CADI 服务器、跟踪或其他插件支持
- 不支持 Thumb2EE
- 不支持 ARMv8 加密扩展

19.2.2 19.2.2 软件要求

运行 ARMv8-A 基础平台所需的软件如下：

操作系统

- 适用于 64 位 Intel 架构的 Red Hat Enterprise Linux 5.x 版本
- 适用于 64 位 Intel 架构的 Red Hat Enterprise Linux 6.x 版本
- 适用于 64 位 Intel 架构的 Ubuntu 10.04 或更高版本

目前尚不支持在其他操作系统上运行该平台。但是，该模型应该在任何最新的 x86 64 位 Linux 操作系统上运行，提供 glibc v2.3.2 或更高版本，并且存在 libstdc++ 6.0.0 或更高版本

UART 输出

要使通用异步接收器/发送器 (UART) 输出可见，必须在主机上安装 xterm 和 telnet ，并在 PATH 中指定

19.2.3 19.2.3 从哪里获得 ARM 基础平台

ARMv8-A 基础平台是一个开源平台，可以从 http://www.arm.com/fvp。

19.2.4 19.2.4 验证安装

Foundation Platform 仅作为预构建的平台二进制文件提供。安装目录结构如下：

Icon	State label	Description
	UNKNOWN	Run status unknown, that is, simulation has not started.
	RUNNING	The core is running, is not idle, and is executing instructions.
	HALTED	An external halt signal is asserted.
	STANDBY_WFE	The last instruction executed was WFE, and standby mode has been entered.
	STANDBY_WFI	The last instruction executed was WFI and standby mode has been entered.
	IN_RESET	An external reset signal is asserted.
	DORMANT	Partial core power down.
	SHUTDOWN	Complete core power down.

image-19-2-2

Figure 19-2 Installed files

在哪里:

例子

包括运行示例程序描述的示例程序的 C 版本和.axf 文件。它还包括 Makefile 和设备树 Foundation_Platform.dts 的示例源代码。

基金会 _ 平台

ARMv8-A 基础平台可执行文件。

libMAXCOREInitSimulationEngine.so

平台所需的辅助库。

libarmctmodel.so

代码翻译库。

FoundationPlatform_Readme.txt

用户指南的简短摘要。

DUI0677E_foundation_platform_ug.pdf

文档。

LES-PRE-20164_V1_-_Foundation_Platform.txt

最终用户许可协议文本。

19.2.5 19.2.5 运行示例程序

提供的示例程序可用于确认 ARMv8-A 基础平台工作正常。

使用以下命令行运行平台：

```
./Foundation_v8 --image hello.axf
```

添加 --quiet 以抑制除示例程序的输出之外的所有内容。

它应该打印类似于以下内容的输出：

```
terminal_0: Listening for serial connection on port 5000
terminal_1: Listening for serial connection on port 5001
terminal_2: Listening for serial connection on port 5002
terminal_3: Listening for serial connection on port 5003
Simulation is started
Hello, 64-bit world!
Simulation is terminating. Reason: Simulation stopped
```

该示例演示了平台正确初始化、加载和执行示例程序，以及半主机调用以打印输出并停止平台工作。

19.2.6 19.2.6 示例程序故障排除

- 如果您尝试在 32 位 Linux 主机上运行示例程序，则会出现类似于以下内容的错误：

```
./Foundation_Platform: /lib64/ld-linux-x86-64.so.2: bad ELF interpreter: No
suchfile or directory
```

- 如果您的系统上未安装 libstdc++ ，则在启动时会出现以下错误：

```
./Foundation_Platform: error while loading shared libraries: libstdc++.so.6:
cannot open shared object file
```

- 如果您的系统 glibc 太旧，或者您的 libstdc++ 太旧，您会收到以下消息：

```
./Foundation_Platform: /usr/lib64/libstdc++.so.6: version GLIBCXX_3.4 not
found (required by Foundation_Platform)

./Foundation_v8: /lib64/libc.so.6: version GLIBC_2.3.2 not found (required by
Foundation_Platform)

./Foundation_Platform: /lib64/libc.so.6: version GLIBC_2.2.5 not found (re-
quired by Foundation_Platform)
```

libstdc++ 和 glibc 通常是核心操作系统安装的一部分。

19.2.7 19.2.7 内核

ARMv8 基础平台不提供内核。但是，可以使用 AArch64 (ARM64) 补丁。使用这些时，主线内核应该在 ARMv8 基础平台中正常运行，而不需要额外的补丁。您将需要一个跨工具链来构建 ARM64 内核。默认内核配置在 Foundation Platform 中工作，无需任何更改。

19.2.8 19.2.8 配置内核命令行

在大多数 Linux 系统上配置内核命令行的常用方法是使用引导加载程序。由于 Foundation Platform 直接引导内核而不调用中间引导加载程序，因此必须以不同的方式执行配置。构建内核映像后，必须添加引导包装器。这以平台可以使用的方式向内核映像添加了额外的配置，包括：

- 内核命令行
- 表示平台硬件配置的扁平设备树 (FDT) blob
- initramfs 映像（如果需要）

与内核一起提供了一个示例引导包装器源代码。通过编辑包含的 Makefile 来配置它。有许多设置可以更改，但通常使用它们中的大多数无需任何更改即可工作。最有可能对配置有用的设置是：

INITRD_FLAGS

将此设置为 -DUSE_INITRD 以附加 initramfs 文件系统。

FILESYSTEM

如果 INITRD_FLAGS 设置为 -DUSE_INITRD，则要使用的 initramfs 映像的路径。

BOOTARG

内核命令行

在配置后使用 make ，引导包装器会将内核及其配置链接到一个文件中，供模型使用，通常是 linux-system.axf

19.2.9 19.2.9 根文件系统的选择

将 initramfs 与内核一起使用是一个简单的选项，但会使用更多内存，因为它还必须存储文件系统。对于更大、更有用的文件系统，ARM 建议使用虚拟块设备（使用 --block-device 选项，请参阅第 19-12 页的命令行概述）或 NFS 根。如果您在平台中交叉构建程序和测试，使用 NFS 根是更方便的途径，但需要更复杂的配置。

