《ARM 汇编技术》

一、 ARM 的基本概念

1. 什么是 ARM

ARM 是一家英国电子公司的名字，全名是 Advanced RISC Machine，这家企业设计了大量高性能、廉价、耗能低的 RISC（精简指令集）处理器，ARM 公司只设计芯片而不生产，它将技术授权给世界上许多公司和厂商。目前采用 ARM 技术知识产权内核的微处理器，即通常所说的 ARM 微处理器，所以 ARM 也是对一类微处理器的通称。

ARM 指令集体系版本号 (软件) 为 V1 ~ V7 目前 V1 ~ V3 已很少见。从 V4 版不再与以前的版本兼容。

ARM 的 CPU 系列（硬件）主要有 ARM7 ~ ARM11。

2. 典型的嵌入式处理器

ARM	占市场 79.5%	ARM
Mips	占市场 13.9%	MIPS
microSPARC	占市场 3.1%	SUN
PowerPc	占市场 2.8%	IBM
其它	占市场 0.8%	

3. ARM 的应用范围：

工业控制：如机床、自动控制等

无线通信：如手机

网络应用：如

电子产品：如音视频播放器、机顶盒、游戏机、数码相机、打印机

其它各领域：如军事、医疗、机器人、智能家居等

4. 计算机体系结构

冯.诺依曼体系结构：处理器使用同一个存储器，经由同一个总线传输；

完成一条指令需要 3 个步骤：即取指令->指令译码->执行指令

指令和数据共享同一总线的结构

哈佛体系结构：将程序指令存储和数据存储分开，中央处理器首先到程序指令存储器中读取程序指令。解码后到数据地址，再到相应的数据存储器读取数据，然后执行指令；

程序指令存储与数据存储分开，可以使指令和数据有不同的数据宽度。

5. 复杂指令集与精简指令集

CISC 复杂指令集：采用冯.诺依曼体系结构。数据线和指令线分时复用（只能通过一辆车）。存储器操作指令多 汇编程序相对简单 指令结束后响应中断 CPU 电路丰富 面积大功耗大。

RISC 精简指令集：采用哈佛体系结构。数据线和指令线分离（同时能通过多辆车）。对存储器操作有限 汇编程序占空间大在适当地方响应中断 CPU 电路较少，体积小、功耗低。

ARM 采用 RISC 精简指令集

Thumb 是 ARM 体系结构中一种 16 位的指令集。从 ARMv4T 之后的 ARM 处理器有一种 16-bit 指令模式，叫做 Thumb，较短的指令码提供整体更佳的编码密度，更有效地使用有限的内存带宽。所有 ARM9 和后来的家族，包括 XScale 都纳入了 Thumb 技术。

即 ARM 有两种指令集：RISC、Thumb

6. ARM 的思想

1) ARM 体系的总体思想

在不牺牲性能的同时，尽量简化处理器。同时从体系结构上灵活支持处理器扩展。采用 RISC 结构，RISC 处理器简化了处理器结构，减少复杂功能指令的同时，提高了处理器速度。ARM 及 MIPS 都是典型的 RISC 处理器。

2) ARM 的流水线结构

ARM 处理器使用流水线来增加处理器指令流的速度，这样可以使几个操作同时进行。并使处理和存储器系统连续操作。

3) ARM 处理器分为三级

取指：指令从存储器中取出；

译码：对指令使用的寄存器进行译码；

执行：从寄存器组中读取寄存器，执行移位和 ALU 操作，寄存器被写回到寄存器组中。

4) ARM 处理器支持的类型

字节 8 位

半字 16 位

字 32 位

@所有数据操作都以字为单位

@ARM 指令的长度刚好是一个字，Thumb 指令长度刚好是半个字

5) ARM 处理器状态

ARM 处理器内核使用 ARM 结构，该结构包含 32 位的 ARM 指令集和 16 位 Thumb 指令集，因此 ARM 有两种操作状态

ARM 状态：32 位

Thumb 状态：16 位

6) 处理器模式

ARM 处理器共有 7 种运行模式：

用户：正常程序工作模式，不能直接切换到其它模式

系统：用于支持操作系统的特权任务，可以直接切换到其它模式

快中断：支持高速数据传输及通道处理，FIQ 异常响应时进入此模式

中断：用于通用中断处理，IRQ 异常响应时进入此模式

管理：操作系统保护代码，系统复位和软件中断响应时进入此模式

中止：用于支持虚拟内存或存储器保护，用于 MMU

未定义：支持硬件协处理器的软件仿真，未定义指令异常响应时进入此模式。

二、 ARMv8 的体系结构

1. ARMv8-A 的系统寄存器

系统寄存器（System Registers）是实现对处理器监控的手段，负责对处理单元的控制，并返回其状态信息。系统寄存器保存了对 ARMv8-A 架构处理单元的配置信息，系统寄存器保存的设置状态组合定义了当前处理器的程序状态上下文。

系统寄存器共有以下几类：

- 通用系统控制寄存器；
- 调试寄存器；
- 通用定时寄存器；
- 性能监视寄存器，可选；
- 活动监视器寄存器（Activity Monitors Registers），可选；
- 跟踪系统寄存器（Trace System Registers），可选；
- 可伸缩向量扩展系统寄存器（Scalable Vector Extension System Registers），可选；
- 通用中断控制器（Generic Interrupt Controller，GIC）系统寄存器，可选，定义在 ARM 通用中断控制器架构规范中；
- RAS扩展系统寄存器，RAS扩展功能对ARMv8.2架构而言是必备的功能，但对ARMv8.0架构和ARMv8.1 架构是可选的。

系统寄存器使用标准的命名规则来界定特定寄存器及其中的控制与状态位：

`<寄存器名>.<字段名>`

也可以使用控制与状态位在寄存器中的数字序号标识位，格式为：

`<寄存器名>[x:y]`

系统寄存器的访问受到当前异常等级的限制。为了使用时更直观，AArch64 状态下的大多数寄存器名中都包含一个后缀，指示该寄存器可以被访问的最低异常等级，其命名格式为：

`<寄存器名>_Elx`

其中，x 为 0、1、2 或 3。

例如，在存储管理单元的地址变换过程中需要用到地址变换表（Translation Table）。ARM采用页式虚拟存储器机制，故该地址变换表也即一般所说的页表。在ARMv8-A架构中，EL0和EL1异常等级使用的地址变换表的基地址保存在**变换表基址寄存器**（Translation Table Base Register）**TTBR0_EL1**中。这一寄存器只能在特权模式下访问，因而其可以被访问的最低异常等级为EL1。在EL0异常等级下访问TTBR0_EL1寄存器将产生一个异常，系统中也不存在TTBR0_EL0寄存器。

由此可以很容易理解，ARMv8-A架构下存在许多具有相似功能的寄存器，其寄存器名只有异常等级后缀不同。但是这些功能相同的寄存器是相互独立的，而且通过不同的硬件实现，在指令格式中也有不同 的寄存器编码。

例如，SCTLR_EL1、SCTLR_EL2 和 SCTLR_EL3这三个寄存器都是实现存储管理单元配置所用的寄存器，相似的名称意味着其功能相近，但三个寄存器完全独立，且各自有其访问方式：SCTLR_EL1 用于 EL0 和 EL1异常等级，SCTLR_EL2用于EL2异常等级，SCTLR_EL3则用于EL3异常等级。与TTBR0_EL1寄存器类似，EL1和EL0两个异常等级

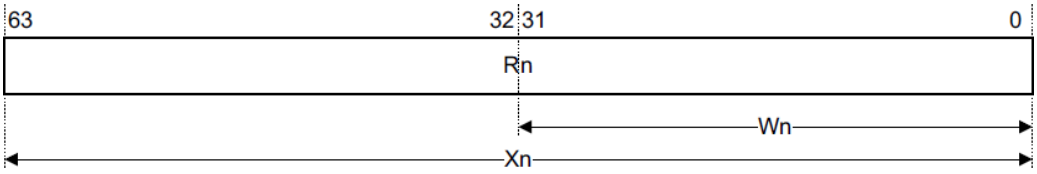
共享同一个存储管理单元配置寄存器，而且只有在特权等级EL1才能实现相关配置操作。其他控制寄存器也遵从这一规则。大部分系统寄存器不允许在EL0级访问。

但是，在高异常等级运行的程序有权限访问低异常等级的寄存器。如果有必要，可以在 EL2 异常等级访问 SCTLR_EL1 寄存器。虽然一般情况下特权异常等级只管理自身级别的配置，但在需要时，在高特权等级运行的程序有时会需要访问更低异常等级的寄存器。例如，在实现虚拟化功能时，或者在处理器上 下文切换或功耗管理操作中需要实现保存并恢复（Save-and-Restore）操作中的寄存器组读写操作时。

注意系统寄存器不能直接在数据处理类指令或加载/存储（Load/Store）指令中作为操作数使用。对系 统寄存器的操作需要通过通用寄存器进行，也即首先通过访问特殊功能寄存器的指令 MRS（从特殊寄存器读数据至通用寄存器）将系统寄存器的内容读到通用寄存器，操作完成后再通过 MSR 指令（从通用寄存器写数据至特殊寄存器）写回系统寄存器。

1) AArch64 状态下的通用寄存器

除了用于系统控制和状态报告的系统寄存器之外，AArch64 状态下的计算操作和异常处理也会用到一些与指令处理有关的寄存器。



AArch64 架构下的通用寄存器命名

通用寄存器 R0~R30

从编译器程序员和汇编语言程序员的角度看，A64指令集的明显特征是通用寄存器的数量增加了，其效果是提升了系统性能并可能减少堆栈的使用。31 个 64 位的通用寄存器在汇编语言中被标识为 X0~X30。由于 X30 被当作过程链接寄存器（Procedure Link Register，PLR），从严格意义上说，过程链接寄存器并不属于通用寄存器，因而也可以说 A64 指令集使用 30 个通用寄存器。

实际上，在AArch64状态下的应用程序可用使用R0~R30共31个通用寄存器，而每个通用寄存器都可以通过64位和32位两种方式访问：当进行64位访问时，可以使用的通用寄存器名为 X0~X30；而当进行32位访问时，可以使用的通用寄存器名为 W0~W30。

从上图所示的寄存器映射关系中可以看出，32位的 Wn 寄存器就是 64 位的 Xn 寄存器的低有效32位.执行Wn寄存器读出操作时将丢弃相应 Xn 寄存器的高 32 位数据;而执行写 Wn 寄存器操作时将会把相应 Xn 寄存器的高 32 位清零。例如，向 W0 寄存器写0xFFFFFFFF 后，X0 寄存器将保存 0x00000000FFFFFFFF。

通用寄存器在任何时刻和任何异常等级下都能被访问。

SIMD 和浮点寄存器 V0~V31

在 AArch64 执行状态下，处理单元有 32 个 SIMD 和浮点寄存器，命名为 V0~V31。这组寄存器独立于通用寄存器，专门用于浮点运算和向量操作。寄存器名中的字母 V 代表向量（Vector），故这组寄存器有时也称为 V 寄存器。

每个寄存器都是 128 位宽的，但是应用程序可以通过多种方式访问其中的每个寄存器：

通过寄存器名 Q0~Q31 访问 128 位寄存器；

通过寄存器名 D0~D31 访问 64 位寄存器；

通过寄存器名 S0~S31 访问 32 位寄存器；

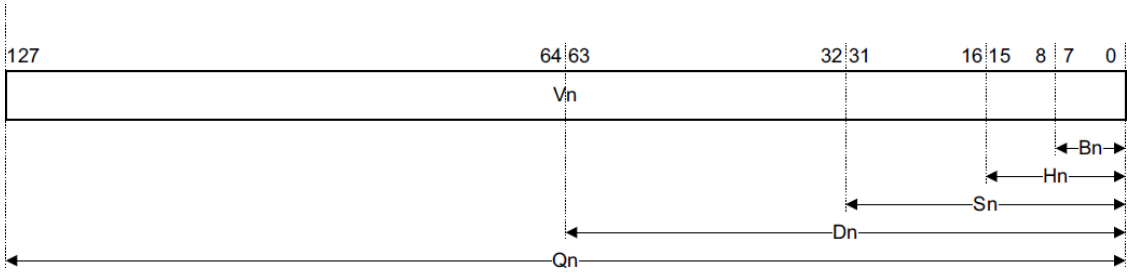
通过寄存器名 H0~H31 访问 16 位寄存器；

通过寄存器名 B0~B31 访问 8 位寄存器；

当作128 位向量的元素访问；

当作64 位向量的元素访问。

当作向量访问时，这组 V 寄存器保存的数据可以被当作一个向量，某个 V 寄存器保存了一组数据中的一个元素。与通用寄存器名 W0~W30 类似，通过寄存器名访问的 SIMD 和浮点寄存器不一定占据完整寄存器的所有位，而是使用其低有效位，如图 2.3 所示。



AArch64 架构下的 SIMD 和浮点寄存器命名

AArch64 执行状态下的处理状态 PSTATE

在 AArch32 执行状态下，当前处理器状态保存在当前程序状态寄存器 CPSR（Current Program Status Register）中，而 AArch64 执行状态的处理状态则保存在名为 PSTATE 的数据结构中。

严格来说，PSTATE 并不是单一的寄存器，而是对处理器状态信息的抽象。处理信息被分类映射到多个寄存器中，可以通过指令访问这些状态信息。因此，PSTATE 可以被看作是保存当前处理状态信息的一组抽象寄存器或者一组标志信息的统称。也正是因为处理状态不是像 ARMv7 架构那样存储在单一寄存器内，因而对一个状态寄存器中的各个字段的读、写或修改不再必须通过原子操作才能实现。

在 EL0 异常等级，可以访问的 PSTATE 信息包括条件标志（Condition Flags）和异常屏蔽位（Exception Masking Bits）。

条件标志可以通过标志设置指令置位，含四个标志位：

- 负条件标志（Negative Condition Flag）N。当指令的执行结果用补码带符号整数（Two's Complement Signed Integer）表示时：若结果为负数，则处理单元设置 N 标志为 1；若结果为正数或零，则处理单元设置 N 标志为 0。
- 零条件标志（Zero Condition Flag）Z。若指令的执行结果为零，则处理单元设置 Z 标志为 1（通常意味着比较的结果为相等）；否则，处理单元设置 Z 标志为 0。
- 进位条件标志（Carry Condition Flag）C。若指令的执行结果有进位（例如加法操作产生无符号数溢出），则处理单元设置 C 标志为 1；否则，处理单元设置 C 标志为 0。
- 溢出条件标志（Overflow Condition Flag）V。若指令的执行结果有溢出（例如加法操作产生带符号数溢出），则处理单元设置 V 标志为 1；否则，处理单元设置 C 标志为 0。

异常屏蔽位被设置为 1 则屏蔽异常，被设置为 0 则允许进行异常处理。PSTATE 状态中共包含四个异常屏蔽位：

- 调试异常屏蔽位（Debug Exception Mask Bit）D。调试异常通常由软件断点指令及断点、观察点、向量捕获、软件单步运行等因素引起。
- 系统错误中断屏蔽位（SError interrupt Mask Bit）A。
- IRQ 中断屏蔽位（IRQ Interrupt Mask Bit）I。
- 快速中断屏蔽位（FIQ Interrupt Mask Bit）F。

