第3章

CHAPTER 3

ARM微处理器指令系统

- ARM微处理器指令系统
  - ARM指令集概述
    - 基本格式
    - 指令的条件码
  - 基本寻址方式
    - 寄存器寻址
    - 立即寻址
    - 寄存器移位寻址
    - 寄存器间接寻址
    - 变址寻址
    - 多寄存器寻址
    - 块复制寻址
    - 堆栈寻址
    - 相对寻址
  - ARM指令集简介
    - 存储器访问指令
    - 数据处理指令
    - 乘法指令
    - ARM跳转指令
    - 程序状态寄存器处理指令
    - 协处理器指令
    - ARM杂项指令
  - Thumb指令集
  - Thumb-2技术
  - 其他扩展指令集
  - ARM汇编程序设计
    - 伪操作
    - 伪指令
    - 汇编语句格式
    - 汇编语言的程序结构概述
  - ATPCS介绍
    - 寄存器使用规则
    - 数据栈使用规则
    - 参数传递规则
  - C语言与汇编语言的混合编程
    - GND内联汇编
    - 汇编程序与C语言程序的相互调用

本章思维导图

学习目标：

(1) 理解 ARM 指令系统基础；
(2) 掌握寻址方式和 ARM 指令集；
(3) 了解 Thumb 指令集及 Thumb-2 技术；
(4) 认识扩展指令集；
(5) 理解 ATPCS 标准；
(6) 掌握汇编语言编程基础、C 语言与汇编语言混合编程技巧。

技能目标：

(1) 具备指令解析能力、指令运用能力、Thumb 技术优化能力；
(2) 具备汇编语言编程实践能力、混合编程实现能力。

视频讲解

3.1 ARM 指令集概述

3.1.1 基本格式

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

其中，“< >”内的项是必需的；“{ }”内的项是可选的。

(1) opcode 为指令助记符，如 LDR、STR 等。

(2) cond 为指令执行条件，如 EQ、NE 等，是可选的，如果不写则使用默认条件 AL(无条件执行)。

(3) S 为可选后缀，指令后加上 S，指令执行成功完成后自动更新 CPSR 寄存器的条件标志位。

(4) Rd 为目标寄存器。

(5) Rn 为第 1 个操作数的寄存器。

(6) shift_operand 为第 2 个操作数。在 ARM 指令中，灵活地使用第 2 个操作数能提高代码效率。

指令格式举例：

```
LDR R0,[R1]            ;读取 R1 地址上的存储器单元内容,装载到寄存器 R0 中(执行条件 AL)
BEQ DT1                ;跳转指令,执行条件 EQ,相等则跳转到 DT1
ADDS R1,R1,#0x01       ;加法指令,R1←R1 + 0x01,带有 S,自动更新 CPSR 寄存器条件标志位
```

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，由 cond 可选条件码来决定，位于 ARM 指令的最高 4 位[31:28]。ARM 条件码如表 3-1 所示。

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]。

表 3-1 ARM 条件码

操作码[31:28]	条件码助记符	标 志	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

视频讲解

3.2 基本寻址方式

寻址方式是根据指令中给出的地址信息,找出操作数存放的地址,以实现对操作数访问的方式。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寄存器寻址、立即寻址、寄存器移位寻址、寄存器间接寻址、变址寻址、多寄存器寻址、块复制寻址、堆栈寻址、相对寻址 9 种基本寻址方式。

1. 寄存器寻址

寄存器寻址是指操作数的值在寄存器中,指令中的地址码字段给出的是寄存器编号,寄存器的内容是操作数,指令执行时直接取出寄存器值的操作。这是各类微处理器常用的一种具有较高执行效率的寻址方式。

示例:

```
MOV     R1,R2                   ;R1←R2
SUB     R0,R1,R2                ;R0←R1 - R2
```

2. 立即寻址

在立即寻址指令中,数据就包含在指令当中,立即寻址指令的操作码字段后面的地址码部分就是操作数本身,取出指令也就取出了可以立即使用的操作数(也称为立即数)。立即数要以"#"为前缀,十六进制数值时以 0x 表示。

示例:

```
ADD     R0,R0,#1                ;R0←R0 + 1
MOV     R0,#0xff00              ;R0←0xff00
```

注意有效立即数问题：指令中的立即数是由一个8bit的常数移动偶数位(0,2,4,…,26,28,30)得到的。因此，每一条指令都包含一个8bit的常数X(占据指令二进制编码[7:0]位)和移位值Y(占据指令二进制编码[11:8]位)，得到的立即数等于8bit常数X循环右移偶数(2×Y)位。立即数表示方式如图3-1所示。

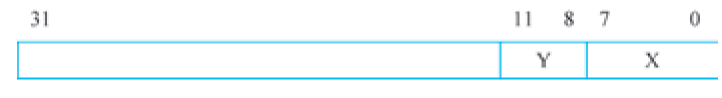


图3-1 立即数表示方式

下面列举了一些有效的立即数：

0xff、0x104、0xff0、0xff00、0xff000、0xff000000、0xf000000f

下面是一些无效的立即数：

0x101、0x102、0xff1、0xff04、0xff003、0xffffffff、0xf000001f

3. 寄存器移位寻址

寄存器移位寻址是ARM指令集特有的寻址方式。指令中的第2个操作数<shifter_operand>是由寄存器中的值移位得到的。寄存器的值在被送到ALU之前，可以事先经过桶形移位寄存器的处理。预处理和移位发生在同一周期内，因此有效地使用移位寄存器，可以提高代码的执行效率。

示例：

```
MOV     R0, R2, LSL #0x03          ;R2 的值逻辑左移 3 位,结果放入 R0,即 R0 = R2 * 23
ANDS    R1, R1, R2, LSL R3         ;R2 的值逻辑左移 R3 位,然后和 R1 相"与"操作,结果放入 R1
```