19.2.10 19.2.10 根文件系统设置块设备映像

此方法假定您已经为要添加到根文件系统的文件创建了一个压缩文件系统。

根文件系统设置块设备映像：

1. 使用以下命令创建一个正确大小的空文件（文件名）：

   ```
   \# dd if=/dev/zero of=<filename> bs=1M count=<number of megabytes>
   ```

2. 在该文件中创建一个文件系统，使用：

   ```
   \# mkfs.ext4 <filename>
   ```

3. 使用以下命令在主机系统上挂载文件系统：

```
\# mkdir /mnt/AArch64

\# mount -o loop <filename> /mnt/AArch64
```

1. 使用以下命令将文件系统提取到设备上：

   ```
   \# cd /mnt/AArch64

   \# zcat /path/to/filesystem.cpio.gz | cpio -divmu --no-absolute-filenames

   \# cd /
   ```

2. 使用以下方法卸载设备：

```
\# umount /mnt/AArch64
```

1. 您的块设备映像可以使用了。要在平台中使用它，请将引导包装器 Makefile 中的根设备配置为 root=/dev/vda 并重建引导包装器。

- 警告

  对于大多数网络协议，使用网络地址转换 (NAT) 或用户模式网络就足够了。但是，如果您尝试使用与平台在同一台机器上运行的 NFS 服务器，则用户模式网络将不适用于此。相反，使用桥接配置网络。请参阅第 19-11 页的网络连接。

- 笔记

  如果您使用的是 nfsroot，请注意主机上的 IP 地址在您使用时不会更改。配置不当的 DHCP 服务器可能会导致发生意外的地址更改。如果您依赖 NFS 服务器作为根文件系统，这会导致问题。

19.2.11 19.2.11 启动 Foundation 平台

有关详细信息，请参阅第 19-12 页的命令行概述。

在块设备上使用 ARM 可信固件、UEFI、Linux 内核和文件系统启动模型

要启动模型，请使用：

```
./Foundation_Platform \

--data=fvp_bl1.bin@0x0 \

--data=fvp_fip.bin@0x8000000 \

--block-device=filesystem.img
```

fvp_bl1.bin 和 fvp_fip.bin 可以从 Linaro 网站下载 http://releases.linaro.org/latest/openembedded/aarch64/。

使用内核和 initramfs 启动模型

要使用内核和 initramfs 启动模型：

1. 在内核中构建一个 initramfs。
2. 使用 boot-wrapper 创建一个 AXF（可能称为 linux-system.axf）。
3. 使用以下命令在 Foundation Platform 上启动 AXF：

```
./Linux64_Foundation_v8/Foundation_v8 --image linux-system.axf
```

使用用户模式网络启动模型

使用：

```
./Foundation_v8 --network=nat --image linux-system.axf
```

使用块设备上的内核和文件系统启动模型

使用：

./Foundation_v8 –image linux-system.axf –block-device filesystem.img

Linux 内核必须使用 CONFIG_VIRTIO、CONFIG_VIRTIO_MMIO、CONFIG_VIRTIO_RING 和 CONFIG_VIRTIO_BLK 进行编译，并且引导包装器中提供的内核引导参数必须包含 root=/dev/vda。这是 virtio 系统块的设备名称。

19.2.12 19.2.12 网络连接

本节介绍如何设置网络连接，然后配置网络环境以在 Linux 平台上使用。以下说明假定您的网络通过 DHCP 提供 IP 地址。如果不是这种情况，请咨询您的网络管理员。

- NAT，基于 IPv4 的网络通过使用用户级 IP 服务提供有限的 IP 连接。这不需要额外的权限来设置或使用，但有固有的限制。可以使用端口重映射来提供系统级服务或与主机上的服务冲突的服务。
- NAT（用户模式）网络支持已知问题会间歇性地导致网络速度变慢。

使用用户模式网络的一个问题是，由于它充当模型中的虚拟网络和主机网络之间的 NAT 路由器，它重新映射来宾连接的源端口。这可能会导致 NFS 服务器出现问题。这些通常默认配置为拒绝来自端口号大于 1023 的源端口的客户端连接。对于大多数 Linux 发行版提供的 NFS 服务器，可以通过在要挂载的 NFS 导出的配置条目中添加不安全选项来解决此问题。

- /etc/exports 目录中列出了使用 NFS 导出的本地文件系统路径（NFS 导出）。可以为/etc/exports 中的任何条目指定不安全选项，使用：

  ```
  /pub *(ro,insecure,all_squash)
  ```

19.2.13 19.2.13 建立网络连接

NAT 网络将满足平台的大多数要求，但在某些情况下，例如使用本地 nfsroot，将需要桥接。必须更新主机 Linux 机器上的网络配置以允许这样做。这有几个阶段：

1. 确保您已在系统上安装了 brctl 实用程序

   ARM 建议使用 Linux 发行版中包含的标准 Linux 桥接实用程序。有关 Linux 网桥实用程序的更多信息，请参阅：http://www.linuxfoundation.org/collaborate/workgroups/networking/bridge

2. 使用以下命令禁用当前以太网设备（在本例中为 eth0）：

   ```
   \# ifconfig eth0 down
   ```

   您必须关闭网络管理器才能执行此操作。它不支持高级网络选项，例如桥接。

3. 要关闭网络管理器，请使用：

```
/etc/init.d/NetworkManager stop
```

关闭网络管理器可能因系统而异。检查您的操作系统文档以了解如何禁用网络管理器。

1. 添加一个新的虚拟设备（<bridge_name>，在本例中为 tap0），使用：

   ```
   \# ip tuntap add dev tap0 mode tap
   ```

2. 使用 eth0 和 tap0 添加新的桥接设备，并启用它，使用：

```
\# brctl addbr br0

\# brctl addif br0 eth0

\# brctl addif br0 tap0
```

