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 中还有一个快速的查阅参考。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。

汇编语言基础

为了给以后的学习扫清障碍，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。

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，通常都给出本指令执行结果的存储地。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操作，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
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       ; 把立即数传送到指令中
    STR     R1, [R0]                ; *R0=R1，执行完此指令后 IRQ #0 被使能。
```

注意：常数定义必须顶格写

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
```

(DCI 也必须空格写——译注)

类似地，你还可以使用 DCB 来定义一串字节常数——允许以字符串的形式表达，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。

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 中的标志 s，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。还有若干个其它的条件。例如： BEQ <Label> ；仅当 EQ 满足时转移

在 Cortex-M3 中，对条件后缀的使用有限制，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。

汇编语言：统一的汇编语言

为了最有力地支持 Thumb-2，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以相同的语法格式书写，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   ; UAL 语法允许的等值写法（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，你可以让汇编器决定用哪个，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 位指令以缩小代码体积，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，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，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 指令则无任何限制。不过，把 R15(PC)作为目的寄存器却很容易走火入魔——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。对 PC 的使用额外的戒律。如果感兴趣，可以参考《ARMv7-M 架构应用级参考手册》。

指令集

Cortex-M3 支持的指令在表 4.2-表 4.9 列出。其中

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

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多格——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	加载多个字，并且在加载后自增基址寄存器
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。如果调试被使能，则进入调试状态（停机）。或者如果调试监视器异常被使能，则调用一个调试异常，否则调用一个 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——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底 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 位整数先用 2 进制表达，再旋转 180 度——译注）
REV	对一个 32 位整数做按字节反转
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	带进位的逻辑右移一格（最高位用 C 填充，且不影响 C 的值——译注）
SFBX	从一个 32 位整数中提取任意的位段，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多个字到若干寄存器
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
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	在本地的处理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）

未支持的指令

有若干条 Thumb 指令没有被 Cortex-M3 支持，下表列出了没有被支持的指令，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 v6 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

有少量在 ARMv7-M 中列出的指令不被 CM3 支持。其中 v7M 允许 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

改变处理器状态指令（CPS）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 v6 中定义的某些位在 CM3 中不存在。

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。

近距离地检视指令

从现在起，我们将介绍一些在 ARM 汇编代码中很通用的语法。有些指令可以带有多种参数，比如预移位操作，但本章并不会讲得面面俱到。

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

- 两个寄存器间传送数据
- 寄存器与存储器间传送数据
- 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反后再传送。

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 处读取一个字节到 Rd
LDRH Rd, [Rn, #offset]	从地址 Rn+offset 处读取一个半字到 Rd
LDR Rd, [Rn, #offset]	从地址 Rn+offset 处读取一个字到 Rd
LDRD Rd1, Rd2, [Rn, #offset]	从地址 Rn+offset 处读取一个双字(64 位整数)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
LDRD 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。每读一个字后 Rd 自增一次，16 位宽度
STMIA Rd!, {寄存器列表}	存储多个字到 Rd 处。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。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。每读一个字前 Rd 自减一次，32 位宽度
STMIA.W Rd!, {寄存器列表}	存储多个字到 Rd 处。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

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  PUSH  {R0-R3, PC}
```

Rd 后面的“!”是什么意思？它表示要自增($I_{ncrement}$)或自减（$D_{ecrement}$）基址寄存器 Rd 的值，时机是在每次访问前(B_{efore})或访问后(A_{fter})。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，每存一次曾一次，先存储后自增
STMDB.W R8,     {R0-R3} ;   R8 值的“一个内部复本”先自减后存储，但是 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。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]!**	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）

LDRD.W Rd1, Rd2, [Rn, #offset]!	
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]!	

CM3 除了支持“预索引”，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，后索引是忠实使用基址寄存器 Rd 的值作为数据传送的地址的。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——相当于有一个“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	

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，则汇编器会给你报错，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
```

看起来令人扫兴，不是吗？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 long 类型）。

PUSH/POP 作为堆栈专用操作，也属于数据传送指令类（具体关系参见译注 13——译者注）。

通常 PUSH/POP 对子的寄存器列表是一致的，但是 PC 与 LR 的使用方式有所通融，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，这些寄存器都是关键部位。因此除了 APSR 在某些场合下可以“露点”之外，其它的都不能“走光”——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，甚至当机。如果以身试法，则 fault 伺候（MemManage fault，如果被除能则“上访”成硬 fault）。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。

程序写多了你就会感觉到，程序中会经常使用立即数。最典型的就是：当你要访问某个地址时，你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，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？当前是要用两条指令来完成了。

如果某指令需要使用 32 位立即数，也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，马上要讲到的一条伪指令能让汇编器来

自动产生这种数组，并且负责计算 offset。这里提到的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，程序很大时可能也需要手工布设(LTORG 指示字)。

不过，为了书写的方便，汇编器通常都支持”LDR Rd, = imm32”伪指令。例如：

```
LDR,        r0,     =0x12345678
```

酷吧！它的名字也是 LDR，但它是伪指令，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，汇编器都在遇到 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。通过组合使用 MOVW 和 MOVT 也能产生 32 位立即数，不过有点麻烦。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。

LDR 伪指令 vs. ADR 伪指令

Both LDR 和 ADR 都有能力产生一个地址，但是语法和行为不同。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       ; R0= 0x4000
    …
address1
0x4000:         DCD     0x0         ;0x4000 处记录的是一个数据
```

ADR 指令则是“厚道人”，它决不会修改 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，故而它的代码效率常常比 LDR 的要高。

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，这里只列出最常用的使用方式。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
```

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 = Rm-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` `SMLAL 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,     ROL #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 正巧是 Rd,则按 DataOp 的计算方式来更新。

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 = Rm»»Rn`
	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

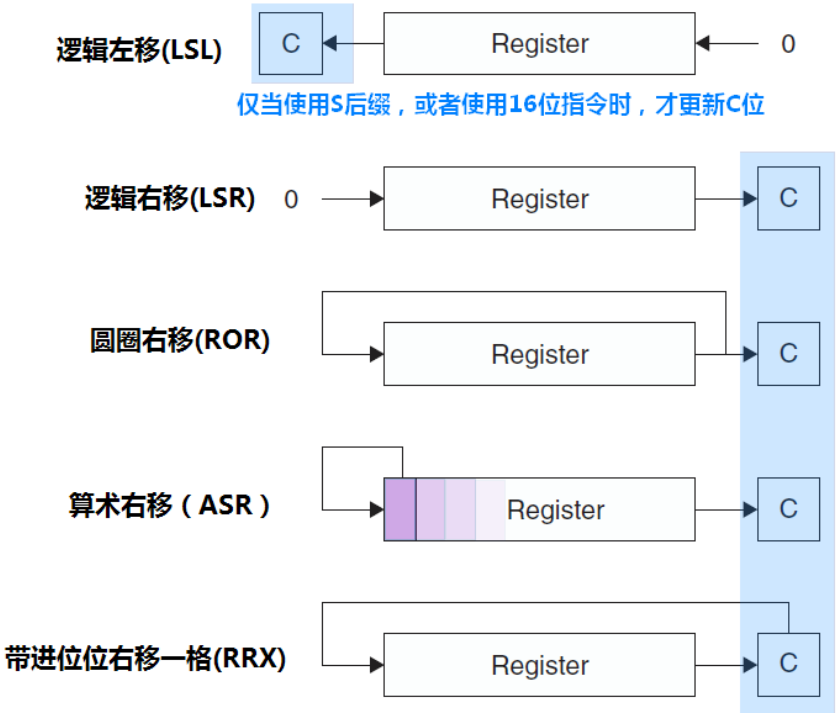


图 4.1 移位与循环指令

> **为啥没有圆圈左移？**
>
> 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。但是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。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，就必须把所有高位全填 1。至于正数或无符号数，则只需简单地把高位清 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，颠来倒去，如表 4.23 所示

表 4.23 数据序转指令

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	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，相信看过图 4.2 后就会豁然开朗了：

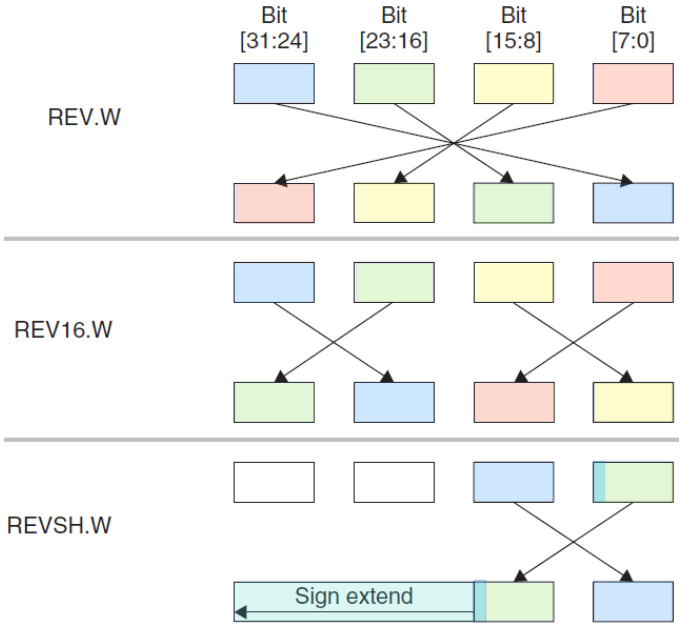


图 4.2 反序操作

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用得比较多，而且在这里面有大量的使用技巧。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	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