A64 指令集提供了访问特殊功能寄存器的指令，AArch64 执行状态下运行于 EL0 异常等级的程序可以通过

特殊功能寄存器读写 N、Z、C、V 和 D、A、I、F 等标志。但在 AArch64 状态下运行于 EL0 异常等级的程序访问 D、A、I 和 F 这四个状态的权限是受限的，取决于 SCTLR_EL1 寄存器的设置。

除了 EL0 等级运行的应用程序，操作系统和其他运行在特权级的系统软件需要通过 PSTATE 为应用程序提供支持和服务。因此，AArch64 架构的系统级程序员的编程模型更复杂，允许操作系统能够为应用程序分配系统资源，并为其他进程和操作系统本身提供保护机制。在 PSTATE 结构中包含了一些反映程序执行状态的控制字段。

PSTATE 执行状态控制（Execution State Controls）类状态位包含以下字段：

- 软件单步运行（Software Step）位 SS。
- 非法执行状态（Illegal Execution State）位 IL：非法异常产生时会置位这个异常执行状态标志。
- 当前执行状态（Current Execution State）位 nRW：表示当前 ELx 所运行的执行架构状态，也即 AArch64 或 AArch32。若当前执行状态为 AArch64，则该位为 0。在系统被复位或者发生异常后进入使用 AArch64 执行状态的异常等级时，该位将被清零。
- 当前异常等级（Current Exception Level）字段 EL：ARM 架构要求处理单元在复位后首先进入实现的最高异常等级，故处理器在复位并进入 AArch64 状态时，在 EL 字段保存了该最高异常等级的编码。
- 堆栈指针寄存器选择（Stack Pointer Register Selection）位 SP。该位为 0 意味着选择 SP_EL0，该位为 1 则意味着选择 SP_Eln。在因复位或者相应异常进入 AArch64 状态时，该位被置 1，意味着选择 SP_Elx。

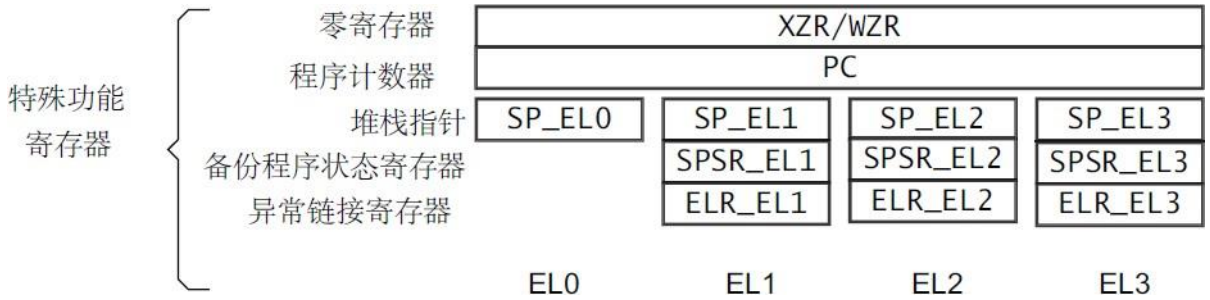
此外，PSTATE 中还包括了一些与 ARMv8 架构可选的扩展功能有关的控制位，如访问控制位（Access Control Bits）、定时控制位（Timing Control Bits）和推测控制位（Speculation Control Bits，或译“投机控制位”）等。

通过 MSR 和 MRS 等访问特殊功能寄存器的指令，AArch64 执行状态下的程序可以直接读写 PSTATE 的各个字段。所有访问 N、Z、C、V 和 D、A、I、F 标志之外的 PSTATE 字段的指令都可以在 EL1 或者更高异常等级执行。

AArch64 执行状态下的特殊功能寄存器

除了通用寄存器之外，AArch64 状态还设置了若干特殊功能寄存器。下图给出了这些寄存器的概况。对传统 ARM 处理器架构比较熟悉的读者会清楚地记得，ARMv7 以前的架构在每个操作状态都设置了

若干备份寄存器（Banked Registers，或称分组寄存器）。这种寄存器组织方式在 ARMv8-A 的 AArch32 执行状态下仍然被保留，但在 AArch64 执行状态下的寄存器组织结构与此完全不同。如图 2.4 所示，只有堆栈指针寄存器 SP、异常链接寄存器 ELR 和备份程序状态寄存器（Saved Program Status Register，SPSR）是分组的，但不是按照操作状态分组，而是针对不同异常等级设置多个功能类似的寄存器。



AArch64 架构下的特殊功能寄存器

零寄存器 XZR 和 WZR

与许多RISC处理器的指令系统特殊设计类似，ARMv8-A 架构也保留了虚拟的零寄存器 ZR（Zero Register）。在许多指令字中，如果通用寄存器编码的位置出现 31 的编码（二进制 0b11111），则该操作数并

不代表 X31 通用寄存器或 W31 通用寄存器（X31 和 W31 并不存在），而是代表 64 位零寄存器 XZR 或 32 位零寄存器 WZR。零寄存器并不是物理存在的寄存器，只是代表操作数为立即数 0。因此，访问零寄存器时，所有写操作被忽略，所有读操作都返回全零。并非所有指令都能够使用零寄存器作为参数。

程序计数器 PC

存放当前指令的内存地址存放在 64 位的程序计数器 PC 中。在 ARMv7 架构中，程序计数器可以被当作通用寄存器 R15 使用，这虽然能给某些特定的程序提供灵活控制的能力，但是却使编译器和流水线设计复杂化。在 ARMv8-A 架构中，软件不能通过寄存器名直接访问程序计数器，PC 也不能作为加载指令或者数据处理指令的目标地址。只有执行分支类等少数几类指令、进入异常处理入口或者从异常退出返回时才 能间接修改该寄存器。

在 AArch64 执行状态下，指令存储必须是 32 位对齐的。如果在取指令时发现有非对齐的 PC 值，也即 PC 的最低两位[1:0]不是 0b00，将会产生 PC 对齐故障异常（PC Alignment Fault Exception）。该异常只有在实际执行非字对齐的 A64 指令时才会被触发。

堆栈指针寄存器 SP 和 WSP

AArch64 执行状态下的堆栈指针通过 64 位的专用堆栈指针寄存器 SP 保存，相应指令中寄存器编码为31（根据特定指令的功能和操作数的位置，该编码所代表的操作数或者是当前堆栈指针，或者是零寄存器）。与32 位架构不同，在 AArch64 执行状态下，SP 寄存器不是通用寄存器。

在 AArch64 执行状态下，每个异常等级都有其自身的堆栈指针，因而共有四个堆栈指针寄存器SP_EL0、SP_EL1、SP_EL2 和 and SP_EL3。而选择当前使用的堆栈指针也需根据当前的异常等级。缺省情况下，异常发生时将选择目标异常等级的堆栈指针。

例如，异常处理进入 EL1 等级时将选择 SP_EL1。

在 AArch64 执行状态的 EL0 异常等级只能访问 SP_EL0；在 AArch64 执行状态的 EL0 以上异常等级 n 执行异常处理时，处理单元可以选择该异常等级专用的堆栈指针 SP_Eln，也可以选择 EL0 等级的堆栈指针SP_EL0。

在指令中引用 SP 作为操作数可以访问当前的堆栈指针。该寄存器的低有效 32 位可以通过名为 WSP的寄存器访问。

当使用堆栈指针作为计算指令的基地址时，若堆栈指针的最低四位[3:0]不是 0b0000，则该指针将被认定为非对齐的堆栈指针（Misaligned Stack Pointer）。可以配置处理单元在实际执行包含非对齐的堆栈指针的指令时产生堆栈指针对齐故障异常（SP Alignment Fault Exception）。

异常链接寄存器 ELR

异常链接寄存器 ELR（Exception Link Register）保存异常返回地址。

AArch64 状态为异常处理需要转入的每一个异常等级设置了一个 ELR 寄存器。

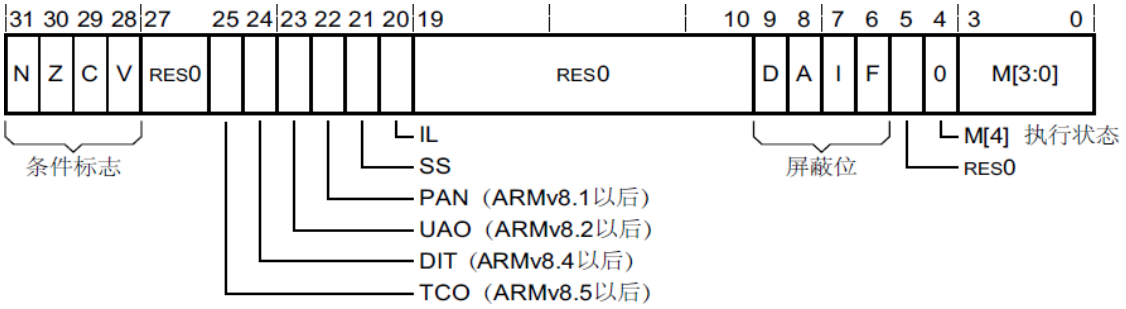
当处理单元发生异常时，异常返回地址将被保存到目标异常等级 n 的异常链接寄存器 ELR_ELn 中。例如，如果处理单元转入 EL1 处理异常，则异常返回地址将保存在 ELR_EL1 中。

异常返回时，软件执行 ERET 指令，程序计数器将被恢复为 ELR_ELn 中保存的断点地址。

备份程序状态寄存器 SPSR

与 ARMv7 的当前程序状态寄存器类似，在 AArch64 状态下异常发生时，当前的处理状态 PSTATE 被保存至备份程序状态寄存器 SPSR 中。SPSR 保存异常发生时的 PSTATE 值，用于在异常返回时恢复程序状态。

下图给出了 AArch64 状态下备份程序状态寄存器 SPSR 的结构。



AArch64 状态下备份程序状态寄存器 SPSR 的结构

N、Z、C、V 字段分别代表负条件标志、零条件标志、进位条件标志和溢出条件标志。

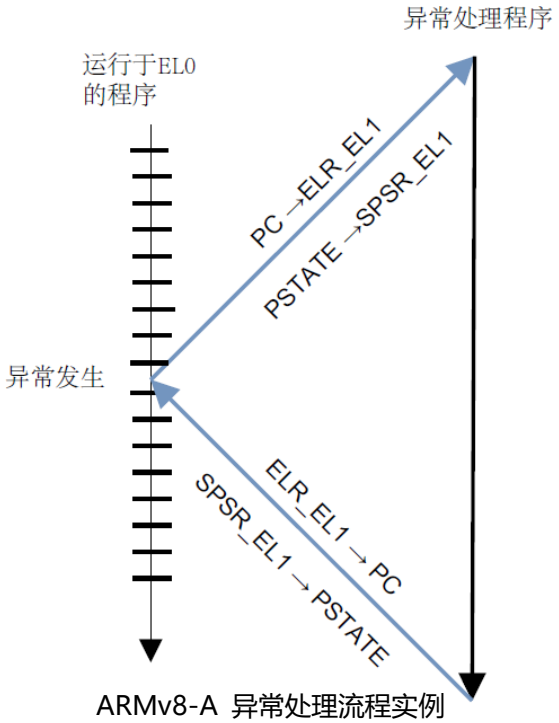
SS 和IL 分别代表软件单步运行标志和非法执行状态标志。

D、A、I、F 标志分别保存了异常发生瞬间的处理状态中调试异常、系统错误中断异常、IRQ 中断和 FIQ 中断的屏蔽位状态。

M[4]保存了 PSTATE.nRW 字段的值，代表异常发生时的执行状态。对 AArch64 状态而言，该值固定为 0。

M[3:0]代表异常发生时的异常等级。 在 ARMv8-A 架构中，异常发生时程序状态存入哪个备份程序状态寄存器SPSR同样依赖于异常等级。如果处理单元在EL1异常等级处理异常，则使用 SPSR_EL1；如果处理单元在 EL2 异常等级处理异常，则使用 SPSR_EL2；以此类推。但在EL0 异常等级不能处理异常。

右图给出了AArch64 架构下的异常处理时程序状态保护和恢复的流程实例。



ARMv8-A 异常处理流程实例

SPSR 寄存器的功能主要有两个。

其一是异常返回时恢复处理单元的状态。当异常返回时，异常处理程序将把处理单元的状态恢复至与异常返回之前程序所处的异常等级相关联的 SPSR 寄存器中所保存的状态。例如，从 EL1 异常返回时，将会从 SPSR_EL1 中保存的状态信息中恢复处理单元的状态。

其二是能让异常处理程序检查异常发生时的PSTATE 值，比如判断引起异常的指令执行时的执行状态和异常等级。

2) 流水线对 PC 值的影响

SPSR 保存的程序状态寄存器，结构与 CPSR 完全一样，用来保存 CPSR 的值。以便出现异常时恢复 CPSR 的值。CPU 内部的组成分为：指令寄存器，指令译码器，指令执行单元（包括 ALU 和通用寄存器组）CPU 执行指令的步骤：取指->译码->执行

取指：将指令从内存或指令 cache 中取入指令寄存器

译码: 指令译码器对指令寄存器中的指令进行译码操作，辨识 add，或是 sub 等操作

执行：指令执行单元根据译码的结果进行运算并保存结果

流水线操作：并发多条流水线（以 3 条为例）

1-取指 译码 执行

2-取指 译码 执行

3-取指　译码　执行

2. ARMv8-A 架构的异常与中断

1) ARMv8-A 的异常类型

异常（Exceptions）是现代处理器必备的程序随机切换机制。最常见的异常即是由外部事件引起的中 断服务过程。在复杂系统中，异常也用于处理需要特权软件权限才能处理的系统事件。每一个异常类型都 有其异常处理程序。

广义的异常包括中断和狭义的异常两类。中断是一种特定类型的异常，典型的中断是由处理器之外的 硬件触发的。而造成正常的指令执行顺序被改变的事件则会引起狭义的异常。在 AArch64 状态下，广义的异常被分为两种类型：**同步异常**（Synchronous Exception）和**异步异常**（Asynchronous Exception）。

如果异常直接由执行指令或者尝试执行指令引起，并且异常返回地址指明了引起异常的特定指令的细 节，则该异常被定义为同步异常。否则，则称为异步异常。在 ARM 语境中，把并非直接由程序执行而引起的异常定义为中断（Interrupt）。异步异常是由 IRQ、FIQ 这两个中断请求引脚引起的中断以及系统错误引起的异常，相应地被分为三类：IRQ、FIQ 和 SError（System Error，系统错误）。

在 AArch64 状态下，有下面几类事件会引起异常。

终止（Aborts)

当指令取指错误时会产生指令终止(Instruction Aborts)，而数据访问错误则会引起数据终止(Data Aborts)。终止可以是因存储器访问失败引起，也可能是因存储管理单元（MMU）需要通过终止异常为应 用程序动态分配内存引起。存储管理单元引起的访问权限故障、访问属性故障等异常、堆栈指针和程序计数器对齐检查、无效指 令和系统调用等都将产生同步终止异常。异步数据终止是引起系统错误异常的最常见原因，当从 Cache 向外部的存储器写回“脏”数据时即会触发这类终止。在 AArch64 状态下，同步终止引起同步异常，而异步终止将产生系统错误异步异常。