1. 启用新的网桥设备并使用 DHCP 请求 IP 地址，使用：

   ```
   \# ifconfig br0 up

   \# dhclient br0
   ```

2. 这个新的网络设备 br0 应该像现有的 eth0 设备一样工作，但会在 eth0 和 tap0 之间共享物理网络连接。模型

   使用 tap0 虚拟设备访问网络。

3. 使用命令行选项 --network=bridged 运行模型

   ```
   --network-bridge=<bridge name>
   ```

   ```
   where <bridge_name> = tap0.
   ```

4. 通过 ping 任何合适的网站检查网络设施是否存在。

19.2.14 19.2.14 命令行概述

命令行参数提供所有模型配置。使用 --help 运行模型以获取可用命令的摘要。

在命令行上使用的语法是:

```
./Foundation_v8 [OPTIONS...]
```

表 19-1 显示了这些选项。

Table 19-1 Command-line options

`--help`	Display this help message and quit.
`--version`	Display the version and build numbers and quit.
`--quiet`	Suppress any non-simulated output on stdout or stderr.
`--cores=N`	Specify the number of cores, where `N` is 1 to 4. The default is 1. See *Multicore configuration* on page 19-14.
`--bigendian`	Start processors in big endian mode. The default is little endian.
`--(no-)secure-memory`	Enable or disable separate Secure and Non-secure address spaces. The default is disabled.
`--(no-)gicv3`	Enable GICv3 or legacy compatible GICv2 as interrupt controller. The default is `--no-gicv3`, that is, GICv2 mode.
`--block-device=file`	Image file to use as persistent block storage.
`--read-only`	Mount block device image in read-only mode.
`--image=file`	*Executable and Linking Format* (ELF) image to load.
`--data=file@address`	Raw file to load at an address in Secure memory.
`--nsdata=file@address`	Raw file to load at an address in Non-secure memory.
`--(no-)semihost`	Enable or disable semihosting support. The default is enabled. See *Semihosting* on page 19-15.
`--semihost-cmd=cmd`	A string used as the semihosting command line. See *Semihosting* on page 19-15.
`--uart-start-port=P`	Attempt to listen on a free TCP port in the range P to P+100 for each UART. The default is 5000.
`--network=(none|nat|bridged)`	Configure mode of network access. The default is none.
`--network-nat-subnet=S`	Subnet used for NAT networking. The default is 172.20.51.0/24.
`--network-nat-ports=M`	Optional comma-separated list of NAT port mappings in the form: `host_port=model_port`, for example, 8022=22.
`--network-mac-address`	MAC address to use for networking. The default is 00:02:f7:ef:f6:74.
`--network-bridge=dev`	Bridged network device name. The default is ARM0.
`--switches=val`	Initial setting of switches in the system register block (default: 0).
`--(no-)visualization`	Starts a small web server to visualize platform state. The default is disabled. See *Web interface*.
`--use-real-time`	Sets the generic timer registers to report a view of real time as it is seen on the host platform, irrespective of how slow or fast the simulation is running.
`--p9_root_dir`	Host folder to be shared with the guest.

image-

19-1-2

如果提供了多个 --image、--data 或 --nsdata 选项，则图像和数据按照它们在命令行中出现的顺序加载，并从最后一个 ELF 的入口点开始模拟指定，如果没有提供 ELF 图像，则为地址 0。您可以指定多个 --image、-- data 或 --nsdata 选项。

19.2.15 19.2.15 网页界面

在命令行上使用的语法如下：

```
./Foundation_v8 --visualization
```

or

```
./Foundation_v8 --no-visualization
```

使用 --visualization 选项运行模型，而不使用 --quiet 选项，会显示额外的输出：

```
terminal_0: Listening for serial connection on port 5000
terminal_1: Listening for serial connection on port 5001
terminal_2: Listening for serial connection on port 5002
terminal_3: Listening for serial connection on port 5003
```

可视化 Web 服务器在端口 2001 上启动

terminal_n 行与 UART 相关。请参阅 UART。

使用您的网络浏览器访问地址 http://127.0.0.1:2001

浏览器显示一个可视化窗口，如图 19-3 所示。

image-19-3

Figure 19-3 Visualization window

可视化窗口提供模型各部分状态的动态视图以及更改平台开关状态的能力。

19.2.16 19.2.16 串口

当 Foundation Platform 启动时，它会初始化四个 UART。对于每个 UART，它会搜索一个空闲的 TCP 端口，用于对 UART 进行 telnet 访问。它通过顺序扫描 100 个端口范围并使用第一个空闲端口来实现此目的。启动端口默认为 5000，您可以使用 --uart-start-port 命令行参数更改它。

将终端或程序连接到给定端口会显示和接收来自相关 UART 的输出，并允许输入到 UART。

UART 输出数据时，如果端口没有连接终端或程序，则自动启动终端。

仅当设置了 DISPLAY 环境变量且不为空时，终端才会自动启动。

19.2.17 19.2.17 串口输出

要使 UART 输出可见，必须在主机上安装 xterm 和 telnet ，并在 PATH 中指定。

19.2.18 19.2.18 多核配置

默认情况下，模型以单个处理器启动，该处理器从最后提供的 ELF 映像中的入口点开始执行，如果没有提供 ELF 映像，则从地址 0 开始执行。

您可以使用 --cores=N 将模型配置为最多具有四个处理器内核。每个内核开始执行相同的图像集，从相同的地址开始。-- 可视化根据处理器是处于 ARM 还是 Thumb 状态，拦截 AArch32 中的 SVC 0x123456 或 0xAB 。与多核选项一起使用的命令行选项会产生一个如图 19-4 所示的可视化窗口。