可采用的移位操作如下。

(1) LSL逻辑左移(Logical Shift Left)，寄存器中字的低端空出的位补0。

(2) LSR逻辑右移(Logical Shift Right)，寄存器中字的高端空出的位补0。

(3) ASR算术右移(Arithmetic Shift Right)，移位过程中保持符号位不变，即如果源操作数为正数，则字的高端空出的位补0，否则补1。

(4) ROR循环右移(Rotate Right)，由字的低端移出的位填入字的高端空出的位。

(5) RRX带扩展的循环右移(Rotate Right eXtended by 1 place)，操作数右移1位，高端空出的位用原C标志值填充。

4. 寄存器间接寻址

寄存器间接寻址方式中，操作数的地址不是直接给出的，而是存储在一个寄存器中，即寄存器中的值不是操作数本身，而是指向操作数所在存储单元的地址。这种方式使得程序可以更加灵活地访问内存中的数据，因为可以通过修改寄存器中的值改变访问的内存地址。寄存器间接寻址主要使用在Load/Store指令中。这种寻址方式提高了程序的灵活性和效率，因为它允许程序在运行时动态地改变访问的内存地址，而不需要在编译时就确定所有的内存地址。

示例：

```
LDR R1, [R2]            ;将 R2 中的值作为地址,取出此地址中的数据保存在 R1 中
STR R1, [R2]            ;将 R2 中的值作为地址,取出 R1 中的数据存入 R2 所指向的地址
```

5. 变址寻址

变址寻址是将基址寄存器的内容与指令中给出的偏移量相加，形成操作数的有效地址。这种方式允许程序在运行时动态地计算内存地址，从而提高了访问内存的灵活性和效率。通过结合基址寄存器和偏移量，变址寻址可以方便地访问数组、表格等功能部件寄存器中的数据。

变址寻址有前变址寻址方式、自动变址寻址方式和后变址寻址方式。

1）前变址寻址方式

前变址寻址方式是指在指令执行前，将基址寄存器的内容与指令中给出的偏移量相加，得到操作数的有效地址，然后访问该地址空间中的数据。这种方式的特点是，基址寄存器的值在指令执行前被修改，但在指令执行后保持不变。

前变址寻址方式适用于需要访问固定偏移量处的数据，但不希望修改基址寄存器的值的情况。例如，在遍历数组时，可以使用前变址寻址方式访问数组中的元素，同时保持数组的起始地址不变。

示例：

```
LDR R0,[R1,#0x04]                    ;R0←[R1 + 0x04]
```

2）自动变址寻址方式

自动变址寻址方式是指在指令执行前，将基址寄存器的内容与指令中给出的偏移量相加，得到操作数的有效地址，然后访问该地址空间中的数据。与前变址寻址方式不同的是，自动变址寻址方式在指令执行后，会将基址寄存器的值更新为新的有效地址。

自动变址寻址方式适用于需要连续访问内存中的数据，并希望每次访问后自动更新基址寄存器的值的情况。例如，在遍历数组时，可以使用自动索引寻址方式连续访问数组中的元素，同时自动更新数组的索引地址。

示例：

```
LDR R0,[R1,#0x04]!                   ;R0←[R1 + 0x04],R1←R1 + 0x04
```

3）后变址寻址方式

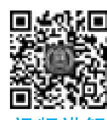
后变址寻址方式是指在指令执行前，先使用基址寄存器的内容作为操作数的有效地址，访问该地址空间中的数据。在指令执行后，将基址寄存器的内容与指令中给出的偏移量相加，得到新的有效地址，并更新基址寄存器的值。

后变址寻址方式适用于需要访问内存中的数据，并希望在访问后自动更新基址寄存器的值以便进行后续访问的场景。例如，在遍历数组时，可以使用后变址寻址方式访问数组中的元素，并在访问后自动更新数组的索引地址。与自动变址寻址方式相比，后变址寻址方式在数据访问和基址寄存器更新之间存在一个明确的先后顺序。

示例：

```
LDR R0,[R1], #0x04                   ;R0←[R1],R1←R1 + 0x04
```

说明，前变址、自动变址和后变址寻址方式各有特点，适用于不同的编程场景，根据具体情况选择使用。


视频讲解

6. 多寄存器寻址

多寄存器寻址一次可传送几个寄存器值，允许一条指令传送16个寄存器的任何子集或所有寄存器。连续的寄存器之间用“-”连接，不连续的中间用“,”分隔。

示例：

```
LDMIA R1!,{R2 - R4,R7}              ;R2←[R1]
;R3←[R1 + 4]
;R4←[R1 + 8]
;R7←[R1 + 12]
;R1←[R1 + 16]
```

7. 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(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，有以下4种寻址方式。

(1) IB(Increment Before)：地址先增加再完成操作，如STMIB、LDMIB。

(2) IA(Increment After)：先完成操作再地址增加，如STMIA、LDMIA。

(3) DB(Decrement Before)：地址先减少再完成操作，如STMDB、LDMDB。

(4) DA(Decrement After)：先完成操作再地址减少，如STMDA、LDMDA。

8. 堆栈寻址

堆栈是按“先进后出”或“后进先出”方式进行存取的存储区。堆栈操作寻址方式和多寄存器Load/Store指令寻址方式十分类似。但对于堆栈的操作，数据写入内存和从内存中读出要使用不同的寻址模式。

根据不同的寻址方式，将堆栈分为以下4种。

(1) 满堆栈：堆栈指针指向最后压入堆栈的数据。

(2) 空堆栈：堆栈指针指向下一个将要放入数据的空位置。

(3) 递减堆栈：堆栈向内存地址减小的方向生长。

(4) 递增堆栈：堆栈向内存地址增加的方向生长。

根据堆栈的不同种类，将其寻址方式分为以下4种。

(1) 满递减堆栈(Full Descending,FD)：SP指向最后一个压入的数据(满)；新数据压入时，SP先递减，再存入数据；数据弹出时，先读取数据，SP再递增。

对应指令：存储(压栈)，STMDB(Store Multiple Decrement Before)；加载(弹栈)，LDMIA(Load Multiple Increment After)。

示例：

```
;压栈(存储 R0 - R3 到堆栈)
STMDB SP!, {R0 - R3}            ;SP 先减 4 字节,再存入寄存器 R3 的数据;SP 再减 4 字节,存入
寄存器 R2 的数据;SP 再减 4 字节,存入寄存器 R1 的数据;SP 再减 4 字节,存入寄存器 R0 的数据
;弹栈(从堆栈加载到 R0 - R3)
LDMIA SP!, {R0 - R3}            ;读取寄存器 R0～R3 的数据后,SP 增 16 字节
```

（2）空递减堆栈(Empty Descending，ED)：SP 指向下一个可用的空存储位置；新数据压入时，先存入数据，SP 再递减；数据弹出时，SP 先递增，再读取数据。