复位（Reset)

复位异常是最高等级的异常，并且不能被屏蔽。所有处理单元在系统复位之后总是转至最高异常等级 执行复位异常，并初始化系统。

执行异常产生指令

异常产生指令（Exception Generating Instructions）就是一般所说的系统调用指令，因而执行异常产生指令将引起软中断。软中断指令通常用于需要提升运行软件的特权等级或申请系统服务的情况，包括用户程序请求操作系 统服务的管理员调用（Supervisor Call，SVC）指令、客户操作系统请求虚拟机管理器服务的虚拟机管理器调用(Hypervisor Call HVC指令和在非安全状态请求安全状态服务的安全监视器调用（Secure Monitor Call，SMC）指令等。

中断（Interrupts)

与 ARMv7 之前的架构类似，ARMv8-A 架构也支持两种中断：IRQ 和 FIQ，后者比前者优先级更高。除了某些加载多个数值的指令可以被中断打断外，中断响应一定是发生在开中断状态下当前指令执行结束 之后。由于 IRQ 和FIQ 中断的发生都不是直接由软件执行引起的，因而都属于异步异常。

2) ARMv8-A 的异常处理

在 AArch64 状态执行程序时，只有进入异常处理或者从异常返回时才能够切换异常等级。进入异常处理时，异常等级可以保持不变或者提升，但不允许降低；相反，从异常返回时，异常等级可以保持不变或 者降低，但不允许提升。

在进入异常处理时，保持的原有异常等级或者改变到的新等级被称为该异常的**目标异常等级**（Target Exception Level）。每一种异常类型都有一个目标异常等级，该等级或者是该异常类型隐含的，或者是由系统寄存器中的配置位定义的。但异常的目标异常等级不能是 EL0。如果在异常等级 n 处理异常，则异常发生时，处理器的硬件自动执行异常处理的下列隐操作：

首先更新备份程序状态寄存器 SPSR_ELn，以保存异常处理结束返回时恢复现场必须的 PSTATE信息。

用新的处理器状态信息更新程序状态 PSTATE。如果需要，通过此步可以提升异常等级。

将异常处理结束返回的地址保存在异常链接寄存器 ELR_ELn 中。

上一张图给出了一个 ARMv8-A 异常处理流程的实例。图 中2 n=1，即异常发生于 EL0 等级的程序运行过程中，并且切换到 EL1 执行异常处理程序。在异常处理程序的最后将执行异常返回指令 ERET。该指令的执行将使 SPSR_ELn 中保存的程序状态信息被恢复到 PSTATE，并通过恢复 ELR_ELn 至程序计数器 PC 令程序返回到被异常打断的断点位置继续执行。

3) AArch64 的异常向量与异常向量表

在 AArch64 状态下，每个异常等级都有其自身的异常向量表，因而共有 EL3、EL2 和 EL1 三个异常向量表。每个异常等级都有其相应的向量基址寄存器（Vector Base Address Register，VBAR），指明在该异常等级的异常向量表的基地址。

某个 ELn 等级的异常向量表保存了发生在 ELn 等级的所有类型的异常的异常向量，而特定的异常向量在异常向量表中的偏移地址是固定的。

当异常发生并转入 AArch64 状态时，异常向量表将提供以下信息：异常是属于同步异常、系统错误、IRQ 和 FIQ 中的哪一类别；引起异常的异常等级的信息，以及使用哪一个堆栈指针的信息和寄存器文件的状态。

可以认为每个异常等级的异常向量表实际上有4组，每组给出的四个异常入口分别对应同步异常、

IRQ、FIQ和系统错误这四种异常类别。至于应该选择哪一组的异常向量，则取决于异常是发生于当前异常等级还是更低的异常等级、异常将使用哪一个堆栈指针（SP0 还是 SPn）以及异常状态所处的执行状态（AArch64 或 AArch32）等因素。具体而言：

如果异常发生于当前异常等级并且使用 SP_EL0 堆栈指针，则使用第一组异常向量。

异常发生于当前异常等级并且使用 SP_EL1、SP_EL2 或 SP_EL3 堆栈指针，则使用第二组异常向量。

如果发生于比当前异常等级更低的异常等级，且比当前异常等级低一级的异常等级处于 AArch64 执行模式，则使用第三组异常向量。

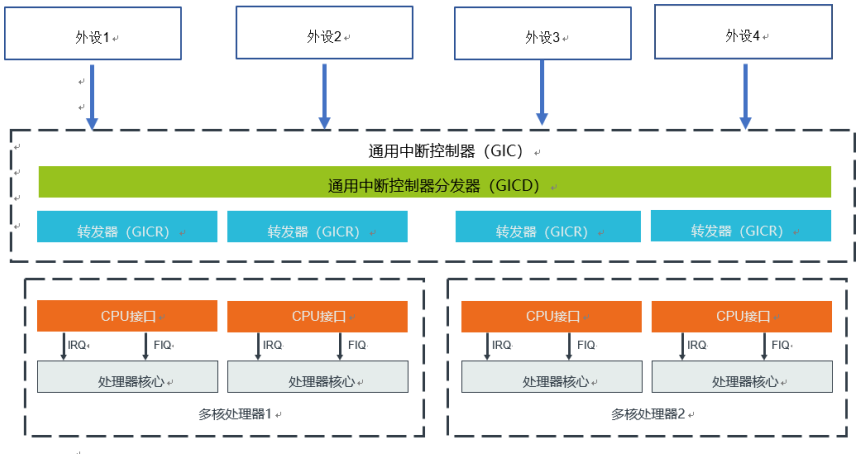
如果发生于比当前异常等级更低的异常等级，且比当前异常等级低一级的异常等级处于 AArch32 执行模式，则使用第四组异常向量。

注意异常向量表的各个表项存放的异常向量不是异常处理程序的入口地址，而是异常处理程序要执行 的指令序列。换句话说，异常向量表的每个表项至少保存了相应异常类型的异常处理程序执行的第一条指 令所对应的指令字。

在 AArch64 状态下，异常向量表的每个表项的长度从 ARMv7-A 架构的 4 字节扩展到 128 字节，可以存放 32 条指令。一般情况下，这足以存放一个完整的顶层异常处理程序的代码。通常异常处理程序会包含查找具体异常源的代码，然后调用相关的处理函数完成具体处理流程。

4) ARMv8-A 的通用中断控制器架构

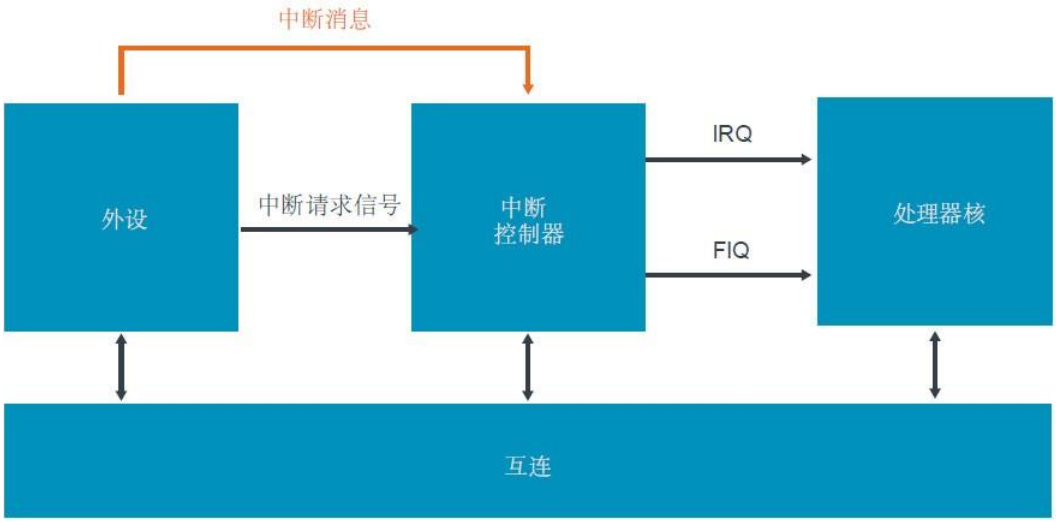
对 ARMv8-A 架构这样的高性能复杂处理器而言，外部中断源应该是相当多的，而 ARM 处理器内核本身只能支持 FIQ 和 IRQ 两级外部中断请求输入，因而在 ARM 架构下，系统通过通用中断控制器实现中断请求的仲裁、优先级排队和向处理器中断申请等操作。



ARMv8-A 的通用中断控制器架构

上图描述了从程序员角度看到的 ARMv8-A 通用中断控制器架构。外设的中断请求由中断控制器接收后进行优先级判优，然后发送给相应的处理器内核。

除了上图所示的外设通过中断请求信号线向中断控制器提交中断请求的传统方式外，ARM 通用中断控制器规范的v3版本还增加了一种通过消息提出中断请求的方式，这种中断被称为**消息信号中断**（message-signaled interrupts，**MSI**）。如下图所示，外设通过向中断控制器中的寄存器执行写操作触发中断请求，不再需要专门的中断请求信号线，这对有非常多中断源的大型系统非常有利。两种中断请求方式的中断处理过程基本相同。



ARM 通用中断控制器的中断请求方式

中断控制器处理的中断源分为以下四种类型：

共享外设中断（Shared Peripheral Interrupt，**SPI**）：外设的这类中断请求可以被连接到任何一个处理器内核。

私有外设中断（Private Peripheral Interrupt，**PPI**）：只属于某一个处理器内核的外设的中断请求，例如通用定时器的中断请求。

软件产生的中断（Software Generated Interrupt,**SGI**）：由软件写入中断控制器内的SGI寄存器引发的中断请求，通常用于处理器间通信；

特定位置外设中断（Locality-specific Peripheral Interrupt，**LPI**）：边沿触发的基于消息的中断，其编程模式与其他类中断源完全不同。

每个中断源都有其唯一的中断标识（INTID），且中断类型通过中断标识的范围确定。表 2.1给出了ARM通用中断控制器中断标识与中断类型的对照关系。SPI共享外设中断可以使用消息信号方式，而 LPI 本地专有外设中断必须是消息信号中断。

表 2.1 ARM 通用中断控制器中断标识与中断类型对照

中断标识（INTID）	中断类型	中断类型缩写
0 ~15	软件产生的中断	SGI
16 ~31 1056 ~ 1119 (GICv3.1*)	私有外设中断	PPI
32 ~ 1019 4096 ~ 5119 (GICv3.1*)	共享外设中断	SPI
1020 ~ 1023	特殊中断号	
1024 ~ 8191	保留	
8192及以上	特定位置外设中断	LPI

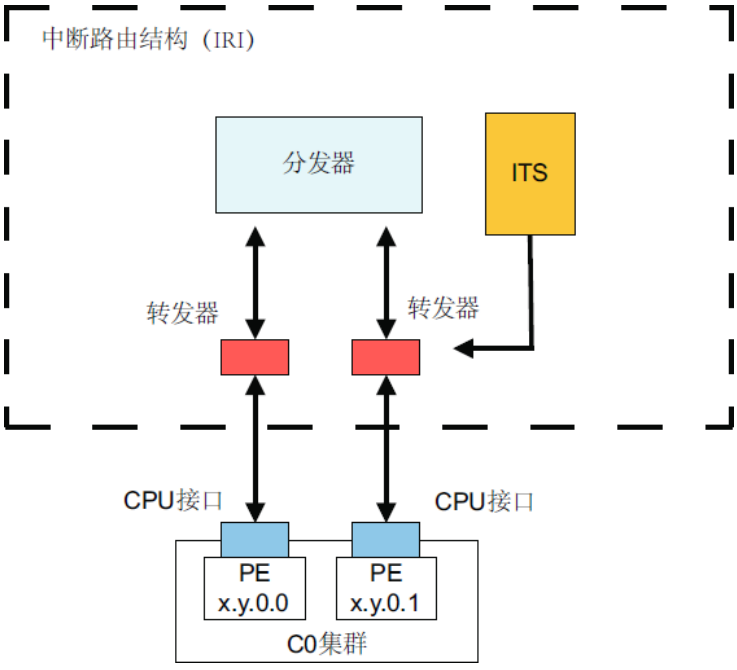
GICv3.1*：与 GIC 早期版本不兼容

右图显示了 ARM 通用中断控制器架构的逻辑结构。

通用中断控制器由四种逻辑部件组成：**分发器**(Distributor)、**转发器**(Redistributor)、**CPU 接口**和 **ITS** (Interrupt Translation Service，**中断翻译服务**)部件。

除了公共的分发器外，每个处理器内核都有一个与之相连的转发器和一个CPU接口。分发器、转发器和ITS组成了中断路由结构（Interrupt Routing Infrastructure，IRI）。

中断架构中也包含了程序员所见的通用中断控制器寄存器接口：分发器接口、转发器接口和 CPU 接口。一般而言，分发器接口和转发器接口用于配置中断，而 CPU 接口则用于处理中断。



ARM 通用中断控制器的逻辑组成

分发器负责管理 SPI 类型的中断，并将中断请求发送给转发器。分发器的寄存器可以通过寄存器映射方式访问，而且分发器的配置直接影响相关的所有处理单元。分发器接口的主要功能有：中断优先级管理和中断请求分发；SPI 的使能和禁止；为每个SPI 中断源设置中断优先级；为每个SPI 中断源配置路由信息；配置每个 SPI 源的中断触发方式(边沿触发还是电平触发)；生成消息信号 SPI中断;控制SPI 的激活和挂起(Pending) 状态;确认在每个安全状态下所用的编程模型 (关联路由方式还是传统方式）等。

转发器管理 PPI、SGI 和 LPI 类型的中断，并将中断请求投递给处理器内核。转发器提供的编程接口可以实现以下功能：使能和禁止 SGI 和PPI 中断源；设置 SGI 和PPI 中断源的中断优先级；配置每个 PPI 中断源的中断

触发方式（边沿触发还是电平触发）；将每个 SGI 和 PPI 中断源分配至一个中断组；控制SGI 和 PPI 的状态；控制在存储器中存放的支持 LPI 属性和挂起状态的相关数据结构的基地址；对其控制的处理单元提供电源管理支持等。ITS 可以将事件翻译成物理 LPI[①]，并发送给转发器和 CPU 接口。在 ARM 通用中断控制器规范 GICv3 架构中，ITS 是一个可选的配置，一个 IRI 中可以没有 ITS，也可以配置多个 ITS。而直接 LRI 并不需要使用 ITS 翻译，可以通过转发器内的寄存器实现。ARM GICv3 要求，兼容该规范的处理器在将 LPI 转发至转发器时需从两种方式中只选择其中一种实现：通过 ITS 转发；通过直接写入转发器内的寄存器实现。

处理器内核内包含了 CPU 接口，用于处理所有异常等级的物理中断。CPU 接口内包含了用于中断处理的系统寄存器（ICC_*_ELn），编程接口支持的功能有：中断处理的通用控制与配置功能；中断响应； 中断优先级降级和中断撤销；设置处理单元的优先级屏蔽状态；定义处理单元的抢占策略；确定处理单元 的最高优先级挂起中断等。

