ARMv8-A指令集及其体系结构

Contents

目录

一、引言与概念辨析

二、架构和处理器

三、异常级别与执行状态

四、寄存器配置

五、ARMv8指令集

六、A64汇编指令集

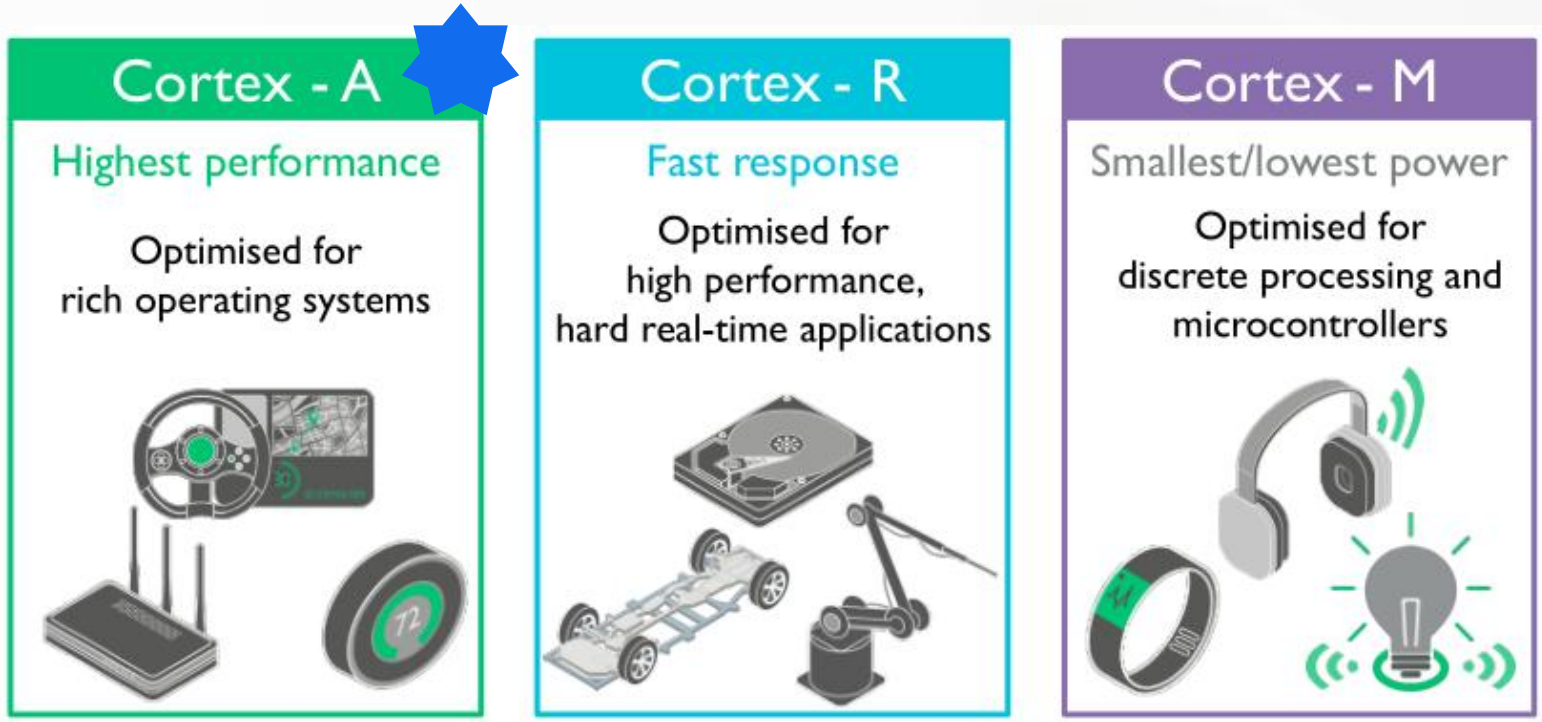
七、AArch64浮点和NEON

八、总结与回顾

Part One

一、引言与概念辨析

ARM Cortex-A/R/M



A系列：面向性能密集型系统的应用处理器内核，主要用于电脑与手机等。可以在上面运行操作系统。

R系列：面向实时应用的高性能内核，主要应用在对实时性要求高的场合，如车载控制产品等。

M系列：面向各类嵌入式应用的微控制器内核，主要用于消费电子领域微控制器，不能运行操作系统，可以类比成单片机。

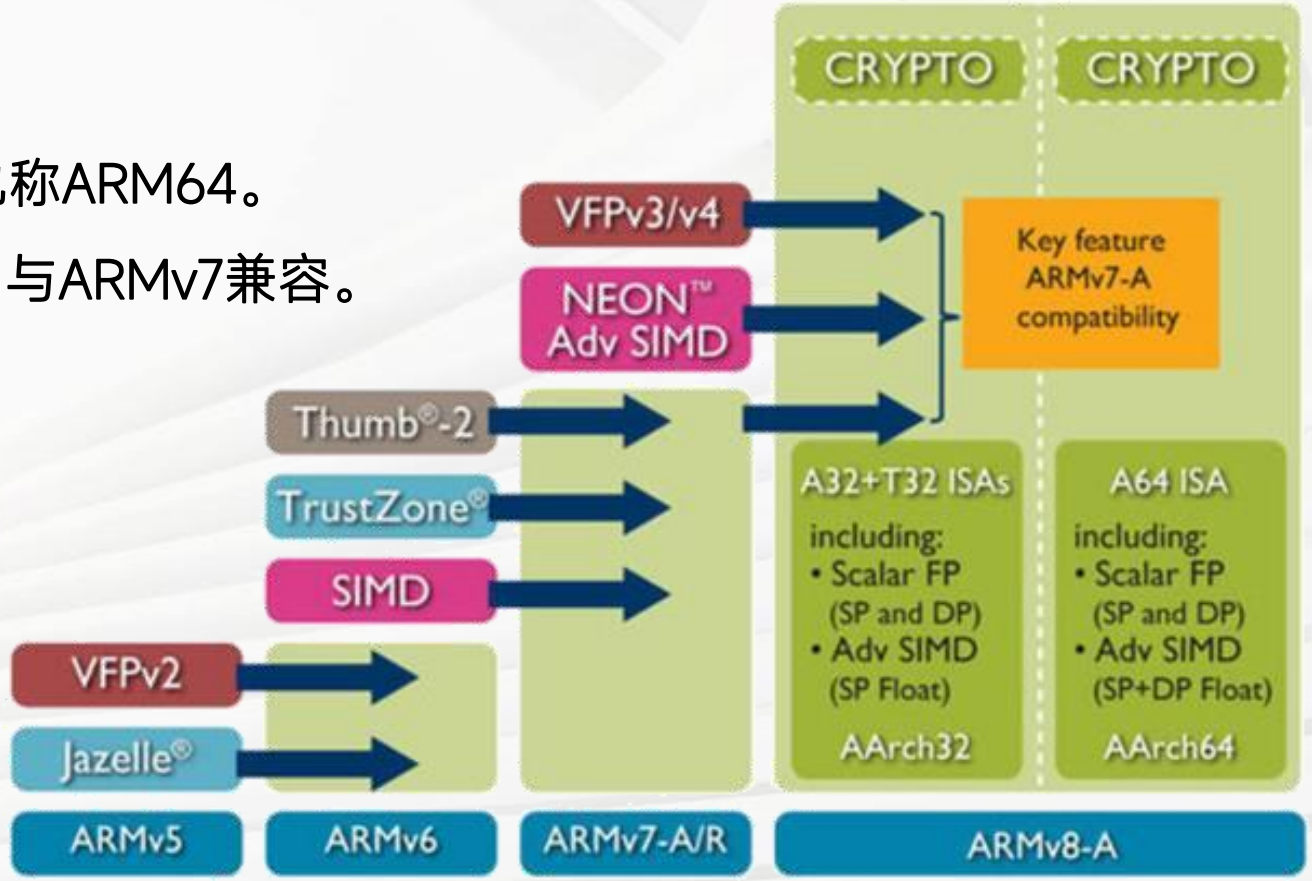
AArch64 与 AArch32 与 ARMv8

ARMv8体系结构包括32位和64位执行状态。

ARMv8引入了使用64位寄存器执行的能力，但提供了向后兼容的机制，使得现有的ARMv7软件能够执行。

AArch64指ARMv8架构的64位执行状态，也称ARM64。

AArch32指ARMv8架构的32位执行状态，它与ARMv7兼容。



Part Two

二、架构与处理器

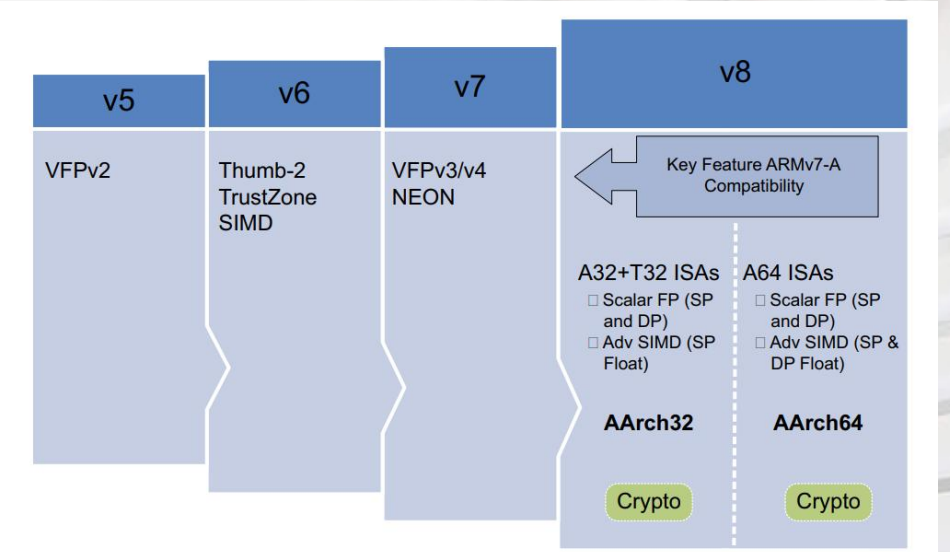


Figure 2-1 Development of the ARMv8 architecture

ARM架构发展历程

ARM架构可以追溯到1985年，自早期ARM核心以来，它已经进行了大规模的开发，在每一步都增加了新特性和功能。

ARMv4及更早 这些早期的处理器只使用ARM 32位指令集。

ARMv4T ARMv4T架构为ARM增加了Thumb 16位指令集到32位指令集中。这是第一个得到广泛许可的架构。ARM7TDMI® 和 ARM9TDMI® 处理器实现了这个架构。

