第 3 章

CHAPTER 3

ARM 指令集

在嵌入式系统开发中，目前最常用的编程语言是汇编语言和 C 语言。在较复杂的嵌入式软件中，由于 C 语言编写程序较方便，结构清晰，而且有大量支持库，所以大部分代码采用 C 语言编写，特别是基于操作系统的应用程序设计。但是在系统初始化、BootLoader、中断处理等，对时间和效率要求较严格的地方仍旧要使用汇编语言来编写相应代码块。本章将介绍 ARM 指令集指令及汇编语言的相关知识。

3.1 ARM 指令集概述

ARM 处理器的指令集主要有：

- ARM 指令集，是 ARM 处理器的原生 32 位指令集，所有指令长度都是 32 位，以字对齐(4 字节边界对齐)方式存储；该指令集效率高，但是代码密度较低。
- Thumb 指令集是 16 位指令集，2 字节边界对齐，是 ARM 指令集的子集；在具有较高代码密度的同时，仍然保持 ARM 的大多数性能优势。
- Thumb-2 指令集是对 Thumb 指令集的扩展，提供了几乎与 ARM 指令集完全相同的功能，同时具有 16 位和 32 位指令，既继承了 Thumb 指令集的高代码密度，又能实现 ARM 指令集的高性能；2 字节边界对齐，16 位和 32 位指令可自由混合。
- Thumb-2EE 指令集是 Thumb-2 指令集的一个变体，用于动态产生的代码；不能与 ARM 指令集和 Thumb 指令集交织在一起。

除了上面介绍的指令集外，ARM 处理器还有针对协处理器的扩展指令集，如普通协处理器指令、NEON 和 VFP 扩展指令集、无线 MMX 技术扩展指令集等。

3.1.1 指令格式

ARM 指令集的指令基本格式如下：

< opcode > {< cond >} { S } < Rd >, < Rn >, < shift_operand >

指令中“< >”内的项是必需的，“{ }”内的项是可选的。各个项目的具体含义如表 3-1 所示。

表 3-1 ARM 指令格式

符 号	说 明
opcode	操作码,即指令助记符,如 MOV、SUB、LDR 等
cond	条件码,描述指令执行的条件
S	可选后缀,指令后加上 S,指令执行成功完成后自动更新 CPSR 寄存器中的条件标志位
Rd	目的寄存器
Rn	存放第 1 个操作数的寄存器
shift_operand	第 2 个操作数,可以是寄存器、立即数等

3.1.2 指令的条件码

ARM 指令集中几乎所有指令都可以是条件执行的,由 cond 可选条件码来决定,位于 ARM 指令的最高 4 位[31：28],可以使用的条件码如表 3-2 所示。

表 3-2 ARM 指令条件码

指令条件码	助记符	CPSR 条件标志位值	含 义
0000	EQ	Z=1	相等
0001	NE	Z=0	不相等
0010	CS/HS	C=1	无符号数大于或等于
0011	CC/LO	C=0	无符号数小于
0100	MI	N=1	负数
0101	PL	N=0	正数或零
0110	VS	V=1	溢出
0111	VC	V=0	没有溢出
1000	HI	C=1,Z=0	无符号数大于
1001	LS	C=0,Z=1	无符号数小于或等于
1010	GE	N=V	有符号数大于或等于
1011	LT	N!=V	有符号数小于
1100	GT	Z=0,N=V	有符号数大于
1101	LE	Z=1,N!=V	有符号数小于或等于
1110	AL	任何	无条件执行(指令默认条件)
1111	NV	任何	从不执行(不要执行)

每种条件码的助记符由两个英文符号表示,在指令助记符的后面和指令同时执行。根据程序状态寄存器 CPSR 中的条件标志位[31：28]判断当前条件是否满足,若满足则执行指令。若指令中有后缀 S,则根据执行结果更新程序状态寄存器 CPSR 中的条件标志位[31：28]。

3.2 ARM 指令的寻址方式

寻址方式是指处理器根据指令中给出的地址信息,找出操作数所存放的物理地址,实现对操作数的访问。根据指令中给出的操作数的不同形式,ARM 指令系统支持的寻址方式有：立即寻址、寄存器寻址、寄存器间接寻址、寄存器移位寻址、变址寻址、多寄存器寻址、相

对寻址、堆栈寻址、块复制寻址等。

3.2.1 立即寻址


视频讲解

立即寻址也叫立即数寻址，指令的操作码字段后面的地址码部分不是操作数地址而是操作数本身，包含在指令的32位编码中。立即数前要加前缀“#”。

示例：

```
ADD     R0,R0,#1                ; R0 ← R0 + 1
MOV     R0,#0x00ff              ; R0 ← 0x00ff
```