PPI 被从中断源直接分配给本地转发器，SPI 的转发路径为从中断源经分发器到目标转发器，最后送入相应的 CPU 接口。SGI 是由软件通过 CPU 接口和转发器产生的，可以通过分发器将其转发至一个或多个目标转发器或相应的 CPU 接口。

三、 汇编语言基本结构

例如：

```
AREA Init,CODE,READONLY     @AREA 定义代码段，段名 Init;代码段，只读
ENTRY                       @伪操作，第一条指令的入口
Start                       @标号，一段代码的开始，用于标记，无意义，必须顶格
MOV  r0,#10                 @将 10 存入 r0 寄存器，整型常量前面用#号
MOV  r1,#3                  @将 3 存入 r1 寄存器，r0 和 r1 相当于两个变量，只是名称固定，在寄存器的存储空间内
ADD  r0,r0,r1               @将 r0 内数据与 r1 内数据相加，相加后数据放在 r0 内
;Stop                       @停止标号，下面的代码用于停止运行程序
;MOV  r0,#0x18              @软件异常中断响应
;LDR  r1,=0x20026           @ADP 停止运行，应用退出
;SWI  0x123456              @ARM 半主机软件中断
END
```

1. 基本概念：

1） 寄存器：如 R0、R1 等

2) ARM 的汇编编程，本质上就是针对 CPU 寄存器的编程

3) 指令：即操作码，直接控制 CPU，如 MOV，指令包括跳转指令、数据处理指令、乘法指令、PSR 访问指令、加载或存储指令、数据交换指令、移位指令等

4) 伪操作：作用于编译器，大多用于定义和控制。如 AREA，包括符号定义伪操作、数据定义伪操作、控制伪操作等

5) 标号：仅是一种标识。在跳转语句中，可以指向要跳转到的标识号位置，在 ARM 汇编中，标号代表一个地址，段内标号的地址在汇编时确定，段外标号的地址值在连接时确定

6) 符号:即标号(代表地址)、变量名、数字常量名等。符号的命名规则如下：

- 符号由大小写字母、数字以及下划线组成；
- 除局部标号以数字开头外，其它的符号不能以数字开头；
- 符号区分大小写，且所有字符都是有意义的；
- 符号在其作用域范围你必须是唯一的；
- 符号不能与系统内部或系统预定义的符号同名；
- 符号不要与指令助记符、伪指令同名。

2. 段定义

在汇编语言中，以相对独立的指令或数据序列的程序段组成程序代码段的划分：数据段、代码段。一个汇编程序至少有一个代码段

1） 代码段

上面的例子为代码段。

AREA 定义一个段，并说明所定义段的相关属性

CODE 用以指明为代码段

ENTRY 标识程序的入口点

END 为程序结束。

2） 数据段

AREA　　DATAAREA, DATA, BIINIT, ALLGN=2

DISPBUF SPACE 200

RCVBUF SPACE 200

DATA 用以指明为数据段，

SPACE 分配 200 字节的存储单元并初始化为 0

3） 汇编语言结构

[标号] [指令或伪操作]

所有标号必须在一行的顶格书写，其后不加冒号；

所有指令均不能顶格写；

指令助记符大小写敏感，不能大小写混合，只能全部大写或全部小写；

;为注释

@代码行注释，同;

#整行注释或直接操作数前缀

\为换行符

ENTRY 为程序的入口

END 为程序的结束

四、 ARM 的寻址方式

最简单的汇编指令格式是操作码和操作数，如：MOV r0,#10

操作码：即 CPU 指令 如 MOV ADD

操作数：即表示数据是在寄存器中还是在内存中，是绝对地址还是相对地址。操作数部分要解决的问题是，到哪里去获取操作数，获取操作数的方式就是寻址方式。

ARM 每一条指令都是 32 位机器码，对应 CPU 的位数

ARM 指令格式：在 32 位中分为 7 个位域，每个位域分别存储不同意义的编码（二进制数）

31 ~ 28 27~25 24 ~ 21 20 19~16 15~12 11 ~ 0

| Cond | 001 | Opcode | S | Rn | Rd | Operand2 |

对应输入的一条指令为：

<Opcode>{<cond>}{s} <Rd>,<Rn>,<Operand2>

Opcode：指令操作码，用 4 个位存储，如 MOV、ADD 每一个操作码和操作数最终都是以二进制形式存在

Cond：指令的条件码 用 4 个位来储，可省略

如：***ADD r0,r0,#1***　　@计算 r0 加 1 后的值，再写入到 r0 内

ADDEQ r0,r0,#1　　@只有在 CPSR 寄存器条件标志位满足指定条件时，才计算 r0 加 1 后的值，再写入到 r0 内

其中条件助记符如下：

- EQ　　相等

- NE　　不相等
- MI　　负数
- VS　　溢出
- PL　　正数或零
- HI　　无符号大于
- LS　　无符号小于或等于
- CS　　无符号大于或等于
- CC　　无符号小于
- GT　　有符号大于
- GE　　有符号大于或等于
- LE　　有符号小于或等于
- LT　　有符号小于
- AL　　无条件执行

S：决定指令的操作是否影响 CPSR 的值,用一个位存储，省略则表示为 0 值，否则为 1 值

如：***SUBS R0,R0,#1***　　@R0 减 1，结果放入 R0，同时响影 CPSR 的值

SUB R0,R0,#1　　@R0 减 1，结果放入 R0，不影响 CPSR 的值

Rd：目标寄存器编码　用 4 位存储

Rn：第 1 个操作数的寄存器编码 用 4 位存储

Operand2：第 2 个操作数　用 12 位存储

寻址方式：是根据指令中给出的地址码字段来实现寻找真实操作数地址的方式,共 8 种寻址方式：寄存器寻址、立即寻址、寄存器间接寻址、基址寻址、多寄存器寻址、堆栈寻址、相对寻址、寄存器移位寻址

1. 立即数寻址

操作数是常量，用#表示常量。

例如：***MOV R0,#0xFF000***　　@指令省略了第 1 个操作数寄存器。将立即数 0xFF000(第 2 操作数)装入 R0 寄存器

SUB R0,R0,#64　　@R0 减 64，结果放入 R0

#表示立即数　0x 或&表示 16 进制数，否则表示十进制数

立即数寻址指令中的地址码就是操作数本身，可以立即使用的操作数。其中，#0xFF000 和#64 都是立即数。该立即数位于 32 位机器码中，占低位的 12 位。也就是说在 ARM 指令中以 12 位存储一个数据，那么，32 位的数据（long 或 float）如何存储?

32 位的数据以一种特殊的方式来处理，其中：高 4 位表示的无符号整数，低 8 位补 0 扩展为 32 位，然后循环右移 x 位来代表一个数。x=高 4 位整数*2

所以，不是每一个 32 位数都是合法的立即数，只有能通过上述构造得到的才是合法的立好数。

如：合法立即数：0xff,0x104,0xff0

不合法立即数：ox101，0x102,0xff1

2. 寄存器寻址

操作数的值在寄存器中，指令执行时直接取出寄存器值来操作

例如：***MOV R1,R2***　　@将 R2 的值存入 R1　在第 1 个操作数寄存器的位置存放 R2 编码

SUB R0,R1,R2 @将 R1 的值减去 R2 的值，结果保存到 R0 在第 2 操作数位置，存放的是寄存器 R2 的编码

寄存器寻址是根据寄存器编码获取寄存器内存储的操作数

3. 寄存器间接寻址

操作数从寄存器所指向的内存中取出，寄存地存储的是内存地址

例如：***LDR R1,[R2]*** @将 R2 指向的存储单元的数据读出，保存在 R1 中 R2 相当于指针变量

STR R1,[R2] @将 R1 的值写入到 R2 所指向的内存

SWP R1,R1,[R2] @将寄存器 R1 的值和 R2 指定的存储单元的内容交换

[R2]表示寄存器所指向的内存，相当于 *p

LDR 指令用于读取内存数据

STR 指令用于写入内存数据

4. 寄存器基址寻址

操作数从内存址偏移后读取数据。常用于查表、数组操作、功能部件寄存器访问等。

基址：相当于首地址，地址偏移后取值作为操作数

基址寄存器：用来存放基址的寄存器

变址寻址：基址寻址也称为变址寻址

1） 前索引寻址

例如：***LDR R2,[R3,#4]*** @读取 R3+4 地址上的存储单元的内容，放入 R2

LDR R2,[R3,#4]! @读取 R3+4 地址上的存储单元的内容，放入 R2，然后 R3 内的地址变为 R3+4,即 R3=R3+4

[R3,#4] 表示地址偏移后取值，相当于*(p+4) 或 p[4]，R3 内保存的地址不变

[R3,#4]! 表示地址偏移后取值，相当于*(p+4) 或 p[4],!表示回写，即 R3=R3-4 ,R3 的值发生了改变

2） 后索引寻址

例如：***LDR R2,[R3],#4*** @先读取 R3 地址上的存储单元的内容，放入 R2，然后 R3 内的地址变为 R3+4,即 R3=R3+4

[R3],#4 类似于 *p++ 只不过自加的不是 1 ，而是指定的 4

[R3,#4]! 类似于 *++p 只不过自加的不是 1，而是指定的 4

3） 寄存器的值作索引

例如：***LDR R2,[R3,R0]*** @R0 内存储索引值，R3 内存地址，读取 R3+R0 地址上的存储单元的内容，放入 R2

[R3,R0] 表示地址偏移后取值，相当于*(p+i) 或 p[i]

5. 多寄存器寻址

一次可传送多个寄存器的值，也称为块拷贝寻址

例如：***LDMIA R1!,{R2-R7,R12}*** @将 R1 指向的存储单元中的数据读写到 R2～R7、R12 中，然后 R1 自加 1

STMIA R1!,{R2-R7,R12} @将寄存器 R2～R7、R12 的值保存到 R1 指向的存储单元中，然后 R1 自加 1

其中 R1 是基址寄存器，用来存基址，R2-R7、R12 用来存数据 赋值编号小的寄存器与低地址相对应，与寄存器列表顺序无关

! 为可选后缀，表示改变 R1 的值，则当数据传送完毕之后，将最后的地址写入基址寄存器

基址寄存器不允许为 R15，寄存器列表可以为 R0～R15 的任意组合。这里 R1 没有写成[R1]!，是因为这个位不是操作数位，而是寄存器位

LDMIA 和 STMIA 是块拷贝指令，LDMIA 是从 R1 所指向的内存中读数据，STMIA 是向 R1 所指向的内存写入数据

R1 指向的是连续地址空间

6. 寄存器堆栈寻址

是按特定顺序存取存储区，按后进先出原则，使用专门的寄存器 SP（堆栈指针)指向一块存储区

例如：***LDMIA SP!,{R2-R7,R12}*** @将栈内的数据，读写到 R2～R7、R12 中，然后下一个地址成为栈顶

STMIA SP!,{R2-R7,R12} @将寄存器 R2～R7、R12 的值保存到 SP 指向的栈中

SP 指向的是栈顶

7. 相对寻址

即读取指令本身在内存中的地址。是相对于 PC 内指令地址偏移后的地址。由程序计数器 PC 提供基准地址，指令中的地址码字段作为偏移量，两者相加后得到的地址即为操作数的有效地址。

例如：***B LOOP*** @B 指令用于跳转到标号 LOOP 指令处执行代码...

LOOP MOV R6 #1

其中 LOOP 仅仅是标号，而不是地址，不是 CPU 指令，所以在指令的机器码中没有标号的机器码，而是计算出了从 B 指令到 MOV 指令之间内存地址的差值，这个差值相当于 PC 的偏移量，即相当于：add pc,pc,#偏移量

B 指令引起了 PC 寄存器值的变化，PC 内永远保存将要运行指令在内存中的地址。

8. 寄存器移位寻址

操作数在被使用前进行移位运算

例如：***MOV R0,R2,LSL #3*** @R2 的值左移 3 位，结果放入 R0，即是 R0=R2×8

LSL 是移位指令，用于将前面寄存器的数据 左移

ARM 汇编语言语句由 指令、伪指令、伪操作、宏指令组成

五、 ARM 指令集

共包括 6 种类型：数据处理指令、跳转指令、程序状态指令、加载存取储指令、协处理指令、软中断指令。

1. 数据处理指令

数据处理指令，只能对寄存器内容进行操作，而不能对内存进行操作，所有数据处理指令均可使用 S 后缀，并影响状态标志，包括：数据传输指令、算数指令、乘法指令、逻辑运算指令、比较指令、移位指令。

1） 数据传输指令

MOV 数据传送指令

格式：***MOV Rd,<op1>***

功能：Rd = 操作数, 操作数可以是寄存器、被移位的寄存器或立即数。

例如：

MOV R0,#0xFF000 @立即寻址，将立即数 0xFF000(第 2 操作数)装入 R0 寄存器

MOV R1,R2 @寄存器寻址，将 R2 的值存入 R1

MOV R0,R2,LSL #3 @移位寻址，R2 的值左移 3 位，结果放入 R0

MVN 数据取反传送指令

格式：***MVN <Rd>,<op1>***

功能：将操作数传送到目的寄存器 Rd 中，但该值在传送前被按位取反，即 Rd=!op1

例如：

MVN R0,#0 ;R0=-1

mvn 意为“取反传输”，它把源寄存器的每一位取反，将得到的结果写入结果寄存器。

movs 和 mvns 指令对 pc 寄存器赋值时有特殊含义，表示要求在赋值的同时从 spsr 中恢复 cpsr。

mov 和 mvn 指令，编译器会进行智能的转化。比如指令“mov r1, 0xffffff00”中的立即数是非法的。

在编译时，编译器将其转化为“mvn r1, 0xff”，这样就不违背立即数的要求。所以对于 mov 和 mvn 指令，可以认为：合法的立即数反码也是合法的立即数。

2) 算术指令

ADD 加法指令

格式：***ADD{<cond>}{S} <Rd>,<Rn>,<op2>***;

功能：Rd = Rn+op2

例如：

ADD R0,R1,#5 @R0=R1+5

ADD R0,R1,R2 @R0=R1+R2

ADD R0,R1,R2,LSL#5 @R0=R1+R2 左移 5 位

ADC 带进位加法指令

格式：***ADC{<cond>}{S} <Rd>,<Rn>,<op2>***;

功能：Rd = Rn+op2+carry ,carry 为进位标志值。该指令用于实现超过 32 位的数的加法。

例如：

第一个 64 位操作数存放在寄存器 R2，R3 中；

第二个 64 位操作数存放在寄存器 R4，R5 中；

64 位结果存放在 R0，R1 中。

ADDS R0,R2,R4 @低 32 位相加，S 表示结果影响条件标志位的值

ADC R1,R3,R5 @高 32 位相加

SUB 减法指令

格式：***SUB{<cond>}{S} <Rd>,<Rn>,<op2>***;

功能：Rd = Rn-op2

例如：

SUB R0,R1,#5 @R0=R1-5

SUB R0,R1,R2 @R0=R1-R2

SUB R0,R1,R2,LSL#5 @R0=R1-R2 左移 5 位

SBC 带借位减法指令

格式：***SBC{<cond>}{S} <Rd>,<Rn>,<op2>***;