ARMv5TE ARMv5TE体系结构改进了对DSP类的操作、饱和运算以及ARM和Thumb的配合进行了改进。ARM926EJ-S® 实现了这种架构。

ARMv6 ARMv6做了一些增强，包括对非对齐内存访问的支持、对内存体系结构的重大改变以及对多处理器的支持。此外，还包括了对32位寄存器中的字节或半字的SIMD操作的一些支持。ARM1136JF-S® 实现了这种架构。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架构要求必须使用Thumb-2扩展，并添加了高级SIMD扩展（NEON）。

ARMv8架构新变化

变化	描述
大物理地址	处理器可以访问超过4GB的物理内存
64位虚拟地址	虚拟内存也超过了4GB的限制
更大的寄存器	31个64位通用寄存器，提高了性能并减少了栈的使用
高效的64位立即数生成机制	减少了对字面池的需求
更大的PC相对寻址范围	+/-4GB的寻址范围
添加了16KB与64KB的页面大小	减少缺页率与页面遍历深度
新的异常模型	-
高效的缓存管理	-
提供硬件加速的加密功能	-
提供Load-Acquire和Store-Release指令	专为C/C++11和Java内存模型设计，提高线程性能

支持ARMv8架构的处理器

Cortex-A53处理器是一个中端、低功耗处理器，在一个族中有1到4个核，每个核都有一个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处理器是一个非常高效的处理器，能够支持运行32位和64位代码。它能够作为一个独立的应用程序处理器部署，也能与Cortex-A57处理器配对，以获得最佳性能、可伸缩性和能源效率。

Cortex-A53处理器具有以下特点:

- 顺序执行，8级流水线
- 通过使用分级的门控时钟、电源域和高级保留模式降低功耗。
- 通过复制执行资源和双指令译码器增加了双发射能力。
- 功率优化的L2缓存设计提供了更低的延迟，并平衡了性能和效率。

