第3章 chapter 3

ARMv8 架构基础知识

ARMv8 架构是 ARM 体系结构的一种，它具有引人注目的特点：首次引入了 64 位指令集，从而能够处理更大的内存空间和更复杂的数据计算。此外，ARMv8 架构还引入了全新的执行模式 AArch64，它能够运行 64 位指令集，支持更广泛的操作模式和高级特性。目前，基于 ARMv8 架构设计的处理器在各个领域都发挥着重要的作用。

本章将从 ARMv8 架构的基础概念、寄存器组、A64 指令集、ARM64 异常处理以及 ARM64 内存管理等方面介绍 ARMv8 架构，旨在帮助读者更好地理解 ARMv8 架构。

3.1 ARMv8 架 构

ARMv8 是 ARM 公司推出的处理器架构，广泛应用于移动设备、嵌入式系统、物联网设备和云服务器等领域，了解 ARMv8 架构可以理解和应用这些设备和系统。许多应用和系统都是基于 ARMv8 架构开发的，学习 ARMv8 架构可以更好地理解和开发这些应用和系统。同时，了解 ARMv8 架构也可以提高编程技能和工作竞争力。本节主要对 ARMv8 中涉及的主要概念进行梳理，介绍基于 ARMv8 架构设计的处理器的运行状态和其支持的数据宽度。

3.1.1 ARMv8 架构介绍

1. ARMv8-A 架构特性

ARMv8-A 是 ARM 公司发布的第一代支持 64 位处理器的指令集和架构，它在扩充 64 位寄存器的同时提供了对上一代架构指令集的兼容，因此它提供了运行 32 位和 64 位应用程序的环境。

ARMv8-A 架构除了提高了处理能力，还引入了很多吸引人的新特性。

- 具有超大物理地址空间，提供超过 4GB 物理内存。
- 具有 64 位宽的虚拟地址空间。32 位宽的虚拟地址空间只能提供 4GB 大小的虚拟地址空间访问，这极大地限制了桌面操作系统和服务器等的性能发挥。64 位宽的虚拟地址空间可以提供更大的访问空间。
- 提供 31 个 64 位宽的通用寄存器，可以减少对栈的访问，从而提高性能。

- 提供 16KB 和 64KB 的页面，有助于降低 TLB 的未命中率(miss rate)。
- 具有全新的异常处理模型，有助于降低操作系统和虚拟化的实现复杂度。
- 具有全新的加载-获取指令(load-acquire instruction)、存储-释放指令(store-release instruction)，专门为 C++ 11、C11 以及 Java 内存模型设计。

2. 采用 ARMv8 架构的常见处理器内核

下面介绍市面上常见的采用 ARMv8 架构的处理器(简称 ARMv8 处理器)内核。

(1) Cortex-A53 处理器内核。ARM 公司第一款采用 ARMv8-A 架构的处理器内核，专门为低功耗设计的处理器。通常可以使用 1～4 个 Cortex-A53 处理器组成一个处理器簇，或者和 Cortex-A57 或 Cortex-A72 等高性能处理器组成大/小核架构。

(2) Cortex-A57 处理器内核。采用 64 位 ARMv8-A 架构的处理器内核，而且通过 AArch32 执行状态，保持与 ARMv7 架构完全后向兼容。除了 ARMv8 架构的优势之外，Cortex-A57 还提高了单个时钟周期的性能，比高性能的 Cortex-A15 高出 20%～40%；它还改进了二级高速缓存的设计和内存系统的其他组件，极大地提高了性能。

(3) Cortex-A72 处理器内核。2015 年年初正式发布的基于 ARMv8-A 架构，并在 Cortex-A57 处理器上做了大量优化和改进。在相同的移动设备电池寿命限制下，Cortex-A72 相较于基于 Cortex-A15 的设备具有 3.5 倍的性能提升，展现出了优异的整体功耗效率。

3.1.2 ARMv8 基础概念

ARM 处理器实现的是精简指令集架构。在 ARMv8-A 架构中有以下基本概念和定义。

(1) 处理机(processing element，PE)。在 ARM 公司的官方技术手册中提到的一个概念，把处理器处理事务的过程抽象为处理机。

(2) 执行状态(execution state)。处理器运行时的环境，包括寄存器的位宽、支持的指令集、异常模型、内存管理以及编程模型等。ARMv8 架构定义了两个执行状态。

① AArch64：64 位的执行状态。

- 提供 31 个 64 位的通用寄存器。
- 提供 64 位的程序计数器(program counter，PC)指针寄存器、栈指针(stack pointer，SP)寄存器以及异常链接寄存器(exception link register，ELR)。
- 提供 A64 指令集。
- 定义 ARMv8 异常模型，支持 4 个异常等级，即 EL0～EL3。
- 提供 64 位的内存模型。
- 定义一组处理器状态(PSTATE)用来保存 PE 的状态。

② AArch32：32 位的执行状态。

- 提供 13 个 32 位的通用寄存器，再加上 PC 指针寄存器、SP 寄存器、链接寄存器(link register，LR)。
- 支持两套指令集，分别是 A32 和 T32 指令集(Thumb 指令集)。
- 支持 ARMv7-A 异常模型，基于 PE 模式并映射到 ARMv8 的异常模型中。

- 提供32位的虚拟内存访问机制。
- 定义一组PSTATE用来保存PE的状态。

(3) ARMv8指令集。ARMv8架构根据不同的执行状态提供不同指令集的支持。

- A64指令集：运行在AArch64状态下，提供64位指令集支持。
- A32指令集：运行在AArch32状态下，提供32位指令集支持。
- T32指令集：运行在AArch32状态下，提供16和32位指令集支持。

(4) 系统寄存器命名。在AArch64状态下，很多系统寄存器会根据不同的异常等级提供不同的变种寄存器。

3.1.3 ARMv8处理器的运行状态

ARMv8处理器支持两种执行状态——AArch64状态和AArch32状态。AArch64状态是ARMv8新增的64位执行状态，而AArch32是为了兼容ARMv7架构的32位执行状态。当处理器运行在AArch64状态下时，运行A64指令集；而当运行在AArch32状态下时，可以运行A32指令集或者T32指令集。

