第4章

# 指令集

* 汇编语言基础
* 指令集
* 近距离地检视指令
* Cortex-M3中的一些新好指令

终于“开荤”了，本章开始把Cortex-M3的指令系统展现出来，并且给出了一些简单却意味深长的例子。在本书的附录A中还有一个快速查阅参考。指令集的详细信息由《ARMv7-M Architecture Application Level Reference Manual》(Ref2)给出——写了两百多页呢。

如果读者以前没有写过ARM汇编程序，可以结合看本书的第20章，那里讲述了Keil RVMDK工具的使用，包括添加汇编源文件的方法。RVMDK带了一个指令模拟器，对于练习汇编程序非常有帮助。那一章虽然不是很短但很简单。值得一提的是，在那一章的末尾，译者添加了少量内容，是专为学习第4章而添加的。

## 4.1 汇编语言基础

为了给本章的学习扫清障碍，这里我们先简要地介绍一下ARM汇编器的基本语法。本书绝大多数的汇编示例都使用ARM汇编器的语法，而第19章则使用GCC汇编器AS的语法。

### 4.1.1 汇编语言：基本语法

汇编指令的最典型书写模式如下所示：

**标号**

**操作码 操作数1, 操作数2, … ;注释**

其中，标号是可选的，如果有，它必须顶格写。标号的作用是让汇编器来计算程序转移的地址。

操作码是指令的助记符，它的前面必须有至少一个空白符，通常使用一至二个“Tab”键来产生。操作码后面往往跟随若干个操作数，而第1个操作数，通常都给出本指令的执行结果存储处。不同指令需要不同数目的操作数，并且对操作数的语法要求也可以不同。举例来说，立即数必须以“#”开头，如

MOV R0, #0x12 ; R0 🡨 0x12

MOV R1, #’A’ ; R1 🡨 字母A的ASCII码

注释均以”;”开头，它的有无不影响汇编器工作，只是给程序员看的，能让程序更易理解。

还可以使用EQU指示字来定义常数，然后在代码中使用它们，例如：

NVIC\_IRQ\_SETEN0 EQU 0xE000E100 ; 注意：常数定义必须顶格写

NVIC\_IRQ0\_ENABLE EQU 0x1

…

LDR R0, =NVIC\_IRQ\_SETEN0 ;在这里的LDR是个伪指令，它会被汇编器转换成

;一条“相对PC的加载指令”

MOV R1, #NVIC\_IRQ0\_ENABLE ; 把立即数传送到R1中

STR R1, [R0] ; \*R0=R1，执行完此指令后IRQ #0被使能。

如果汇编器不能识别某些特殊指令的助记符，你就要“手工汇编”——查出该指令的确切二进制机器码，然后使用DCI编译器指示字。例如，BKPT指令的机器码是0xBE00，即可以按如下格式书写：

DCI 0xBE00 ; 断点(BKPT)，这是一个16位指令

(在使用DCI时也必须在前面留出空白符——译注)

类似地，你还可以使用DCB来定义一串字节常数，字节常数还允许以字符串的形式来表达；还可以使用DCD来定义一串32位整数。它们最常被用来在代码中书写表格。例如：

LDR R3, =MY\_NUMBER ; R3= MY\_NUMBER

LDR R4, [R3] ; R4= \*R3

…

LDR R0, =HELLO\_TEXT ; R0= HELLO\_TEXT

BL PrintText ; 呼叫PrintText以显示字符串，R0传递参数

…

MY\_NUMBER

DCD 0x12345678

HELLO\_TEXT

DCB ”Hello\n”,0

请注意：不同汇编器的指示字和语法都可以不同。上述示例代码都是按ARM汇编器的语法格式写的。如果使用其它汇编器，最好看一看它附带的示例代码。

### 4.1.2 汇编语言：后缀的使用

在ARM处理器中，指令可以带有后缀，如表4.1所示。

|  |  |
| --- | --- |
| 后缀名 | 含义 |
| S | 要求更新APSR中的相关标志，例如：  ADDS R0, R1 ; 根据加法的结果更新APSR中的标志 |
| EQ,NE,LT,GT等 | 有条件地执行指令。EQ=Euqal, NE= Not Equal, LT= Less Than, GT= Greater Than，例如：  BEQ <Label> ; 仅当EQ满足时转移  除了这4种，还有若干个其它的条件。 |

在Cortex-M3中，对条件后缀的使用有很大的限制：只有转移指令（B指令）才可随意使用。而对于其它指令，CM3引入了IF-THEN指令块，在这个块中才可以加后缀，且必须加以后缀。IF-THEN块由IT指令定义，本章稍后将介绍它。另外，S后缀可以和条件后缀在一起使用。共有15种不同的条件后缀，稍后介绍。

### 4.1.3 汇编语言：统一汇编语言书写语法

为了最有力地支持Thumb-2，也作为对汇编程序员的人文关怀，ARM汇编器引了一个“统一汇编语言（UAL）”语法机制。对于16位指令和32位指令均能实现的一些操作（常见于数据处理操作），有时虽然指令的实际操作数不同，或者对立即数的长度有不同的限制，但是汇编器允许开发者统一使用32位Thumb-2指令的语法格式书写（很多Thumb-2指令的用法也与32位ARM指令相同），并且由汇编器来决定是使用16位指令，还是使用32位指令。以前，Thumb的语法和ARM的语法不同，在有了UAL之后，两者的书写格式就统一了。

ADD R0, R1 ; 使用传统的Thumb语法

ADD R0, R0, R1 ; 引入UAL后允许的等效写法（R0=R0+R1）

虽然引入了UAL，但是仍然允许使用传统的Thumb语法。不过有一项必须注意：如果使用传统的Thumb语法，有些指令会默认地更新APSR，即使你没有加上S后缀。如果使用UAL语法，则必须指定S后缀才会更新。例如：

AND R0, R1 ;传统的Thumb语法

ANDS R0, R0, R1 ;等值的UAL语法（必须有S后缀）

在Thumb-2指令集中，有些操作既可以由16位指令完成，也可以由32位指令完成。例如，R0=R0+1这样的操作，16位的与32位的指令都提供了助记符为“ADD”的指令。在UAL下，汇编器能主动决定用哪个，也可以手工指定是用16位的还是32位的：

ADDS R0, #1 ;汇编器将为了节省空间而使用16位指令

ADDS.N R0, #1 ;指定使用16位指令（N＝Narrow）

ADDS.W R0, #1 ;指定使用32位指令（W=Wide）

.W(Wide)后缀指定32位指令。如果没有给出后缀，汇编器会先试着用16位指令以给代码瘦身，如果不行再使用32位指令。因此，使用“.N”其实是多此一举，不过汇编器可能仍然允许这样的语法。

再次重申，这是ARM公司汇编器的语法，其它汇编器的可能略有区别，但如果没有给出后缀，汇编器就总是会尽量选择更短的指令。

其实在绝大多数情况下，应用程序是用C写的，C编译器也会尽可能地使用短指令。然而，当立即数超出一定范围时，或者32位指令能更好地适合某个操作，将使用32位指令。

32位Thumb-2指令也可以按半字对齐（以前ARM 32位指令都必须按字对齐——译注），因此下例是允许的：

0x1000: LDR r0, [r1] ;一个16位的指令

0x1002: RBIT.W r0 ;一个32位的指令，以0x1002为起始地址，跨越了字的边界

绝大多数16位指令只能访问R0-R7；32位Thumb-2指令则可以随意访问R0-R15。不过，把R15(PC)作为目的寄存器很容易走火入魔——用对了会有意想不到的妙处，出错时则会使程序跑飞。通常只有系统软件才会不惜冒险地做此高危行为，因此还需慎用。对PC的使用还有其它的戒律，如果感兴趣，可以参考《ARMv7-M架构应用级参考手册》。

## 4.2 指令集

Cortex-M3支持的指令在表4.2至表4.9列出。其中，译者添加了如下格式

边框加粗的是从ARMv6T2才支持的指令。

双线边框的是从Cortex-M3才支持的指令（v7的其它款式不一定支持）

译者添加

在讲指令之前，先简单地介绍一下Cortex-M3中支持的算术与逻辑标志。本书在后面还会展开论述。它们是：

APSR中的5个标志位

* + N: 负数标志(Negative)
  + Z： 零结果标志(Zero)
  + C: 进位/借位标志(Carry)
  + V: 溢出标志(oVerflow)
  + S: 饱和标志(Saturation)，它不做条件转移的依据

### 4.2.1 分类指令表

#### 表4.2 16位数据操作指令

|  |  |
| --- | --- |
| **名字** | **功能** |
| **ADC** | 带进位加法 |
| **ADD** | 加法 |
| **AND** | 按位与（原文为逻辑与，有误——译注）。这里的按位与和C的”&”功能相同 |
| **ASR** | 算术右移 |
| **BIC** | 按位清0（把一个数跟另一个无符号数的反码按位与） |
| **CMN** | 负向比较（把一个数跟另一个数据的二进制补码相比较） |
| **CMP** | 比较（比较两个数并且更新标志） |
| **CPY** | 把一个寄存器的值拷贝到另一个寄存器中 |
| **EOR** | 近位异或 |
| **LSL** | 逻辑左移（如无其它说明，所有移位操作都可以一次移动最多31格——译注） |
| **LSR** | 逻辑右移 |
| **MOV** | 寄存器加载数据，既能用于寄存器间的传输，也能用于加载立即数 |
| **MUL** | 乘法 |
| **MVN** | 加载一个数的NOT值（取到逻辑反的值） |
| **NEG** | 取二进制补码 |
| **ORR** | 按位或（原文为逻辑或，有误——译注） |
| **ROR** | 圆圈右移 |
| **SBC** | 带借位的减法 |
| **SUB** | 减法 |
| **TST** | 测试（执行按位与操作，并且根据结果更新Z） |
| **REV** | 在一个32位寄存器中反转字节序 |
| **REVH** | 把一个32位寄存器分成两个16位数，在每个16位数中反转字节序 |
| **REVSH** | 把一个32位寄存器的低16位半字进行字节反转，然后带符号扩展到32位 |
| **SXTB** | 带符号扩展一个字节到32位 |
| **SXTH** | 带符号扩展一个半字到32位 |
| **UXTB** | 无符号扩展一个字节到32位 |
| **UXTH** | 无符号扩展一个半字到32位 |