对应指令：存储，STMDA（Store Multiple Decrement After)；加载，LDMIB（Load Multiple Increment Before)。

示例：

```
;压栈(存储 R0 - R3 到堆栈)
STMDA SP!, {R0 - R3}
;弹栈(从堆栈加载到 R0 - R3)
LDMIB SP!, {R0 - R3}
```

（3）满递增堆栈(Full Ascending，FA)：SP 指向最后一个压入的数据；新数据压入时，SP 先递增，再存入数据；数据弹出时，先读取数据，SP 再递减。

对应指令后缀：存储，STMIB(Store Multiple Increment Before)；加载，LDMDA(Load Multiple Decrement After)。

示例：

```
;压栈(存储 R0 - R3 到堆栈)
STMIB SP!, {R0 - R3}
;弹栈(从堆栈加载到 R0 - R3)
LDMDA SP!, {R0 - R3}
```

（4）空递增堆栈(Empty Ascending，EA)：SP 指向下一个可用的空存储位置；新数据压入时，先存入数据，SP 再递增；数据弹出时，SP 先递减，再读取数据。

对应指令：存储，STMIA（Store Multiple Increment After)；加载，LDMDB（Load Multiple Decrement Before)。

示例：

```
;压栈(存储 R0 - R3 到堆栈)
STMIA SP!, {R0 - R3}
;弹栈(从堆栈加载到 R0 - R3)
LDMDB SP!, {R0 - R3}
```

9. 相对寻址

相对寻址是以程序计数器(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。子程序调用指令即相对寻址方式。

示例：

```
BL     SUBR1                         ;调用到 SUBR1 子程序
BEQ    LOOP                          ;条件跳转到 LOOP 标号处
        ...
LOOP:MOV R6, #1
        ...
SUBR1: …
```

3.3 ARM 指令集简介

视频讲解

3.3.1 存储器访问指令

加载/存储指令用于在寄存器和存储器之间传送数据。加载指令用于将存储器中的数据传送到寄存器,存储指令用于将寄存器中的数据保存到存储器。ARM 指令中有 3 种基本的内存访问指令。

(1) 单寄存器的 Load/Store 指令:这些指令可以在 ARM 寄存器和存储器之间灵活地进行单数据项数据交换。数据项可以是字节、16 位半字或 32 位字。

(2) 多寄存器的 Load/Store 内存访问指令:这些指令的灵活性比单寄存器传送指令差,但可以进行批量的数据交换。它们用于进程的进入和退出、保存和恢复工作寄存器以及复制存储器中的一块数据。

(3) 单寄存器交换指令:这个指令允许寄存器和存储器中的数值进行交换,在一条指令中有效地完成内存与寄存器之间的数据交换。它们在用户级编程中很少用到。它的主要用途是在多处理器系统中实现信号量(Semaphores)的操作,以保证不会同时访问公用的数据结构。

1. 单寄存器的 Load/Store 指令

这种指令用于把单一的数据传入或者传出一个寄存器。支持的数据类型有字节(8 位)、半字(16 位)和字(32 位)。

1) LDR 指令

格式:

```
LDR {<cond>} Rd, addr
```

功能:将存储器地址 addr 中的内容传输到 Rd 寄存器中。如果 Rd 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,#0x04]        ;将存储器地址为 R0 + 0x04 的字数据读入寄存器 R1
LDRB R0,[R1]             ;将存储器地址为 R1 的字节数据读入寄存器 R0,将 R0 的高 24 位清零
```

2) STR 指令

格式:

```
STR {<cond>} Rd, addr
```

功能:将寄存器 Rd 的内容传输到存储器地址 addr 中。

示例:

```
STR R1,[R0]              ;将寄存器 R1 的数据保存到以 R0 数据为存储器地址中
STRB R0,[R0,#0x08]       ;将寄存器 R0 的字节数据保存到存储器地址 R0 + 0x08 中
```

2. 多寄存器的 Load/Store 内存访问指令

多寄存器的 Load/Store 内存访问指令也叫批量加载/存储指令，它可以实现在一组寄存器和一块连续的内存单元之间传送数据。LDM 用于加载多个寄存器，STM 用于存储多个寄存器。多寄存器的 Load/Store 内存访问指令允许一条指令传送 16 个寄存器的任何子集或所有寄存器。多寄存器的 Load/Store 内存访问指令主要用于现场保护、数据复制和参数传递等。

格式：

```
LDM|STM{<cond>}<addressing_mode>   Rn{!}, registers{^}
```

功能：LDM 和 STM 指令可实现一片连续的存储空间和一组寄存器之间的数据传输。LDM 指令用于加载多个寄存器，STM 指令用于存储多个寄存器。<addressing_mode>共有 8 种模式。

(1) IA：每次数据传送后地址加 4。

(2) IB：每次数据传送前地址加 4。

(3) DA：每次数据传送后地址减 4。

(4) DB：每次数据传送前地址减 4。

(5) FD：满递减堆栈。

(6) ED：空递减堆栈。

(7) FA：满递增堆栈。

(8) EA：空递增堆栈。

其中，寄存器 Rn 为基址寄存器，装有传送数据的初始地址，Rn 不允许为 R15；后缀“!”表示最后的地址写回到 Rn 中；寄存器列表 registers 可包含多于一个寄存器或寄存器范围，用“,”隔开，如{R1,R2,R4-R6}；后缀“^”不允许在用户模式下使用，只能在特权模式下使用，当指令为 LDM 且寄存器列表中有 R15(PC)，则除了完成数据传输外，还将 SPSR 复制到 CPSR。

示例：

```
LDMIA R0,{R2 - R7}          ;将 R0 指向的存储器单元数据保存到 R2 - R7 中,R0 值不更新
```

3. 单寄存器交换指令

数据交换指令是 Load/Store 指令的一种特例，它把存储器单元的内容与寄存器内容交换。

1) 字交换 SWP 指令

SWP 指令用于将内存中的字单元和一个指定寄存器的值相交换。

格式：

```
SWP{<cond>} Rd,Rm,[Rn]
```

功能：寄存器 Rn 内容为某内存单元地址，指令将内存单元[Rn]中的数据读取到目标寄存器 Rd 中，同时将另一个寄存器 Rm 的内容写入该内存单元[Rn]中。当 Rd 和 Rm 为同一个寄存器时，指令交换该寄存器和内存单元的内容。

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互换
```

