ARMv8/ARMv9指令集概述

作者：代码改变世界*ctw*

Release history

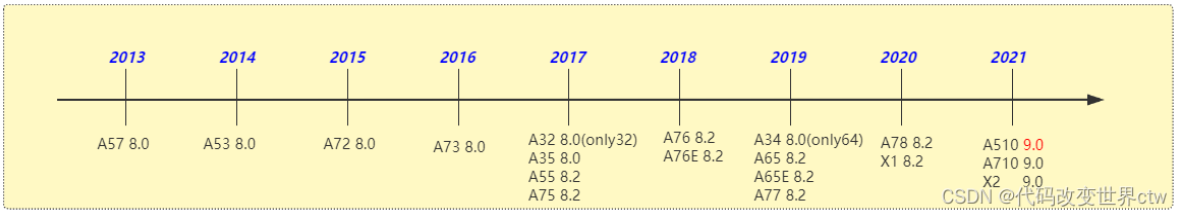
Data	Author	Change
2022/03/28	代码改变世界ctw	Beta draft v1.0

1 前言

为什么要写这篇文档

ARMV8都出来10年了，可是一本中文的手册都没用。真的很好奇，为什么没有人翻译这类文档呢。 (不过最近好像有相关中文的文档了).

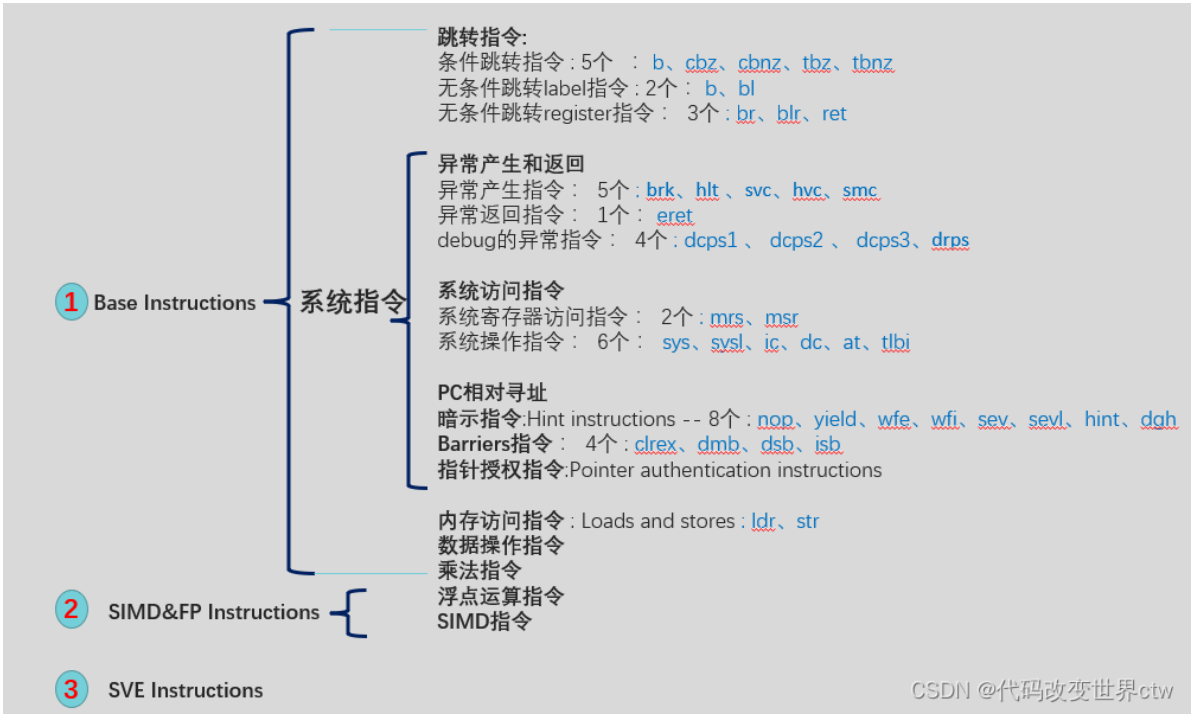
本文仅仅是翻译ARM的一篇官方的指令集文档，仅仅是指令集文档哦，不会介绍架构等知识。说实话，你不理解的是汇编吗？你是看不懂指令吗？ 不你的瓶颈是硬件架构知识吧



(本文大多数都是直译吧，翻译的也不是太好，浪费时间。另外遗留了一些章节没有翻译，懒得弄了。如果你有兴趣可联系我，我给你markdown原文，一起补充下剩余的章节吧)

我们要学习什么?

这些指令，咋一看，真简单，也不多吗，大几十个？ 但是算上变体等，那就有数百个甚至好几千个了。很多指令有和具体的feature和架构相关，很难去记住。 所以呢，我们可以学习一些基本的指令集，其余的使用的时候再查阅即可。如下列举了指令的分类，我们只要对着这个分类，学习一些基础的指令即可。



推荐序

TODO

（联系方式）

2 简介

本文档概述了 ARMv8 指令集。大部分文档描述了处理器在 AArch64 寄存器宽度状态下运行时使用的新 A64 指令集，并定义了其首选的
架构汇编语言。

下面的第 6 节列出了 ARMv8 对 A32 和 T32 指令集（在 ARMv7 中分别称为 ARM 和 Thumb 指令集）引入的扩展，这些扩展在处理器在
AArch32 寄存器宽度状态下运行时可用。 A32 和 T32 汇编语言语法与 ARMv7 相同。

在下面的语法描述中，使用了以下约定：

- `UPPER` UPPER-CASE文本是固定的，而小写文本是可变的。所以寄存器名 Xn 表示'X'是必需的，后跟可变寄存器编号，例如 X29。
- `<>` 由 `< >` 括起来的任何项目都是对用户在该位置提供的值类型的简短描述。项目的较长描述通常由后续文本提供
- `{ }` 任何用花括号 `{ }` 括起来的项目都是可选的。项目的描述以及它的存在或不存在如何影响指令通常由后续文本提供。在某些情

况下，花括号是语法中的实际符号，例如围绕一个寄存器列表，并且这种情况将在周围的文本中被调用
- `[ ]` 替代字符列表可以用 `[ ]` 括起来。可以在该位置使用单个字符，随后的文本将描述替代的含义。在某些情况下，符号 [和] 是
  语法本身的一部分，例如寻址模式和向量元素，这些情况将在周围的文本中被调用
- `a | b` 替代词由竖线分隔 |并且可以用括号括起来以分隔它们，例如 U(ADD|SUB)W 表示 UADDW 或 USUBW。
- `+/-` 这表示可选的 + 或 - 符号。如果两者都没有编码，则假定为 +

3 A64概述

A64 指令集提供与 AArch32 或 ARMv7 中的 A32 和 T32 指令集类似的功能。然而，正如在 T32 指令集中添加 32 位指令使一些 ARM ISA 行为合理化一样，A64 指令集包括进一步的合理化。新指令集的亮点如下:

- 清晰、固定长度的指令集 指令为 32 位宽
- 访问一个更大的通用寄存器文件，其中包含 31 个未分组的寄存器 (0-30)，每个寄存器扩展为 64 位。通用寄存器被编码为 5 位字段，其中寄存器编号 31 (0b11111) 是一种特殊情况：
  (1) 表示零寄存器XZR：在大多数情况下，当用作源寄存器时，寄存器编号 31 读取为零，并且用作目标寄存器时丢弃结果
  (2) 表示栈寄存器SP：当用作加载/存储基址寄存器时，以及在少量算术指令中，31 号寄存器提供对当前堆栈指针的访问
- PC 永远不能作为命名寄存器访问。它的使用隐含在某些指令中，例如 PC相对加载和地址生成。唯一会导致 PC 发生非顺序更改的指令是指定的控制流指令（参见第 5.1 节）和异常。不能将 PC 指定为数据处理指令或加载指令的目标。
- 链接寄存器(LR) 是unbanked的通用寄存器X30, 异常链接寄存器ELR是系统寄存器
- 标量加载/存储寻址模式在标量整数、浮点和向量寄存器的所有大小和符号上都是统一的
- 加载/存储立即偏移量可以根据访问大小缩放，增加其有效偏移量范围。
- 反映加载/存储寻址模式的地址生成算法指令，见3.3
- PC相对的加载/存储和生成范围为±4GiB的地址只需要两条指令就可以实现，而不需要从文字池中加载偏移量。
- 对于文字池访问和大多数条件分支，pc相对偏移量被扩展为±1MiB，对于无条件分支和调用，pc相对偏移量被扩展为±128MiB
- 没有多寄存器 LDM、STM、PUSH 和 POP 指令，但可以加载存储不连续的一对寄存器。
- 大多数加载和存储都允许未对齐的地址，包括配对寄存器访问、浮点和SIMD 寄存器，但排他和有序访问除外（请参阅第3.5.2 节）。
- 减少条件。 更少的指令可以设置条件标志。 只有条件分支和少数数据处理指令读取条件标志。 没有提供条件或谓词执行，也没有等效于 T32 的 IT 指令（参见第 3.2 节）。
- 数据处理指令的最后一个寄存器操作数的移位选项可用：
  (1) 仅立即换班（如 T32 中）。
  (2) ADD/SUB 没有 RRX 移位，也没有 ROR 移位。
  (3) ADD/SUB/CMP 指令可以先对最后一个寄存器操作数中的一个字节、半字或字进行符号或零扩展，然后是可选的 1 到 4 位左移
- 立即生成将 A32 的旋转 8 位立即数替换为特定于操作的编码：
  (1) 算术指令有一个简单的 12 位立即数，可选择左移 12。
  (2) 逻辑指令提供复杂的复制位掩码生成。
  (3) 其他立即数可以在 16 位“块”中内联构造，扩展 AArch32 的 MOVW 和 MOVT 指令。
- 浮点支持类似于 AArch32 VFP，但有一些扩展，如 §3.6 中所述。
- 浮点和高级 SIMD 处理共享一个寄存器文件，以类似于 AArch32 的方式，但扩展到 32 个 128 位寄存器。 较小的寄存器不再打包到较大的寄存器中，而是一对一映射到 128 位寄存器的低位，如 4.4.2 中所述。

- 没有对通用寄存器进行操作的 SIMD 或饱和算术指令，此类操作仅作为高级 SIMD 处理的一部分可用，如第 5.7 节所述。
- 无法将 CPSR 作为单个寄存器访问，但新的系统指令提供了以原子方式修改各个处理器状态字段的能力，请参阅第 5.8.2 节。
- 从架构中删除了“协处理器”的概念。 5.8 中描述的一组系统指令提供：
  (1)System register access
  (2)Cache/TLB management
  (3)VA<--->PA address translation
  (4)Barriers and CLREX
  (5)Architectural hints (WFI, etc)
  (6)Debug

3.1 32位和64位之类的区别

A64 指令集中的大多数整数指令有两种形式，它们对 64 位通用寄存器文件中的 32 位或 64 位值进行操作。在选择 32 位指令形式的情况下，以下情况成立：

- 源寄存器的高 32 位被忽略；
- 目标寄存器的高 32 位设置为零；
- 在第 31 位而不是第 63 位右移/旋转注入；
- 由指令设置的条件标志是从低 32 位计算的。

即使当 32 位指令形式的结果与由等效 64 位指令形式计算的低 32 位无法区分时，这种区别也适用。例如，可以使用 64 位 ORR 执行 32 位按位 ORR，并简单地忽略结果的前 32 位。但是 A64 指令集包括单独的 32 位和 64 位形式的 ORR 指令。

基本原理：C/C++ LP64 和 LLP64 数据模型——预计将是 AArch64 上最常用的——都将常用的 int、short 和 char 类型定义为 32 位或更少。通过在指令集中维护此语义信息，实现可以利用此信息来避免消耗能量或周期来计算、转发和存储此类数据类型的未使用的高 32 位。实现可以自由地以他们选择的任何方式来利用这种自由来节省能源

除了不同的符号/零扩展指令外，A64 指令集还提供扩展和移位 ADD、SUB 或 CMP 指令的最终源寄存器以及加载/存储指令的索引寄存器的能力。这允许有效实现涉及 64 位数组指针和 32 位数组索引的数组索引计算。

汇编语言符号旨在允许将保存 32 位值的寄存器与保存 64 位值的寄存器区分开来。除了提高可读性外，工具还可以使用它来执行有限的类型检查，以识别因寄存器大小变化而导致的编程错误。

3.2 条件指令

A64 指令集不包括谓词或条件执行的概念。基准测试表明，现代分支预测器工作得很好，以至于指令的预测执行并没有提供足够的好处来证明它对操作码空间的大量使用以及它在高级实现中的实现成本。

提供了一组非常小的“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。该集合已被证明在条件分支预测不佳或效率低下的情况下是有益的。