#### 表4.3 16位转移指令

|  |  |
| --- | --- |
| **名字** | **功能** |
| **B** | 无条件转移 |
| **B<cond>** | 条件转移 |
| **BL** | 转移并连接。用于呼叫一个子程序，返回地址被存储在LR中 |
| **BLX #im** | 使用立即数的BLX不要在CM3中使用 |
| **CBZ** | 比较，如果结果为0就转移（只能跳到后面的指令——译注） |
| **CBNZ** | 比较，如果结果非0就转移（只能跳到后面的指令——译注） |
| **IT** | If-Then |

#### 表4.4 16位存储器数据传送指令

|  |  |
| --- | --- |
| **名字** | **功能** |
| **LDR** | 从存储器中加载字到一个寄存器中 |
| **LDRH** | 从存储器中加载半字到一个寄存器中 |
| **LDRB** | 从存储器中加载字节到一个寄存器中 |
| **LDRSH** | 从存储器中加载半字，再经过带符号扩展后存储一个寄存器中 |
| **LDRSB** | 从存储器中加载字节，再经过带符号扩展后存储一个寄存器中 |
| **STR** | 把一个寄存器按字存储到存储器中 |
| **STRH** | 把一个寄存器存器的低半字存储到存储器中 |
| **STRB** | 把一个寄存器的低字节存储到存储器中 |
| **LDMIA** | 加载多个字，并且在加载后自增基址寄存器 |
| **STMIA** | 存储多个字，并且在存储后自增基址寄存器 |
| **PUSH** | 压入多个寄存器到栈中 |
| **POP** | 从栈中弹出多个值到寄存器中 |

16数据传送指令没有任何新内容，因为它们是Thumb指令，在v4T时就已经定格了——译注

#### 表4.5 其它16位指令

|  |  |
| --- | --- |
| **名字** | **功能** |
| **SVC** | 系统服务调用 |
| **BKPT** | 断点指令。如果使能了调试，则进入调试状态（停机）。否则的话产生调试监视器异常。在调试监视器异常被使能时，调用其服务例程；如果连调试监视器异常也被除能，则无奈下只好诉诸于一个fault异常 |
| **NOP** | 无操作 |
| **CPSIE** | 使能PRIMASK(CPSIE i)/ FAULTMASK(CPSIE f)——清0相应的位 |
| **CPSID** | 除能PRIMASK(CPSID i)/ FAULTMASK(CPSID f)——置位相应的位 |

#### 表4.6 32位数据操作指令

|  |  |
| --- | --- |
| **名字** | **功能** |
| **ADC** | 带进位加法 |
| **ADD** | 加法 |
| **ADDW** | 宽加法（可以加12位立即数） |
| **AND** | 按位与（原文是逻辑与，有误。对应C言的“|”运算符——译注） |
| **ASR** | 算术右移 |
| **BIC** | 位清零（把一个数按位取反后，与另一个数逻辑与） |
| **BFC** | 位段清零 |
| **BFI** | 位段插入 |
| **CMN** | 负向比较（把一个数和另一个数的二进制补码比较，并更新标志位） |
| **CMP** | 比较两个数并更新标志位 |
| **CLZ** | 计算前导零的数目 |
| **EOR** | 按位异或 |
| **LSL** | 逻辑左移 |
| **LSR** | 逻辑右移 |
| **MLA** | 乘加 |
| **MLS** | 乘减 |
| **MOVW** | 把16位立即数放到寄存器的低16位，高16位清0 |
| **MOV** | 加载16位立即数到寄存器（其实汇编器会产生MOVW——译注） |
| **MOVT** | 把16位立即数放到寄存器的高16位，低16位不影响 |
| **MVN** | 移动一个数的补码 |
| **MUL** | 乘法 |
| **ORR** | 按位或（原文为逻辑或，有误——译注） |
| **ORN** | 把源操作数按位取反后，再执行按位或（原文为逻辑或，有误——译注） |
| **RBIT** | 位反转（把一个32位整数用2进制表达后，再旋转180度——译注） |
| **REV** | 对一个32位整数按字节反转 |
| **REVH/**  **REV16** | 对一个32位整数的高低半字都执行字节反转 |
| **REVSH** | 对一个32位整数的低半字执行字节反转，再带符号扩展成32位数 |
| **ROR** | 圆圈右移 |
| **RRX** | 带进位位的逻辑右移一格（最高位用C填充，执行后不影响C的值——译注） |
| **SBFX** | 从一个32位整数中提取任意长度和位置的位段，并且带符号扩展成32位整数 |
| **SDIV** | 带符号除法 |
| **SMLAL** | 带符号长乘加（两个带符号的32位整数相乘得到64位的带符号积，再把积加到另一个带符号64位整数中） |
| **SMULL** | 带符号长乘法（两个带符号的32位整数相乘得到64位的带符号积） |
| **SSAT** | 带符号的饱和运算 |
| **SBC** | 带借位的减法 |
| **SUB** | 减法 |
| **SUBW** | 宽减法，可以减12位立即数 |
| **SXTB** | 字节带符号扩展到32位数 |
| **TEQ** | 测试是否相等（对两个数执行异或，更新标志但不存储结果） |
| **TST** | 测试（对两个数执行按位与，更新Z标志但不存储结果） |
| **UBFX** | 无符号位段提取 |
| **UDIV** | 无符号除法 |
| **UMLAL** | 无符号长乘加（两个无符号的32位整数相乘得到64位的无符号积，再把积加到另一个无符号64位整数中） |
| **UMULL** | 无符号长乘法（两个无符号的32位整数相乘得到64位的无符号积） |
| **USAT** | 无符号饱和操作（但是源操作数是带符号的——译注） |
| **UXTB** | 字节被无符号扩展到32位（高24位清0——译注） |
| **UXTH** | 半字被无符号扩展到32位（高16位清0——译注） |

#### 表4.7 32位存储器数据传送指令

|  |  |
| --- | --- |
| **名字** | **功能** |
| **LDR** | 加载字到寄存器 |
| **LDRB** | 加载字节到寄存器 |
| **LDRH** | 加载半字到寄存器 |
| **LDRSH** | 加载半字到寄存器，再带符号扩展到32位 |
| **LDM** | 从一片连续的地址空间中加载若干个字，并选中相同数目的寄存器放进去 |
| **LDRD** | 从连续的地址空间加载双字（64位整数）到2个寄存器 |
| **STR** | 存储寄存器中的字 |
| **STRB** | 存储寄存器中的低字节 |
| **STRH** | 存储寄存器中的低半字 |
| **STM** | 存储若干寄存器中的字到一片连续的地址空间中，占用相同数目的字 |
| **STRD** | 存储2个寄存器组成的双字到连续的地址空间中 |
| **PUSH** | 把若干寄存器的值压入堆栈中 |
| **POP** | 从堆栈中弹出若干的寄存器的值 |

#### 表4.8 32位转移指令

|  |  |
| --- | --- |
| **名字** | **功能** |
| **B** | 无条件转移 |
| **BL** | 转移并连接（呼叫子程序） |
| **TBB** | 以字节为单位的查表转移。从一个字节数组中选一个8位**前向**跳转地址并转移 |
| **TBH** | 以半字为单位的查表转移。从一个半字数组中选一个16位**前向**跳转的地址并转移 |

#### 表4.9 其它32位指令

|  |  |
| --- | --- |
| **LDREX** | 加载字到寄存器，并且在内核中标明一段地址进入了互斥访问状态 |
| **LDREXH** | 加载半字到寄存器，并且在内核中标明一段地址进入了互斥访问状态 |
| **LDREXB** | 加载字节到寄存器，并且在内核中标明一段地址进入了互斥访问状态 |
| **STREX** | 检查将要写入的地址是否已进入了互斥访问状态，如果是则存储寄存器的字 |
| **STREXH** | 检查将要写入的地址是否已进入了互斥访问状态，如果是则存储寄存器的半字 |
| **STREXB** | 检查将要写入的地址是否已进入了互斥访问状态，如果是则存储寄存器的字节 |
| **CLREX** | 在本地处理器上清除互斥访问状态的标记（先前由LDREX/LDREXH/LDREXB做的标记） |
| **MRS** | 加载特殊功能寄存器的值到通用寄存器 |
| **MSR** | 存储通用寄存器的值到特殊功能寄存器 |
| **NOP** | 无操作 |
| **SEV** | 发送事件 |
| **WFE** | 休眠并且在发生事件时被唤醒 |
| **WFI** | 休眠并且在发生中断时被唤醒 |
| **ISB** | 指令同步隔离（与流水线和MPU等有关——译注） |
| **DSB** | 数据同步隔离（与流水线、MPU和cache等有关——译注） |
| **DMB** | 数据存储隔离（与流水线、MPU和cache等有关——译注） |

### 4.2.2 未支持的指令

有若干条Thumb指令没有得到Cortex-M3的支持，下表列出了未被支持的指令，以及不支持的原因。

表4.10 因为不再是传统的架构，导致有些指令已失去意义

|  |  |
| --- | --- |
| 未支持的指令 | 以前的功能 |
| BLX #im | 在使用立即数做操作数时，BLX总是要切入ARM状态。因为Cortex-M3只在Thumb态下运行，故以此指令为代表的，凡是试图切入ARM态的操作，都将引发一个用法fault。 |
| SETEND | 由ARMv6引入的，在运行时改变处理器端设置的指令（大端或小端）。因为Cortex-M3不支持动态端的功能，所以此指令也将引发fault |

CM3也不支持有少量在ARMv7-M中列出的指令。比如，ARMv7M支持Thumb2的协处理器指令，但是CM3却不能挂协处理器。表4.11列出了这些与协处理器相关的指令。如果试图执行它们，则将引发用法fault（NVIC中的NOCP（No CoProcessor）标志置位）。

表4.11 不支持的协处理器相关指令

|  |  |
| --- | --- |
| 未支持的指令 | 以前的功能 |
| MCR | 把通用寄存器的值传送到协处理器的寄存器中 |
| MCR2 | 把通用寄存器的值传送到协处理器的寄存器中 |
| MCRR | 把通用寄存器的值传送到协处理器的寄存器中，一次操作两个 |
| MRC | 把协处理器寄存器的值传送到通用寄存器中 |
| MRC2 | 把协处理器寄存器的值传送到通用寄存器中 |
| MRRC | 把协处理器寄存器的值传送到通用寄存器中，一次操作两个 |
| LDC | 把某个连续地址空间中的一串数值传送至协处理器中 |
| STC | 从协处理器中传送一串数值到地址连续的一段地址空间中 |

还有一个是改变处理器状态指令（CPS），它的一些用法也不再支持。这是因为PSRs的定义已经变了，以前在ARMv6中定义的某些位在CM3中并不存在。

表4.12 不支持的CPS指令用法

|  |  |
| --- | --- |
| 未支持的指令 | 以前的功能 |
| CPS<IE/ID>.W A | CM3没有“A”位 |
| CPS.W #mode | CM3的PSR中没有“mode”位 |