汇编语言：子程呼叫与无条件转移指令

最基本的无条件转移指令有两条：

```
B       Label       ;转移到 Label 处对应的地址
BX      reg         ;转移到由寄存器 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)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 fault 伺候。

呼叫子程序时，需要保存返回地址，正点的指令是：

```
BL      Label       ;转移到 Label 处对应的地址，并且把转移前的下条指令地址保存到 LR
BLX     reg         ;转移到由寄存器 reg 给出的地址，根据 REG 的 LSB 切换处理器状态，
                    ;并且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常人想不到的绝活，常见形式有：

```
MOV     PC,     R0          ;转移地址由 R0 给出
LDR     PC,     [R0]        ;转移地址存储在 R0 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，
                            ;从而实现转移（这也是 OS 惯用的一项必杀技——译注）
LDMIA   SP!,    {…, PC}     ;POP 的另一种等效写法
```

同理，使用这些密技，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"。

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）。

担任条件转移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 种转移判据，如下表 4.26 所示

表 4.26 转移及条件执行判据

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（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）
MOVGT   R2,     R0
MOVGT   R3,     R1
MOVLE   R2,     R0
MOVGT   R3,     R1
```

（本章的后面有对 IT 指令和 If-Then 块进行详细说明——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，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
```

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，因为有了 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。

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，它们的目的就是更新标志位，因此是会影响标志位的，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，但是它无条件更新标志位。它的用法和 CMP 的相同：

```
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，但是它无条件更新标志位。它的用法和 CMP 的相同：

```
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
```

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，如果存储器的映射关系，或者内存保护区的设置可以在运行时更改，（通过写 MMU/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 MMU/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。

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都执行完毕后，才提交(commit)在它后面的存储器访问动作。
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都执行完毕后，才执行它在后面的指令（亦即任何指令都要等待——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——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，通通清洗缓冲区。大虾们可以在有绝对信心时使用 DMB，新手还是保险点好。

同 DMB/DSB 相比，ISB 指令看起来似乎最小白。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。

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。

CM3 中的饱和运算指令分为两种：一种是“没有直流分量”的饱和——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。

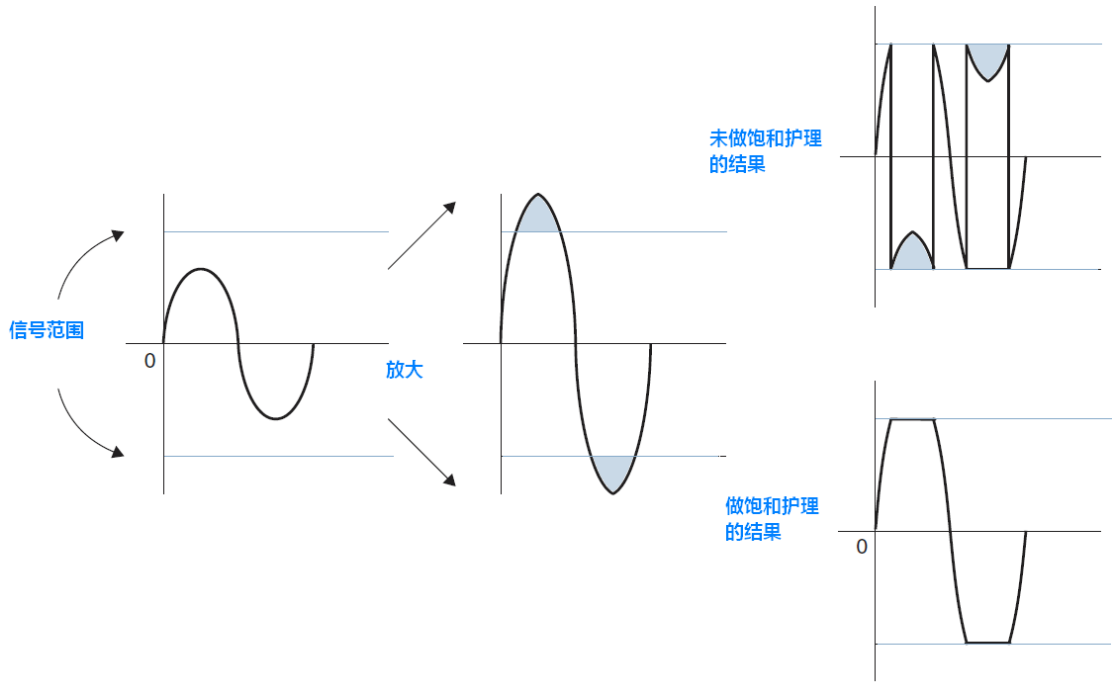


图 4.3　带符号饱和运算

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，这也是 APSR“露点”的部位。

Rn 存储“放大后的信号”，（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

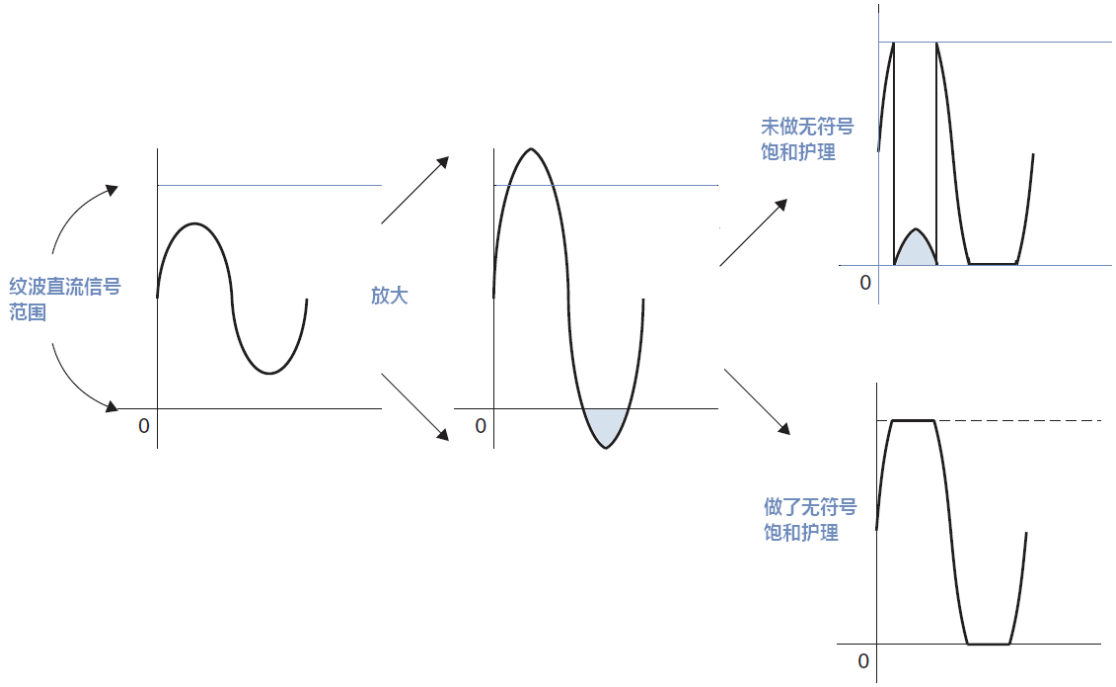


图 4.4　无符号饱和运算

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

CM3 中的一些有用的新指令

这里列出几条从 v6 和 v7 开始才支持的最新指令。

MRS 和 MSR

这两条指令是访问特殊功能寄存器的“绿色通道”——当然必须在特权级下，除 APSR 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（同时还包含了 PRIMASK 的功能，因为 faults 的优先级更高）
CONTROL	控制寄存器（堆栈选择，特权等级）

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
```

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 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 结构中的小型 if 块

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
    ADDEQ   R3, R4, R5      ; 相等时加法
    ASREQ   R3, R3, #1      ; 相等时算术右移
    ADDNE   R3, R6, R7      ; 不等时加法
    ASRNE   R3, R3, #1      ; 不等时算术右移
```

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。

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。

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

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（不信你去写段程序完成它的功能，看看要执行多久）。

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。

SXTB, SXTH, UXTB, UXTH

这 4 个指令是为优化 C 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
SXTB    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=0xffff8765
UXTB    R1, R0    ; R1=0x00000065
UXTH    R1, R0    ; R1=0x00008765
```