如图3-1所示，AArch64状态的异常等级(exception level)决定了处理器当前运行的特权级别，类似于ARMv7架构中的特权等级。

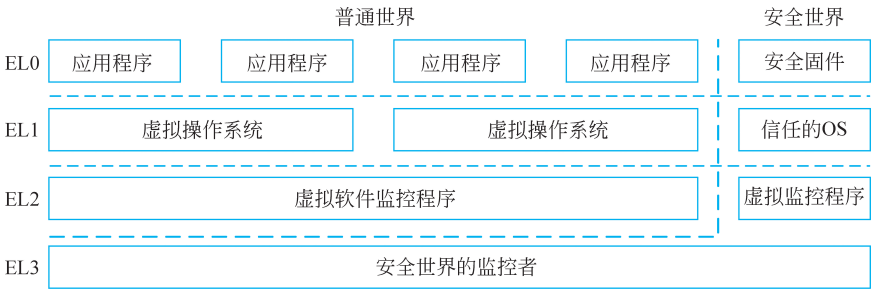

图3-1 AArch64状态的异常等级

- EL0：用户特权，用于运行普通用户程序。
- EL1：系统特权，通常用于运行操作系统。
- EL2：运行虚拟化扩展的虚拟监控程序(hypervisor)。
- EL3：运行安全世界中的安全监控器(secure monitor)。

ARMv8架构允许切换应用程序的运行模式。例如在一个运行64位操作系统的ARMv8处理器中，可以同时运行A64指令集的应用程序和A32指令集的应用程序。但是在一个运行32位操作系统的ARMv8处理器中就不能运行A64指令集的应用程序了。当需要运行A32指令集的应用程序时，需要通过一条管理员调用(supervisor call，SVC)指令切换到EL1，操作系统会做任务的切换并返回AArch32的EL0中，这时操作系统就为这个应用程序准备好了AArch32的运行环境。

3.1.4 ARMv8架构支持的数据宽度

ARMv8支持如下几种数据宽度。

- 字节(byte)：8 位。
- 半字(halfword)：16 位。
- 字(word)：32 位。
- 双字(doubleword)：64 位。
- 四字(quadword)：128 位。

不对齐访问有两种情况，一种是指令不对齐访问，另一种是数据不对齐访问。A64 指令集要求指令存放的位置必须以字(word，32 位宽)为单位对齐。访问一条存储位置不以字为单位对齐的指令会导致 PC 对齐异常(PC alignment fault)。

对于数据访问，需要区分不同的内存类型。内存类型是设备内存的不对齐访问会触发一个对齐异常(alignment fault)。

对于访问普通内存，除了独占加载/独占存储(load-exclusive/store-exclusive)指令或者加载-获取/存储-释放(load-acquire/store-release)指令外，对于其他加载或者存储单个或多个寄存器的所有指令，如果访问地址和要访问数据不对齐，则按照以下两种情况进行处理。

- 若对应的异常等级中的 SCTLR_Elx.A 设置为 1，则说明打开了地址对齐检查功能，那么会触发一个对齐异常。
- 若对应的异常等级中的 SCTLR_Elx.A 设置为 0，那么处理器支持不对齐访问。

当然，处理器对不对齐访问也有一些限制。

- 不能保证单次原子地完成访问，可能多次复制。
- 不对齐访问比对齐访问需要更多的处理时间。
- 不对齐访问可能会造成中止(abort)。

3.2 ARMv8 寄存器

了解 ARMv8 寄存器的结构、数量和使用方式，对理解指令集的执行、优化代码、调试程序以及进行性能分析非常有必要。因此本节通过介绍 ARMv8 架构的主要寄存器：通用寄存器、处理状态寄存器、特殊寄存器和系统寄存器，帮助读者认识 ARMv8 寄存器。

3.2.1 通用寄存器

AArch64 运行状态支持 31 个 64 位的通用寄存器，分别是 X0～X30 寄存器，而 AArch32 状态支持 16 个 32 位的通用寄存器。

通用寄存器除了用于数据运算和存储之外，还可以在函数调用过程中起到特殊作用，ARM64 架构的函数调用标准和规范对此有所约定，如图 3-2 所示。

在 AArch64 状态下，使用 X 表示 64 位通用寄存器，如 X0、X30 等。另外，还可以使用 W 表示 X 寄存器的低 32 位的数据，如 W0 表示 X0 寄存器的低 32 位数据，W1 表示 X1 寄存器的低 32 位数据，如图 3-3 所示。

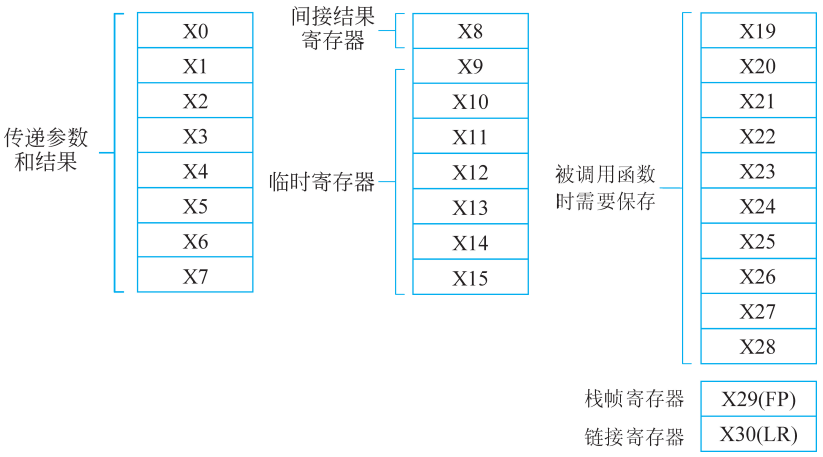

图 3-2 AArch64 状态的 31 个通用寄存器

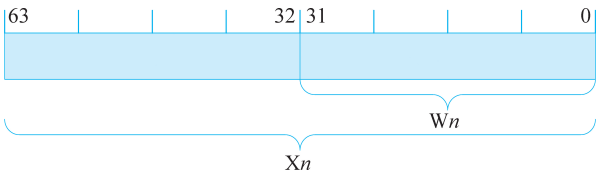

图 3-3 64 位通用寄存器和低 32 位数据

3.2.2 处理器状态寄存器

在 ARMv7 架构中，使用当前程序状态寄存器(current program status register，CPSR)表示当前的处理器状态(processor state)，而在 AArch64 中使用 PSTATE 寄存器表示，如表 3-1 所示。