条件指令类型有：

- 条件分支：传统的ARM 条件分支，连同比较和寄存器零/非零分支，以及测试寄存器中的单个位和零/非零分支——所有这些都增加了位移。
- 进位加/减：传统的 ARM 指令，用于多精度算术、校验和等。
- 带递增、取反或取反的条件选择：有条件地在一个源寄存器和第二个递增/取反/取反/未修改的源寄存器之间进行选择。基准测试显示这些是单个条件指令的最高频率使用，例如用于计数、绝对值等。这些指令还实现：
  (1) 条件选择（移动）：将目标设置为两个源寄存器之一，由条件标志选择。短条件序列可以用无条件指令替换，然后是条件选择。

(2) 条件集：有条件地在 0 和 1 或 -1 之间选择，例如将条件标志物化为通用寄存器中的布尔值或掩码。

- 条件比较：如果原始条件为真，则将条件标志设置为比较结果，否则设置为立即值。允许在不使用条件分支或在通用寄存器中执行布尔运算的情况下扁平化嵌套条件表达式。

3.3 寻址功能

64 位架构的主要动机是访问更大的虚拟地址空间。 AArch64 内存转换系统支持 49 位虚拟地址（每个转换表 48 位）。 虚拟地址从 49 位符号扩展，并存储在 64 位指针中。 可选地，在系统寄存器的控制下，64 位指针的最高有效 8 位可以保存一个“标记”，当用作加载/存储地址或间接分支的目标时，该标记将被忽略。

3.3.1 寄存器变址寻址

A64 指令集扩展了 32 位 T32 寻址模式，允许将 64 位索引寄存器添加到 64 位基址寄存器，并可以根据访问大小对索引进行可选缩放。此外，它还提供了索引寄存器中 32 位值的符号或零扩展，同样具有可选的缩放比例。

如果可以在单个周期内执行这些寄存器索引寻址模式，它们将提供有用的性能增益，并且相信至少一些实现将能够做到这一点。但是，根据 AArch32 的实现经验，预计其他实现将需要一个额外的周期来执行此类寻址模式。

基本原理：架构师希望实现可以自由地微调每个实现中的性能权衡，并注意提供在某些实现中需要两个周期的指令比要求在一个可以在单个周期内执行此地址算术的实现。

3.3.2 PC 相对寻址

改进了对位置无关代码和数据寻址的支持：

- PC 相关文字负载的偏移范围为 ±1MiB。 这允许更少的文字池，并在函数之间更多地共享文字数据——减少 I-cache 和 TLB 污染。
- 大多数条件分支的范围为 ±1MiB，预计足以满足在单个函数中发生的大多数条件分支。
- 无条件分支，包括分支和链接，范围为 ±128MiB。 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蹦床或“胶合板”。
- PC 相关的加载/存储和范围为 ±4GiB 的地址生成可以仅使用两条指令内联执行，即无需从文字池加载偏移量。

3.4 程序计数器 (PC)

当前的程序计数器 (PC) 不能像通用寄存器文件的一部分一样通过数字来引用，因此不能用作算术指令的源或目标，也不能用作加载/存储指令的基址、索引或传输寄存器。 读取 PC 的唯一指令是那些功能是计算 PC 相对地址（ADR、ADRP、文字加载和直接分支）的指令，以及将其存储在链接寄存器中的分支和链接指令（BL 和 BLR)。 修改程序计数器的唯一方法是使用显式控制流指令：条件分支、无条件分支、异常生成和异常返回指令。

如果指令读取 PC 以计算 PC 相对地址，那么它的值就是指令的地址，即与 A32 和 T32 不同，没有隐含的 4 或 8 个字节的偏移量。

3.5 内存加载-存储

3.5.1 批量传输

A64 中不存在 LDM、STM、PUSH 和 POP 指令，但是可以使用 LDP 和 STP 指令构建批量传输，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，并且在访问普通内存时支持未对齐的地址 . LDNP 和 STNP 指令还提供“流”或“非临时”提示，表明数据不需要保留在缓存中。 PRFM（预取存储器）指令还包括“流式”或“非临时”访问的提示，并允许将预取定位到特定的缓存级别。

3.5.2 独占访问

字节、半字、字和双字的独占加载存储。 对一对双字的独占访问允许对一对指针进行原子更新，例如循环列表插入。 所有独占访问必须自然对齐，并且独占对访问必须对齐到两倍的数据大小（即 64 位对的 16 个字节）。

3.5.3 Load-Acquire, Store-Release

显式同步加载和存储指令实现了释放一致性 (RCsc) 内存模型，减少了对显式内存屏障的需求，并为共享内存的新兴语言标准提供了良好的匹配。 这些指令以排他和非排他的形式存在，并且需要自然地址对齐。 有关详细信息，请参阅第 5.2.8 节。

3.6 整数乘法/除法

3.7 浮点数

AArch64 在任何需要浮点运算的地方都强制使用硬件浮点——AArch64 过程调用标准 (PCS) 没有“软浮点”变体。

浮点功能类似于 AArch32 VFP，但有以下变化：

- 删除了VFP 已弃用的“小向量”功能。
- 有32 个S 寄存器和32 个D 寄存器。 S 寄存器不打包到 D 寄存器中，而是占用相应 D 寄存器的低 32 位。例如 S31=D31[31:0]，而不是 D15[63:32]。
- 加载/存储寻址模式与整数加载/存储相同。
- 加载/存储一对浮点寄存器。
- 浮点FCSEL 和FCCMP 等效于整数CSEL 和CCMP。
- 浮点FCMP 和FCCMP 指令直接设置整数条件标志，不修改FPSR 中的条件标志。
- 所有浮点乘加和乘减指令都是“融合”的。
- 在 64 位整数和浮点之间转换。
- 将FP 转换为具有明确舍入方向的整数（朝向零、朝向+Inf、朝向-Inf、到最接近的关系到偶数，以及最接近的关系以及远离零的关系）。
- 使用明确的舍入方向（如上）将 FP 舍入到最接近的整数 FP。
- 半精度和双精度之间的直接转换。
- FMINNM 和FMAXNM 实现IEEE754-2008 minNum() 和maxNum() 操作，如果其中一个操作数是安静的NaN，则返回数值。

3.8 高级 SIMD

详见下面5.7的详细描述

4 A64汇编语言

字母 W 是 32 位字的简写，X 是 64 位扩展字的简写。使用字母 X（扩展）而不是 D（双精度），因为 D 与其用于浮点和 SIMD“双精度”寄存器以及 T32 加载/存储“双寄存器”指令（例如 LDRD）的使用相冲突。

A64 汇编器将识别指令助记符和寄存器名称的大写和小写变体，但不能识别大小写混合。 A64 反汇编程序可以输出大写或小写的助记符和寄存器名称。程序和数据标签的情况很重要。

基本语句格式和操作数顺序遵循 AArch32 UAL 汇编器和反汇编器使用的，即每个源代码行一个语句，由一个或多个可选程序标签组成，后跟指令助记符，然后是目标寄存器和一个或多个源操作数被逗号隔开。

`{label:*} {opcode {dest{, source1{, source2{, source3}}}}}`

与 AArch32 UAL 一样，存储指令的 dest/source 顺序是相反的。 A64 汇编语言不需要“#”符号来引入立即值，尽管汇编器必须允许它。为了便于阅读，A64 反汇编程序应始终在立即值之前输出“#”。

如果用户定义的符号或标签与预定义的寄存器名称（例如“X0”）相同，那么如果在其解释不明确的上下文中使用它 - 例如在可以接受寄存器名称的操作数位置或立即表达式——然后汇编器必须将其解释为寄存器名称。可以通过在表达式上下文中使用符号来消除歧义，即将其放在括号内和/或在其前面加上显式的“#”符号。

在下面的示例中，序列“//”用作注释前导符，但 A64 汇编器也应支持其旧版 ARM 注释语法

4.1 基本结构

A64指令形式可以通过以下属性组合来识别：

- 指示指令语义的`operation name`（例如 ADD）。
- `operand container`，通常是寄存器类型。 一条指令写入整个*container*，但如果它不是同类中最大的，则该类中最大*container*的其余部分设置为零。
- `operand data subtype`，其中一些操作数与主*container*的大小不同。
- `final source operand type`，可以是寄存器或立即数。

`operand container`是以下之一：

Integer Class	
W	32-bit integer
X	64-bit integer
SIMD Scalar & Floating Point Class	
B	8-bit scalar
H	16-bit scalar & half-precision float
S	32-bit scalar & single-precision float
D	64-bit scalar & double-precision float
Q	128-bit scalar

`operand data subtype`是以下之一：

Load-Store / Sign-Zero Extend	
B	byte
SB	signed byte
H	halfword
SH	signed halfword
W	word
SW	signed word
Register Width Changes	
H	High (dst gets top half)
N	Narrow (dst < src)
L	Long (dst > src)
W	Wide (dst == src1, src1 > src2)
etc	

4.2 指令助记符

这些属性以汇编语言符号组合在一起，以标识特定的指令形式。为了保持与现有 ARM 汇编语言的紧密外观，采用了以下格式：

`<name>{<subtype>} <container>`

换句话说，操作名称和子类型由指令助记符描述，容器大小由操作数名称描述。省略子类型的地方，它是从容器继承的。

通过这种方式，汇编程序员可以编写指令而无需记住大量新的助记符；反汇编清单的读者可以直接阅读一条指令，一眼就能看出每个操作数的类型和大小。

这意味着A64汇编语言重载了指令助记符，并根据操作数寄存器名称来区分指令的不同形式。例如，下面的 ADD 指令都有不同的操作码，但程序员只需记住一个助记符，汇编器会根据操作数自动选择正确的操作码——反汇编器则相反。

4.3 条件代码

在 AArch32 汇编语言中，条件执行指令通过直接将条件附加到助记符来表示，没有分隔符。这会导致一些歧义，从而使汇编代码难以解析：例如，ADCS、BICS、LSLS 和 TEQ 乍一看就像条件指令。

A64 ISA 设置或测试条件代码的指令要少得多。那些这样做的人将被识别如下：

1. 设置条件标志的指令在概念上是不同的指令，并且将通过在基本助记符后附加“S”来继续识别，例如添加。
2. 真正有条件执行的指令（即，当条件为假时，它们对体系结构状态没有影响，除了推进程序计数器）将条件附加到带有“。”的指令中。分隔符。例如 B.EQ。
3. 如果有多个指令扩展，则条件扩展总是最后一个。
4. 如果条件指令具有限定符，则限定符遵循条件。
5. 无条件执行但使用条件标志作为源操作数的指令，将在其最终操作数位置指定要测试的条件，例如 CSEL Wd,Wm,Wn,NE

为了提高可移植性，A64 汇编器还可以提供旧的 UAL 条件助记符，只要它们在 A64 ISA 中具有直接等效项。但是，UAL 助记符不会由 A64 反汇编程序生成——在 64 位汇编代码中不推荐使用它们，如果程序员没有明确要求向后兼容，可能会导致警告或错误。

条件代码的完整列表如下：

Encoding	Name (& alias)	Meaning (integer)	Meaning (floating point)	Flags
0000	EQ	Equal	Equal	Z==1
0001	NE	Not equal	Not equal, or unordered	Z==0
0010	HS (CS)	Unsigned higher or same (Carry set)	Greater than, equal, or unordered	C==1
0011	LO (CC)	Unsigned lower (Carry clear)	Less than	C==0
0100	MI	Minus (negative)	Less than	N==1
0101	PL	Plus (positive or zero)	Greater than, equal, or unordered	N==0
0110	VS	Overflow set	Unordered	V==1
0111	VC	Overflow clear	Ordered	V==0
1000	HI	Unsigned higher	Greater than, or unordered	C==1 && Z==0
1001	LS	Unsigned lower or same	Less than or equal	!(C==1 && Z==0)
1010	GE	Signed greater than or equal	Greater than or equal	N==V
1011	LT	Signed less than	Less than or unordered	N!=V
1100	GT	Signed greater than	Greater than	Z==0 && N==V
1101	LE	Signed less than or equal	Less than, equal, or unordered	!(Z==0 && N==V)
1110	AL	Always	Always	Any
1111	NV†			

†条件代码 NV 的存在仅用于提供对“1111b”编码的有效反汇编，否则其行为与 AL 相同。

4.4 寄存器名称

4.4.1 通用寄存器

general purpose registers组中的 31 个通用寄存器命名为 R0 到 R30，特殊寄存器编号 31 具有不同的名称，具体取决于使用它的上下文。但是，当寄存器以特定指令形式使用时，它们必须进一步限定以指示操作数数据大小（32 位或 64 位），从而指示指令的数据大小。