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，这对于系统程序和单片机程序往往非常有用。

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。

位段不支持首尾拼接。如 BFC R0, #27, #9 将产生不可预料的结果——译者注

例如，

```
LDR    R0, =0x1234FFFF
BFC    R0, #4, #10
```

执行完后，R0= 0x1234C00F

BFI（位段插入指令），则把某个寄存器按 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

LDRD/STRD

CM3 在一定程度上支持对 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。

TBB,TBH

高级语言都提供了“分类讨论”式控制结构，如 C 的 switch，Basic 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

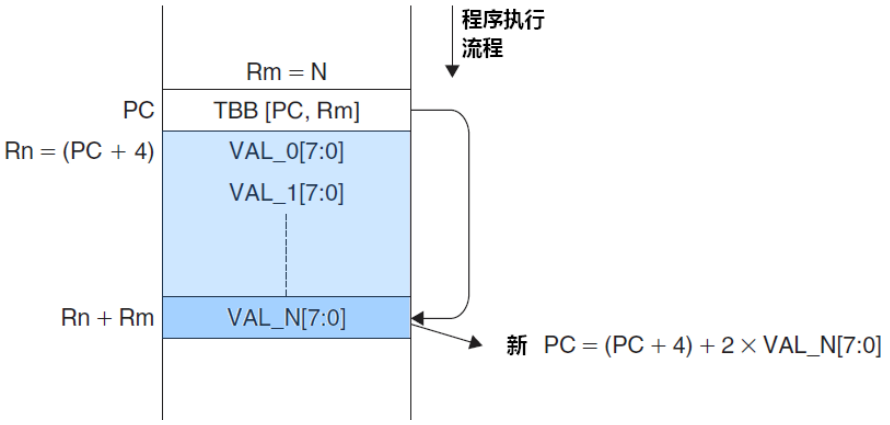


图 4.5 TBB 功能演示

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，甚至在特殊情况下，还要严重违反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：

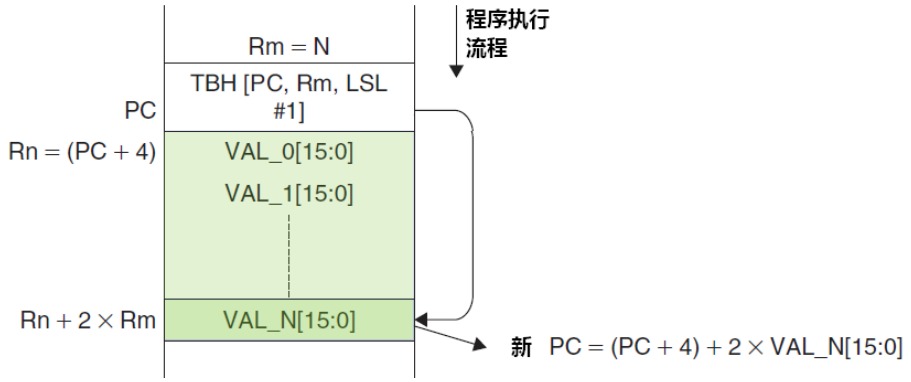


图 4.6 TBH 功能演示

TBH 跳转表的创建方式与 TBB 的类似，如下所示：

```
        TBH.W [pc, r0, LSL #1]   ; 执行此指令时，PC 的值正好等于 branchtable
branchtable
    DCI ((dest0 – branchtable)/2) ; 注意：数值是 16 位的，故使用 DCI 指示字
    DCI ((dest1 – branchtable)/2)
    DCI ((dest2 – branchtable)/2)
    DCI ((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
```

第5章

存储器系统

- 存储器系统的功能概览
- 存储器映射
- 存储器访问属性
- 缺省的存储器访问许可
- 位带操作
- 非对齐数据传送
- 互斥访问
- 端模式

存储系统功能概览

CM3 的存储器系统与从传统 ARM 架构的相比，已经脱胎换骨了：

第一， 它的存储器映射是预定义的，并且还规定好了哪个位置使用哪条总线。

第二， CM3 的存储器系统支持所谓的“位带”（bit-band）操作。通过它，实现了对单一比特的原子操作。位带操作仅适用于一些特殊的存储器区域中，见本章论述。

第三， CM3 的存储器系统支持非对齐访问和互斥访问。这两个特性是直到了 v7M 时才出来的。

最后，CM3 的存储器系统支持 both 小端配置和大端配置。

存储器映射

CM3 只有一个单一固定的存储器映射。这一点极大地方便了软件在各种 CM3 单片机间的移植。举个简单的例子，各款 CM3 单片机的 NVIC 和 MPU 都在相同的位置布设寄存器，使得它们变得通用。尽管如此，CM3 定出的条条框框是粗线条的，它依然允许芯片制造商灵活地分配存储器空间，以制造出各具特色的单片机产品。

存储空间的一些位置用于调试组件等私有外设，这个地址段被称为“私有外设区”。私有外设区的组件包括：

- 闪存地址重载及断点单元(FPB)
- 数据观察点单元(DWT)
- 指令跟踪宏单元(ITM)
- 嵌入式跟踪宏单元(ETM)
- 跟踪端口接口单元(TPIU)
- ROM 表

在后续讨论调试特性的章节中，将详细讲述这些组件。

CM3 的地址空间是 4GB, 程序可以在代码区，内部 SRAM 区以及外部 RAM 区中执行。但是因为指令总线与数据总线是分开的，最理想的是把程序放到代码区，从而使取指和数据访问各自使用自己的总线，并行不悖。

让我们先看一看这 4GB 的粗线条划分：

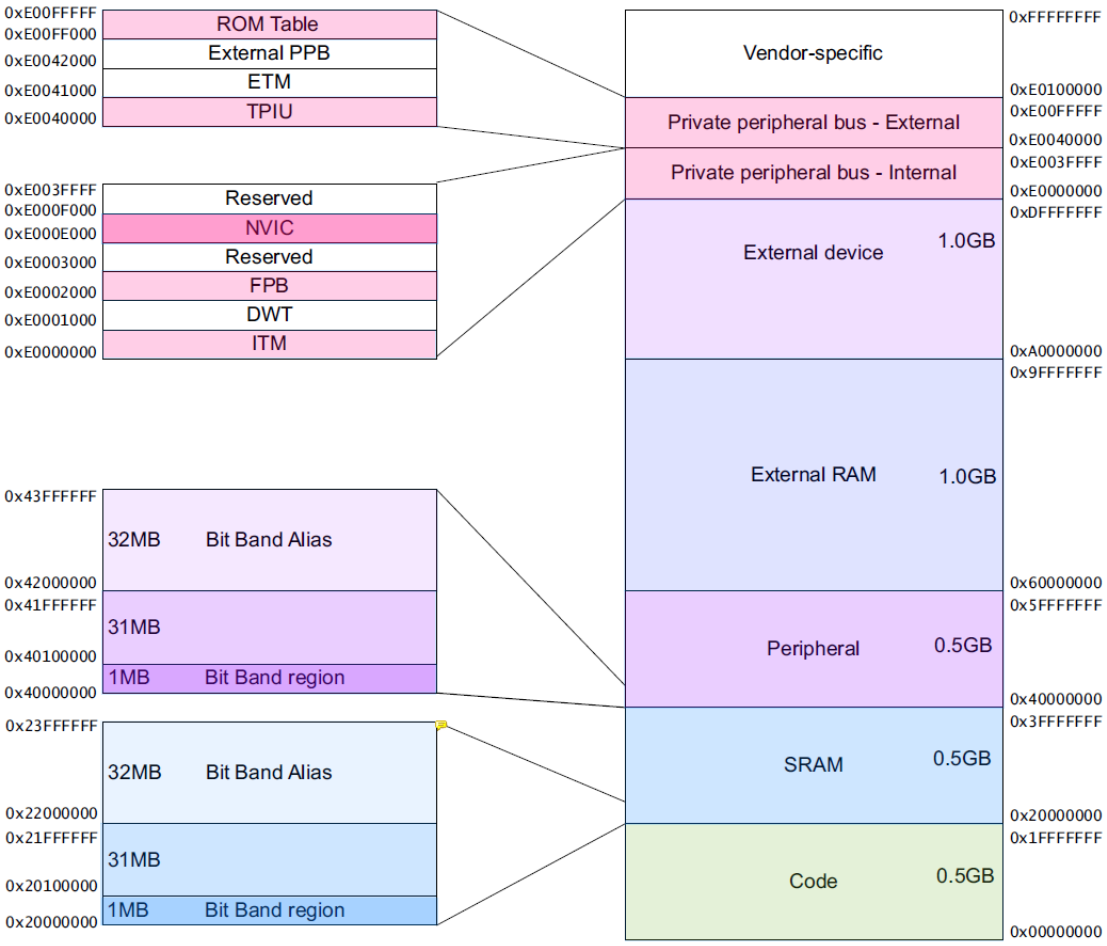


图 5.1 Cortex-M3 预定义的存储器映射

内部 SRAM 区的大小是 512MB，用于让芯片制造商连接片上的 SRAM，这个区通过系统总线来访问。在这个区的下部，有一个 1MB 的位带区，该位带区还有一个对应的 32MB 的“位带别名(alias)区”，容纳了 8M 个“位变量”（对比 8051 的只有 128 个位）。位带区对应的是最低的 1MB 地址范围，而位带别名区里面的每个字对应位带区的一个比特。位带操作只适用于数据访问，不适用于取指。通过位带的功能，可以把多个布尔型数据打包在单一的字中，却依然可以从位带别名区中，像访问普通内存一样地使用它们。位带别名区中的访问操作是原子的，消灭了传统的“读－改－写”三步曲。位带操作的细节待会还要讲到。