表 3-1 PSTATE 寄存器

分 类	字 段	描 述
条件标志位	N	负数标志位 在结果是有符号的二进制代码的情况下，如果结果为负数，则 N=1；如果结果为非负数，则 N=0
	Z	0 标志位 如果结果为 0，则 Z=1；如果结果为非 0，则 Z=0
	C	进位标志位 当发生无符号数溢出时，C=1 其他情况下，C=0
	V	有符号数溢出标志位 对于加/减指令，在操作数和结果是有符号的整数时，如果发生溢出，则 V=1；如果未发生溢出，则 V=0 对于其他指令，V 通常不发生变化

续表

分　类	字　段	描　述
运行状态控制	SS	软件单步。该位为 1，说明在异常处理中使能了软件单步功能
	IL	不合法的异常状态
	nRW	当前执行模式 0：处于 AArch64 状态 1：处于 AArch32 状态
	EL	当前异常等级 0：表示 EL0 1：表示 EL1 2：表示 EL2 3：表示 EL3
	SP	选择 SP 寄存器。当运行在 EL0 时，处理器选择 EL0 的 SP 寄存器，即 SP_EL0；当处理器运行在其他异常等级时，处理器可以选择使用 SP_EL0 或者对应的 SP_ELn 寄存器
异常掩码标志位	D	调试位。使能该位可以在异常处理过程中打开调试断点和软件单步等功能
	A	用来屏蔽系统错误(SError)
	I	用来屏蔽 IRQ
	F	用来屏蔽 FIQ
访问权限	PAN	特权不访问(privileged access never)位是 ARMv8.1 的扩展特性 1：在 EL1 或者 EL2 访问属于 EL0 的虚拟地址时会触发一个访问权限错误 0：不支持该功能，需要软件模拟
	UAO	用户特权访问覆盖标志位，是 ARMv8.2 的扩展特性 1：当运行在 EL1 或者 EL2 时，没有特权的加载存储指令可以和有特权的加载存储指令一样访问内存，如 LDTR 指令 0：不支持该功能

3.2.3　特殊寄存器

ARMv8 架构除了支持 31 个通用寄存器之外，还提供多个特殊寄存器，如图 3-4 所示。

1. 零寄存器

ARMv8 架构提供两个零寄存器(zero register)，这些寄存器的内容全是 0，可以用作源寄存器，也可以用作目标寄存器。WZR 寄存器是 32 位的零寄存器，XZR 是 64 位的零寄存器。

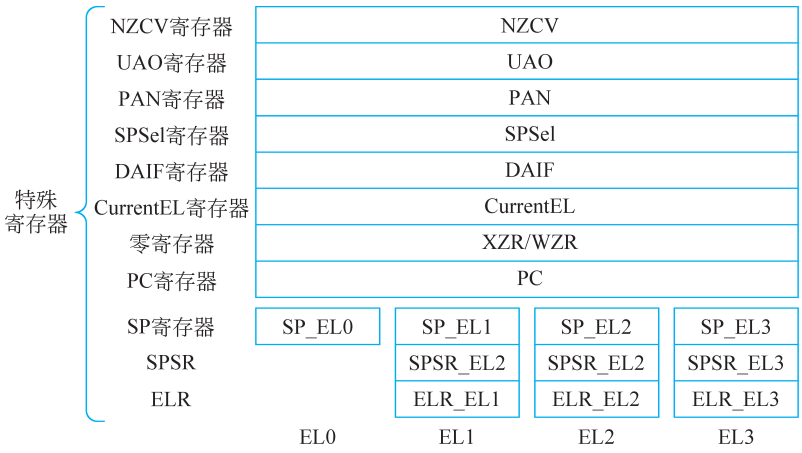

图 3-4 特殊寄存器

2. PC 寄存器

PC 寄存器通常用来指向当前运行指令的下一条指令的地址，用于控制程序中指令的运行顺序，但是编程人员不能通过指令直接访问它。

3. SP 寄存器

ARMv8 架构支持 4 个异常等级，每一个异常等级都有一个专门的 SP 寄存器 SP_ELn，例如处理器运行在 EL1 时选择 SP_EL1 寄存器作为 SP 寄存器。

- SP_EL0：EL0 下的 SP 寄存器。
- SP_EL1：EL1 下的 SP 寄存器。
- SP_EL2：EL2 下的 SP 寄存器。
- SP_EL3：EL3 下的 SP 寄存器。

当处理器运行在比 EL0 高的异常等级时，处理器可以访问如下寄存器。

- 当前异常等级对应的 SP 寄存器 SP_ELn。
- EL0 对应的 SP 寄存器 SP_EL0 可以当作一个临时寄存器，如 Linux 内核里使用该寄存器存放进程的 task_struct 数据结构的指针。

当处理器运行在 EL0 时，它只能访问 SP_EL0，而不能访问其他高级的 SP 寄存器。

4. 保存处理状态寄存器

当运行一个异常处理器时，处理器的处理状态会保存到保存处理状态寄存器（saved process status register，SPSR）中，这个寄存器非常类似于 ARMv7 架构中的 CPSR。当异常将要发生时，处理器会把 PSTATE 寄存器的值暂时保存到 SPSR 中；当异常处理完成并返回时，再把 SPSR 的值恢复到 PSTATE 寄存器。SPSR 的格式如图 3-5 所示，SPSR 的重要字段如表 3-2 所示。