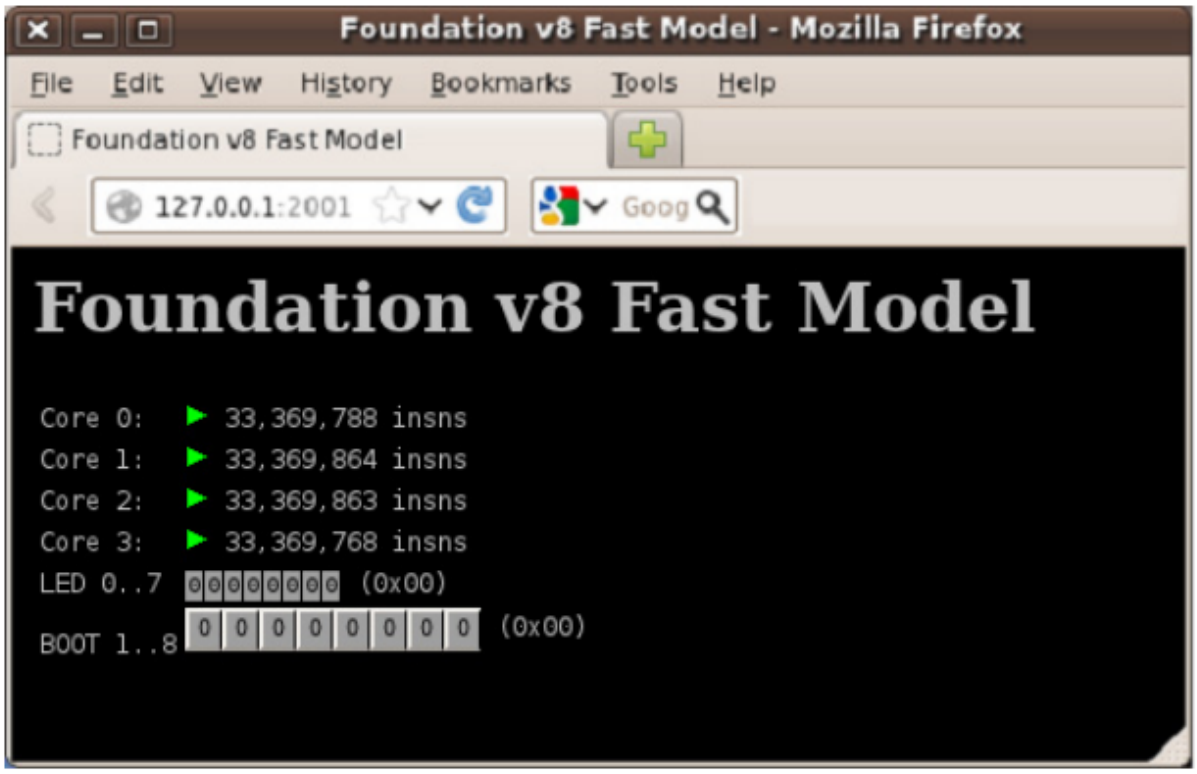

image-19-4

· **Figure 19-4 Multicore option with number of cores = 4**

19.2.19 19.2.19 半主机

半主机使在平台模型上运行的代码能够直接访问主机上的 I/O 设施。这些工具的示例包括控制台 I/O 和文件 I/O。有关半主机的更多信息，请参阅 ARM® 编译器软件开发指南。

模拟器通过以下任一方式处理半主机：

- 根据处理器是处于 ARM 还是 Thumb 状态，拦截 AArch32 中的 SVC 0x123456 或 0xAB
- 在 AArch64 中拦截 HLT 0xF000

19.2.20 19.2.20 半主机配置

在命令行上启用或禁用半主机的语法如下：

```
./Foundation_v8 --(no-)semihost
```

用于在命令行上设置半主机命令行字符串的语法如下：

```
./Foundation_v8 --semihost-cmd=<*command string*>
```

19.3 19.3 基础平台 FVP

固定虚拟平台 (FVP) 使您无需实际硬件即可开发软件。AEMv8-A 基础平台 FVP 使您能够在 ARMv8 开发平台的虚拟实现上运行软件应用程序，并为 ARM 软件可交付成果的开发、分发和演示提供标准参考平台。

它包含两个功能齐全的 ARM v8 架构包络模型的处理器集群 (AEMv8-A) 和标准外设集，旨在支持软件开发和移植。该平台是早期 VE 实时系统模型 (RTSM) 的演变，基于 ARM 生产的 Versatile™ Express (VE) 硬件开发平台。

AEMv8-A 基础平台 FVP 具有：

- 最多四个 AEMv8-A 多处理器型号的两个可配置集群实施：
- 所有异常级别的 AArch64
- 所有异常级别的可配置 AArch32 支持
- 在所有异常级别上对小端和大端的可配置支持
- 通用计时器
- 自托管调试
- CADI 调试
- GICv3 内存映射处理器接口和分配器
- 多媒体或网络环境的外围设备
- 四个 PL011 UART

- CoreLink CCI-400 缓存一致性互连
- 架构 GICv3 模型
- ARM 高清 LCD 显示控制器，1920 × 1080 分辨率，60fps，具有单个 I2S 和四个立体声通道
- 64 MB NOR 闪存和电路板外围设备
- CoreLink TZC-400 TrustZone 地址空间控制器

19.3.1 19.3.1 软件要求

该平台应安装以下软件组件：

Linux

The following software is supported:

操作系统

- 适用于 64 位和 32 位架构的 Red Hat Enterprise Linux 5.x 版本
- 适用于 64 位和 32 位体系结构的 Red Hat Enterprise Linux 6.x 版本

Shell

与 sh 兼容的 shell，例如 bash 或 tcsh。该模型应该在任何最新的 x86 64 位 Linux 操作系统上运行，只要存在 glibc v2.3.2（或更高版本）和 libstdc++ 6.0.0（或更高版本）。

微软 Windows

Microsoft Windows 需要以下软件：

操作系统

- 带有 Service Pack 1 的 Microsoft Windows 7，32 位或 64 位
- Microsoft Visual C++ 2008 可再发行包 ATL 安全更新 (KB973551)