2）字节交换 SWPB 指令

SWPB 指令用于将内存中的一字节单元和一个指定寄存器的低 8 位相交换。

格式：

```
SWPB{<cond>} Rd,Rm,[Rn]
```

功能：寄存器 Rn 内容为某内存单元地址，指令将内存单元[Rn]中的数据读取到目标寄存器 Rd 中，目标寄存器 Rd 的高 24 位设为 0，同时将另一个寄存器 Rm 的低 8 位内容写入该内存字节单元中。当 Rd 和 Rm 为同一个寄存器时，指令交换该寄存器低 8 位内容和内存字节单元的内容。

示例：

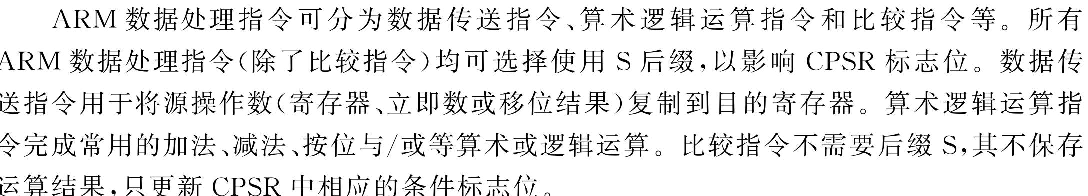
```
SWPB R1,R2,[R0]       ;将 R0 指向的存储单元低字节数据读取到 R1 中(高 24 位清零),
                      ;并将 R2 的内容写入 R0 指向的存储单元中
                      ;(最低字节有效)
```

ARM 存储访问指令表如表 3-2 所示。

表 3-2　ARM 存储访问指令表

助　记　符	说　　明	操　　作	条件码位置
LDR Rd,addressing	加载字数据	Rd←[addressing],addressing 索引	LDR{cond}
LDRB Rd,addressing	加载无符号字节数据	Rd←[addressing],addressing 索引	LDR{cond}B
LDRT Rd. addressing	以用户模式加载字数据	Rd←[addressing],addressing 索引	LDR{cond}T
LDRBT Rd,addressing	以用户模式加载无符号字数据	Rd←[addressing],addressing 索引	L. DR{cond}BT
LDRH Rd,addressing	加载无符号半字数据	Rd←[addressing],addressing 索引	LDR{cond}H
LDRSB Rd,addressing	加载有符号字节数据	Rd←[addressing],addressing 索引	LDR{cond}SB
LDRSH Rd. addressing	加载有符号半字数据	Rd←[addressing]. addressing 索引	LDR{cond}SH
STR Rd,addressing	存储字数据	[addressing]←Rd,addressing 索引	STR{cond}
STRB Rd. addressing	存储字节数据	[addressing]←Rd,addressing 索引	STR{cond}B
STRT Rd,addressing	以用户模式存储字数据	[addressing]←Rd,addressing 索引	STR{cond}T
STRBT Rd,addressing	以用户模式存储字节数据	[addressing]←Rd,addressing 索引	STR{cond}BT
STRH Rd,addressing	存储半字数据	[addressing]←Rd,addressing 索引	STR{cond}H
LDM{mode} Rn(!},reglist	批量寄存器加载	reglist←[Rn…],Rn 回写	LDM{cond}{mode}
STM{mode} Rn{!},reglist	批量寄存器存储	[Rn…]←reglist,Rn 回写	STM{cond}{mode}
SWP Rd,Rm,[Rn]	寄存器和存储器字数据交换	Rd←[Rn],[Rn]←Rm (Rn≠Rd 或 Rm)	SWP{cond}
SWPB Rd,Rm,[Rn]	寄存器和存储器字节数据交换	Rd←[Rn],[Rn]←Rm (Rn≠Rd 或 Rm)	SWPB{cond}

3.3.2 数据处理指令



ARM 数据处理指令可分为数据传送指令、算术逻辑运算指令和比较指令等。所有 ARM 数据处理指令(除了比较指令)均可选择使用 S 后缀,以影响 CPSR 标志位。数据传送指令用于将源操作数(寄存器、立即数或移位结果)复制到目的寄存器。算术逻辑运算指令完成常用的加法、减法、按位与/或等算术或逻辑运算。比较指令不需要后缀 S,其不保存运算结果,只更新 CPSR 中相应的条件标志位。

1. 数据传送指令

1) MOV 指令

格式:

```
MOV{<cond>}{S} Rd, operand2
```

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,#0x02            ;将立即数 0x02 装入 R0
MOV   R0,R1               ;将寄存器 R1 的值传送到 R0
MOVS  R0,R1, LSL #2       ;将寄存器 R1 的值逻辑左移 2 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

格式:

```
MVN{<cond>}{S} Rd, operand2
```

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, #0x0F             ;将立即数 0xF 按位求反后装入 R0,操作后 R0 = 0xFFFFFFF0
MVN R0,R1                 ;将寄存器 R1 的值按位求反后传送到 R0
```

2. 算术逻辑运算指令

1) ADD 指令

格式:

```
ADD{<cond>}{S} Rd, Rn,operand2
```

ADD 指令将 Rn 和 operand2 操作数相加后,结果放入目的寄存器 Rd 中,若有后缀 S,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)
```

2）SUB 指令

格式：

```
SUB{<cond>}{S} Rd, Rn,operand2
```

SUB 指令用于寄存器 Rn 值减去 operand2 操作数，将结果放入目的寄存器 Rd 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)
```

3）RSB 指令

格式：

```
RSB{<cond>}{S} Rd, Rn,operand2
```

RSB 指令称为逆向减法指令，用于把 operand2 操作数减去寄存器 Rn 值，将结果放入目的寄存器 Rd 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
```

4）ADC 指令

格式：

```
ADC{<cond>}{S} Rd, Rn,operand2
```

ADC 指令将 Rn 和 operand2 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 Rd 中，同时根据操作的结果影响标志位。

示例：

```
;用于 64 位数据加法 (R1,R0) = (R1,R0) + (R3,R2)
ADDS R0, R0, R2             ;R0←R0 + R2,更新 CPSR 条件标志位
ADC R1, R1, R3              ;R1←R1 + R3 + C
```