地址空间的另一个 512MB 范围由片上外设（的寄存器）使用。这个区中也有一条 32MB 的位带别名，以便于快捷地访问外设寄存器。例如，可以方便地访问各种控制位和状态位。要注意的是，外设内不允许执行指令。

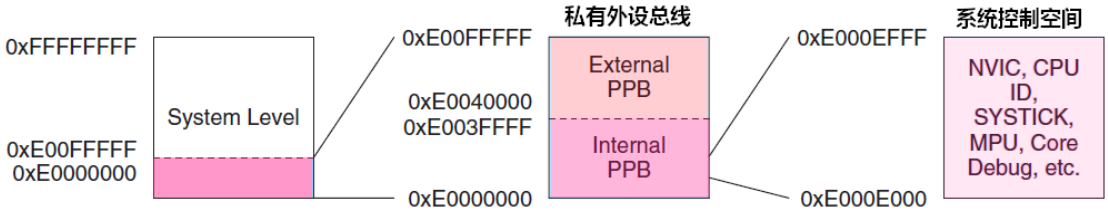
还有两个 1GB 的范围，分别用于连接外部 RAM 和外部设备，它们之中没有位带。两者的区别在于外部 RAM 区允许执行指令，而外部设备区则不允许。

最后还剩下 0.5GB 的隐秘地带，CM3 内核的闺房就在这里面，包括了系统级组件，内部私有外设总线 s，外部私有外设总线 s，以及由提供者定义的系统外设。

私有外设总线有两条：

- AHB 私有外设总线，只用于 CM3 内部的 AHB 外设，它们是：NVIC, FPB, DWT 和 ITM。
- APB 私有外设总线，既用于 CM3 内部的 APB 设备，也用于外部设备（这里的“外部”是对内核而言）。CM3 允许器件制造商再添加一些片上 APB 外设到 APB 私有总线上，它们通过 ABP 接口来访问。

NVIC 所处的区域叫做“系统控制空间（SCS）”，在 SCS 里的还有 SysTick、MPU 以及代码调试控制所用的寄存器，如图 5.2 所示：



最后，未用的提供商指定区也通过系统总线来访问，但是不允许在其中执行指令。

CM3 中的 MPU 是选配的，由芯片制造商决定是否配上。

上述的存储器映射只是个粗线条的模板，半导体厂家会提供更展开的图示，来表明芯片中片上外设的具体分布，RAM 与 ROM 的容量和位置信息。

存储器访问属性 s

CM3 在定义了存储器映射之外，还为存储器的访问规定了 4 种属性，分别是：

- 可否缓冲(Bufferable)
- 可否缓存(Cacheable)
- 可否执行(Executable)
- 可否共享(Sharable)

如果配了 MPU，则可以通过它配置不同的存储区，并且覆盖缺省的访问属性。CM3 片内没有配备缓存，也没有缓存控制器，但是允许在外部添加缓存。通常，如果提供了外部内存，芯片制造商还要附加一个内存控制器，它可以根据可否缓存的设置，来管理对片内和片外 RAM 的访问操作。地址空间可以通过另一种方式分为 8 个 512MB 等份：

1. 代码区（0x0000_0000- 0x1FFF_FFFF）。该区是可以执行指令的，缓存属性为 WT（“写通”，Write Through），即不可以缓存。此区亦可写数据。在此区上的数据操作是通过数据总线接口的（读数据使用 D-Code，写数据使用 System），且在此区上的写操作是缓冲的。
2. SRAM 区（0x2000_0000 – 0x3FFF_FFFF）。此区用于片内 SRAM，写操作是缓冲的，并且可以选择 WB-WA(Write Back, Write Allocated)缓存属性。此区亦可以执行指令，以允许把代码拷贝到内存中执行——常用于固件升级等维护工作。
3. 片上外设区(0x4000_0000 – 0x5FFF_FFFF)。该区用于片上外设，因此是不可缓存的，也不可以在此区执行指令（这也称为 eXecute Never, XN。ARM 的参考手册大量使用此术语）。
4. 外部 RAM 区的前半段（0x6000_0000 - 0x7FFF_FFFF）。该区用于片外 RAM，可缓存（缓存属性为 WB-WA），并且可以执行指令。
5. 外部 RAM 区的后半段（0x8000_0000 – 0x9FFF_FFFF）。除了不可缓存(WT)外，同前半段。
6. 外部外设区的前半段(0xA000_0000 – 0xBFFF_FFFF)。用于片外外设的寄存器，也用于多核系统中的共享内存（需要严格按顺序操作，即不可缓冲）。该区也是个不可执行区。
7. 外部外设区的后半段(0xC000_0000 – 0xDFFF_FFFF)。目前与前半段的功能完全一致。
8. 系统区(0xE000_0000 – 0xFFFF_FFFF)。此区是私有外设和供应商指定功能区。此区不可执行代码。系统区涉及到很多关键部位，因此访问都是严格序列化的（不可缓存，不可缓冲）。而供应商指定功能区则是可以缓存和缓冲的。

需要注意的是，在 CM3 的第一版中，代码区的存储器属性是被硬件连接成可缓存可缓冲的，无法通过 MPU 来更改。

译者添加

写通，写回，与写时申请

- 写回(Write Back)：写入的数据先逗留在缓存中，待到必要时再落实到最终目的地，这也是 cache 的最基本职能，用于改善数据传送的效率。
- 写通(Write Through)：写操作“穿透”中途的缓存，直接落入最终的目的地中。可见，写通操作架空了 cache，但它使写操作的结果立即生效。这常用于和片上外设或其它处理器共享的内存中，如显卡的显存，片上外设寄存器，以及双核系统中的共享内存。写通操作和 C 中的“volatile”有密切的联系。
- 写时申请(Write Allocate)：俺也不太清楚~

存储器的缺省访问许可

CM3 有一个缺省的存储访问许可，它能防止使用户代码访问系统控制存储空间，保护 NVIC、MPU 等关键部件。缺省访问许可在下列条件时生效：

- 没有配备 MPU
- 配备了 MPU，但是 MPU 被除能

如果启用了 MPU，则 MPU 可以在地址空间中划出若干个 regions，并为不同的 region 规定不同的访问许可权限。

缺省的存储器访问许可权限如表 5.1 所示

表 5.1 存储器的缺省访问许可

存储器区域	地址范围	用户级许可权限
代码区	0000_0000 – 1FFF_FFFF	无限制
片内 SRAM	2000_0000 – 3FFF_FFFF	无限制
片上外设	4000_0000 – 5FFF_FFFF	无限制
外部 RAM	6000_0000 – 9FFF_FFFF	无限制
外部外设	A000_0000 – DFFF_FFFF	无限制
ITM	E000_0000 – E000_0FFF	可以读。对于写操作，除了用户级下允许时的 stimulus 端口外，全部忽略
DWT	E000_1000 – E000_1FFF	阻止访问，访问会引发一个总线 fault
FPB	E000_2000 – E000_3FFF	阻止访问，访问会引发一个总线 fault
NVIC	E000_E000 – E000_EFFF	阻止访问，访问会引发一个总线 fault。但有个例外：软件触发中断寄存器可以被编程为允许用户级访问。
内部 PPB	E000_F000 – E003_FFFF	阻止访问，访问会引发一个总线 fault
TPIU	E004_0000 – E004_0FFF	阻止访问，访问会引发一个总线 fault
ETM	E004_1000 – E004_1FFF	阻止访问，访问会引发一个总线 fault
外部 PPB	E004_2000 – E004_2FFF	阻止访问，访问会引发一个总线 fault
ROM 表	E00F_F000 – E00F_FFFF	阻止访问，访问会引发一个总线 fault
供应商指定	E010_0000 – FFFF_FFFF	无限制