Adobe Acrobat reader

阅读器版本 8 或更高版本

19.3.2 19.3.2 验证安装

AEMv8-A 基础平台 FVP 仅作为预构建平台二进制文件提供。安装目录格式如下：

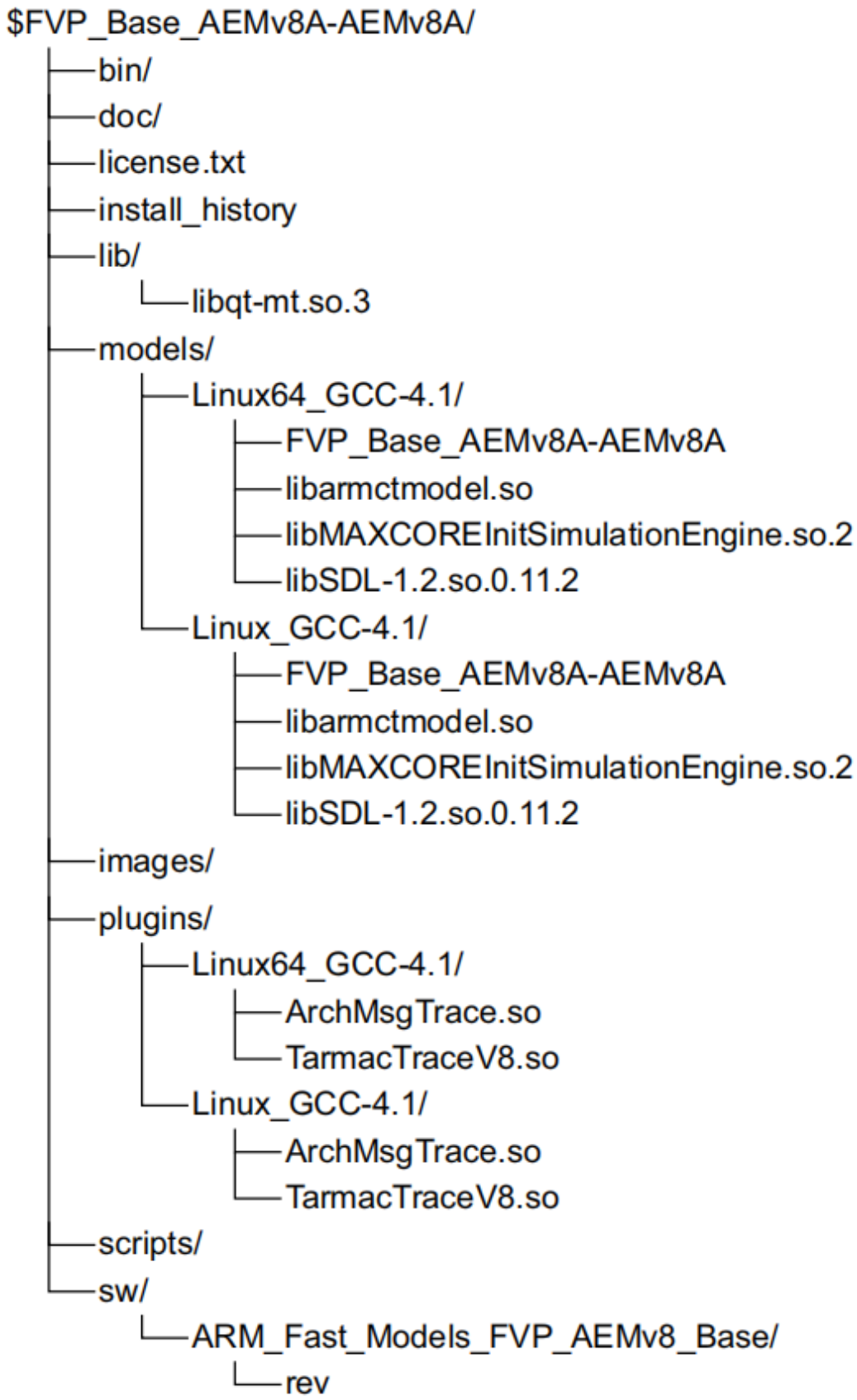
$FVP_Base_AEMv8A-AEMv8A/
bin/
doc/
license.txt
install_history
lib/
libqt-mt.so.3
models/
Linux64_GCC-4.1/
FVP_Base_AEMv8A-AEMv8A
libarmctmodel.so
libMAXCOREInitSimulationEngine.so.2
libSDL-1.2.so.0.11.2
Linux_GCC-4.1/
FVP_Base_AEMv8A-AEMv8A
libarmctmodel.so
libMAXCOREInitSimulationEngine.so.2
libSDL-1.2.so.0.11.2
images/
plugins/
Linux64_GCC-4.1/
ArchMsgTrace.so
TarmacTraceV8.so
Linux_GCC-4.1/
ArchMsgTrace.so
TarmacTraceV8.so
scripts/
sw/
ARM_Fast_Models_FVP_AEMv8_Base/
rev

image-19-5

Figure 19-5 AEMv8-A Base Platform FVP directory tree

路径：

bin

模型调试器可执行文件。有关详细信息，请参阅快速模型用户指南的模型调试器。

doc

模型和工具的文档

license.txt

模型使用许可条款

install_history

安装程序日志文件

images

简单演示程序的源代码和可执行文件

**lib **

libqt-mt.so.3 模型调试器支持库

models

compiler

```
FVP_Base_AEMv8A-AEMv8A
The AEMv8-A Base Platform FVP executable file.
libarmctmodel.so
Code translation library.
libMAXCOREInitSimulationEngine.so.2
Helper library required by the model.
libSDL-1.2.so.0.11.2
Helper library required by the model.
```

plugins

TarmacTraceV8 用于生成已执行代码的文本跟踪输出的插件

ArchMsgTrace 当模型检测到目标代码正在使用处理器的 IMPLEMENTATION DEFINED 或 UNPREDICTABLE 行为时生成警告的插件

scripts

使用 TAP 形式的模型网络设置脚本

sw

ARM_Fast_Models_FVP_AEMv8_Base 模型构建号和修订标识符

19.3.3 19.3.3 半主机支持

半主机使在平台模型上运行的代码能够直接访问主机上的 I/O 设施。这些工具的示例包括控制台 I/O 和文件 I/O。查看 ARM® 编译器软件开发指南。

模拟器通过以下任一方式处理半主机：

- 根据处理器是处于 ARM 还是 Thumb 状态，拦截 AArch32 中的 SVC 0x123456 或 0xAB
- 在 AArch64 中拦截 HLT 0xF000