功能：Rd = Rn-op2-!carry

SUB 和 SBC 生成进位标志的方式不同于常规，如果需要借位则清除进位标志，所以指令要对进位标志进行一个非操作。

例如：

第一个 64 位操作数存放在寄存器 R2，R3 中；

第二个 64 位操作数存放在寄存器 R4，R5 中；

64 位结果存放在 R0，R1 中。

SUBS R0,R2,R4; 低 32 位相减，S 表示结果影响条件标志位的值

SBC R1,R3,R5; 高 32 位相减

其它减法指令

RSB 反向减法指令,同 SUB 指令，但倒换了两操作数的前后位置，即 Rd = op2-Rn。

RSC 带借位的反向减法指令,同 SBC 指令，但倒换了两操作数的前后位置，即 Rd = op2-Rn-!carry。

rsb r0, r1, r2 /* r0 = r2 – r1 */

rsc r0, r1, r2 /* r0 = r2 – r1 + carry – 1 */

adds 和 adcs 在进位时将 cpsr 的 C 标志置 1；否则置 0。

subs 和 sbcs 在产生借位时将 cpsr 的 C 标志置 0；否则置 1。

3） 乘法指令

MUL 32 位乘法指令

格式：***MUL{<cond>}{S} <Rd>,<Rn>,<op2>***;

功能：Rd = Rn×op2

该指令根据 S 标志，决定操作是否影响 CPSR 的值；其中 op2 必须为寄存器。Rn 和 op2 的值为 32 位的有符号数或无符号数。

例如：

MULS R0,R1,R2 ;R0 = R1×R2，结果影响寄存器 CPSR 的值

MLA 32 位乘加指令

格式：***MLA{<cond>}{S} <Rd>,<Rn>,<op2>,<op3>***;

功能：Rd = Rn×op2+op3 op2 和 op3 必须为寄存器。Rn、op2 和 op3 的值为 32 位的有符号数或无符号数。

例如：

MLA R0,R1,R2,R3 ;R0 = R1×R2+R3

SMULL 64 位有符号数乘法指令

格式：***SMULL{<cond>}{S} <Rdl>,<Rdh>,<Rn>,<op2>***;

功能：Rdh Rdl = Rn×op2 Rdh、Rdl 和 op2 均为寄存器。Rn 和 op2 的值为 32 位的有符号数。

例如：

SMULL R0,R1,R2,R3 @R0 = R2×R3 的低 32 位 R1 = R2×R3 的高 32 位

SMLAL 64 位有符号数乘加指令

格式：SMLAL{<cond>}{S} <Rdl>,<Rdh>,<Rn>,<op2>;

功能：Rdh Rdl = Rn×op2+Rdh Rdl；Rdh、Rdl 和 op2 均为寄存器。Rn 和 op2 的值为 32 位的有符号数，Rdh Rdl 的值为 64 位的加数。

例如：

SMLAL R0,R1,R2,R3 @R0 = R2×R3 的低 32 位+R0 R1 = R2×R3 的高 32 位+R1

UMULL 64 位无符号数乘法指令

格式：***UMULL{<cond>}{S} <Rdl>,<Rdh>,<Rn>,<op2>***;

功能：同 SMULL 指令，但指令中 Rn 和 op2 的值为 32 位的无符号数。

例如：

UMULL R0,R1,R2,R3 @R0 = R2×R3 的低 32 位 R1 = R2×R3 的高 32 位 其中 R2，R3 的值为无符号数

UMLAL 64 位无符号数乘加指令

格式：***UMLAL {<cond>}{S} <Rdl>,<Rdh>,<Rn>,<op2>***;

功能：同 SMLAL 指令，但指令中 Rn，op2 的值为 32 位的无符号数，Rdh Rdl 的值为 64 位无符号数。

例如：

UMLAL R0,R1,R2,R3 @R0 = R2×R3 的低 32 位+R0 R1 = R2×R3 的高 32 位+R1

其中 R2，R3 的值为 32 位无符号数 R1，R0 的值为 64 位无符号数

4） 逻辑运算指令

AND 逻辑与指令

格式：***AND{<cond>}{S} <Rd>,<Rn>,<op2>***;

功能：Rd = Rn AND op2 一般用于清除 Rn 的特定几位。

例如：

AND R0,R0,#5 @保持 R0 的第 0 位和第 2 位，其余位清 0

ORR 逻辑或指令

格式：***ORR{<cond>}{S} <Rd>,<Rn>,<op2>***;

功能：Rd = Rn OR op2 一般用于设置 Rn 的特定几位。

例如：

ORR R0,R0,#5 @R0 的第 0 位和第 2 位设置为 1，其余位不变

EOR 逻辑异或指令

格式：***EOR{<cond>}{S} <Rd>,<Rn>,<op2>***;

功能：Rd = Rn EOR op2 一般用于将 Rn 的特定几位取反。

例如：

EOR R0,R0,#5 @R0 的第 0 位和第 2 位取反，其余位不变

BIC 位清除指令

格式：BIC{<cond>}{S} <Rd>,<Rn>,<op2>;

功能：Rd = Rn AND (!op2) 用于清除寄存器 Rn 中的某些位，并把结果存放到目的寄存器 Rd 中

例如：

BIC R0,R0,＃5　　@R0 中第 0 位和第 2 位清 0，其余位不变

5) 比较指令

CMP 比较指令

格式：***CMP{<cond>} <Rn>,<op1>***;

功能：Rn-op1，根据结果更新 CPSR 中条件标志位的值。

例如：

CMP R0,＃5　　@计算 R0-5，根据结果设置条件标志位

ADDGT R0,R0,＃5　　@ADD 为加法指令，GT 为判断条件标志位是 否大于 5, 如果 R0>5，则执行 ADD 指令

CMN 反值比较指令

格式：***CMN{<cond>} <Rn>,<op1>***;

功能：同 CMP 指令，但寄存器 Rn 的值是和 op1 取负的值进行比较。

例如：

CMN R0,＃5　　@把 R0 与-5 进行比较

TST 位测试指令

格式：***TST{<cond>} <Rn>,<op1>***;

功能： Rn AND op1 按位与后，更新 CPSR 中条件标志位，用于检查寄存器 Rn 是否设置了 op1 中相应的位。

例如：

TST R0，＃5　　@测试 R0 中第 0 位和第 2 位是否为 1

TEQ 相等测试指令

格式：***TEQ{<cond>} <Rn>,<op1>***;

功能：Rn EOR op1 按位异或后，更新 CPSR 中条件标志位，用于检查寄存器 Rn 的值是否和 op1 所表示的值相等。

例如：

TEQ R0,＃5　　@判断 R0 的值是否和 5 相等

6) 移位指令（位运算指令）

LSL (或 ASL)左移

格式：寄存器,LSL(或 ASL) 操作数

功能：将寄存器内的数据左移，操作数是移位的位数在 0-31 之间

例如：

MOV R0, R1, LSL#2　　@将 R1 中的内容左移两位后传送到 R0 中

LSR 操作右移

格式: 寄存器 LSR 操作数

功能: 将寄存器内的数据右移

例如：

MOV R0, R1, LSR#2　　@将 R1 中的内容右移两位后传送到 R0 中,左端用零来填充。

其它移位

ASR 右移，左端用第 31 位值来填充

ROR 右移，循环右移，左端用右端移出的位来填充

RRX 右移，循环右移，左端用进位标志位 C 来填充

2. 跳转指令

1） B 跳转指令

格式：***B{<cond>} <addr>***;

功能：跳转到 addr 地址。

例如：

```
B exit                          @程序跳转到标号 exit 处
```

2） BL 带返回的跳转指令

格式：***BL{<cond>} <addr>***;

功能：同 B 指令，但 BL 指令执行跳转操作的同时，还将 PC（寄存器 R15）的值保存到 LR 寄存器（寄存器 R14）中。该指令用于实现子程序调用，程序的返回可通过把 LR 寄存器的值复制到 PC 寄存器中来实现。

例如：

```
BL func                         @调用子程序 func
…
func
…
MOV R15,R14                     @子程序返回
```

3） 其它跳转指令

BLX 带返回和状态切换的跳转指令，用于子程序调用和程序 Thumb 状态的切换。

BX 带状态切换的跳转指令，处理器跳转到目标地址处，目标地址处的指令可以是 ARM 指令，也可以是 Thumb 指令。

跳转指令用于实现程序的跳转和程序状态的切换。ARM 程序设计中，实现程序跳转有两种方式：跳转指令、直接向程序寄存器 PC 中写入目标地址值。

3. 程序状态指令

用于状态寄存器和通用寄存器间传送数据。总共有两条指令：MRS 和 MSR。两者结合可用来修改程序状态寄存器的值。

1） MRS 程序状态寄存器到通用寄存器的数据传送指令

格式：***MRS{<cond>} <Rd>,CPSR/SPSR***;

功能：用于将程序状态寄存器的内容传送到目标寄存器 Rd 中。

例如：

```
MRS R0,CPSR                     @状态寄存器 CPSR 的值存入寄存器 R0 中
```

2） MSR 通用寄存器到程序状态寄存器的数据传送指令

格式：***MSR{<cond>} CPSR/SPSR_ <field>,<op1>***;

功能：用于将寄存器 Rd 的值传送到程序状态寄存器中。

<field>：用来设置状态寄存器中需要操作的位。

32 位的状态寄存器可以分为 4 个域：

位[31：24]为条件标志位域，用 f 表示。

位[23：16]为状态位域，用 s 表示。

位[15：8]为扩展位域，用 x 表示。

位[7：0]为控制位域，用 c 表示。

例如：

MSR CPSR_f,R0	@用 R0 的值修改 CPSR 的条件标志域
MSR CPSR_fsxc,#5	@CPSR 的值修改为 5

4. 加载存储指令

该集合的指令使用频繁，当数据存放在内存中时，必须先把数据从内存装载到寄存器，执行完后再把寄存器中的数据存储到内存中。

单数据访存指令

1） 单数据加载指令

格式：LDR(或 LDRB、LDRBT、LDRH、LDRSB、LDRSH、LDRT、STR、STRB、STRBT、STRH、STRT) <Rd>,<addr>;

功能：内存地址中的数据装载到目标寄存器 Rd 中，同时还可以把合成的有效地址写回到基址寄存器。

寻址方式：Rn：基址寄存器。Rm：变址寄存器。Index：偏移量，12 位的无符号数。

LDR Rd,[Rn]	@把内存中地址为 Rn 的字数据装入寄存器 Rd 中
LDR Rd,[Rn,Rm]	@将内存中地址为 Rn+Rm 的字数据装入寄存器 Rd 中
LDR Rd,[Rn,#index]	@将内存中地址为 Rn+index 的字数据装入 Rd 中
LDR Rd,[Rn,Rm,LSL#5]	@将内存中地址为 Rn+Rm×32 的字数据装入 Rd
LDR Rd,[Rn,Rm]!	@将内存中地址为 Rn+Rm 的字数据装入 Rd，并将新地址 Rn+Rm 写入 Rn
LDR Rd,[Rn,#index]!	@将内存中地址为 Rn+index 的字数据装入 Rd，并将新地址 Rn+index 写入 Rn
LDR Rd,[Rn,Rm，LSL#5]!	@将内存中地址为 Rn+Rm×32 的字数据装入 Rd，并将新地址 Rn+Rm×32 写入 Rn
LDR Rd,[Rn],Rm	@将内存中地址为 Rn 的字数据装入寄存器 Rd，并将新地址 Rn+Rm 写入 Rn
LDR Rd,[Rn],#index	@将内存中地址为 Rn 的字数据装入寄存器 Rd，并将新地址 Rn+index 写入 Rn
LDR Rd,[Rn],Rm,LSL#5	@将内存中地址为 Rn 的字数据装入寄存器 Rd，并将新地址 Rn+Rm×32 写入 Rn

各指令的区别：

LDR 字数据加载指令：将内存地址中的字数据装载到目标寄存器 Rd 中

例如：

LDR R0,[R1,R2,LSL＃5]! @将内存中地址为 R1+R2×32 的字数据装入寄存器 R0，并将新地址 R1+R2×32 写入 R1

LDRB 字节数据加载指令：同 LDR，只是从内存读取一个 8 位的字节数据而不是一个 32 位的字数据，并将 Rd 的高 24 位清 0。

例如：

LDRB R0,[R1] @将内存中起始地址为 R1 的一个字节数据装入 R0 中

LDRBT 用户模式的字节数据加载指令：同 LDRB 指令，但无论处理器处于何种模式，都将该指令当作一般用户模式下的内存操作。

LDRH 半字数据加载指令：同 LDR 指令，但该指令只是从内存读取一个 16 位的半字数据而不是一个 32 位的字数据，并将 Rd 的高 16 位清 0。

例如：

LDRH R0,[R1] @将内存中起始地址为 R1 的一个半字数据装入 R0 中

LDRSB 有符号的字节数据加载指令：同 LDRB 指令，但该指令将寄存器 Rd 的高 24 位设置成所装载的字节数据符号位的值。

例如：

LDRSB R0,[R1] @将内存中起始地址为 R1 的一个字节数据装入 R0 中，R0 的高 24 位设置成该字节数据的符号位

LDRSH 有符号的半字数据加载指令：同 LDRH 指令，但该指令将寄存器 Rd 的高 16 位设置成所装载的半字数据符号位的值。

例如：

LDRSH R0,[R1] @将内存中起始地址为 R1 的一个 16 位半字数据装入 R0 中，R0 的高 16 位设置成该半字数据的符号位

LDRT 用户模式的字数据加载指令：同 LDR 指令，但无论处理器处于何种模式，都将该指令当作一般用户模式下的内存操作。有效地址必须是字对齐的

2) 单数据存储指令

格式：STR(或 STR、STRB、STRBT、STRH、STRT) <Rd>,<addr>;