当一个用户级访问被阻止时，会立即产生一个总线 fault。

位带操作

支持了位带操作后，可以使用普通的加载/存储指令来对单一的比特进行读写。在 CM3 中，有两个区中实现了位带。其中一个是 SRAM 区的最低 1MB 范围，第二个则是片内外设区的最低 1MB 范围。这两个区中的地址除了可以像普通的 RAM 一样使用外，它们还都有自己的“位带别名区”，位带别名区把每个比特膨胀成一个 32 位的字。当你通过位带别名区访问这些字时，就可以达到访问原始比特的目的。

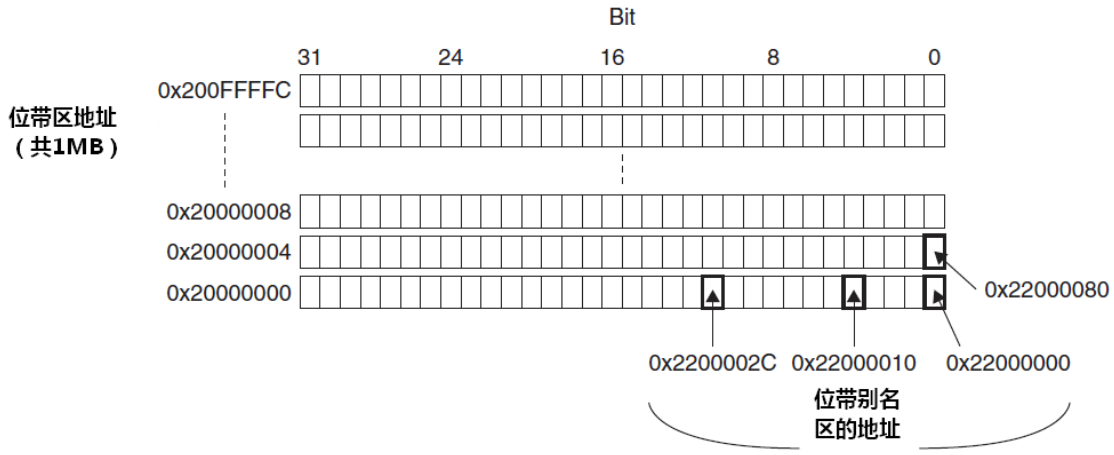


图 5.3A 位带区与位带别名区的膨胀关系图 A：

译者添加 下图从另一个侧面演示比特的膨胀对应关系

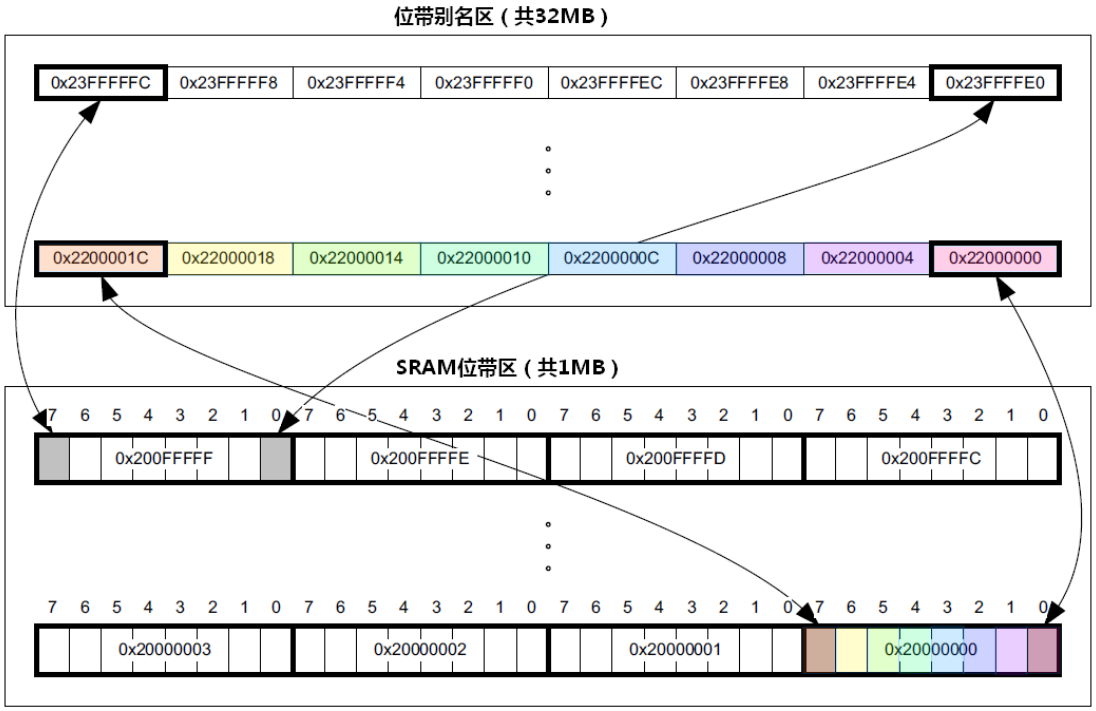


图 5.3B 位带区与位带别名区的膨胀对应关系图 B

举例：欲设置地址 0x2000_0000 中的比特 2，则使用位带操作的设置过程如下图所示：

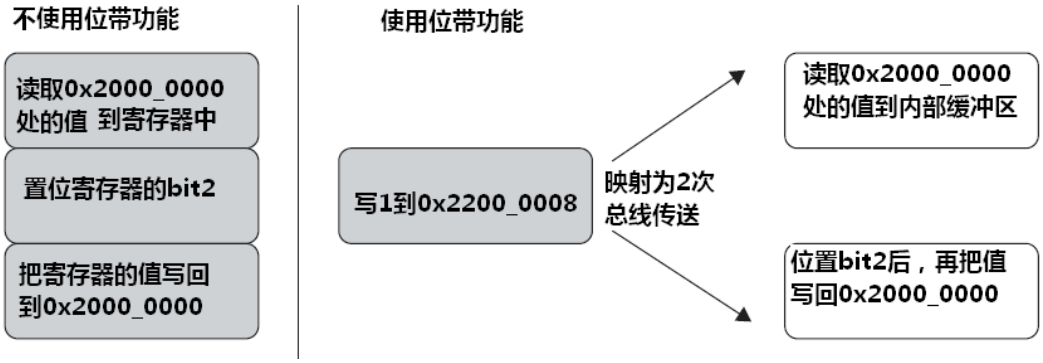


图 5.4　写数据到位带别名区

对应的汇编代码如图 5.5 所示

Without Bit-Band

```
LDR   R0,=0x20000000 ; Setup address
LDR   R1, [R0]       ; Read
ORR.W R1, #0x4       ; Modify bit
STR   R1, [R0] ; Write back result
```

With Bit-Band

```
LDR   R0,=0x22000008 ; Setup address
MOV   R1, #1         ; Setup data
STR   R1, [R0]       ; Write
```

图 5.5　位带操作与普通操作的对比，在汇编程序的角度上

位带读操作相对简单些：

图 5.6　从位带别名区中读取比特

无位带

```
LDR     R0,=0x20000000 ; 建立地址
LDR     R1, [R0]       ; Read
UBFX.W R1,R1, #2, #1   ; 提取bit2
```

有位带

```
LDR   R0,=0x22000008 ; 建立地址
LDR   R1, [R0]       ; Read
```

图 5.7　读取比特时传统方法与位带方法的比较

位带操作的概念其实 30 年前就有了，那还是 8051 单片机开创的先河。如今，CM3 将此能力进化，这里的位带操作是 8051 位寻址区的威力大幅加强版。

CM3 使用如下术语来表示位带存储的相关地址

- 位带区：　支持位带操作的地址区
- 位带别名：　对别名地址的访问最终作用到位带区的访问上（注意：这中途有一个

地址映射过程）

在位带区中，每个比特都映射到别名地址区的一个字——这是只有 LSB 有效的字。当一个别名地址被访问时，会先把该地址变换成位带地址。对于读操作，读取位带地址中的一个字，再把需要的位右移到 LSB，并把 LSB 返回。对于写操作，把需要写的位左移至对应的位序号处，然后执行一个**原子**的“读－改－写”过程。

支持位带操作的两个内存区的范围是：

0x2000_0000-0x200F_FFFF（SRAM 区中的最低 1MB）

0x4000_0000-0x400F_FFFF（片上外设区中的最低 1MB）

对于 SRAM 位带区的某个比特，记它所在字节地址为 A,位序号为 n(0<=n<=7)，则该比特在别名区的地址为：

AliasAddr＝ 0x22000000+((A-0x20000000)*8+n)*4 =0x22000000+ (A-0x20000000)*32 + n*4

对于片上外设位带区的某个比特，记它所在字节的地址为 A,位序号为 n(0<=n<=7)，则该比特在别名区的地址为：

AliasAddr＝ 0x42000000+((A-0x40000000)*8+n)*4 =0x42000000+ (A-0x40000000)*32 + n*4

