# 第6章ARM 指令系统

本章讨论 ARM 处理器所支持的机器指令。通常计算机系统中所有机器指令的集合，被称作指令系统。机器指令是用二进制位串来表示的，表示一条特定指令的二进制位串（比特串）称为指令字，简称指令。指令字需要包含操作码、操作数（可能有多个）和操作数地址等字段，其中指令操作码不仅需要指明指令的操作类型，还要指明操作数的类型和寻址方式等。这些指令字的相关信息在计算机指令集中予以规定。

## 6.1 ARM 处理器指令集概述

如前章节所述，ARM 公司的处理器经历了多次版本演进和更新，不同时期处理器对指令集的支持存在较大差异。目前（2019 年），ARM 公司将其不同系列处理器所支持的指令集体系结构统一为三个：A64、A32 和 T32。A32（过去称为 ARM 指令集）和 A64 的指令长度固定为 32 比特，而 T32（过去称为 Thumb2 指令集）的指令长度既有 16 比特，也有 32 比特，是一种混合长度的指令集。依据 ARM 公司各个微处理器体系结构版本的描述，ARMv5、ARMv6-M、ARMv7、ARMv8 以及截止 2019 年 12 月的 ARMv8.1-M 版本的处理器所支持的指令集如表 6.1 所示。

表 6.1 ARM 各处理器版本支持的指令集体系结构

| 微架构 | 支持的指令集架构 | | | 重要特性 |
|---|---|---|---|---|
| | A64 | A32 | T32 | |
| ARMv8.1-M | | | 是 | 支持矢量扩展（MVE） |
| ARMv8-A | 是 | 是 | 是 | 支持虚拟内存管理（VMSA[1]） |
| ARMv8-R | | 是 | 是 | 支持内存保护管理（PMSA[2]） |
| ARMv8-M | | | 是 | 可选支持内存保护管理（PMSA） |
| ARMv7-A | | 是 | 是 | 支持虚拟内存管理（VMSA） |
| ARMv7-R | | 是 | 是 | 支持内存保护管理（PMSA） |
| ARMv7-M | | | 是 | 支持 16 比特和 32 比特的 Thumb2 指令 |
| ARMv6-M | | | 是 | 支持 ARMv7-M 指令集的子集 |
| ARMv5 | | | | 支持 16 比特 Thumb 指令 |

从表 6.1 可看出，T32 是 ARM 较新版本处理器均支持的指令集。在不同处理器的设计中，除了对指令集体系结构的支持外，会依据处理器的用途增加指令集扩展（也称扩展指令）。常见的指令集扩展有：数字信号处理、浮点数运算、单指令多数据、矢量处理、安全增强、Java 加速等。表 6.2 列出了 Cortex-M 系列处理器所支持的指令集架构和指令集扩展。表中 Y 表示支持，N 表示不支持，P 表示部分支持，O 表示可选支持。受篇幅限制，表 6.2 中并未列出所有的指令名称，详细指令信息请参阅相应体系结构版本的 ARM 公司技术手册。

表 6.2 Cortex-M 系列处理器支持的指令集和扩展指令

[1]虚拟内存管理（VMSA，Virtual Memory System Architecture），基于基于内存管理单元（MMU）实现。

[2]内存保护管理（PMSA，Protected Memory System Architecture），基于内存保护单元（MPU）实现。

| 指令集 | 指令长度 (bits) | 指令 | m0 | m0+ | m1 | m3 | m4 | m7 | m23 | m33 |
|---|---|---|---|---|---|---|---|---|---|---|
| Thumb | 16 | 51 条基本指令[3] | Y | Y | Y | Y | Y | Y | Y | Y |
| | | CBNZ, CBZ | N | N | N | Y | Y | Y | Y | Y |
| | | IT | N | N | N | Y | Y | Y | N | Y |
| Thumb2 | 32 | BL, DMB, DSB, ISB, MRS, MSR | Y | Y | Y | Y | Y | Y | Y | Y |
| | | 94 条基本指令 | N | N | N | P | Y | Y | N | Y |
| | | SDIV, UDIV | N | N | N | Y | Y | Y | Y | Y |
| 数字信号处理 | 32 | 80 条基本指令 | N | N | N | N | Y | Y | N | O |
| 单精度浮点 | 32 | 25 条基本指令 | N | N | N | N | O | O | N | O |
| 双精度浮点 | 32 | 14 条基本指令 | N | N | N | N | N | O | N | N |
| TrustZone | 16 | BLXNS, BXNS | N | N | N | N | N | N | O | O |
| | 32 | SG, TT, TTT, TTA, TTAT | N | N | N | N | N | N | O | O |

从通用性和易学性角度出发，本章依据 ARMv7-M 微处理器架构版本的技术文档整理 T32 指令集架构的相关知识点。T32 指令集架构既和早期 16 比特的 Thumb 指令集有交集、又和 32 比特的 ARM 指令集有交集，也和 Thumb2 中 16 比特与 32 比特混合的指令集有交集。在 ARM 公司 ARMv7 版本后的文档中，逐渐不再对上述不同指令集进行区别，在 2019 年 12 月发布的第十个 ARMv8 版本中，已经**统一描述为 T32**[4]。故本章描述中也不对 Thumb 指令、Thumb2 指令和 ARM 指令做严格区分，在不至混淆的情形下，本章将指令描述为 T32 的 Thumb 指令（含 16 比特指令、32 比特指令）以及指令集扩展。

## 6.2 T32 指令格式

指令格式，即指令字所对应的二进制位串的格式定义。由于 ARM 的 T32 指令集既有 16 比特指令，又有 32 比特指令，故本节在介绍指令的基本字段后，分两种不同长度分别介绍 T32 的指令格式。毫无疑问，直接用二进制位串表示的机器指令可读性很差，习惯上常用汇编语言来表示机器指令。本节首先说明 T32 指令集中 16 比特指令和 32 比特指令的二进制格式（即机器码），然后以示例的形式阐述 T32 中机器指令对应的汇编语言格式。

### 6.2.1 16 比特指令二进制格式

T32 的指令由半字对齐（halfword-aligned）的序列构成。若为 16 比特 Thumb 的指令，则指令中含有一个半字；若为 32 比特 Thumb 指令，则指令包含两个半字。通过半字中最高五个比特区分是 16 比特指令还是 32 比特指令。图 6.1 所示最高五个比特含义解析如表 6.3，

[3] 此表中的“基本指令”指的是在多个版本处理器上被支持的指令。

[4] 需要注意的是，T32 在不同版本下指令会略有差异。例如，ARMv8 版本中 T32 所含指令与 ARMv7 版本 T32 所含指令是存在差异的。不过这些细节的差异并不会影响对指令系统原理的学习。

具体为，如果一个半字的最高五个比特（bits[15:11]）为如下三种情况则该半字是一个 32 比特指令的第一个半字：①0b11101，②0b11110，③0b11111。其他情况的半字均为 16 比特指令。

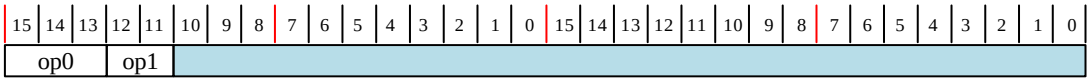


图 6.1　T32 指令顶层（Top Level）编码格式

根据上述规则，最高五个比特可以确定半字是对应一条完整的 16 比特指令还是一条 32 比特指令的高半字。如果是 16 比特的指令，则该二进制位串编码格式如图 6.2 所示，高六比特为操作码。

表 6.3　T32 指令的子类型（Subgroup）

| op0 | op1 | 指令类型（subgroup） |
|---|---|---|
| != 111 | - | 16 比特的 T32 指令编码 |
| 111 | != 00 | 32 比特的 T32 指令编码 |

图 6.2 所示高六比特的操作码定义了不同的指令类别，表 6.4 所示即为 16 比特指令可能的操作类别。


图 6.2　T32 中 16 比特指令编码格式

表 6.4 中操作码部分显示为“x”的位代表了任意值，即可以为“0”，也可以为“1”。不同位组合后会产生不同的指令。

表 6.4　16 比特 Thumb 指令操作码格式

| 操作码 | 指令类别 |
|---|---|
| 00xxxx | 基于立即数的指令如移位、加法、减法、比较和数据移动 |
| 010000 | 常规数据处理指令如位运算、移位、加法、减法、比较 |
| 010001 | 特殊的数据处理指令 |
| 01001x | 基于 PC 的载入指令 |
| 0101xx | 载入/保存（Load/Store）单个数据的指令 |
| 011xxx | |
| 100xxx | |
| 10100x | 产生基于 PC（PC-relative）的地址 |
| 10101x | 产生基于 SP（SP-relative）的地址 |
| 1011xx | 一些不容易分类的杂类指令 |
| 11000x | 保存多个寄存器的数到存储器区域 |
| 11001x | 从存储器区域载入多个数到多个寄存器 |
| 1101xx | 条件跳转指令 |
| 11100x | 无条件跳转指令 |

### 6.2.2　32 比特指令二进制格式

图 6.3 所示为 T32 指令集中 32 比特指令的二进制编码格式，可以看到，操作码被分成了三段：“op1”、“op2”、“op”。按照表 6.3 定义，“op1 == 00B”表示是该指令是 T32 指令

集中的 16 比特指令。

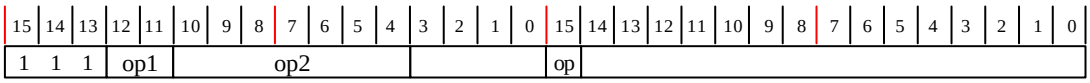


图 6.3 T32 中 32 比特指令编码格式

“op1 != 00B”时，根据“op1”、“op2”、“op”可以对 32 比特指令进行分类。表 6.5 给出了 ARMv7-M 中定义的 32 比特指令的类别。

表 6.5 32 比特 Thumb 指令操作码格式

| op1 | op2 | op | 指令类别/功能 |
|---|---|---|---|
| 01 | 00xx0xx | x | 载入或存储多个数 |
| 01 | 00xx1xx | x | 载入或存储双字，或者以独占方式执行载入或存储 |
| 01 | 01xxxxx | x | 常规数据处理如位运算、移位、加法、减法、比较等 |
| 01 | 1xxxxxx | x | 协处理器指令 |
| 10 | x0xxxxx | 0 | 使用移位后立即数的数据处理指令 |
| 10 | x1xxxxx | 0 | 使用未经过移位处理立即数的数据处理指令 |
| 10 | xxxxxxx | 1 | 分支控制指令和杂类指令 |
| 11 | 000xxx0 | x | 存储单个数据到存储器区域 |
| 11 | 00xx001 | x | 载入字节 |
| 11 | 00xx011 | x | 载入半字 |
| 11 | 00xx101 | x | 载入字 |
| 11 | 00xx111 | x | 未定义 |
| 11 | 010xxxx | x | 寄存器处理指令 |
| 11 | 0110xxx | x | 产生 32 比特结果的乘法、乘并累加指令 |
| 11 | 0111xxx | x | 产生 64 比特结果的乘法、乘并累加指令 |
| 11 | 1xxxxxx | x | 协处理器指令 |

### 6.2.3 T32 指令的汇编语法

如前所述，直接用二进制位串表示的机器指令可读性很差，习惯上用汇编语言来表示机器指令。在不同的设备中，汇编语言对应着不同的机器语言指令集，通过汇编过程转换成机器指令。特定的汇编语言和特定的机器语言指令集是一一对应的，不同平台之间不可直接移植。

ARM 早期设计中，由于不同版本的处理器所支持的指令集有较大差异，所用汇编语言也是不同的。ARMv7 之后，为了增强兼容性，提出了采用统一汇编语言（UAL，Unified Assembly Language）进行机器指令的描述。使用 UAL 后，16 比特的汇编指令和 32 比特的汇编指令可以无缝的出现在同一份代码中，只是其编码格式不同。

在定义一条机器指令的时候，需要将指令的功能、源操作数、目标操作数、操作数地址等信息予以明确。ARM 处理器中汇编指令的通用格式构成要素如下：

**<opcode>** ***[cond] [q] [S]*** **<Rd> , <Rn>** ***[, Oprand2]***

其中，<>内的参数是必选参数，而*[]*内参数是可参数。

- opcode，操作码，也称为助记符。指示指令需要执行的操作类型。指令系统的每一条指令都要规定一个操作码，来说明该指令应进行什么性质的操作，如进行加法、减法、乘法、

除法等。CPU 中的专门电路（指令译码）来解释每个操作码，并根据操作码控制 CPU 内部的计算电路单元的工作状态。

- cond：条件码（可选后缀），描述指令的执行条件。这是 ARM 处理器和其他很多其他处理器有较大区别的地方，在 ARM 处理器中，可以通过条件码指示在 APSR 寄存器的标志位满足特定条件时才执行指令。
- q，可选后缀，指令宽度选择，“.N”表示指令为 16 比特，“.W”表示指令为 32 比特。
- S，可选后缀，加上“S”，在指令执行完毕后自动更新 APSR 中的标志位的值。在 ARM 早期产品中，如 ARM7TDMI，几乎所有的数据操作指令都会更新 APSR[5]中的标志位；后期 ARM 处理器在采用 Thumb-2 技术同时支持 16 比特和 32 比特指令后，允许指令不更新 APSR 中的标志位。
- Rd，ARM 指令中的目标操作数，总是一个寄存器。
- Rn，存放第一源操作数寄存器，该操作数必须是寄存器。
- Oprand2，第二源操作数，不仅可以是寄存器，还能是立即数，而且能用经过偏移量计算的寄存器和立即数。

### 6.2.4 T32 的条件执行指令

T32 中多数 Thumb 指令可以根据应用程序状态寄存器[6]（APSR，Application Program Status Register）中的标志位决定当前指令是否被执行。这种在特定条件满足时才执行的指令被称作条件执行（Conditional execution）指令。APSR 寄存器的标志位定义如图 6.4 所示。

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| N | Z | C | V | Q | 保留 | | | | | | | GE[3:0] | | | | 保留 | | | | | | | | | | | | | | | |

图 6.4 应用程序状态寄存器的标志位定义

处理器执行指令的时候，其运算过程可能会改变 APSR 中的标志位。当指令助记符添加了后缀“S”时，指令运算会影响到 APSR 中的标志位。图 6.4 所示 APSR 寄存器的各标志位定义如下。

- N bit[31] 负数标志位。N == 1 表示上一次运算结果为负数，否则为正数或零。
- Z bit[30] 零标志位。Z == 1 表示上一次运算结果为零。
- C bit[29] 进位标志位。C == 1 表示上一次运算结果产生了进位。
- V bit[28] 溢出标志位。V == 1 表示上一次运算结果产生了溢出。
- Q bit[27] 饱和标志位。Q == 1 表示上一次运算结果产生了饱和操作。
- GE[3:0] bits[19:16]，DSP 扩展指令中 SIMD 类指令指示上一次运算结果信息。

表 6.6 列出了整数运算时条件码的含义，浮点数运算的定义与此相似，详细信息可参阅 ARM 公司相应版本的架构技术手册。在 Thumb 指令中，只要条件码不为“1110”，均为条件执行指令。

[5] 早期 ARM 处理器中，标志寄存器的名称是 CPSR（Current Program Status Register），后更名 APSR。

[6] 从 ARMv7 版架构开始，ARM 处理器中采用 PSR 整合了更早版本中 APSR、EPSR 和 IPSR 三个状态寄存器的位域，详见第 5 章。本章沿用 ARMv7-M 技术手册描述，不对 PSR 或 APSR 做描述上的区分。