有些提示（hint）指令的功能不支持，它们在CM3中按“NOP”指令对待

表4.13 不支持的hint指令

|  |  |
| --- | --- |
| 未支持的指令 | 以前的功能 |
| DBG | 服务于跟踪系统的一条hint指令 |
| PLD | 预取数据。这是服务于cache系统的一条hint指令。因为在CM3中没有cache，该指令就相当于NOP |
| PLI | 预取指令。这是服务于cache系统的一条hint指令。因为在CM3中没有cache，该指令就相当于NOP |
| YIELD | 用于多线程处理。线程使用该指令通知给硬件：我正在做的任务可以被交换出去（swapped out），从而提高系统的整体性能。 |

## 4.3 近距离检视指令

从现在起，我们将介绍一些在ARM汇编代码中很通用的指令及其语法。有些指令可以带有多种附加处理，比如预移位操作。本章不会讲得面面惧到，但理解本章后足以应付大多数大型汇编程序开发。

### 4.3.1 汇编语言：数据传送

处理器的基本功能之一就是数据传送。CM3中的数据传送类型包括

* + 在两个寄存器间传送数据
  + 在寄存器与存储器间传送数据
  + 在寄存器与特殊功能寄存器间传送数据
  + 把一个立即数加载到寄存器

用于在寄存器间传送数据的指令是MOV。比如，如果要把R3的数据传送给R8，则写作：

MOV R8, R3

MOV的一个衍生物是MVN，它把寄存器的内容取二进制补码后再传送。

用于访问存储器的基础指令是“加载（Load）”和“存储（Store）”。加载指令LDR把存储器中的内容加载到寄存器中，存储指令STR则把寄存器的内容存储至存储器中，传送过程中数据类型也可以变通，最常使用的格式有：

表 4.14 常用的存储器访问指令

|  |  |
| --- | --- |
| 示例 | 功能描述 |
| LDRB Rd, [Rn, #offset] | 从地址Rn+offset处读取一个字节送到Rd |
| LDRH Rd, [Rn, #offset] | 从地址Rn+offset处读取一个半字送到Rd |
| LDR Rd, [Rn, #offset] | 从地址Rn+offset处读取一个字送到Rd |
| LDRD Rd1, Rd2, [Rn, #offset] | 从地址Rn+offset处读取一个双字(64位整数)送到Rd1（低32位）和Rd2（高32位）中。 |
| STRB Rd, [Rn, #offset] | 把Rd中的低字节存储到地址Rn+offset处 |
| STRH Rd, [Rn, #offset] | 把Rd中的低半字存储到地址Rn+offset处 |
| STR Rd, [Rn, #offset] | 把Rd中的低字存储到地址Rn+offset处 |
| STRD Rd1, Rd2, [Rn, #offset] | 把Rd1（低32位）和Rd2（高32位）表达的双字存储到地址Rn+offset处 |

如果嫌一口一口地蚕食太不过瘾，也可以使用LDM/STM来鲸吞。它们相当于把若干个LDR/STR给合并起来了，有利于减少代码量，如表4.15所示

表 4.15 常用的多重存储器访问方式

|  |  |
| --- | --- |
| 示例 | 功能描述 |
| LDMIA Rd!, {寄存器列表} | 从Rd处读取多个字，并依次送到寄存器列表中的寄存器。每读一个字后Rd自增一次，16位宽度 |
| STMIA Rd!, {寄存器列表} | 依次存储寄存器列表中各寄存器的值到Rd给出的地址。每存一个字后Rd自增一次，16位宽度 |
| LDMIA.W Rd!, {寄存器列表} | 从Rd处读取多个字，并依次送到寄存器列表中的寄存器。每读一个字后Rd自增一次，32位宽度 |
| LDMDB.W Rd!, {寄存器列表} | 从Rd处读取多个字，并依次送到寄存器列表中的寄存器。每读一个字前Rd自减一次，32位宽度 |
| STMIA.W Rd!, {寄存器列表} | 依次存储寄存器列表中各寄存器的值到Rd给出的地址。每存一个字后Rd自增一次，32位宽度 |
| STMDB.W Rd!, {寄存器列表} | 存储多个字到Rd处。每存一个字前Rd自减一次，32位宽度 |

译注：上表中，加粗的是符合CM3堆栈操作的LDM/STM使用方式。并且，如果Rd是R13（即SP），则与POP/PUSH指令等效。(LDMIA->POP, STMDB -> PUSH)

STMDB SP!, {R0-R3, LR} 等效于 PUSH {R0-R3, LR}

LDMIA SP!, {R0-R3, PC} 等效于 POP {R0-R3, PC}

Rd后面的“！”是什么意思？它表示要自增(Increment)或自减（Decrement）基址寄存器Rd的值，时机是在每次访问前(Before)或访问后(After)。增/减单位：字（4字节）。例如，记R8=0x8000，则下面两条指令：

STMIA.W R8!, {r0-R3} ; R8值变为0x8010，每存一次增一次，先存储后自增

STMDB.W R8, {R0-R3} ; R8值的“一个内部复本”先自减后再存储数据，但R8的值不变

感叹号还可以用于单一加载与存储指令——LDR/STR。这也就是所谓的 “带预索引”(Pre-indexing)的LDR和STR。例如：

LDR.W R0, [R1, #20]! ;预索引

该指令先把地址R1+offset处的值加载到R0，然后，R1 🡨 R1+ 20（offset也可以是负数——译注）。这里的“！”就是指在传送后更新基址寄存器R1的值。“！”是可选的。如果没有“！”，则该指令就是普通的带偏移量加载指令，不会自动调整R0的值。带预索引的数据传送可以用在多种数据类型上，并且既可用于加载，又可用于存储。

表4.16 预索引数据传送的常见用法

|  |  |
| --- | --- |
| 示例 | 功能描述 |
| LDR.W Rd, [Rn, #offset]！  LDRB.W Rd, [Rn, #offset]！  LDRH.W Rd, [Rn, #offset]！  LDRD.W Rd1, Rd2, [Rn, #offset]！ | 字/字节/半字/双字的带预索引加载（不做带符号扩展，没有用到的高位全清0——译注） |
| LDRSB.W Rd, [Rn, #offset]！  LDRSH.W Rd, [Rn, #offset]！ | 字节/半字的带预索引加载，并且在加载后执行带符号扩展成32位整数 |
| STR.W Rd, [Rn, #offset]！  STRB.W Rd, [Rn, #offset]！  STRH.W Rd, [Rn, #offset]！  STRD.W Rd1, Rd2, [Rn, #offset]！ | 字/字节/半字/双字的带预索引存储 |

CM3除了支持“预索引”外，还支持“后索引”(Post-indexing)。后索引也要使用一个立即数offset，但与预索引不同的是，后索引忠实地使用基址寄存器Rd的值，把它作为传送的目的地址。待到数据传送后，再执行Rd 🡨 Rd+offset（offset可以是负数——译注）。如：

STR.W R0, [R1], #-12 ;后索引

该指令是把R0的值存储到地址R1处的。在存储完毕后， R1 🡨 R1+(-12)

注意，[R1]后面是没有“！”的。可见，在后索引中，基址寄存器是无条件被更新的——也可以理解为有一个“隐藏”的“！”

表4.17 后索引的常见用法

|  |  |
| --- | --- |
| 示例 | 功能描述 |
| LDR.W Rd, [Rn], #offset  LDRB.W Rd, [Rn], #offset  LDRH.W Rd, [Rn], #offset  LDRD.W Rd1, Rd2, [Rn], #offset | 字/字节/半字/双字的带预索引加载（不做带符号扩展，没有用到的高位全清0——译注） |
| LDRSB.W Rd, [Rn], #offset]  LDRSH.W Rd, [Rn], #offset] | 字节/半字的带预索引加载，并且在加载后执行带符号扩展成32位整数 |
| STR.W Rd, [Rn], #offset  STRB.W Rd, [Rn], #offset  STRH.W Rd, [Rn], #offset  STRD.W Rd1, Rd2, [Rn], #offset | 字/字节/半字/双字的后预索引存储 |

译者添加

立即数的位数是有限制的，且不同指令的限制可以不同。这下岂不是要有的背了？其实不必！因为如果在使用中超过了限制，则汇编器会报错，所以不用担心会背成书呆子。

那能彻底消灭这种限制吗？办法是有的，只是要使用另一种形式的LDR/STR。事实上，在CM3中的偏移量，除了可以使用形如#offset的立即数，还可以使用一个寄存器。使用寄存器来提供偏移量，就可以“天南地北任我行”了。不过，如果使用寄存器提供偏移量，就不能使用“预索引”和“后索引”了——也就是说不能修改基址寄存器的值。因此下面的写法就是非法的：

ldr r2, [r0, r3]！ ;错误，寄存器提供偏移量时不支持预索引

ldr r2, [r0], r3 ;错误，寄存器提供偏移量时不支持后索引

这看起来令人扫兴吗？不过也有好消息。当使用寄存器作索引时，可以“预加工”索引寄存器的值——逻辑左移。显然，这与C语言数组下标的寻址方式刚好吻合，如

ldr r2, [r0, r3, lsl #2]

如果r3给出了某32位整数数组的下标，则这条指令即可取出该下标处的数组元素。还有一个注意事项：左移的位数只能是1、2或者3。（最常用的就是2，对应32位整数）。

译注：PUSH/POP作为堆栈专用操作，也属于数据传送指令类。

通常PUSH/POP对子的寄存器列表是严格一致的，但是PC与LR的使用方式有新意，如

;子程序入口

PUSH {R0-R3, LR}

…

;子程序出口

POP {R0-R3, PC}

在这个例子中，旁路了LR，直截了当地返回。

数据传送指令还包括MRS/MSR。还记得第3章讲到过CM3有若干个特殊功能寄存器吗？MRS/MSR就是专门用于访问这些寄存器的。不过，这些寄存器都是维持系统正常工作的重地，因此，按理说是不能允许随意访问它们的。然而，在CM3上，它们是“春色满园关不住，一枝红杏出墙来”——大原则是必须在特权级下才允许访问，以免系统因误操作或恶意破坏而功能紊乱，可是APSR却允许在用户级下访问，想必是为了让我们有妙用非主流技巧的余地吧。CM3的这个禁律是由硬件强制执行的，如果在用户级下以身试法，则fault伺候（产生MemManage fault，若被除能则“上访”成硬fault）［译注］。通常，只有系统软件（如OS）才会操作这类寄存器，应用程序，尤其是用C编写的应用程序，是从来不关心这些的。