3.2.2 寄存器寻址

寄存器寻址是指将操作数放在寄存器中，指令中地址码部分给出寄存器编号。这是各类微处理器常用的一种有较高执行效率的寻址方式。

示例：

```
ADD     R0,R1,R2                ; R0 ← R1 + R2
MOV     R0,R1                   ; R0 ← R1
```

3.2.3 寄存器间接寻址


视频讲解

操作数存放在存储器中，并将所存放的存储单元地址放入某一通用寄存器中，在指令中的地址码部分给出该通用寄存器的编号。

示例：

```
LDR     R0,[R1]                 ; R0 ← [R1]
```

如图3-1所示，该指令将寄存器R1中存放的值0xA0000008作为存储器地址，将该存储单元中的数据0x00000003传送到寄存器R0中。

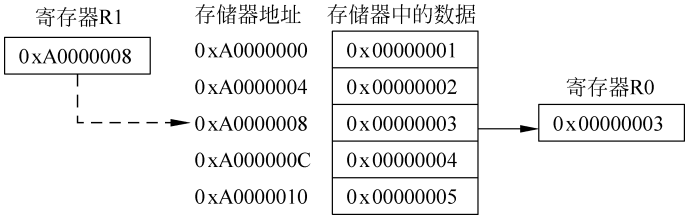

图3-1 寄存器间接寻址方式示意图

3.2.4 寄存器移位寻址

该指令中，寄存器的值在被送到ALU之前，先进行移位操作。移位的方式由助记符给出，移位的位数可由立即数或寄存器直接寻址方式表示。

可以采用的移位操作有：

- LSL：逻辑左移，寄存器值低端空出的位补 0。
- LSR：逻辑右移，寄存器值高端空出的位补 0。
- ASR：算术右移，算术移位操作对象是有符号数，位移过程中要保证操作数的符号不变，若操作数是正数，高端空出位补 0；若操作数是负数，高端空出位补 1。
- ROR：循环右移，从低端移出的位填入高端空出的位中。
- RRX：带扩展的循环右移，操作数右移 1 位，高端空出的位用 C 标志位填充。

示例：

```
MOV     R0,R1, LSL #2                ; R0 ← R1 中的数左移 2 位
ADD     R0, R1, R2, LSR #3           ; R0 ←R1 + R2 中的数右移 3 位
```

3.2.5　变址寻址

变址寻址方式是将某个寄存器(基址寄存器)的值与指令中给出的偏移量相加，形成操作数的有效地址，再根据该有效地址访问存储器。该寻址方式常用于访问在基址附近的存储单元。

示例：

```
LDR     R0,[R1, #2]                  ; R0 ← [R1+2]
```

该指令将 R1 寄存器的值 0xA0000008 加上位移量 2，形成操作数的有效地址，将该有效地址单元中的数据传送到寄存器 R0 中。

3.2.6　多寄存器寻址

多寄存器寻址方式可以在一条指令中传送多个寄存器的值，一条指令最多可以传送 16 个通用寄存器的值。连续的寄存器之间用“-”连接，不连续的中间用“,”分隔。

示例：

```
LDMIA     R0!,{R1-R3, R5}            ; R1 ← [R0]
                                     ; R2 ← [R0 + 4]
                                     ; R3 ← [R0 + 8]
                                     ; R5 ← [R0 + 12]
```

如图 3-2 所示，该指令将 R0 寄存器的值 0xA0000004 作为操作数地址，将存储器中该地址开始的连续单元中的数据传送到寄存器 R1、R2、R3、R5 中。

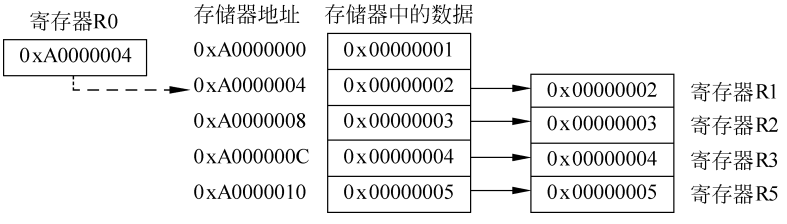

图 3-2　多寄存器寻址方式示意图

3.2.7 相对寻址

相对寻址方式就是以程序寄存器(PC)为基址寄存器,以指令中的地址标号为偏移量,两者相加形成操作数的有效地址。偏移量指出的是当前指令和地址标号之间的相对位置。子程序调用指令即是相对寻址方式。

示例:

```
BL      ADDR1                ; 跳转到子程序 ADDR1 处执行
...
ADDR1:
    ...
MOV     PC,LR                ; 从子程序返回
```

3.2.8 堆栈寻址

视频讲解

堆栈是按“先进后出”或“后进先出”方式进行存取的存储区。堆栈寻址是隐含的,使用一个叫作堆栈指针的专门寄存器,指示当前堆栈的栈顶。

根据堆栈的生成方式不同,分为递增堆栈和递减堆栈。当堆栈向高地址方向生长时,叫作递增堆栈(向上生长);当堆栈向低地址方向生长时,叫作递减堆栈(向下生长)。

堆栈指针指向最后压入堆栈的数据时,称为满堆栈;堆栈指针指向下一个将要放入数据的空位置时,称为空堆栈。

这样就有四种类型的堆栈工作方式:满递增堆栈(FA)、满递减堆栈(FD)、空递增堆栈(EA)、空递减堆栈(ED)。

示例:

```
STMFD     SP!,{R1-R3, LR}     ; 将寄存器 R1～R3 和 LR 压入堆栈,满递减堆栈
LDMFD     SP!,{R1-R3, LR}     ; 将堆栈数据出栈,放入寄存器 R1～R3 和 LR
```

3.2.9 块复制寻址

块复制寻址方式是多寄存器传送指令 LDM/STM 的寻址方式。LDM/STM 指令可以将存储器中的一个数据块复制到多个寄存器中,或将多个寄存器中的值复制到存储器中。寻址操作中使用的寄存器可以是 R0～R15 这 16 个寄存器的所有或子集。

根据基地址的增长方向是向上还是向下,以及地址的增减与指令操作的先后顺序(操作先进行还是地址先增减)的关系,有四种寻址方式:

- IB(Increment Before):地址先增加再完成操作,如 STMIB、LDMIB。
- IA(Increment After):先完成操作再地址增加,如 STMIA、LDMIA。
- DB(Decrement Before):地址先减少再完成操作,如 STMDB、LDMDB。
- DA(Decrement After):先完成操作再地址减少,如 STMDA、LDMDA。