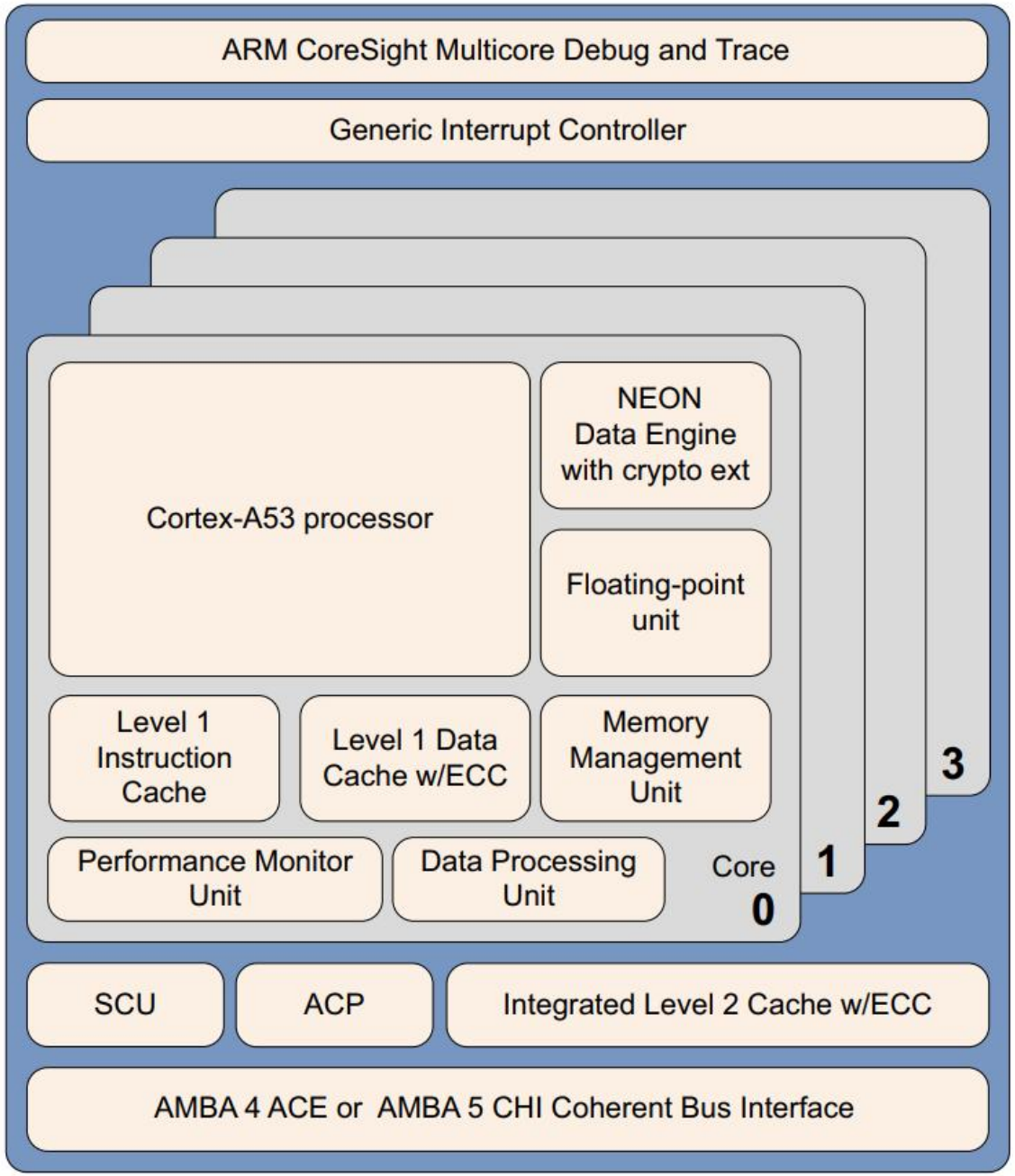


Figure 2-2 Cortex-A53 processor

支持ARMv8架构的处理器

Cortex-A57处理器针对移动和企业计算应用，包括计算密集型64位应用，如高端计算机、平板电脑和服务器产品。它可以与Cortex-A53处理器一起使用组合成ARM big.LITTLE，以实现可伸缩的性能和更高效的功耗。

Cortex-A57处理器完全实现了ARMv8-A架构。它支持多核运行，在单个簇中使用1个到4个核的多核处理。它提供了比ARMv7 Cortex-A15处理器更高的性能，有更高的能效水平。密码学扩展的加入使密码学算法的性能比上一代处理器提高了10倍。

Cortex-A57处理器具有以下特点:

- 乱序，15+级流水线
- 节能特性：路径预测、标记缩减和缓存查找压制
- 通过复制执行资源提高指令峰值吞吐量。
- 优化了性能的L2缓存设计，使簇中的多个核心能够同时访问L2。

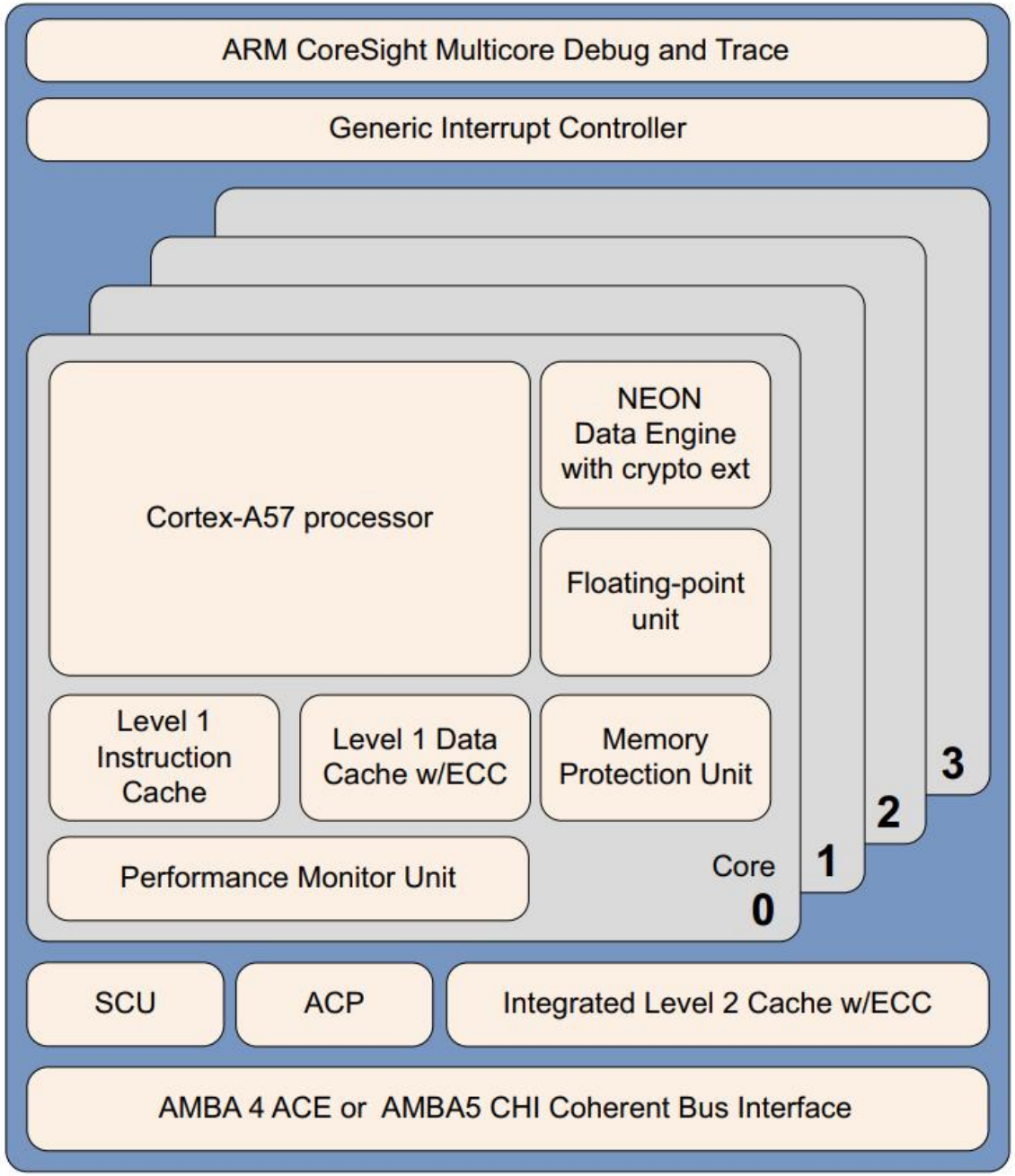


Figure 2-3 Cortex-A57 processor core

Part Three

三、异常级别与执行状态

异常级别

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

EL0：普通用户的应用程序

EL1：操作系统内核，通常认为是有特权的

EL2：Hypervisor（如虚拟机管理程序VMM）

EL3：低层固件，包括安全监视器。

在ARMv8中，指令执行发生在4个异常级别之一。在AArch64中，异常级别决定了特权级别，其方式类似于ARMv7中定义的特权级别。异常级别决定了特权级别，所以指令在ELn状态执行对应了特权PLn。n的数值越大，异常级别越高，权限越多。