通用寄存器的限定名称如下，其中“n”是寄存器编号 0 到 30：

Size (bits)	32b	64b
Name	Wn	Xn

其中寄存器编号 31 表示读取零或丢弃结果（又名“零寄存器”）：

Size (bits)	32b	64b
Name	WZR	XZR

其中寄存器号 31 表示堆栈指针：

Size (bits)	32b	64b
Name	WSP	SP

更详细地说：

- 名称Xn 和Wn 指的是相同的架构寄存器。
- 没有名为W31 或X31 的寄存器。
- 对于寄存器31 被解释为64 位堆栈指针的指令操作数，它由名称SP 表示。对于不将寄存器 31 解释为 64 位堆栈指针的操作数，此名称将导致汇编错误。
- WSP 名称将寄存器 31 表示为 32 位上下文中的堆栈指针。提供它只是为了允许有效的反汇编，并且不应在行为正确的 64 位代码中看到。
- 对于将寄存器 31 解释为零寄存器的指令操作数，它在 64 位上下文中由名称 XZR 表示，在 32 位上下文中由名称 WZR 表示。在不将寄存器 31 解释为零寄存器的操作数位置中，这些名称将导致汇编错误。
- 如果助记符过载（即可以根据数据大小生成不同的指令编码），则汇编程序应根据第一个寄存器操作数的大小确定指令的精确形式。通常其他操作数寄存器应该与第一个操作数的大小相匹配，但在某些情况下，一个寄存器可能有不同的大小（例如，地址基址寄存器总是 64 位），如果源寄存器包含被指令扩展为 64 位的字、半字或字节。
- 该体系结构没有为寄存器 30 定义一个特殊名称，以反映其作为过程调用中的链接寄存器的特殊作用。此类软件名称可以定义为过程调用标准的一部分。

4.4.2 FP/SIMD 寄存器

FP/SIMD 寄存器组中名为 V0 到 V31 的 32 个寄存器用于保存标量浮点指令的浮点操作数，以及高级 SIMD 指令的标量和向量操作数。 与通用整数寄存器一样，当它们以特定指令形式使用时，必须进一步限定名称以指示其中保存的数据形状（即数据元素大小和元素或通道数）。

但是请注意，数据类型，即每个寄存器或向量元素中的位解释——整数（有符号、无符号或不相关）、浮点、多项式或加密哈希——不是由寄存器名称描述的，而是由指令助记符描述的 对它们进行操作。 有关更多详细信息，请参阅第 5.7 节中的高级 SIMD 描述。

4.5 加载/存储寻址模式

A64 指令集中的加载/存储寻址模式大致遵循 T32，包括一个 64 位基址寄存器（Xn 或 SP）加上一个立即数或寄存器偏移量。

Type	Immediate Offset	Register Offset	Extended Register Offset
Simple register (exclusive)	`[base{,#0}]`	n/a	n/a
Offset	`[base{,#imm}]`	`[base,Xm{,LSL #imm}]`	`[base,Wm,(S|U)XTW {#imm}]`
Pre-indexed	`[base,#imm]!`	n/a	n/a
Post-indexed	`[base],#imm`	n/a	n/a
PC-relative (literal) load	`label`	n/a	n/a

- 立即偏移量以各种方式编码，具体取决于加载/存储指令的类型：

Bits	Sign	Scaling	Write-back?	Load/Store Type
0	-	-	-	exclusive / acquire / release
7	signed	scaled	option	register pair
9	signed	unscaled	option	single register
12	unsigned	scaled	no	single register

- 在立即偏移量被缩放的地方，它被编码为数据访问大小的倍数（PCrelative 加载除外，它总是一个字倍数）。汇编器总是接受一个字节偏移量，它被转换成缩放的偏移量进行编码，反汇编器解码缩放的偏移量编码并将其显示为字节偏移量。因此，支持的字节偏移范围根据加载/存储指令的类型和数据访问大小而有所不同。
- “后索引”形式意味着内存地址是基址寄存器的值，然后基址加偏移量被写回基址寄存器。
- “预索引”形式意味着内存地址是基址寄存器值加上偏移量，然后计算的地址被写回基址寄存器。
- “寄存器偏移”意味着内存地址是基址寄存器值加上 64 位变址寄存器 Xm 的值，可以选择按访问大小（以字节为单位）缩放，即左移 log2(size)。
- “扩展寄存器偏移”意味着存储器地址是基址寄存器值加上 32 位变址寄存器 Wm 的值，符号或零扩展为 64 位，然后可以根据访问大小进行缩放。
- 汇编器应接受 Xm 作为扩展变址寄存器，但首选 Wm。
- 前/后索引表格不适用于寄存器偏移。
- 没有“向下”选项，因此从基址寄存器中减法需要负符号立即偏移（二进制补码）或索引寄存器中的负值。
- 当基址寄存器为 SP 时，堆栈指针需要在地址计算和回写之前进行四字（16 字节，128 位）对齐 - 未对齐将导致堆栈对齐错误。堆栈指针不能用作索引寄存器。
- 使用程序计数器 (PC) 作为基址寄存器隐含在文字加载指令中，并且不允许在其他加载或存储指令中使用。文字加载不包括字节和半字形式。标签的定义见下文第 5 节。

5 A64指令集

5.1 控制流程

5.1.1 条件分支

- B.cond label 跳转指令：如果cond 为真，则有条件地跳转到程序相关标签。
- CBNZ Wn, label 比较和非零分支：如果 Wn 不等于零，则有条件地跳转到程序相关标签。
- CBNZ Xn, label 比较和非零分支（扩展）：如果 Xn 不等于零，有条件地跳转到标签。
- CBZ Wn, label 比较和分支零：如果 Wn 等于零，有条件地跳转到标签。
- CBZ Xn, label 比较和分支零（扩展）：如果 Xn 等于零，有条件地跳转到标签。
- TBNZ Xn|Wn, #uimm6, label 测试和非零分支：如果寄存器 Xn 中的位号 uimm6 不为零，则有条件地跳转到标签。位号表示寄存器的宽度，如果 uimm 小于 32，则可以写入并反汇编

为 Wn。限制在 ±32KiB 的分支偏移范围内。

- TBZ Xn|Wn, #uimm6, label 测试和分支零：如果寄存器 Xn 中的位号 uimm6 为零，则有条件地跳转到标签。位号表示寄存器的宽度，如果 uimm6 小于 32，则可以写入并反汇编为 Wn。限制在 ±32KiB 的分支偏移范围内。

5.1.2 无条件分支（立即数）

无条件跳转支持 ±128MiB 的立即分支偏移范围。

- B label 跳转：无条件跳转到pc-relative label。
- BL label 链接跳转：无条件跳转到pc相对标签，将下一条顺序指令的地址写入X30寄存器。

5.1.3 无条件分支（寄存器）

- BLR Xm 跳转链接寄存器：无条件跳转到Xm中的地址，将下一条连续指令的地址写入X30寄存器。
- BR Xm 分支寄存器：跳转到Xm 中的地址，提示CPU 这不是子程序返回。
- RET {Xm} 返回：跳转到寄存器Xm，提示CPU这是一个子程序返回。 如果省略 Xm，汇编器将默认注册 X30。

5.2 内存访问

除了排他和显式排序的加载和存储之外，地址可能具有任意对齐，除非启用了严格的对齐检查 (SCTLR.A==1)。 但是，如果 SP 用作基址寄存器，则在添加任何偏移之前的堆栈指针的值必须是四字（16 字节）对齐，否则将产生堆栈对齐异常。

由与传输大小对齐的单个通用寄存器的加载或存储生成的内存读取或写入是原子的。 由一对与寄存器大小对齐的通用寄存器的非独占加载或存储产生的内存读取或写入被视为两个原子访问，每个寄存器一个。 在所有其他情况下，除非另有说明，否则没有原子性保证。

5.2.1 加载-存储单个寄存器

最通用的加载存储形式支持多种寻址模式，包括基址寄存器 Xn 或 SP，以及以下之一：

- 缩放的 12 位无符号立即偏移量，没有索引前和索引后选项。
- 未缩放的 9 位有符号立即偏移量，带有索引前或索引后写回。
- 缩放或未缩放的 64 位寄存器偏移量。
- 缩放或未缩放的 32 位扩展寄存器偏移量。

如果 Load 指令指定写回，并且正在加载的寄存器也是基址寄存器，则可能发生以下行为之一：

- 指令未分配
- 该指令被视为 NOP
- 指令使用指定的寻址模式执行加载，基址寄存器变为 UNKNOWN。此外，如果在此类指令期间发生异常，则基地址可能会被破坏，从而无法重复该指令。

如果 Store 指令执行回写并且正在存储的寄存器也是基址寄存器，则可能会发生以下行为之一：

- 指令未分配
- 该指令被视为 NOP
- 指令执行使用指定寻址模式指定的寄存器的存储，但存储的值是 UNKNOWN

- LDR Wt, addr
  加载寄存器：从 addr 寻址的内存中加载一个字到 Wt。

- LDR Xt, addr
  加载寄存器（扩展）：从 addr 寻址的内存中加载一个双字到 Xt。
- LDRB Wt, addr
  加载字节：从 addr 寻址的内存中加载一个字节，然后将其零扩展至 Wt。
- LDRSB Wt, addr
  加载有符号字节：从 addr 寻址的内存中加载一个字节，然后将其符号扩展到 Wt 中。
- LDRSB Xt, addr
  加载有符号字节（扩展）：从 addr 寻址的内存中加载一个字节，然后将其符号扩展到 Xt 中。
- LDRH Wt, addr
  加载半字：从 addr 寻址的内存中加载半字，然后将其零扩展为 Wt。
- LDRSH Wt, addr
  Load Signed Halfword：从 addr 寻址的内存中加载一个半字，然后将其符号扩展为 Wt。
- LDRSH Xt, addr
  加载有符号半字（扩展）：从 addr 寻址的内存中加载一个半字，然后将其符号扩展到 Xt 中。
- LDRSW Xt, addr
  加载带符号的字（扩展）：从 addr 寻址的内存中加载一个字，然后将其符号扩展到 Xt 中。
- STR Wt, addr
  存储寄存器：将字从 Wt 存储到由 addr 寻址的存储器。
- STR Xt, addr
  存储寄存器（扩展）：将双字从 Xt 存储到由 addr 寻址的内存。
- STRB Wt, addr
  存储字节：将来自 Wt 的字节存储到由 addr 寻址的内存中。
- STRH Wt, addr
  存储半字：将半字从 Wt 存储到由 addr 寻址的内存中。

5.2.2 加载-存储单个寄存器（未缩放的偏移量）

加载-存储单寄存器（未缩放偏移）指令支持基址寄存器 Xn 或 SP 的寻址模式，此外：

- 未缩放、9 位、有符号立即偏移量，没有索引前和索引后选项

这些指令使用独特的助记符将它们与正常的加载存储指令区分开来，因为当偏移为正且自然对齐时，功能与缩放的 12 位无符号立即偏移寻址模式重叠。

当立即偏移量明确时，即当它为负数或未对齐时，对程序员友好的汇编程序可以生成这些指令以响应标准 LDR/STR 助记符。 类似地，当编码的立即数为负数或未对齐时，反汇编程序可以使用标准 LDR/STR 助记符显示这些指令。 然而，架构汇编语言不需要这种行为。

- LDUR Wt, [base,#simm9]
  加载（未缩放）寄存器：将一个字从由 base+simm9 寻址的内存加载到 Wt。
- LDUR Xt, [base,#simm9]
  加载（未缩放）寄存器（扩展）：将一个双字从由 base+simm9 寻址的内存加载到 Xt。
- LDURB Wt, [base,#simm9]
  加载（未缩放）字节：从内存中加载一个字节，地址为 base+simm9，然后将其零扩展为 Wt。
- LDURSB Wt, [base,#simm9]
  加载（未缩放）有符号字节：从由 base+simm9 寻址的内存中加载一个字节，然后将其符号扩展为 Wt。

- LDURSB Xt, [base,#simm9]
  Load (Unscaled) Signed Byte (extended)：从base+simm9寻址的内存中加载一个字节，然后将其符号扩展到Xt中。
- LDURH Wt, [base,#simm9]
  加载（未缩放）半字：从 base+simm9 寻址的内存中加载半字，然后将其零扩展为 Wt。
- LDURSH Wt, [base,#simm9]
  Load (Unscaled) Signed Halfword：从 base+simm9 寻址的内存中加载一个半字，然后将其符号扩展为 Wt。
- LDURSH Xt, [base,#simm9]
  加载（未缩放）有符号半字（扩展）：从 base+simm9 寻址的内存中加载一个半字，然后将其符号扩展为 Xt。
- LDURSW Xt, [base,#simm9]
  加载（未缩放）有符号字（扩展）：从内存中加载一个由 base+simm9 寻址的字，然后将其符号扩展为 Xt。
- STUR Wt, [base,#simm9]
  存储（未缩放）寄存器：将字从 Wt 存储到由 base+simm9 寻址的存储器。
- STUR Xt, [base,#simm9]
  存储（未缩放）寄存器（扩展）：将双字从 Xt 存储到由 base+simm9 寻址的内存。
- STURB Wt, [base,#simm9]
  存储（未缩放）字节：将字节从 Wt 存储到由 base+simm9 寻址的内存。
- STURH Wt, [base,#simm9]
  存储（未缩放）半字：将半字从 Wt 存储到由 base+simm9 寻址的内存。

5.2.3 加载单个寄存器（pc-relative，literal load)