5）SBC 指令

格式：

```
SBC{<cond>}{S} Rd, Rn,operand2
```

SBC 指令用于寄存器 Rn 值减去操作数 operand2，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：

```
;用于 64 位数据减法(R1,R0) = (R1,R0) - (R3,R2)
SUBS R0,R0,R2              ;R0←R0 - R2,更新 CPSR 条件标志位
SBC R1.R1,R3               ;R1←R1 - R3 - (～C),更新 CPSR 条件标志位
```

6）RSC 指令

格式：

```
RSC{<cond>}{S} Rd, Rn,operand2
```

RSC 指令用于操作数 operand2 减去寄存器 Rn 值，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
```

7）AND 指令

格式：

```
AND{<cond>}{S} Rd, Rn,operand2
```

AND 指令实现 Rn 和 operand2 两个操作数的逻辑与操作，将结果放入目的寄存器 Rd 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
```

8）ORR 指令

格式：

```
ORR{<cond>}{S} Rd, Rn,operand2
```

ORR 指令实现 Rn 和 operand2 两个操作数的逻辑或操作，将结果放入目的寄存器 Rd 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
```

9）EOR 指令

格式：

```
EOR{<cond>}{S} Rd, Rn,operand2
```

EOR 指令实现 Rn 和 operand2 两个操作数的逻辑异或操作,将结果放入目的寄存器 Rd 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,#0x0F          ;R0 的低 4 位取反
```

10) BIC 指令

格式:

```
BIC{<cond>}{S} Rd, Rn,operand2
```

BIC 指令用于清除 Rn 的某些位,并将结果存储至目的寄存器中,同时根据操作的结果更新标志位。第二操作数 operand2 为一个 32 位的掩码值,掩码中的置 1 位对应寄存器 Rn 中需要清除的位域位置。

示例:

```
BIC R0,R0,#0x0F          ;R0 的低 4 位清 0,其他位不变
```

3. 比较指令

1) CMP 指令

格式:

```
CMP{<cond>} Rn, operand2
```

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,#0x02           ;将 R1 的值减去 0x02,并根据结果设置 CPSR 的标志位
```

2) CMN 指令

格式:

```
CMN{<cond>} Rn, operand2
```

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 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,#0x02            ;将 R1 的值和立即数 0x02 相加,并根据结果设置 CPSR 的标志位
```

3) TST 指令

格式:

```
TST{<cond>} Rn, operand2
```

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 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 R0,#0x0F          ;检测 R0 的低 4 位是否为 0
```

4) TEQ 指令

格式：

```
TEQ{<cond>} Rn, operand2
```

TEQ 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
```

ARM 数据处理指令如表 3-3 所示。

表 3-3 ARM 数据处理指令

助记符号	说明	操作	条件码位置
MOV Rd,operand2	数据传送指令	Rd←operand2	MOV{cond}{S}
MVN Rd,operand2	数据取反传送指令	Rd←(～operand2)	MVN{cond}{S}
ADD Rd,Rn,operand2	加法运算指令	Rd←Rn+operand2	ADD{cond}{S}
SUB Rd,Rn,operand2	减法运算指令	Rd←Rn-operand2	SUB{cond}{S}
RSB Rd,Rn,operand2	逆向减法指令	Rd←operand2-Rn	RSB{cond}{S}
ADC Rd,Rn,operand2	带进位加法指令	Rd←Rn+operand2+Carry	ADC{cond}{S}
SBC Rd,Rn,operand2	带进位减法指令	Rd←Rn-operand2-(NOT)Carry	SBC{cond}{S}
RSC Rd,Rn,operand2	带进位逆向减法指令	Rd-operand2-Rn-(NOT)Carry	RSC{cond}{S}
AND Rd,Rn,operand2	逻辑与操作指令	Rd←Rn&operand2	AND{cond}{S}
ORR Rd,Rn,operand2	逻辑或操作指令	Rd←Rn|operand2	ORR{cond}{S}
EOR Rd,Rn,operand2	逻辑异或操作指令	Rd←Rn⊕operand2	EOR{cond}(S)
BIC Rd,Rn,operand2	位清除指令	Rd←Rn&(～operand2)	BIC{cond}{S}
CMP Rn,operand2	比较指令	标志 N、Z、C、V←Rn-operand2	CMP{cond}
CMN Rn,operand2	负数比较指令	标志 N、Z、C、V←Rn+operand2	CMN{cond}
TST Rn,operand2	位测试指令	标志 N、Z、C、V←Rn&operand2	TST{cond}
TEQ Rn,operand2	相等测试指令	标志 N、Z、C、V←Rn⊕operand2	TEQ{cond}

3.3.3 乘法指令

ARM 有两类乘法指令：一类为 32 位的乘法指令，即乘法操作后的结果是 32 位；另一类为 64 位乘法指令，即乘法操作后的结果是 64 位。

1. MUL 指令

格式：

```
MUL{<cond>}{S} Rd, Rn, Rm
```

MUL(Multiply)指令的作用是将寄存器 Rm 和 Rm 中的值相乘,乘积结果的低 32 位保存到寄存器 Rd 中。

示例:

```
MUL R1, R2, R3              ;R1 = R2 × R3
MULS R0, R3, R7             ;R0 = R3 × R7,同时根据运算结果更新 CPSR 中的 N 位和 Z 位
```

2. MLA 指令

格式:

```
MLA{<cond>}{S} Rd, Rn, Rm, Rs
```

MLA(Multiply Accumulate)指令的作用是将寄存器 Rn 和 Rm 中的值相乘,再将乘积加上第 3 个寄存器 Rs 的值,结果的低 32 位保存到寄存器 Rd 中。

示例:

```
MLA   R1, R2, R3, R0        ;R1 = R2 × R3 + R0
```

3. UMULL 指令

格式:

```
UMULL{<cond>}{S} RdLo, RdHi,Rm,Rs
```

UMULL(Unsigned Multiply Long)为 64 位无符号乘法指令。它将 Rm 和 Rs 中的值做无符号数相乘,结果的低 32 位保存到寄存器 RdLo 中,高 32 位保存到寄存器 RdHi 中。

示例:

```
UMULL R0, R1,R5,R8          ;R0←R5 × R8[31:0],R1←R5 × R8[63:32],
```

4. UMLAL 指令

格式:

```
UMLAL{<cond>}{S} RdLo, RdHi,Rm,Rs
```

UMLAL(Unsigned Multiply Accumulate Long)为 64 位无符号长乘累加指令。指令将寄存器 Rm 和 Rs 中的值做无符号数相乘,64 位乘积与 RdHi、RdLo 原有值相加,结果的低 32 位保存到寄存器 RdLo 中,高 32 位保存到 RdHi 中。