异常级别提供了软件执行特权的逻辑分离，适用于ARMv8架构的所有运行状态。

安全状态

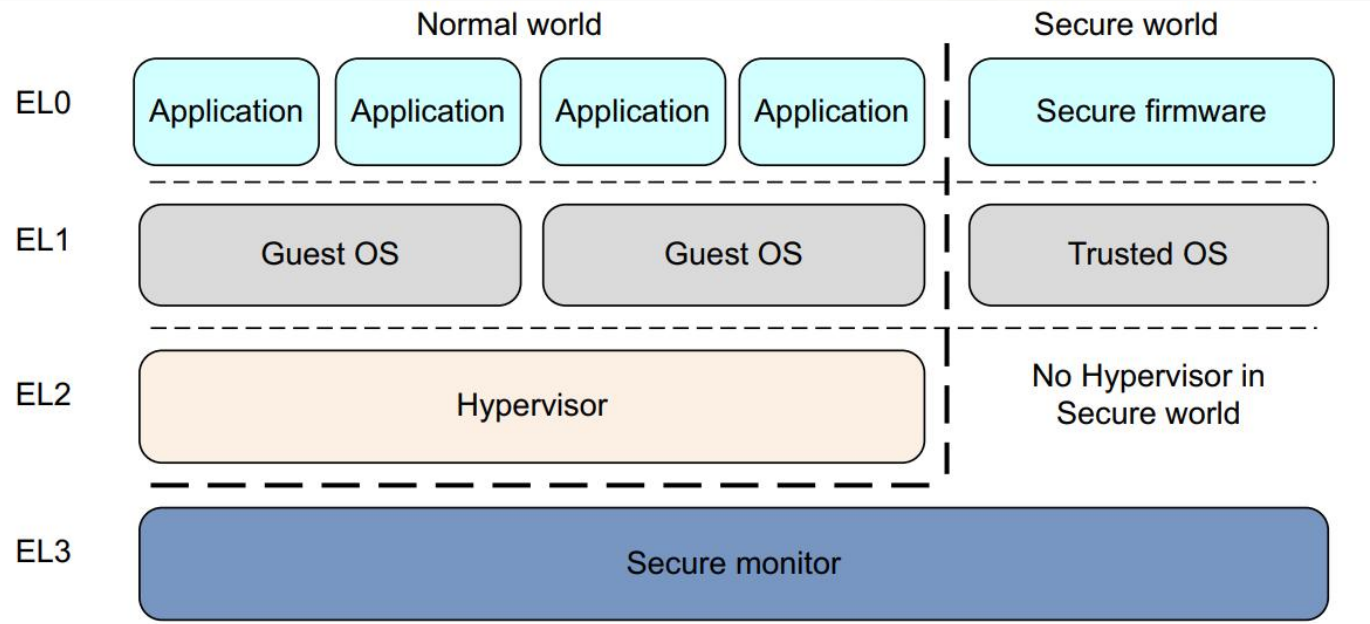


Figure 3-2 ARMv8 Exception levels in the Normal and Secure worlds

ARMv8-A提供了两种安全状态：安全世界和非安全世界。这使得操作系统可以与受信任的另一个操作系统在同一硬件上同时运行，并提供对某些软件攻击和硬件攻击的保护。ARM架构的TrustZone技术使系统能够在用户之间进行区分普通和安全世界的状态。安全监视器Secure Monitor充当普通和安全世界之间切换的网关。

执行状态

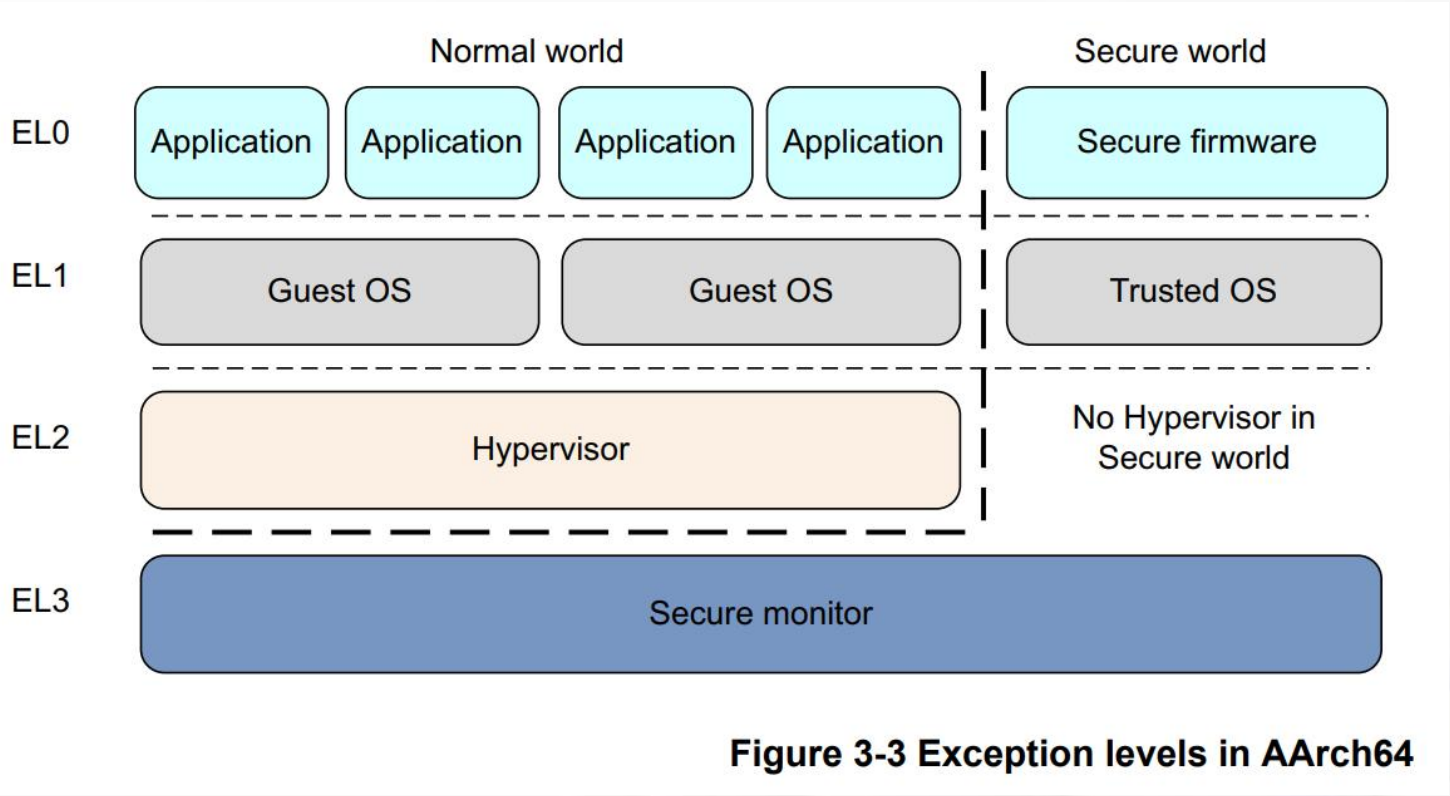


Figure 3-3 Exception levels in AArch64

ARMv8架构定义了两种执行状态，AArch64和AArch32，分别使用64位通用寄存器和32位通用寄存器。AArch32保留了ARMv7对特权的定义，而在AArch64中，特权级别由异常级别决定。在AArch64状态下，处理器执行A64指令集。当处于AArch32状态时，处理器可以执行A32或Thumb32指令集。