表 6.6 整数运算时 T32 指令集的条件码（Condition codes）含义

| 条件码 | 助记符 | 含义 | 标志位取值 |
|---|---|---|---|
| 0000 | EQ | 等于（Equal） | Z == 1 |
| 0001 | NE | 不等于（Not equal） | Z == 0 |
| 0010 | CS | 产生了进位（Carry set） | C == 1 |
| 0011 | CC | 进位为零（Carry clear） | C == 0 |
| 0100 | MI | 负数（Minus, negative） | N == 1 |
| 0101 | PL | 整数或零（Plus, positive or zero） | N == 0 |
| 0110 | VS | 溢出（Overflow） | V == 1 |
| 0111 | VC | 没有溢出（No overflow） | V == 0 |
| 1000 | HI | 无符号比较大于（Unsigned higher） | C == 1 且 Z == 0 |
| 1001 | LS | 无符号比较小于等于（Unsigned lower or same） | C == 0 或 Z == 1 |
| 1010 | GE | 有符号比较大于等于（Signed greater than or equal） | N == V |
| 1011 | LT | 有符号比较小于（Signed less than） | N != V |
| 1100 | GT | 有符号比较大于（Signed greater than） | Z == 0 且 N == V |
| 1101 | LE | 无符号比较小于等于（Signed less than or equal） | Z == 1 或 N != V |
| 1110 | AL | 无条件执行（Always） | N/A |

### 6.2.5 T32 指令格式示例

如前所述，汇编语言所描述的机器指令往往具有如下的形式。依据指令的不同功能，操作数的数目可以是一个、两个或者更多个。

❒ **助记符 操作数 1, 操作数 2, …;注释**

助记符常采用英文单词的缩写，表 6.7 所示为几条常见指令的助记符。从表 6.7 可看出，了解助记符的英文描述非常有利于区分并记住不同的助记符。

表 6.7 T32 指令集中常见操作码的助记符含义中英文对照

| 助记符 | 指令功能英文描述 | 指令功能中文描述 |
|---|---|---|
| ADD | Add | 加法 |
| LDM | Load multiple registers | 将多个 32 比特数值存入多个寄存器 |
| LDR | Load register with word | 将 32 比特数值存入寄存器 |
| MOV | Move | 数据移动 |
| STM | Store multiple registers | 将多个寄存器的数值存入特定存储器区域 |
| STR | Store register word | 将一个寄存器的 32 比特数值存入存储器 |
| SUB | Subtract | 减法 |

下面以数据移动指令作为示例，分析 T32 指令集中指令的汇编语言描述方式和二进制位串之间的关系。例，如果要把寄存器 R1 中的数值装载到寄存器 R0 中，那么汇编指令格式如下。其中“MOV”为助记符，“R0”是目标源操作数，“R1”为源操作数；助记符和第一操作数之间需要放置一个空格，两个操作数之间需要使用英文的逗号“,”，英文的分号“;”后可书写该指令的注释。

❒ **MOV R0, R1 ;寄存器 R1 中的数值装载到寄存器 R0**

## 1. 不更新 APSR 的 MOV 指令

用户书写的“**MOV R0, R1**”指令经过汇编语言编译器（assembler）转换为二进制的位串。相同的汇编指令在不同的处理器中往往被编码为不同格式的二进制串，甚至同一处理器也允许多种格式的二进制编码方式。如在 ARMv7-M 版本中，允许 MOV 指令生成 T1 编码格式（如图 6.5 所示）或 T2 编码格式（如图 6.6 所示）或 T3 编码格式（如图 6.7 所示）等三种不同二进制格式，其中“Rd”为目标操作数，“Rm”为源操作数。

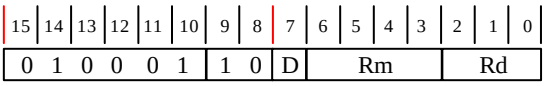


图 6.5 编码方式 T1（Encoding T1）的 16 比特 MOV 指令

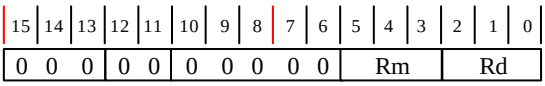


图 6.6 编码方式 T2（Encoding T2）的 16 比特 MOV 指令

T1、T2、T3 编码格式在二进制比特含义定义方面有格式差异。上述“**MOV R0, R1**”指令，如果按照图 6.5 所示的 T1 编码格式，目标操作数由比特(D:Rd)四个比特指定，源操作数由(Rm)的四个比特指定。

### ❒ T1 格式“0100 0110 0000 1000”

类似地，“**MOV R0, R1**”指令如果按照图 6.7 所示的 T3[7]编码格式，目标操作数由比特(Rd)四个比特指定，源操作数由(Rm)的四个比特指定。图 6.7 中比特“S”为可选后缀，“S==1”则在指令执行完毕后自动更新 APSR 中的标志位的值。由于“**MOV R0, R1**”指令中助记符“MOV”并未添加后缀“S”，故该指令所对应的二进制比特（机器指令）如下，其中目标操作数为 R0，故“Rd==0000”；源操作数为 R1，故“Rm==0001”。

### ❒ T3 格式“1110 1010 0100 1111 0000 0000 0000 0001”

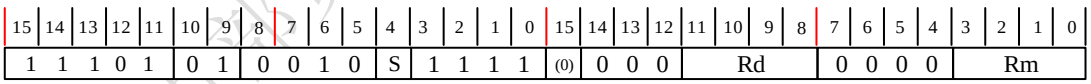


图 6.7 编码方式 T3（Encoding T3）的 32 比特 MOV 指令

T1、T2、T3 编码格式所得指令的长度和适用处理器也不同，T1 和 T2 格式的指令均为 16 比特，T1 格式在 ARMv6-M 和 ARMv7-M 版本的处理器中可支持，T2 格式则被所有支持 Thumb 指令的处理器中可被解析。与此同时，T3 格式的指令是 32 比特，ARMv6-M 版本不支持该格式。

需要注意的是，ARMv7-A/R/M 等不同版本中，除了 T1、T2、T3 编码格式，还支持 T4、A1、A2 编码格式。相同的汇编指令被汇编语言编译器转换为哪种二进制的位串，取决于需要支持的处理器具体型号和不同汇编软件工具内置的优化规则。

但如果在“**MOV R0, R1**”指令中使用了可选后缀[q]，那么会根据该后缀进行二进制指令宽度的选择。如“**MOV.W R0, R1**”指令就会被编码为 T3 格式，而“**MOV.N R0, R1**”指

[7] 事实上，“MOV R0, R1”指令只可能被转换为 T1 格式或 T2 格式，原因在下面“更新 APSR 的 MOV 指令”部分分析。

令就会被编码为 T1 格式。

### 2. 更新 APSR 的 MOV 指令

上面所讨论的“**MOV R0, R1**”指令，由于助记符“MOV”没有携带后缀“S”，表示该指令执行不会影响（更新）APSR 中的标志位。如果希望指令执行产生的运算结果影响 APSR 中的标志位（如记录是否溢出、是否为零等信息），需要使用助记符加后缀“S”形式，具体指令应为如下格式。

- **MOVS R0, R1 ;寄存器 R1 中的数值装载到寄存器 R0，更新 APSR**

MOV 指令的三种编码格式 T1、T2 和 T3 中，T1 格式默认不更新标志位，而 T2 格式默认更新标志位，T3 格式则显式地在二进制位串中定义了一个比特表示是否更新 APSR 标志位（如图 6.7 所示的“S”比特）。正是因为 T32 指令集中这样的默认规则，使得“**MOV R0, R1**”指令不会被编码为 T2 格式。更新 APSR 标志位的“**MOVS R0, R1**”指令可以被编码为 T2 格式或 T3 格式，其二进制位串分别如下所示。

- **T2 格式“0000 0000 0000 1000”**
- **T3 格式“1110 1010 0101 1111　0000 0000 0000 0001”**

注意：**并非所有的指令编码为 T1 格式都默认不更新标志位**。当 MOV 指令的源操作数为立即数的时候，T1 编码格式也可以更新标志位。汇编语言编译器在编译过程中会根据软件工具内置的优化规则做出选择。

### 3. 条件执行的 MOV 指令

如果用户需要在满足特定条件（APSR 的某些标志位符合要求）时才执行 MOV 指令，那么就需要使用条件码。依据表 6.6 所定义的条件码和后缀助记符的映射关系，如果需要在 APSR 的标志位“Z == 1”时才执行从寄存器 R1 到寄存器 R0 的数据移动操作，那么“**MOV R0, R1**”指令就需要修改为“**MOVEQ R0, R1**”指令。

回顾一下图 6.5 所示 T1 格式、图 6.6 所示 T2 格式或图 6.7 所示 T3 格式，我们会发现在三种不同二进制比特定义中，并没有条件码的四个比特出现。T32 指令集是通过另外一条指令来帮助实现“**MOVEQ R0, R1**”指令的，这条额外的指令是“IT”指令，“IT”意为“If Then”。“IT”指令允许跟随其后的最多四条指令是条件执行的，跟随在“IT”指令后面的几条指令被称作一个 IT 块（IT block）。

“IT”指令只能被编码为 T1 格式的二进制位串。如图 6.8 所示。其中“firstcond”字段表示跟随在“IT”指令后第一条指令的条件码，“mask”字段分别对应跟随在“IT”指令后的四条指令是否使能条件执行功能。

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | firstcond | | | | mask | | | |

图 6.8　编码方式 T1 的 16 比特 IT 指令

“IT”指令的汇编语法格式为：IT{x{y{z}}} cond。其中可选的后缀 x、y、z 的含义如下所示，汇编语言编译器会将后缀 x、y、z 转换为图 6.8 中的“mask”字段。

- cond 为 IT 块中第一条指令使用的条件码；

- x 指定 IT 块中第二条指令的是否执行的开关；
- y 指定 IT 块中第三条指令的是否执行的开关；
- z 指定 IT 块中第四条指令的是否执行的开关。

IT 块中的每一条指令必须指定一个条件后缀来选择相等或逻辑相反的情况下执行。即，后缀 x、y、z 的取值只能是“T”或“E”，“T”意为“Then”，“E”意为“Else”。由于 IT 块最多允许四条指令，故 x、y、z 的取值组合包含如下情况。

- IT，意为“If-Then”，下一条指令是条件执行的；
- ITT，意为“If-Then-Then”，下两条指令是条件执行的；
- ITE，意为“If-Then-Else”，下两条指令是条件执行的；
- ITTE，意为“If-Then-Then-Else”，下三条指令是条件执行的；
- ITTEE，意为“If-Then-Then-Else-Else”，下四条指令是条件执行的。

下面按照上面的规则，给出具体的示例。如果需要条件执行“**MOVEQ R0, R1**”指令，则与“IT”指令配合使用时，汇编代码如下所示。以下示例根据 ARM 公司 Cortex-M3 用户手册（Cortex™-M3 Devices Generic User Guide）改编。

```
IT EQ                ;随后的一条指令是条件执行的
MOVEQ R0, R1         ;当 APSR 中标志位“Z == 1”时执行，否则不执行
```

考虑有两条 MOV 指令需要条件执行的情形，如“**MOVEQ R0, R1**”和“**MOVEQ R2, R3**”，那么汇编代码如下所示。

```
ITT EQ               ;随后的两条指令是条件执行的
MOVEQ R0, R1         ;当 APSR 中标志位“Z == 1”时执行，否则不执行
MOVEQ R2, R3         ;当 APSR 中标志位“Z == 1”时执行，否则不执行
```

如果需要条件执行的两条 MOV 指令分别是“**MOVEQ R0, R1**”和“**MOVNE R2, R3**”，那么汇编代码如下所示。

```
ITE EQ               ;随后的两条指令是条件执行的
MOVEQ R0, R1         ;当 APSR 中标志位“Z == 1”时执行，否则不执行
MOVNE R2, R3         ;当 APSR 中标志位“Z == 0”时执行，否则不执行
```

下面这个例子更加复杂一些，有四条 MOV 指令需要条件执行。这四条指令依次是“**MOVEQ R0, R1**”和“**MOVNE R2, R3**”，那么汇编代码如下所示。各条指令的逻辑关系请读者自行分析。

```
ITTEE EQ             ;随后的四条指令是条件执行的
MOVEQ R0, R1         ;当 APSR 中标志位“Z == 1”时执行，否则不执行
MOVEQ R4, R5         ;当 APSR 中标志位“Z == 1”时执行，否则不执行
MOVNE R2, R3         ;当 APSR 中标志位“Z == 0”时执行，否则不执行
MOVNE R6, R7         ;当 APSR 中标志位“Z == 0”时执行，否则不执行
```

另外，如果用户书写的“IT”指令和随后的 IT 块中的指令冲突，汇编语言编译器在编译过程中会给出错误提示。

## 6.3 T32 指令集寻址方式

所谓寻址方式就是处理器根据指令中给出的地址信息来寻找操作数有效地址的方式。寻址包括两种情形：①确定当前指令中的**操作数地址**，称作操作数寻址或简称数据寻址；②确定下一条待执行指令的地址，常称作指令寻址。

ARM 处理器支持多种寻址方式（Addressing modes），分别是立即数寻址、寄存器寻址、寄存器移位寻址、寄存器间接寻址、变址寻址、多寄存器寻址（块拷贝寻址）、堆栈寻址、相对寻址等。依据操作数所在的不同位置，可以把这些寻址方式分为四个大的类别，如表 6.8 所示。

表 6.8　寻址方式的分类

| 操作数位置 | 可能的寻址方式 | 英文原称 |
|---|---|---|
| 内含在指令中 | 1) 立即数寻址 | Immediate addressing |
| 存放在寄存器中 | 2) 寄存器寻址 | Register addressing |
| | 3) 寄存器移位寻址 | Shifted register addressing |
| 存放在存储器数据区 | 4) 寄存器间接寻址 | Register addressing |
| | 5) 寄存器偏移寻址 | Offset addressing |
| | 6) 前变址寻址 | Pre-indexed addressing |
| | 7) 后变址寻址 | Post-indexed addressing |
| | 8) 多寄存器寻址 | multiple registers |
| | 9) 堆栈寻址 | SP addressing |
| 存放在存储器代码区 | 10) PC 相对寻址 | PC-related addressing (Literal) |

为了说明表 6.8 所示不同寻址方式的差异，本小节中会用到一些简单的汇编指令，所用到指令的助记符和功能描述参见表 6.7。

### 6.3.1　立即数寻址

立即数寻址也叫立即寻址，是一种特殊的寻址方式，操作数本身包含在指令中，只要取出指令也就得到了操作数。这个操作数叫做立即数，对应的寻址方式叫做立即寻址。如下三条指令均用到了立即数寻址。

- MOV R0,#66　　;将立即数 66 传送到寄存器 R0
- ADD R0, R0, #66　　;将立即数 66 与寄存器 R0 结果相加，结果保存在 R0
- SUB R0, R0, #0x33　　;寄存器 R0 数值减去立即数 0x33，结果保存在 R0

在立即数寻址中，要求立即数以“#”为前缀，对于十六进制表示的立即数，还要求在“#”后加上“0X”或“0x”。

受到指令长度的限制，T32 指令集中合法的立即数[8]需要满足一定的规则。①一个八位二进制数值及其循环移位得到的数值是合法的立即数。如，0x01FC 是合法的，把它写成二进制形式为：0001 1111 1100，可看出用 0xFE（二进制形式为：1111 1110）这个八位的数值在 16 位寄存器中循环左移一位就可以得到 0X01FC。但是 0x07FC 就是非法的立即数，因为