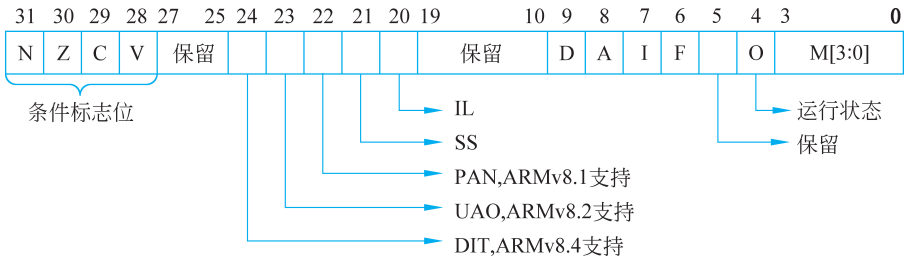

图 3-5 SPSR 的格式

表 3-2 SPSR 的重要字段

字 段	描 述
N	负数标志位
Z	零标志位
C	进位标志位
V	有符号数溢出标志位
DIT	与数据无关的指令时序(data independent timing),ARMv8.4 的扩展特性
UAO	用户特权访问覆盖标志位,ARMv8.2 的扩展特性
PAN	特权模式禁止访问(privileged access never)位,ARMv8.1 的扩展特性
SS	表示是否使能软件单步功能。若该位为 1,说明在异常处理中使能了软件单步功能
IL	不合法的异常状态
D	调试位。使能该位可以在异常处理过程中打开调试断点和软件单步等功能
A	用来屏蔽系统错误
I	用来屏蔽 IRQ
F	用来屏蔽 FIQ
M[4]	用来表示异常处理过程中处于哪个执行状态,若为 0,表示 AArch64 状态
M[3:0]	异常模式

5. ELR 寄存器

该寄存器存放了异常返回地址。

6. CurrentEL 寄存器

该寄存器表示 PSTATE 寄存器中的 EL 字段,其中保存了当前异常等级。使用 MRS 指令可以读取当前异常等级。

- 0:表示 EL0。
- 1:表示 EL1。
- 2:表示 EL2。

- 3：表示EL3。

7. DAIF寄存器

该寄存器表示PSTATE寄存器中的{D,A,I,F}字段。

8. SPSel寄存器

该寄存器表示PSTATE寄存器中的SP字段，用于在SP_EL0和SP_ELn中选择SP寄存器。

9. PAN寄存器

该寄存器表示PSTATE寄存器中的PAN(privileged access never，特权禁止访问)字段。可以通过MRS和MSR指令设置PAN寄存器。

10. UAO寄存器

该寄存器表示PSTATE寄存器中的UAO(user access override，用户访问覆盖)字段。可以通过MRS和MSR指令设置UAO寄存器。

11. NZCV寄存器

该寄存器表示PSTATE寄存器中的{ N,Z,C,V }字段。

3.2.4 系统寄存器

除了上面介绍的通用寄存器和特殊寄存器之外，ARMv8架构还定义了很多的系统寄存器，通过访问和设置这些系统寄存器可以完成对处理器不同功能的配置。在ARMv7架构中，需要通过访问CP15协处理器间接访问这些系统寄存器，而在ARMv8架构中没有协处理器，可直接访问系统寄存器。ARMv8架构支持如下7类系统寄存器。

- 通用系统控制寄存器。
- 调试寄存器。
- 性能监控寄存器。
- 活动监控寄存器。
- 统计扩展寄存器。
- RAS寄存器。
- 通用定时器寄存器。

系统寄存器支持不同异常等级的访问，通常系统寄存器会使用Reg_ELn的方式表示。

- Reg_EL1：处理器处于EL1、EL2以及EL3时可以访问该寄存器。
- Reg_EL2：处理器处于EL2和EL3时可以访问该寄存器。
- 大部分系统寄存器不支持处理器处于EL0时访问，但也有一些例外，如CTR_EL0寄存器。

程序可以通过 MRS 和 MSR 指令访问系统寄存器。

```
MRS X0, TTBR0_EL1          //把 TTBR0_EL1 的值复制到 x0 寄存器
MSR TTBR0_EL1, X0          //X0 寄存器的值复制到 TTBR0_EL1
```

3.3 A64 指令集

指令集是处理器体系结构设计的重点之一。ARM 公司定义和实现的指令集一直在变化和发展中。ARMv8 体系结构最大的改变是增加了一个新的 64 位指令集，这是早前 ARM 指令集的有益补充和增加，它可以处理 64 位宽的寄存器和数据，并且使用 64 位的指针访问内存。这个新的指令集称为 A64 指令集，运行在 AArch64 状态。ARMv8 兼容旧的 32 位指令集——A32 指令集，它运行在 AArch32 状态。A64 和 A32 指令集并不兼容，它们是两套不同的指令集，指令编码是不一样的。需要注意的是，A64 指令集支持 64 位宽的数据和地址寻址，但其编码宽度是 32 位，而不是 64 位。A64 指令集具有以下特点：具有特有的指令编码格式；只能运行在 AArch64 状态；指令的宽度为 32 位。