19.3.4 19.3.4 在调试器中使用配置 GUI

在您的调试器中，可以在连接到模型并启动它之前配置 FVP 参数。请参阅调试器随附的文档。

要连接到 AEMv8-A 基础平台 FVP，您的调试器必须具有 CADI 接口。

19.3.5 19.3.5 从 Model Shell 设置模型配置选项

您可以通过命令行或模型的 CADI 属性中提供的配置设置来控制 AEMv8-A Base Platform FVP 的初始状态。

使用配置文件

要配置从命令行使用 Model Shell 启动的模型，请在启动 AEMv8-A Base Platform FVP 时包含对可选纯文本配置文件的引用。

配置文件中的注释行必须以 # 字符开头。

配置文件的每个非注释行包含：

- 组件实例的名称
- 要修改的参数及其值
- 您可以使用 true/false 或 1/0 设置布尔值。如果字符串包含空格，则必须用双引号将字符串括起来

Example 19-1 Typical configuration file

```
# Disable semihosting using true/false syntaxsemihosting-enable=false
#
# Enable the boot switch using 1/0 syntax
bp.sp810_sysctrl.use_s8=1
#
# Set the boot switch position
bp.ve_sysregs.user_switches_value=1
```

您可以通过传递 -l 开关来获取模型参数的完整列表。

例如：

```
models/Linux64_GCC-4.1/FVP_Base_AEMv8A-AEMv8A -l
# Parameters:
# instance.parameter=value #(type, mode) default = 'def value' : description :
[min..max]
#-------------------------------------------------------------------------
cache_state_modelled=1 # (bool, init-time) default = '1'         // Enabled d-cache␣
↪and

↪/ i-cache state for

↪/ all components
cluster0.NUM_CORES=0x4 # (int , init-time) default = '0x4'         // Number of cores␣
↪in

↪/ cluster0:[0x1..0x4]
cluster1.NUM_CORES=0x4 # (int , init-time) default = '0x4'         // Number of cores␣
↪in

↪/ cluster1:[0x0..0x4]
gicv3.gicv2-only=0 # (bool, init-time) default = '0'                  // When using the

↪/ GICv3 model,pretend

↪/ to be a GICv2

↪/ system.
semihosting-enable=1 # (bool, init-time) default = '1'                  // Enable␣
↪semihosting

↪/ for all cores
spiden=1 # (bool, init-time) default = '1'                                        //␣
↪Debug authentication

↪/ signal spiden
spniden=1 # (bool, init-time) default = '1'                                    // Debug␣
↪authentication

↪/ signal spniden
<output truncated>
```

使用命令行

您可以在调用模型时使用 -C 开关来定义模型参数。您还可以使用 --parameter 作为 -C 开关的同义词。使用与配置文件相同的语法，但在每个参数前加上 -C 开关。

19.3.6 19.3.6 在 AEMv8-A 基础平台 FVP 上加载和运行应用程序

提供了与 AEMv8-A 基础平台 FVP 一起使用的示例应用程序。

这些应用程序仅用于演示目的，ARM 不支持。示例或实现细节的数量可能会随着系统模型的不同版本而变化。

在所有版本的 AEMv8-A Base Platform FVP 上运行的有用示例应用程序是：

brot_ve_64.axf

此应用程序提供了将图像渲染到 CLCD 显示器的演示，提供源代码。这些示例位于%PVLIB_HOME%\images 目录

中。

19.3.7 19.3.7 从命令行运行示例程序

1. 打开终端窗口并导航到安装目录
2. 使用以下命令打开并运行提供的示例程序：

```
models/Linux64_GCC-4.1/FVP_Base_AEMv8A-AEMv8A \
-a cluster\*.cpu\*=images/brot_ve_64.axf \
-C bp.secure_memory
```

models/Linux64_GCC-4.1/FVP_Base_AEMv8A-AEMv8A

是模型可执行文件。此版本仅适用于 64 位主机。Linux_GCC-4.1 中的文件适用于 32 位和 64 位主机。

-a cluster*.cpu*

指示模型启动映像，为所有集群的所有核心设置 pc 的初始值。

-C bp.secure_memory=false

禁用 CoreLink TZC-400 对 DRAM 的访问控制。这是必需的，因为这个简单的示例映像不包含初始化 TZC-400 的固件实现

19.3.8 19.3.8 使用模型调试器运行示例程序

1. 启动模型调试器。它位于安装目录中的 bin/modeldebugger 中，或者使用 Windows 中的“开始”菜单
2. 选择文件 → 调试 Isim 系统

   出现一个对话框
3. 对于系统，选择以下模型可执行文件：

   models/Linux64_GCC-4.1/FVP_Base_AEMv8A-AEMv8A

4. 在其他命令行选项中，键入：

   -C bp.secure_memory=false

   点击 ok

5. 选择 cluster0 中的一个或多个目标，然后单击“ok”

6. 在 Load Application 对话框中，确保选中 Enable SMP Application Loading，然后找到并选择 images/brot_ve_64.axf 文件

7. 在打开的 ModelDebugger 窗口之一上单击 run

19.3.9 19.3.9 使用 CLCD 窗口

当 AEMv8-A 基础平台 FVP 启动时，FVP CLCD 窗口打开，显示模拟彩色 LCD 帧缓冲区的内容。它会自动调整大小以匹配 CLCD 外设寄存器中设置的水平和垂直分辨率。

图 **19-6** 显示了 FVP CLCD 在其默认状态下，即启动后的状态。

image-19-6

Figure 19-6 CLCD window at startup

CLCD 窗口的顶部显示以下状态信息：

Total Instr

一个计数器，显示执行的指令总数。

Total Time

间显示总经过时间的计数器，以秒为单位，这是挂钟时间，不是模拟时间。

Rate Limit

制此选项限制内核处于 WFI、重置或否则闲置。

速率限制默认启用。模拟时间受到限制，使其更接近实时。

单击方形按钮以禁用或启用速率限制。禁用速率限制时，文本从 ON 变为 OFF，彩色框变暗。

第 **19-22** 页的图 **19-7** 显示了禁用速率限制的 CLCD。

您可以使用 rate_limit-enable 控制是否启用速率限制参数，实例化模型时 AEMv8-A 基础平台 FVP 可视化组件的可视化参数之一。

Instr/sec

显示挂钟时间每秒执行的指令数。

Perf Index

实时与仿真时间的比率。比例越大，速度越快模拟运行。如果启用速率限制功能，则性能指数接近统一。

CLCD display

The large area at the bottom of the window displays the contents of the CLCD buffer, as in Figure **19-7**.