上式中，“*4”表示一个字为 4 个字节，“*8”表示一个字节中有 8 个比特。

对于 SRAM 内存区，位带别名的重映射如表 5.2 所示：

表 5.2 SRAM 区中的位带地址映射

位带区	等效的别名地址
0x20000000.0	0x22000000.0
0x20000000.1	0x22000004.0
0x20000000.2	0x22000008.0
…	
0x20000000.31	0x2200007C.0
0x20000004.0	0x22000080.0
0x20000004.1	0x22000084.0
0x20000004.2	0x22000088.0
…	
0x200FFFFC.31	0x23FFFFFC.0

对于片上外设，映射关系如下表所示：

表 5.3 SRAM 区中的位带地址映射

位带区	等效的别名地址
0x40000000.0	0x42000000.0
0x40000000.1	0x42000004.0
0x40000000.2	0x42000008.0
…	
0x40000000.31	0x4200007C.0
0x40000004.0	0x42000080.0
0x40000004.1	0x42000084.0
0x40000004.2	0x42000088.0
…	
0x400FFFFC.31	0x43FFFFFC.0

这里再不嫌啰嗦地举一个例子：

1. 在地址 0x20000000 处写入 0x3355AACC
2. 读取地址 0x22000008。本次读访问将读取 0x20000000，并提取比特 2，值为 1。
3. 往地址 0x22000008 处写 0。本次操作将被映射成对地址 0x20000000 的“读－改－写”操作（原子的），把比特 2 清 0。
4. 现在再读取 0x20000000，将返回 0x3355AAC8（bit[2]已清零）。

位带别名区的字只有 LSB 有意义。另外，在访问位带别名区时，不管使用哪一种长度的数据传送指令（字/半字/字节），都把地址对齐到字的边界上，否则会产生不可预料的结果。

位带操作的优越性

位带操作有什么优越性呢？最容易想到的就是通过 GPIO 的管脚来单独控制每盏 LED 的点亮与熄灭。另一方面，也对操作串行接口器件提供了很大的方便（典型如 74HC165,CD4094）。总之位带操作对于硬件 I/O 密集型的底层程序最有用处了。

CM3 中还有一个称为“bit-bang”的概念，它通常是通过“bit-band”实现的，但是它俩在学术上是两个不同的概念。

位带操作还能用来化简跳转的判断。当跳转依据是某个位时，以前必须这样做：

- 读取整个寄存器
- 掩蔽不需要的位
- 比较并跳转

现在只需：

- 从位带别名区读取状态位
- 比较并跳转

使代码更简洁，这只是位带操作优越性的初等体现，位带操作还有一个重要的好处是在多任务中，用于实现共享资源在任务间的“互锁”访问。多任务的共享资源必须满足一次只有一个任务访问它——亦即所谓的“原子操作”。以前的读－改－写需要 3 条指令，导致这中间留有两个能被中断的空当。于是可能会出现如下图所示的紊乱危象：

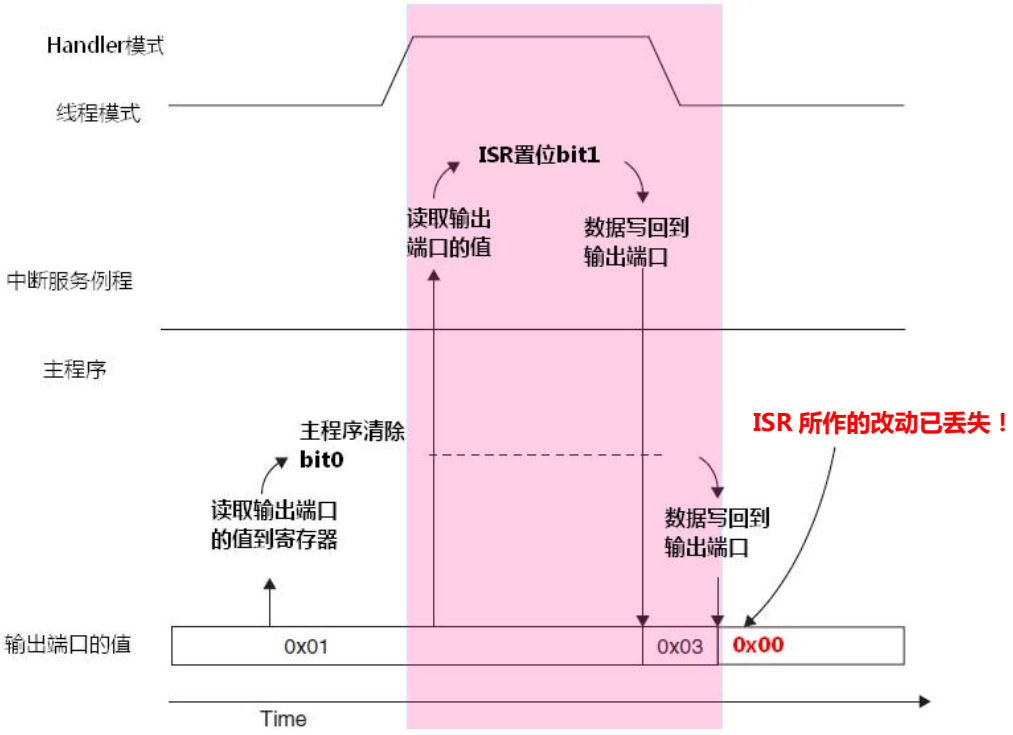


图 5.8 共享资源在紊乱危象下丢失数据演示

同样的紊乱危象可以出现在多任务的执行环境中。其实，图 5.8 所演示的情况可以看作是多任务的一个特例：主程序是一个任务，ISR 是另一个任务，这两个任务并发执行。

通过使用 CM3 的位带操作，就可以消灭上例中的紊乱危象。CM3 把这个“读－改－写”做成一个硬件级别支持的原子操作，不能被中断，如图 5.9 所演示

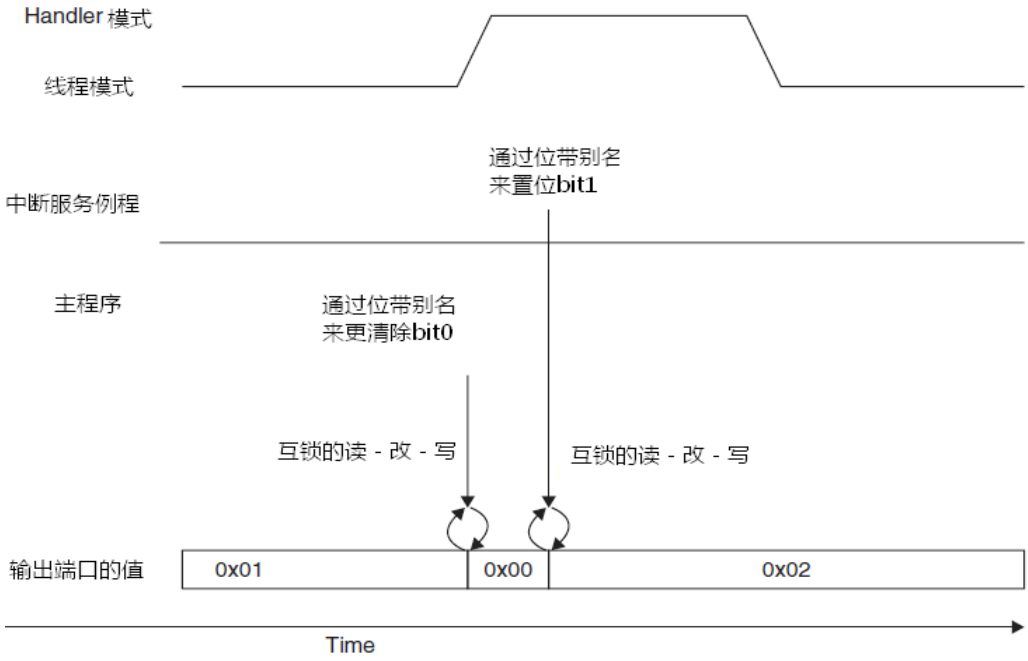


图 5.9 通过位带操作实现互锁访问，从而避免紊乱危象的演示

同样道理，多任务环境中的紊乱危象亦可以通过互锁访问来避免。

其它数据长度上的位带操作

位带操作并不只限于以字为单位的传送。亦可以按半字和字节为单位传送。例如，可以使用 LDRB/STRB 来以字节为长度单位去访问位带别名区，同理可用于 LDRH/STRH。但是不管用哪一个对子，都必须保证目标地址对齐到字的边界上。