3.3　ARM 指令集简介

ARM 指令集主要有跳转指令、数据处理指令、程序状态寄存器处理指令、加载/存储指令、协处理器指令和异常产生指令六大类。

ARM 指令集是加载/存储型的，指令的操作数都存储在寄存器中，处理结果直接放入目的寄存器中。采用专门的加载/存储指令来访问系统存储器。

本节介绍 ARM 指令集中常用指令的用法和使用要点。


视频讲解

3.3.1　跳转指令

跳转指令用于实现程序流程的跳转。在 ARM 程序中有两种方式可以实现程序流程的跳转：

① 直接向程序计数器 PC 中写入跳转地址，可以实现 4GB 地址空间内的任意跳转。例如：

```
LDR     PC, [PC, # + 0x00FF]                    ; PC ← [PC + 8 + 0x00FF]
```

② 使用专门的跳转指令。

ARM 指令集中的跳转指令可以完成从当前指令向前或向后的 32MB 地址空间的跳转。跳转指令主要包含以下几种指令。

1. B 指令

```
B{条件}  目标地址
```

跳转指令 B 是最简单的跳转指令，跳转到给定的目标地址，从那里继续执行。

示例：

```
B   WAITA          ; 无条件跳转到标号 WAITA 处执行
B   0x1234         ; 跳转到绝对地址 0x1234 处
```

2. BL 指令

```
BL{条件}  目标地址
```

BL 指令用于子程序调用，在跳转之前，将下一条指令的地址复制到链接寄存器 R14(LR)中，然后跳转到指定地址执行。

示例：

```
BL   FUNC1         ; 将当前 PC 值保存到 R14 中，然后跳转到标号 FUNC1 处执行
```

3. BLX 指令

```
BLX{条件}  目标地址
```

BLX 指令从 ARM 指令集跳转到指定地址执行，并将处理器的工作状态由 ARM 状态

切换到 Thumb 状态,同时将 PC 值保存到链接寄存器 R14 中。

示例:

```
BLX     FUNC1     ; 将当前 PC 值保存到 R14 中,然后跳转到标号 FUNC1 处执行,
                  ; 并切换到 Thumb 状态
BLX     R0        ; 将当前 PC 值保存到 R14 中,然后跳转 R0 中的地址处执行,
                  ; 并切换到 Thumb 状态
```

4. BX 指令

BX{条件} 目标地址

BX 指令是带状态切换的跳转指令,跳转到指定地址执行。若目标地址寄存器的位[0]为 1,处理器的工作状态切换为 Thumb 状态,同时将 CPSR 中的 T 标志位置 1,目标地址寄存器的位[31:1]复制到 PC 中;若目标地址寄存器的位[0]为 0,处理器的工作状态切换为 ARM 状态,同时将 CPSR 中的 T 标志位清 0,目标地址寄存器的位[31:1]复制到 PC 中。

示例:

```
BX   R0          ; 跳转 R0 中的地址处执行,如果 R0[0]=1,切换到 Thumb 状态
```

3.3.2 数据处理指令

视频讲解

数据处理指令主要完成寄存器中数据的各种运算操作。数据处理指令的使用原则:

- 所有操作数都是 32 位,可以是寄存器或立即数。
- 如果数据操作有结果,结果也为 32 位,放在目的寄存器中。
- 指令使用“两操作数”或“三操作数”方式,即每一个操作数寄存器和目的寄存器分别指定。
- 数据处理指令只能对寄存器的内容进行操作。指令后都可以选择 S 后缀来影响标志位。比较指令不需要后缀 S,这些指令执行后都会影响标志位。

1. MOV 指令

MOV{条件}{S} 目的寄存器,源操作数

MOV 指令将一个立即数、一个寄存器或被移位的寄存器传送到目的寄存器中。后缀 S 表示指令的操作是否影响标志位。如果目的寄存器是寄存器 PC 可以实现程序流程的跳转,寄存器 PC 作为目的寄存器且后缀 S 被设置,则在跳转的同时,将当前处理器工作模式下的 SPSR 值复制到 CPSR 中。

示例:

```
MOV   R0, #0x01              ; 将立即数 0x01 装入 R0
MOV   R0, R1                 ; 将寄存器 R1 的值传送到 R0
MOVS   R0, R1, LSL #3        ; 将寄存器 R1 的值左移 3 位后传送到 R0,并影响标志位
MOV   PC, LR                 ; 将链接寄存器 LR 的值传送到 PC 中,用于子程序返回
```

2. MVN 指令

MVN{条件}{S}　目的寄存器,源操作数

MVN 指令将一个立即数、一个寄存器或被移位的寄存器的值先按位求反,再传送到目的寄存器中,后缀 S 表示是否影响标志位。

示例:

```
MVN   R0, #0x0FF              ; 将立即数 0xFF 按位求反后装入 R0,操作后 R0=0xFFFFFF00
MVN   R0, R1                  ; 将寄存器 R1 的值按位求反后传送到 R0
```

3. ADD 指令

ADD{条件}{S}　目的寄存器,操作数 1,操作数 2

ADD 指令将两个操作数相加后,结果放入目的寄存器中,同时根据操作的结果影响标志位。

示例:

```
ADD   R0, R0, #1              ; R0 = R0 + 1
ADD   R0, R1, R2              ; R0 = R1 + R2
ADD   R0, R1, R2, LSL #3      ; R0 = R1 + (R2 << 3)
```

4. SUB 指令

SUB{条件}{S}　目的寄存器,操作数 1,操作数 2

SUB 指令用于把操作数 1 减去操作数 2,将结果放入目的寄存器中,同时根据操作的结果影响标志位。

示例:

```
SUB   R0, R0, #1              ; R0 = R0 - 1
SUB   R0, R1, R2              ; R0 = R1- R2
SUB   R0, R1, R2, LSL #3      ; R0 = R1 - (R2 << 3)
```