示例:

```
UMLAL R0,R1,R5,R8           ;R0 = (R5 * R8)[31:0] + R0,R1 = (R5 * R8)[63:32] + R1
```

5. SMULL 指令

格式:

```
SMULL{<cond>}{S} RdLo, RdHi,Rm,Rs
```

SMULL(Signed Multiply Long)为64位有符号长乘法指令。指令将寄存器Rm和Rs中的值做有符号数相乘，结果的低32位保存到RdLo中，高32位保存到RdHi中。

示例：

```
SMULL R0,R1,R5,R8          ;R0 = (R5 * R8)[31:0] + R0,R1 = (R5 * R8)[63:32] + R1
```

6. SMLAL 指令

SMLAL(Signed Multiply Accumulate Long)为64位有符号长乘累加指令。指令将寄存器Rm和Rs中的值做有符号数相乘，64位乘积与寄存器原有值RdHi、RdLo相加，结果的低32位保存到RdLo中，高32位保存到RdHi中。

格式：

```
SMLAL{<cond>}{S} RdLo,RdHi,Rm,Rs
```

示例：

```
SMLAL R0,R1,R5,R8          ;R0 = (R5 * R8)[31:0] + R0,R1 = (R5 * R8)[63:32] + R1
```

3.3.4 ARM 跳转指令

视频讲解

跳转指令用于实现程序流程的跳转，在ARM中有两种方式可以实现程序的跳转，一种是使用跳转指令直接跳转，另一种则是直接向PC寄存器赋值实现跳转。

通过向PC写入跳转地址值，可以实现在4GB的地址空间中的任意跳转，在跳转之前结合使用“MOV LR,PC”等类似指令，可以保存将来的返回地址值，从而实现在4GB连续的线性地址空间的子程序调用。

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的地址空间的跳转，包括以下4条指令。

1) B——跳转指令

格式：

```
B{<cond>} 目标地址
```

B指令是最简单的跳转指令。一旦遇到一个B指令，ARM微处理器将立即跳转到给定的目标地址，从那里继续执行。注意存储在跳转指令中的实际值是相对当前PC值的一个偏移量，而不是一个绝对地址，它的值由汇编器来计算(参考寻址方式中的相对寻址)。它是24位有符号数，左移2位后扩展为26位，表示的有效偏移为26位(前后32MB的地址空间)。

示例：

```
B Label                  ;程序无条件跳转到标号 Label 处执行
CMP R1,#0
BEQ Label                ;当 CPSR 寄存器中的 Z 条件码置位时，程序跳转到标号 Label 处执行
```

2）BL——带返回的跳转指令

格式：

```
BL{<cond>} 目标地址
```

BL是另一个跳转指令，在跳转之前，会在寄存器R14中保存PC的当前内容，因此，可以通过将R14的内容重新加载到PC中来返回到跳转指令之后的那个指令处执行。该指令是实现子程序调用的一个基本但常用的手段。

示例：

```
BL Label            ;当程序无条件跳转到标号 Label 处执行时,同时将当前的 PC 值保存到 R14 中
```

3）BLX——带返回和状态切换的跳转指令

BLX是ARM架构中支持ARM与Thumb状态无缝切换的函数调用指令。它在单一操作中完成以下功能：

- 将返回地址保存到链接寄存器R14(LR)；
- 跳转到目标地址；
- 根据目标地址自动切换处理器状态。

BLX指令有两种格式：第1种格式记作BLX <label>，目标地址在链接时由编译器计算。跳转范围为当前地址±32MB范围内，常用于ARM主程序调用Thumb优化的子程序；第2种格式记作BLX <Rm>，通过寄存器间接跳转，目标地址保存在ARM寄存器R0～R14。

4）BX——带状态切换的跳转指令

格式：

```
BX{条件} 目标地址
```

BX指令跳转到指令中所指定的目标地址，目标地址处的指令既可以是ARM指令，也可以是Thumb指令。

视频讲解

3.3.5 程序状态寄存器处理指令

ARM微处理器支持程序状态寄存器访问指令，用于在程序状态寄存器和通用寄存器之间传送数据，程序状态寄存器访问指令包括MRS和MSR两条指令。

1. MRS——程序状态寄存器到通用寄存器的数据传送指令

格式：

```
MRS{<cond>} Rn,CPSR|SPSR
```

功能：MRS指令用于将程序状态寄存器CPSR或SPSR的内容传送到通用寄存器Rn中。

该指令一般用在以下几种情况：当需要改变程序状态寄存器的内容时，可用MRS将程序状态寄存器的内容读入通用寄存器，修改后再写回程序状态寄存器；当在异常处理或进程切换时，需要保存程序状态寄存器的值，可先用该指令读出程序状态寄存器的值，然后保存。

示例：

```
MRS  R0, CPSR              ;将程序状态寄存器 CPSR 的内容传送到寄存器 R0 中
```

2. MSR——通用寄存器到程序状态寄存器的数据传送指令

格式：

```
MSR{<cond>} CPSR 或 SPSR_field,Rn|#immed_8r
```

功能：MSR 指令用于将操作数(Rn 的内容或 8 位立即数#immed_8r)传送到程序状态寄存器的特定域中。field 用于设置程序状态寄存器中需要操作的位，32 位的程序状态寄存器可分为 4 个域：

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

该指令通常用于恢复或改变程序状态寄存器的内容，在使用时，一般要在 MSR 指令中指明将要操作的域。

示例：

```
MSR CPSR_c,#0xD3          ;CPSR[7:0]=0xD3,切换到管理模式
MSR CPSR_cxsf, R3         ;CPSR←R3
```

3.3.6 协处理器指令

ARM 微处理器支持协处理器操作，协处理器的控制要通过协处理器指令实现。在程序执行的过程中，每个协处理器只执行针对自身的协处理指令，忽略 ARM 微处理器和其他协处理器的指令。