异常级别的转移

异常级别之间的转移遵循以下规则：

（1）转移到更高的异常级别，如从EL0到EL1，表示软件的执行权限增加了。

（2）异常不能被提到更低的异常级别。

（3）在EL0级别没有异常处理，异常必须在更高的异常级别处理。

（4）异常导致程序流的改变。在异常级别高于EL0的异常级别上，异常处理程序从异常向量开始执行，该异常向量已预定义并关联了各个异常。异常包括：①中断，比如IRQ和FIQ。②内存系统中止。③未定义的指令。④系统调用，允许非特权软件对操作系统进行系统调用。⑤安全监视器或hypervisor陷阱。

（5）通过执行ERET指令来结束异常处理并返回到之前的异常级别。

（6）从异常返回可以保持在相同的异常级别或进入更低的异常级别。不能移动到更高的异常级别。

（7）除非从EL3返回到非安全状态，否则安全状态会随着异常级别的改变而改变。

执行状态的转化

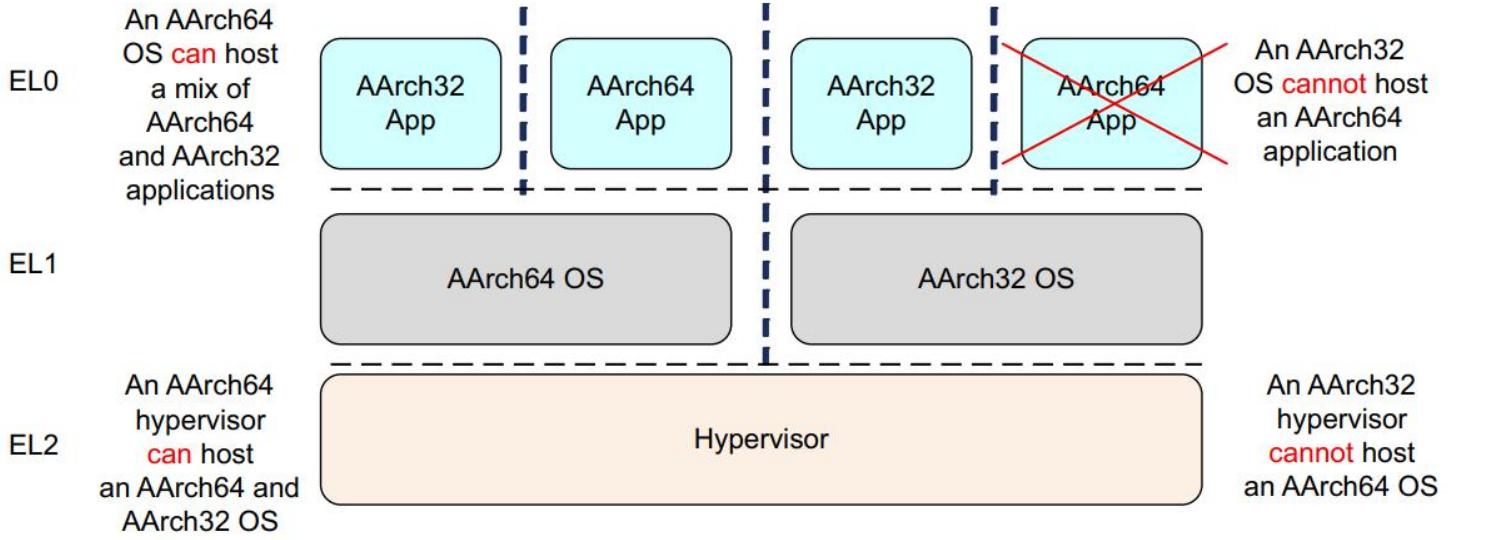


Figure 3-7 Moving between AArch32 and AArch64

有时必须更改系统的执行状态。例如，如果正在运行一个64位操作系统，并且希望在EL0运行一个32位的应用程序，那么系统必须改变到AArch32状态，当应用程序已经完成或执行返回到操作系统时，系统可以切换回AArch64。但是不能反过来做，一个AArch32操作系统不能承载64位的应用程序。

只能通过改变异常级别来改变执行状态。要在相同异常级别的执行状态之间进行更改，必须先切换到更高的异常级别，然后返回到原始异常级别。EL3上的代码不能将异常提升到更高的异常级别，因此不能更改执行状态。

Part Four

四、寄存器配置

通用寄存器

AArch64执行状态提供了31个64位的通用寄存器，可以在任何时间和所有异常级别访问，它们通常被称为X0-X30寄存器。每个AArch64 64位通用寄存器(X0-X30)也有一个32位(W0-W30)形式。32位的W寄存器形成了相应的64位X寄存器的低半部分。也就是说，W0映射到X0的低字，W1映射到X1的低字，依次类推。从W寄存器读取时忽略相应X寄存器的高32位，并保持它们不变。写入W寄存器将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。

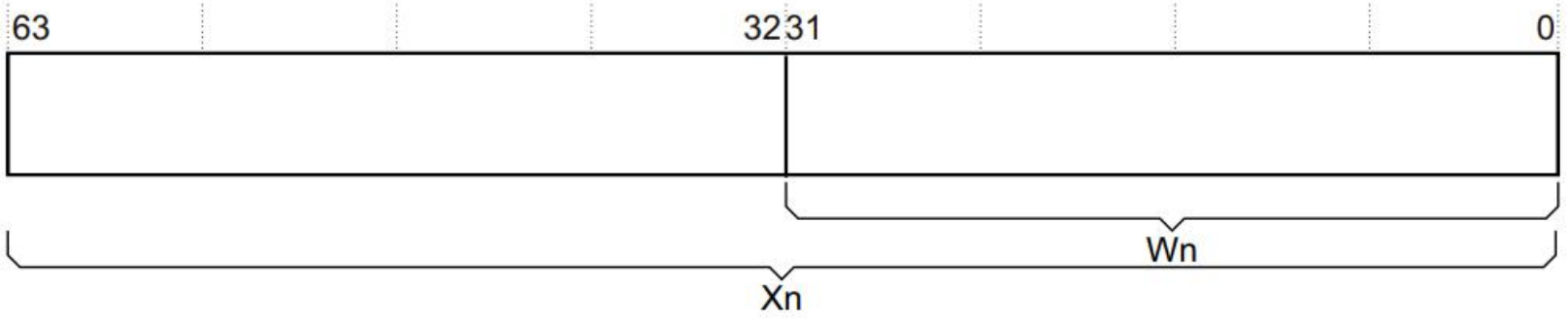


Figure 4-2 64-bit register with W and X access.