[8] 传统 ARM 处理器所使用的 32 位 ARM 指令中，要求 32 位立即数必须是一个“位图”数据：一个任意的 8 位立即数经过循环移位得到的数据。

它无法通过对一个八比特二进制数经过循环移位得到。②满足格式“0x00XY00XY”或“0xXY00XY00”或“0xXYXYXYXY”的数是合法的立即数，其中 X 和 Y 为 16 进制数字。

### 6.3.2 寄存器寻址

寄存器寻址（或称为寄存器直接寻址）就是把寄存器中的数值作为操作数，也称为寄存器直接寻址。这种寻址方式在各类微处理器经常被采用，执行效率高。以下指令用到了寄存器寻址。

- ADD R0, R1, R2 ;将 R1 和 R2 相加结果送入 R0

### 6.3.3 寄存器间接寻址

寄存器间接寻址就是把寄存器中存放的数值作为操作数地址，通过这个地址去取得操作数，操作数本身存放在存储器中。如下两条指令均用到了寄存器间接寻址。

- LDR R0, [R1] ;以寄存器 R1 的数值作操作数的地址，取得操作数后传送到 R0
- ADD R0, R1, [R2] ;以寄存器 R2 的数值作为地址取得操作数后与 R1 相加，结果存入 R0

### 6.3.4 寄存器移位寻址

寄存器移位寻址（Shifts applied to a register）是 ARM 指令集特有的寻址方式，在其他很多类型的处理器中并不支持。其寻址方式为：先由寄存器寻址得到操作数，再对该操作数进行移位操作后得到最终的操作数。如下两条指令均用到了寄存器偏移寻址。

- MOV R0, R2, LSL #3 ;R2 中的值左移 3 位，结果送入 R0
- MOV R0, R2, LSL R1 ;R2 中的值左移 R1 位，结果送入 R0

在 32 位的处理器中，移位操作可移动的位数最多为 32 位。ARM 处理器在寄存器偏移寻址的时候，可采用的移位操作包括如下几种方式。

- LSL：逻辑左移（Logical Shift Left），寄存器中字的低端空出的位补零。
- LSR：逻辑右移（Logical Shift Right），寄存器中字的高端空出的位补零。
- ASR：算术右移（Arithmetic Shift Right），移位过程中符号位不变，即如果源操作数是正数，则字的高端空出的位补零，否则补“1”。
- ROR：循环右移（Rotate Right），由字的低端移出的位填入字的高端空出的位。
- RRX：带扩展的循环右移（Rotate Right eXtended），操作数右移一位，高端空出的位用进位标志 C 的值来填充，低端移出的位填入进位标志位。

### 6.3.5 寄存器偏移寻址

可以对前述寄存器间接寻址进行拓展，操作数地址由一个寄存器中存放的数值与指令中给出的地址偏移量相加得到。这种寻址方式称作寄存器偏移寻址（Offset addressing）。其寻址过程为：将寄存器（即基址寄存器）中的基址（base address）与指令中给出的地址偏移量（offset）相加，得到一个新的地址，通过这个地址取得操作数。其汇编语法为“opcode Rd [<Rn>, <offset>]”。

通常把存放在寄存器中的地址信息称作基址，该寄存器称作**基址寄存器**。而指令中给出的地址偏移量存在如下三种不同的形式。

- 立即数，偏移量是一个立即数，该数值与基址寄存器相加或相减得到操作数的地址。
- 寄存器值，偏移量是另外一个寄存器中的数值，该数值与基址寄存器中数值相加或

相减得到操作数地址。

❒ 寄存器值移位（Scaled register），偏移量是另外一个寄存器中数值经过移位运算得到的数值，并与基址寄存器中数值相加或相减得到操作数地址。

下面以 LDR/STR 指令为例说明寄存器偏移寻址的寻址过程。LDR 指令的功能是从存储器特定位置读取 32 比特数值后存入指定的寄存器；STR 指令则与之相反，把指定寄存器内的数值保存到存储器的特定位置。如下指令使用了寄存器基址变址寻址方式。

```
❒ LDR R0, [R1, #4]      ;R1 中的值加 4 形成操作数的地址，取得的操作数送入 R0
❒ LDR R0, [R1, R2]      ;R1 中的值加上 R2 中的值形成操作数地址，取得的操作数送入 R0
❒ STR R0, [R1, #-4]     ;R1 中的数值减 4 作为操作数地址，把 R0 中的数值存放到这个地址中
```

考虑到保存偏移量信息的寄存器（变址寄存器）是可以做移位操作的，如前述寄存器移位寻址。那么，采用寄存器偏移寻址的 LDR 指令的典型汇编语法可存在如下三种形式。

```
❒ LDR <Rt>, [<Rn>{,#<imm>}]
❒ LDR <Rt>, [<Rn>,<Rm>]
❒ LDR <Rt>, [<Rn>, <Rm> {, LSL #<shift>}]
```

其中“<Rn>”表示基址寄存器，而“<offset>”表示偏移量，偏移量可以是：①5 位立即数（<imm5>）或 8 位立即数（<imm8>）或 12 位立即数（<imm12>）；②寄存器值<Rm>；③寄存器值移位，如<Rm>, LSL #<shift>，#<shift>表示移位的位数。

### 6.3.6 前变址寻址

前变址寻址（Pre-indexed addressing）建立在寄存器偏移寻址的基础之上。如前所述，形如“LDR R0, [R1, #4]”或“LDR R0, [R1, R2]”的指令中操作数的地址信息包含两个部分：基址和偏移。前变址寻址方式是指，在执行指令的时候自动把基址与偏移加和形成的**操作数地址写回到基址寄存器**中。由于基址寄存器中的数值和立即数偏移量的加和计算发生在寻址前，故称作前变址（Pre-indexed，有些资料中翻译为“前序”）。

这种寻址方式非常有利于一些通过循环处理一个数组数据的算法简化代码。其汇编语法为“opcode Rd [<Rn>,<offset>]!”，该语法与前述寄存器偏移寻址仅仅相差最后的“!”后缀。“!”后缀表示指令完成时是更新存放基地址的基址寄存器（写回）。

下面仍然以 LDR/STR 指令为例，对比寄存器偏移寻址和前变址寻址方式的差异。LDR 指令的功能是从存储器特定位置读取 32 比特数值后存入指定的寄存器；STR 指令则与之相反，把指定寄存器内的数值保存到存储器的特定位置。如下四条指令分别使用了寄存器偏移寻址、前变址寻址、寄存器偏移寻址、前变址寻址方式。

```
❒ LDR R0, [R1, #4]      ;R1 中的值加 4 形成操作数的地址，取得的操作数送入 R0
❒ LDR R0, [R1, #4]!     ;与上一条指令的区别：“!”表示指令执行后操作数地址存入 R1
❒ LDR R0, [R1, R2]      ;R1 中的值加上 R2 中的值形成操作数地址，取得的操作数送入 R0
❒ LDR R0, [R1, R2]!     ;与上一条指令的区别：“!”表示指令执行后操作数地址存入 R1
```

### 6.3.7 后变址寻址

后变址寻址（Post-indexed addressing）也是建立在寄存器偏移寻址的基础之上的。如前所述，形如“LDR R0, [R1, #4]”或“LDR R0, [R1, R2]”的指令中，用基址寄存器保存了操

作数的地址信息，还用立即数或者变址寄存器保存了一个偏移量。后变址寻址方式是指，在执行指令的时候，操作数地址从基址寄存器获取，指令执行后再将**操作数地址加上偏移量**生成一个新的地址，并将该新地址**写入基址寄存器**。

由于指令中的偏移量在存储器访问期间不会用到偏移，而是在数据传输结束后更新基址寄存器，故此模式称为后变址（Post-indexed，有些资料中翻译为“后序”）。其汇编语法为“opcode Rd [<Rn>],<offset>”，注意该语法格式与寄存器偏移寻址即前变址寻址非常相似。

下面仍然以 LDR/STR 指令为例，对比寄存器偏移寻址、后变址寻址、后变址寻址方式的差异。LDR 指令的功能是从存储器特定位置读取 32 比特数值后存入指定的寄存器；STR 指令则与之相反，把指定寄存器内的数值保存到存储器的特定位置。如下三条指令分别使用了寄存器偏移寻址、前变址寻址、后变址寻址方式。

- LDR R0, [R1, #4]　　;R1 中的值加 4 形成操作数的地址，取得的操作数送入 R0
- LDR R0, [R1, #4]!　　;与上一条指令的区别：“!”表示指令执行后操作数地址存入 R1
- LDR R0, [R1], #4　　;R1 中的值做操作数地址，取得操作数送入 R0，R1 中数值加 4

综合前面讨论过的寄存器偏移寻址、前变址寻址、后变址寻址方式。上述寻址方式中，操作数地址生成时均使用了基址寄存器的值，有的直接使用，有的在基址寄存器的值上加了一个偏移量。并且，有的会把基址寄存器数值进行更新（称为寄存器写回），有的不更新。三类存储器的访问方式特性对比如表 6.9 所示。

表 6.9　三类存储器的访问方式的对比

| 寻址模式 | 汇编语法 | 操作数地址 | 基址寄存器变化 |
|---|---|---|---|
| 偏移寻址 | [<Rn>, <offset>] | 基址加偏移量 | 无变化 |
| 前变址寻址 | [<Rn>, <offset>]! | 基址加偏移量 | 基址寄存器被更新为原基址寄存器数值加偏移量 |
| 后变址寻址 | [<Rn>], <offset> | 基址 | 基址寄存器被更新为原基址寄存器数值加偏移量 |

### 6.3.8　多寄存器寻址

在 T32 指令集中，有些指令可以从一块连续的存储器区域装载多个数据到多个寄存器中。此时采用一种特殊的寻址方式，这种寻址方式可以一次完成多个寄存器值的传送。LDM 是一条使用此寻址方式的指令，可以从连续的存储器区域装载多个数据。其典型汇编语法为“**LDM {addr_mode} <Rn>{!}, <registers>**”，其中<Rn>为基址寄存器，<registers>为需要载入数据的的寄存器集合，可选项{!}表示需要将修改后的地址写入基址寄存器<Rn>。可选后缀{addr_mode}可选择如下四种方式（Cortex-M3/M4 只支持其中 IA 和 DB 方式）。

- IA（Increment address After each access），每取一个操作数后基址寄存器递增
- IB（Increment address Before each access），每取一个操作数前基址寄存器递增
- DB（Decrement address Before each access），每取一个操作数前基址寄存器递减
- DA（Decrement address After each access），，每取一个操作数后基址寄存器递减

类似地，STM（Store Multiple registers）指令也使用多寄存器寻址方式。其作用和 LDM 相反，是将一组寄存器中的数值保存到连续的存储器区域。其典型汇编语法为“**STM {addr_mode} <Rn>{!}, <registers>**”。

例如，如下格式指令可以完成四个寄存器数据的载入。其中 R1 载入数据的存储器地址是[R0]，即 R0 中的数值为操作数的地址；而 R2 载入的数据在存储器中的地址是[R0+4]；R3 载入的数据在存储器中的地址是[R0+8]；R4 载入的数据在存储器中的地址是[R0+12]。

- LDMIA R0!, {R1, R2, R3, R4}

该指令的后缀 IA 表示在每次执行完加载/存储操作后，R0 按字长度增加，对于 32 位的 ARM 处理器，每次地址的增加（或减少）的单位都是 4 个字节单位。因此，该指令可将连续存储单元的 32 比特数值传送到 R1~R4。在使用多寄存器寻址指令时，寄存器集合的顺序如果由小到大的顺序排列，可以使用“-”连接，否则用“,”分隔书写。下面的指令完成和上面一条指令相同的功能。

- LDMIA R0!, {R1-R4}

多寄存器寻址是 ARM 处理器较有特色的功能。需要注意的是，在很多 ARM 的资料中，也把上述多寄存器寻址方式称作块拷贝寻址。这种称呼更加突出多个寄存器数据批量复制的特点。以下为另外四个示例。

- STMIA R0!, {R1-R7} ;R1~R7 的数保存到 R0 指向的地址，每取一个数后 R0 递增 4
- STMIB R0!, {R1-R7} ;R1~R7 的数保存到 R0 指向的地址，每取一个数前 R0 递增 4
- STMDA R0!, {R1-R7} ;R1~R7 的数保存到 R0 指向的地址，每取一个数后 R0 递减 4
- STMDB R0!, {R1-R7} ;R1~R7 的数保存到 R0 指向的地址，每取一个数前 R0 递减 4

### 6.3.9 堆栈寻址

如果把前述多寄存器寻址方式中 LDM 或 STM 指令中的基址寄存器更换为堆栈指针寄存器 SP，并添加可选项{!}（意为每次存/取操作数就更新一下 SP），则寻址操作数为堆栈中存放的数值，寻址方式变成堆栈寻址。如果是 LDM，则每次取操作数自动 POP 堆栈中的数据到指定的寄存器中；反之，STM 指令则每次将寄存器中的数自动 PUSH 到堆栈中。

如第 5 章所述，Cortex-M 系列处理器支持满递减类型的堆栈，可以使用 LDMFD 指令从堆栈装载数据，使用 STMFD 指令来保存数据到堆栈。

LDMFD 指令与前述 LDMIA 指令格式相同，区别在于 LDMFD 指令中基址寄存器为 SP；LDMEA 指令与前述 LDMDB 指令格式相同，区别在于 LDMFD 指令中基址寄存器为 SP。

STMFD 指令则与前述 STMDB 指令格式相同，区别在于 STMFD 指令中基址寄存器为 SP。STMEA 指令则与前述 STMIA 指令格式相同，区别在于 STMEA 指令中基址寄存器为 SP。以下为 LDMFD 和 STMFD 指令使用示例。

- STMFD SP!, {R1-R7} ;将 R1~R7 寄存器中的数压入堆栈
- LDMFD SP!, {R1-R7} ;将堆栈中的数取出存入 R1~R7 寄存器

### 6.3.10 PC 相对寻址

PC 相对（PC-relative）寻址是一种特殊的基址变址寻址，常简称为相对寻址。PC 相对寻址以程序计数器（PC）寄存器的当前值作为基地址，指令中的地址标号作为偏移量，将两者相加获得操作数的地址。用一个相对 PC 的偏移量来指示存储器地址时，该偏移量可以由汇编语句的标号（Label）指定，也可以是按照文本池（Literal pool）方式定义的数据块相对

当前代码的位置。

下面先来看下使用汇编语句标号作为地址偏移量的的示例。如下例中“BL”指令为跳转指令，其功能是跳转到“MY_SUB”标号所对应的语句。此时，基址寄存器是 PC（取 PC+4 作为基地址），偏移量是“MY_SUB”标号语句相对于当前指令的偏移量（汇编语言编译器依据代码将汇编语句的标号转换为具体数值）。

```
    BL    MY_SUB   ;相对寻址，跳转到 MY_SUB 处执行
    ;……;其他指令
MY_SUB
    ;……;其他指令
```

ADR 指令也是一条用到了 PC 相对寻址的指令，ADR 允许用 12 比特表示偏移量，故寻址范围是 PC 前后的 4095。如下为一个使用示例，如果标号（Label）为“start”的指令相对于当前 PC 为#0x1C，那么 ADR 生成的地址是 PC+0x1C。

```
start MOV R0, #10
    ;……;其他指令
    ADR    R2，start ;将标号（Label）为“start”语句的地址送入 R2
```