ARM 协处理器指令可完成下面 3 类操作。

(1) ARM 协处理器的数据处理操作。

(2) ARM 微处理器的寄存器和协处理器的寄存器之间传送数据。

(3) ARM 协处理器的寄存器和存储器之间传送数据。

ARM 协处理器指令如表 3-4 所示。

表 3-4 ARM 协处理器指令

助 记 符	说 明	功 能	条件码位置
CDP coproc,opcode1,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	CDP{cond}
LDC{L} coproc,CRd,<地址>	协处理器数据加载指令	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	LDC{cond}{L}
STC{L} coproc,CRd,<地址>	协处理器数据存储指令	将协处理器的寄存器数据写入某一连续的存储单元中	STC{cond}{L}

续表

助　记　符	说　　明	功　　能	条件码位置
MCR coproc,opcodel,Rd,CRn,{,opcode2}	ARM 寄存器到协处理器寄存器的数据传送指令	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	MCR{cond}
MRC coproc,opcodel,Rd,CRn,{,opcode2}	协处理器寄存器到 ARM 寄存器的数据传送指令	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	MRC{cond}

3.3.7　ARM 杂项指令

ARM 微处理器所支持的异常指令有如下两条。

(1) SWI——软件中断指令。

格式：

```
SWI{<cond>} immed24
```

SWI 指令用于产生软件中断，实现从用户模式切换到管理模式，用于用户程序调用操作系统内核服务(系统调用)。操作系统在 SWI 异常处理程序中解析请求并执行对应服务，指令中 24 位的立即数 immd24 指定用户程序调用系统例程的类型，相关参数通过通用寄存器 R0-R3 传递。当指令中 24 位的立即数被设置为特定值时，用户程序调用系统例程类型由通用寄存器 R0 的内容决定。