在 C 语言中使用位带操作

不幸的是，在 C 编译器中并没有直接支持位带操作。比如，C 编译器并不知道同一块内存能够使用不同的地址来访问，也不知道对位带别名区的访问只对 LSB 有效。欲在 C 中使用位带操作，最简单的做法就是#define 一个位带别名区的地址。例如：

```
#define DEVICE_REG0 ((volatile unsigned long *) (0x40000000))
#define DEVICE_REG0_BIT0 ((volatile unsigned long *) (0x42000000))
#define DEVICE_REG0_BIT1 ((volatile unsigned long *) (0x42000004))
...
*DEVICE_REG0 = 0xAB;                    //使用正常地址访问寄存器
...
```

```
*DEVICE_REG0 = *DEVICE_REG0 | 0x2; //使用传统方法设置 bit1
...
*DEVICE_REG0_BIT1 = 0x1;            // 通过位带别名地址设置 bit1
```

为简化位带操作，也可以定义一些宏。比如，我们可以建立一个把“位带地址＋位序号”转换成别名地址的宏，再建立一个把别名地址转换成指针类型的宏：

```
//把“位带地址＋位序号”转换成别名地址的宏
#define BITBAND(addr, bitnum) ((addr & 0xF0000000)+0x2000000+((addr & 0xFFFFF)<<5)+(bitnum<<2))
//把该地址转换成一个指针
#define MEM_ADDR(addr) *((volatile unsigned long *) (adr))
```

在此基础上，我们就可以如下改写代码：

```
MEM_ADDR(DEVICE_REG0) = 0xAB;       //使用正常地址访问寄存器
MEM_ADDR(DEVICE_REG0)= MEM_ADDR(DEVICE_REG0) | 0x2;  //传统做法
MEM_ADDR(BITBAND(DEVICE_REG0,1)) = 0x1;              //使用位带别名地址
```

请注意：当你使用位带功能时，要访问的变量必须用 volatile 来定义。因为 C 编译器并不知道同一个比特可以有两个地址。所以就要通过 volatile，使得编译器每次都如实地把新数值写入存储器，而不再会出于优化的考虑，在中途使用寄存器来操作数据的复本，直到最后才把复本写回（这和 cache 的原理是一样的）。

译者添加

在 GCC 和 RealView MDK (即 Keil) 开发工具中，允许定义变量时手工指定其地址。如：

```
volatile unsigned long bbVarAry[7] __attribute__(( at(0x20003014) ));
volatile unsigned long* const pbbaVar= (void*)(0x22000000+0x3014*8*4);
```

这样，就在 0x20003014 处分配了 7 个字，共得到了 32*7=224 个比特。

在 long*后面的“const”通知编译器：该指针不能再被修改而指向其它地址。

注意：at()中的地址必须对齐到 4 字节边界。

再使用这些比特时，可以通过如下的形式：

```
pbbaVar[136]=1;          //置位第 136 号比特
```

不过这有个局限：编译器无法检查是否下标越界。那为什么不定义成“bbaVarAry[224]”的数组呢？这也是一个编译器的局限：它不知道这个数组其实就是 bbVarAry[7]，从而在计算程序对内存的占用量上，会平白无故地多计入 224*4 个字节。对于指针形式的定义，可以使用宏定义，为每个需要使用的比特取一个字面值的名字，在下标中只使用字面值名字，不再写真实的数字，就可以极大程度地避免数组越界。

请注意：在定义这“两个”变量时，前面加上了“volatile”。如果不再使用 bbVarAry 来访问这些比特，而仅仅使用位带别名的形式访问时，这两个 volatile 均不再需要。

非对齐数据传送

CM3 支持在单一的访问中使用非（地址）对齐的传送，数据存储器的访问无需对齐。在以前，ARM 处理器只允许对齐的数据传送。这种对齐是说：以字为单位的传送，其地址的最低两位必须是 0；以半字为单位的传送，其地址的 LSB 必须是 0；以字节为单位的传送则无所谓对不对齐。如果使用 0x1001,0x1002 或 0x1003 这样的地址做字传送，在以前的 ARM 处理器中则会触发一个数据流产（Data abort）异常——与 CM3 中总线 fault 异常的作用相同。

那么，非对齐访问看起来是什么样子呢？图 5.12-5.16 给出了 5 个例子。对于字的传送来说，任何一个不能被 4 整除的地址都是非对齐的。而对于半字，任何不能被 2 整除的地址（也就是奇数地址）都是非对齐的：

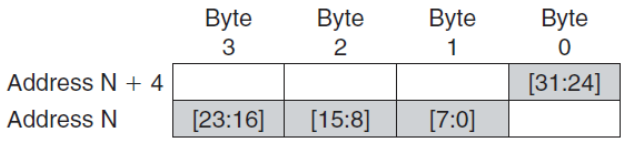


图 5.12 非对齐传送示例 1

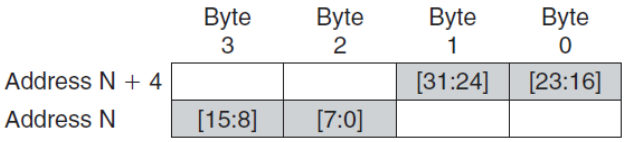


图 5.13 非对齐传送示例 2

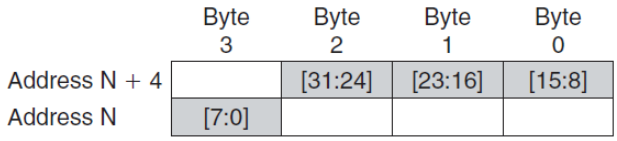


图 5.14 非对齐传送示例 3

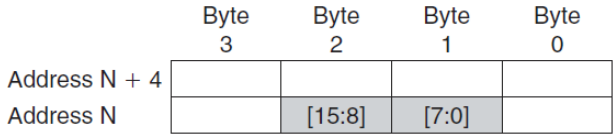


图 5.15 非对齐传送示例 4

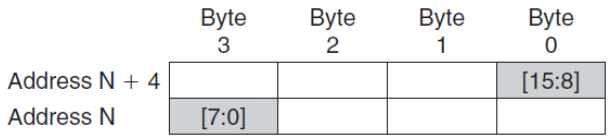


图 5.16 非对齐传送示例 5

在 CM3 中，非对齐的数据传送只发生在常规的数据传送指令中，如 LDR/LDRH/LDRSH。其它指令则不支持，包括：

- 多个数据的加载/存储(LDM/STM)
- 堆栈操作 PUSH/POP
- 互斥访问(LDREX/STREX)。如果非对齐会导致一个用法 fault
- 位带操作。因为只有 LSB 有效，非对齐的访问会导致不可预料的结果。

事实上，在内部是把非对齐的访问转换成若干个对齐的访问的，这种转换动作由处理器总线单元来完成。这个转换过程对程序员是透明的，因此写程序时不必操心。但是，因为它通过若干个对齐的访问来实现一个非对齐的访问，会需要更多的总线周期。事实上，节省内存有很多方法，但没有一个是通过压缩数据的地址，不惜破坏对齐性的这种歪门邪道。因此，应养成好习惯，总是保证地址对齐，这也是让程序可以移植到其它 ARM 芯片上的必要条件。

为此，可以编程 NVIC，使之监督地址对齐。当发现非对齐访问时触发一个 fault。具体的办法是设置“配置控制寄存器”中的 UNALIGN_TRP 位。这样，在整个调试期间就可以保证非对齐访问能当场被发现。

互斥访问

细心的读者可能会发现，CM3 中没有类似“SWP”的指令。在传统的 ARM 处理器中，SWP 指令是实现互斥体所必需的。到了 CM3，由所谓的互斥访问取代了 SWP 指令，以实现更加老练的共享资源访问保护机制。

互斥体在多任务环境中使用，也在中断服务例程和主程序之间使用，用于给任务申请共享资源（如一块共享内存）。在某个（排他型）共享资源被一个任务拥有后，直到这个任务释放它之前，其它任务是不得再访问它的。为建立一个互斥体，需要定义一个标志变量，用指示其对应的共享资源是否已经被某任务拥有。当另一个任务欲取得此共享资源时，它要先检查这个互斥体，以获知共享资源是否无人使用。在传统的 ARM 处理器中，这种检查操作是通过 SWP 指令来实现的。SWP 保证互斥体检查是原子操作的，从而避免了一个共享资源同时被两个任务占有（这是紊乱危象的一种常见表现形式）。

在新版的 ARM 处理器中，读/写访问往往使用不同的总线，导致 SWP 无法再保证操作的原子性，因为只有在同一条总线上的读/写能实现一个互锁的传送。因此，互锁传送必须用另外的机制实现，这就引入了“互斥访问”。互斥访问的理念同 SWP 非常相似，不同点在于：在互斥访问操作下，允许互斥体所在的地址被其它总线 master 访问，也允许被其它运行在本机上的任务访问，但是 CM3 能够“驳回”有可能导致竞态条件的互斥写操作。

互斥访问分为加载和存储，相应的指令对子为 LDREX/STREX, LDREXH/STREXH, LDREXB/STREXB，分别对应于字/半字/字节。为了介绍方便，以 LDREX/STREX 为例讲述它们的使用方式。