	X0/W0
	X1/W1
	X2/W2
	X3/W3
	X4/W4
	X5/W5
	X6/W6
	X7/W7
	X8/W8
	X9/W9
	X10/W10
	X11/W11
	X12/W12
	X13/W13
	X14/W14
	X15/W15
	X16/W16
	X17/W17
	X18/W18
	X19/W19
	X20/W20
	X21/W21
	X22/W22
	X23/W23
	X24/W24
	X25/W25
	X26/W26
	X27/W27
	X28/W28
Frame pointer	X29/W29
Procedure link register	X30/W30

EL0, EL1, EL2, EL3

专用寄存器

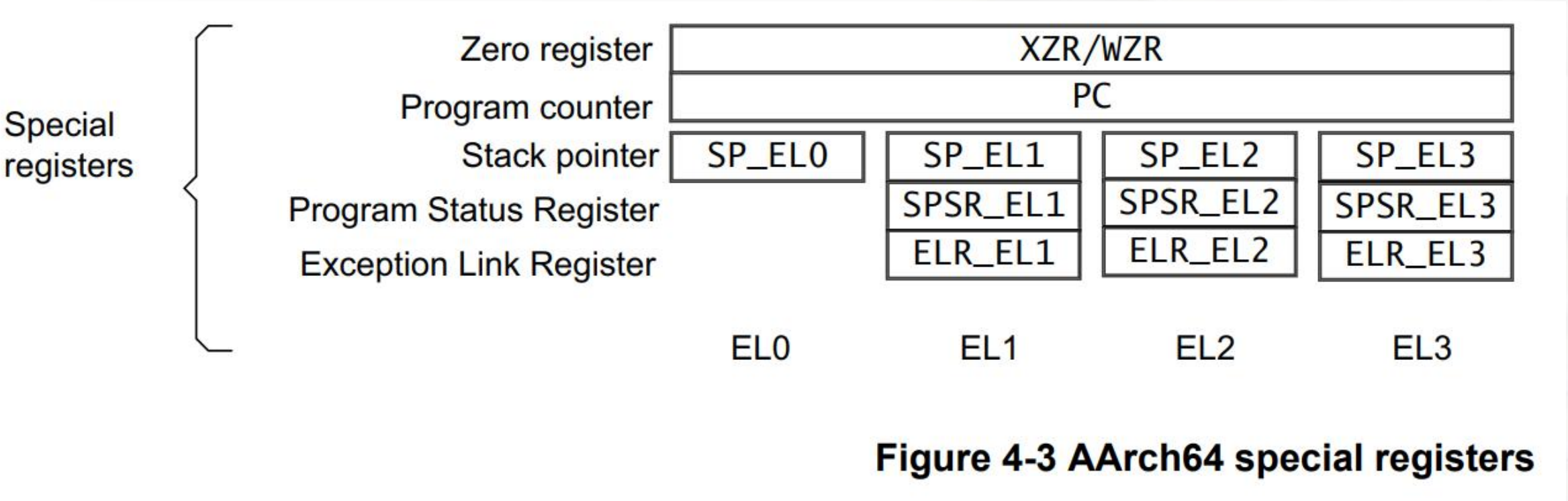


Figure 4-3 AArch64 special registers

专用寄存器：SPSR

当捕获一个异常时，处理器状态存储在相关的Saved Program Status register(SPSR)中。SPSR在捕获异常之前保存PSTATE的值，并用于在执行异常返回时恢复PSTATE的值。

在ARMv8中，对SPSR的写入依赖于异常级别。如果在EL1中发生异常，则使用SPSR_EL1。

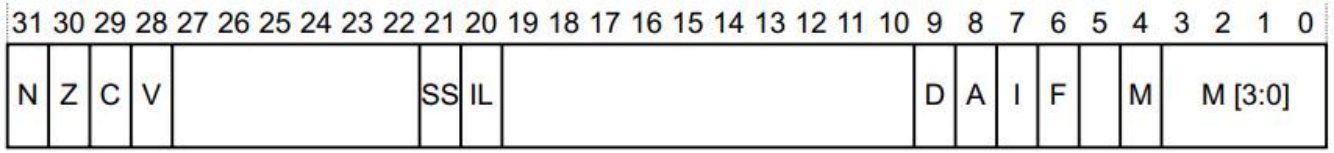


Figure 4-4 SPSR

The individual bits represent the following values for AArch64:

N	Negative result (N flag).
Z	Zero result (Z) flag.
C	Carry out (C flag).
V	Overflow (V flag).
SS	Software Step. Indicates whether software step was enabled when an exception was taken.
IL	Illegal Execution State bit. Shows the value of PSTATE.IL immediately before the exception was taken.
D	Process state Debug mask. Indicates whether debug exceptions from watchpoint, breakpoint, and software step debug events that are targeted at the Exception level the exception occurred in were masked or not.
A	SError (System Error) mask bit.
I	IRQ mask bit.
F	FIQ mask bit.
M[4]	Execution state that the exception was taken from. A value of 0 indicates AArch64.
M[3:0]	Mode or Exception level that an exception was taken from.

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指令存取。对比ARMv7-A，在ARMv7-A中这些寄存器通常是通过协处理器操作访问的。寄存器名的后缀标识可以访问它的最低的异常级别。如TTBR0_EL2仅可从EL2和EL3访问。

系统寄存器：SCTLR

系统控制寄存器(SCTLR)是系统寄存器其中之一，它控制标准内存、系统设施，并为核心中实现的功能提供状态信息。

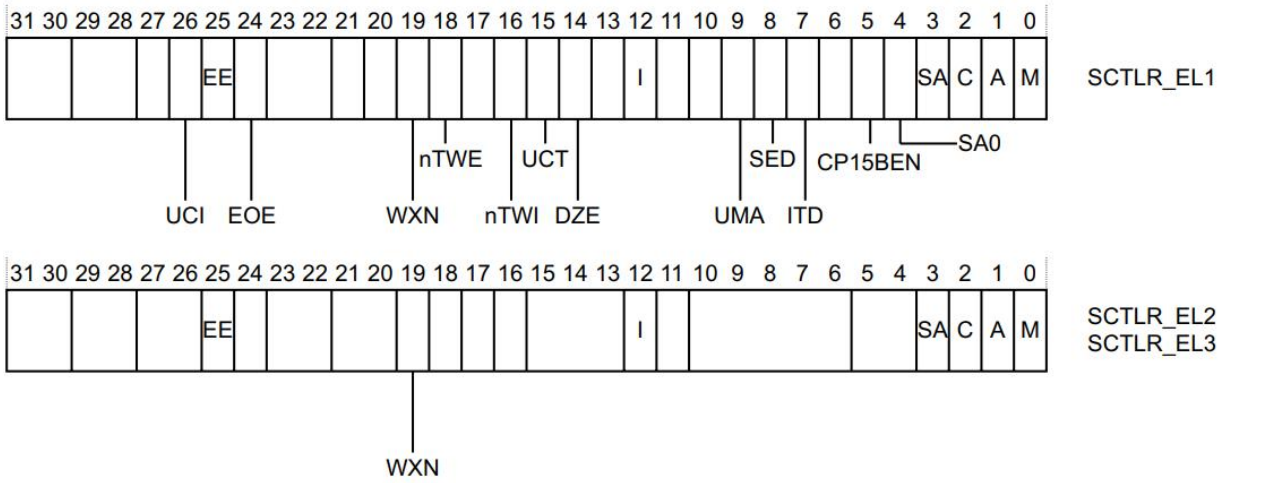


Figure 4-5 SCTLR bit assignments

```
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
```