5. RSB 指令

RSB{条件}{S}　目的寄存器,操作数 1,操作数 2

RSB 指令称为逆向减法指令,用于把操作数 2 减去操作数 1,将结果放入目的寄存器中,同时根据操作的结果影响标志位。

示例:

```
RSB   R0, R0, #0xFFFF         ; R0 = 0xFFFF  - R0
RSB   R0, R1, R2              ; R0 = R2 - R1
```

6. ADC 指令

ADC{条件}{S}　目的寄存器,操作数 1,操作数 2

ADC 指令将两个操作数相加后，再加上 CPSR 中的 C 标志位的值，将结果放入目的寄存器中，同时根据操作的结果影响标志位。

示例：

```
ADDS   R0, R0, R2
ADC    R1, R1, R3          ; 用于 64 位数据加法,(R1,R0)=(R1,R0)+(R3,R2)
```

7. SBC 指令

SBC{条件}{S} 目的寄存器,操作数 1,操作数 2

SBC 指令用于操作数 1 减去操作数 2，再减去 CPSR 中的 C 标志位值的反码，将结果放入目的寄存器中，同时根据操作的结果影响标志位。

示例：

```
SUBS   R0, R0, R2
SBC    R1, R1, R3          ; 用于 64 位数据减法,(R1,R0)=(R1,R0)-(R3,R2)
```

8. RSC 指令

RSC{条件}{S} 目的寄存器,操作数 1,操作数 2

RSC 指令用于操作数 2 减去操作数 1，再减去 CPSR 中的 C 标志位值的反码，将结果放入目的寄存器中，同时根据操作的结果影响标志位。

示例：

```
RSBS   R2, R0, #0
RSC    R3, R1, #0          ; 用于求 64 位数据的负数
RSC    R0, R1, R2          ; R0 = R2 - R1-!C
```

9. AND 指令

```
AND{条件}{S}  目的寄存器,操作数 1,操作数 2
```

AND 指令实现两个操作数的逻辑与操作，将结果放入目的寄存器中，同时根据操作的结果影响标志位；常用于将操作数某些位清 0。

示例：

```
AND    R0, R1, R2          ; R0 = R1 & R2
AND    R0, R0, #3          ; R0 的位 0 和位 1 不变,其余位清 0
```

10. ORR 指令

ORR{条件}{S} 目的寄存器,操作数 1,操作数 2

ORR 指令实现两个操作数的逻辑或操作，将结果放入目的寄存器中，同时根据操作的结果影响标志位。常用于将操作数某些位置 1。

示例：

```
ORR   R0, R0, #3              ; R0 的位 0 和位 1 置 1,其余位不变
```

11. EOR 指令

EOR{条件}{S}　目的寄存器,操作数 1,操作数 2

EOR 指令实现两个操作数的逻辑异或操作,将结果放入目的寄存器中,同时根据操作的结果影响标志位。常用于将操作数某些位置取反。

示例:

```
EOR   R0, R0, #0F             ; R0 的低 4 位取反
```

12. BIC 指令

BIC{条件}{S}　目的寄存器,操作数 1,操作数 2

BIC 指令用于清除操作数 1 的某些位,将结果放入目的寄存器中,同时根据操作的结果影响标志位。操作数 2 为 32 位掩码,掩码中设置了哪些位则清除操作数 1 中这些位。

示例:

```
BIC   R0, R0, #0F             ; 将 R1 的低 4 位清 0,其他位不变
```

13. CMP 指令

CMP{条件}　操作数 1,操作数 2

CMP 指令用于把一个寄存器的值减去另一个寄存器的值或立即数,根据结果设置 CPSR 中的标志位,但不保存结果。

示例:

```
CMP   R1, R0          ; 将 R1 的值减去 R0 的值,并根据结果设置 CPSR 的标志位
CMP   R1, #0x200      ; 将 R1 的值减去 0x200,并根据结果设置 CPSR 的标志位
```

14. CMN 指令

CMN{条件}　操作数 1,操作数 2

CMN 指令用于把一个寄存器的值减去另一个寄存器或立即数取反的值,根据结果设置 CPSR 中的标志位,但不保存结果。该指令实际完成两个操作数的加法。

示例:

```
CMN   R1, R0          ; 将 R1 的值和 R0 的值相加,并根据结果设置 CPSR 的标志位
CMN   R1, #0x200      ; 将 R1 的值和立即数 0x200 相加,并根据结果设置 CPSR 的标志位
```

15. TST 指令

TST{条件}　操作数 1,操作数 2

TST 指令用于把一个寄存器的值和另一个寄存器的值或立即数进行按位与运算,根据

结果设置 CPSR 中的标志位,但不保存结果。该指令常用于检测特定位的值。

示例:

```
TST   R1, #0x0F              ; 检测 R1 的低 4 为是否为 0
```

16. TEQ 指令

TEQ{条件} 操作数 1,操作数 2

TST 指令用于把一个寄存器的值和另一个寄存器的值或立即数进行按位异或运算,根据结果设置 CPSR 中的标志位,但不保存结果。该指令常用于检测两个操作数是否相等。

示例:

```
TEQ   R1, R2          ; 将 R1 的值和 R2 的值进行异或运算,并根据结果设置 CPSR 的标志位
```

3.3.3 程序状态寄存器处理指令

视频讲解

MRS 指令和 MSR 指令用于在状态寄存器和通用寄存器间传输数据。状态寄存器的值要通过“读取→修改→写回”三个步骤操作来实现,可先用 MRS 指令将状态寄存器的值复制到通用寄存器中,修改后再通过 MSR 指令把通用寄存器的值写回状态寄存器。