功能：将寄存器数据写入到内存中

寻址方式：Rn：基址寄存器。Rm：变址寄存器。Index：偏移量，12 位的无符号数。

STR Rd,[Rn] @将寄存器 Rd 中的字数据写入到内存中地址为 Rn 内存中

STR Rd,[Rn,Rm] @将寄存器 Rd 中的字数据写入到内存中地址为 Rn+Rm 的内存中

STR Rd,[Rn,＃index] @将寄存器 Rd 中的字数据写入到内存中地址为 Rn+index 内存中

STR Rd,[Rn,Rm,LSL＃5] @将寄存器 Rd 中的字数据写入到内存中地址为 Rn+Rm×32 内存中

STR Rd,[Rn,Rm]! @将寄存器 Rd 中的字数据写入到内存中地址为 Rn+Rm 的内存中

STR Rd,[Rn,＃index]! @将寄存器 Rd 中的字数据写入到内存中地址为 Rn+index 的内存中，并将新地址 Rn+index 写入 Rn

STR Rd,[Rn,Rm,LSL＃5]! @将寄存器 Rd 中的字数据写入到内存中地址为 Rn+Rm×32 的内存中，并将新地址 Rn+Rm×32 写入 Rn

STR Rd,[Rn],Rm @将寄存器 Rd 中的字数据写入到内存中地址为 Rn 的内存中，并将新地址 Rn+Rm 写入 Rn

STR Rd,[Rn],#index @将寄存器 Rd 中的字数据写入到内存中地址为 Rn 的内存中，并将新地址 Rn+index 写入 Rn

STR Rd,[Rn],Rm,LSL#5 @将寄存器 Rd 中的字数据写入到内存中地址为 Rn 的内存中，并将新地址 Rn+Rm×32 写入 Rn

STR 字数据存储指令：把寄存器 Rd 中的字数据（32 位）保存到 addr 所表示的内存地址中，同时还可以把合成的有效地址写回到基址寄存器。

例如：

STR R0,[R1,#5]! @把 R0 中的字数据保存到以 R1+5 为地址的内存中，然后 R1 = R1+5

STRB 字节数据存储指令：把寄存器 Rd 中的低 8 位字节数据保存到 addr 所表示的内存地址中。

例如：

STRB R0,[R1] @将寄存器 R0 中的低 8 位数据存入 R1 表示的内存地址中

STRBT 用户模式的字节数据存储指令：同 STRB 指令，但无论处理器处于何种模式，该指令都将被当作一般用户模式下的内存操作。

STRH 半字数据存储指令：把寄存器 Rd 中的低 16 位半字数据保存到 addr 所表示的内存地址中，而且 addr 所表示的地址必须是半字对齐的。

例如：

STRH R0,[R1] @将寄存器 R0 中的低 16 位数据存入 R1 表示的内存地址中

STRT 用户模式的字数据存储指令：同 STR 指令，但无论处理器处于何种模式，该指令都将被当作一般用户模式下的内存操作。

多数据访存指令

1） 批量数据加载指令

格式：***LDM{<cond>}{<type>} <Rn>{!},<regs>{^};***

功能：从一片连续的内存单元读取数据到各个寄存器中，内存单元的起始地址为基址寄存器 Rn 的值，各个寄存器由寄存器列表 regs 表示。该指令一般用于多个寄存器数据的出栈。

type 字段种类：

- IA 每次传送后地址加 1。
- IB 每次传送前地址加 1。
- DA 每次传送后地址减 1。
- DB 每次传送前地址减 1。
- FD 满递减堆栈。
- ED 空递减堆栈。
- FA 满递增堆栈。
- EA 空递增堆栈。

堆栈寻址的命令 LDMFA/STMFA、LDMEA/STMEA、LDMFD/STMFD、LDMED/STMED。

LDM 和 STM 表示多寄存器寻址，即一次可以传送多个寄存器值。

LDM：一次装载多个，这里用来出栈。

STM：一次存储多个，这里用来入栈。

F/E 表示指针指向的位置

F：full 满堆栈，表示堆栈指针指向最后一个入栈的有效数据项。

E：empty 空堆栈，表示堆栈指针指向下一个要放入的空地址。

A/D 表示堆栈的生长方式

A：堆栈向高地址生长，即递增堆栈。

D：堆栈向低地址生长，即递减堆栈。

注意：有一个约定，编号低的寄存器在存储数据或者加载数据时对应于存储器的低地址。FD、ED、FA 和 EA 指定是满栈还是空栈，是升序栈还是降序栈，用于堆栈寻址。一个满栈的栈指针指向上次写的最后一个数据单元. 空栈的栈指针指向第一个空闲单元。一个降序栈是在内存中反向增长而升序栈在内存中正向增长。

{!}：若选用了此后缀，则当指令执行完毕后，将最后的地址写入基址寄存器。

{^}：当 regs 中不包含 PC 时，该后缀用于指示指令所用的寄存器为用户模式下的寄存器，否则指示指令执行时，将寄存器 SPSR 的值复制到 CPSR 中。

2） 批量数据存储指令

格式：***STM{<cond>}{<type>} <Rn>{!},<regs>{^}***;

功能：将各个寄存器的值存入一片连续的内存单元中，内存单元的起始地址为基址寄存器 Rn 的值各个寄存器由寄存器列表 regs 表示。 该指令一般用于多个寄存器数据的入栈。

{^}：指示指令所用的寄存器为用户模式下的寄存器。其他参数用法同 LDM 指令。

例如：

STMEA R13!,{R0-R12,PC}　@将寄存器 R0~R12 以及程序计数器 PC 的值保存到 R13 指示的堆栈中

数据交换指令

1） 字数据交换指令

格式：***SWP <Rd>,<op1>,[<op2>]***;

功能：Rd = [op2], [op2] = op1 从 op2 所表示的内存装载一个字并把这个字放置到目的寄存器 Rd 中，然后把寄存器 op1 的内容存储到同一内存地址中。

例如：

SWP R0,R1,[R2]　　@将 R2 所表示的内存单元中的字数据装载到 R0，然后将 R1 中的字数据保存到 R2 所表示的内存单元中

2） 字节数据交换指令

格式：***SWPB <Rd>,<op1>,[<op2>]***;

功能：从 op2 所表示的内存装载一个字节并把这个字节放置到目的寄存器 Rd 的低 8 位中，Rd 的高 24 位设置为 0；然后将寄存器 op1 的低 8 位数据存储到同一内存地址中。

例如：

SWPB R0,R1,[R2]　　@将 R2 所表示的内存单元中的一个字节数据装载到 R0 的低 8 位，然后将 R1 中的低 8 位字节，数据保存到 R2 所表示的内存单元中

5. 协处理指令

1) CDP 协处理器操作指令

格式：***CDP{<cond>}<p>,<opcode1>,<CRd>,<CRm>,<CRn>,<opcode2>***;

功能：用于传递指令给协处理器 p，要求其在寄存器 CRn 和 CRm 上，进行操作 opcode1，并把结果存放到 CRd 中，可以使用 opcode2 提供与操作有关的补充信息。指令中的所有寄存器均为协处理器的寄存器，操作由协处理器完成。指令中:

P 为协处理器编号；

CRd 为目的寄存器的协处理器寄存器；

CRm 和 CRn 为存放操作数的协处理器寄存器；

Opcode1 和 opcode2 为协处理器即将执行的操作。

例如：

CDP p5，5，c0，c1，c2，9 @该指令用于通知协处理器 p5，在 c1 和 c2 上执行操作 5 和 9，并将结果存放到 c0 中

2) LDC 协处理器数据读取指令

格式：***LDC {<cond>}{L}<p>,<CRd>,<addr>***;

功能：将 addr 表示的内存地址中的连续数据传送到目的寄存器 CRd 中。

L 表示指令为长读取操作，比如用于双精度数据的传输；

目的寄存器 CRd 为协处理器的寄存器；

addr 的寻址方式同 LDR 指令，其寄存器为 ARM 处理器的寄存器。

例如：

LDC p5，c1，[R1+5] @该指令用于将 R1 + 5 所对应的存储单元中的数据，传送到协处理器 p5 的寄存器 c1 中

3) STC 协处理器数据存储指令

格式：***STC{<cond>}{L}<p>,<CRd>,<addr>***;

功能：将寄存器 CRd 的值传送到 addr 表示的内存地址中。指令中各参数用法同 LDC。

例如：

STC p5，c1，[R1+5] @该指令用于将协处理器 p5 中寄存器 c1 的数据传送到 R1 + 5 所对应的存储单元中

4) MCR ARM 寄存器到协处理器寄存器的数据传送指令

格式：MCR{<cond>}<p>,<op1>,<Rd>,<CRn>,<CRm>{,op2};

功能：将 ARM 处理器的寄存器 Rd 中的数据传送到协处理器 p 的寄存器 CRn，CRm 中；op1，op2 为协处理器将要执行的操作。

例如：

MCR p5,5,R1,C1,C2,9 @该指令将 R1 中的数据传送到协处理器 p5 的寄存器 C1，C2 中，协处理器执行操作 5 和 9

5) MRC 协处理器寄存器到 ARM 寄存器的数据传送指令

格式：***MRC {<cond>}<p>,<op1>,<Rd>,<CRn>,<CRm>{,op2}***;

功能：将协处理器 p 的寄存器 CRn，CRm 的数据传送到 ARM 处理器的寄存器 Rd 中；op1，op2 为协处理器将要执行的操作。

例如：

MRC p5,5,R1,C1,C2,9 @该指令将寄存器 C1，C2 中的数据传送到 R1 中，协处理器 p5 协处理器执行操作 5 和 9

6. 异常中断指令

异常中断产生指令：用于系统调用和调试。

1) SWI 软件中断指令

格式：***SWI {<cond>}*** 24 位的立即数；

功能：用于产生软件中断，以使用户程序调用操作系统的系统例程。

指令中 24 位的立即数指定用户程序调用系统例程的类型，其参数通过通用寄存器传递。当 24 位的立即数被忽略时，系统例程类型由寄存器 R0 指定，其参数通过其他通用寄存器传递。

例如：

SWI 0X05 @调用编号为 05 的系统例程

2) BKPT 断点中断指令

格式：BKPT 16 位的立即数；

功能：用于产生软件断点中断，以便软件调试时使用。16 位的立即数用于保存软件调试中额外的断点信息。

7. 信号量操作指令

信号量操作指令：用于进程间的同步互斥，提供对信号量的原子操作。

六、 ARM 程序常见结构

1. 子函数和主函数

使用 BL 指令进行调用，该指令会把返回的 PC 值保存在 LR

```
AREA Example,CODE,READONLY          @声明代码段 Example
    ENTRY                           @程序入口
Start
   MOV  R0,#0                       @设置实参,将传递给子程骗子的实参存放在 r0 和 r1 内
   MOV  R1,#10
   BL   ADD_SUM                     @调用子程序 ADD_SUM
   B    OVER                        @跳转到 OVER 标号处，进入结尾
ADD_SUM
   ADD  R0,R0,R1                    @实现两数相加
   MOV  PC,LR                       @子程序返回，R0 内为返回的结果
OVER
   END
```

运行过程：

1) 程序从 ENTRY 后面指令处开始运行（即主函数）

2) R0 和 R1 先准备好数据令子函数运算时使用

3) BL 为跳转指令，用于调用子函数 后面为函数标号，在调用函数的过程中，自动将 PC 内的值保存到 LR(R14)内备份，PC 内当前的值为下一条要执行的指令地址 (即 B OVER 指令地址)，在子函数结束前将该地址恢复到 PC 内

4) B 为跳转指令，用于跳转到指定的标号后,此处跳转到程序结尾

5) MOV PC,LR 是子函数内的最后一条语句，用于将 LR 内保存的地址恢复到 PC 内，PC(R15) 程序计数器存储要执行的指令在内存中的地址

6) PC 的值 = 当前正在执行指令在内存中的地址 + 8

2. 条件跳转语句

```
 AREA Example,CODE,READONLY          @声明代码段 Example
      ENTRY                          @程序入口
 Start
    MOV  R0,#2                      @将 R0 赋初值 2
    MOV  R1,#5                      @将 R1 赋初值 5
    ADD  R5,R0,R1                   @将 R0 和 R1 内的值相加并存入 R5
    CMP  R5,#10
    BEQ  DOEQUAL                    @若 R5 为 10，则跳转到 DOEQUAL 标签处
 WAIT
    CMP  R0,R1
    ADDHI R2,R0,#10                 @若 R0 >R1 则 R2=R0+10
    ADDLS R2,R1,#5                  @若 R1<=R2 则 R2=R1+5
 DOEQUAL
    ANDS R1,R1,#0x80                @R1=R1& 0x80,并设置相应标志位
    BNE  WAIT                       @若 R1 的 d7 位为 1 则跳转到 WAIT 标签
 OVER
 END
```

运行过程

1) 程序从 ENTRY 后面指令处开始运行 (即主函数)；

2) R0 和 R1 赋初值 2 和 5

3) 将 R0 与 R1 相加后存入 R5 内

4) CMP 用于比较两个数据，指令格式如下：

CMP 操作数 1,操作数 2

CMP 用于把一个寄存器的内容和另一个寄存器或立即数进行比较,同时更新 CPSR 中条件标志位的值。标志位表示操作数 1 和操作数 2 的关系。然后执行后面的语句

5) 条件助记符 BEQ：B 为跳转指令，EQ 为条件相等，读取 CPSR 内的条件标志位，如相等则跳转到所指定的标号处；BNE B 为跳转指令，NE 为不相等（0），如不相等则跳转到所指定的标号处

6) ADDHI ADD 为相加指令，HI 为无符号大于，如大于则执行相加

7) ADDLS ADD 为相加指令，LS 为无符号小于或等于，如小于或等于则相加 4 位运算

8) ANDS AND 按位与，0x80 取出第 7 位， S 为将该位与的值写入标志位

3. 循环语句

```
AREA Example,CODE,READONLY          @声明代码段 Example
    ENTRY                           @程序入口
Start
    MOV  R1,#0                      @将 R1 赋初值 0
LOOP
    ADD  R1,R1,#1
    CMP  R1,#10
    BCC  LOOP                       @R1 小于 10 则执行跳转到 LOOP 处执行循环, 即 R1
从 0 到 10 后退出循环
END
```

例如：编写一具有完整汇编格式的程序，实现冒泡法排序功能。

设无符号字数据存放在从 0x400004 开始的区域，字数据的数目字存放在 0x400000 中。

```
AREA   SORT，CODE，READONLY
ENTRY
START
MOV     R1，#0x400000
LP
SUBS     R1，R1，#1
BEQ     EXIT
MOV     R7，R1
LDR     R0，=0x400004
LP1
LDR     R2，[R0]，#4
LDR     R3，[R0]
CMP     R2，R3
STRLO   R3，[R0, # -4]
STRLO   R2，[R0]
SUBS     R7，R7，#1
BNE     LP1
B     LP
EXIT
END
```

练习：

1. 编写 1+2+3+...+100 的汇编程序。
2. 实现子函数，该函数返回两个参数中的最大值，在主函数内调用。

七、 伪操作和宏指令

伪指令——是汇编语言程序里的特殊指令助记符，在汇编时被合适的机器指令替代。

伪操作——为汇编程序所用，在源程序进行汇编时由汇编程序处理，只在汇编过程起作用，不参与程序运行。

宏指令——通过伪操作定义的一段独立的代码。在调用它时将宏体插入到源程序中。也就是常说的宏。

说明：所有的伪指令、伪操作和宏指令，均与具体的开发工具中的编译器有关。

1. 宏定义(MACRO、MEND)

格式：

```
MACRO
    {$标号名} 宏名{ $ 参数 1 ， $ 参数 2 ， ……}
    指令序列
MEND
```

MACRO 、 MEND 伪指令可以将一段代码定义为一个整体，称为宏指令，在程序中通过宏指令多次调用该段代码。

{}为可选项

$ 标号在宏指令被展开时，标号会被替换为用户定义的符号

在宏定义体的第一行应声明宏的原型(包含宏名、所需的参数)，然后就可以在汇编程序中通过宏名来调用该指令序列，写在代码段或数据段前面

MEXIT 跳出宏

例如：没有参数的宏（实现子函数返回）

```
MACRO
    MOV_PC_LR               @宏名
    MOV PC,LR               @子程序返回，R0 内为返回的结果
MEND
AREA Example,CODE,READONLY          @声明代码段 Example
    ENTRY                           @程序入口
Start
    MOV  R0,#0                  @设置实参,将传递给子程骗子的实参存放在 r0 和 r1 内
    MOV  R1,#10
    BL   ADD_NUM                @调用子程序 ADD_NUM
    BL   SUB_NUM                @调用子程序 SUB_NUM
    B    OVER                   @跳转到 OVER 标号处，进入结尾
EXPORT ADD_NUM
    ADD_NUM
    ADD  R0,R0,R1                   @实现两数相加
    MOV_PC_LR                       @调用宏，代表子函数结束
EXPORT SUB_NUM
    SUB_NUM
```

```
    SUB  R0,R1,R0                    @实现两数相减
    MOV_PC_LR                        @调用宏，代表子函数结束
OVER
END
```