用于加载的 pc 相对地址被编码为 19 位有符号字偏移量，该偏移量左移 2 并添加到程序计数器，从而可以访问 PC 的 ±1MiB 内的任何字对齐位置。

为方便起见，汇编程序通常允许符号“=value”与相对 pc 的文字加载指令一起自动将立即值或符号地址放置在附近的文字池中，并生成引用它的隐藏标签。但是该语法不是架构的，并且永远不会出现在反汇编中。 A64 有其他指令可以在寄存器中构造立即值（第 5.3.3 节）和地址（第 5.3.4 节），这可能比从文字池中加载它们更好。

- LDR Wt, label | =价值
  加载文字寄存器（32 位）：从标签寻址的内存中加载一个字到 Wt。
- LDR Xt，标签 | =价值
  加载文字寄存器（64 位）：从标签寻址的内存中加载一个双字到 Xt。
- LDRSW Xt, 标签 | =价值
  Load Literal Signed Word (extended)：从内存中加载一个按标签寻址的单词，然后将其符号扩展到 Xt 中。

5.2.4 加载-存储一对寄存器

加载-存储对指令支持由基址寄存器 Xn 或 SP 组成的寻址模式，以及：

- 缩放的 7 位有符号立即偏移量，具有索引前和索引后写回选项

如果加载对指令为正在加载的两个寄存器指定相同的寄存器，则其中一个
可能会出现以下行为：

- 指令未分配

- 该指令被视为 NOP
- 该指令使用指定的寻址模式执行所有加载，并且正在加载的寄存器采用 UNKNOWN 值

如果加载对指令指定回写并且正在加载的寄存器之一也是基址寄存器，则
可能会出现以下行为之一：

- 指令未分配
- 该指令被视为 NOP
- 该指令使用指定的寻址模式执行所有加载，并且基址寄存器变为 UNKNOWN。此外，如果在此类指令期间发生异常，则基地址可能会被破坏，从而无法重复该指令。

如果存储对指令执行回写并且正在存储的寄存器之一也是基址寄存器，则可能发生以下行为之一：

- 指令未分配
- 该指令被视为 NOP
- 该指令执行所有使用指定寻址模式指定的寄存器的存储，但为基址寄存器存储的值是 UNKNOWN

- LDP Wt1, Wt2, addr
  加载对寄存器：将两个字从 addr 寻址的存储器加载到 Wt1 和 Wt2。
- LDP Xt1, Xt2, addr
  加载对寄存器（扩展）：从 addr 寻址的内存中加载两个双字到 Xt1 和 Xt2。
- LDPSW Xt1, Xt2, addr
  Load Pair Signed Words (extended) 从 addr 寻址的内存中加载两个字，然后将它们符号扩展为 Xt1 和 Xt2。
- STP Wt1, Wt2, addr
  存储对寄存器：将两个字从 Wt1 和 Wt2 存储到由 addr 寻址的存储器。
- STP Xt1, Xt2, addr
  存储对寄存器（扩展）：将两个双字从 Xt1 和 Xt2 存储到由 addr 寻址的内存中。

5.2.5 加载-存储Non-temporal Pair

LDNP 和 STNP 非时间对指令向内存系统提供了一个提示，即访问是“非时间”或“流式”的，并且不太可能在不久的将来再次访问，因此不需要保留在数据缓存中。但是，根据内存类型，它们可能允许预加载内存读取并收集内存写入，以加速大容量内存传输。

此外，作为正常内存排序规则的一个特殊例外，其中两次内存读取之间存在地址依赖性，并且第二次读取是由 Load Non-temporal Pair 指令生成的，那么在没有任何其他屏障机制来实现顺序的情况下，在被访问的内存地址的可共享域内，其他观察者可以以任何顺序观察这些内存访问。

LDNP 和 STNP 指令支持基址寄存器 Xn 或 SP 的寻址模式，此外：

- 缩放的 7 位有符号立即偏移量，没有索引前和索引后选项

如果 Load Non-temporal Pair 指令为正在加载的两个寄存器指定相同的寄存器，则
可能会出现以下行为之一：

- 指令未分配
- 该指令被视为 NOP
- 该指令使用指定的寻址模式执行所有加载，并且正在加载的寄存器采用 UNKNOWN 值

- LDNP Wt1, Wt2, [base,#imm]
  Load Non-temporal Pair：从内存中通过 base+imm 寻址的两个字加载到 Wt1 和 Wt2，并带有非时间提示。

- LDNP Xt1, Xt2, [base,#imm]
  Load Non-temporal Pair (extended)：将两个双字从 base+imm 寻址的内存中加载到 Xt1 和 Xt2，并带有非临时提示。
- STNP Wt1, Wt2, [base,#imm]
  存储非时间对：将 Wt1 和 Wt2 中的两个单词存储到由 base+imm 寻址的内存中，并带有非时间提示。
- STNP Xt1, Xt2, [base,#imm]
  存储非时间对（扩展）：将两个双字从 Xt1 和 Xt2 存储到由 base+imm 寻址的内存中，并带有非时间提示。

5.2.6 加载-存储非特权

当处理器处于 EL1 异常级别时，可以使用加载-存储非特权指令来执行内存访问，就像它处于 EL0（非特权）异常级别一样。 如果处理器处于任何其他异常级别，则执行该级别的正常内存访问。 （这些助记符中的字母“T”基于历史上的 ARM 约定，该约定将对非特权虚拟地址的访问描述为“翻译”）。 加载-存储非特权指令支持基址寄存器 Xn 或 SP 的寻址模式，此外：

- 未缩放、9 位、有符号立即偏移量，没有索引前和索引后选项

- LDTR Wt, [base,#simm9]
  加载非特权寄存器：在 EL1 时使用 EL0 特权将由 base+simm9 寻址的内存中的字加载到 Wt。
- LDTR Xt, [base,#simm9]
  加载非特权寄存器（扩展）：将双字从由 base+simm9 寻址的内存加载到 Xt，在 EL1 时使用 EL0 特权。
- LDTRB Wt, [base,#simm9]
  加载非特权字节：从 base+simm9 寻址的内存中加载一个字节，然后将其零扩展为 Wt，在 EL1 时使用 EL0 特权。
- LDTRSB Wt, [base,#simm9]
  Load Unprivileged Signed Byte：从base+simm9寻址的内存中加载一个字节，然后将其符号扩展到Wt，在EL1时使用EL0权限。
- LDTRSB Xt, [base,#simm9]
  Load Unprivileged Signed Byte (extended)：从内存中加载一个由 base+simm9 寻址的字节，然后在 EL1 时使用 EL0 权限将其符号扩展到 Xt。
- LDTRH Wt, [base,#simm9]
  Load Unprivileged Halfword：从base+simm9寻址的内存中加载一个半字，然后将其零扩展为 Wt，在EL1时使用EL0权限。
- LDTRSH Wt, [base,#simm9]
  Load Unprivileged Signed Halfword：从 base+simm9 寻址的内存中加载一个半字，然后将其符号扩展为 Wt，在 EL1 时使用 EL0 权限。
- LDTRSH Xt, [base,#simm9]
  Load Unprivileged Signed Halfword (extended)：从内存中加载一个由 base+simm9 寻址的半字，然后在 EL1 时使用 EL0 权限将其符号扩展到 Xt 中。
- LDTRSW Xt, [base,#simm9]
  Load Unprivileged Signed Word (extended)：从内存中加载一个由 base+simm9 寻址的字，然后在 EL1 时使用 EL0 权限将其符号扩展到 Xt。
- STTR Wt, [base,#simm9]
  存储非特权寄存器：将 Wt 中的一个字存储到由 base+simm9 寻址的内存中，在 EL1 时使用 EL0

特权。

- STTR Xt, [base,#simm9]
  存储非特权寄存器（扩展）：将双字从 Xt 存储到由 base+simm9 寻址的内存，在 EL1 时使用 EL0 特权。
- STTRB Wt, [base,#simm9]
  存储非特权字节：将一个字节从 Wt 存储到由 base+simm9 寻址的内存中，在 EL1 时使用 EL0 特权。
- STTRH Wt, [base,#simm9]
  Store Unprivileged Halfword：将一个半字从 Wt 存储到由 base+simm9 寻址的内存中，使用在 EL1 时的 EL0 特权

5.2.7 加载存储独占

加载独占指令将访问的物理地址标记为独占访问，由存储独占检查，允许对共享内存变量、信号量、互斥锁、自旋锁等进行“原子”读-修改-写操作。

加载-存储独占指令仅支持基址寄存器 Xn 或 SP 的简单寻址模式。 #0 的可选偏移量必须被汇编器接受，但在反汇编时可以省略。

需要自然对齐：未对齐的地址将导致对齐错误。 由加载独占对或存储独占对生成的内存访问必须与对的大小对齐，并且当存储独占对成功时，将导致整个内存位置的单副本原子更新。

- LDXR Wt, [base{,#0}]
  加载独占寄存器：从基址寻址的内存中加载一个字到 Wt。将物理地址记录为独占访问。
- LDXR Xt, [base{,#0}]
  加载独占寄存器（扩展）：从基址寻址的内存中加载一个双字到 Xt。将物理地址记录为独占访问。
- LDXRB Wt, [base{,#0}]
  加载独占字节：从基址寻址的内存中加载一个字节，然后将其零扩展为 Wt。将物理地址记录为独占访问。
- LDXRH Wt, [base{,#0}]
  加载独占半字：从基址寻址的内存中加载半字，然后将其零扩展为 Wt。将物理地址记录为独占访问。
- LDXP Wt, Wt2, [base{,#0}]
  加载独占对寄存器：从基址寻址的内存中加载两个字，并将其加载到 Wt 和 Wt2。将物理地址记录为独占访问。
- LDXP Xt, Xt2, [base{,#0}]
  Load Exclusive Pair Registers (extended)：从基址寻址的内存中加载两个双字到 Xt 和 Xt2。将物理地址记录为独占访问。
- STXR Ws, Wt, [base{,#0}]
  存储独占寄存器：将字从 Wt 存储到由基址寻址的内存中，并将 Ws 设置为返回的独占访问状态。
- STXR Ws, Xt, [base{,#0}]
  存储独占寄存器（扩展）：将双字从 Xt 存储到由基址寻址的内存中，并将 Ws 设置为返回的独占访问状态。
- STXRB Ws, Wt, [base{,#0}]
  存储独占字节：将字节从 Wt 存储到由基址寻址的内存中，并将 Ws 设置为返回的独占访问状态。
- STXRH Ws, Wt, [base{,#0}]
  存储独占半字：将半字从 Xt 存储到由基址寻址的内存中，并将 Ws 设置为返回的独占访问状态。

- STXP Ws, Wt, Wt2, [base{,#0}]
  Store Exclusive Pair：将 Wt 和 Wt2 两个字存储到由基址寻址的内存中，并将 Ws 设置为返回的独占访问状态。
- STXP Ws, Xt, Xt2, [base{,#0}]
  Store Exclusive Pair (extended)：将Xt和Xt2的两个双字存储到由基址寻址的内存中，并将Ws设置为返回的独占访问状态

5.2.8 Load-Acquire / Store-Release

加载获取是一种加载，它保证在加载获取之后按程序顺序出现的所有加载和存储将在该观察者观察到加载获取之后被每个观察者观察到，但是对于加载和存储之前出现的加载和存储只字未提。获得。

在每个观察者观察到在 store-release 之前按程序顺序出现的任何加载或存储之后，每个观察者都会观察到 store-release，但对于在 store-release 之后出现的加载和存储只字不提。此外，每个观察者将按程序顺序观察存储释放和加载获取。

进一步的考虑是所有的 store-release 操作必须是多副本原子的：也就是说，如果一个 agent 已经看到了 store-release，那么所有的 agent 都已经看到了 store-release。普通 store 不需要多副本原子。

load-acquire 和 store-release 指令仅支持基址寄存器 Xn 或 SP 的简单寻址模式。 #0 的可选偏移量必须被汇编器接受，但在反汇编时可以省略

需要自然对齐：未对齐的地址将导致对齐错误。

5.2.8.1 Non-exclusive

- LDAR Wt, [base{,#0}]
  Load-Acquire Register：将一个字从由base寻址的内存加载到Wt。
- LDAR Xt, [base{,#0}]
  Load-Acquire Register (extended)：从基址寻址的内存中加载一个双字到 Xt。
- LDARB Wt, [base{,#0}]
  Load-Acquire Byte：从基址寻址的内存中加载一个字节，然后将其零扩展为 Wt。
- LDARH Wt, [base{,#0}]
  Load-Acquire Halfword：从基址寻址的内存中加载一个半字，然后将其零扩展为 Wt。
- STLR Wt, [base{,#0}]
  存储释放寄存器：将一个字从 Wt 存储到由基址寻址的内存中。
- STLR Xt, [base{,#0}]
  存储释放寄存器（扩展）：将一个双字从 Xt 存储到由基址寻址的内存中。
- STLRB Wt, [base{,#0}]
  Store-Release Byte：将一个字节从 Wt 存储到由基址寻址的内存中。
- STLRH Wt, [base{,#0}]
  Store-Release Halfword：将一个半字从 Wt 存储到由基址寻址的内存中。