译注：当使用MRS访问APSR时，是把各个标志位按照它们在xPSR中占用的位序号，直接复制到寄存器中的。例如，进位标志C是在xPSR.30中，在执行了MRS R0, APSR指令后，则R0.30=C；反之亦然，欲设置C位，必须在R0.30中给出新的C的值。另外，译者在模拟器和STM32单片机中尝试了在用户级下访问PSR，并没有产生fault，只是改写APSR以外的部分被忽略而已。但我觉得不必太过争论处理器的具体处理方式，因为不管怎么说这都是错误的编程行为。

下面轮到立即数上场。代码写多了我们就常常会感觉到，程序中会经常使用立即数。最典型的就是：当我们要访问某个地址时，必须先把该地址加载到一个寄存器中，这就包含了一个32位立即数加载操作。CM3中的MOV/MVN指令族负责加载立即数，族中各个成员支持的立即数位数不同。例如，16位指令MOV支持8位立即数加载，如：

MOV R0, #0x12

32位指令MOVW和MOVT可以支持16位立即数加载。

那要加载32位立即数怎么办呢？如果要直来直去，当前是要用两条指令来完成了。通过组合使用MOVW和MOVT就能产生32位立即数，但是要注意，必须先使用MOVW，再使用MOVT。这种顺序是不能颠倒的，因为MOVW会清零高16位。

不过，更流行的是另一种方法：使用汇编器提供的”LDR Rd, = imm32”伪指令。例如：

LDR, r0, **=**0x12345678

酷吧！它的名字也是LDR，而且能加载32位立即数！但可别忘了，它是伪指令，是“妖怪变的”，而且有若干种原形。所以不要因为名字相同就混淆。

大多数情况下，当汇编器遇到LDR伪指令时，都会把它转换成一条相对于PC的加载指令，来产生需要的数据。。大可依赖汇编器，它会明智地使用最合适的形式来实现该伪指令。

译者添加：如果某指令需要使用32位立即数，可以在该指令地址的附近定义一个32位整数数组，把这个立即数放到该数组中。然后使用一条LDR Rd, [PC, #offset] 来查表。offset的值需要计算，它其实是LDR指令的地址与该数组元素地址的距离。手工计算offset是很自虐的作法，而刚才讲到的LDR伪指令则能让汇编器来自动产生这种数组，并且负责计算offset。这种数组被广泛使用，它的学名叫“文字池”（literal pool），通常由汇编器自动布设，汇编程序很大时可能也需要手工布设(通过LTORG指示字)。

#### LDR伪指令 vs. ADR伪指令

Both LDR和ADR都有能力产生一个地址，语法和行为都有相似处，但却不尽相同。对于LDR，如果汇编器发现要产生立即数是一个程序地址，它会自动地把LSB置位，例如：

LDR r0, =address1 ; R0= 0x4000 | 1

…

address1

0x4000: MOV R0, R1

在这个例子中，汇编器会认出address1是一个程序地址，所以自动置位LSB。另一方面，如果汇编器发现要加载的是数据地址，则不会自作聪明，多机灵啊！看：

LDR R0, =address1 ; 会把0x4000原封不动地加载到R0

…

address1

0x4000: DCD 0x0 ;0x4000处记录的是一个数据

ADR指令则永远是“憨厚”的，它决不会擅自修改LSB。例如：

ADR r0, address1 ; R0= 0x4000。注意：**没有“=”号**

…

address1

0x4000: MOV R0, R1

ADR将如实地加载0x4000。注意，语法略有不同，没有“=”号。

前面已经提到，LDR通常是把要加载的数值预先定义，再使用一条PC相对加载指令来取出。而ADR则尝试对PC作算术加法或减法来取得立即数。因此ADR未必总能求出需要的立即数。其实顾名思义，ADR是为了取出附近某条指令或者变量的地址，而LDR则是取出一个通用的32位整数。因为ADR更专一，所以得到了优化——它产生的代码效率常常比LDR的要高。

### 4.3.2 汇编语言：数据处理

数据处理乃是处理器的看家本领，CM3当然要出类拔萃，它提供了丰富多彩的相关指令，每种指令的用法也是花样百出。限于篇幅，这里只列出常用的使用方式。就以加法为例，常见的有：

ADD R0, R1 ; R0 += R1

ADD R0, #0x12 ; R0 += 12

ADD.W R0, R1, R2 ; R0 = R1+R2

注意：虽然助记符都是“ADD”，但是二进制机器码是不同的。

当使用16位加法时，会自动更新APSR中的标志位。然而，在使用了“.W”显式指定了32位指令后，就可以通过“S”后缀手工控制对APSR的更新，如：

ADD.W R0, R1, R2 ; 不更新标志位

ADDS.W R0, R1, R2 ; 更新标志位

除了ADD指令之外，CM3中还包含SUB, MUL, UDIV/SDIV等用于算术四则运算，如表4.18所列

#### 表4.18 常见的算术四则运算指令

|  |  |
| --- | --- |
| 示例 | 功能描述 |
| ADD Rd, Rn, Rm ; Rd = Rn+Rm  ADD Rd, Rm ; Rd += Rm  ADD Rd, #imm ; Rd += imm | 常规加法  imm的范围是im8（16位指令）或im12（32位指令） |
| ADC Rd, Rn, Rm ; Rd = Rn+Rm+C  ADC Rd, Rm ; Rd += Rm+C  ADC Rd, #imm ; Rd += imm+C | 带进位的加法  imm的范围是im8（16位指令）或im12（32位指令） |
| ADDW Rd, #imm12 ; Rd += imm12 | 带12位立即数的常规加法 |
| SUB Rd, Rn ; Rd -= Rn  SUB Rd, Rn, #imm3 ; Rd = Rn-imm3  SUB Rd, #imm8 ; Rd -= imm8  SUB Rd, Rn, Rm ; Rd = Rn-Rm | 常规减法 |
| SBC Rd, Rm ; Rd -= Rm+C  SBC.W Rd, Rn, #imm12 ; Rd = Rn-imm12-C  SBC.W Rd, Rn, Rm ; Rd = Rn-Rm-C | 带借位的减法 |
| RSB.W Rd, Rn, #imm12 ; Rd = imm12-Rn  RSB.W Rd, Rn, Rm ; Rd = Rm-Rn | 反向减法 |
| MUL Rd, Rm ; Rd \*= Rm  MUL.W Rd, Rn, Rm ; Rd = Rn\*Rm | 常规乘法 |
| MLA Rd, Rm, Rn, Ra ; Rd = Ra+Rm\*Rn  MLS Rd, Rm, Rn, Ra ; Rd = Ra-Rm\*Rn | **乘加与乘减**  **（译者添加）** |
| UDIV Rd, Rn, Rm ; Rd = Rn/Rm （无符号除法）  SDIV Rd, Rn, Rm ; Rd = Rn/Rm （带符号除法） | 硬件支持的除法 |

CM3还片载了硬件乘法器，支持乘加/乘减指令，并且能产生64位的积，如表4.19所示

#### 表4.19 64位乘法指令

|  |  |
| --- | --- |
| 示例 | 功能描述 |
| SMULL RL, RH, Rm, Rn ;[RH:RL]= Rm\*Rn  SMLAL RL, RH, Rm, Rn ;[RH:RL]+= Rm\*Rn | 带符号的64位乘法 |
| UMULL RL, RH, Rm, Rn ;[RH:RL]= Rm\*Rn  UMLAL RL, RH, Rm, Rn ;[RH:RL]+= Rm\*Rn | 无符号的64位乘法 |

逻辑运算以及移位运算也是基本的数据操作。表4.20列出CM3在这方面的常用指令

#### 表4.20 常用逻辑操作指令

|  |  |
| --- | --- |
| 示例 | 功能描述 |
| AND Rd, Rn ; Rd &= Rn  AND.W Rd, Rn, #imm12 ; Rd = Rn & imm12  AND.W Rd, Rm, Rn ; Rd = Rm & Rn | 按位与 |
| ORR Rd, Rn ; Rd |= Rn  ORR.W Rd, Rn, #imm12 ; Rd = Rn | imm12  ORR.W Rd, Rm, Rn ; Rd = Rm | Rn | 按位或 |
| BIC Rd, Rn ; Rd &= ~Rn  BIC.W Rd, Rn, #imm12 ; Rd = Rn & ~imm12  BIC.W Rd, Rm, Rn ; Rd = Rm & ~Rn | 位段清零 |
| ORN.W Rd, Rn, #imm12 ; Rd = Rn | ~imm12  ORN.W Rd, Rm, Rn ; Rd = Rm | ~Rn | 按位或反码 |
| EOR Rd, Rn ; Rd ^= Rn  EOR.W Rd, Rn, #imm12 ; Rd = Rn ^ imm12  EOR.W Rd, Rm, Rn ; Rd = Rm ^ Rn | （按位）异或，异或总是按位的 |

译者添加

大多数涉及3个寄存器的32位数据操作指令，都可以在计算之前，对其第3个操作数Rn进行“预加工”——移位，格式为：

DataOp Rd, Rm, Rn, LSL #imm5 ;先对Rn逻辑左移imm5格

DataOp Rd, Rm, Rn, LSR #imm5 ;先对Rn逻辑右移imm5格

DataOp Rd, Rm, Rn, ASR #imm5 ;先对Rn算术右移imm5格

DataOp Rd, Rm, Rn, ROR #imm5 ;先对Rn圆圈右移imm5格

DataOp Rd, Rm, Rn, RO**L** #imm5 ;（错误）先对Rn循环左移imm5格

DataOp Rd, Rm, Rn, RRX ;先对Rn带进位位右移一格

注意：“预加工”是对Rn的一个“内部复本”执行操作，不会因此而影响Rn的值。但如果Rn正巧也是Rd,则按DataOp的计算方式来更新Rn。

其中，DataOp可以是所有“传统”的32位数据操作指令，包括：

**ADD/ADC; SUB/SBC/RSB; AND/ORR/EOR; BIC/ORN**

CM3还支持为数众多的移位运算。移位运算既可以与其它指令组合使用（传送指令和数据操作指令中的一些，参见文本框中的说明），也可以独立使用，如表4.21所示。

#### 表4.21 移位和循环指令

|  |  |
| --- | --- |
| 示例 | 功能描述 |
| LSL Rd, Rn, #imm5 ; Rd = Rn<<imm5  LSL Rd, Rn ; Rd <<= Rn  LSL.W Rd, Rm, Rn ; Rd = Rm<<Rn | 逻辑左移 |
| LSR Rd, Rn, #imm5 ; Rd = Rn>>imm5  LSR Rd, Rn ; Rd >>= Rn  LSR.W Rd, Rm, Rn ; Rd = Rm>>Rn | 逻辑右移 |
| ASR Rd, Rn, #imm5 ; Rd = Rn  imm5  ASR Rd, Rn ; Rd  = Rn  ASR.W Rd, Rm, Rn ; Rd = RmRn | 算术右移 |
| ROR Rd, Rn ; Rd >> = Rn  ROR.W Rd, Rm, Rn ; Rd = Rm >> Rn | 圆圈右移 |
| RRX.W Rd, Rn ; Rd = (Rn>>1)+(C<<31)  译者添加  (因为在RRX上使用S后缀比较特殊，故提出来单独讲解)  RRXS.W Rd, Rn ; tmpBit = Rn & 1  ; Rd = (Rn>>1)+(C<<31)  ; C= tmpBit | 带进位的右移**一格**  亦可写作 RRX{S} Rd 。此时，Rd也要担当Rn的角色——译注 |