下面以前变基模式的 LDR 指令为例，介绍 A64 指令集的编码风格，如图 3-6 所示。

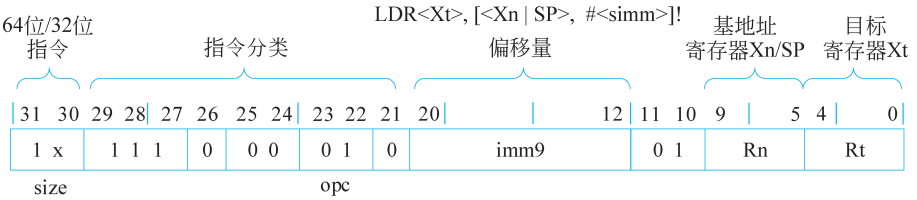

图 3-6 前变基模式的 LDR 指令的编码

第 0～4 位为 Rt 字段，用来描述目标寄存器 Xt，可以从 X0～X30 中选择。

第 5～9 位为 Rn 字段，用来描述基地址寄存器 Xn，可以从 X0～X30 中选择，也可以选择 SP 寄存器作为第 31 个寄存器。

第 12～20 位为 imm9 字段，用于偏移量 imm。

第 21～29 位用于指令分类。

第 30～31 位为 size 字段，当 size 为 0b11 时，表示 64 位宽数据；当 size 为 0b10 时，表示 32 位宽数据。

A64 指令集可以分为如下几类：

- 内存加载和存储指令；
- 多字节内存加载和存储指令；
- 算术和移位指令；
- 移位操作指令；
- 位操作指令；
- 条件操作指令；
- 跳转指令；
- 独占访问指令；

- 内存屏障指令；
- 异常处理指令；
- 系统寄存器访问指令。

3.3.1 加载与存储指令

和早期的ARM体系结构一样，ARMv8体系结构基于指令加载和存储体系结构。在这种体系结构下，所有的数据处理都需要在通用寄存器中完成，而不能直接在内存中完成。因此，首先把待处理数据从内存加载到通用寄存器，然后进行数据处理，最后把数据写入内存。

常见的内存加载指令是LDR指令，内存写入指令是STR指令。LDR指令和STR指令的基本格式如下。

```
LDR 目标寄存器, <存储器地址>          //把存储器地址中的数据加载到目标寄存器中
STR 源寄存器, <存储器地址>            //把源寄存器的数据存储到存储器中
```

1. 基地址模式的寻址

基地址模式首先是使用寄存器的值表示一个地址，然后把这个内存地址的内容加载到通用寄存器中。

以下指令以Xn寄存器中的内容作为内存地址，加载此内存地址的内容到Xt寄存器。

```
LDR Xt, [Xn]
```

以下指令是把Xt寄存器中的内容存储到Xn寄存器的内存地址中。

```
STR Xt, [Xn]
```

2. 基地址加偏移量模式的寻址

基地址加偏移量模式是指在基地址的基础上再加上偏移量，从而计算内存地址，并把这个内存地址的值加载到通用寄存器中，偏移量可以是正数，也可以是负数。

以下指令是把Xn寄存器的内容加上一个偏移量(offset必须是8的倍数)，以相加的结果作为基地址，加载此内存地址的内容到Xt寄存器。

```
LDR Xt, [Xn, #offset]
```

基地址加偏移量模式的存储指令格式如下。该指令是将Xt寄存器中的值存储到以Xn寄存器的值加一个偏移量(offset必须是8的倍数)表示的地址中。

```
STR Xt, [Xn, #offset]
```

3. 基地址扩展模式的寻址

基地址扩展模式的命令如下。

```
LDR <Xt>, [<Xn>, (<Xm>){,<extend>{<amount>}}]
STR <Xt>, [<Xn>, (<Xm>){,<extend>{<amount>}}]
```

Xt：目标寄存器。

Xn：基地址寄存器。

Xm：用来表示偏移的寄存器。

extend：扩展/移位指示符，默认是 LSL，也可以是 UXTW、SXTW、SXTX。

amount：索引偏移量，amount 的值只能是 0 或者 3，如果是其他值，汇编器将报错。

【例 3-1】 如下代码使用了基于基地址加偏移量模式。

```
LDR X0, [X1]              //内存地址为 X1 寄存器的值，加载此内存地址的值到 X0 寄存器

LDR X0, [X1,#8]
          //内存地址为 X1 寄存器的值再加上偏移量(8)，加载此内存地址的值到 X0 寄存器

LDR X0,[X1,X2]
          //内存地址为 X1 寄存器的值加 X2 寄存器的值，加载此内存地址的值到 X0 寄存器

LDR X0,[X1,X2,LSL #3]
          //内存地址为 X1 寄存器的值加(X2 寄存器的值<<3)，加载此内存地址的值到 X0
          寄存器

LDR X0,[X1,W2,SXTW]
          //先对 W2 的值做有符号的扩展，和 X1 寄存器的值相加后，将结果作为内存地址，
          //加载此内存地址的值到 X0 寄存器

LDR X0,[X1,W2,SXTW #3]
          //先对 W2 的值做有符号的扩展，然后左移 3 位，和 X1 寄存器的值相加后，将
          //结果作为内存地址，加载此内存地址的值到 X0 寄存器
```

4. 前变基模式的寻址

前变基模式指先更新偏移量地址，后访问内存地址。前变基模式的指令格式如下。首先更新 Xn|SP 寄存器的值为 Xn|SP 寄存器的值加 imm，然后以新的 Xn|SP 的值为内存地址，加载该内存地址的值到 Xt 寄存器。

```
LDR <Xt>, [<Xn|SP>, #<imm>]!
```

以下指令首先更新 Xn|SP 的值为 Xn|SP 寄存器的值加 imm，然后把 Xt 寄存器的值存储到 Xn|SP 寄存器的新值为地址的内存单元中。

```
STR <Xt>, [<Xn|SP>, #<imm>]!
```

5. 后变基模式的寻址

后变基模型指先访问内存地址，后更新偏移量地址。首先以 Xn|SP 寄存器的值为内存地址，取该内存地址上的值到 Xt 寄存器，再更新 Xn|SP 寄存器的值为 Xn|SP 寄存器的值加 imm。

```
LDR <Xt>, [<Xn|SP>], #<imm>
```

以下指令先将 Xt 寄存器的值存储到以 Xn|SP 寄存器的值为地址的内存单元中，然后更新 Xn|SP 寄存器的值为 Xn|SP 寄存器的值加 imm。

```
STR <Xt>, [<Xn|SP>], #<imm>
```