如果在程序的代码段开辟一块区域存放数据，这块区域被称作文本池。可以通过 LDR 指令从这个特定的区域把数据传送到目标寄存器。此时 LDR 指令就需要使用 PC 相对寻址方式。例如，以下代码可以将相对当前代码位置后 12 字节位置的数据（32 比特）传送到 R0 寄存器。

```
LDR R0, [PC, #0xC]
```

## 6.4 Cortex-M3/M4 指令集

ARM Cortex-M3 和 Cortex-M4 处理器的指令可以按照功能分为不同的类别。常见的指令类别包括：处理器内数据传送、存储器访问、算术运算、逻辑运算、移位和循环移位运算、数据格式转换、位域处理指令、分支跳转、除法指令、乘并累加类指令、存储器屏障指令、异常相关指令、休眠模式相关指令等。另外，Cortex-M4 处理器支持增强 DSP 指令，如 SIMD 运算和打包指令、快速乘法、额外的乘并累加类指令、饱和运算指令、浮点运算指令等。

Cortex-M3/M4 处理器支持的指令繁多，但使用 C 语言编写程序时，可由 C 编译器可以生成高效的代码，一般情况下不需要了解全部指令的细节。鉴于此，本节并不详细介绍每一条指令，而侧重说明不同类别指令功能上的差异。要了解每条指令的使用细节，可以参考 Cortex-M3/Cortex-M4 设备用户指南和 ARM®v7-M 架构参考手册。

### 6.4.1 处理器内数据传送指令

在微处理器内，由于计算的需要，经常要在微处理器不同电路单元之间来回传送数据。例如，数据从一个寄存器送到另外一个寄存器，通用寄存器和特殊功能寄存器之间传送数据，将立即数送到寄存器。

表 6.10 中列出了 Cortex-M3/M4 共同支持的此类指令。表中 MOVS 指令和 MOV 指令类似（助记符“MOV”后携带后缀“S”，表示该指令执行会更新 APSR 中的标志位）。对于将一个八位立即数送到通用目标寄存器组中的一个寄存器来说，若目标操作数为低寄存器

（R0~R7），16 位 Thumb 指令即可实现。若要将立即数送到高位寄存器，需要使用 32 位的 MOV/MOVS 指令。具有浮点单元的 Cortex-M4 处理器，还需要在处理器寄存器与浮点单元电路的寄存器之间进行数据传送。故提供了一些额外的指令支持。

表 6.10 Cortex-M3/M4 处理器内数据传送指令

| 指令 | 目标寄存器 | 源 | 操作 |
|---|---|---|---|
| MOV | R1 | R0 | 从 R0 复制数据至 R1 |
| MOVS | R1 | R0 | 从 R0 复制数据至 R1，并更新 APSR 标志位 |
| MRS | R1 | PRIMASK | 将数据从特殊寄存器 RPIMASK 复制至 R1 |
| MSR | CONTROL | R1 | 将数据从 R1 复制到特殊寄存器 CONTROL |
| MOV | R1 | #0x34 | 设置 R1 为 0x34 |
| MOVW | R1 | #0x1234 | 设置 R1 为 16 位常量 0x1234 |
| MOVT | R1 | #0x1234 | 设置 R1 的高 16 位为 0x1234 |
| MVN | R1 | R0 | 将 R0 中数据取反后送至 R1 |

### 6.4.2 存储器访问指令

Cortex-M3/M4 处理器的存储器支持 32 位、16 位或 8 位宽度的数据访问，而寄存器是 32 位的。为了适应不同位宽的存储器操作数访问，设计了多条存储器访问指令指令。不同的存储器访问指令的寻址模式、操作数类型和数据传输方向各异。在 6.3 小节，已介绍过基本的 LDR 和 STR 指令。更多用于数据传输的指令如表 6.11 所示。若 Cortex-M4 处理器内部有浮点单元，还支持一些用于从存储器装载浮点数或保存浮点数到存储器的指令。

表 6.11 Cortex-M3/M4 不同操作数类型的存储器访问指令

| 数据类型 | 加载（读存储器） | 存储（写存储器） |
|---|---|---|
| 8 位 | - | STRB |
| 8 位无符号 | LDRB | - |
| 8 位有符号 | LDRSB | - |
| 16 位 | - | STRH |
| 16 位无符号 | LDRH | - |
| 16 位有符号 | LDRSH | - |
| 32 位 | LDR | STR |
| 多个 32 位 | LDM | STM |
| 双字（64 位） | LDRD | STRD |
| 栈操作（32 位） | POP | PUSH |

表 6.11 中 LDRSB 和 LDRSH 指令会对被加载数据自动执行符号位扩展[9]，将其转换为有符号的 32 位数据。例如，若 LDRSB 指令读取的是 0x88（其符号位为“1”，负数），则数据在被放到目标寄存器前会被转换为 0xFFFF FF88。

[9] **符号位扩展**：低位宽的数存到到更高位宽的存储单元时，需要考虑多出来的位填充什么信息。例如，8 位的有符号数 1111 0000B 存储到 16 位的寄存器中，寄存器的低 8 位存放 1111 0000，高 8 位填充 1111 1111，这种操作称作符号位扩展，常简称符号扩展。有符号数为负数时高 8 位填充的都是“1”，有符号数为正数时，则符号位扩展会将高 8 位均填为“0”。与之对应还有一个术语，**零扩展**。一个 8 位的数 b 零扩展至 16 位就是将高 8 位全部填充“0”。习惯上把符号位扩展和零扩展统称为数据扩展。

在前述章节分析 T32 指令集寻址方式（参见 6.3 小节）的时候，我们提到 ARM 处理器支持多种寻址方式。包括立即数寻址、寄存器寻址、寄存器移位寻址、寄存器间接寻址、基址变址寻址、多寄存器寻址（块拷贝寻址）、堆栈寻址、相对寻址等。接下来，我们结合表 6.11 所示各存储器访问指令进一步分析不同指令的功能差异，以及它们在寻址方式方面的差异。

### 1. 偏移寻址模式

在偏移寻址模式下，操作数的存储器地址为基址寄存器中的数值和立即数（偏移）的加和。偏移值可以为正数或负数，表 6.12 列出了一些常用的加载和存储指令在偏移寻址模式下的语法和功能。

表 6.12 偏移量为立即数的基址变址寻址指令语法

| 指令语法 | 描述 |
|---|---|
| LDRB Rd, [Rn, #offset] | 从存储器位置 Rn+offset 读取字节 |
| LDRSB Rd, [Rn, #offset] | 从存储器位置 Rn+offset 读取字节并做符号扩展 |
| LDRH Rd, [Rn, #offset] | 从存储器位置 Rn+offset 读取半字 |
| LDRSH Rd, [Rn, #offset] | 从存储器位置 Rn+offset 读取半字并做符号扩展 |
| LDR Rd, [Rn, #offset] | 从存储器位置 Rn+offset 读取字 |
| LDRD Rd1, Rd2, [Rn, #offset] | 从存储器位置 Rn+offset 读取双字 |
| STRB Rd, [Rn, #offset] | 往存储器位置 Rn+offset 存储字节 |
| STRH Rd, [Rn, #offset] | 往存储器位置 Rn+offset 存储半字 |
| STR Rd, [Rn, #offset] | 往存储器位置 Rn+offset 存储字 |
| STRD Rd1, Rd2, [Rn, #offset] | 往存储器位置 Rn+offset 存储双字 |

如下为一个偏移量为立即数的基址变址寻址示例。该语句从基址寄存器 R1 中取出数值，加上立即数#0x3 后得到操作数的地址，再从该地址取出操作数传送至 R0。

- LDRB R0, [R1, #0x8] ;从存储器地址 R1+0x8 中读取一个字节并存入 R0

### 2. 前变址寻址模式

前变址寻址模式中操作数地址的计算与偏移寻址模式相同，即操作数的存储器地址为基址寄存器中的数值和立即数（偏移）的加和。区别在于，前变址寻址会将计算得到的操作数地址写回到基址寄存器。表 6.13 所示为前变址寻址模式下的指令语法。

表 6.13 寻址前变址模式的存储器访问指令语法

| 指令语法 | 描述 |
|---|---|
| LDRB Rd, [Rn, #offset]! | 从存储器位置 Rn+offset 读取字节至 Rd 并更新 Rn 为 Rn+offset |
| LDRSB Rd, [Rn, #offset]! | 从存储器位置 Rn+offset 读取字节后进行符号扩展至 Rd 并更新 Rn 为 Rn+offset |
| LDRH Rd, [Rn, #offset]! | 从存储器位置 Rn+offset 读取半字至 Rd 并更新 Rn 为 Rn+offset |
| LDRSH Rd, [Rn, #offset]! | 从存储器位置 Rn+offset 读取半字后进行符号扩展至 Rd 并更新 Rn 为 Rn+offset |
| LDR Rd, [Rn, #offset]! | 从存储器位置 Rn+offset 读取字至 Rd 并更新 Rn 为 Rn+offset |

| | |
|---|---|
| LDRD Rd1, Rd2, [Rn, #offset]! | 从存储器位置 Rn+offset 读取双字并更新 Rn 为 Rn+offset |
| STRB Rd, [Rn, #offset]! | 往存储器位置 Rn+offset 存储字节并更新 Rn 为 Rn+offset |
| STRH Rd, [Rn, #offset]! | 往存储器位置 Rn+offset 存储半字并更新 Rn 为 Rn+offset |
| STR Rd, [Rn, #offset]! | 往存储器存储字并更新 Rn 为 Rn+offset |
| STRD Rd1, Rd2, [Rn, #offset]! | 往存储器存储双字并更新 Rn 为 Rn+offset |

如下为一个前变址寻址模式的存储器访问指令示例。该指令把基址寄存器 R1 中取出的数值和立即数 0x8 相加得到操作数的存储器地址，将该地址的数值传送到寄存器 R0，同时将该地址写回到基址寄存器 R1。指令中的“!”后缀表示指令完成时需要更新存放基地址的基址寄存器（写回）。

- LDR R0, [R1, #0x8]! ;读取存储器地址 R1+0x8 数值至 R0，随后 R1 被更新为 R1+0x8

### 3. 后变址寻址模式

后变址寻址模式下，操作数的地址从基址寄存器获取，指令执行后再将基址寄存器数值加上偏移量生成一个新的地址，并将该地址写入基址寄存器。表 6.14 所示为后变址寻址模式下的指令语法。

表 6.14 寻址后变址模式的存储器访问指令语法

| 指令语法 | 描述 |
|---|---|
| LDRB Rd, [Rn], #offset | 读取存储器[Rn]处的字节到 Rd，然后更新 Rn 为 Rn+offset |
| LDRSB Rd, [Rn], #offset | 读取存储器[Rn]处的字节到 Rd 并进行符号扩展，然后更新 Rn 为 Rn+offset |
| LDRH Rd, [Rn], #offset | 读取存储器[Rn]处的半字到 Rd，然后更新 Rn 为 Rn+ offset |
| LDRSH Rd, [Rn], #offset | 读取存储器[Rn]处的半字到 Rd 并进行符号位扩展，然后更新 Rn 为 Rn+offset |
| LDR Rd, [Rn], #offset | 读取存储器[Rn]处的字到 Rd，然后更新 Rn 为 Rn+offset |
| LDRD Rd1, Rd2, [Rn], #offset | 读取存储器[Rn]处的双字到 Rd1 和 Rd2，然后更新 Rn 为 Rn+offset |
| STRB Rd, [Rn], #offset | 存储字节到存储器[Rn]，然后更新 Rn 为 Rn+offset |
| STRH Rd, [Rn], #offset | 存储半字到存储器[Rn]，然后更新 Rn 为 Rn+offset |
| STR Rd, [Rn], #offset | 存储字到存储器[Rn]，然后更新 Rn 为 Rn+offset |
| STRD Rd1, Rd2, [Rn], #offset | 存储双字到存储器[Rn]，然后更新 Rn 为 Rn+offset |

如下为一个后变址寻址模式的存储器访问指令示例。该指令把基址寄存器 R1 中取出的数值作为操作数的存储器地址，取出操作数传送到 R0，随后更新 R1 的数值为 R1 原有数值加 0x8。

- LDR R0, [R1], #0x8 ;读取存储器位置 R1 数值至 R0，然后 R1 被更新为 R1+0x8

后变址的寻址模式在处理数组时非常有用，在访问数组中的元素时，基址寄存器的自动调整，有助于精简代码并加快执行。

### 4. 寄存器移位寻址模式

存储器访问指令还可以使用寄存器移位寻址模式。待访问操作数的地址由一个基址寄存器和一个偏移量相加得到，该偏移量是另一个寄存器中数值经过移位（允许 0~3 位的移位）运算得到的。与立即数偏移类似，不同的操作数类型对应多种指令形式，如表 6.15 所示。

表 6.15 寄存器偏移的存储器访问指令语法

| 指令语法 | 描述 |
|---|---|
| LDRB Rd, [Rn, Rm, LSL #n] | 从存储器位置 Rn+(Rm<<n)处读取字节存入 Rd |
| LDRSB Rd, [Rn, Rm, LSL #n] | 从存储器位置 Rn+(Rm<<n)处读取字节并进行符号位扩展后存入 Rd |
| LDRH Rd, [Rn, Rm, LSL #n] | 从存储器位置 Rn+(Rm<<n)处读取半字存入 Rd |
| LDRSH Rd,[Rn,Rm,LSL#n] | 从存储器位置 Rn+(Rm<<n)处读取半字并进行符号位扩展后存入 Rd |
| LDR Rd, [Rn, Rmf, LSL #n] | 从存储器位置 Rn+(Rm<<n)处读取字存入 Rd |
| STRB Rd, [Rn, Rm, LSL #n] | 往存储器位置 Rn+(Rm<<n)存储字节存入 Rd |
| STRH Rd, [Rn, Rm, LSL #n] | 往存储器位置 Rn+(Rm<<n)存储半字存入 Rd |
| STR Rd, [ Rn, Rm, LSL #n] | 往存储器位置 Rn+(Rm<<n)存储字存入 Rd |

如下为寄存器移位寻址的存储器访问指令示例。第一条指令中，基址寄存器为 R1，偏移量由 R2 数值左移两位得到，故操作数的存储器地址为“R1+(R2<<2)”，该操作数会被传送到 R3。第二条指令则将 R5 的数值保存到 R1+R2 对应的地址去。

- LDR R3, [ R1, R2, LSL #2] ;将位于 R1+(R2<<2)的存储器内容加载至 R3
- STR R3, [R1, R2] ;将 R3 写入存储器位置 R1+R2，移位运算是可选的

### 5. 多加载和多存储模式

T32 指令集中，有些指令可以从一块连续的存储器区域装载多个数据到多个寄存器中。此时会采用一种特殊的寻址方式，多寄存器寻址。如 6.3.8 所述，Cortex-M3/M4 中，多寄存器寻址可以选择两种不同的基址寄存器的改变方式：IA（Increment address After each access）方式，每取一个操作数后基址寄存器递增；DB（Decrement address Before each access）方式，每取一个操作数前基址寄存器递减。

相应的指令包括 LDM 和 STM 指令。LDM 和 STM 指令在使用时可以不进行基地址写回，指令语法如表 6.16 所示。

表 6.16 多加载/存储指令语法

| 指令语法 | 描述 |
|---|---|
| LDMIA Rn,<reg list> | 从 Rn 指定的存储器位置读取多个字，地址在每次读取后增加 |
| LDMDB Rn, <reg list> | 从 Rn 指定的存储器位置读取多个字，地址在每次读取前减小 |
| STMIA Rn, <reg list> | 往 Rn 指定的存储器位置写入多个字，地址在每次写入后增加 |
| STMDB Rn, <reg list> | 往 Rn 指定的存储器位置写入多个字，地址在每次写入前减小 |