示例:

```
MRS   R0, CPSR          ; 将 CPSR 的值复制到 R0 中
ORR   R0, R0, #C0       ; R0 的位 6 和位 7 置 1,即屏蔽外部中断和快速中断
MSR   CPSR, R0          ; 将 R0 值写回到 CPSR 中
```

MRS 指令和 MSR 指令的格式如下:

MRS{条件} 通用寄存器, 程序状态寄存器(CPSR 或 SPSR)
MSR{条件} 程序状态寄存器(CPSR 或 SPSR)_<域>, 操作数

其中,MSR 指令中的<域>可用于设置程序状态寄存器中需要操作的位:

- 位[31 : 24]为条件标志位域,用 f 表示。
- 位[23 : 16]为状态位域,用 s 表示。
- 位[15 : 8]为扩展位域,用 x 表示。
- 位[7 : 0]为控制位域,用 c 表示。

示例:

```
MSR   CPSR_cxsf, R3
```

3.3.4 加载/存储指令

视频讲解

加载/存储指令用于在寄存器和存储器之间传输数据,Load 指令用于将存储器中的数据传输到寄存器中,Store 指令用于将寄存器中的数据保存到存储器中。

1. LDR 指令

LDR{条件}　目的寄存器, <存储器地址>

LDR 指令将一个 32 位字数据传输到目的寄存器中。如果目的寄存器是 PC,从存储器中读出的数据将作为目的地址,以实现程序流程的跳转。

示例:

```
LDR   R1, [R0, #0x12]          ; 将存储器地址为 R0+0x12 的字数据写入 R1
LDR   R1, [R0, R2]             ; 将存储器地址为(R0+R2)的字数据写入 R1
```

2. STR 指令

STR{条件}　源寄存器, <存储器地址>

STR 指令用于从源寄存器中将一个 32 位字数据写入存储器中。

示例:

```
STR   R1, [R0, #0x12]          ; 将 R1 中的字数据写入以 R0+0x12 为地址的存储器中
STR   R1, [R0], #0x12          ; 将 R1 中的字数据写入以 R0+0x12 为地址的存储器中,
                               ; 并将新地址 R0+0x12 写入 R0
```

3. LDM 和 STM 指令

LDM(或 STM){条件}{类型}　基址寄存器{!},寄存器列表{^}

LDM 指令和 STM 指令实现一组寄存器和一片连续存储空间之间的数据传输。LDM 指令加载多个寄存器,STM 指令存储多个寄存器,它们常用于现场保护、数据复制、参数传输等,有 8 种模式:

- IA:每次传送后地址加 4。
- IB:每次传送前地址加 4。
- DA:每次传送后地址减 4。
- DB:每次传送前地址减 4。
- FD:满递减堆栈。
- ED:空递减堆栈。
- FA:满递增堆栈。
- EA:空递增堆栈。

可选后缀{!},选用该后缀,当数据传输完成后,将最后地址写入基址寄存器中,否则基址寄存器值不变;基址寄存器不能为 R15(PC),寄存器列表可以是 R0～R15 的任意组合。

可选后缀{^},当指令为 LDM 且寄存器列表中有 R15(PC),选用该后缀表示除了完成数据传输以外,还将 SPSR 复制到 CPSR。

示例:

```
LDMIA   R0, {R3-R9}      ; R0 指向的存储器单元的数据,保存到 R3～R9 中,R0 值不更新
STMIA   R1!, {R3-R9}     ; 将 R3～R9 数据存储到 R1 指向的存储器单元中,R1 的值更新
```

4. SWP 指令

SWP{条件}　目的寄存器,源寄存器 1,[源寄存器 2]

SWP 指令用于将源寄存器 2 所指向的存储器中的字数据传输到目的寄存器中,同时将源寄存器 1 中的字数据传输到源寄存器 2 所指向的存储器中。当源寄存器 1 和目的寄存器为同一个寄存器时,该指令完成该寄存器和存储器内容的交换。

示例:

```
SWP   R1, R1, [R0]     ; 将 R1 的内容与 R0 指向的存储单元的内容进行交换
SWP   R1, R2, [R0]     ; 将 R0 指向的存储单元的内容写入 R1 中,并将 R2 的内容写入
                       ; 该内存单元中
```

3.3.5 协处理器指令

视频讲解

ARM 体系结构允许通过增加协处理器来扩展指令集。ARM 协处理器具有自己专用的寄存器组,它们的状态由控制 ARM 状态的指令的镜像指令来控制。程序的控制流指令由 ARM 处理器来处理,所有的协处理器指令只能同数据处理和数据传送有关。

ARM 协处理器指令可完成下面 3 类操作:

- ARM 协处理器的数据处理操作。
- ARM 处理器和协处理器的寄存器之间数据传输。
- ARM 协处理器的寄存器和存储器之间数据传输。

ARM 协处理器指令主要包括 5 条,它们的格式和功能如表 3-3 所示。

表 3-3　ARM 协处理器指令

助　记　符	说　　明	功　　能
CDP coproc, opcodel, CRd, CRn, CRm{,opcode2}	协处理器数据操作指令	用于 ARM 处理器通知协处理器执行特定的操作
LDC{L} coproc,CRd <地址>	协处理器数据读取指令	从某一连续的存储单元将数据读取到协处理器的寄存器中
STC{L} coproc,CRd <地址>	协处理器数据写入指令	将协处理器的寄存器数据写入某一连续的存储单元中
MCR coproc, opcodel, Rd, CRn{, opcode2}	ARM 寄存器到协处理器寄存器的数据传输指令	将 ARM 处理器的寄存器中的数据传输到协处理器的寄存器中
MRC coproc, opcodel, Rd, CRn{, opcode2}	协处理器寄存器到 ARM 寄存器的数据传输指令	将协处理器的寄存器中的数据传输到 ARM 处理器的寄存器中