5.2.8.2 Exclusive

- LDAXR Wt, [base{,#0}]
  Load-Acquire Exclusive Register：将字从由base寻址的内存加载到Wt。将物理地址记录为独占访问。
- LDAXR Xt, [base{,#0}]
  Load-Acquire Exclusive Register (extended)：将双字从由基址寻址的内存加载到 Xt。将物理地址记录为独占访问。

- LDAXRB Wt, [base{,#0}]
  Load-Acquire Exclusive Byte：从基址寻址的内存中加载字节，然后将其零扩展为 Wt。将物理地址记录为独占访问。
- LDAXRH Wt, [base{,#0}]
  Load-Acquire Exclusive Halfword：从基址寻址的内存中加载半字，然后将其零扩展为 Wt。将物理地址记录为独占访问。
- LDAXP Wt, Wt2, [base{,#0}]
  Load-Acquire Exclusive Pair Registers：从基址寻址的存储器中加载两个字到 Wt 和 Wt2。将物理地址记录为独占访问。
- LDAXP Xt, Xt2, [base{,#0}]
  Load-Acquire Exclusive Pair Registers (extended)：从基址寻址的内存中加载两个双字到 Xt 和 Xt2。将物理地址记录为独占访问。
- STLXR Ws, Wt, [base{,#0}]
  Store-Release Exclusive Register：将字从 Wt 存储到由基址寻址的内存中，并将 Ws 设置为返回的独占访问状态。
- STLXR Ws, Xt, [base{,#0}]
  Store-Release Exclusive Register (extended)：将双字从 Xt 存储到由基址寻址的内存中，并将 Ws 设置为返回的独占访问状态。
- STLXRB Ws, Wt, [base{,#0}]
  Store-Release Exclusive Byte：将字节从 Wt 存储到由 base 寻址的内存中，并将 Ws 设置为返回的独占访问状态。
- STLXRH Ws, Xt|Wt, [base{,#0}]
  Store-Release Exclusive Halfword：将来自Wt的半字存储到由base寻址的内存中，并将Ws设置为返回的独占访问状态。
- STLXP Ws, Wt, Wt2, [base{,#0}]
  Store-Release Exclusive Pair：将 Wt 和 Wt2 两个字存储到 base 寻址的内存中，并将 Ws 设置为返回的独占访问状态。
- STLXP Ws, Xt, Xt2, [base{,#0}]
  Store-Release Exclusive Pair (extended)：将 Xt 和 Xt2 中的两个双字存储到由基址寻址的内存中，并将 Ws 设置为返回的独占访问状态。

5.2.9 预取内存

预取存储器指令向存储器系统发出信号，表明在不久的将来可能会从指定地址访问存储器。 内存系统可以通过采取预期在内存访问确实发生时加快内存访问的动作来做出响应，例如将指定地址预加载到一个或多个高速缓存中。 由于这些只是提示，因此 CPU 将任何或所有预取指令视为无操作是有效的。

预取指令支持多种寻址模式，包括基址寄存器 Xn 或 SP，以及以下之一：

- 缩放的 12 位无符号立即偏移量，没有索引前和索引后选项。
- 未缩放、9 位、有符号立即偏移量，没有索引前和索引后选项。
- 缩放或未缩放的 64 位寄存器偏移量。
- 缩放或未缩放的 32 位扩展寄存器偏移量。

此外：

- 相对于 PC 的地址或标签，在当前 PC 的 ±1MB 范围内。
- 如果偏移量被缩放，就好像访问大小为 8 字节。

`PRFM <prfop>, addr|label`
预取内存，使用 提示，其中 是以下之一：
PLDL1KEEP, PLDL1STRM, PLDL2KEEP, PLDL2STRM, PLDL3KEEP, PLDL3STRM
PSTL1KEEP, PSTL1STRM, PSTL2KEEP, PSTL2STRM, PSTL3KEEP, PSTL3STRM

> `<prfop> ::= <type><target><policy> | #uimm5`
> ::= “PLD” (prefetch for load) | “PST” (prefetch for store)
> ::= “L1” (L1 cache) | “L2” (L2 cache) | “L3” (L3 cache)
> ::= “KEEP” (retained or temporal prefetch, i.e. allocate in cache normally)
> |“STRM” (streaming or non-temporal prefetch, i.e. memory used only once)
> #uimm5 ::= represents the unallocated hint encodings as a 5-bit immediate

5.3 数据处理（立即数）

数据处理（立即数）支持以下指令组：

- 算术（立即）
- 逻辑（立即）
- 移动（立即）
- 位域（操作）
- 班次（立即）
- 符号/零扩展

5.3.1 算术（立即数）

这些指令接受显示为`aimm`的算术立即数，它被编码为左移0或12位的12位无符号立即数。 在汇编语言中，这可以写成：

- `#uimm12, LSL #sh`
  12 位无符号立即数，显式左移 0 或 12。
- `#uimm24`
  24 位无符号立即数。 汇编器应确定 uimm12 的适当值，并以 0 或 12 的最低可能移位来生成请求的值； 如果该值在 bits23:12 和 bits11:0 中包含非零位，则将产生错误。
- `#nimm25`
  “对程序员友好”的汇编器可以接受介于 -(224-1) 和 -1 之间的负立即数，导致它将请求的 ADD 操作转换为 SUB，反之亦然，然后将立即数的绝对值编码为 对于 uimm24。 然而，架构汇编语言不需要这种行为。

反汇编程序通常应该使用 uimm24 形式输出算术立即数，除非编码的移位量不是可以使用的最低可能移位（例如，#0，LSL #12 不能使用 uimm24 形式输出）。

不设置条件标志的算术指令可以读取和/或写入当前堆栈指针，例如在函数序言或结尾调整堆栈指针；标志设置指令可以读取堆栈指针，但不能写入。

- ADD Wd|WSP, Wn|WSP, #aimm
  Add (immediate): Wd|WSP = Wn|WSP + aimm.
- ADD Xd|SP, Xn|SP, #aimm
  Add (extended immediate): Xd|SP = Xn|SP + aimm.
- ADDS Wd, Wn|WSP, #aimm
  Add and set flags (immediate): Wd = Wn|WSP + aimm, setting the condition flags.
- ADDS Xd, Xn|SP, #aimm
  Add and set flags (extended immediate): Xd = Xn|SP + aimm, setting the condition flags.

- SUB Wd|WSP, Wn|WSP, #aimm
  Subtract (immediate): Wd|WSP = Wn|WSP - aimm.
- SUB Xd|SP, Xn|SP, #aimm
  Subtract (extended immediate): Xd|SP = Xn|SP - aimm.
- SUBS Wd, Wn|WSP, #aimm
  Subtract and set flags (immediate): Wd = Wn|WSP - aimm, setting the condition flags.
- SUBS Xd, Xn|SP, #aimm
  Subtract and set flags (extended immediate): Xd = Xn|SP - aimm, setting the condition flags.
- CMP Wn|WSP, #aimm
  Compare (immediate): alias for SUBS WZR,Wn|WSP,#aimm.
- CMP Xn|SP, #aimm
  Compare (extended immediate): alias for SUBS XZR,Xn|SP,#aimm.
- CMN Wn|WSP, #aimm
  Compare negative (immediate): alias for ADDS WZR,Wn|WSP,#aimm.
- CMN Xn|SP, #aimm
  Compare negative (extended immediate): alias for ADDS XZR,Xn|SP,#aimm.
- MOV Wd|WSP, Wn|WSP
  Move (register): alias for ADD Wd|WSP,Wn|WSP,#0, but only when one or other of the registers is WSP. In other cases the ORR Wd,WZR,Wn instruction is used.
- MOV Xd|SP, Xn|SP
  Move (extended register): alias for ADD Xd|SP,Xn|SP,#0, but only when one or other of the registers is SP. In other cases the ORR Xd,XZR,Xn instruction is used

5.3.2 逻辑（立即数）

逻辑立即数指令接受位掩码立即数 bimm32 或 bimm64。 这样的立即数包括在 2、4、8、16、32 或 64 位的元素内具有至少一个非零位和至少一个零位的单个连续序列； 然后元素被复制到整个寄存器宽度，或者这个值的按位反转。 全零和全一的立即数可能不会被编码为位掩码立即数，因此汇编程序必须为具有这种立即数的逻辑指令生成错误，或者程序员友好的汇编程序可以将其转换为其他指令 这达到了预期的结果。

逻辑（立即）指令可以写入当前堆栈指针，例如在函数序言中对齐堆栈指针。

注意：除 ANDS 外，逻辑立即数指令不设置条件标志，但“有趣”的结果通常可以直接控制 CBZ、CBNZ、TBZ 或 TBNZ 条件分支。

- AND Wd|WSP, Wn, #bimm32
  Bitwise AND (immediate): Wd|WSP = Wn AND bimm32.
- AND Xd|SP, Xn, #bimm64
  Bitwise AND (extended immediate): Xd|SP = Xn AND bimm64.
- - ANDS Wd, Wn, #bimm32
  Bitwise AND and Set Flags (immediate): Wd = Wn AND bimm32, setting N & Z condition flags based on the result and clearing the C & V flags.
  ANDS Xd, Xn, #bimm64
  Bitwise AND and Set Flags (extended immediate): Xd = Xn AND bimm64, setting N & Z condition flags based on the result and clearing the C & V flags.

- EOR Wd|WSP, Wn, #bimm32
  Bitwise exclusive OR (immediate): Wd|WSP = Wn EOR bimm32.
- EOR Xd|SP, Xn, #bimm64
  Bitwise exclusive OR (extended immediate): Xd|SP = Xn EOR bimm64.
- ORR Wd|WSP, Wn, #bimm32
  Bitwise inclusive OR (immediate): Wd|WSP = Wn OR bimm32.
- ORR Xd|SP, Xn, #bimm64
  Bitwise inclusive OR (extended immediate): Xd|SP = Xn OR bimm64.
- MOVI Wd, #bimm32
  Move bitmask (immediate): alias for ORR Wd,WZR,#bimm32, but may disassemble as MOV, see below.
- MOVI Xd, #bimm64
  Move bitmask (extended immediate): alias for ORR Xd,XZR,#bimm64, but may disassemble as MOV, see below.
- TST Wn, #bimm32
  Bitwise test (immediate): alias for ANDS WZR,Wn,#bimm32.
- TST Xn, #bimm64
  Bitwise test (extended immediate): alias for ANDS XZR,Xn,#bimm64

5.3.3 Move (wide immediate)

这些指令将 16 位立即数（或反转立即数）插入目标寄存器中的 16 位对齐位置，其他目标寄存器位的值取决于所使用的变体。 移位量pos可以是寄存器大小的16的任意倍数。 省略“LSL #pos”意味着偏移 0。

- MOVZ Wt, #uimm16{, LSL #pos}
  Move with Zero (immediate): Wt = LSL(uimm16, pos).Usually disassembled as MOV, see below.
- MOVZ Xt, #uimm16{, LSL #pos}
  Move with Zero (extended immediate): Xt = LSL(uimm16, pos). Usually disassembled as MOV, see below.
- MOVN Wt, #uimm16{, LSL #pos}
  Move with NOT (immediate): Wt = NOT(LSL(uimm16, pos)). Usually disassembled as MOV, see below.
- MOVN Xt, #uimm16{, LSL #pos}
  Move with NOT (extended immediate): Xt = NOT(LSL(uimm16, pos)). Usually disassembled as MOV, see below.
- MOVK Wt, #uimm16{, LSL #pos}
  Move with Keep (immediate): Wtpos+15:pos = uimm16
- MOVK Xt, #uimm16{, LSL #pos}
  Move with Keep (extended immediate): Xtpos+15:pos = uimm16.