表 6.16 中，< reg list>为寄存器列表。至少包括一个寄存器，如果是多个寄存器，寄存器列表开始为“{”，结束为“}”；寄存器列表可以使用“-”（连字符）表示范围，如，R0-R4

表示 R0、R1、R2、R3 以及 R4。

下面的指令功能为，读取地址 0x2000000~2000000F（四个字）的内容到 R0、R2、R3 和 R4。由于寄存器不连续，所示寄存器列表表示为“{R0, R2-R4}”。

- MOV R1, 0x20000000 ;将 R1 设置为 0x20000000
- LDMIA R1, {R0, R2-R4} ;从[R1]存储器位置读取 4 个字并将其存入 R0、R2、R3、R4

与其他的加载/存储指令类似，可以在 STM 和 LDM 中使用写回。使用{!}表示需要将修改后的地址写入基址寄存器<Rn>。如表 6.17 所示。

表 6.17 带写回的多加载/存储指令语法

| 指令语法 | 描述 |
|---|---|
| LDMIA Rn!, <reg list> | 从 Rn 指定的存储器位置读取多个字，地址在每次读取后增加(IA)，Rn 在传输完成后写回 |
| LDMDB Rn!, <reg list> | 从 Rn 指定的存储器位置读取多个字，地址在每次读取前减小(DB)，Rn 在传输完成后写回 |
| STMIA Rn!, <reg list> | 往 Rn 指定的存储器位置写入多个字，地址在每次写入后增加，Rn 在传输完后写回 |
| STMDB Rn!, < reg list> | 往 Rn 指定的存储器位置写入多个字，地址在每次写入前减小，Rn 在传输完成后写回 |

## 6. 压栈和出栈

栈的压栈（PUSH）和出栈（POP）可视为另一种形式的多存储和多加载，此时利用当前选定的栈指针来生成地址。当前栈指针可以是主栈指针 MSP，也可以是进程栈指针 PSP（由处理器模式和 CONTROL 寄存器值决定）。若浮点单元存在，除了 PUSH 和 POP，还提供了 VPUSH 和 VPOP 指令，压栈和出栈的指令如表 6.18 所示。

表 6.18 压栈和出栈指令语法

| 栈操作示例 | 描述 | 备注 |
|---|---|---|
| PUSH <reg list> | 将寄存器的值存入栈中 | |
| POP <reg list> | 从栈中恢复数值到寄存器 | |
| VPUSH.32 <s reg list> | 将 32 位单精度寄存器存入栈中 | 仅用于浮点单元 |
| VPUSH 64 <d reg list> | 将 64 位双精度寄存器存入栈中 | 仅用于浮点单元 |
| VPOP32 <s reg list> | 从栈中恢复单精度寄存器 | 仅用于浮点单元 |
| VPOP64 <d reg list> | 从栈中恢复双精度寄存器 | 仅用于浮点单元 |

寄存器列表的语法和 LDM 与 STM 相同。< reg list>为寄存器列表，至少包括一个寄存器，如果是多个寄存器，寄存器列表开始为“{”，结束为“}”；寄存器列表可以使用“-”（连字符）表示范围。如下为使用示例。

- PUSH {R0, R4-R6, R8} ;将 R0、R4、R5、R6 和 R8 压入栈中
- POP {R1, R2} ;将栈中内容存入 R1 和 R2

## 7. PC 相对寻址

存储器访问还可以使用 PC 寄存器作为基地址，再加上偏移量形成待访问操作数地址，如表 6.19 所示。常用于将立即数加载到寄存器中，即 6.3.10 小节所述文本池访问。

表 6.19 PC 相对寻址的存储器访问指令

| 指令语法 | 描述 |
|---|---|
| LDRB Rt, [PC, #offset] | 利用 PC 偏移加载无符号字节到 Rt |
| LDRSB Rt, [PC, #offset] | 对字节数据进行符号位扩展并利用 PC 偏移加载到 Rt |
| LDRH Rt, [PC, #offset] | 利用 PC 偏移加载无符号半字到 Rt |
| LDRSH Rt, [PC, #offset] | 对半字数据进行符号位扩展并利用 PC 偏移加载到 Rt |
| LDR Rt, [PC, #offset] | 利用 PC 偏移加载字数据到 Rt |
| LDRD Rt1, Rt2, [PC, #offset] | 利用 PC 偏移加载双字数据到 Rt1 和 Rt2 |

在 6.3.10 小节已经提到文本池，文本池的本质就是 ARM 汇编语言代码段（section）中的一块用来存放常量数据而非可执行代码的内存块。使用文字池是有原因的，例如，需要在一条指令中使用一个四字节长度的常量数据（这个数据可以是内存地址，也可以是数字常量）时，由于 ARM 指令是定长的（T32 指令集最多支持 32 比特指令长度），因而无法把这个四字节的常量数据编码在一条指令中。此时，ARM 编译器（编译 C 源程序）/汇编器（编译汇编程序）就会在代码节中分配一块存储区域，并把这个四字节的数据常量保存于此。随后，再使用一条指令把这个四字节的常量加载到寄存器中参与运算。

以下代码可以将相对当前代码位置后四字节位置的数据传送到 Rt。

- LDR Rt, [PC, #0x4]

### 8. 非特权等级加载和存储

ARM 处理器区分特权和非特权访问等级。运行于非特权访问等级的程序无法访问特权访问等级程序中的数据。例如在有操作系统的情形下，操作系统的代码运行在特权访问等级，而普通用户的代码则运行在非特权访问等级。用户程序通过操作系统提供的 API 函数访问存储器的时候，可能会因目标存储位置的访问等级设置而无法正常访问。故 ARM 中提供了一组特殊的加载和存储指令，在特权访问等级程序中使用这些指令加载或保存的数据，其权限等同于非特权访问等级的程序。例如，工作于特权等级的操作系统代码在 API 中向非特权等级的普通用户程序传递数据时，可使用这些特殊指令存取目标存储器位置，如果存取过程发生异常，则操作系统可知用户程序是无法访问这些存储器位置的，不能把这些存储器位置上存储的数据传递给普通的用户程序，需要调整。相关指令语法如表 6.20 所示。

表 6.20 非特权访问等级的存储器访问指令

| 指令语法 | 描述 |
|---|---|
| LDRBT Rd, [Rn, #offset] | 从存储器位置 Rn+ offset 读取字节存入 Rd |
| LDRSBT Rd, [Rn, #offset] | 从存储器位置 Rn+ offset 读取字节并做符号位扩展后存入 Rd |
| LDRHT Rd, [Rn, #offset] | 从存储器位置 Rn+ offset 读取半字存入 Rd |
| LDRSHT Rd, [Rn, #offset] | 从存储器位置 Rn+offset 读取半字并做符号位扩展后存入 Rd |
| LDRT Rd, [Rn, #offset] | 从存储器位置 Rn+ offset 读取字存入 Rd |
| STRBT Rd, [Rn, #offset] | 往存储器位置 Rn+ offset 存储字节存入 Rd |
| STRHT Rd, [Rn, #offset] | 往存储器位置 Rn+ offset 存储半字存入 Rd |
| STRT Rd, [Rn, #offset] | 往存储器位置 Rn+ offset 存储字存入 Rd |

### 9. 排他式访问

排他式（Exclusive，或称为独占式）访问指令为一组特殊的存储器访问指令，用于实现信号量或互斥量操作。常用于某个资源被多个应用甚至多个处理器共享的情形。排他式数据加载和存储需要使用如表 6.21 所示的访问指令。

表 6.21 排他式访问指令

| 指令语法 | 描述 |
|---|---|
| LDREXB Rt, [Rn] | 从存储器位置 Rn 排他读取字节，存放至 Rt |
| LDREXH Rt, [Rn] | 从存储器位置 Rn 排他读取半字，存放至 Rt |
| LDREX Rt, [Rn, #offset] | 从存储器位置 Rn 排他读取字，存放至 Rt |
| STREXB Rd, Rt, [Rn] | 将 Rt 最低字节排他存储至存储器位置 Rn，返回状态存于 Rd（“0”表示操作成功） |
| STREXH Rd, Rt, [Rn] | 将 Rt 低半字排他存储至存储器位置 Rn，返回状态存于 Rd（“0”表示操作成功） |
| STREX Rd, Rt, [Rn, #offset] | 将 Rt 内容排他存储至存储器位置 Rn + offset，返回状态存于 Rd（“0”表示操作成功） |
| CLREX | 清空一个排他访问标识位 |

### 6.4.3 算术运算指令

Cortex-M3/M4 处理器提供多条用于算术运算的指令，表 6.22 列出了一部分常用运算指令。许多算数运算指令依据操作数的不同、对 APSR 影响不同，可以携带不同后缀，从而有多种形式。如，ADD 指令完成两个寄存器中数值的相加，或一个寄存器中数值和一个立即数的相加，实现不同功能的时候，同样是 ADD 指令会形成不同的机器码。

表 6.22 Cortex-M3/M4 算数运算指令（可选后缀未列出）

| 算术指令 | 指令功能 | 操作 |
|---|---|---|
| ADD Rd, Rn, Rm | Rd= Rn+ Rm | 加法运算 |
| ADD Rd, Rn, #immed | Rd= Rn+#immed | |
| ADC Rd, Rn, Rm | Rd=Rn+Rm+进位 | 带进位（APSR.C）的加法运算 |
| ADC Rd, #immed | Rd=Rd+#immed+进位 | |
| ADDW Rd, Rn, #immed | Rd= Rn +#immed | 寄存器和立即数相加 |
| SUB Rd, Rn, Rm | Rd= Rn-Rm | 减法 |
| SUB Rd, # immed | Rd= Rd-#immed | |
| SUB Rd, Rn,#immed | Rd= Rn-#immed | |
| SBC Rd,Rn,# immed | Rd=Rn-#immed-借位 | 带借位（APSR.C）的减法 |
| SBC Rd, Rn, Rm | Rd=Rn-Rm-借位 | |
| SUBW Rd, Rn, # immed | Rd= Rn-#immed | 寄存器和立即数相减 |
| RSB Rd, Rn, # immed | Rd=#immed-Rn | 减反转 |
| RSB Rd, Rn, Rm | Rd= Rm-Rn | |
| MUL Rd, Rn, Rm | Rd= Rn*Rm | 32 位乘法 |
| UDIV Rd, Rn, Rm | Rd= Rn/Rm | 无符号和有符号除法 |
| SDIV Rd, Rn, Rm | Rd= Rn/Rm | |

这里以 ADD 指令为例来说明不同后缀生成指令的差异。如下三行代码虽然都属于加法

运算，但是操作数、进位方式、对 APSR 标志位的更新都不一样，对应的二进制机器码也不同。按照传统的 Thumb 汇编语法，在使用 16 位 Thumb 代码时，ADD 指令会修改 APSR 中的标志。不过，32 位 Thumb-2 指令可以修改这些标志，也可以不修改。为了区分这两种操作，根据统一汇编语言语法，如需更新 APSR 标志位要使用 S 后缀。

- ADD R0, R0, R1 ;R0=R0+R1
- ADDS R0, R0, 0x1234 ;R0=R0+0x1234，更新 APSR 标志位
- ADC R0, R1, R2 ;R0=R1+R2+进位

Cortex-M3 和 Cortex-M4 处理器都支持 32 位或 64 位结果的 32 位乘法指令和乘并累加（MAC）指令，APSR 标志不受这些指令的影响。如表 6.23 所示，这些指令支持有符号和无符号数这两种不同数据类型。Cortex-M4 处理器还支持额外的 MAC 运算指令。

表 6.23 Cortex-M3/M4 乘法和 MAC 指令

| 指令 | 指令功能 | 操作 |
|---|---|---|
| MLA Rd, Rn, Rm, Ra | Rd= Ra+ Rn * Rm | 32 位 MAC 运算，32 位结果 |
| MLS Rd, Rn, Rm, Ra | Rd= Ra-Rn * Rm | 32 位乘减运算，32 位结果 |
| SMULL RdLo, RdHi, Rn, Rm | {RdHi, RdLo}=Rn*Rm | 有符号数据的 32 位乘及 MAC 运算，64 位结果 |
| SMLAL RdLo, RdHi, Rn, Rm | {RdHi, RdLo}+=Rn*Rm | |
| UMULL RdLo, RdHi, Rn, Rm | {RdHi, RdLo} =Rn*Rm | 无符号数据的 32 位乘及 MAC 运算，64 位结果 |
| UMLAL RdLo, RdHi, Rn, Rm | {RdHi, RdLo}+=Rn*Rm | |

### 6.4.4 逻辑运算指令

Cortex-M3/M4 处理器支持多种逻辑运算指令，如“与”操作、“或”操作以及“异或”操作等。如表 6.24 所示。

表 6.24 Cortex-M3/M4 逻辑运算指令(可选 S 后缀未列出)

| 指令 | 指令功能 | 操作 |
|---|---|---|
| AND Rd, Rn | Rd=Rd & Rn | 按位与 |
| AND Rd, Rn, # immed | Rd=Rn & #immed | |
| AND Rd, Rn, Rm | Rd=Rn & Rm | |
| ORR Rd, Rn | Rd=Rd \| Rn | 按位或 |
| ORR Rd, Rn, #immed | Rd= Rn \| #immed | |
| ORR Rd, Rn, Rm | Rd=Rn \| Rm | |
| BIC Rd, Rn | Rd=Rd & (~Rn) | 位清除 |
| BIC Rd, Rn,#immed | Rd=Rn & (~#immed) | |
| BIC Rd, Rn, Rm | Rd=Rn & (~Rm) | |
| ORN Rd, Rn, #immed | Rd=Rn \| (~#immed) | 按位或非 |
| ORN Rd, Rn, Rm | Rd=Rn \| (~Rm) | |
| EOR Rd, Rn | Rd=Rd ^ Rn | 按位异或 |
| EOR Rd, Rn, #immed | Rd=Rn \| #immed | |
| EOR Rd, Rn, Rm | Rd=Rn \| Rm | |

### 6.4.5 移位和循环移位指令

Cortex-M3/M4 处理器支持多种移位和循环移位指令，如表 6.25 所示。

表 6.25 Cortex-M3/M4 移位和循环移位运算指令(可选 S 后缀未列出)

| 指令 | 指令功能 | 操作 |
|---|---|---|
| ASR Rd, Rn, #immed | Rd=Rn>>immed | Rn 算术右移，结果存入 Rd |
| ASR Rd, Rn, Rm | Rd=Rn>>Rm | |
| LSL Rd, Rn, #immed | Rd=Rn<<#immed | Rn 逻辑左移，结果存入 Rd |
| LSL Rd, Rn, Rm | Rd=Rn<<Rm | |
| LSR Rd, Rn, #immed | Rd=Rn>>#immed | Rn 逻辑右移，结果存入 Rd |
| LSR Rd, Rn, R | Rd=Rn>>R | |
| ROR Rd, Rn, Rm | Rd=Rn>>Rm | Rn 循环右移，结果存入 Rd |
| RRX Rd, Rn | {C, Rd} = {Rn, C} | Rn 右移一位将 APSR.C 填充至最高位，所得结果存入 Rd |

几种不同移位方式的原理如图 6.9 所示。若使用了 S 后缀，这些循环和移位指令会更新 APSR 中的进位标志 C。要注意不同移位方式中 APSR.C 是否参与移位是有差异的