【例 3-2】 如下代码使用了前变基模式和后变基模式。

```
LDR X0, [X1, #8]!   //前变基模式。先更新 X1 寄存器的值为 X1 寄存器的值加 8
                    //然后以新的 X1 寄存器的值为内存地址，加载该内存地址的值到 X0 寄存器中

LDR X0, [X1], #8
     //后变基模式。以 X1 寄存器的值为内存地址，加载该内存地址的值到 X0 寄存器，然后更
                    //新 X1 寄存器的值为 X1 寄存器的值加 8
```

6. PC 相对地址模式的寻址

汇编代码中常常会使用标签(label)标记代码片段。LDR 指令还提供一种访问标签的地址模式，指令的格式如下。这条指令驱动 label 所在内存地址的内容到 Xt 寄存器中。但是这个 label 必须在当前 PC 地址前后 1MB 的范围内，如果超过这个范围，则汇编器会报错。

```
LDR <Xt>, <label>
```

【例 3-3】 如下 LDR 指令会把标签 my_data 的数据读出来。

```
my_data:
          .word 0x40

ldr x0, my_data          //最终 X0 寄存器的值为 0x40
```

【例 3-4】 假设当前 PC 值为 0x806E4，那么这条 LDR 指令读取 0x806E4＋0x20 地址的内容到 X6 寄存器中。

```
#define MY_LABEL 0x20

ldr x6,MY_LABEL
```

7. 多字节内存加载和存储

A32 指令集提供 LDM 和 STM 指令以实现多字节内存加载与存储，而 A64 指令集不再提供 LDM 和 STM 指令，而是提供 LDP 和 STP 指令。LDP 和 STP 指令支持 3 种寻址模式。

基地址偏移量模式 LDP 指令的格式如下，它以 Xn|SP 寄存器的值为基地址，然后读取 Xn|SP 寄存器的值加 imm 地址的值到 Xt1 寄器，读取 Xn|SP 寄存器的值加 imm＋8 地址的值到 Xt2 寄存器中。

```
LDP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

基地址偏移量模式 STP 指令的格式如下，它以 Xn|SP 寄存器的值为基地址，然后把 Xt1 寄存器的内容存储到[Xn|SP+imm]处，把 Xt2 寄存器的内容存储到[Xn|SP+imm+8]处。

```
STP <Xt1>, <Xt2>, [<Xn|SP>{, #<imm>}]
```

前变基模式 LDP 指令的格式如下，它先计算 Xn 寄存器的值加 imm，并存储到 Xn 寄存器中，然后以 Xn 寄存器的最新值作为基地址，读取 Xn 寄存器的值加 imm 地址的值到 Xt1 寄存器，读取[Xn+imm+8]的值到 Xt2 寄存器中。Xn 寄存器可以使用 SP 寄存器。

```
LDP <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]!
```

前变基模式 STP 指令的格式如下，它先计算 Xn 寄存器的值加 imm，并存储到 Xn 寄存器中，然后以 Xn 寄存器的最新值作为基地址，把 Xt1 寄存器的内容存储到 Xn 内存地址处，把 Xt2 寄存器的值存储到 Xn 寄存器的值加 8 对应的内存地址处。

```
STP <Xt1>, <Xt2>, [<Xn|SP>, #<imm>]!
```

后变基模式 LDP 指令的格式如下，它以 Xn 寄存器的值为基地址，读取[Xn+imm]的值到 Xt1 寄存器，读取[Xn+imm+8]的值到 Xt2 寄存器中，最后更新 Xn 寄存器。Xn 寄存器可以使用 SP 寄存器。

```
LDP <Xt1>, <Xt2>, [<Xn|SP>], #<imm>
```

后变基模式 STP 指令格式如下，它以 Xn 寄存器的值为基地址，把 Xt1 寄存器的内容存储到 Xn 寄存器的值加 imm 对应的内存地址处，Xt2 寄存器的值存储到[Xn+imm+8]处，并更新 Xn 寄存器。Xn 寄存器可以使用 SP 寄存器。

```
STP <Xt1>, <Xt2>, [<Xn|SP>], #<imm>
```

【例 3-5】 如下代码使用了基地址偏移量模式和前变基模式。

```
LDP X3, X7, [X0]
    //以 X0 寄存器的值为内存地址，加载此内存地址的值到 X3 寄存器中，然后以 X0 寄存器的
    //值加 8 作为内存地址，加载此内存地址的值到 X7 寄存器中

LDP X1, X2, [X0, #0x10]!
    //前变基模式。先计算 X0+0x10 作为 X0 的值，然后以 X0 寄存器的值作为内存地址
    //加载此内存地址的值到 X1 寄存器中，接着以 X0 寄存器的值加 8 作为内存地址
    //加载此内存地址的值到 X2 寄存器中

STP X1, X2, [X4]
    //存储 X1 寄存器的值到地址为 X4 寄存器的值的内存单元中
    //然后存储 X2 寄存器的值到地址为 X4 寄存器的值加 8 的内存单元中
```

8. 不同位宽的加载与存储指令

LDR 和 STR 指令根据不同的数据位宽有多种变种，如表 3-3 所示。

表 3-3 不同位宽的 LDR 和 STR 指令

指 令	描 述
LDR	数据加载指令
LDRSW	有符号的数据加载指令,单位为字
LDRB	数据加载指令,单位为字节
LDRSB	有符号的数据加载指令,单位为字节
LDRH	数据加载指令,单位为半字
LDRSH	有符号的数据加载指令,单位为半字
STR	数据存储指令
STRB	数据存储指令,单位为字节
STRH	数据存储指令,单位为半字

9. 不可扩展的加载和存储指令

LDR 指令中的基地址加偏移量模式为可扩展模式,即偏移量按照数据大小来扩展且是正数,取值范围为 0～32760。A64 指令集还支持一种不可扩展模式的加载和存储指令,即偏移量只能按照字节来扩展,可以是正数或者负数,取值范围为－256～255,例如 LDUR 指令。因此,可扩展模式和不可扩展模式的区别在于是否按照数据大小进行扩展,从而扩大寻址范围。

LDUR 指令的格式如下。LDUR 指令的意思是以 Xn|SP 寄存器的内容加一个偏移量(imm)作为内存地址,加载此内存地址的内容(8 字节数据)到 Xt 寄存器。