例如:有参数宏:宏定义从 MACRO 伪指令开始,到 MEND 结束,并可以使用参数。类似于 C 的#define

```
MACRO                                @宏定义
    CALL $Function,$dat1,$dat2       @宏名称为 CALL,带 3 个参数
    IMPORT $Function                 @声明外部子程序 宏开始
    MOV R0,$dat1                     @设置子程序参数,R0=$dat1
    MOV R1,$dat2
    BL $Function                     @调用子程序 宏最后一句
MEND                                 @宏定义结束
CALL FADD1,#3,#2                     @宏调用，后面是三个参数
```

汇编预处理后，宏调用将被展开，程序清单如下:

```
 IMPORT FADD1
 MOV R0,#3
 MOV R1,#3
 BL FADD1
```

2. 符号定义伪操作

1) 定义常量(EQU)

格式：标号名称 EQU 表达式 { ，类型 }

用于为程序中的常量、标号等定义一个等效的字符名称，类似于 C 语言中的# define 。

其中 EQU 可用 * 代替。

标号名称：为常量名。

表达式：寄存器的地址值、程序中的标号、32 位地址常量、32 位常量，当表达式为 32 位的常量时，可以指定表达式的数据类型，可以有以下三种类型：CODE16 、 CODE32 和 DATA。

例如：EQU 的使用

```
X    EQU 45                  @即#define X 45，必须顶格
Y    EQU 64
stack_top EQU 0x30200000，CODE32
AREA Example,CODE,READONLY
CODE32
ENTRY
Start
LDR SP,=stack_top            @stack_top 内的值 0x30200000 是地址,=stack_top 是取 stack_top 常量地址(即指针的指针)
MOV R0,#X                    @将 X 替换为 45
STR R0,[SP]                  @将 R0 内的 45 存入到 SP 所指向的内存中（SP 此时是指针的指针）
```

```
MOV R0,#Y
LDR R1,[SP]                    @从内存中读取数据到 R1 内
ADD R0,R0,R1
STR R0,[SP]
END
```

注：X,Y,stack_top 为标号，必须顶格写，大多写在代码段外面

2) 定义变量

常量：数字常量,有三种表示方式：十进制数、十六进制数、字符串常量、布尔常量(如 testno SETS {FALSE})

变量：数字变量、逻辑变量、字符串变量

1) GBLA、GBLL、GBLS 定义全局变量

格式：GBLA(GBLL、GBLS) 全局变量名

GBLA 伪指令用于定义一个全局的数字变量，并初始化为 0；

GBLL 伪指令用于定义一个全局的逻辑变量，并初始化为 F (假)；

GBLS 伪指令用于定义一个全局的字符串变量，并初始化为空；

由于以上三条伪指令用于定义全局变量，因此在整个程序范围内变量名必须唯一。

例如：全局变量的定义与赋值

```
GBLA count                     @定义全局变量
count SETA 2                   @给全局变量赋值为 2，必须顶格
AREA Example,CODE,READONLY
CODE32
ENTRY
Start
MOV R0,#count                  @将 count 内的值写入 R0 内
ADD R0,R0,#2
B Start
END
```

注： 在赋值过程中，全局变量名必须顶格写，全局变量常在代码段外定义和赋值

例如：变量与内存地址

```
GBLA globv
globv SETA 23
AREA Example,CODE,READONLY     @声明代码段 Example
ENTRY                          @程序入口
Start
LDR R0,=globv                  @golbv 是全局变量，将内存地址读入到 R0 内
LDR R1,[R0]                    @将内存数据值读入到 R1 内
ADD R1,R1,#2
STR R1,[R0]                    @将修改后数据再赋给变量
```

```
    MOV R0,#0
    OVER
    END
```

注：#取变量值 =取变量地址 [R0] 读取 R0 内地址所指向的数据值

2) LCLA、LCLL 、LCLS 定义局部变量

格式：LCLA (LCLL 或 LCLS) 局部变量名

LCLA 伪指令用于定义一个局部的数字变量，并初始化为 0；

LCLL 伪指令用于定义一个局部的逻辑变量，并初始化为 F (假)；

LCLS 伪指令用于定义一个局部的字符串变量，并初始化为空；

以上三条伪指令必须写在宏定义内，用于声明局部变量，宏结束，局部变量不再起作用

例如：

```
LCLA num                @声明一个局部的数字变量，变量名为 num
num  SETA 0xaa          @将该变量赋值为 0xaa
LCLL isOk               @声明一个局部的逻辑变量，变量名为 isOk
isOk SETL               @将该变量赋值为真
LCLS str1               @定义一个局部的字符串变量，变量名为 str1
str1 SETS "Testing"     @将该变量赋值为"Testing"
```

例如：局部变量的定义与赋值

```
MACRO
MOV_START               @宏名
LCLA x                  @定义局部变量
LCLA y
x SETA 12               @必须顶格写
y SETA 24
    MOV R0,#2
    MOV R1,#3
    ADD R0,R0,R1
MEND
AREA Example,CODE,READONLY      @声明代码段 Example
    ENTRY                       @程序入口
Start
    MOV_START
    MOV  R0,#0
OVER
END
```

注： 在赋值过程中，局部变量名必须顶格写，局部变量必须在宏定义内使用

3) SETA、SETL 和 SETS 用于给一个已经定义的全局变量或局部变量赋值

SETA 伪指令用于给一个数学变量赋值；

SETL 伪指令用于给一个逻辑变量赋值；

SETS 伪指令用于给一个字符串变量赋值；

其中，变量名为已经定义过的全局变量或局部变量，表达式为将要赋给变量的值。

4) 变量代换 $

$在数字变量前，将变值转换为十六进制字符串

$在逻辑变量前，将变量转换为真或假

$在字符串变量前,替换后面变量的字符串

例如：

```
LCLS  Y1                    @定义局部字符串变量 Y1 和 Y2
LCLS  Y2
Y1    SETS "WORLD!"
Y2    SETS "LELLO,$Y1"   @将字符串 Y2 的值替换$Y1，形成新的字符串
```

5) 定义一个寄存器(RN)

格式：名称 RN 表达式

RN 伪指令用于给一个寄存器定义一个别名。

例如：

```
Temp RN R0 ； 将 R0 定义一个别名 Temp
```

6) 定义寄存器列表(RLIST)

格式：名称 RLIST { 寄存器列表 }

用于对一个通用寄存器列表定义名称，使用该伪指令定义的名称可在 ARM 指令 LDM/STM 中使用。在 LDM/STM 指令中，寄存器列表中的寄存器访问次序总是先访问编号较低的寄存器，再访问编号较高的寄存器，而不管寄存器列表中各寄存器的排列顺序。

例如：

```
RegList RLIST {R0-R5 ， R8 ， R10}        @将寄存器列表名称定义为 RegList，用于多寄存器寻址（后面详解）
```

7) 定义协处理器寄存器(CN)

格式：名称 CN 协处理器的寄存器编号

```
例如：Power CN 6                          @将协处理器的寄存器 6 名称定义为 Power
```

8) 定义协处理器(CP)

格式: 名称 CP 协处理器名

```
例如：Dmu CP 6 ;将协处理器 6 名称定义为 Dmu
```

9) 定义浮点或精度寄存器(DN,SN,FN)

格式：名称 DN 双精度寄存器编号 ;DN 为双精度 VFP 寄存器定义名称

格式：名称 SN 单精度寄存器编号 ;SN 为单精度 VFP 寄存器定义名称

格式：名称 FN 浮点寄存器编号 ;FN 为浮点寄存器定义名称

例：

```
height DN  6  ;将 VFP 双精度寄存器 6 名称定义为 height
width  SN  16 ;将 VFP 单精度寄存器 16 名称定义为 width
```

height FN 6 ;将浮点寄存器 6 名称定义为 height

3. 数据定义伪操作（申请内存）

数据定义伪指令用于为特定的数据分配存储单元，同时可完成已分配存储单元的初始化。

1） 按类型分配内存

格式：标号 伪指令 表达式

标号

表达式：初始化的值，表达式可以为程序标号或字符、数字表达式

伪指令如下：

DCB 用于分配一片连续的字节存储单元（字符数组），可用 =号代替

Str DCB "This is a test!" @分配一片连续的字节存储单元并初始化。

DCW(或 DCWU) 用于分配一片连续的半字存储单元(16 位 短整型数组)，DCW 半字对齐，DCWU 不严格半字对齐。

DataTest DCW 1,2,3

DCD(或 DCDU) 用于分配一片连续的字存储单元(32 位，整型数组),DCD 可用 &代替，DCD 字对齐的

DataTest DCD 4,5,6

DCFD(或 DCFDU) 为双精度的浮点数分配一片连续的字存储单元,每个双精度的浮点数占据两个字单元。

FDataTest DCFD 2E115,-5E7

DCFS(或 DCFSU) 为单精度的浮点数分配一片连续的字存储单元,每个单精度的浮点数占据一个字单元。

FDataTest DCFS 2E5,-5E-7

DCQ(或 DCQU) 用于分配一片以 8 个字节为单位的连续存储区域(每 8 字节为一个数据的数组)

DataTest DCQ 100 @分配一片连续的存储单元并初始化为指定的值 100。

2） 申请连续内存

申请一个连续内存（SPACE)

用于分配一片连续的存储区域并初始化为 0,可用%代替

格式: 标号 SPACE 表达式

表达式为要分配的字节数

例如：DataSpace SPACE 100 @分配连续 100 字节的存储单元并初始化为 0 。

声明一个数据缓冲池的开始(LTORG)

通常，把数据缓冲池放在代码段的最后面，下一个代码段之前，或 END 之前

例如：

```
AREA Example,CODE,READONLY      @声明代码段 Example
    ENTRY                       @程序入口
Start
    BL funcl
    Funcl
    LDR R0,=0x12345678
    ADD R1,R1,R0
```

```
    MOV PC,LR
    LTORG                                @定义缓冲池 0x12345678 LTORG 根据 LDR 确定内存地址
    data SPACE 4200     ;从当前位置开始分配 4200 字节内存
END
```

定义一个结构化的内存表首地址（MAP）

格式: MAP 表达式 { , 基址寄存器 }
用于定义一个结构化的内存表的首地址。 可用 ^ 代替。
表达式可以为程序中的标号或数学表达式，基址寄存器为可选项。
** 当基址寄存器选项不存在时，表达式的值即为内存表的首地址，
** 当该选项存在时，内存表的首地址为表达式的值与基址寄存器的和。
例如：

```
Datastruc   SPACE    280              @分配 280 个字节单元
MAP         Datastruc                 @内存表的首地址为 Datastruc 内存块
```

定义一个结构化内存表的数据域（FILED）

用于定义一个结构化内存表中的数据域。可用#代替
格式：标号 FILED 表达式
FIELD 伪指令常与 MAP 伪指令配合使用来定义结构化的内存表。表达式的值为当前数据域所占的字节数。
标号为数据域（字段、成员变量）名
** MAP 伪指令定义内存表的首地址，
** FIELD 伪指令定义内存表中的各个数据域，并可以为每个数据域指定一个标号供其他的指令引用。

内存首地址 (MAP)

MAP 伪指令通常与 FIELD 伪指令配合使用来定义结构化的内存表。
例如：

```
Datastruc   SPACE    280              @分配 280 个字节单元
MAP         Datastruc                 @内存表的首地址为 Datastruc 内存块
consta      FIELD    4                @字段 consta 长度 4 字节，相对地址 0
constab     FIELD    4                @字段 constab 长度 4 字节，相对地址 4
x           FIELD    8                @字段 x 长度 8 字节，相对地址 8
y           FIELD    8                @字段 y 长度 8 字节，相对地址 16
string      FIELD    256              @字段 string 长度 256 字节，相对地址 24
```

LDR R6，[R9,consta] @引用内存表中的数据域

注意：MAP 伪操作和 FIELD 伪操作仅仅是定义数据结构，他们并不实际分配内存单元，而 SPACE 用于分配内存

4. 汇编控制伪操作

用于控制汇编程序的执行流程，常用的汇编控制伪指令包括以下几条：

(1) IF 逻辑表达式...ELSE...ENDIF 条件控制

(2) WHILE 逻辑表达式 ...WEND 循环控制

例如：条件编译

```
AREA Example,CODE,READONLY
CODE32
Data_in *  100              @定义标号 Data_in 的值为 100 在 ENTRY 入口之前
GBLA count                  @定义全局变量
count SETA 20
   ENTRY
Start
   IF count < Data_in       @条件编译
      MOV R0,#3
   ELSE
      MOV R1,#24
   ENDIF
      MOV R1,#12
      ADD R0,R0,R1
END
```

例如：循环编译

```
GBLA Counter            @声明一个全局的数学变量，变量名为 Counter
Counter SETA 3          @由变量 Counter 控制循环次数
……
WHILE Counter < 10
   指令序列
         IF continue
 MEXIT                  @退出宏
         ENDIF
WEND
```

5. 其他常用的伪指令

1) AREA

格式：AREA 段名 属性 1 ， 属性 2 ， ……

AREA 伪指令用于定义一个代码段或数据段。其中，段名若以数字开头，则该段名需用 " | " 括起来，如 |1_test| 。

属性字段表示该代码段(或数据段)的相关属性，多个属性用逗号分隔开

— CODE 属性：用于定义代码段，默认为 READONLY

— DATA 属性：用于定义数据段，默认为 READWRITE

— READONLY 属性：指定本段为只读，代码段默认为 READONLY

— READWRITE 属性：指定本段为可读可写，数据段的默认属性为 READWRITE

— ALIGN 属性：使用方式为 ALIGN 表达式。在默认时， ELF (可执行连接文件)的代码段和数据段是按字对齐的

— COMMON 属性：定义一个通用的段，不包含任何的用户代码和数据。各源文件中同名的 COMMON 段共享同一段存储单元

一个汇编语言程序至少要包含一个段，当程序太长时，也可以将程序分为多个代码段和数据段。

例如：

```
AREA Init， CODE， READONLY        @该伪指令定义了一个代码段，段名为 Init，属性为只读
```

2) ENTRY

格式：ENTRY

用于指定汇编程序的入口点。一个源文件里最多只能有一个 ENTRY (可以没有)

3) END

格式：END

用于通知编译器已经到了源程序的结尾。

4) CODE16、CODE32

格式：CODE16 (或 CODE32)

CODE16 伪指令通知编译器，其后的指令序列为 16 位的 Thumb 指令

CODE32 伪指令通知编译器，其后的指令序列为 32 位的 ARM 指令

在使用 ARM 指令和 Thumb 指令混合编程的代码里，可用这两条伪指令进行切换

例如：

```
AREA Init， CODE， READONLY
CODE32                      @通知编译器其后的指令为 32 位的 ARM 指令
LDR R0， = NEXT + 1         @将跳转地址放入寄存器 R0
BX R0                       @程序跳转到新的位置执行，并将处理器切换到 Thumb 工作状态
……
CODE16                      @通知编译器其后的指令为 16 位的 Thumb 指令
NEXT LDR R3，=0x3FF
……
END                         @程序结束
```