NEON和浮点寄存器

除了通用寄存器外，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操作的标量和向量操作数。

Part Five

五、ARMv8指令集

A64、A32、T32指令集

ARMv8体系结构中引入的最重要的变化之一是增加了一个64位指令集。这个指令集补充了现有的32位指令集体系结构。这一添加提供了对64位整数寄存器和数据操作的访问，以及使用64位大小的内存指针的能力。新的指令被称为A64，并在AArch64执行态中执行。ARMv8还包括原始的ARM指令集，现在被叫做A32和Thumb(T32)指令集。A32和T32都在AArch32状态下执行，并提供与ARMv7的向后兼容性。

尽管ARMv8-A提供了与32位ARM架构的向后兼容性，A64指令集与旧的指令集编码也是不同。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的速度或能源效率的功能。ARMv8架构还包括对32位指令集(A32和T32)的一些增强，但使用这些特性的代码与旧的ARMv7实现不兼容。A64指令集中的指令仍然是32位长，而不是64位长。

A64中的32位操作

在A64指令集中，大多数整数指令都有两种形式，可以对64位通用寄存器文件中的32位或64位值中的任意一种进行操作 。

指令使用的寄存器名决定了选择的位数。如果寄存器名以X开头，则为64位值；如果寄存器名以W开头，则为32位值。选择32位指令时，若写入W寄存器，会将X寄存器的位[63:32]设置为零。

寻址

在32位内核上执行的单个线程被限制访问4GB的地址空间。大部分可寻址空间都保留给操作系统内核、库代码、外围设备等使用。因此，空间不足意味着程序在执行时可能需要将一些数据映射到内存中或映射到内存外。

使用更大的地址空间和64位指针可以避免这个问题。当处理器可以在单个寄存器中存储64位值时，在程序中访问大量内存就变得简单得多。它还使诸如内存映射文件之类的技术更有吸引力，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可能不够大，不足以包含整个文件。

同时，根据PC进行相对寻址的加载、存储和地址生成范围也扩大为±4GB，仅使用两条指令就可以内联执行，也就是说，不需要从字面池加载偏移量。

指令集切换

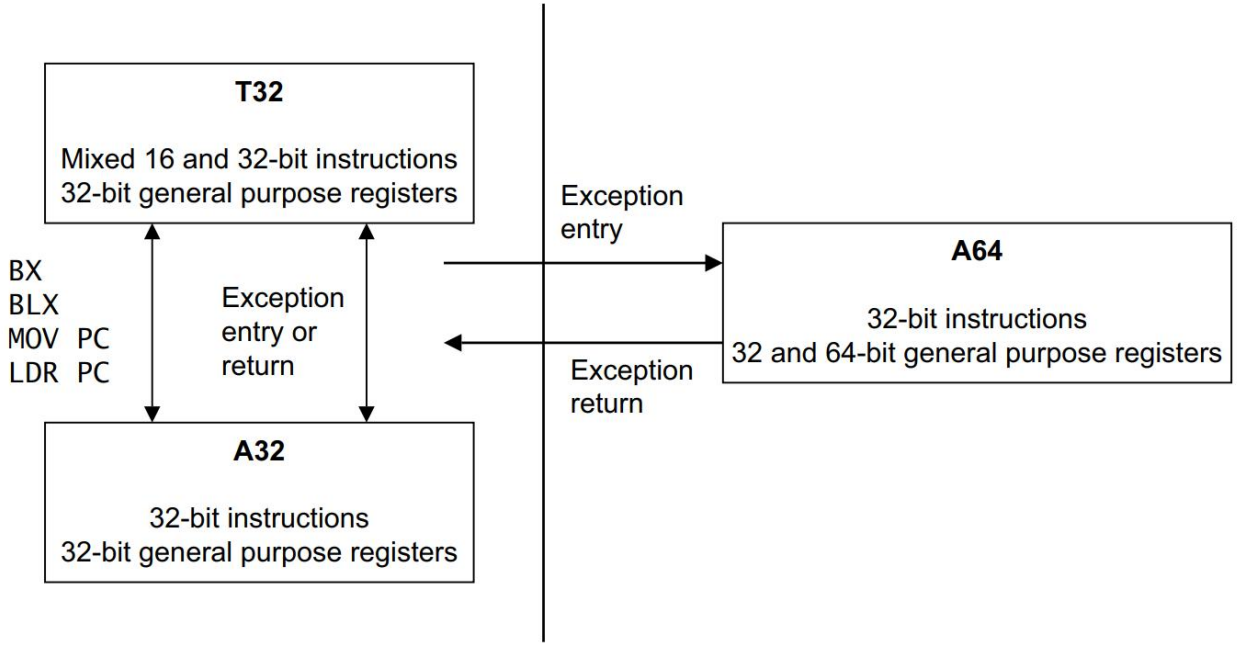


Figure 5-1 Switching between instruction sets

在一个应用程序中不可能使用来自两个执行状态的代码。在ARMv8中，A64和A32或T32指令集之间没有交互，而在A32和T32指令集两者之间有交互 。为ARMv8处理器写的A64代码不能在ARMv7 Cortex-A系列处理器上面运行。然而，为ARMv7-A处理器编写的代码可以在ARMv8处理器上以AArch32执行状态运行。

Part Six

六、A64汇编指令集

指令助记符

A64的汇编语言重载了指令助记符，并根据操作数寄存器名区分不同形式的指令。同一类指令有不同的指令编码，但只需要使用同一个助记符，汇编器就会根据操作数自动选择正确的指令编码。

```
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
```

数据处理指令

数据处理指令通常使用一个目标寄存器和两个源操作数。一般格式是指令后面跟着操作数。

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

数据处理指令包括:

- 算术和逻辑运算
- 移动和移位操作
- 符号扩展和零扩展的指令
- 位和位域操作
- 条件比较和数据处理

数据处理指令例：算术运算ADD

Example 6-1 Arithmetic instructions

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

汇编指令

机器指令

ADD (immediate)

Add (immediate) adds a register value and an optionally-shifted immediate value, and writes the result to the destination register.

This instruction is used by the alias MOV (to/from SP).