5.3.3.1 Move (immediate)

- MOV Wd, #simm32
  生成单个 MOVZ、MOVN 或 MOVI 指令的合成汇编指令，将 32 位立即值加载到寄存器 Wd 中。 如果这些指令中的一条指令不能创建立即数，则将导致汇编程序错误。 如果有选择，那么为了确保可逆性，汇编器必须优先选择 MOVZ 而不是 MOVN，以及 MOVZ 或 MOVN 而不是 MOVI。 反汇

编程序可以将 MOVI、MOVZ 和 MOVN 作为 MOV 助记符输出，除非 MOVI 具有可由 MOVZ 或 MOVN 指令生成的立即数，或 MOVN 具有可由 MOVZ 编码的立即数，或 MOVZ/MOVN #0 具有除 LSL #0 以外的移位量，在这种情况下必须使用机器指令助记符。

- MOV Xd, #simm64
  与 MOV 相同，但用于将 64 位立即数加载到寄存器 Xd 中。

5.3.4 地址生成（Address Generation）

- ADRP Xd, label
  页面地址：符号扩展了一个 21 位偏移量，将其左移 12 位并将其与 PC 的值相加，并清除其低 12 位，将结果写入寄存器 Xd。 这将计算包含标签的 4KiB 对齐内存区域的基地址，并设计用于与提供标签地址的低 12 位的加载、存储或添加指令结合使用。 这允许使用两条指令对 PC 的 ±4GiB 内的任何位置进行与位置无关的寻址，前提是动态重定位以 4KiB 的最小粒度完成（即标签地址的底部 12 位不受重定位的影响）。 术语“页面”是 4KiB 重定位粒度的简写，不一定与虚拟内存页面大小有关。
- ADR Xd, label
  地址：将 21 位有符号字节偏移量添加到程序计数器，将结果写入寄存器 Xd。 用于计算 PC 的 ±1MiB 内任何位置的有效地址

5.3.5 位域操作

- BFM Wd, Wn, #r, #s
  Bitfield Move: if s>=r then Wds-r:0 = Wns:r, else Wd<32+s-r,32-r> = Wns:0.
  保持 Wd 中的其他位不变。
- BFM Xd, Xn, #r, #s
  Bitfield Move: if s>=r then Xds-r:0 = Xns:r, else Xd<64+s-r,64-r> = Xns:0.
  保持 Xd 中的其他位不变。
- SBFM Wd, Wn, #r, #s
  Signed Bitfield Move: if s>=r then Wds-r:0 = Wns:r, else Wd<32+s-r,32-r> = Wns:0.
  将目标位域左侧的位设置为其最左侧位的副本，并将右侧的位设置为零。
- SBFM Xd, Xn, #r, #s
  Signed Bitfield Move: if s>=r then Xds-r:0 = Xns:r, else Xd<64+s-r,64-r> = Xns:0.
  将目标位域左侧的位设置为其最左侧位的副本，并将右侧的位设置为零。
- UBFM Wd, Wn, #r, #s
  Unsigned Bitfield Move: if s>=r then Wds-r:0 = Wns:r, else Wd<32+s-r,32-r> = Wns:0.
  将目标位域左侧和右侧的位设置为零。
- UBFM Xd, Xn, #r, #s
  Unsigned Bitfield Move: if s>=r then Xds-r:0 = Xns:r, else Xd<32+s-r,32-r> = Xns:0.
  将目标位域左侧和右侧的位设置为零。

以下别名提供了更熟悉的位域插入和提取助记符，具有常规位域 lsb 和宽度操作数，必须满足约束 lsb >= 0 && width >= 1 && lsb+width <= reg.size

- BFI Wd, Wn, #lsb, #width
  Bitfield Insert: alias for BFM Wd,Wn,#((32-lsb)&31),#(width-1).
  Preferred for disassembly when s < r.
- BFI Xd, Xn, #lsb, #width
  Bitfield Insert (extended): alias for BFM Xd,Xn,#((64-lsb)&63),#(width-1).
  Preferred for disassembly when s < r.

- BFXIL Wd, Wn, #lsb, #width
  Bitfield Extract and Insert Low: alias for BFM Wd,Wn,#lsb,#(lsb+width-1).
  Preferred for disassembly when s >= r.
- BFXIL Xd, Xn, #lsb, #width
  Bitfield Extract and Insert Low (extended): alias for BFM Xd,Xn,#lsb,#(lsb+width-1).
  Preferred for disassembly when s >= r.
- SBFIZ Wd, Wn, #lsb, #width
  Signed Bitfield Insert in Zero: alias for) SBFM Wd,Wn,#((32-lsb)&31),#(width-1).
  Preferred for disassembly when s < r.
- SBFIZ Xd, Xn, #lsb, #width
  Signed Bitfield Insert in Zero (extended): alias for SBFM Xd,Xn,#((64-lsb)&63),#(width-1).
  Preferred for disassembly when s < r.
- SBFX Wd, Wn, #lsb, #width
  Signed Bitfield Extract: alias for SBFM Wd,Wn,#lsb,#(lsb+width-1).
  Preferred for disassembly when s >= r.
- SBFX Xd, Xn, #lsb, #width
  Signed Bitfield Extract (extended): alias for SBFM Xd,Xn,#lsb,#(lsb+width-1).
  Preferred for disassembly when s >= r.
- UBFIZ Wd, Wn, #lsb, #width
  Unsigned Bitfield Insert in Zero: alias for UBFM Wd,Wn,#((32-lsb)&31),#(width-1).
  Preferred for disassembly when s < r.
- UBFIZ Xd, Xn, #lsb, #width
  Unsigned Bitfield Insert in Zero (extended): alias for UBFM Xd,Xn,#((64-lsb)&63),#(width-1).
  Preferred for disassembly when s < r.
- UBFX Wd, Wn, #lsb, #width
  Unsigned Bitfield Extract: alias for UBFM Wd,Wn,#lsb,#(lsb+width-1).
  Preferred for disassembly when s >= r.
- UBFX Xd, Xn, #lsb, #width
  Unsigned Bitfield Extract (extended): alias for UBFM Xd,Xn,#lsb,#(lsb+width-1).
  Preferred for disassembly when s >= r.

5.3.6 提取（立即数）-- Extract (immediate)

- EXTR Wd, Wn, Wm, #lsb
  Extract: Wd = Wn:Wm<lsb+31,lsb>. The bit position lsb must be in the range 0 to 31.
- EXTR Xd, Xn, Xm, #lsb
  Extract (extended): Xd = Xn:Xm<lsb+63,lsb>. The bit position lsb must be in the range 0 to 63.

5.3.7 Shift（立即数）

所有立即移位和旋转都是别名，使用 Bitfield 或 Extract 指令实现。 在所有情况下，立即移位量 uimm 必须在 0 到 (reg.size - 1) 的范围内。

- ASR Wd, Wn, #uimm
  Arithmetic Shift Right (immediate): alias for SBFM Wd,Wn,#uimm,#31.
- ASR Xd, Xn, #uimm
  Arithmetic Shift Right (extended immediate): alias for SBFM Xd,Xn,#uimm,#63.

- LSL Wd, Wn, #uimm
  Logical Shift Left (immediate): alias for UBFM Wd,Wn,#((32-uimm)&31),#(31-uimm).
- LSL Xd, Xn, #uimm
  Logical Shift Left (extended immediate): alias for UBFM Xd,Xn,#((64-uimm)&63),#(63-uimm)
- LSR Wd, Wn, #uimm
  Logical Shift Left (immediate): alias for UBFM Wd,Wn,#uimm,#31.
- LSR Xd, Xn, #uimm
  Logical Shift Left (extended immediate): alias for UBFM Xd,Xn,#uimm,#31.
- ROR Wd, Wm, #uimm
  Rotate Right (immediate): alias for EXTR Wd,Wm,Wm,#uimm.
- ROR Xd, Xm, #uimm
  Rotate Right (extended immediate): alias for EXTR Xd,Xm,Xm,#uimm.

5.3.8 符号/零扩展

- SXT[BH] Wd, Wn
  Signed Extend Byte|Halfword: alias for SBFM Wd,Wn,#0,#7|15.
- SXT[BHW] Xd, Wn
  Signed Extend Byte|Halfword|Word (extended): alias for SBFM Xd,Xn,#0,#7|15|31.
- UXT[BH] Wd, Wn
  Unsigned Extend Byte|Halfword: alias for UBFM Wd,Wn,#0,#7|15.
- UXT[BHW] Xd, Wn
  Unsigned Extend Byte|Halfword|Word (extended): alias for UBFM Xd,Xn,#0,#7|15|31.

5.4 数据处理（寄存器）

数据处理（寄存器）支持以下指令组：

- 算术（移位寄存器）
- 算术（扩展寄存器）
- 逻辑（移位寄存器）
- 算术（未移位寄存器）
- 移位（寄存器）
- 按位运算

5.4.1 算术（移位寄存器）

移位寄存器指令在执行算术运算之前对最终源操作数值应用可选移位。 指令的寄存器大小控制在右移或旋转时将新位输入到中间结果的位置（即位 63 或 31）。

移位运算符 LSL、ASR 和 LSR 接受 0 到 reg.size - 1 范围内的立即移位量。

省略移位运算符意味着“LSL #0”（即没有移位），并且“LSL #0”不应由反汇编程序输出； 必须输出所有其他零位移。

寄存器名称 SP 和 WSP 不能与此类指令一起使用，请参阅第 5.4.2 节

- ADD Wd, Wn, Wm{, ashift #imm}
  Add (register): Wd = Wn + ashift(Wm, imm).
- ADD Xd, Xn, Xm{, ashift #imm}
  Add (extended register): Xd = Xn + ashift(Xm, imm).

- ADDS Wd, Wn, Wm{, ashift #imm}
  Add and Set Flags (register): Wd = Wn + ashift(Wm, imm), setting condition flags.
- ADDS Xd, Xn, Xm{, ashift #imm}
  Add and Set Flags (extended register): Xd = Xn + ashift(Xm, imm), setting condition flags.
- SUB Wd, Wn, Wm{, ashift #imm}
  Subtract (register): Wd = Wn - ashift(Wm, imm).
- SUB Xd, Xn, Xm{, ashift #imm}
  Subtract (extended register): Xd = Xn - ashift(Xm, imm).
- SUBS Wd, Wn, Wm{, ashift #imm}
  Subtract and Set Flags (register): Wd = Wn - ashift(Wm, imm), setting condition flags.
- SUBS Xd, Xn, Xm{, ashift #imm}
  Subtract and Set Flags (extended register): Xd = Xn - ashift(Xm, imm), setting condition flags.
- CMN Wn, Wm{, ashift #imm}
  Compare Negative (register): alias for ADDS WZR, Wn, Wm{, ashift #imm}.
- CMN Xn, Xm{, ashift #imm}
  Compare Negative (extended register): alias for ADDS XZR, Xn, Xm{, ashift #imm}.
- CMP Wn, Wm{, ashift #imm}
  Compare (register): alias for SUBS WZR, Wn, Wm{,ashift #imm}.
- CMP Xn, Xm{, ashift #imm}
  Compare (extended register): alias for SUBS XZR, Xn, Xm{, ashift #imm}.
- NEG Wd, Wm{, ashift #imm}
  Negate: alias for SUB Wd, WZR, Wm{, ashift #imm}.
- NEG Xd, Xm{, ashift #imm}
  Negate (extended): alias for SUB Xd, XZR, Xm{, ashift #imm}.
- NEGS Wd, Wm{, ashift #imm}
  Negate and Set Flags: alias for SUBS Wd, WZR, Wm{, ashift #imm}.
- NEGS Xd, Xm{, ashift #imm}
  Negate and Set Flags (extended): alias for SUBS Xd, XZR, Xm{, ashift #imm}

5.4.2 算术（扩展寄存器）

扩展寄存器指令与移位寄存器形式的不同之处在于：
（一）。 非标志设置变体允许将堆栈指针用作目标寄存器和第一源寄存器中的一个或两个。 标志设置变体只允许堆栈指针作为第一个源寄存器。
（2）。 它们提供了第二个源寄存器值的一部分的可选符号或零扩展，然后是可选的立即左移 1 和 4（含）。

“扩展移位”由强制扩展运算符 SXTB、SXTH、SXTW、SXTX、UXTB、UXTH、UXTW 或 UXTX 描述，其后是可选的左移量。如果省略移位量，则默认为零，并且反汇编程序不应输出零移位量。

对于 64 位指令形式，操作符 UXTX 和 SXTX（首选 UXTX）都执行第二个源寄存器的“无操作”扩展，然后是可选的移位。当且仅当 UXTX 在至少一个操作数中与寄存器名称 SP 结合使用时，则首选别名 LSL，在这种情况下，运算符和移位量都可以省略，表示“LSL #0”。

类似地，对于 32 位指令形式，运算符 UXTW 和 SXTW（首选 UXTW）都执行第二个源寄存器的“无操作”扩展，然后是可选的移位。当且仅当 UXTW 与至少一个操作数中的寄存器名称 WSP 结合使用时，别名 LSL 是首选。在这些指令的 64 位形式中，除了（可能省略）UXTX/LSL 和 SXTX 运算符之外，最后的寄存器操作数写为 Wm。例如：

```
CMP X4, w5, SXTW
ADD X1, X2, w3, UXTB #2
SUB SP, SP, X1 // SUB SP, SP, X1, UXTX #0
```

- ADD Wd|WSP, Wn|WSP, Wm, extend {#imm}
  Add (register, extending): Wd|WSP = Wn|WSP + LSL(extend(Wm),imm).
- ADD Xd|SP, Xn|SP, Wm, extend {#imm}
  Add (extended register, extending): Xd|SP = Xn|SP + LSL(extend(Wm),imm).
- ADD Xd|SP, Xn|SP, Xm{, UXTX|LSL #imm}
  Add (extended register, extending): Xd|SP = Xn|SP + LSL(Xm,imm).
- ADDS Wd, Wn|WSP, Wm, extend {#imm}
  Add and Set Flags (register, extending): Wd = Wn|WSP + LSL(extend(Wm),imm), setting the condition flags.
- ADDS Xd, Xn|SP, Wm, extend {#imm}
  Add and Set Flags (extended register, extending): Xd = Xn|SP + LSL(extend(Wm),imm), setting the condition flags.
- ADDS Xd, Xn|SP, Xm{, UXTX|LSL #imm}
  Add and Set Flags (extended register, extending): Xd = Xn|SP + LSL(Xm,imm), setting the condition flags.
- SUB Wd|WSP, Wn|WSP, Wm, extend {#imm}
  Subtract (register, extending): Wd|WSP = Wn|WSP - LSL(extend(Wm),imm).
- SUB Xd|SP, Xn|SP, Wm, extend {#imm}
  Subtract (extended register, extending): Xd|SP = Xn|SP - LSL(extend(Wm),imm).
- SUB Xd|SP, Xn|SP, Xm{, UXTX|LSL #imm}
  Subtract (extended register, extending): Xd|SP = Xn|SP - LSL(Xm,imm).
- SUBS Wd, Wn|WSP, Wm, extend {#imm}
  Subtract and Set Flags (register, extending): Wd = Wn|WSP - LSL(extend(Wm),imm), setting the condition flags.
- SUBS Xd, Xn|SP, Wm, extend {#imm}
  Subtract and Set Flags (extended register, extending): Xd = Xn|SP - LSL(extend(Wm),imm), setting the condition flags.
- SUBS Xd, Xn|SP, Xm{, UXTX|LSL #imm}
  Subtract and Set Flags (extended register, extending): Xd = Xn|SP - LSL(Xm,imm), setting the condition flags.
- CMN Wn|WSP, Wm, extend {#imm}
  Compare Negative (register, extending): alias for ADDS WZR,Wn,Wm,extend {#imm}.
- CMN Xn|SP, Wm, extend {#imm}
  Compare Negative (extended register, extending): alias for ADDS XZR,Xn,Wm,extend {#imm}.
- CMN Xn|SP, Xm{, UXTX|LSL #imm}
  Compare Negative (extended register, extending): alias for ADDS XZR,Xn,Xm{,UXTX|LSL #imm}.
- CMP Wn|WSP, Wm, extend {#imm}
  Compare (register, extending): alias for SUBS WZR,Wn,Wm,extend {#imm}.
- CMP Xn|SP, Wm, extend {#imm}
  Compare (extended register, extending): alias for SUBS XZR,Xn,Wm,extend {#imm}.