```
LDUR <Xt>, [<Xn|SP>{, #<imm>}]
```

同理,不可扩展模式的存储指令为 STUR,其指令格式如下。STUR 指令是把 Xt 寄存器的内容存储到 Xn|SP 寄存器加上 imm 偏移量的地方。

```
STUR <Xt>, [<Xn|SP>{, #<imm>}]
```

不可扩展模式的 LDUR 和 STUR 指令根据数据位宽有多种变种,如表 3-4 所示。

表 3-4 不可扩展模式的 LDUR 和 STUR 指令

指 令	描 述
LDUR	数据加载指令
LDURSW	有符号的数据加载指令,单位为字
LDURB	数据加载指令,单位为字节
LDURSB	有符号的数据加载指令,单位为字节
LDURH	数据加载指令,单位为半字

续表

指　令	描　　述
LDURSH	有符号的数据加载指令，单位为半字
STUB	数据存储指令
STURB	数据存储指令，单位为字节
STURH	数据存储指令，单位为半字

10. 独占内存访问指令

ARMv8 体系结构提供独占内存访问(exclusive memory access)的指令。LDXR 指令尝试在内存总线中申请一个独占访问的锁，然后访问一个内存地址。STXR 指令往刚才 LDXR 指令已经申请独占访问的内存地址中写入新内容。LDXR 和 STXR 指令通常组合使用以完成一些同步操作，如 Linux 内核的自旋锁。

另外，ARMv7 和 ARMv8 还提供多字节独占内存访问指令，如表 3-5 所示。

表 3-5　独占内存访问指令

指　令	描　　述
LDXR	独占内存访问指令。指令的格式如下 `LDXR Xt,[Xn|SP{,#0}];`
STXR	独占内存访问指令。指令的格式如下 `STXR Ws,Xt,[Xn|SP{,#0}];`
LDXP	多字节独占内存访问指令。指令的格式如下 `LDXP Xt1,Xt2,[Xn|SP{,#0}];`
STXP	多字节独占内存访问指令。指令的格式如下 `STXP Ws,Xt1,Xt2,[Xn|SP{,#0}];`

11. 隐含加载-获取/存储-释放内存屏障原语

ARMv8 体系结构提供一组新的加载和存储指令，其中包含内存屏障原语，如表 3-6 所示。

表 3-6　隐含屏障原语的加载和存储指令

指 令	描　　述
LDAR	加载-获取(load-acquire)指令。LDAR 指令后面的读写内存指令必须在 LDAR 指令之后执行
STLR	存储-释放(store-release)指令。所有的加载和存储指令必须在 STLR 指令之前完成

12. 非特权访问级别的加载和存储指令

ARMv8 体系结构实现了一组非特权访问级别的加载和存储指令，它适用于在 EL0 进行的访问，如表 3-7 所示。

表 3-7 非特权访问级别的加载和存储指令

指 令	描 述
LDTR	非特权加载指令
LDTRB	非特权加载指令,加载1字节
LDTRSB	非特权加载指令,加载有符号的1字节
LDTRH	非特权加载指令,加载2字节
LDTRSH	非特权加载指令,加载有符号的2字节
LDTRSW	非特权加载指令,加载有符号的4字节
STTR	非特权存储指令,存储8字节
STTRB	非特权存储指令,存储1字节
STTRH	非特权存储指令,存储2字节

当PSTATE寄存器中的UAO字段为1时,在EL1和EL2执行这些非特权指令的效果和执行特权指令是一样的,这个特性是在ARMv8.2的扩展特性中加入的。

3.3.2 算术与移位指令

1. 条件操作码

A64指令集沿用了A32指令集中的条件操作,在PSTATE寄存器中有4个条件标志位,即N、Z、C、V,如表3-8所示。

表 3-8 条件标志位

条件标志位	描 述
N	负数标志(上一次运算结果为负值)
Z	零结果标志(上一次运算结果为零)
C	进位标志(上一次运算结果发生了无符号数溢出)
V	溢出标志(上一次运算结果发生了有符号数溢出)

常见的条件操作后缀如表3-9所示。

表 3-9 常见的条件操作后缀

后 缀	含义(整数运算)	条件标志位	条 件 码
EQ	相等	Z=1	0b0000
NE	不相等	Z=0	0b0001
CS/HS	发生了无符号数溢出	C=1	0b0010
CC/LO	没有发生无符号数溢出	C=0	0b0011

续表

后　缀	含义(整数运算)	条件标志位	条　件　码
MI	负数	N=1	0b0100
PL	正数或零	N=0	0b0101
VS	溢出	V=1	0b0110
VC	未溢出	V=0	0b0111
HI	无符号数大于	(C==1)&&(Z==0)	0b1000
LS	无符号数小于或等于	(C==0)||(Z==1)	0b1001
GE	有符号数大于或等于	N==V	0b1010
LT	有符号数小于	N! =V	0b1011
GT	有符号数大于	(Z==0)&&(N==V)	0b1100
LE	有符号数小于或等于	(Z==1)||(N! =V)	0b1101
AL	永远执行	—	0b1110
NV	永不执行	—	0b1111

2. 加法与减法指令

1) ADD 指令

普通的加法指令有下面几种用法。

- 使用立即数的加法。
- 使用寄存器的加法。
- 使用移位操作的加法。

使用立即数的加法指令格式如下,它的作用是把 Xn|SP 寄存器的值再加上立即数 imm,把结果写入 Xd|SP 寄存器。Shift 表示可选项,默认表示算术左移操作。