3.3.6 异常产生指令

视频讲解

ARM 处理器有两条异常产生指令,软中断指令(SWI)和断点中断指令(BKPT)。

1. SWI 指令

SWI{条件}　24 位立即数

SWI 指令用于产生 SWI 异常中断，实现从用户模式切换到管理模式，CPSR 保存到管理模式下的 SPSR 中，执行转移到 SWI 向量。其他模式下也可使用 SWI 指令，同样切换到管理模式。该指令不影响条件码标志。

示例：

```
SWI   0x02          ; 软中断，调用操作系统编号为 0x02 的系统例程
```

2. BKPT 指令

BKPT　16 位立即数

BKPT 指令产生软件断点中断，软件调试程序可以使用该中断。立即数会被 ARM 硬件忽视，但能被调试工具利用来得到有用的信息。

示例：

```
BKPT   0xFF32
```

3.4　Thumb 指令集简介

视频讲解

为了兼容存储系统总线宽度为 16 位的应用系统，ARM 体系结构中提供了 16 位 Thumb 指令集，它可以被看作 ARM 指令压缩形式的子集，是针对代码密度的问题而提出的，它具有 16 位的代码密度，这对于嵌入式系统来说至关重要。

Thumb 不是一个完整的体系结构，不能指望处理器只执行 Thumb 指令而不支持 ARM 指令集。因此，Thumb 指令只需要支持通用功能，必要时可以借助完善的 ARM 指令集。只要遵循一定的调用规则，Thumb 子程序和 ARM 子程序可以互相调用。当处理器在执行 ARM 程序段时，称 ARM 处理器处于 ARM 工作状态；当处理器在执行 Thumb 程序段时，称 ARM 处理器处于 Thumb 工作状态。

Thumb 指令集没有协处理器指令、信号量指令以及访问 CPSR 或 SPSR 的指令，没有乘加指令及 64 位乘法指令等，并且指令的第二操作数受到限制；除了分支指令 B 有条件执行功能外，其他指令均为无条件执行；大多数 Thumb 数据处理指令采用 2 地址格式。

Thumb 指令集与 ARM 指令集的区别一般有如下 4 点：

① 跳转指令。程序相对转移，特别是条件跳转与 ARM 代码下的跳转相比，在范围上有更多的限制，转向子程序是无条件的转移。

② 数据处理指令。Thumb 数据处理指令是对通用寄存器进行操作，在大多数情况下，操作的结果必须放入其中一个操作数寄存器中，而不是第 3 个寄存器中。Thumb 数据处理操作比 ARM 状态的更少。访问 R8～R15 受到一定限制：除 MOV 和 ADD 指令访问寄存器 R8～R15 外，其他数据处理指令总是更新 CPSR 中 ALU 状态标志；访问寄存器 R8～R15 的 Thumb 数据处理指令不能更新 CPSR 中的 ALU 状态标志。

③ 单寄存器加载和存储指令。在 Thumb 状态下，单寄存器加载和存储指令只能访问寄存器 R0～R7。

④ 多寄存器加载和多寄存器存储指令。LDM 和 STM 指令可以将任何范围为 R0～R7

的寄存器子集加载或存储。PUSH 和 POP 指令使用堆栈指针 R13 作为基址实现满递减堆栈。除 R0～R7 外，PUSH 指令还可以存储链接寄存器 R14，并且 POP 指令可以加载程序计数器 PC。

3.5 ARM 汇编语言编程简介

3.5.1 伪操作

ARM 汇编语言程序是由机器指令、伪指令和伪操作组成的。伪操作是 ARM 汇编语言程序里的一些特殊的指令助记符，和指令系统中的助记符不同，这些助记符没有相应的操作码。伪操作主要是为完成汇编程序做一些准备工作，在源程序汇编过程中起作用，一旦汇编完成，伪操作的使命就完成。

宏是一段独立的程序代码，通过伪操作定义，在程序中使用宏指令即可调用宏。当程序被汇编时，汇编程序校对每个宏调用进行展开，用宏定义代替源程序中的宏指令。

1. 符号定义伪操作

符号定义伪操作用于定义 ARM 汇编程序中的变量、对变量赋值及定义寄存器名称等。常用伪操作有：

- GBLA、GBLL 和 GBLS：定义全局变量。
- LCLA、LCLL 和 LCLS：定义局部变量。
- SETA、SETL 和 SETS：为变量赋值。
- RLIST：为通用寄存器列表定义名称。
- CN：为协处理器的寄存器定义名称。
- CP：为协处理器定义名称。
- DN 和 SN：为 VFP 的寄存器定义名称。
- FN：为 FPA 的浮点寄存器定义名称。

2. 数据定义伪操作

数据定义伪操作用于数据表定义、文字池定义、数据空间分配等。常用伪操作有：

- LTORG：声明一个数据缓冲池的开始。
- MAP：定义一个结构化的内存表的首地址。
- FIELD：定义结构化内存表的一个数据域。
- SPACE：分配一块内存空间，并用 0 初始化。
- DCB：分配一段字节的内存单元，并用指定的数据初始化。
- DCD 和 DCDU：分配一段字的内存单元，并用指定的数据初始化。
- DCFD 和 DCFDU：分配一段双字的内存单元，并用双精度的浮点数据初始化。
- DCFS 和 DCFSU：分配一段字的内存单元，并用单精度的浮点数据初始化。
- DCQ 和 DCQU：分配一段双字的内存单元，并用 64 位整型数据初始化。
- DCW 和 DCWU：分配一段半字的内存单元，并用指定的数据初始化。