如果在移位和循环指令上加上“S”后缀，这些指令会更新进位位C。如果是16位Thumb指令，则总是更新C的。图4.1给出了一个直观的印象

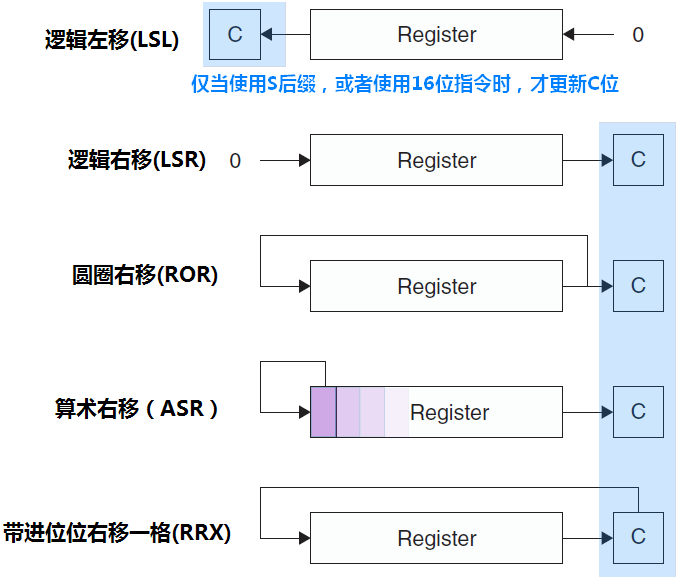


图4.1 移位与循环指令

为什么没有圆圈左移？

在圆圈移位中， 寄存器的32个位其实是手拉手组成一个圈的。那么这个圈向右转动n格，与向左转动32-n格是等效的。这种简单的道理，玩过“丢手绢”的小朋友们都知道。因此欲圆圈左移n格时，只要使用圆圈右移指令，并且转动32-n格即可。

介绍完了移位指令，接下来讲带符号扩展指令。

我们知道，在2进制补码表示法中，最高位是符号位，且所有负数的符号位都是1。负数还有另一个性质，就是不管在符号位的前面再添加多少个1，值都不变，只不过表达带符号整数的位数增多了。于是，在把一个8位或16位负数扩展成32位时，欲使其数值不变，就必须把所有新增的高位全填1。至于正数或无符号数，则只需简单地把新增的高位清0。因此，必须给带符号数开小灶，于是就有了整数扩展指令，如表4.22所示。

#### 表4.22 带符号扩展指令

|  |  |
| --- | --- |
| 示例 | 功能描述 |
| SXTB Rd, Rm ; Rd = Rm的带符号扩展 | 把带符号字节整数扩展到32位 |
| SXTH Rd, Rm ; Rd = Rm的带符号扩展 | 把带符号半字整数扩展到32位 |

我们知道，32位整数可以被认为是由4个字节拼接成的，也可以被认为是2个半字拼接成的。有时，需要把这些子元素倒腾倒腾，如表4.23所示

#### 表4.23 数据序翻转指令

|  |  |
| --- | --- |
| 示例 | 功能描述 |
| REV.W Rd, Rn | 在字中反转字节序 |
| REV16.W Rd, Rn | 在高低半字中反转字节序 |
| REVSH.W Rd, Rn | 在低半字中反转字节序，并做带符号扩展 |

这些指令乍一看不太好理解，但相信看过图4.2后就会豁然开朗了：

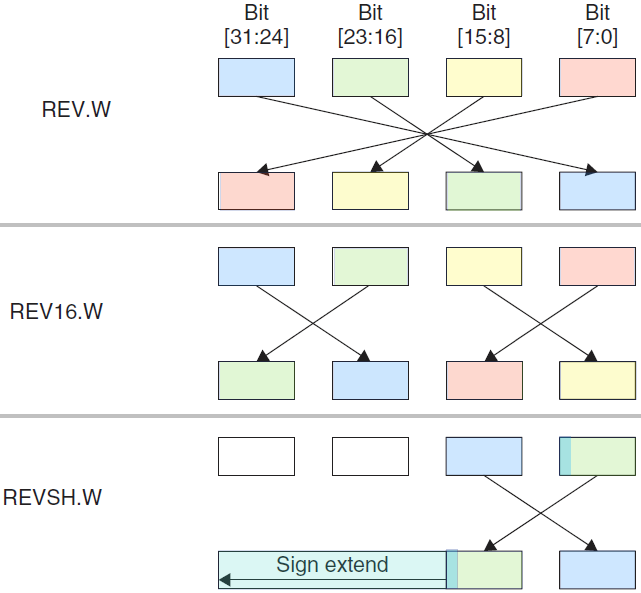


图4.2 反序操作

数据操作指令的最后一批，是位操作指令。位操作在单片机程序中，以及在系统软件中应中常常大显身手，而且在使用这类指令时，有很多新奇但是却妙用横生的技巧。这里在表4.24中先列出它们，本书在后续的小节中还要展开论述。

#### 表4.24 位段处理及把玩指令

|  |  |
| --- | --- |
| 指令 | 功能描述 |
| BFC.W Rd, Rn, #<width> |  |
| BFI.W Rd, Rn, #<lsb>, #<width> |  |
| CLZ.W Rd, Rn | 计算前导0的数目 |
| RBIT.W Rd, Rn | 位反转，按位旋转180度 |
| SBFX.W Rd, Rn, #<lsb>, #<width> | 拷贝位段，并带符号扩展到32位 |
| SBFX.W Rd, Rn, #<lsb>, #<width> | 拷贝位段，并无符号扩展到32位 |

### 4.3.3 汇编语言：子程呼叫与无条件跳转指令

最基本的无条件跳转指令有两条：

B Label ;跳转到Label处对应的地址

BX reg ;跳转到由寄存器reg给出的地址

在BX中，reg的最低位指示出在转移后将进入的状态：是ARM(LSB=0)呢，还是Thumb(LSB=1)。既然CM3只在Thumb中运行，就必须保证reg的LSB=1，否则一个fault打过来。

呼叫子程序时，需要保存返回地址，正点的指令是：

BL Label ;跳转到Label对应的地址，并且把跳转前的下条指令地址保存到LR

BLX reg ;跳转到由寄存器reg给出的地址，并根据REG的LSB切换处理器状态，

;还要把转移前的下条指令地址保存到LR

执行这些指令后，就把返回地址存储到LR（R14）中了，从而才能使用”BX LR”等形式返回。

使用BLX要小心，因为它还带有改变状态的功能。因此reg的LSB必须是1，以确保不会试图进入ARM状态。如果忘记置位LSB，则fault伺候。

对于艺高胆大的玩家来说，使用以PC为目的寄存器的MOV和LDR指令也可以实现转移，并且往往能借此实现很多“老实”的程序达不到的功效，常见形式有：

MOV PC, Rn ;转移地址由Rn给出

LDR PC, [Rn] ;转移地址存储在Rn所指向的存储器中

POP {…,PC} ;把返回地址以弹出堆栈的风格送给PC，

;从而实现跳转，这也是比较平易近人的技巧

LDMIA SP!, {…, PC} ;POP的另一种等效写法

同理，使用这些技巧，你也必须保证送给PC的值必须是奇数（LSB＝1）。

注意：有心的读者可能已经发现，ARM的BL虽然省去了耗时的访内操作，却只能支持一级子程序调用。如果子程序再呼叫 “孙程序”，则返回地址会被覆盖。因此当函数嵌套多于一级时，必须在调用“孙程序”之前先把LR压入堆栈——也就是所谓的“溅出”。

### 4.3.4 汇编语言：标志位与条件转移

在应用程序状态寄存器中有5个标志位，但只有4个被条件转移指令参考。绝大多数ARM的条件转移指令根据它们来决定是否转移，如表4.25所示

#### 表 4.25 Cortex-M3 APSR中可以影响条件转移的4个标志位

|  |  |  |
| --- | --- | --- |
| 标志位 | PSR位序号 | 功能描述 |
| N | 31 | 负数（上一次操作的结果是个负数）。N=操作结果的MSB |
| Z | 30 | 零（上次操作的结果是0）。当数据操作指令的结果为0,或者比较/测试的结果为0时，Z置位。 |
| C | 29 | 进位／借位（上次操作导致了进位或者借位）。C用于无符号数据处理，最常见的就是当加法进位及减法借位时C被置位。此外，C还充当移位指令的中介（详见v7M参考手册的指令介绍节）。 |
| V | 28 | 溢出（上次操作结果导致了数据的溢出）。该标志用于带符号的数据处理。比如，在两个正数上执行ADD运算后，和的MSB为1（视作负数），则V置位。 |

在ARM中，数据操作指令可以更新这4个标志位。这些标志位除了可以当作条件转移的判据之外，还能在一些场合下作为指令是否执行的依据（详见If-Then指令块），或者在移位操作中充当各种中介角色（仅进位位C）。