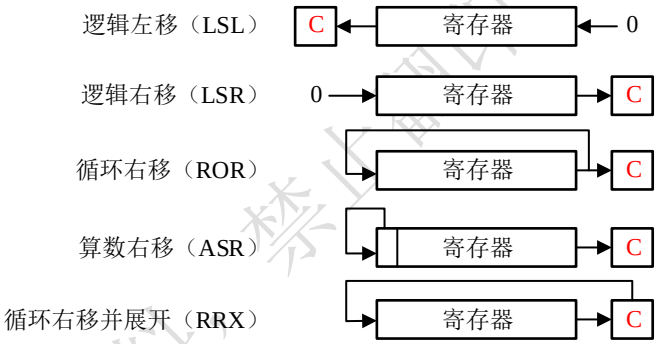


图 6.9 移位和循环移位运算

之所以 ARM 处理器能够支持图 6.9 所示所示的多种移位运算，是因为在处理器中内置了桶形移位器。前述寻址方式中寄存器移位寻址也是建立在桶形移位器电路基础之上的。例如，若第二个操作数为保存在寄存器，一些数据处理指令可以在进行数据处理前选择移位操作，如图 6.10 所示。图中 Rd 为目标寄存器，Rn 为第一个操作数，Rm 为第二个操作数。

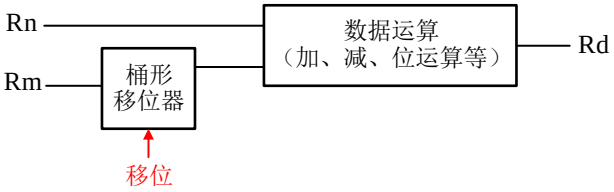


图 6.10 桶形移位器

表 6.26 所示数据处理指令均支持桶形移位器。除了表中列出的数据处理指令，桶形移位器还可用于存储器访问指令，如指令“LDR Rd, [Rn, Rm, LSL #n]”。

表 6.26 支持桶形移位器的数据处理指令

| 指令 | 功能 |
|---|---|
| MOV{S} Rd, Rm, <shift> | 传送 |
| MVN{S} Rd, Rm, <shift> | 传送取负 |
| ADD{S} Rd, Rm, Rn, <shift> | 加法 |
| ADC{S} Rd, Rm, Rn, <shift> | 带进位（APSR.C）的加法 |
| SUB{S} Rd, Rm, Rn, <shift> | 减法 |
| SBC{S} Rd, Rm, Rn, <shift> | 带借位（APSR.C）的减法 |
| RSB{S} Rd, Rm, Rn,<shift> | 反转减法 |
| AND{S} Rd, Rm, Rn, <shift> | 逻辑与 |
| ORR{S} Rd, Rm, Rn, <shift> | 逻辑或 |
| EOR{S} Rd, Rm, Rn, <shift> | 逻辑异或 |
| BIC{S} Rd, Rm, Rn, <shift> | 逻辑与非（位清除） |
| ORN{S} Rd, Rm, Rn, <shift> | 逻辑或非 |
| CMP Rn, Rm, <shift> | 比较 |
| CMN Rn, Rm, <shift> | 负比较 |
| TEQ Rn, Rm, <shift> | 测试相等（按位异或） |
| TST Rn, Rm, <shift> | 测试（按位与） |

### 6.4.6 数据格式转换

在 C 等高级语言中，不同类型的数可以做强制类型转换，例如 8 比特字符型转换为 32 比特有符号整数。同理，在微处理器中，针对不同类型数据格式转换也定义了专门指令。

在 Cortex-M3/M4 处理器中，用于处理数据的符号位扩展（Sign extend）和零扩展（Zero extend）的指令有多条指令，如将 8 位数转换为 32 位或将 16 位转换为 32 位。有符号和无符号指令都有 16 位和 32 位的形式，如表 6.27 所示。

表 6.27 有符号和无符号数的扩展

| 指令 | 指令功能 | 操作 |
|---|---|---|
| SXTB Rd, Rn | Rd=符号位扩展(Rn[7:0]) | 字节符号位扩展为字 |
| SXTH Rd, Rn | Rd=符号位扩展(Rn[15:0]) | 半字符号位扩展为字 |
| UXTB Rd, Rn | Rd=零扩展(Rn[7:0]) | 字节零扩展为字 |
| UXTH Rd, Rn | Rd=零扩展(Rn[15:0]) | 半字零扩展为字 |

表 6.27 中指令可以选择在进行符号位扩展运算前将输入数据循环右移，语法如表 6.28 所示。

表 6.28 具有可选循环移位的有符号和无符号展开

| 指令 | 操作 |
|---|---|
| SXTB Rd, Rn{, ROR #n } ;n=8/16/24 | 字节符号位扩展为字 |
| SXTH Rd, Rn{, ROR #n } ;n=8/16/24 | 半字符号位扩展为字 |
| UXTB Rd, Rn{, ROR #n } ;n=8/16/24 | 字节零扩展为字 |
| UXTH Rd, Rn{, ROR #n } ;n=8/16/24 | 半字零扩展为字 |

SXTB/SXTH 使用 Rn 的 bit7/bit15 进行符号位扩展，而 UXTB 和 UXTH 则将数据以零扩展的方式扩展为 32 位。例如，若 R0 中数值为 0x22AA8765，如下指令完成不同形式的数据扩展。

- SXTB Rl, R0 ;R1=0x00000065
- SXTH R1, R0 ;R1=0xFFFF8765
- UXTB Rl, R0 ;R1=0x00000065
- UXTH R1, R0 ;R1=0x00008765

另外，还有一组数据转换运算则用于反转寄存器中的字节，如表 6.29 和图 6.11 所示。这些指令通常用于大端和小端间的数据转换。

表 6.29 数据反转指令

| 指令 | 指令功能 | 操作 |
|---|---|---|
| REV Rd, Rn | Rd=rev(Rn) | 反转字中的字节 |
| REV16 Rd, Rn | Rd=rev16(Rn) | 反转每个半字中的字节 |
| REVSH Rd, Rn | Rd=revsh(Rn) | 反转低半字中的字节并将结果做符号位扩展 |

REV 反转字数据中字节顺序，而 REV16 则反转半字中的字节顺序。例如，若 R0 为 0x12345678，执行下面的指令后，R1 会变为 0x78563412，而 R2 则会变为 0x34127856。

- REV R1, R0
- REVH R2, R0

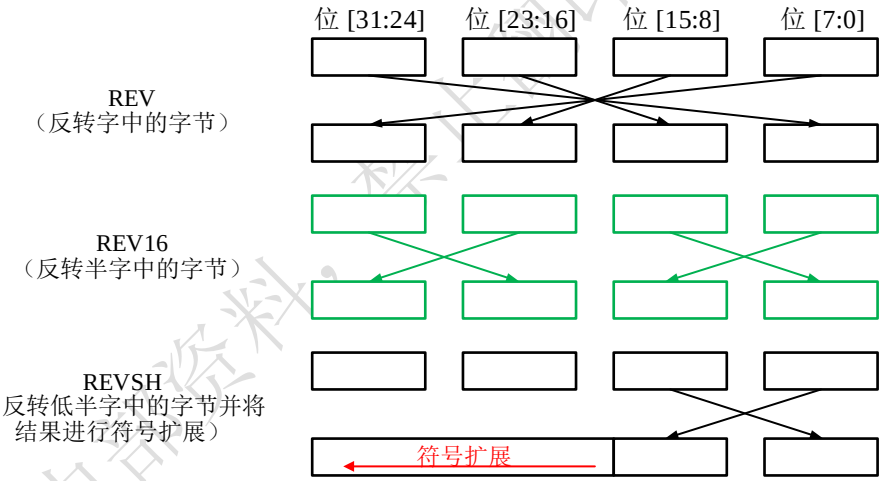


图 6.11 数据反转操作示意图

REVSH 和 REVH 类似，只是它只能在处理低半字后将结果符号位扩展。例如，若 R0 为 0x44558899，执行如下指令后，R1 中数值为 0xFFFF9988。

- REVSH Rl, R0

### 6.4.7 位域处理指令

Cortex-M3 和 Cortex-M4 处理器支持多种位域（Bitfield）处理运算，如表 6.30 所示。

表 6.30 位域处理指令

| 指令 | 操作 |
|---|---|
| BFC Rd, #<lsb>, #<width> | 清除寄存器中的位域 |
| BFI Rd, Rn, #<lsb>, #<width> | 将位域插入寄存器 |

| | |
|---|---|
| CLZ Rd, Rn | 前导零计数 |
| RBIT Rd, Rn | 反转寄存器中的位顺序 |
| SBFX Rd, Rn, #<lsb>, #<width> | 从源中复制位域并做符号位扩展 |
| UBFX Rd, Rn, #<lsb>, #<width> | 从源寄存器中复制位域 |

下面以具体的示例说明表 6.30 中指令的用法。例如，位域清除 BFC 指令可以清零寄存器任意相邻的 1~31 位。该指令的语法为：BFC <Rd>, <#lsb>, <#width>。如下指令将得到结果 R0=0x1234F00F。

- LDR R0, =0x12345678 ;伪指令，把 0x12345678 装载到 R0（不是从地址 0x12345678 装载）
- BFC R0, #4, #8 ;R0=0x12345008

再如，位域插入 BFI 指令将一个寄存器 1~31 位中某些位（#width）复制到另外一个寄存器的指定位置（#lsb+#width 与#lsb 之间的位）上，语法为：BFI <Rd>, <Rn>, <#lsb>, <#width>。如下指令得到的结果为 R1=0xAA5678DD。

- LDR R0, =0x12345678 ;把 0x12345678 装载到 R0
- LDR R1, =0xAABBCCDD ;把 0xAABBCCDD 装载到 R1
- BFI R1, R0, #8, #16 ;将 R0[15:0]插入 R1[23:8]，将得到 R1=0xAA5678DD

### 6.4.8 比较和测试指令

比较和测试指令用于更新 APSR 中的标志位，这些标志位可以被条件跳转指令使用。表 6.31 列出了这些指令。这些指令执行总是会更新 APSR，因此这些指令中不存在 S 后缀。

表 6.31 比较测试指令

| 指令 | 操作 |
|---|---|
| CMP <Rn>, <Rm> | 比较：计算 Rn-Rm，APSR 更新，计算结果不保存 |
| CMP <Rn>, #<immed> | 比较：计算 Rn-立即数#immed，APSR 更新，计算结果不保存 |
| CMN <Rn>, <Rm> | 负比较：计算 Rn+Rm，APSR 更新，计算结果不保存 |
| CMN <Rn>, #<immed> | 负比较：计算 Rn+立即数#immed，APSR 更新，计算结果不保存 |
| TST <Rn>, <Rm> | 测试（按位与）：计算 Rn 和 Rm 相与后的结果，APSR 中的 N 位和 Z 位更新，但与运算的结果不保存，若使用了桶形移位则更新 C 位 |
| TST <Rn>, #<immed> | 测试（按位与）：计算 Rn 和立即数#immed 相与后的结果，APSR 中的 N 位和 Z 位更新，但与运算的结果不保存 |
| TEQ <Rn>, <Rm> | 测试（按位异或）：计算 Rn 和 Rm 异或后的结果，APSR 中的 N 位和 Z 位更新，但运算的结果不保存，若使用了桶形移位则更新 C 位 |
| TEQ <Rn>, #<immed> | 测试（按位异或）：计算 Rn 和立即数#immed 异或后的结果，APSR 中的 N 位和 Z 位更新，但运算的结果不保存 |

### 6.4.9 程序流控制指令

用于程序流控制的指令（也称为分支控制指令）有多种，如跳转、函数调用、条件跳转、比较和条件跳转的组合、条件执行（IF-THEN 指令）、表格跳转等。在 ARM 处理器中，更新 R15（PC）的数据处理指令（如 MOV、ADD），或写入 PC 的读存储器指令（如 LDR、LDM、POP）也会引起跳转操作。

表 6.32 Cortex-M3 的程序流控制指令

| 助记符 | 英文名称 | 中文名称 |
|---|---|---|
| B | Branch | 无条件跳转 |
| BL | Branch with Link | 无条件跳转，并保存返回地址，用于函数调用 |
| BLX | Branch and Link with eXchange | 寄存器间接寻址跳转，并保存返回地址，用于函数调用 |
| BX | Branch and eXchange | 寄存器间接寻址跳转 |
| CBNZ | Compare and Branch if Non Zero | 比较不为零则跳转 |
| CBZ | Compare and Branch if Zero | 比较为零则跳转 |
| IT | If-Then | 条件执行指令 |
| TBB | Table Branch Byte | 按照跳转表跳转（字节） |
| TBH | Table Branch Halfword | 按照跳转表跳转（半字） |

### 1. 无条件跳转

使用表 6.33 所示的无条件跳转指令可以跳转到标号（<label>）所在的语句，或跳转到<Rm>中的存放地址位置。不同跳转指令所支持的最大跳转范围是不一样的，表 6.33 所示。

表 6.33 Cortex-M3/M4 无条件跳转指令

| 跳转指令 | 跳转的范围 | 功能 |
|---|---|---|
| B <label> | −1MB to +1MB | 跳转到标号地址 |
| BL <label> | −16MB to +16MB | 跳转到标号地址并将返回地址保存在 LR（R14）中 |
| BX <Rm> | 寄存器中的任何值 | 跳转到 Rm 指定的地址 |
| BLX <Rm> | 寄存器中的任何值 | 跳转到 Rm 指定的地址并将返回地址保存在 LR（R14)中 |

### 2. 条件跳转

条件跳转指的是，根据 APSR 寄存器中的标志位（N、Z、C 和 V 标志位）决定是否跳转（APSR 寄存器的标志位定义如图 6.4 所示）。无条件跳转指令添加条件码后缀即条件跳转指令，其跳转范围如表 6.34 所示。注意，由于 B 指令的二进制编码格式中预留了四比特的条件码，故与一般的条件执行指令（如 MOVEQ）不同，不需要与 IT 指令配合，条件跳转指令即可执行。

表 6.34 Cortex-M3/M4 条件跳转指令

| 跳转指令 | 跳转的范围 | 功能 |
|---|---|---|
| B[cond] <label> (在 IT block 外) | −1 MB to +1 MB | 跳转到标号地址 |
| B[cond] <label> (在 IT block 内) | −16 MB to +16 MB | 跳转到标号地址 |
| BL[cond] <label> | −16 MB to +16 MB | 跳转到标号地址并将返回地址保存在 LR（R14）中 |
| BX[cond] <Rm> | 寄存器中的任何值 | 跳转到 Rm 指定的地址 |
| BLX[cond] <Rm> | 寄存器中的任何值 | 跳转到 Rm 指定的地址，并将返回地址保存在 LR（R14）中 |

表 6.34 中所示可选的条件码后缀“[cond]”的取值及含义可参考表 6.6。例如，BEQ 对应功能为 APSR 标志位 Z 为 1 时跳转，BNE 对应功能为 APSR 的标志位 Z 为 0 时跳转。如下为一个简单的使用示例。

```
CMP R0, #1          ;比较 R0 和 1，如果相等 APSR 的标志位 Z 会被置为 1
BEQ  R1             ;若 APSR 的 Z 标志位为 1 则跳转到 R1 中存放数值对应的地址
```

### 3. 比较和跳转

ARMv7-M 架构提供了两条新的跳转指令，它们合并了和零比较以及条件跳转操作。这两个指令为 CBZ（比较为零则跳转）和 CBNZ（比较非零则跳转）。

```
CBZ R0, label       ;比较 R0 和 0，如果相等则跳转到 label 对应的地址
CNBZ R5, label      ;比较 R5 和 0，如果不相等则跳转到 lable 对应的地址
```