```
ADD <Xd|SP>, <Xn|SP>, #<imm>{, <shift>}
```

【例 3-6】 以下代码是使用立即数的加法指令示例。

```
add x0, x1, #1        //把 x1 寄存器的值加上立即数 1,结果写入 x0 寄存器
add x0, x1, #1 LSL 12
                //把立即数 1 算术左移 12 位,然后加上 x1 寄存器的值,结果写入 x0 寄存器
```

使用寄存器的加法指令格式如下。这条指令的作用是先对 Rm 寄存器做一些扩展,例如左移操作,然后加上 Xn|SP 寄存器的值,把结果写入 Xd|SP 寄存器。

```
ADD <Xd|SP>, <Xn|SP>, <Rm>{, <extend>{#<amount>}}
```

【例 3-7】 下面是使用寄存器的加法指令。

```
add x0, x1, x2              //x0=x1+x2
```

```
add x0, x1, x2, LSL 2          //x0=x1+x2<<2
```

【例 3-8】 下面也是使用寄存器的加法指令。

```
mov  x1, #1
mov  x2, #0x108a
add  x0, x1, x2, UXTB
add  x0, x1, x2, SXTB
```

上面的示例代码中，第 3 行的运行结果为 0x8B，因为 UXTB 对 X2 寄存器的低 8 位数据进行了无符号扩展，结果为 0x8A，然后加上 X1 寄存器的值，最终结果为 0x8B。在第 4 行中，SXTB 对 X2 寄存器的低 8 位数据进行有符号扩展，结果为 0xFFFFFFFFFFFFF8A，然后加上 X1 寄存器的值，最终结果为 0xFFFFFFFFFFFFF8B。

使用移位操作的加法指令的格式如下。这条指令的作用是先对 Xm 寄存器做一些移位操作，然后加上 Xn 寄存器的值，结果写入 Xd 寄存器。

```
ADD <Xd>,<Xn>,<Xm>{,<shift>#<amount>}
```

【例 3-9】 以下代码用于实现移位操作加法。

```
add x0, x1, x2, LSL 3          //x0=x1+x2<<3
```

2）ADDS 指令

ADDS 指令是 ADD 指令的变种，唯一的区别是指令执行结果会影响 PSTATE 寄存器的 N、Z、C、V 标志位，例如当计算结果发生无符号数溢出时，C=1。

【例 3-10】 下面的代码使用了 ADDS 指令。

```
mov x1, 0xFFFFFFFFFFFFFFFF

adds x0, x1, #2

mrs x2, nzcv
```

X1 的值(0xFFFFFFFFFFFFFFFF)加上立即数 2 一定会触发无符号数溢出，最终 X0 寄存器的值为 1，同时设置 PSTATE 寄存器的 C 标志位为 1。通过读取 NZCV 寄存器进行判断，最终 X2 寄存器的值为 0x20000000，说明第 29 位的 C 字段置 1，如图 3-7 所示。

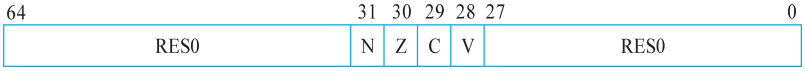

图 3-7 NZCV 寄存器

3）ADC 指令

ADC 是进位的加法指令，最终的计算结果需要考虑 PSTATE 寄存器的 C 标志位。ADC 指令的格式如下。Xd 寄存器的值等于 Xn 寄存器的值再加上 Xm 寄存器的值再加上 C，其中，C 表示 PSTATE 寄存器的 C 标志位。

```
ADC <Xd>, <Xn>, <Xm>
```

【例 3-11】 如下代码使用了 ADC 指令。

```
mov  x1, 0xFFFFFFFFFFFFFFFF
mov  x2, #2

adc  x0, x1, x2

mrs  x3 nzcv
```

ADC 指令的计算过程是0xFFFFFFFFFFFFFFFF+2+C,因为 0xFFFFFFFFFFFFFFFF+2 的过程中已经触发了无符号数溢出,C=1,所以最终计算 X0 寄存器的值为 2。若读取 NZCV 寄存器,会发现 C 标志位也被置位了。

4) SUB 指令

普通的减法指令与加法指令类似,也有下面几种用法。

- 使用立即数的减法。
- 使用寄存器的减法。
- 使用移位操作的减法。

使用立即数的减法指令格式如下,它的作用是把 Xn|SP 寄存器的值减去立即数 imm,结果写入 Xd|SP 寄存器。

```
SUB <Xd|SP>, <Xn|SP>, #<imm>{, <shift>}
```

【例 3-12】 如下代码使用了 SUB 指令。

```
Sub x0, x1, #1        //把 x1 寄存器的值减去立即数 1,结果写入 x0 寄存器

sub x0, x1, #1, LSL 12
       //把立即数 1 算术左移 12 位,然后把 x1 寄存器中的值减去立即数(1<<12),把结果值
                      //写入 x0 寄存器
```

使用寄存器的减法指令格式如下。这条指令的作用是先对 Rm 寄存器做一些扩展,例如左移操作,然后 Xn|SP 寄存器的值减去 Rm 寄存器的值,把结果写入 Xd|SP 寄存器。

```
SUB <Xd|SP>,<Xn|SP>,<Rm>{,<extend>{#<amount>}}
```

【例 3-13】 如下代码使用了寄存器的减法指令。

```
sub x0, x1, x2             //x0=x1-x2
sub x0, x1, x2, LSL 2      //x0=x1-x2<<2
```

【例 3-14】 下面的代码也使用了寄存器的减法指令。

```
mov  x1, #1
mov  x2, #0x108a
sub  x0, x1, x2, UXTB
sub  x0, x1, x2, SXTB
```

上面的示例代码中,UXTB 对 X2 寄存器的低 8 位数据进行了无符号扩展,结果为 0x8A,然后计算 1-0x8A 的值,最终结果为 0xFFFFFFFFFFFFFF77。