LDREX/STREX 的语法格式为：

```
LDREX      Rxf,          [Rn,       #offset]
STREX      Rd,    Rxf,   [Rn,       #offset]
```

（本节的以下内容是译者改编的）

LDREX 的语法同 LDR 相同，这里不再赘述。而 STREX 则不同。STREX 指令的执行是可以被“驳回”的。当处理器同意执行 STREX 时，Rxf 的值被存储到(Rn+offset)处，并且把 Rd 的值更新为 0。但若处理器驳回了 STREX 的执行，则不会发生存储动作，并且把 Rd 的值更新为 1。

其实，奥妙就在于这个“驳回”的规则上。规则可宽可严，最严格的规则是：

当遇到 STREX 指令时，仅当在这之前执行过 LDREX 指令，且在 LDREX 指令执行后没有执行过其它的 STR/STREX 指令，才允许执行 STREX 指令——也就是说只有在 LDREX 执行后最近的一条 STREX 才能成功执行。

其它情况下，驳回此 STREX。包括：

- 中途有其它的 STR 指令执行
- 中途有其它的 STREX 指令执行。

在使用互斥访问时，LDREX/STREX 必须成对使用。

为什么这种有条件的驳回可以避免紊乱危象呢？让我们举个简单的例子来演示。这个例子由主程序和一个中断服务例程组成。主程序尝试对(R0)自增两次，中断服务例程则把(R0).5 置位。计(R0)的初始值为 0。

```
MainProgram
    ;第一次互斥自增
TryInc1st
    LDREX  r2, [R0]
    ADD    r2, #1
    ;执行到这里时，处理器接收到外中断 3 请求，于是转到其中断服务例程 ISREx3 中
    STREX  R1, R2, [R0]       ; STREX 被驳回，R1=1，(R0)=0x20
TryInc2nd
    ;第二次互斥自增
    LDREX  r2, [R0]
    ADD    r2, #1
    STREX  R1, R2, [R0]       ; STREX 得到执行，R1=0，(R0)=0x21
    …
ISREx3
    ;处理器已经自动把 R0-R3, R12, LR, PC, PSR 压入栈
    LDR    R2,    [R0]
    ORR    R2,    #0x20
    STR    R2,    [R0]        ;在 ISREx3 中设置了(r0)的 Bit2
    BX     LR     ;返回时，处理器会自动把 R0-R3,R12,LR,PC,PSR 弹出堆栈
```

上例中，主程序在即将执行第一条 STREX 时，产生了外部中断#3。处理器打断主程序的执行，进入其服务例程 ISREx3，它对(R0)执行了一个写操作(STR)，因此在 ISRExt3 返回后，STREX 不再是 LDREX 执行后的第一条存储指令，故而被驳回。从而 ISREx3 对(R0)的改动就不会遭到破坏。随后主程序再次尝试自增运算，这一次在 STREX 执行前没有其它任何形式的存储指令，所有 STREX 成功执行。

如果主程序使用普通的 STR 会怎么样呢？对于第一次自增，主程序的 R2=1，于是执行后(R0)=1，结果，中断服务程序对(R0)的改动在此丢失！

上例是为演示方便才写了第 2 次自增尝试。实际情况是用循环实现的：

```
TryInc
    LDREX  r2, [R0]
    ADD    r2, #1
    STREX  R1, R2, [R0]
    CMP    R1, #1             ;检查 STREX 是否被驳回
```

```
BEQ     TryInc                  ;如果发现 STREX 被驳回，则重试。
```

LDREX/STREX 的工作原理其实很简单。仍然以上一段程序为例：当执行了 LDREX 后，处理器会在内部标记出一段地址。原则上，这段地址从 R0 开始，范围由芯片制造商定义。技术手册推荐的范围是在 4 字节至 4KB 之间，但是很多粗线条的实现会标记整个 4GB 的地址。在标记以后，对于第一个执行到的 STR/STREX 指令，只要其存储的地址落在标记范围内，就会清除此标记（对于整个 4GB 地址都被标记的情况，则任何存储指令都会清除此标记）。如果先后执行了两次 LDREX，则以后一个 LDREX 标记的地址为准。

执行 STREX 时，会先检查有没有做出过标记，如果有，还要检查存储地址是否落在标记范围内。只有通过了这两个关卡，STREX 才会执行。否则，就驳回 STREX。

当使用互斥访问时，在 CM3 总线接口上的内部写缓冲会被旁路，即使是 MPU 规定此区是可以缓冲的也不行。这保证了互斥体的更新总能在第一时间内完成，从而保证数据在各个总线 master 之间是一致的。SoC 系统的设计师如果设计多核系统，则必须保证各核之间看到的数据也是一致的。

译者添加的选读材料——互斥访问的深入研究

互斥访问可以递归使用，且最后一次递归的 LDREX/STREX 对子最先完成。如下例所示：

```
LDREXTestRecursive
    ldr     r3,     =N      ;递归次数 N，是一个预定义的常数
LoopWrapper
    push    {r0-r2, lr}
    ldr     r0,     =0x20003000
    sub     r3,     #1
TryInc
    ldrex   r1,     [r0]
    add     r1,     #1
    ldr     lr,     =DoSTREXRcsv
    cmp     r3,     #0
    bne     LoopWrapper
DoSTREXRcsv
    strex   r2,     r1,     [r0]
    cmp     r2,     #1
    beq     TryInc
    pop     {r0-r2, pc}
```

若执行前(0x20003000)=0，则执行后(0x20003000)＝N，且函数被递归调用 N 次。这段代码的工作流程难以用文字说清，一定要用模拟器跑过才容易理解

本例只是为了抛砖引玉。在实际的程序中，极少会这样钻牛角尖地直接递归。但是在多任务环境下，底层的函数库往往会“重入”，这也和递归的情形很相似。本例的执行结果，可以看出互斥访问能用于可重入函数在非关中情况下，保护共享资源的访问。

端模式

CM3 支持 both 小端模式和大端模式。但是，单片机其它部分的设计，包括总线的连接，内存控制器以及外设的性质等，也共同决定可以支持的内存类型。所以在设计软件之前，一定要先在单片机的数据手册上查清楚可以使用的端。在绝大多数情况下，基于 CM3 的单片机都使用小端模式——为了避免不必要的麻烦，在这里推荐读者清一色地使用小端模式。

CM3 中对大端模式的定义还与 ARM7 的不同（小端的定义都是相同的）。在 ARM7 中，大端的方式被称为“字不变大端”，而在 CM3 中，使用的是“字节不变大端”。如表 5.4 所示 。

表 5.4 CM3 的字节不变大端：存储器视图

地址，长度	Bits 31-24	Bits 23-16	Bits 15-8	Bits 7-0
0x1000，字	D[7:0]	D[15:8]	D[23:16]	D[31:24]
0x1000，半字	D[7:0]	D[15:8]	-	-
0x1002，半字	D[7:0]	D[15:8]		
0x1000，字节	D[7:0]			
0x1001，字节		D[7:0]		
0x1002，字节			D[7:0]	
0x1003，字节				D[7:0]

表 5.5 CM3 的字节不变大端：在 AHB 上的数据

地址，长度	Bits 31-24	Bits 23-16	Bits 15-8	Bits 7-0
0x1000，字	D[7:0]	D[15:8]	D[23:16]	D[31:24]
0x1000，半字			D[7:0]	D[15:8]
0x1002，半字	D[7:0]	D[15:8]		
0x1000，字节				D[7:0]
0x1001，字节			D[7:0]	
0x1002，字节		D[7:0]		
0x1003，字节	D[7:0]			

请注意：在 AHB 总线上的 BE-8 模式下，数据字节 lane 的传送格式是与小端模式一致的。

这是不同于 ARM7TDMI 的行为，它在大端模式下会有另一种总线 lane 安排，如表 5.6 所示。

表 5.6 ARM7 的字不变大端：在 AHB 上的数据

地址，长度	Bits 31-24	Bits 23-16	Bits 15-8	Bits 7-0
0x1000，字	D[7:0]	D[15:8]	D[23:16]	D[31:24]
0x1000，半字	D[7:0]	D[15:8]	-	-
0x1002，半字	D[7:0]	D[15:8]		
0x1000，字节	D[7:0]			
0x1001，字节		D[7:0]		
0x1002，字节			D[7:0]	
0x1003，字节				D[7:0]

在 CM3 中，是在复位时确定使用哪种端模式的，且运行时不得更改。指令预取永远使

用小端模式，在配置控制存储空间的访问也永远使用小端模式（包括 NVIC, FPB 之流）。另外，外部私有总线地址区 0xE0000000 至 0xE00FFFFF 也永远使用小端模式。

当你的 SoC 设计不支持大端模式，却有一些外设包含了大端模式时，可以轻易地使用 REV/REVH 指令来完成端模式的转换。