3. 汇编控制伪操作

汇编控制伪操作用于条件汇编、宏定义、重复汇编控制等，常用伪操作有：

- IF、ELSE 和 ENDIF：根据条件把一段源程序代码包括在汇编程序内或排除在程序之外。

- WHILE 和 WEND：根据条件重复汇编相同的源程序代码段。
- MACRO 和 MEND：MACRO 标识宏定义的开始，MEND 标识宏定义结束。用 MACRO 和 MEND 定义一段代码，称为宏定义体，在程序中可以通过宏指令多次调用该代码段。
- MEXIT：用于从宏中跳转出去。

4. 其他伪操作

其他伪操作常用的有段定义伪操作、入口点设置伪操作、包含文件伪操作、标号导出或引入声明等。

- ALIGN：边界对齐。
- AREA：段定义。
- CODE16 和 CODE32：指令集定义。
- END：汇编结束。
- ENTRY：程序入口。
- EQU：常量定义。
- EXPORT 和 GLORBAL：声明一个符号可以被其他文件引用。
- IMPORT 和 EXTERN：声明一个外部符号。
- GET 和 INCLUDE：包含文件。
- INCBIN：包含不被汇编的文件。
- RN：给特定的寄存器命名。
- ROUT：标记局部标号使用范围的界限。

视频讲解

3.5.2 伪指令

ARM 中的伪指令并不是真正的 ARM 或 Thumb 指令，这些伪指令在汇编编译器对源程序进行汇编处理时被替换成对应 ARM 或 Thumb 指令(序列)。常用的伪指令如下。

(1) ADR

ADR 为小范围的地址读取伪指令，该指令将基于 PC 的相对偏移地址或基于寄存器的相对偏移地址读取到寄存器中。格式为

```
ADR {cond} register, expr
```

- cond 是可选的指令执行条件。
- register 是目的寄存器。
- expr 是基于 PC 或基于寄存器的地址表达式，当地址值是字节对齐时，取值范围为－255～255B；当地址值是字对齐时，取值范围为－1020～1020B；当地址值是 16 字节对齐时，取值范围更大。

(2) ADRL

ADRL 为中等范围的地址读取伪指令，该指令比 ADR 的取值范围更大。格式为

```
ADRL {cond} register, expr
```

- cond 是可选的指令执行条件。
- register 是目的寄存器。

- expr 是基于 PC 或基于寄存器的地址表达式，当地址值是字节对齐时，取值范围为－64～64KB；当地址值是字对齐时，取值范围为－256～256KB；当地址值是 16 字节对齐时，取值范围更大；在 32 位的 Thumb-2 指令中，取值范围可为－1～1MB。

(3) LDR

LDR 为大范围的地址读取伪指令，将一个 32 位的常数或者一个地址值读取到寄存器中。格式为

```
LDR {cond} register, = [expr|label-expr]
```

- cond 是可选的指令执行条件。
- register 是目的寄存器。
- expr 是 32 位常量。

(4) NOP

NOP 是空操作伪指令，在汇编时被替换成 ARM 中的空操作。

3.5.3 汇编语句格式

ARM(Thumb)汇编语言的语句格式为

```
[标号] <指令|条件|S> <操作数>[; 注释]
```

- 在 ARM 汇编程序中，ARM 指令、伪操作、伪指令、伪操作的助记符全部用大写字母，或者全部用小写字母，不能既有大写也有小写字母。
- 所有标号在一行的顶格书写，后面不要添加“：”，所有指令不能顶格书写。
- 注释内容以“;”开头到本行结束。
- 源程序中允许有空行，如果单行太长，可以用字符“\”将其分开，“\”后不能有任何字符，包括空格和制表符等。
- 变量的设置，常量的定义，其标识符必须在一行顶格书写。

3.5.4 汇编语言的程序结构

段(section)是 ARM 汇编语言组织源文件的基本单位，是独立的、具有特定名称的、不可分割的指令或数据序列。段分为代码段和数据段，代码段存放执行代码，数据段存放代码执行时需要的数据。一个 ARM 汇编程序至少需要一个代码段，较大的程序可以包含多个代码段和数据段。

ARM 汇编语言源程序经过汇编后生成可执行的映像文件，文件格式有 axm、bin、elf、hex 等。可执行的映像文件通常包括 3 部分：

- 一个或多个代码段，代码段的属性为只读。
- 0 个或多个包含初始化数据的数据段，属性为可读可写。
- 0 个或多个不包含初始化数据的数据段，属性为可读可写。

链接器根据一定的规则将各个段安排到内存的相应位置。源程序中段之间的相对位置与可执行的映像文件中段的相对位置不一定相同。

下面的程序说明了 ARM 汇编语言程序的基本结构：

```
AREA   BUF, DATA, READWRITE                ; 声明数据段 BUF
count  DCB    30                           ; 定义一个字节单元 count
AREA   EXAMPLE1, CODE, READONLY            ; 声明代码段 EXAMPLE1
ENTRY                                      ; 程序入口
CODE32                                     ; 声明 32 位 ARM 指令
START
    LDRB    R0, count                      ; R0 = count
    MOV     R1, #10                        ; R1 = 8
    ADD     R0, R0, R1                     ; R0 = R0 + R1
    B       START
END
```

3.6　C 语言与汇编语言的混合编程

视频讲解

在嵌入式开发中，C 语言是一种常见的程序设计语言。C 语言程序可读性强、易维护、可移植性和可靠性高。ARM 体系结构不仅支持汇编语言也支持 C 语言，在一些情况下需要采用汇编语言和 C 语言混合编程。