担任条件跳转及条件执行的判据时，这4个标志位既可单独使用，又可组合使用，以产生共15种跳转判据，如下表4.26所示

#### 表4.26 跳转及条件执行判据

|  |  |  |
| --- | --- | --- |
| 符号 | 条件 | 关系到的标志位 |
| EQ | 相等(EQual) | Z==1 |
| NE | 不等（NotEqual） | Z==0 |
| CS/HS | 进位(CarrySet)  无符号数高于或相同 | C==1 |
| CC/LO | 未进位(CarryClear)  无符号数低于 | C==0 |
| MI | 负数(MInus) | N==1 |
| PL | 非负数 | N==0 |
| VS | 溢出 | V==1 |
| VC | 未溢出 | V==0 |
| HI | 无符号数大于 | C==1 && Z==0 |
| LS | 无符号数小于等于 | C==0 || Z==1 |
| GE | 带符号数大于等于 | N==V |
| LT | 带符号数小于 | N!=V |
| GT | 带符号数大于 | Z==0 && N==V |
| LE | 带符号数小于等于 | Z==1 || N!=V |
| AL | 总是 | - |

表中共有15个条件组合（AL相当于无条件——译注），通过把它们点缀在无条件转移指令（B）的后面，即可做成各式各样的条件转移指令，例如：

BEQ label ;当Z=1时转移

亦可以在指令后面加上“.W”，来强制使用Thumb-2的32位指令来做更远的转移（其实没必要，汇编器会自行判断——译注），例如：

BEQ.W label

这些条件组合还可以用在If-Then语句块中，比如：

CMP R0, R1 ;比较R0,R1

ITTET **GT** ;If R0>R1 Then（T代表Then，E代表Else）

MOV**GT** R2, R0

MOV**GT** R3, R1

MOV**LE** R2, R0

MOV**GT** R3, R1

（本章的后面有对IT指令和If-Then块进行详细说明。请留意上例中对字体和格式的严格使用——译注）

在CM3中，下列指令可以更新PSR中的标志：

* + 16位算术逻辑指令
  + 32位带S后缀的算术逻辑指令
  + 比较指令（如，CMP/CMN）和测试指令（如TST/TEQ）
  + 直接写PSR/APSR (MSR指令)

大多数16位算术逻辑指令不由分说就会更新标志位（不是所有的16位指令都这样，例如ADD.N Rd, Rn, Rm是16位指令，但不更新标志位——译注），32位的都可以让你使用S后缀来控制。例如：

ADDS.W R0, R1, R2 ;使用32位Thumb-2指令，并更新标志

ADD.W R0, R1, R2 ;使用32位Thumb-2指令，但不更新标志位

ADD R0, R1 ;使用16位Thumb指令，无条件更新标志位

ADDS R0, #0xcd ;使用16位Thumb指令，无条件更新标志位

译注：虽然真实指令的行为如上所述。但是在你用汇编语言写代码时，如果使用了UAL（统一汇编语言），汇编器会做调整，最终生成的指令不一定和与你在字面上写的指令相同。对于ARM汇编器而言，调整的结果是：**如果没有写后缀S，汇编器就一定会产生不更新标志位的指令**。可见，使用UAL的一大好处，就是我们完全能控制是否更新标志位，决不会出现标志位被意外更新的情况。

S后缀的使用要当心。16位Thumb指令可能会无条件更新标志位，但也可能不更新标志位。为了让你的代码能在不同汇编器下有相同的行为，当你需要更新标志，以作为条件指令的执行判据时，一定不要忘记加上S后缀。

CM3中还有比较和测试指令，它们存在的目的就是更新标志位，因此是会无条件影响标志位的，如下所述。

CMP指令。CMP指令在内部做两个数的减法，并根据差来设置标志位，但是不把差写回。CMP可有如下的形式：

CMP R0, R1 ; 计算R0-R1的差， 并且根据结果更新标志位

CMP R0, 0x12 ; 计算R0-0x12的差，并且根据结果更新标志位

CMN指令。CMN是CMP的一个孪生姊妹，只是它在内部做两个数的加法（相当于减去减数的相反数），如下所示：

CMN R0, R1 ; 计算R0+R1的和， 并根据结果更新标志位

CMN R0, 0x12 ; 计算R0+0x12的和，并根据结果更新标志位

TST指令。TST指令的内部其实就是AND指令，只是不写回运算结果，它也无条件更新标志位。它的用法与CMP的用法相同：

TST R0, R1 ; 计算R0 & R1， 并根据结果更新标志位

TST R0, 0x12 ; 计算R0 & 0x12， 并根据结果更新标志位

TEQ指令。TEQ指令的内部其实就是EOR指令，只是不写回运算结果，它也无条件更新标志位。它的用法与CMP的用法相同：

TEQ R0, R1 ; 计算R0 ^ R1， 并根据结果更新标志位

TEQ R0, 0x12 ; 计算R0 ^ 0x12， 并根据结果更新标志位

### 4.3.5 汇编语言：指令隔离(barrier)指令和存储器隔离指令

CM3中的另一股新鲜空气是一系列的隔离指令（亦可以译成“屏障”、“路障”，可互换使用——译者注）。它们在一些结构比较复杂的存储器系统中是需要的（典型地用于流水线和写缓冲——译者注）。在这类系统中，如果没有必要的隔离，会导致系统发生紊乱危象（race condition），（相当于数电中的“竞争与冒险”——译者注）.

举例来说，如果可以在运行时更改存储器的映射关系或者内存保护区的设置，（通过写MPU的寄存器），就必须在更改之后立即补上一条DSB指令（数据同步指令）。因为对MPU的写操作很可能会被放到一个写缓冲中。写缓冲是为了提高存储器的总体访问效率而设的，但它也有副作用，其中之一，就是会导致写内存的指令被延迟几个周期执行，因此对存储器的设置不能即刻生效，这会导致紧临着的下一条指令仍然使用旧的存储器设置——但程序员的本意显然是使用新的存储器设置。这种紊乱危象是后患无穷的，常会破坏未知地址的数据，有时也会产生非法地址访问fault。紊乱危象还有其它的表现形式，后续章节会一一介绍。CM3提供隔离指令族，就是要消灭这些紊乱危象（在有些讲解计算机体系体系结构的书中，这类紊乱危象也被称为“存储器相关”——译注）。

CM3中共有3条隔离指令，如表4.27所列

表4.27 隔离指令

|  |  |
| --- | --- |
| 指令名 | 功能描述 |
| DMB | 数据存储器隔离。DMB指令保证： 仅当所有在它前面的存储器访问操作都执行完毕后，才提交(commit)在它后面的存储器访问操作。 |
| DSB | 数据同步隔离。比DMB严格： 仅当所有在它前面的存储器访问操作都执行完毕后，才执行在它后面的指令（亦即任何指令都要等待存储器访问操作——译者注） |
| ISB | 指令同步隔离。最严格：它会清洗流水线，以保证所有它前面的指令都执行完毕之后，才执行它后面的指令。 |

DMB在双口RAM以及多核架构的操作中很有用。如果RAM的访问是带缓冲的，并且写完之后马上读，就必须让它“喘口气”——用DMB指令来隔离，以保证缓冲中的数据已经落实到RAM中。DSB比DMB更保险（当然也是有执行代价的），它是宁可错杀也不漏网——清空了写缓冲，使得任何它后面的指令，不管要不要使用先前的存储器访问结果，通通等待访问完成。大虾们可以在有绝对信心时使用DMB，新手还是使用DSB比较保险。

同DMB/DSB相比，ISB指令看起来似乎最强悍，但是却一身都是 “愣劲”，不由分说就“动粗”。不过它还有其它的用场——对于高级底层技巧：“自我更新”(self-mofifying)代码，非常有用。举例来说，如果某个程序从下一条要执行的指令处更新了自己，但是先前的旧指令已经被预取到流水线中去了，此时就必须清洗流水线，把旧版本的指令洗出去，再预取新版本的指令。因此，必须在被更新代码段的前面使用ISB，以保证旧的代码从流水线中被清洗出去，不再有机会执行（译者觉得这种做法太工于技巧，有点“作秀”，现实编程中应该极少会用到，因此读者不必太钻它）。

### 4.3.6 汇编语言：饱和运算

饱和运算可能是读者在以前不太听说的。不过其实很简单。如果读者学过模电，或者知道放大电路中所谓的“饱和削顶失真”，理解饱和运算就更加容易。而且饱和运算指令确实是打算用于信号处理程序的。

CM3中的饱和运算指令分为两种：一种是“没有直流分量”的交流信号饱和——带符号饱和运算；另一种无符号饱和运算则类似于“削顶失真＋单向导通”。

饱和运算多用于信号处理。比如，信号放大。当信号被放大后，有可能使它的幅值超出允许输出的范围。如果傻乎乎地只是清除MSB，则常常会严重破坏信号的波形，而饱和运算则只是使信号产生削顶失真。如图4.3所示。

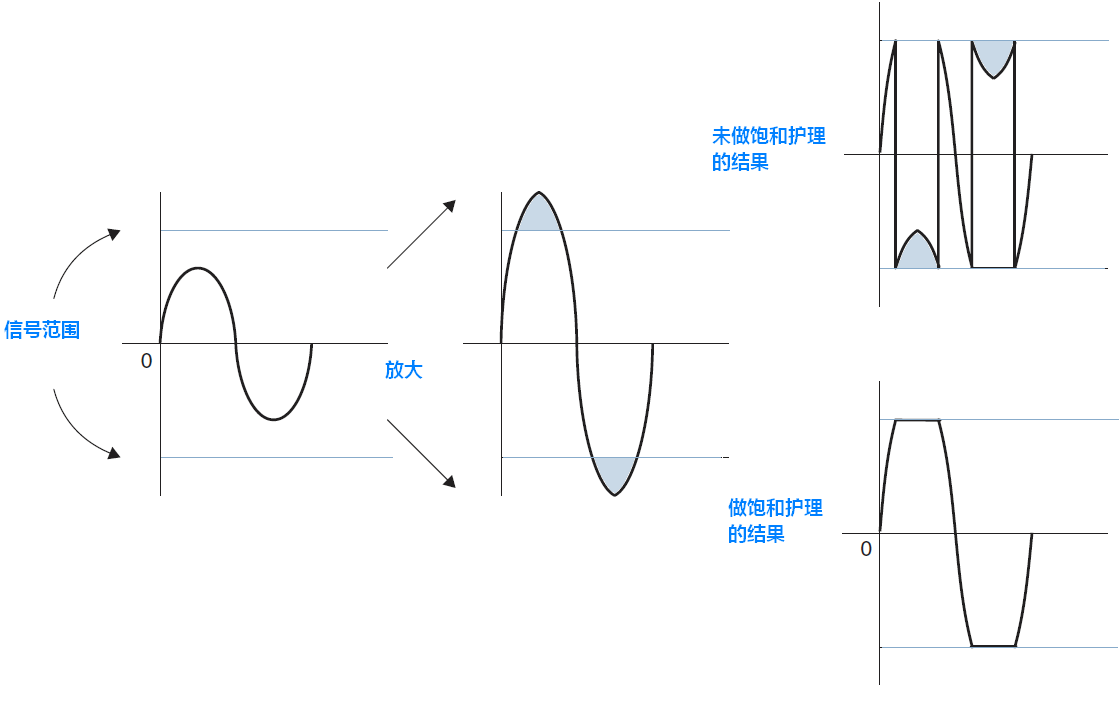


图4.3 带符号饱和运算

可见，饱和运算的“护理”虽然不能消灭失真，但那种委琐的变形是可以消灭的。表4.28列出饱和运算指令。

表4.28 饱和运算指令

|  |  |
| --- | --- |
| 指令名 | 功能描述 |
| SSAT.W Rd, #imm5, Rn, {,shift} | 以带符号数的边界进行饱和运算（交流） |
| SSAT.W Rd, #imm5, Rn, {,shift} | 以无符号数的边界进行饱和运算（带纹波的直流） |