CBZ 指令的作用相当于上一个示例中 CMP 指令和 BEQ 指令的功能组合，区别在于 APSR 的值不受 CBZ 和 CBNZ 指令的影响。另外，CBZ 和 CBNZ 指令只支持前向跳转，不支持向后跳转，跳转范围为当前指令后的 4~130 字节。且，只能使用 R0~R7。鉴于这样的特性，CBZ 和 CBNZ 指令适用于较小的循环体内的分支控制。

### 4. 条件执行指令

前述条件跳转指令根据 APSR 寄存器中的标志位来决定是否跳转。除此之外， Cortex-M3/M4 处理器还支持条件执行（If-Then）。对应的指令为 IT，“IT”意为“If Then”。在 6.2.5 小节介绍指令寻址格式的时候，我们曾以 MOV 指令为例分析过条件执行的基本过程。

IT 指令允许跟随其后的最多四条指令是条件执行的，跟随在 IT 指令后面的几条指令被称作一个 IT 块（IT block）。“IT”指令的汇编语法格式为：IT{x{y{z}}} cond。其中 cond 为 IT 块中第一条指令使用的条件码；x、y、z 分别指定 IT 块中第二条、第三条、第四条指令的是否执行的开关

如下两个例子中用到了条件码“EQ”或“NE”，指令执行的条件都和 APSR 寄存器中的 Z 标志位的取值有关系。

```
IT EQ               ;意为“If-Then”，下一条指令是条件执行的
ADDEQ R0, R1, R2    ;如果 APSR 的 Z 标志位是 1 则将 R1+R2 后结果传送至 R0
```

```
ITETT NE            ;意为“If-Then-Else-Then-Then”，随后四条指令是条件执行的
ADDNE R2, R0, R1    ;如果 APSR 的 Z 标志位是 0 则将 R0+R1 后结果传送至 R2
ADDEQ R3, R0, R1    ;如果 APSR 的 Z 标志位是 1 则将 R0+R1 后结果传送至 R3
ADDNE R1, R0, #1    ;如果 APSR 的 Z 标志位是 0 则将 R0+1 后结果传送至 R1
MOVNE R1, R0        ;如果 APSR 的 Z 标志位是 0 则将 R0 中的数传送至 R1
```

很多汇编工具软件会在带有条件码后缀的语句前自动插入所需的 IT 指令，故程序员无须自己在代码中使用 IT 指令。应注意，IT 指令块中的数据处理指令不应修改 APSR 的数值。

一般来说，使用 IT 指令可降低跳转指令的个数，从而提高程序代码的性能。例如，一个简短的 If-Then-Else 的程序流程通常需要一个条件跳转和无条件跳转，而用一个 IT 指令就可以代替了。但有时传统跳转方式可能会比 IT 指令更好，这是因为 IT 指令块中的条件失败也会执行一个周期，故发生条件失败时，使用条件跳转会比 IT 指令块更快。

### 5. 按跳转表跳转

汇编编程时直接实现类似 C 语言中 switch (case)这样的多分支跳转控制十分困难，这是因为需要跳转的目标地址与一个变量有关。前述过无条件跳转或条件跳转，跳转的目标地址是一个绝对物理地址或者一个 PC 相对地址，而 switch (case)中变量取不同值的时候需要跳转的偏移量是不同的。如果预先定义一个数组，每个数组元素是一个目标地址，那么通过改变数组下标，就可以通过数组元素得到不同的目标地址。跳转表就是一个这样的数组，跳转表的表项就是数组元素，通过跳转表首地址和表内的偏移可以实现类似数组不同下标的访问方式。

Cortex-M3/M4 处理器中，跳转表可以被组织成字节数组的形式，或者半字数组的形式。由于 T32 指令集含有 16 比特指令和 32 比特指令两种固定长度的，指令是按半字对齐的，表中的数值都是在左移一位后才作为前向跳转的偏移量的。故而跳转表被组织成字节数组时，相对于表首地址（基地址）的偏移小于 $2\times2^8$=512B；而跳转表被组织为半字数组时，相对于表首地址（基地址）的偏移小于 $2\times2^{16}$=128KB。相应的，Cortex-M3/M4 支持两条跳转表指令：TBB（按字节跳转）、TBH（按半字跳转）。

TBB 指令的语法为：“TBB [Rn, Rm]”。其中，Rn 中存放跳转表的基地址，不能是 SP；Rm 则为跳转表偏移（或称索引），不能是 SP 或 PC。TBB 指令执行的时候，跳转偏移量由 Rn 和 Rm 计算得到（[Rn+Rm]）。TBH 指令的情况非常类似，但是跳转表中的每个表项都是双字节大小，因此跳转表（数组）的索引不同，且偏移范围较大。为了表示跳转表（数组）索引的差异，TBH 的语法稍微不同：“TBH [Rn, Rm, LSL #1]”。其中“LSL #1”为寄存器移位寻址（参见 6.3.4 ），将 Rm 左移位。

TBB 和 TBH 提供了 PC 相对寻址的灵活方式。例如下面的代码，TBB 所定义的跳转表从标号为“BranchTable_Byte”位置开始，该位置利用 ADR 指令保存到 R0 中。每个表项占据一个字节，表项的数值则是与标号为“Case1”语句的偏移量。程序执行到 TBB 所在行的时候，根据 R1 取值的不同，会分别跳转到不同的分支，如 R1=1，则跳转到标号“Case2”所在的行执行。

```
ADR.W R0, BranchTable_Byte
TBB [R0, R1] ; R1 存放索引（偏移量），R0 存放跳转表的基地址
Case1
    ; 分支 1 的相关代码……
Case2
    ; 分支 2 的相关代码……
Case3
    ; 分支 3 的相关代码……
BranchTable_Byte
    DCB 0 ; 分支 1 相关代码的偏移量
    DCB ((Case2-Case1)/2) ; 分支 2 相关代码的偏移量
    DCB ((Case3-Case1)/2) ; 分支 3 相关代码的偏移量
```

上述示例代码中 DCB 是 ARM 汇编语言中的伪指令，用于分配一片连续的字节存储单元并用指定的数据初始化。ADR 指令可用于生成 PC 相对寻址后的地址。

### 6.4.10 饱和运算

饱和运算多用于数字信号处理类算法。如，在放大处理等操作后，信号的幅度可能会超出允许的输出范围，若只是简单地将数据的最高位去掉（这种操作称作溢出），最终得到的波形可能会出现严重的畸变，如图 6.12(a)所示。饱和运算把超出动态范围的数据强制置为最大允许值，从而减小畸变。畸变仍然存在，但相比于溢出操作，至少保留了信号波形的大致形状，如图 6.12(b)和(c)所示。

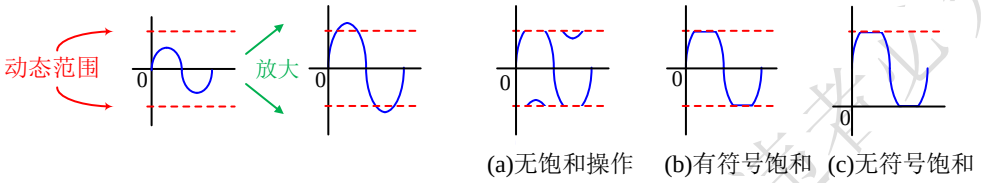


图 6.12 有符号饱和运算

Cortex-M3 处理器支持两条用于有符号和无符号数据饱和调整的指令：SSAT（用于有符号数据）和 USAT（用于无符号数据）。注意，Cortex-M4 处理器除了支持这两条饱和指令，而且还支持其他用于饱和算法的指令。SSAT 和 USAT 指令的语法如下。

- SSAT < Rd>, <#immed>, <Rn>, {,<shift>}
- USAT< Rd>, <#immed>, <Rn>, {,< shift>}

其中，Rn 为输入值；shift 为饱和前可选的移位操作，可以为#LSL N 或#ASR N； #immed 为执行饱和的位的位置；Rd 为目标寄存器。例如，把一个 32 位有符号数饱和为 16 位有符号数，可以使用下面的指令:

- SSAT R1, #16, R0

表 6-38 列出了 R0 寄存器存储不同数值时上述 SSAT 指令运算的结果。

表 6.35 有符号饱和结果示例

| 执行执行前 R0 数值 | 指令执行后 R1 数值 |
|---|---|
| 0x00020000 | 0x00007FFF |
| 0x00008000 | 0x00007FFF |
| 0x00007FFF | 0x00007FFF |
| 0x00000000 | 0x00000000 |
| 0xFFFF8000 | 0xFFFF8000 |
| 0xFFFF7FFF | 0xFFFF8000 |
| 0xFFFE0000 | 0xFFFF8000 |

USAT 则稍微有些不同，它的结果为无符号数据。其饱和运算的情况如图 6.12 (c)。例如，可以利用下面的代码将一个 32 位有符号数转换为 16 位无符号数:

- USAT R1, #16, R0

表 6.36 列出了几种 R0 寄存器存储不同数时上述 USAT 指令运算的结果。

表 6.36 无符号饱和结果示例

| 执行执行前 R0 数值 | 指令执行后 R1 数值 |
|---|---|
| 0x00020000 | 0x0000FFFF |
| 0x00008000 | 0x00008000 |
| 0x00007FFF | 0x00007FFF |
| 0x00000000 | 0x00000000 |
| 0xFFFF8000 | 0x00000000 |
| 0xFFFF8001 | 0x00000000 |
| 0xFFFFFFFF | 0x00000000 |

### 6.4.11 其他杂类指令

除了前述指令外，有些指令不易归类，如表 6.37 所示。这些杂类指令有些用于设置处理器的休眠模式，有些用于配置存储器的访问顺序，还有些和处理器状态设置有关。

表 6.37 Cortex-M3/M4 的杂类指令

| 助记符 | 英文名称 | 中文名称 |
|---|---|---|
| BKPT | Breakpoint | 断点 |
| CPSID | Change Processor State, Disable Interrupts | 改变处理器状态并禁止中断 |
| CPSIE | Change Processor State, Enable Interrupts | 改变处理器状态并使能中断 |
| DMB | Data Memory Barrier | 数据内存屏障：在 DMB 指令之前的内存访问完成后，才可以执行 DMB 之后的内存访问相关的指令。 |
| DSB | Data Synchronization Barrier | 数据同步屏障：在 DSB 指令之前的指令完成后才可以执行 DSB 指令后的指令。 |
| ISB | Instruction Synchronization Barrier | 指令同步隔离：清空流水线，ISB 指令之后执行的指令都是从内存或缓存中获得的。 |
| NOP | No Operation | 空指令 |
| SEV | Send Event | 发送事件，多处理器系统中用于向其他处理器传递信号 |
| SVC | Supervisor Call | 产生 SVC 异常，常用于操作系统中请求特权操作或访问系统资源。 |
| WFE | Wait For Event | 有条件地进入休眠状态 |
| WFI | Wait For Interrupt | 等待中断（进入休眠状态） |

表 6.37 所示指令多数不含操作数，有些支持立即数方式的操作数。指令的语法相对简单，如下为一些示例。

- NOP ;什么也不做
- BKPT #<immed> ;产生断点异常，指令中的 8 位立即数可由调试器提取
- SEV ;发送事件，多处理器系统中可用于向其他处理器传递信号
- SVC #<immed> ;产生 SVC 异常，立即数指示要请求的服务
- WFI ;等待中断（进入休眠）
- WFE ;等待事件（有条件地进入休眠）

下面对表 6.37 所示指令的用途做简要说明，各指令的详细说明可参阅 Cortex-M3/Cortex-M4 设备用户指南和 ARM®v7-M 架构参考手册。

### 1. 存储器屏障指令 DMB、DSB、ISB

ARM7TDMI 等经典处理器按照程序代码的先后顺序来执行指令或访问数据。但较新架构的 ARM 处理器可以对执行指令和访问数据的顺序进行优化。例如，具有超标量或乱序执行能力的处理器中，检测到当前访问存储器指令需要等待，而下一条指令并不依赖当前指令执行结果时，则不等待当前指令执行完而直接执行下一条指令。

在多处理器共享数据的情形，这种对存储器访问的重新排序，可能会引起错误。故而有时不希望发生这种乱序执行的处理器优化，此时可通过内存屏障指令（Memory barrier instructions）来指示，告知处理器存储器访问的顺序和程序代码顺序要保持一致。Cortex-M3/M4 中，支持 DMB、DSB、ISB 三种存储器屏障指令，其功能如表 6.37 所示。

### 2. 休眠模式指令 WFI、WFE

Cortex-M3/M4 处理器可显式调用休眠指令进入休眠模式，也可在异常退出时进入休眠。WFI 指令使处理器立即进入休眠模式，中断、复位或调试操作可以将处理器从休眠中唤醒。WFE 则会使处理器有条件地进入休眠。

Cortex-M3/M4 处理器中有一个只有一位的寄存器用来记录事件。若该寄存器置位，WFE 指令不会进入休眠而只是清除事件寄存器并继续执行下一条指令；若该寄存器清零，则处理器会进入休眠，该休眠状态可以被事件唤醒，事件可以是中断、调试操作、复位或外部事件输入。

另外一条于休眠模式控制有关的指令是事件输出（SEV）指令。多处理器系统中 SEV 用于向其他处理器传递信号。

### 3. 异常相关指令

管理调用（SVC）指令用于产生 SVC 异常（异常类型为 11）。SVC 一般用于嵌入式操作系统（OS），运行在非特权执行状态的应用可以请求运行在特权状态的 OS 服务。SVC 异常机制提供了从非特权到特权的转换。SVC 机制也可以作为应用任务访问各种服务（包括 OS 服务或其他 API 函数）的入口，这样应用任务就可以在无须了解服务的实际存储器地址的情况下请求所需服务，只需知道 SVC 服务编号、输入参数和返回结果。

SVC 指令要求 SVC 异常的优先级高于当前的优先级，而且异常没有被 PRIMASK 等寄存器屏蔽，不然就会触发错误异常。因此，由于 NMI 和 HardFault 异常的优先级总是比 SVC 异常大，也就无法在这两个处理中使用 SVC。

CPS 是另一条和异常相关的指令，用来改变处理器状态。使用 CPS 可以设置或清除 PRIMASK 和 FAULTMASK 等中断屏蔽寄存器。CPS 指令使用时必须要带后缀 IE（中断使能）或 ID（中断禁止）。使用 CPS 切换 PRIMASK 和 FAULTMASK 可禁止或使能中断，经常用于确保时序关键的任务在不被打断的情况下快速完成。如下为具体使用示例。

- CPSID i ；禁止中断并配置 PRIMASK
- CPSID f ；禁止中断并配置 FAULTMASK
- CPSIE i ；使能中断并配置 PRIMASK
- CPSIE f ；使能中断并配置 FAULTMASK

#### 4. 空指令和断点指令

Cortex-M 处理器支持 NOP 指令，可用于产生指令延时。需要注意，NOP 产生的延时在不同系统间可能会有差异，精确的延时应使用硬件定时器。

在软件开发/调试过程中，断点（BKPT）指令用于实现程序中的软件断点，一般由调试器插入拟调试的代码。当到达断点时，处理器会被暂停，用户通过调试器执行调试任务。BKPT 指令也可以用于产生调试监控异常，调试器或调试监控异常可以把 BKPT 指令中的立即数提取出来，并根据该信息确定要执行的动作。

### 6.4.12 Cortex-M4 特有指令

与 Cortex-M3 处理器相比，Cortex-M4 支持的指令更多。新增的功能包括：单指令多数据、饱和指令、其他的乘法和 MAC 指令、打包和解包指令、可选的浮点指令（若浮点单元存在）等。这些指令可以让 Cortex-M4 高效地进行实时数字信号处理。由于涉及到较多的指令，下面仅选取一部分指令来说明这些 Cortex-M4 特有指令在数字信号处理方面的优势，以便读者了解 Cortex-M4 的功能。