image-19-6

Figure 19-7 CLCD window active

Icon	State label	Description
	UNKNOWN	Run status unknown, that is, simulation has not started.
	RUNNING	The core is running, is not idle, and is executing instructions.
	HALTED	An external halt signal is asserted.
	STANDBY_WFE	The last instruction executed was WFE, and standby mode has been entered.
	STANDBY_WFI	The last instruction executed was WFI and standby mode has been entered.
	IN_RESET	An external reset signal is asserted.
	DORMANT	Partial core power down.
	SHUTDOWN	Complete core power down.

image-19-2

Table 19-2 Core run state icon descriptions

在您开始模拟之前，图标不会出现。

您可以通过按左 Ctrl+ 左 Alt 键来隐藏主机鼠标指针。再次按这些键可重新显示主机鼠标指针。只有左 Ctrl 键是可操作的。键盘右侧的 Ctrl 键没有相同的效果。

如果您喜欢使用不同的键，请使用 trap_key 配置选项，它是 Visualization 组件的可视化参数之一。

19.3.10 19.3.10 将以太网与 AEMv8-A 基础平台 FVP 结合使用

AEMv8-A 基础平台 FVP 为您提供了一个虚拟以太网组件。这是 SMSC91C111 以太网控制器的模型，使用 TAP 设备与网络通信。默认情况下，以太网组件被禁用。

AEMv8-A 基础平台 FVP 包括 SMSC91C111 以太网控制器的软件实现。因此，您的目标操作系统必须包含此特定设备的驱动程序，并且您必须配置内核以使用 SMSC 芯片。Linux 是支持 SMSC91C111 的操作系统。

共有三个可配置的 SMSC91C111 组件参数：

- enabled
- mac_address
- promiscuous

enabled

当设备被禁用时，内核无法检测到它。有关详细信息，请参阅快速模型参考手册中的 SMSC_91C111 组件部分。第 19-24 页的图 19-8 显示了模型网络结构的框图。

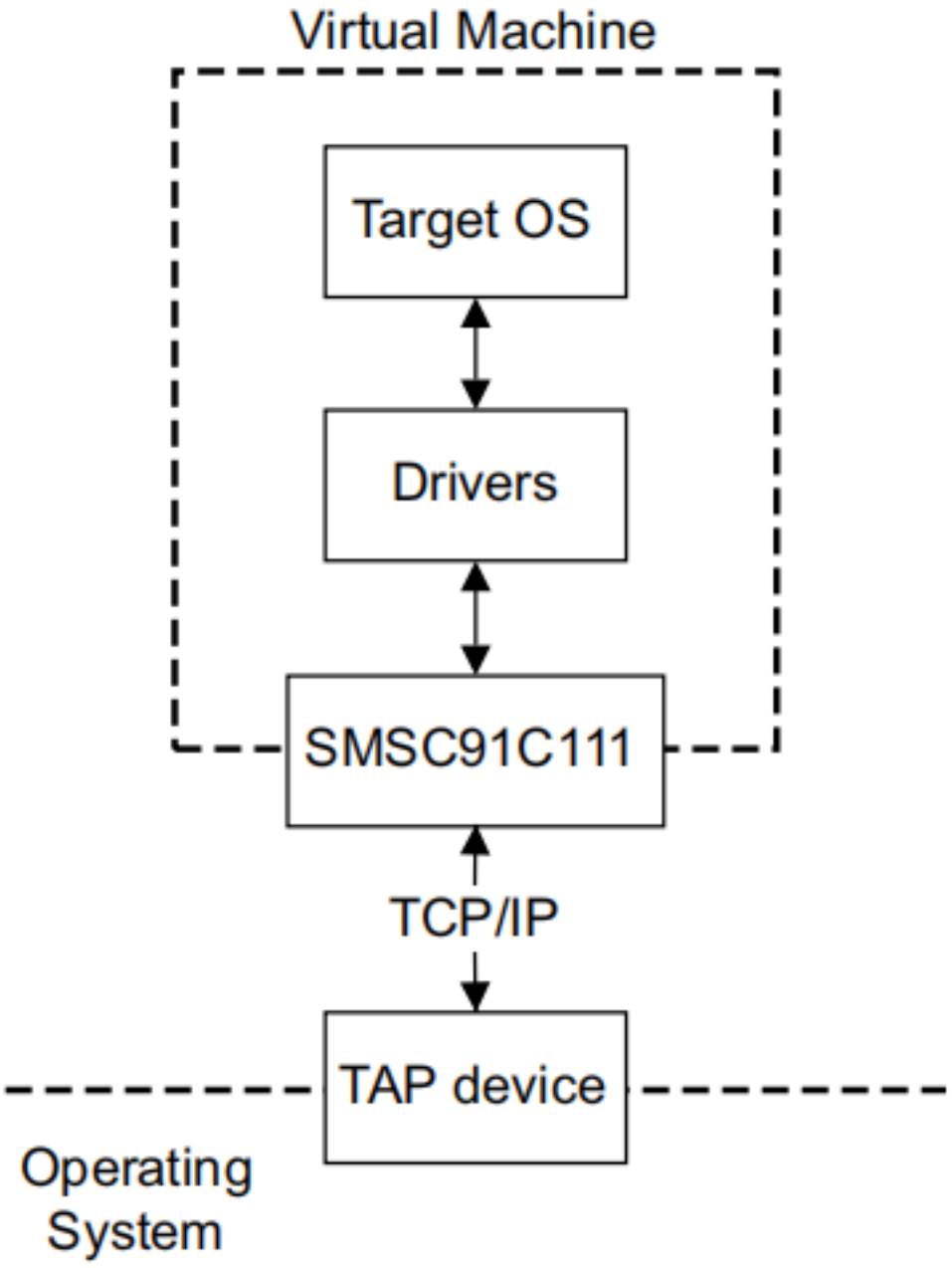

image-19-8

Figure 19-8 Model networking structure block diagram

您必须配置 HostBridge 组件才能在 TAP 设备上执行读取和写入操作。HostBridge 组件是一个虚拟程序员视图模型，充当网络网关与主机上的 TAP 设备交换以太网数据包，并将数据包转发到 NIC 模型。