饱和运算的结果可以拿去更新Q标志（在APSR中）。Q标志在写入后可以用软件清0——通过写APSR——还记得APSR是“红杏出墙”的吗？

Rn存储“放大后待做饱和运算的信号”，（Rn总是32位带符号整数——译者注）。同很多其它数据操作指令类似，Rn也可以使用移位来“预加工”。

Rd存储饱和运算的结果。

#imm5用于指定饱和边界——该由多少位的带符号整数来表达允许的范围（奇数也可以使用），取值范围是1－32。举例来说，如果要把一个32位（带符号）整数饱和到12位带符号整数（-2048至2047），则可以如下使用SSAT指令

SSAT{.W} R1, #12, R0

这条指令对于R0不同值的执行结果如表4.29所示

表4.29 带符号饱和运算的示例运算结果

|  |  |  |
| --- | --- | --- |
| 输入(R0) | 输出(R1) | Q标志位 |
| 0x2000(8192) | 0x7FF(2047) | 1 |
| 0x537(1335) | 0x537(1335) | 无变化 |
| 0x7FF(2047) | 0x7FF(2047) | 无变化 |
| 0 | 0 | 无变化 |
| 0xFFFFE000(-8192) | 0xFFFFF800(-2048) | 1 |
| 0xFFFFFB32(-1230) | 0xFFFFFB32(-1230) | 无变化 |

如果需要把32位整数饱和到无符号的12位整数（0-4095），则可以如下使用USAT指令

USAT{.W} R1, #12, R0

该指令的执行情况如图4.4演示

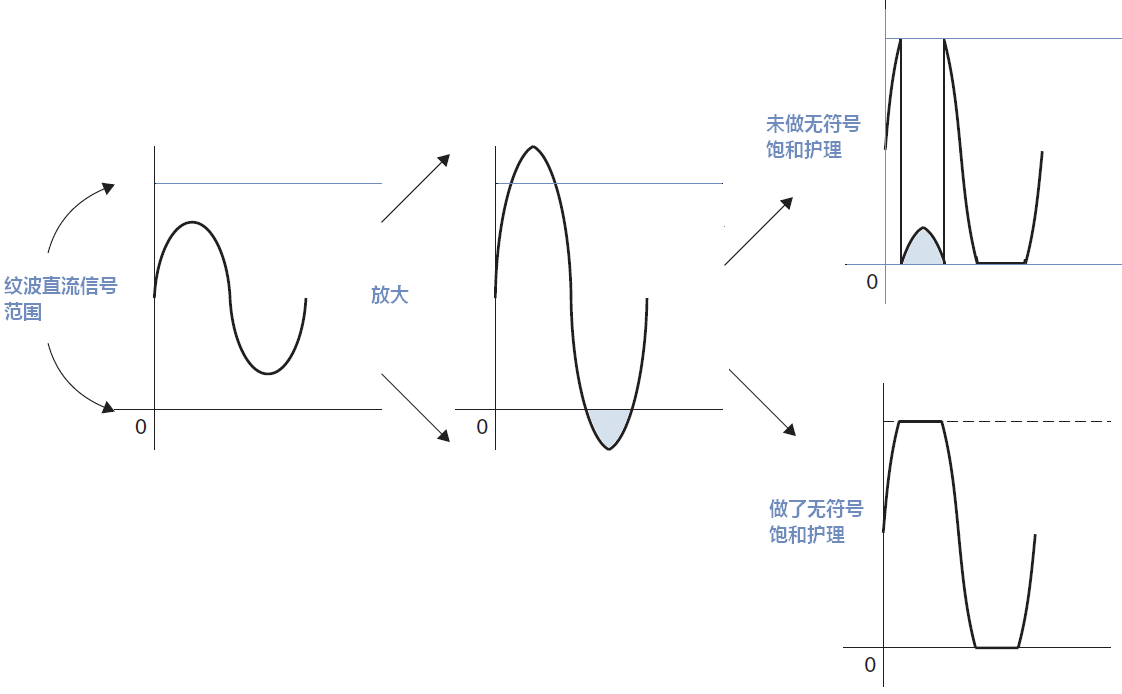


图4.4 无符号饱和运算

表4.30 无符号饱和运算的示例运算结

|  |  |  |
| --- | --- | --- |
| 输入(R0) | 输出(R1) | Q标志位 |
| 0x2000(8192) | 0xFFF(4095) | 1 |
| 0xFFF(4095) | 0xFFF(4095) | 无变化 |
| 0x1000(4096) | 0xFFF(4095) | 1 |
| 0x800(2048) | 0x800(2048) | 无变化 |
| 0 | 0 | 无变化 |
| 0x80000000(-2G) | 0 | 1 |
| 0xFFFFFB32(-1230) | 0 | 1 |

## 4.4 CM3中一些前卫的指令

这里列出几条从v6和v7开始才支持的最新指令。

### 4.4.1 MRS和MSR

虽然名字与以前的ARM相同，但功能变了。这两条指令是访问特殊功能寄存器的“专用通道”——当然必须在特权级下使用，除了APSR可以在用户级下访问外。

指令语法如下：

**MRS <Rn>, <SReg>** ;加载特殊功能寄存器的值到Rn

**MSR <Sreg>,<Rn>** ;存储Rn的值到特殊功能寄存器

SReg可以是下表中的一个：

表4.31 MRS/MSR可以使用的特殊功能寄存器

|  |  |
| --- | --- |
| 符号 | 功能 |
| IPSR | 当前服务中断号寄存器 |
| EPSR | 执行状态寄存器（读回来的总是0）。它里面含T位，在CM3中T位必须是1,所以在更改EPSR时要格外小心——译注。 |
| APSR | 上条指令结果的标志 |
| IEPSR | IPSR+EPSR |
| IAPSR | IPSR+APSR |
| EAPSR | EPSR+APSR |
| PSR | xPSR = APSR+EPSR+IPSR |
| MSP | 主堆栈指针 |
| PSP | 进入堆栈指针 |
| PRIMASK | 常规异常屏蔽寄存器 |
| BASEPRI | 常规异常的优先级阈值寄存器 |
| BASEPRI\_MAX | 等同BASEPRI，但是施加了写的限制：新的优先级比较比旧的高（更小的数） |
| FAULTMASK | fault屏蔽寄存器（除了包含PRIMASK的全部功能外，还能除能硬fault） |
| CONTROL | 控制寄存器（堆栈选择，特权等级设置） |

下面给出一个指定PSP进行更新的例子：

LDR R0, =0x20008000

MSR PSP, R0

BX LR ;如果是从异常返回到线程状态，则使用新的PSP的值作为栈顶指针

### 4.4.2 IF-THEN

IF-THEN(IT)指令围起一个块，里面最多有4条指令，它里面的指令可以条件执行。

IT指令已经带了一个“T”，因此还可以最多再带3个“T”或者“E”。并且对T和E的顺序没有要求。其中T对应条件成立时执行的语句，E对应条件不成立时执行的语句。在If-Then块中的指令必须加上条件后缀，且T对应的指令必须使用和IT指令中相同的条件，E对应的指令必须使用和IT指令中相反的条件。

IT的使用形式总结如下：

**IT <cond>**  ;围起1条指令的IF-THEN块

**IT<x> <cond>**  ;围起2条指令的IF-THEN块

**IT<x><y> <cond>**  ;围起3条指令的IF-THEN块

**IT<x><y><z> <cond>** ;围起4条指令的IF-THEN块

其中<x>, <y>, <z>的取值可以是“T”或者“E”。而<cond>则是在表4.26中列出的条件（AL除外）。

［译注17］：IT指令使能了指令的条件执行方式，并且使CM3不再预取不满足条件的指令。又因为它在使用时取代了条件转移指令，还避免了在执行流转移时，对流水线的清洗和重新指令预取的开销，所以能优化C结构中的微小if块和很多“?:”运算符

IT指令优化C代码的例子如下面伪代码所示：

if (R0==R1)

{

R3 = R4 + R5;

R3 = R3 / 2;

}

else

{

R3 = R6 + R7;

R3 = R3 / 2;

}

可以写作：

CMP R0, R1 ; 比较R0和R1

ITTEE **EQ** ; 如果 R0 == R1, Then-Then-Else-Else

ADD**EQ** R3, R4, R5 ; 相等时加法

ASR**EQ** R3, R3, #1 ; 相等时算术右移

ADD**NE** R3, R6, R7 ; 不等时加法

ASR**NE** R3, R3, #1 ; 不等时算术右移

### 4.4.3 CBZ和CBNZ

比较并条件跳转指令专为循环结构的优化而设，它只能做前向跳转。语法格式为：

**CBZ <Rn>, <label>**

**CBNZ <Rn>, <label>**

它们的跳转范围较窄，只有0-126。

典型范围如下所示：

while (R0!=0)

{

Function1();

}

变成

Loop

CBZ R0, LoopExit

BL Function1

B Loop

LoopExit:

与其它的比较指令不同，CBZ/CBNZ不会更新标志位。

### 4.4.4 SDIV和UDIV

突破性的32位硬件除法指令，如下所示：

**SDIV.W Rd, Rn, Rm**

**UDIV.W Rd, Rn, Rm**

运算结果是 Rd= Rn/Rm，余数被丢弃。例如：

LDR r0, =300

MOV R1, #7

UDIV.W R2, R0, R1

则R2= 300/7 =44

为了捕捉被零除的非法操作，你可以在NVIC的配置控制寄存器中置位DIVBYZERO位。这样，如果出现了被零除的情况，将会引发一个用法fault异常。如果没有任何措施，Rd将在除数为零时被清零。