31	30	29	28	27	26	25	24	23	22	21 … 10	9 … 5	4 … 0
sf	0	0	1	0	0	0	1	0	sh	imm12	Rn	Rd
	op	S										

内存访问指令

与所有之前的ARM处理器一样，ARMv8架构是一个Load/Store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。数据必须首先装入寄存器，修改，然后再存储到内存中。程序必须指定一个地址、要传输的数据的大小和源寄存器或目的寄存器。

Load指令的一般形式如下：

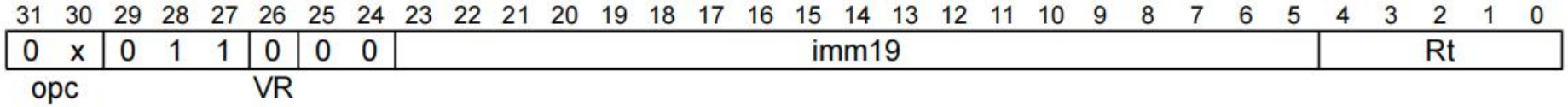
```
LDR Rt, <addr>
```

Store指令的一般形式如下：

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

LDR (literal)

Load Register (literal) calculates an address from the PC value and an immediate offset, loads a word from memory, and writes it to a register. For information about addressing modes, see *Load/Store addressing modes*.



流程控制指令

A64指令集提供了许多不同种类的分支指令。对于简单的相对分支，即距当前地址有偏移量的分支，使用B指令。无条件简单相对分支可以从当前程序计数器位置向后或向前跳转，最高可达128MB。有条件简单相对分支(条件代码被附加到B)的范围更小，为±1MB。

调用子例程的过程中如有必要将返回地址存储在链接寄存器中(X30)，使用BL指令。它无有条件版本。BL表现为B指令，额外附加的效果是将返回地址(即BL后的指令地址)存储在寄存器X30中。

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.

系统控制指令和其他指令

A64指令集中还包含以下系统控制指令：

- 异常处理指令
- 系统寄存器访问指令
- 调试指令
- 提示指令，使用在有电源管理应用的系统中

此外还有NEON指令、浮点指令和作为可选扩展的密码指令（软件硬化）。

Part Seven

七、AArch64浮点和NEON

NEON技术

ARM的SIMD（Single Instruction Multiple Data，单指令流多数据流）架构及其相关实现和支持软件通常被称为NEON技术。AArch32（相当于ARMv7的NEON指令）和AArch64都有NEON指令集，他们都可以显著加快大型数据集上的重复操作，这在媒体编解码器等应用程序中很有用。

AArch64的NEON架构使用32个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。浮点数和NEON在所有标准ARMv8实现中都是必需的。然而，针对特定市场的实现可能支持以下组合：

- 没有浮点或NEON。
- 具有异常捕获的全浮点和SIMD支持。
- 完全浮点和SIMD支持，但没有异常捕获。

NEON和浮点架构

NEON寄存器的内容是相同数据类型元素的向量。一个向量被划分为多个车道，每个车道包含一个叫做元素的数据值。NEON向量中的车道数取决于向量的大小和向量中数据元素的大小。每条NEON指令会并行执行n个操作，其中n是输入向量划分的车道数。不能从一个车道进位或溢出到另一个车道。

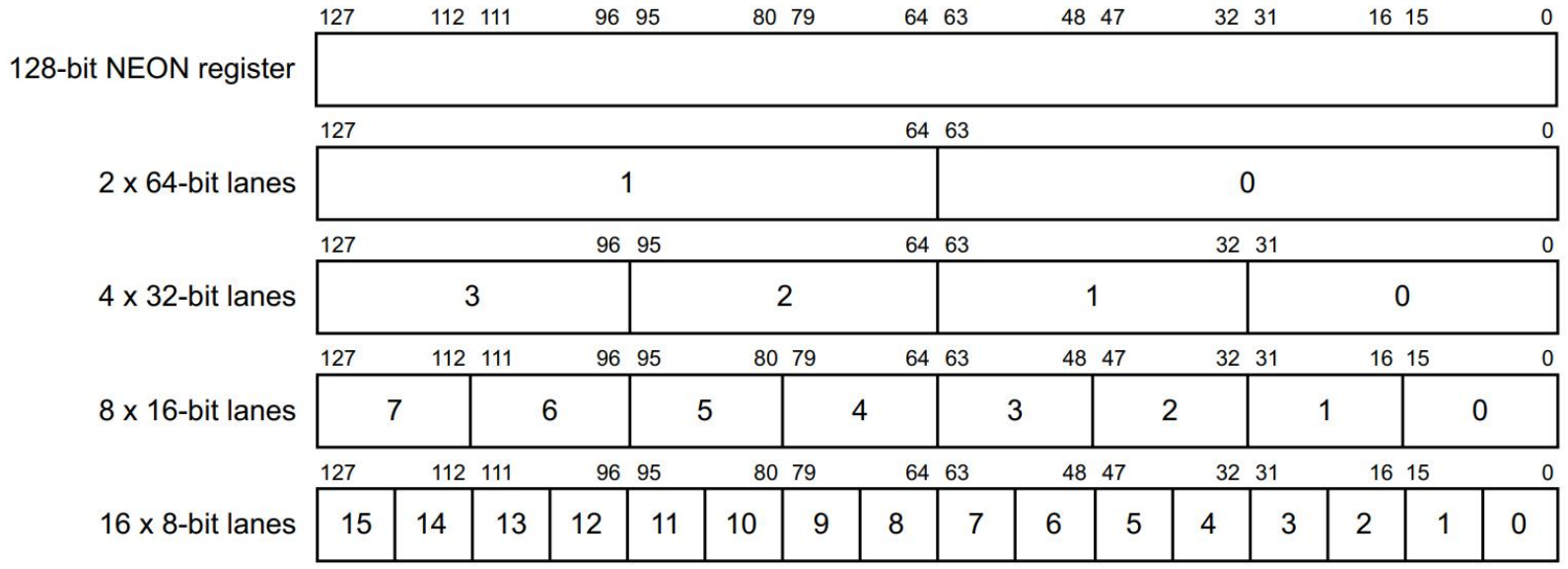


Figure 7-1 Divisions of the V register

Part Eight

八、总结与回顾

总结与回顾

一、引言与概念辨析

- ARM Cortex-A/R/M
- AArch64 与 AArch32 与 ARMv8

二、架构与处理器

- 发展历程：A32、Thumb16、Thumb-2（T32）
- ARMv8架构新变化
- 支持ARMv8架构的处理器：Cortex-A53、Cortex-A57

三、异常级别与执行状态

- 异常级别：EL0~EL3
- 安全状态：安全世界和非安全世界、安全监视器
- 执行状态：AArch64 和 AArch32
- 级别转移与状态转化

四、寄存器配置

- 通用寄存器：31 × 64,、W寄存器和X寄存器
- 专用寄存器：ZR、PC、SP、SPSR
- 系统寄存器：MSR/MRS、SCTLR
- NEON和浮点寄存器：V0~V31

五、ARMv8指令集

- A64、A32、T32指令集
- A64中的32位操作
- 寻址
- 指令集切换

六、A64汇编指令集

- 指令助记符
- 数据处理指令：ADD及其机器指令
- 内存访问指令：LDR/STR及其机器指令
- 流程控制指令
- 系统控制指令和其他指令

七、AArch64浮点和NEON技术

- NEON技术：SIMD架构及其实现、32 × 128寄存器
- NEON和浮点架构：向量寄存器、车道划分

感谢观看与聆听

22计科8第一小组

陈煜祺 李康桂 冯程羿 陈卓麟

李智鹏 林治同 刘怀安 林冠鹏 罗序童