- CMP Xn|SP, Xm{, UXTX|LSL #imm}
  Compare (extended register, extending): alias for SUBS XZR,Xn,Xm{,UXTX|LSL #imm}.

5.4.3 逻辑（移位寄存器）

逻辑（移位寄存器）指令在执行主操作之前将可选的移位运算符应用于其最终源操作数。 指令的寄存器大小控制在右移或旋转时将新位输入到中间结果的位置（即位 63 或 31）。

移位运算符 LSL、ASR、LSR 和 ROR 接受 0 到 reg.size - 1 范围内的立即移位量。

省略移位运算符意味着“LSL #0”（即没有移位），并且反汇编程序不应输出“LSL #0” - 但是必须输出所有其他零移位。

注意：除了 ANDS 和 BICS 逻辑指令不设置条件标志，但“有趣”的结果通常可以直接控制 CBZ、CBNZ、TBZ 或 TBNZ 条件分支。

- AND Wd, Wn, Wm{, lshift #imm}
  Bitwise AND (register): Wd = Wn AND lshift(Wm, imm).
- AND Xd, Xn, Xm{, lshift #imm}
  Bitwise AND (extended register): Xd = Xn AND lshift(Xm, imm).
- ANDS Wd, Wn, Wm{, lshift #imm}
  Bitwise AND and Set Flags (register): Wd = Wn AND lshift(Wm, imm), setting N & Z condition flags based on the result and clearing the C & V flags.
- ANDS Xd, Xn, Xm{, lshift #imm}
  Bitwise AND and Set Flags (extended register): Xd = Xn AND lshift(Xm, imm), setting N & Z condition flags based on the result and clearing the C & V flags.
- BIC Wd, Wn, Wm{, lshift #imm}
  Bit Clear (register): Wd = Wn AND NOT(lshift(Wm, imm)).
- BIC Xd, Xn, Xm{, lshift #imm}
  Bit Clear (extended register): Xd = Xn AND NOT(lshift(Xm, imm)).
- BICS Wd, Wn, Wm{, lshift #imm}
  Bit Clear and Set Flags (register): Wd = Wn AND NOT(lshift(Wm, imm)), setting N & Z condition flags based on the result and clearing the C & V flags.
- BICS Xd, Xn, Xm{, lshift #imm}
  Bit Clear and Set Flags (extended register): Xd = Xn AND NOT(lshift(Xm, imm)), setting N & Z condition flags based on the result and clearing the C & V flags.
- EON Wd, Wn, Wm{, lshift #imm}
  Bitwise exclusive OR NOT (register): Wd = Wn EOR NOT(lshift(Wm, imm)).
- EON Xd, Xn, Xm{, lshift #imm}
  Bitwise exclusive OR NOT (extended register): Xd = Xn EOR NOT(lshift(Xm, imm)).
- EOR Wd, Wn, Wm{, lshift #imm}
  Bitwise exclusive OR (register): Wd = Wn EOR lshift(Wm, imm).
- EOR Xd, Xn, Xm{, lshift #imm}
  Bitwise exclusive OR (extended register): Xd = Xn EOR lshift(Xm, imm).
- ORR Wd, Wn, Wm{, lshift #imm}
  Bitwise inclusive OR (register): Wd = Wn OR lshift(Wm, imm).

- ORR Xd, Xn, Xm{, lshift #imm}
  Bitwise inclusive OR (extended register): Xd = Xn OR lshift(Xm, imm).
- ORN Wd, Wn, Wm{, lshift #imm}
  Bitwise inclusive OR NOT (register): Wd = Wn OR NOT(lshift(Wm, imm)).
- ORN Xd, Xn, Xm{, lshift #imm}
  Bitwise inclusive OR NOT (extended register): Xd = Xn OR NOT(lshift(Xm, imm)).
- MOV Wd, Wm
  Move (register): alias for ORR Wd,WZR,Wm.
- MOV Xd, Xm
  Move (extended register): alias for ORR Xd,XZR,Xm.
- MVN Wd, Wm{, lshift #imm}
  Move NOT (register): alias for ORN Wd,WZR,Wm{,lshift #imm}.
- MVN Xd, Xm{, lshift #imm}
  Move NOT (extended register): alias for ORN Xd,XZR,Xm{,lshift #imm}.
- TST Wn, Wm{, lshift #imm}
  Bitwise Test (register): alias for ANDS WZR,Wn,Wm{,lshift #imm}.
- TST Xn, Xm{, lshift #imm}
  Bitwise Test (extended register): alias for ANDS XZR,Xn,Xm{,lshift #imm}.

5.4.4 变体位(Variable Shift)

Wm 或 Xm 中的可变移位量为正，并以寄存器大小为模。 例如，Xm 包含值 65 的扩展 64 位移位将导致位移 (65 MOD 64) = 1 位。 机器指令如下：

- ASRV Wd, Wn, Wm
  Arithmetic Shift Right Variable: Wd = ASR(Wn, Wm & 0x1f).
- ASRV Xd, Xn, Xm
  Arithmetic Shift Right Variable (extended): Xd = ASR(Xn, Xm & 0x3f).
- LSLV Wd, Wn, Wm
  Logical Shift Left Variable: Wd = LSL(Wn, Wm & 0x1f).
- LSLV Xd, Xn, Xm
  Logical Shift Left Variable (extended register): Xd = LSL(Xn, Xm & 0x3f).
- LSRV Wd, Wn, Wm
  Logical Shift Right Variable: Wd = LSR(Wn, Wm & 0x1f).
- LSRV Xd, Xn, Xm
  Logical Shift Right Variable (extended): Xd = LSR(Xn, Xm & 0x3f).
- RORV Wd, Wn, Wm
  Rotate Right Variable: Wd = ROR(Wn, Wm & 0x1f).
- RORV Xd, Xn, Xm
  Rotate Right Variable (extended): Xd = ROR(Xn, Xm & 0x3f)

然而，“可变移位”机器指令有一组首选的“移位（寄存器）”别名，它们与别处描述的移位（立即）别名相匹配：

- ASR Wd, Wn, Wm
  Arithmetic Shift Right (register): preferred alias for ASRV Wd, Wn, Wm.
- ASR Xd, Xn, Xm
  Arithmetic Shift Right (extended register): preferred alias for ASRV Xd, Xn, Xm.
- LSL Wd, Wn, Wm
  Logical Shift Left (register): preferred alias for LSLV Wd, Wn, Wm.
- LSL Xd, Xn, Xm
  Logical Shift Left (extended register): preferred alias for LSLV Xd, Xn, Xm.
- LSR Wd, Wn, Wm
  Logical Shift Right (register): preferred alias for LSRV Wd, Wn, Wm.
- LSR Xd, Xn, Xm
  Logical Shift Right (extended register): preferred alias for LSRV Xd, Xn, Xm
- ROR Wd, Wn, Wm
  Rotate Right (register): preferred alias for RORV Wd, Wn, Wm.
- ROR Xd, Xn, Xm
  Rotate Right (extended register): preferred alias for RORV Xd, Xn, Xm.

5.4.5 位运算

- CLS Wd, Wm
  Count Leading Sign Bits: 将 Wd 设置为 Wm 中最高位之后的连续位数，与最高位相同。 该计数不包括最高位本身，因此结果将在 0 到 31 的范围内。
- CLS Xd, Xm
  Count Leading Sign Bits (extended): 将 Xd 设置为 Xm 中最高位之后的连续位数，与最高位相同。 该计数不包括最高位本身，因此结果将在 0 到 63 的范围内。
- CLZ Wd, Wm
  Count Leading Zeros: 将 Wd 设置为 Wm 最高有效端的二进制零的数量。 结果将在 0 到 32 的范围内。
- CLZ Xd, Xm
  Count Leading Zeros: （扩展）将 Xd 设置为 Xm 最高有效端的二进制零的数量。 结果将在 0 到 64 的范围内。
- RBIT Wd, Wm
  Reverse Bits: reverses the 32 bits from Wm, writing to Wd.
- RBIT Xd, Xm
  Reverse Bits (extended): reverses the 64 bits from Xm, writing to Xd.
- REV Wd, Wm
  Reverse Bytes: reverses the 4 bytes in Wm, writing to Wd.
- REV Xd, Xm
  Reverse Bytes (extended): reverses 8 bytes in Xm, writing to Xd.
- REV16 Wd, Wm
  Reverse Bytes in Halfwords: reverses the 2 bytes in each 16-bit element of Wm, writing to Wd.
- REV16 Xd, Xm
  Reverse Bytes in Halfwords (extended): reverses the 2 bytes in each 16-bit element of Xm, writing to Xd.

- REV32 Xd, Xm
  Reverse Bytes in Words (extended): reverses the 4 bytes in each 32-bit element of Xm, writing to Xd

5.4.6 条件数据处理

这些指令支持两个未移位的源寄存器，条件标志作为第三个源。 请注意，指令不是有条件地执行的：始终写入目标寄存器。

- ADC Wd, Wn, Wm
  Add with Carry: Wd = Wn + Wm + C.
- ADC Xd, Xn, Xm
  Add with Carry (extended): Xd = Xn + Xm + C
- ADCS Wd, Wn, Wm
  Add with Carry and Set Flags: Wd = Wn + Wm + C, setting the condition flags.
- ADCS Xd, Xn, Xm
  Add with Carry and Set Flags (extended): Xd = Xn + Xm + C, setting the condition flags.
- CSEL Wd, Wn, Wm, cond
  Conditional Select: Wd = if cond then Wn else Wm.
- CSEL Xd, Xn, Xm, cond
  Conditional Select (extended): Xd = if cond then Xn else Xm.
- CSINC Wd, Wn, Wm, cond
  Conditional Select Increment: Wd = if cond then Wn else Wm+1.
- CSINC Xd, Xn, Xm, cond
  Conditional Select Increment (extended): Xd = if cond then Xn else Xm+1.
- CSINV Wd, Wn, Wm, cond
  Conditional Select Invert: Wd = if cond then Wn else NOT(Wm).
- CSINV Xd, Xn, Xm, cond
  Conditional Select Invert (extended): Xd = if cond then Xn else NOT(Xm).
- CSNEG Wd, Wn, Wm, cond
  Conditional Select Negate: Wd = if cond then Wn else -Wm.
- CSNEG Xd, Xn, Xm, cond
  Conditional Select Negate (extended): Xd = if cond then Xn else -Xm.
- CSET Wd, cond
  Conditional Set: Wd = if cond then 1 else 0. Alias for CSINC Wd,WZR,WZR,invert(cond).
- CSET Xd, cond
  Conditional Set (extended): Xd = if cond then 1 else 0. Alias for CSINC Xd,XZR,XZR,invert(cond)
- CSETM Wd, cond
  Conditional Set Mask: Wd = if cond then -1 else 0. Alias for CSINV Wd,WZR,WZR,invert(cond).
- CSETM Xd, cond
  Conditional Set Mask (extended): Xd = if cond then -1 else 0. Alias for CSINV Xd,WZR,WZR,invert(cond).
- CINC Wd, Wn, cond
  Conditional Increment: Wd = if cond then Wn+1 else Wn. Alias for CSINC Wd,Wn,Wn,invert(cond).