### 4.4.5 REV, REVH,REV16以及REVSH

REV反转32位整数中的字节序，REVH则以半字为单位反转，且只反转低半字。语法格式为：

**REV Rd, Rm**

**REVH Rd, Rm**

**REV16 Rd, Rm**

**REVSH Rd, Rm**

例如，记R0=0x12345678，在执行下列两条指定后：

REV R1, R0

REVH R2, R0

REV16 R3, R0

则R1=0x78563412，R2=0x12347856，R3=0x34127856。这些指令专门服务于小端模式和大端模式的转换，最常用于网络应用程序中（网络字节序是大端，主机字节序常是小端）。

REVSH在REVH的基础上，还把转换后的半字做带符号扩展。例如，记R0=0x33448899,则

REVSH R1, R0

执行后，R1=0x**FFFF**9988

### 4.4.6 RBIT

RBIT比前面的REV之流更精细，它是按位反转的，相当于把32位整数的二进制表示法水平旋转180度。其格式为：

**RBIT.W Rd, Rn**

这个指令在处理串行比特流时大有用场，而且几乎到了没它不行的地步（不信可以去写段程序完成它的功能，看看颠来倒去的能不能把你“转晕”）。

例如，记R1=0xB4E10C23（二进制数值为 1011,0100,1110,0001,0000,1100,0010,0011），

RBIT.W R0, R1

执行后，则R0=0xC430872D（二进制数值为1100,0100,0011,0000,1000,0111,0010,1101）

这条指令单独使用时看不出什么作用，但是与其它指令组合使用时往往有特效，高级技巧常用到它。

### 4.4.7 SXTB, SXTH, UXTB, UXTH

这4个指令是为了体贴C语言的强制数据类型转换而设的，把数据宽度转换成处理器喜欢的32位长度（处理器字长是多少，就喜欢多长的整数，其操作效率和存储效率都最高）。它们的语法如下：

**SXTB Rd, Rn**

**SXTH Rd, Rn**

**UXTB Rd, Rn**

**UXTH Rd, Rn**

对于SXTB/SXTH，数据带符号位扩展成32位整数。对于UXTB/UXTH，高位清零。例如，记R0=0x55aa8765,则

SXTB R1, R0 ; R1=0x00000065

SXTH R1, R0 ; R1=0x**ffff**8765

UXTB R1, R0 ; R1=0x00000065

UXTH R1, R0 ; R1=0x00008765

### 4.4.8 BFC/BFI，UBFX/SBFX

这些是CM3提供的位段操作指令，这里所讲的位段与C语言中的位段是一致的，它们与系统程序和单片机程序非常对口。

BFC（位段清零）指令把32位整数中任意一段连续的2进制位s清0，语法格式为：

**BFC.W Rd, #lsb, #width**

其中，lsb为位段的末尾，width则指定在lsb和它的左边（更高有效位），共有多少个位参与操作。

例如，

LDR R0, =0x1234FFFF

BFC R0, #4, #10

执行完后，R0= 0x1234C00F

译注：位段不支持首尾拼接。例如， BFC R0, #27, #9将产生不可预料的结果

BFI（位段插入指令），把某个寄存器按LSB对齐的数值，拷贝到另一个寄存器的某个位段中，其格式为

**BFI.W Rd, Rn, #lsb, #width**

例如，

LDR R0, =0x12345678

LDR R1, =0xAABBCCDD

BFI.W R1, R0, #8, #16

则执行后，R1= 0xAA5678DD （总是从Rn的最低位提取，#lsb只对Rd起作用——译注）

UBFX/SBFX都是位段提取指令，语法格式为：

**UBFX.W Rd, Rn, #lsb, #width**

**SBFX.W Rd, Rn, #lsb, #width**

UBFX从Rn中取出任一个位段，执行零扩展后放到Rd中（请比较与BFI的不同）。例如：

LDR R0, =0x5678ABCD

UBFX.W R1, R0, #12,#16

则R0=0x0000678A

类似地，SBFX也抽取任意的位段，但是以带符号的方式进行扩展。例如：

LDR R0, =0x5678ABCD

SBFX.W R1, R0, #8,#4

则R0=0xFFFFFFFB

上述例子为了描述方便使用了4比特对齐的#lsb和#width，但事实上并无此限制——译注

### 4.4.9 LDRD/STRD

CM3在一定程度上支持64位整数。其中LDRD/STRD就是为64位整数的数据传送而设的，语法格式为：

**LDRD.W RL, RH, [Rn, #+/-offset] {!};**可选预索引的64位整数加载

**LDRD.W RL, RH, [Rn],#+/-offset ;**后索引的64位整数加载

**STRD.W RL, RH, [Rn, #+/-offset] {!};**可选预索引的64位整数存储

**STRD.W RL, RH, [Rn],#+/-offset ;**后索引的64位整数存储

例如，记 (0x1000)= 0x1234\_5678\_ABCD\_EF00：则

*LDR* R2, =0x1000 ;

LDRD.W R0, R1, [R2]

执行后， R0= 0xABCD\_EF00, R1=0x1234\_5678

同理，我们也可以使用STRD来存储64位整数。在上面的例子执行完毕后，若执行如下代码：

STRD.W R1, R0, [R2]

执行后， (0x1000)=0xABCD\_EF00\_1234\_5678，从而实现了双字的字序反转操作。

### 4.4.10 TBB,TBH

高级语言都提供了“分类讨论”式控制结构，如C语言的switch，Basic语言的Select Case。通常，给我们的印象是比较靠后的case执行起来效率比较低，因为要一个一个地查。有了TBB/TBH后，则改善了这类结构的执行效率（可以对比51中的MOVC）

TBB（查表跳转字节范围的偏移量）指令和TBH（查表跳转半字范围的偏移量）指令，分别用于从一个字节数组表中查找转移地址，和从半字数组表中查找转移地址。TBH的转移范围已经足以应付任何臭长的switch结构。如果写出的switch连TBH都搞不定，只能说那人有严重自虐倾向。

因为CM3的指令至少是按半字对齐的，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。又因为PC的值为当前地址+4，故TBB的跳转范围可达255\*2+4=514；TBH的跳转范围更可高达65535\*2+4=128KB+2。请注意：Both TBB和TBH都只能作前向跳转，也就是说偏移量是一个无符号整数。

TBB的语法格式为：

TBB.W [Rn, Rm] ; PC+= Rn[Rm]\*2

在这里，Rn指向跳转表的基址，Rm则给出表中元素的下标。图4.5指示了这个操作

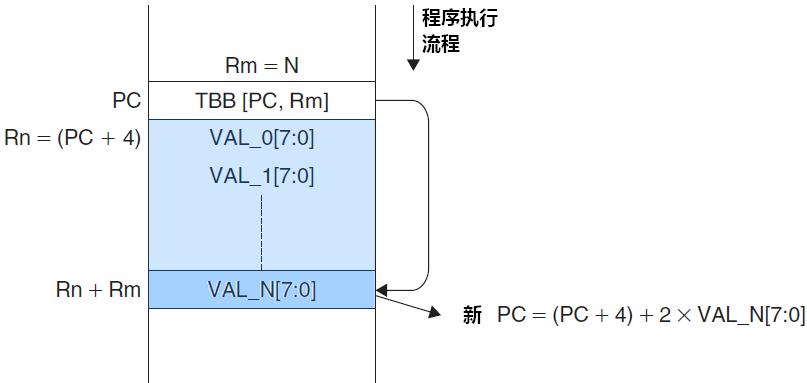


图4.5 TBB功能演示

如果Rn是R15，则由于指令流水线的影响，Rn的值将是PC+4。通常很少有人会手工计算表中偏移量，因为很繁，而且程序修改后要重新计算，尤其是当跨源文件查表时（由连接器负责分配地址）。所以这种指令在汇编中很少用到，通常是C编译器专用的，它可以在每次编译时重建该表。不过，可以为各入口地址取个标号，而且此指令还有其它的使用方式。在系统程序的开发中，此指令可以提高程序的运行效率。为了提供一个节能高效的操作系统或者基础函数库，必须挖空心思地使用各种奇异的技巧，甚至在极端情况下，还要严重违反常规程序设计的基本原则。

另外还要注意的是，不同的汇编器可能会要求不同的语法格式。在ARM汇编器（armasm.exe）中，TBB跳转表的创建方式如下所示：

TBB.W [pc, r0] ; 执行此指令时，PC的值正好等于branchtable

branchtable

DCB ((dest0 – branchtable)/2) ; 注意：因为数值是8位的，故使用DCB指示字

DCB ((dest1 – branchtable)/2)

DCB ((dest2 – branchtable)/2)

DCB ((dest3 – branchtable)/2)

dest0

... ; r0 = 0时执行

dest1

... ; r0 = 1时执行

dest2

... ; r0 = 2时执行

dest3

... ; r0 = 3时执行

TBH的操作原理与TBB相同，只不过跳转表中的每个元素都是16位的。故而下标为Rm的元素要从Rn+2\*Rm处去找。如图4.6所演示：

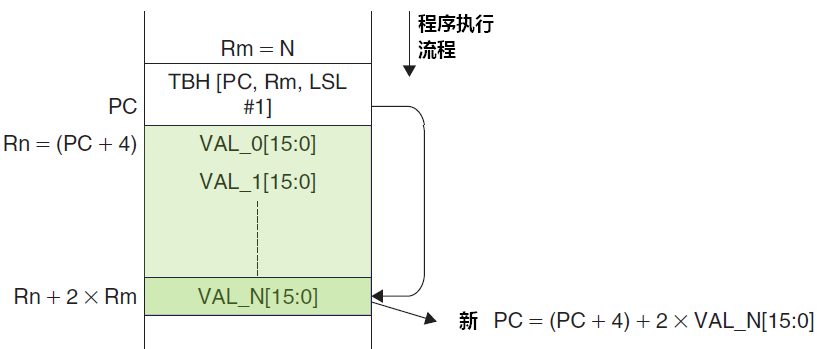


图4.6 TBH功能演示

TBH跳转表的创建方式与TBB的类似，如下所示：

TB**H**.W [pc, r0, **LSL #1**] ; 执行此指令时，PC的值正好等于branchtable

branchtable

DC**I** ((dest0 – branchtable)/2) ; 注意：数值是16位的，故使用DCI指示字

DC**I** ((dest1 – branchtable)/2)

DC**I** ((dest2 – branchtable)/2)

DC**I** ((dest3 – branchtable)/2)

dest0

... ; r0 = 0时执行

dest1

... ; r0 = 1时执行

dest2

... ; r0 = 2时执行

dest3

... ; r0 = 3时执行