mac_address

如果未指定 MAC 地址，则在运行模拟器时，它会采用随机生成的默认 MAC 地址。当在本地网络的多个主机上运行模型时，这提供了一定程度的 MAC 地址唯一性。

promiscuous

默认情况下，以太网组件以混杂模式启动。这意味着它可以接收所有网络流量，即使是没有专门针对设备的

任何流量。如果您将单个网络设备用于多个 MAC 地址，则必须使用此模式。例如，如果您在主机操作系统和 AEMv8-A Base Platform FVP 以太网组件之间共享同一个网卡，请使用此模式。

默认情况下，AEMv8-A Base Platform FVP 上的以太网设备具有随机生成的 MAC 地址并以混杂模式启动。

19.3.11 19.3.11 与 VE 模型和平台的兼容性

在以前的 VE 模型上运行的软件应该与 AEMv8-A 基础兼容平台 FVP，但可能需要更改以下配置选项：

- GICv2
- GICv3
- System global counter
- Disable platform security.

GICv2

AEMv8-A 基础平台 FVP 默认使用 GICv3。它可以配置为使用 GICv2 或 GICv2m 兼容模式。

要将模型配置为 GICv2m，请设置以下内容：

```
-C gicv3.gicv2-only=1 \
-C cluster0.gic.GICD-offset=0x10000 \
-C cluster0.gic.GICC-offset=0x2F000 \
-C cluster0.gic.GICH-offset=0x4F000 \
-C cluster0.gic.GICH-other-CPU-offset=0x50000 \
-C cluster0.gic.GICV-offset=0x6F000 \
-C cluster0.gic.PERIPH-size=0x80000 \
-C cluster1.gic.GICD-offset=0x10000 \
-C cluster1.gic.GICC-offset=0x2F000 \
-C cluster1.gic.GICH-offset=0x4F000 \
-C cluster1.gic.GICH-other-CPU-offset=0x50000 \
-C cluster1.gic.GICV-offset=0x6F000 \
-C cluster1.gic.PERIPH-size=0x80000 \
-C gic_distributor.GICD-alias=0x2c010000
```

要将模型配置为 GICv2，请设置以下内容：

```
-C gicv3.gicv2-only=1 \
-C cluster0.gic.GICD-offset=0x1000 \
-C cluster0.gic.GICC-offset=0x2000 \
-C cluster0.gic.GICH-offset=0x4000 \
-C cluster0.gic.GICH-other-CPU-offset=0x5000 \
-C cluster0.gic.GICV-offset=0x6000 \
-C cluster0.gic.PERIPH-size=0x8000 \
-C cluster1.gic.GICD-offset=0x1000 \
```

(continues on next page)

(continued from previous page)

```
-C cluster1.gic.GICC-offset=0x2000 \
-C cluster1.gic.GICH-offset=0x4000 \
-C cluster1.gic.GICH-other-CPU-offset=0x5000 \
-C cluster1.gic.GICV-offset=0x6000 \
-C cluster1.gic.PERIPH-size=0x8000 \
-C gic_distributor.GICD-alias=0x2c010000
```

要为 GICv2m 配置 MSI 帧，可以使用额外的参数来设置 16 个可能的帧（8 个安全帧和 8 个非安全帧）的基地址和配置：

```
-C gic_distributor.MSI_S-frame0-base=ADDRESS \
-C gic_distributor.MSI_S-frame0-min-SPI=NUM \
-C gic_distributor.MSI_S-frame0-max-SPI=NUM
```

在此示例中，您可以将 MSI_S 替换为 MSI_NS，对于 NS 帧，您可以替换 frame0 对于可能的 16 帧中的每一个，使用 frame1 到 frame7 。如果未为给定帧指定基地址，或者 SPI 编号超出范围，则不会实例化相应的帧。

GICv3

AEMv8-A 核心模型包括 GICv3 系统寄存器的实现。这是默认启用的。

GIC 分配器和 CPU 接口有几个参数，允许配置模型以匹配不同的实现选项。使用 --list-params 获取完整列表。

GIC 模型的配置选项应在以下位置可用：

- `cpu/cluster.gic.*`
- `cpu/cluster.gicv3.*`
- `gic_distributor.*`

System global counter

VE 模型没有为系统全局计数器提供内存映射接口，并启用了自由运行定时器从复位。但是，架构要求是在重置时不启用此类计数器。这意味着内核的通用定时器寄存器不会运行，除非：

- 软件通过在 0x2a43000 的 CNTCR 中写入 FCREQ[0] 和 EN 位来启用计数器外设。这是首选方法。
- -C bp.refcounter.non_arch_start_at_default =1 参数已设置。这是一种与旧软件兼容的备份方法。

Disable platform security

VE 模型根据固定的安全映射选择性地限制访问。在 AEMv8-A 基础平台 FVP 中，外围设备的安全映射得到了增强，现在默认启用。软件必须对 TZC-400 进行编程才能对 DRAM 进行任何访问，因为所有访问都在复位配置中被阻止。

为了与尚未更新以对 TZC-400 进行编程的软件兼容，提供了以下参数，这将导致无论安全状态如何都允许所有访问：

```
-C bp.secure_memory=false
```

19.3.12 19.3.12 从哪里获得 ARMv8-A 基础平台 FVP

有关 ARMv8-A 基础平台 FVP 的详细信息，请访问 http://www.arm.com/fvp

CHAPTER

TWENTY

附录

20.1 术语表