5) EXPORT(或 GLOBAL)

格式：EXPORT 标号

export 伪指令用于在程序中声明一个全局的标号，该标号可在其他的文件中引用。

6) IMPORT

格式：IMPORT 标号

IMPORT 伪指令用于通知编译器要使用的标号在其他的源文件中定义，但要在当前源文件中引用，如果当前源文件实际并未引用该标号，该标号也会被加入到当前源文件的符号表中。

7) EXTERN

格式：EXTERN 标号

EXTERN 伪指令用于通知编译器要使用的标号在其他的源文件中定义，但要在当前源文件中引用，如果当前源文件实际并未引用该标号，该标号就不会被加入到当前源文件的符号表中。

8) GET(或 INCLUDE)

格式：GET 文件名

GET 伪指令用于将一个源文件包含到当前的源文件中，并将被包含的源文件在当前位置进行汇编处理。汇编程序中常用的方法是在某源文件中定义一些宏指令，用 EQU 定义常量的符号名称，用 MAP 和 FIELD 定义结构化的数据类型，然后用 GET 伪指令将这个源文件包含到其他的源文件中。使用方法与 C 语言中的 "include" 相似。GET 伪指令只能用于包含源文件，包含目标文件需要使用 INCBIN 伪指令

例如：

```
AREA Init， CODE， READONLY
GET a1.s                    @通知编译器当前源文件包含源文件 a1.s
GET C：\a2.s                @通知编译器当前源文件包含源文件 C：\ a2.s ……
END
```

9) INCBIN

格式：INCBIN 文件名

INCBIN 伪指令用于将一个目标文件或数据文件包含到当前的源文件中

例如：

```
AREA Init， CODE， READONLY
INCBIN a1.dat               @通知编译器当前源文件包含文件 a1.dat
INCBIN C：\a2.txt           @通知编译器当前源文件包含文件 C：\a2.txt……
END
```

10) ROUT

格式：｛名称｝ROUT

ROUT 伪指令用于给一个局部变量定义作用范围。

在程序中未使用该伪指令时，局部变量的作用范围为所在的 AREA，而使用 ROUT 后，局部变量的作为范围为当前 ROUT 和下一个 ROUT 之间。

11) ALIGN

格式：ALIGN｛表达式｛，偏移量｝｝

ALIGN 伪指令可通过添加填充字节的方式，使当前位置满足一定的对其方式

表达式的值用于指定对齐方式，可能的取值为 2 的幂，如 1、2、4、8、16 等。

偏移量也为一个数字表达式，若使用该字段，则当前位置的对齐方式为：2 的表达式次幂＋偏移量。

例如：

```
AREA Init， CODE， READONLY， ALIEN = 3          @指定后面的指令为 8 字节对齐。
```

6. ARM 汇编伪指令(读取内存地址)

1） ADR 及 ADRL

将 PC 相对偏移的地址或基于寄存器相对偏移的地址值读取到寄存器中

格式：ADR(ADRL) 寄存器,地址表达式

ADR 小范围的地址读取伪指令，ADRL 中等范围的地址读取伪指令

例如：查表

```
ADR     R0,D_TAB                    @加载转换表地址
LDRB    R1,[R0,R2]                  @使用 R2 作为参数，进行查表
……
D_TAB
DCB    0xC0, 0xF9, 0xA4, 0xB0, 0x99, 0x92
```

LDR

用于加载 32 位立即数或一个地址值到指定的寄存器，大范围的地址读取伪指令。 LDR 通常都是作加载指令，但是它也可以作伪指令。作用是装载一个 32bit 常数和一个地址到寄存器。

格式：LDR 寄存器,=地址表达式

```
COUNT EQU       0x56000054          @COUNT 是一个变量，地址为 0x56000054。
LDR       R1,=COUNT                 @将 COUNT 这个变量的值（地址），也就是 0x56000054 放到 R1 中
MOV       R0,#0
STR       R0,[R1]                   @是一个典型的存储指令，将 R0 中的值放到以 R1 中的值为地址的存储单元去，这三条指令是为了完成对变量 COUNT 赋值。
```

2） NOP

空操作伪指令，可用于延时操作

例如：延时子程序

```
Delay
    NOP       ;空操作
    NOP
    NOP
    SUBS    R1,R1,#1    ;循环次数减 1
    BNE     Delay
    MOV     PC,LR
```

八、 C 语言与汇编混合编程

完全使用汇编语言来编写程序会非常的繁琐，通常，只是使用汇编程序来完成少量必须由汇编程序才能完成的工作，而其它工作则由 C 语言程序来完成。

1. ATPCS 规则

混合编程中，双方都须遵守 ATPCS 规则。这些基本规则包括：子程序调用过程中寄存器的使用规则、数据栈的使用规则和参数的传递规则。

1） 寄存器使用规则

- 寄存器： R4-R11 用来保存局部变量

- R0-R3 （a1-a4）用于保存参数/返回结果/scratch（临时寄存器）
- R4-R11（v1-v8）用于保存 ARM 状态局部变量
- R12（IP） 子程序内部调用的 scratch
- R13（SP） 数据栈指针
- R14（LR） 连接寄存器
- R15（PC） 程序计数器
- R7 又可称为 wr 用于 Thumb 状态工作寄存器
- R9 又可称为 sb 在支持 RWPI 的 ATPCS 中为静态基址寄存器
- R10 又可称为 sl 在支持 PWPI 的 ATPCS 中为数据栈限制指针
- R11 又可称为 fp 用于帧指针

2） 数据栈使用规则

ATPCS 标准规定，数据栈为 FD（满递减类型），并且对数据栈的操作是 8 字节对齐。在进行出栈和入栈操作，则必须使用 ldmfd 和 strnfd 指令（或 ldmia 和 stmdb)

3） 参数的传递规则

参数：参数小于等于 4，用 R0-R3 保存参数，参数多于 4，剩余的传入堆栈

函数返回：结果为 32 位整数，通过 R0 返回

结果为 64 位整数，通过 R0，R1 返回

对于位数更多的结果，通过内存传递

例如：参数传递及结果返回(r0-r3 做参数，r0 做返回值)

```
AREA Example,CODE,READONLY                @声明代码段 Example
    ENTRY                                 @程序入口
Start
    MOV  R3,#4                            @设置实参,将参数写入 R0-R3
    MOV  R2,#3
    MOV  R1,#2
    MOV  R0,#1
    BL   func1                            @调用子程序 ADD_SUM
 B    OVER                                @跳转到 OVER 标号处，进入结尾
func1
 ADD  R0,R0,R1                            @实现两数相加
 ADD  R0,R0,R2
 ADD  R0,R0,R3
 MOV  PC,LR                               @子程序返回，R0 内为返回的结果
OVER
END
```

相当于如下 C 语言：

```
int func1(int a,int b,int c,int d){
    return a+b+c+d;
```

```
}
int main(){
   func1(1,2,3,4);
}
```

例如：多于 4 个参数，前 4 个通过寄存器 R0-R3 传递，其它参数通过数据栈传递

```
AREA Example,CODE,READONLY                @声明代码段 Example
    ENTRY                                 @程序入口
Start
STMFD SP!,{R1-R4,LR}                      @先将 R1-R4，及 LR 内原有数据压入栈，需要使用这五个寄存器
MOV  R0,#1                                @准备好 7 个寄存器存入 7 个数据 LR,IP,R4 作临时寄存器使用
MOV  IP,#2
MOV  LR,#3
MOV  R4,#4
MOV  R1,#5
MOV  R2,#6
MOV  R3,#7
STMFD SP!,{R1-R3}                         @先将 R1-R3 数据从前向后入栈，然后将 IP,LR,R4 内的数据装入 R1-R3
MOV  R3,R4                                @其中 IP,LR,R4 是临时使用的寄存器
MOV  R2,LR
MOV  R1,IP
BL   func1                                @调用子程序 funcl R0 是返回结果
    LDMFD SP!,{R1-R4,PC}                  @从栈中取出最初的数据，恢复原始值
B    OVER                                 @跳转到 OVER 标号处，进入结尾
func1
     STMFD SP!,{R4,LR}                    @当调用函数时，LR 和 R4 都已发生了变化，其中 LR 是指令地址所以也压入栈
LDR  R4,[SP,#0x10]                        @目前共压入 5 个数据，每一个数据占两个字节，当前栈顶偏移 10 为前 5 个数据 7
ADD  LR,SP,#8                             @将前第 4 个数据的地址(栈项+偏移)赋给 LR
LDMIA LR,{IP,LR}                          @连续将 LR 地址处的两个数据取出写入 IP 和 LR 内，从右向左写，LDMIA 即出栈指令
ADD  R0,R0,R1                     @从此行开始相当于 return a+b+c+d+e+f+g;
ADD  R0,R0,R2
ADD  R0,R0,R3
```

```
    ADD  R0,R0,IP
    ADD  R0,R0,LR
    ADD  R0,R0,R4
    LDMFD SP!,{R4,PC}                    @从栈内取数据加载入 R4 和 PC,PC 跳转回主函数
    OVER
    END
```

相当于如下 C 语言：

```
int func1(int a,int b,int c,int d,int e,int f,int g){
    return a+b+c+d+e+f+g;
}
int main(){
 int a=1,b=2,c=3,d=4,e=5,f=6,g=7;
 func1(a,b,c,d,e,f,g);
}
```

九、 C 和 ARM 汇编程序间的相互调用

1. 汇编程序调用 C 子程序

为保证程序调用时参数正确传递，必须遵守 ATPCS。

在 C 程序中函数不能定义为 static 函数。在汇编程序中需要在汇编语言中使用 IMPORT 伪操作来声明 C 子函数

@C 语言程序代码

```
int sum5(int a, int b ,int c, int d)
{
return (a+b+c+d);
}
```

@汇编代码

```
    AREA Example,CODE,READONLY                @声明代码段 Example
    IMPORT sum5
        ENTRY                                 @程序入口
    Start
    MOV  R3,#4                                @设置实参,将参数写入 R0-R3
    MOV  R2,#3
    MOV  R1,#2
    MOV  R0,#1
    BL   sum5                                 @调用子程序 sum5
    B    OVER                                 @跳转到 OVER 标号处，进入结尾
    OVER
    END
```

2. 汇编程序访问全局 C 变量

汇编程序中可以通过 C 全局变量的地址来间接访问 C 语言中定义的全局变量在编编程序中用 IMPORT 引入 C 全局变量，该 C 全局变量的名称在汇编程序中被认为是一个标号。通过 ldr 和 str 指令访问该编号所代表的地址

@C 语言程序代码

```
int i=3;
int sum5(int a, int b ,int c, int d)
{
 return (a+b+c+d+i);
}
```

@汇编代码

```
AREA Example,CODE,READONLY          @声明代码段 Example
    IMPORT sum5
    IMPORT i
ENTRY                               @程序入口
Start
    LDR  R1,I                       @将 i 读入 R1 内
    MOV  R0,#2
    ADD  R0,R0,R1
    STR  R0,I                       @将寄存器值写入 i 内
    MOV  R3,#4                      @设置实参,将参数写入 R0-R3
    MOV  R2,#3
    MOV  R1,#2
    MOV  R0,#1
    BL   sum5                       @调用子程序 ADD_SUM
    B    OVER                       @跳转到 OVER 标号处，进入结尾
OVER
END
```

在 C 语言中调用汇编子程序为保证程序调用时参数的正确传递，在汇编程序中需要使用 EXPORT 伪操作来声明汇编子程序，同时在 C 语言中使用 extern 扩展声明汇编子程序。

@汇编代码

```
EXPORT func1                        @func1 为子函数名
    AREA Example,CODE,READONLY      @声明代码段 Example func1
    ADD  R0,R0,R1                   @实现两数相加
    ADD  R0,R0,R2
    ADD  R0,R0,R3
    MOV  PC,LR                      @子程序返回，R0 内为返回的结果
END
```

@C 语言程序代码

```
extern int func1(int a,int b,int c,int d);
int main(int argc,char **argv)
{
    int a=1,b=2,c=3,d=4;
    int z=func1(a,b,c,d);
    printf("%d",z);
    return 0;
  }
```

3. 在 C 语言中调用汇编全局变量

汇编中用 DCD 为全局变量分配空间并赋值，并定义一个标号代表该存储位置。在汇编中用 EXPORT 导出标号（这个标号就是全局变量），在 C 程序中用 extern 扩展声明名该变量。

例如：

@汇编代码

```
EXPORT func1
EXPORT tmp
AREA Example,CODE,READONLY          @声明代码段 Example ， tmp 为全局变量名
DCD 0x0005                          @全局变量创建内存空间及赋初值，func1 为子函数名
ADD  R0,R0,R1                       @实现两数相加
ADD  R0,R0,R2
ADD  R0,R0,R3
MOV  PC,LR                          @子程序返回，R0 内为返回的结果
END
```

@C 代码

```
extern int func1(int a,int b,int c,int d);
extern int tmp;
int main(int argc,char **argv)
{
    int a=1,b=2,c=3,d=4;
    int z=func1(a,b,c,tmp);
    printf("%d",z);
    return 0;
}
```

4. 在 C 语言中内嵌汇编

有些操作 C 语言程序无法实现，如改变 CPSP 寄存器值，初始化堆栈指针寄存器 SP 等，这些只能由汇编来完成。但出于编程简洁等一些因素，有时需要在 C 源代码中实现上述操作，此时就需要在 C 中嵌入少量汇编代码。内嵌的汇编不能直接引用 C 的变量定义，必须通过 ATPCS 进行，语法格式如下：

```
__asm {
@内嵌汇编
```

```
    }
    例如：在 C 语言中嵌入汇编
    int f(){ @C 函数
    __asm{ @内嵌汇编，禁用中断例子
        MRS R0,CPSR
        ORR R0,R0,#0x80
        MSR CPSR_c,R0
    }

}
    int main(int argc,char **argv){
    int a;
    int z=f(a);
    printf("%d",z);
    return 0;
}
```

- 出于完整性考虑，内嵌汇编相对于一般汇编的不同特点如下：
- 操作数可以是寄存器、常量或 C 表达式。可以是 char、short、或 int 类型，而且是无符号数进行操作
- 常量前的#号可以省略
- 只有指令 B 可以使用 C 程序中的标号,指令 BL 不可以使用
- 不支持汇编语言中用于内存分配的伪操作
- 内嵌汇编不支持通过“.”指示符或 PC 获取当前指令地址
- 不支持 LDR Rn,=expression 伪指令，而使用 MOV Rn,expression 指令向寄存器赋值
- 不支持标号表达式
- 不支持 ADR 和 ADRL 伪指令
- 不支持 BX 和 BLX 指令
- 不可以向 PC 赋值
- 使用 0x 前缀替代 &表示十六进制数
- 不使用寄存寻址变量
- ldm 和 stm 指令的寄存器列表只允许物理寄存器
- 必须小心使用物理寄存器，如 R0-R3,LR 和 PC