3.6.1　C 程序中内嵌汇编

视频讲解

视频讲解

在 C 语言程序中嵌入汇编可以完成一些 C 语言不能完成的操作，同时代码效率也比较高。

在 ARM C 语言程序中使用关键词__asm 来标识一段汇编指令代码，格式为：

```
__asm                          //asm 前 2 个下画线
{
    instruction [; instruction]
    ...
    [instruction]
}
```

如果一行有多条汇编指令，指令间用“;”隔开；如果一条指令占多行，要使用续行符号“\”。

在 ARM C 语言程序中也可以使用关键词 asm 来标识一段汇编指令代码，格式为：

```
asm("instruction [; instruction]");
```

3.6.2　汇编中访问 C 语言程序变量

在 C 语言中声明的全局变量可以被汇编语言通过地址间接访问。

例如，在 C 语言程序中已经声明了一个全局变量 glovbvar，通过 IMPORT 伪指令声明外部变量的方式访问：

```
AREA     EXAMPLE2, CODE, REDAONLY
IMPORT   glovbvar                ; 声明外部变量 glovbvar
START
```

```
    LDR     R1, =glovbvar      ; 装载外部变量地址
    LDR     R0, [R1]           ; 读出全局变量 glovbvar 数据
    ADD     R0, R0, #1
    STR     R0,[R1]            ; 保存变量值
    MOV     PC, LR
END
```

3.6.3 ARM 中的汇编和 C 语言相互调用

为了使单独编译的 C 语言程序和汇编语言程序之间能够相互调用,必须遵守 ATPCS 规则。ATPCS 即 ARM-Thumb Procedure Call Standard(ARM-Thumb 过程调用标准)的简称。

ATPCS 规定了应用程序的函数可以如何分开地写,分开地编译,最后将它们连接在一起,所以它实际上定义了一套有关过程(函数)调用者与被调用者之间的协议。基本 ATPCS 规定了在子程序调用时的一些基本规则,包括 3 个方面的内容:

- 各寄存器的使用规则及其相应的名称。
- 数据栈的使用规则。
- 参数传递的规则。

1. C 程序调用汇编程序

C 程序调用汇编程序首先通过 extern 声明要调用的汇编程序模块,声明中形参个数要与汇编程序模块中需要的变量个数一致,且参数传递要满足 ATPCS 规则,然后在 C 程序中调用。

示例:

```
#include <stdio.h>
extern void strcopy(char *d, char *s);          //使用关键词声明
int main()
{
char *srcstr = "first";
char *dststr = "second";
strcopy(dststr,srcstr);                         //汇编模块调用
}
```

被调用的汇编程序:

```
AREA    Scopy, CODE, REDAONLY
EXPORT  strcopy                          ; 使用 EXPORT 伪操作声明本汇编程序
strcopy
LDRB    R2, [R1], #1
STRB    R2, [R0], #1
CMP     R2, #0
BNE strcopy
MOV     PC, LR
END
```

2. 汇编程序调用 C 程序

在调用之前必须根据 C 语言模块中需要的参数个数,以及 ATPCS 参数规则,完成参数

传递，即前 4 个参数通过 R0～R3 传递，后面的参数通过堆栈传递，然后再利用 B、BL 指令调用。

示例：

```
int g(int a,int b,int c,int d,int e)      //C 语言函数原型
{
return (a+b+c+d+e);
}
//汇编程序调用 C 程序 g()计算 i+2*i+3*i+4*i+5*i 的结果
EXPORT f
AREA     f, CODE, REDAONLY
IMPORT g                                  //声明 C 程序函数 g()
STR LR, {SP, #-4}!                        //保存 PC
ADD R1 ,R0, R0
ADD R2, R1, R0
ADD R3, R1, R2
STR R3, {SP, #-4}!
ADD R3, R1, R1
BL g                                      //调用 C 程序函数 g()
ADD SP, SP, #4
LDR PC, [SP], #4
END
```

3.7　本章小结

本章首先对 ARM 处理器指令的 9 种寻址方式进行了说明，并详细介绍了 ARM 指令集中各种指令的格式、功能和使用方法，简单介绍了 16 位的 Thumb 指令；随后介绍了 ARM 汇编语言的伪操作、伪指令和汇编语句格式，通过示例讲述了汇编语言程序的结构；最后讲述了 C 语言和汇编语言混合编程的规则和方法。通过本章，读者了解到 ARM 程序设计的基本知识，为基于 ARM 处理器的嵌入式软硬件开发奠定基础。

习题

1. 请写出对应 C 代码的 ARM 指令。

C 代码：

```
If(a > b)
a++;
Else
b++;
```

2. 请写出下列 ARM 指令的功能。

```
MOV   R1, #0x10          ;
MOV   R0, R1             ;
MOVS R3, R1, LSL #2      ;
```

```
MOV   PC,LR
```

3. 在前文提到了实现 Thumb 状态和 ThumbEE 状态之间切换的指令为 ENTERX 指令和 LEAVEX 指令，请查阅相关资料对这两个命令进行比较。

4. 请查阅资料，试比较 ARM、Thumb、Thumb-2、Thumb-2EE 指令集的区别。

5. 存储器从 0x400000 开始的 100 个单元中存放着 ASCII 码，编写汇编语言程序，将其所有的小写字母转换成大写字母，对其他的 ASCII 码不做转换。

6. 使用 ARM 汇编语言指令编写一个实现冒泡排序功能的程序段。

7. 编写一个实现数组排序的 C 语言程序，要求调用第 6 题中用汇编语言编写的冒泡排序程序段。

8. 嵌入式系统中经常要用到无限循环，怎样用 C 语言编写死循环呢？

9. 请评论如下代码段：

```
unsigned int zero = 0;
unsigned int compzero = 0xFFFF; //1's complement of zero
```