#### 1. SIMD 和饱和指令

32 位处理器的默认数据操作单位是 32 比特，但很多应用中，待处理数据往往是八比特或 16 比特的（如图像中像素点的颜色值，或声音信号的样本）。为了能够使用 32 位的处理器在一条指令内同时处理四个八比特数据，或两个 16 比特数据，就产生了单指令多数据（SIMD, Single Instrution Multiple Data）的方法。一个 32 位寄存器可用作四种类型的 SIMD 数据，如图 6-13 所示。

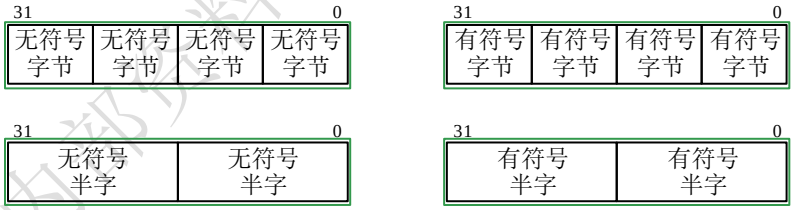


图 6.13 32 位寄存器中各种可能的 SIMD 数据类型

通常 SIMD 指令操作的每个数据的类型都是一样的，不存在有符号和无符号数据混用、16 位和八位数混用的情况，这样可以简化 SIMD 指令集的设计。ARM 的处理器和英特尔处理器对 SIMD 指令都是按照这样的原则设计的。如表 6.38 所示为 Cortex-M4 中支持的 SIMD 指令，同时处理的数据类型都是相同的，或为四个无符号八比特数，或为两个 16 比特数，等等。

表 6.38 Cortex-M4 的 SIMD 指令的基本运算

| 指令 | 指令功能 |
|---|---|
| ADD8 | 4 对 8 位数加法 |
| SUB8 | 4 对 8 位数减法 |

| | |
|---|---|
| ADD16 | 2 对 16 位数相加 |
| SUB16 | 减 2 对 16 位数 |
| ASX | 交换第二个操作数寄存器的半字，然后把两个操作数的高半字相加，两个操作数的低半字相减 |
| SAX | 交换第二个操作数寄存器的半字，然后把两个操作数的低半字相加，两个操作数的高半字相减 |

SIMD 指令可以结合饱和功能，使用不同的前缀表示指令用于有符号数或无符号数，如表 6.39 所示。

表 6.39 用前缀区分的带饱和操作的运算指令

| 前缀 / 操作 | S Signed 有符号 | U Unsigned 无符号 | Q Saturating 有符号饱和 | UQ Unsigned saturating 无符号饱和 | SH Signed halving 有符号半分 | UH Unsigned halving 无符号半分 |
|---|---|---|---|---|---|---|
| | GE 位更新 | | 饱和时 Q 置位 | | 每个数据会除 2 | |
| ADD8 | SADD8 | UADD8 | QADD8 | UQADD8 | SHADD8 | UHADD8 |
| SUB8 | SSUB8 | USUB8 | QSUB8 | UQSUB8 | SHSUB8 | UHSUB8 |
| ADD16 | SADD16 | UADD16 | QADD16 | UQADD16 | SHADD16 | UHADD16 |
| SUB16 | SUB16 | USUB16 | QSUB16 | UQSUB16 | SHSUB16 | UHSUB16 |
| ASX | SASX | UASX | QASX | UQASX | SHASX | UHASX |
| SAX | SSAX | USAX | QSAX | UQSAX | SHSAX | UHSAX |

除了表 6.38 和表 6.39 所示指令外，还有一些 SIMD 与饱和指令此处并未列出，也没有给出上述指令的语法，详情可参阅 Cortex-M4 设备用户指南和 ARM®v7-M 架构参考手册。

## 2. 乘法和 MAC 指令

数字信号处理算法中常用到乘并累加的运算。乘法和 MAC 运算的指令有很多条，在本章前面一节中介绍了一些 Cortex-M3 和 Cortex-M4 处理器都支持的乘法和 MAC 指令，如表 6.40 所示。

表 6.40 Cortex-M3/M4 都支持的乘法和 MAC 指令

| 指令 | 描述(大小) | 标志 |
|---|---|---|
| MUL/MULS | 无符号乘法(32b×32b=32b) | 无，或 N 和 Z |
| UMULL | 无符号乘法(32b×32b=64b) | 无 |
| UMLAL | 无符号 MAC((32b×32b)+64b=64b) | 无 |
| SMULL | 有符号乘法(32b×32b=64b) | 无 |
| SMLAL | 有符号 MAC(32b×32b)+64b=64b) | 无 |

除此之外，Cortex-M4 处理器还支持额外的乘法和 MAC 指令，指令常具有多种形式，可以选择输入参数的低半字和高半字，表 6.41 以有符号数乘法 SMUL 和有符号数 MAC 运算为例，向读者展示此类指令灵活的执行方式。更多的指令以及指令的语法此处不再赘述，详情可参阅 Cortex-M4 设备用户指南和 ARM®v7-M 架构参考手册。

表 6.41 乘法和 MAC 运算的操作数

| 指令 | 描述(大小) |
|---|---|
| SMULxy | 有符号乘法(16b×16b=32b) |
| | SMULBB：第 1 操作数低半字×第 2 操作数低半字 |
| | SMULBT：第 1 操作数低半字×第 2 操作数高半字 |
| | SMULTB：第 1 操作数高半字×第 2 操作数低半字 |
| | SMULTT：第 1 操作数高半字×第 2 操作数高半字 |
| SMLAxy | 有符号 MAC((16b×16b)+32b=32b) |
| | SMLABB：(第 1 操作数低半字×第 2 操作数低半字)+累加字 |
| | SMLABT：(第 1 操作数低半字×第 2 操作数高半字)+累加字 |
| | SMLATB：(第 1 操作数高半字×第 2 操作数低半字)+累加字 |
| | SMLATT：(第 1 操作数高半字×第 2 操作数高半字)+累加字 |

### 3. 打包和解包

SIMD 指令涉及到同时操作多个数据，把待处理的多个数据整合到一个 32 位寄存器的过程常称作数据的打包（packing），其逆过程称为解包（unpacking）。表 6.42 列出了少数几条与打包和解包有关的指令，更多指令可参阅 Cortex-M4 设备用户指南和 ARM®v7-M 架构参考手册。其中有些指令可支持第二个操作数桶形移位或循环移位。移位或循环移位是可选的，下面表格中用于循环移位（ROR）的 *n* 可以为 8、16 或 24。PKHBT 和 PKHTB 可以进行任意数量的移位。

表 6.42 打包和解包指令示例

| 助记符 | 操作数 | 简介 |
|---|---|---|
| PKHBT | {Rd,} Rn, Rm{, LSL #imm } | 把 Rn 的低半字和 Rm（或 Rm 移位后结果）的高半字打包至 Rd |
| PKHTB | {Rd,}Rn, Rm{, ASR #imm } | 把 Rn 的高半字和 Rm（或 Rm 移位后结果）的低半字打包至 Rd |
| SXTB16 | Rd, Rm {, ROR #n} | Rm（或 Rm 移位后结果）最低字节、最高字节分别做符号位扩展至 16 位后打包存入 Rd |
| SXTAB16 | {Rd,} Rn, Rm {,ROR#n } | Rm（或 Rm 移位后结果）最低字节、最高字节分别做符号位扩展至 16 位后分别与 Rn 的低半字、高半字相加后存入 Rd |
| UXTAH | {Rd,} Rn, Rm {,ROR#n } | Rm（或 Rm 移位后结果）低半字做零扩展至 32 位后与 Rn 相加，结果存入 Rd |

### 4. 浮点运算

数字信号处理算法常需要浮点数运算。在有浮点运算单元的 Cortex-M4 中，支持多条用于浮点数据处理和浮点数据传输的指令。浮点数运算经常涉及到无符号数与有符号数的相互转换、整数与小数的相互转换、单精度浮点数和双精度浮点数的相互转换；同一条指令源操作数和目标操作数类型不同时也需要经过不同的处理，故在指令集中通过在助记符后添加后缀来区分操作数的类型。表 6.43 列出了少数浮点运算指令，以便于读者了解浮点数运算指令的一些功能及其常用后缀，完整的指令列表可查阅 Cortex-M4 设备用户指南和 ARM®v7-M 架构参考手册。浮点指令都是以字母 V 开头的。

表 6.43 浮点指令示例

| 助记符 | 操作数 | 简介 |
|---|---|---|
| VADD.F32 | {Sd, } Sn, Sm | 浮点加法 |
| VFMA.F32 | Sd, Sn, Sm | 浮点融合乘累加 Sd=Sd+(Sn *Sm) |
| VLDR.32 | Sd, [Rn{,#imm}] | 从存储器中加载一个单精度数据 |
| VLDR.64 | Dd, [PC,#imm] | 从存储器中加载一个双精度数据 |
| VMLA.F32 | Sd, Sn, Sm | 浮点乘累加 Sd=Sd+(Sn *Sm) |
| VMOV{.F32} | Sd, Sm | 复制浮点寄存器 Sm 到 Sd（单精度） |
| VMRS.F32 | Rt, FPCSR | 复制浮点单元系统寄存器 FPSCR 中的数据到 Rt |
| VMUL.F32 | {Sd, } Sn, Sm | 浮点乘法 |
| VPUSH.64 | {D_regs} | 浮点双精度寄存器压栈 |
| VPOP.32 | {S_regs} | 浮点单精度寄存器出栈 |
| VSQRT.F32 | Sd, Sm | 浮点平方根 |
| VSTMIA.32 | Rn{!}, <S_regs> | 浮点多存储后地址增加 |
| VSTMDB.64 | Rn{!}, <D_regs> | 浮点多存储前地址减小 |

## 6.5 习题

[6-1] 名词解释：指令，指令系统，汇编语言，指令字。

[6-2] ARM 公司的指令集有哪些？

[6-3] ARM 公司 T32 指令集如何区分 16 位指令和 32 位指令？

[6-4] ARM 处理器中“字（Word）”、“半字（Halfword）”是如何定义的。

[6-5] 名词解释：指令二进制编码格式。

[6-6] 一条机器指令应该包含哪些要素？

[6-7] 请解释 ARM 处理器的条件执行指令。

[6-8] 解释 APSR 寄存器标志位 N、Z、C、V、Q 取值的含义。

[6-9] 请依据指令功能在下表中填写指令操作码的助记符和空缺的中文描述。

| 助记符 | 指令功能英文描述 | 指令功能中文描述 |
|---|---|---|
| | Add | 加法 |
| | Add with Carry | |
| | Subtract | 减法 |
| | Multiply, 32-bit result | |
| | Signed Divide | |
| | Multiply with Accumulate | |
| | Multiply and Subtract | |
| | Arithmetic Shift Right | |
| | Logical Shift Right | |
| | Move | 数据移动 |
| | Load Register with word | 将 32 比特数值存入寄存器 |
| | Store Register word | 将一个寄存器的 32 比特数值存入存储器 |
| | Load Multiple registers | 将多个 32 比特数值存入多个寄存器 |
| | Store Multiple registers | 将多个寄存器的数值存入特定存储器区域 |
| | Branch | |
| | Branch with Link | |
| | Branch indirect with Link | |
| | Compare | |

| | | |
|---|---|---|
| | If-Then condition block | |

[6-10] 名词解释：寻址方式，操作数寻址，指令寻址，立即数，立即数寻址，寄存器寻址，寄存器移位寻址。

[6-11] Cortex-M3 处理器指令系统对立即数有什么限制？

[6-12] 指令中的操作数可以存放在哪些地方？

[6-13] 解释寄存器寻址和寄存器间接寻址的区别。

[6-14] 寄存器偏移寻址方式中地址偏移量有哪几种形式？

[6-15] 对比寄存器偏移寻址、前变址寻址和后变址寻址的异同。

[6-16] 前变址寻址汇编语法为“opcode [<Rn>,<offset>]!”，解释其中“!”的含义。

[6-17] 解释如下两条指令的含义。

LDR R0, [R1, #0xA]

LDR R0, [R1, R4]

[6-18] 解释如下两条指令的含义。

LDR R0, [R9, R1]

LDR R0, [R1, R3]!

[6-19] 解释如下两条指令的含义。

LDR R0, [R1, #0xB]

LDR R0, [R1, #0xB]!

LDR R0, [R1], #0xB

[6-20] 解释汇编语法“LDM {addr_mode} <Rn>{!}, <registers>”中各部分要素的含义。

[6-21] 解释如下四条指令的含义。

STMIA R9, {R1-R4}

STMIB R9!, {R1-R4}

STMDA R9!, {R1-R4}

STMDB R9!, {R1-R4}

[6-22] 解释如下两条指令的含义。

STMFD SP!, {R1-R8}

STMDB SP!, {R1-R8}

[6-23] 解释如下两条指令的含义。

LTMFD SP!, {R1-R8}

LDMIA SP!, {R1-R8}

[6-24] 解释堆栈寻址和 PC 相对寻址两种方式中基地址各存放在哪里？

[6-25] 一般处理器指令集应包含哪些类别的指令？

[6-26] 请写出功能“从[R1]+0x3 中读取一个字节并将其存入 R8”的汇编指令。

[6-27] 请写出功能“从[R1]+0x4 中读取一个半字并将其存入 R8”的汇编指令。

[6-28] 请写出功能“从[R1]+0x8 中读取一个字并将其存入 R8”的汇编指令。

[6-29] 请写出功能“R0 中数值的第 2 位取反”的汇编指令。

[6-30] 请写出功能“R0 中数值的第 2 位置 1”的汇编指令。

[6-31] 请写出功能“R0 中数值的第 2 位置 0”的汇编指令。

[6-32] 请写出功能“R0 和 R1 中数值按位与”的汇编指令。

[6-33] 请解释 AND、ORR、ORN、EOR 的区别。

[6-34] 什么是符号扩展？

[6-35] MOV 指令是否可以完成从一个存储器单元到一个寄存器的数据传送？为什么？

[6-36] 解释“LSR”和“ASR”的区别。

[6-37] 解释“LSR”和“ROR”的区别。

[6-38] 解释“ROR”和“RRX”的区别。

[6-39] 解释“CMP”和“CMN”的区别。

[6-40] 为什么“BL”或“BLX”指令适用于函数调用，而“B”指令不适合。

[6-41] 解释使用指令“B”或“BL”进行条件跳转和“MOVEQ”条件执行指令的区别。

[6-42] “CBZ”指令的作用与“CMP”指令组合“BEQ”指令有什么区别？

[6-43] 解释饱和和溢出的区别。

[6-44] 定性解释存储器屏障指令的作用。

[6-45] 为什么 ARM 的存储器访问指令要提供非特权加载和存储功能？

[6-46] *阐述 SIMD 的产生背景和技术思想。

[6-47] *乘并且累加的指令适用于什么样的算法？

[6-48] *在什么情况下可以用相同的电路完成整数和定点小数的运算？

[6-49] *浮点数的运算电路和定点小数的运算电路有哪些区别？

[6-50] *为什么指令系统中乘法和除法往往用不同的指令对无符号数和有符号数进行运算？

[6-51] *为什么在支持数字信号处理的处理器中要提供“符号扩展”和“补零”的功能？

[6-52] *请思考为什么 ARM 处理器的数据扩展对无符号数和有符号数要用不同的指令？