- CINC Xd, Xn, cond
  Conditional Increment (extended): Xd = if cond then Xn+1 else Xn. Alias for CSINC Xd,Xn,Xn,invert(cond).
- CINV Wd, Wn, cond
  Conditional Invert: Wd = if cond then NOT(Wn) else Wn. Alias for CSINV Wd,Wn,Wn,invert(cond).
- CINV Xd, Xn, cond
  Conditional Invert (extended): Xd = if cond then NOT(Xn) else Xn. Alias for CSINV Xd,Xn,Xn,invert(cond).
- CNEG Wd, Wn, cond
  Conditional Negate: Wd = if cond then -Wn else Wn. Alias for CSNEG Wd,Wn,Wn,invert(cond).
- CNEG Xd, Xn, cond
  Conditional Negate (extended): Xd = if cond then -Xn else Xn. Alias for CSNEG Xd,Xn,Xn,invert(cond).
- SBC Wd, Wn, Wm
  Subtract with Carry: Wd = Wn - Wm - 1 + C.
- SBC Xd, Xn, Xm
  Subtract with Carry (extended): Xd = Xn - Xm - 1 + C.
- SBCS Wd, Wn, Wm
  Subtract with Carry and Set Flags: Wd = Wn - Wm - 1 + C , setting the condition flags.
- SBCS Xd, Xn, Xm
  Subtract with Carry and Set Flags (extended): Xd = Xn - Xm - 1 + C , setting the condition flags.
- NGC Wd, Wm
  Negate with Carry: Wd = -Wm - 1 + C. Alias for SBC Wd,WZR,Wm.
- NGC Xd, Xm
  Negate with Carry (extended): Xd = -Xm - 1 + C. Alias for SBC Xd,XZR,Xm.
- NGCS Wd, Wm
  Negate with Carry and Set Flags: Wd = -Wm - 1 + C, setting the condition flags. Alias for SBCS Wd,WZR,Wm.
- NGCS Xd, Xm
  Negate with Carry and Set Flags (extended): Xd = -Xm - 1 + C, setting the condition flags. Alias for SBCS Xd,XZR,Xm.

5.4.7 条件比较

条件比较为 NZCV 条件标志提供“条件选择”，如果输入条件为真，则将标志设置为比较结果，如果输入条件为假，则将标志设置为立即值。 有寄存器和立即数形式，立即数形式接受一个小的 5 位无符号值。

#uimm4 操作数是用于在输入条件为假时设置 NZCV 标志的位掩码，第 3 位是 N 标志的新值，第 2 位是 Z 标志，第 1 位是 C 标志，第 0 位是 V 标志

- CCMN Wn, Wm, #uimm4, cond
  Conditional Compare Negative (register):
  NZCV = if cond then CMP(Wn,-Wm) else uimm4.
- CCMN Xn, Xm, #uimm4, cond
  Conditional Compare Negative (extended register):

NZCV = if cond then CMP(Xn,-Xm) else uimm4.

- CCMN Wn, #uimm5, #uimm4, cond
  Conditional Compare Negative (immediate):
  NZCV = if cond then CMP(Wn,-uimm5) else uimm4.
- CCMN Xn, #uimm5, #uimm4, cond
  Conditional Compare Negative (extended immediate):
  NZCV = if cond then CMP(Xn,-uimm5) else uimm4.
- CCMP Wn, Wm, #uimm4, cond
  Conditional Compare (register):
  NZCV = if cond then CMP(Wn,Wm) else uimm4.
- CCMP Xn, Xm, #uimm4, cond
  Conditional Compare (extended register):
  NZCV = if cond then CMP(Xn,Xm) else uimm4.
- CCMP Wn, #uimm5, #uimm4, cond
  Conditional Compare (immediate):
  NZCV = if cond then CMP(Wn,uimm5) else uimm4.
- CCMP Xn, #uimm5, #uimm4, cond
  Conditional Compare (extended immediate):
  NZCV = if cond then CMP(Xn,uimm5) else uimm4

5.5 整数乘法/除法

5.5.1 乘法

- MADD Wd, Wn, Wm, Wa
  Multiply-Add: Wd = Wa + (Wn × Wm).
- MADD Xd, Xn, Xm, Xa
  Multiply-Add (extended): Xd = Xa + (Xn × Xm.)
- MSUB Wd, Wn, Wm, Wa
  Multiply-Subtract: Wd = Wa – (Wn × Wm).
- MSUB Xd, Xn, Xm, Xa
  Multiply-Subtract (extended): Xd = Xa – (Xn × Xm).
- MNEG Wd, Wn, Wm
  Multiply-Negate: Wd = –(Wn × Wm). Alias for MSUB Wd, Wn, Wm, WZR.
- MNEG Xd, Xn, Xm
  Multiply-Negate (extended): Xd = –(Xn × Xm). Alias for MSUB Xd, Xn, Xm, XZR.
- MUL Wd, Wn, Wm
  Multiply: Wd = Wn × Wm. Alias for MADD Wd, Wn, Wm, WZR.
- MUL Xd, Xn, Xm
  Multiply (extended): Xd = Xn × Xm. Alias for MADD Xd, Xn, Xm, XZR.
- SMADDL Xd, Wn, Wm, Xa
  Signed Multiply-Add Long: Xd = Xa + (Wn × Wm), treating source operands as signed.
- SMSUBL Xd, Wn, Wm, Xa
  Signed Multiply-Subtract Long: Xd = Xa – (Wn × Wm), treating source operands as signed.
- SMNEGL Xd, Wn, Wm
  Signed Multiply-Negate Long: Xd = -(Wn × Wm), treating source operands as signed. Alias for

SMSUBL Xd, Wn, Wm, XZR.

- SMULL Xd, Wn, Wm
  Signed Multiply Long: Xd = Wn × Wm, treating source operands as signed. Alias for SMADDL Xd, Wn, Wm, XZR.
- SMULH Xd, Xn, Xm
  Signed Multiply High: Xd = (Xn × Xm)127:64, treating source operands as signed.
- UMADDL Xd, Wn, Wm, Xa
  Unsigned Multiply-Add Long: Xd = Xa + (Wn × Wm), treating source operands as unsigned.
- UMSUBL Xd, Wn, Wm, Xa
  Unsigned Multiply-Subtract Long: Xd = Xa – (Wn × Wm), treating source operands as unsigned.
- UMNEGL Xd, Wn, Wm
  Unsigned Multiply-Negate Long: Xd = -(Wn × Wm), treating source operands as unsigned. Alias for UMSUBL Xd, Wn, Wm, XZR.
- UMULL Xd, Wn, Wm
  Unsigned Multiply Long: Xd = Wn × Wm, treating source operands as unsigned. Alias for UMADDL Xd, Wn, Wm, XZR.
- UMULH Xd, Xn, Xm
  Unsigned Multiply High: Xd = (Xn × Xm)127:64, treating source operands as unsigned

5.5.2 除法

整数除法指令计算 (分子÷分母) 并提供商，该商向零舍入。 然后可以使用 MSUB 将余数计算为分子–（商*分母）
操作说明。

如果执行有符号整数除法 (INT_MIN ÷ -1)，其中 INT_MIN 是所选寄存器大小中可表示的最大负整数值，则结果将溢出有符号整数范围。 不会产生此溢出的指示，写入目标寄存器的结果将为 INT_MIN。

注意：除法指令不会在除以零时生成陷阱，而是将零写入目标寄存器。

- SDIV Wd, Wn, Wm
  Signed Divide: Wd = Wn ÷ Wm, treating source operands as signed.
- SDIV Xd, Xn, Xm
  Signed Divide (extended): Xd = Xn ÷ Xm, treating source operands as signed.
- UDIV Wd, Wn, Wm
  Unsigned Divide: Wd = Wn ÷ Wm, treating source operands as unsigned.
- UDIV Xd, Xn, Xm
  Unsigned Divide (extended): Xd = Xn ÷ Xm, treating source operands as unsigned.

5.6 标量浮点

5.6.1 浮点/SIMD 标量内存访问

5.7.17 标量浮点/整数转换

5.7.18 向量缩减（跨车道）

5.7.19 向量成对算术

5.7.20 标量归约（成对）

5.7.21 向量表查找

5.7.22 向量加载存储结构

5.7.23 AArch32 等效高级 SIMD 助记符

5.7.24 加密扩展

5.8 系统说明

系统指令分为以下：

- 异常生成指令
- 系统寄存器访问
- 系统管理
- hints
- Barriers和CLREX

5.8.1 异常的产生和返回

5.8.1.1 Non-debug exceptions

- SVC #uimm16
  生成针对异常级别 1（系统）的异常，在 uimm16 中具有 16 位有效负载。
- HVC #uimm16
  生成针对异常级别 2（管理程序）的异常，在 uimm16 中具有 16 位有效负载。
- SMC #uimm16
  生成针对异常级别 3（安全监视器）的异常，在 uimm16 中具有 16 位有效负载。
- ERET
  异常返回：从当前异常级别的 SPSR_ELn 寄存器重构处理器状态，并跳转到 ELR_ELn 中的地址。

5.8.1.2 Debug exceptions

- BRK #uimm16
  监视模式软件断点：异常路由到在 EL1 或 EL2 中执行的调试监视器，在 uimm16 中具有 16 位有效负载。
- HLT #uimm16
  暂停模式软件断点：如果启用则进入暂停模式调试状态，否则视为未分配。 在 uimm16 中具有 16 位有效负载。
- DCPS1 {#uimm16}
  调试 将处理器状态更改为 EL1（仅在暂停模式调试状态下有效），可选的 16 位立即数 uimm16 默认为零并被硬件忽略。
- DCPS2 {#uimm16}
  Debug 将处理器状态更改为 EL2（仅在暂停模式调试状态下有效），可选的 16 位立即数 uimm16 默认为零并被硬件忽略。

- DCPS3 {#uimm16}
  Debug 将处理器状态更改为 EL3（仅在暂停模式调试状态下有效），可选的 16 位立即数 uimm16 默认为零并被硬件忽略。
- DRPS
  Debug Restore Processor State：将处理器恢复到当前异常级别的SPSR_ELn寄存器中记录的异常级别和模式（仅在暂停模式调试状态下有效）

5.8.2 系统寄存器访问

- MRS Xt, <system_register>
  将 <system_register> 移动到 Xt，其中 <system_register> 是系统寄存器名称，或者对于实现定义的寄存器，使用“S”形式的名称，例如“S3_4_c13_c9_7”。
- MSR <system_register>, Xt
  将 Xt 移动到 <system_register>，其中 <system_register> 是系统寄存器名称，或者对于实现定义的寄存器，使用“S”形式的名称，例如“S3_4_c13_c9_7”..
- MSR DAIFClr, #uimm4
  使用 uimm4 作为位掩码来选择清除一个或多个 DAIF 异常掩码位：位 3 选择 D 掩码，位 2 选择 A 掩码，位 1 选择 I 掩码，位 0 选择 F 掩码。
- MSR DAIFSet, #uimm4
  使用 uimm4 作为位掩码来选择一个或多个 DAIF 异常掩码位的设置：位 3 选择 D 掩码，位 2 选择 A 掩码，位 1 选择 I 掩码，位 0 选择 F 掩码。
- MSR SPSel, #uimm4
  使用 uimm4 作为控制值来选择堆栈指针：如果设置位 0，则选择当前异常级别的堆栈指针，如果位 0 清零，则选择共享 EL0 堆栈指针。 uimm4 的位 1 到 3 被保留并且应该为零。

5.8.3 系统管理

如果 SYS 指令的操作数与下面 <xx_op> 表中的条目匹配，则关联的别名是首选反汇编。 否则应使用 SYS 或 SYSL 助记符，允许生成和反汇编任意实现定义的系统指令

- SYS #op1, Cn, Cm, #op2{, Xt}
- SYSL Xt, #op1, Cn, Cm, #op2
- IC <ic_op>{, Xt} 指令cache维护操作指令
- DC <dc_op>, Xt 数据cache维护操作指令
- AT <at_op>, Xt 地址翻译指令
- TLBI <tlbi_op>{, Xt} TLB维护操作指令

5.8.4 hints

- NOP
- YIELD
- WFE
- WFI
- SEV
- SEVL
- HINT #uimm7

5.8.5 Barriers和CLREX

- CLREX {#uimm4}
- DSB
  |#uimm4
- DMB
  |#uimm4
- ISB {SY|#uimm4}

6 A32和T32指令集

6.1 部分弃用 的之类

6.2 加载-获取/存储-释放

6.2.1 非排他性

6.2.2 独占

6.3 VFP 标量浮点

6.3.1 浮点条件选择

6.3.2 浮点minNum/maxNum

6.3.3 浮点转换（浮点到整数）

6.3.4 浮点转换（半精度到/从双精度）

6.3.5 浮点取整

6.4 高级 SIMD 浮点

6.4.1 浮点minNum/maxNum

6.4.2 浮点转换

6.4.3 浮点取整到积分

6.5 加密扩展

6.6 系统说明

6.6.1 停止调试

6.6.2 Barriers 和 Hints