示例：

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

(2) BKPT——断点中断指令。

格式：

```
BKPT immed16
```

BKPT 指令产生软件断点中断，可用于程序的调试。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
```

视频讲解

3.4　Thumb 指令集

ARM 体系结构除了支持执行效率很高的 32 位 ARM 指令集以外，同时支持 16 位的 Thumb 指令集。Thumb 指令集是 ARM 指令集的一个子集，允许指令编码长度为 16 位。与等价的 32 位代码相比较，Thumb 指令集在保留 32 位代码优势的同时，大大节省了系统的存储空间。

所有的 Thumb 指令都有对应的 ARM 指令，而且 Thumb 的编程模型也对应于 ARM 的编程模型。在应用程序的编写过程中，只要遵循一定的调用规则，Thumb 子程序和 ARM 子程序就可以互相调用。当处理器在执行 ARM 程序段时，称 ARM 微处理器处于 ARM 工作状态，当处理器在执行 Thumb 程序段时，称 ARM 微处理器处于 Thumb 工作状态。

与 ARM 指令集相比较，Thumb 指令集中的数据处理指令的操作数仍然是 32 位，指令地址也为 32 位；但 Thumb 指令集为实现 16 位的指令长度，舍弃了 ARM 指令集的一些特性。大多数的 Thumb 指令是无条件执行的，而几乎所有的 ARM 指令都是有条件执行的；大多数的 Thumb 数据处理指令的目的寄存器与其中一个源寄存器相同。

Thumb 指令集不是一个完整的体系结构。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，且指令的第 2 操作数受到限制；除了跳转指令 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。因此，Thumb 指令集只需要支持通用功能，必要时可以借助完善的 ARM 指令集，如所有异常自动进入 ARM 状态。

在编写 Thumb 指令时，先要使用伪指令 CODE16 声明，而且，在 ARM 指令中要使用 BX 指令跳转到 Thumb 指令，以切换处理器状态。编写 ARM 指令时，则可使用伪指令 CODE32 声明。

Thumb 存储器访问指令如表 3-5 所示。Thumb 指令集的 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。批量寄存器加载和存储指令只有 LDMIA、STMIA 指令，即每次传送先加载/存储数据，然后地址加 4。对堆栈处理只能使用 PUSH 指令和 POP 指令。

表 3-5　Thumb 存储器访问指令

助　记　符	说　　明	操　　作	影响标志
LDR Rd,[Rn,#immed_5×4]	加载字数据	Rd←[Rm,#immed_5×4],Rd 和 Rn 为 R0～R7	无
LDRH Rd,[Rn,#immed_5×2]	加载无符号半字数据	Rd←[Rm,#immed_5×2],Rd 和 Rn 为 R0～R7	无
LDRB Rd,[Rn,#immed_5×1]	加载无符号字节数据	Rd←[Rm,#immed_5×1],Rd 和 Rn 为 R0～R7	无
STR Rd,[Rn,#immed_5×4]	存储字数据	[Rn,#immed_5×4]←Rd,Rd 和 Rn 为 R0～R7	无
STRH Rd,[Rn,#immed_5×2]	存储无符号半字数据	[Rn,#immed_5×2]←Rd,Rd 和 Rn 为 R0～R7	无
STRB Rd,[Rn,#immed_5×1]	存储无符号字节数据	[Rn,#immed_5×1]←Rd,Rd 和 Rn 为 R0～R7	无
LDR Rd,[Rn,Rm]	加载字数据	Rd←[Rn,Rm],Rd、Rn、Rm 为 R0～R7	无
LDRH Rd,[Rn,Rm]	加载无符号半字数据	Rd←[Rn,Rm],Rd、Rn、Rm 为 R0～R7	无
LDRB Rd,[Rn,Rm]	加载无符号字节数据	Rd←[Rn,Rm],Rd、Rn、Rm 为 R0～R7	无
LDRSH Rd,[Rn,Rm]	加载有符号半字数据	Rd←[Rn,Rm],Rd、Rn、Rm 为 R0～R7	无

续表

助　记　符	说　　明	操　　作	影响标志
LDRSB Rd,[Rn,Rm]	加载有符号字节数据	Rd←[Rn,Rm],Rd、Rn、Rm 为 R0～R7	无
STR Rd,[Rn,Rm]	存储字数据	[Rn,Rm]←Rd,Rd、Rn、Rm 为 R0～R7	无
STRH Rd,[Rn,Rm]	存储无符号半字数据	[Rn,Rm]←Rd,Rd、Rn、Rm 为 R0～R7	无
STRB Rd,[Rn,Rm]	存储无符号字节数据	[Rn,Rm]←Rd,Rd、Rn、Rm 为 R0～R7	无
LDR Rd,[PC,#immed_8×4]	基于 PC 加载字数据	Rd←(PC,#immed_8×4),Rd 为 R0～R7	无
LDR Rd,label	基于 PC 加载字数据	Rd←[label],Rd 为 R0～R7	无
LDR Rd,[SP,#immed_8×4]	基于 SP 加载字数据	Rd←[SP,#immed_8×4],Rd 为 R0～R7	无
STR Rd,[SP,#immed_8×4]	基于 SP 存储字数据	[SP,#immed_8×4]←Rd,Rd 为 R0～R7	无
LDMIA Rn,{reglist}	批量寄存器加载	regist←[Rn]	无
STMIA Rn,{reglist}	批量寄存器存储	[Rn…]←reglist	无
PUSH{reglist[,LR]}	寄存器入栈指令	[SP…]←reglist[,LR]	无
POP {reglist[,PC]}	寄存器出栈指令	reglist[,PC]←[SP…]	无

Thumb 数据处理指令如表 3-6 所示。大多数 Thumb 处理指令采用 2 地址格式,数据处理操作比 ARM 状态的更少,访问寄存器 R8～R15 受到一定限制。

Thumb 跳转指令有 B、BL、BLX 和 BX 这 4 条指令;Thumb 杂项指令有 SWI(软件中断指令)和 BKPT(断点中断指令);Thumb 伪指令有 ADR、LDR 和 NOP。

表 3-6　Thumb 数据处理指令

助　记　符	说　　明	操　　作	影 响 标 志
MOV Rd,#expr	数据传送指令	Rd←expr,Rd 为 R0～R7	影响 N、Z
MOV Rd,Rm	数据传送指令	Rd←Rm,Rd 和 Rm 均可为 R0～R15	Rd 和 Rm 均为 R0～R7 时,影响 N、Z、C、V
MVN Rd,Rm	数据非传送指令	Rd←(～Rm),Rd 和 Rm 均为 R0～R7	影响 N、Z
NEG Rd,Rm	数据取负指令	Rd←(－Rm),Rd 和 Rm 均为 R0～R7	影响 N、Z、C、V
ADD Rd,Rn,Rm	加法运算指令	Rd←Rn＋Rm,Rd、Rn 和 Rm 均为 R0～R7	影响 N、Z、C、V
ADD Rd,Rn,#expr3	加法运算指令	Rd←Rn＋expr3,Rd 和 Rn 均为 R0～R7	影响 N、Z、C、V
ADD Rd,#expr8	加法运算指令	Rd←Rd＋expr8,Rd 为 R0～R7	影响 N、Z、C、V
ADD Rd,Rm	加法运算指令	Rd←Rd＋Rm,Rd 和 Rm 均为 R0～R15	Rd 和 Rm 均为 R0～R7 时,影响 N、Z、C、V
ADD Rd,SP/PC,#expr	SP/PC 加法运算指令	Rd←SP＋expr 或 PC＋expr,Rd 为 R0～R7	无
ADD SP,#expr	SP 加法运算指令	SP←SP＋expr	无

续表

助　记　符	说　　明	操　　作	影 响 标 志
SUB Rd,Rn,Rm	减法运算指令	Rd←Rn－Rm,Rd、Rn 和 Rm 均为 R0～R7	影响 N、Z、C、V
SUB Rd,Rn,＃expr3	减法运算指令	Rd←Rn－expr3,Rd 和 Rn 均为 R0～R7	影响 N、Z、C、V
SUB Rd,＃expr8	减法运算指令	RD←Rd－expr8,Rd 为 R0～R7	影响 N、Z、C、V
SUB SP,＃expr	SP 减法运算指令	SP←SP－exp	无
ADC Rd,Rm	带进位加法指令	Rd←Rd＋Rm＋Carry,Rd 和 Rm 为 R0～R7	影响 N、Z、C、V
SBC Rd,Rm	带位减法指令	Rd←Rd－Rm－(NOT)Carry,Rd 和 Rm 为 R0～R7	影响 N、Z、C、V
MUL Rd,Rm	乘法运算指令	Rd←Rd * Rm,Rd 和 Rm 为 R0～R7	影响 N、Z
AND Rd,Rm	逻辑与操作指令	Rd←Rd&. Rm,Rd 和 Rm 为 R0～R7	影响 N、Z
ORR Rd,Rm	逻辑或操作指令	Rd←Rd | Rm,Rd 和 Rm 为 R0～R7	影响 N、Z
EOR Rd,Rm	逻辑异或操作指令	Rd←Rd⊕Rm,Rd 和 Rm 为 R0～R7	影响 N、Z
BIC Rd,Rm	位清除指令	Rd←Rd&(～Rm),Rd 和 Rm 为 R0～R7	影响 N、Z
ASR Rd,Rs	算术右移指令	Rd←Rd 算术右移 Rs 位,Rd 和 Rs 为 R0～R7	影响 N、Z、C
ASR Rd,Rm,＃expr	算术右移指令	Rd←Rm 算术右移 expr 位,Rd 和 Rm 为 R0～R7	影响 N、Z、C
LSL Rd,Rs	逻辑左移指令	Rd←Rd＜＜Rs,Rd 和 Rs 为 R0～R7	影响 N、Z、C
LSL Rd,Rm,＃expr	逻辑左移指令	Rd←Rm＜＜expr,Rd 和 Rm 为 R0～R7	影响 N、Z、C
LSR Rd,Rs	逻辑右移指令	Rd←Rd＞＞Rs,Rd 和 Rs 为 R0～R7	影响 N、Z、C
LSR Rd,Rm,＃expr	逻辑右移指令	Rd←Rm＞＞expr,Rd 和 Rm 为 R0～R7	影响 N、Z、C
ROR Rd,Rs	循环右移指令	Rd←Rm 循环右移 Rs 位,Rd 和 Rs 为 R0～R7	影响 N、Z、C
CMP Rn,Rm	比较指令	状态标志←Rn－Rm,Rn 和 Rm 为 R0～R15	影响 N、Z、C、V
CMP Rn,＃expr	比较指令	状态标志←Rn－expr,Rn 为 R0～R7	影响 N、Z、C、V
CMN Rn,Rm	负数比较指令	状态标志←Rn＋Rm,Rn 和 Rm 为 R0～R7	影响 N、Z、C、V
TST Rn,Rm	位测试指令	状态标志←Rn&Rm,Rn 和 Rm 为 R0～R7	影响 N、Z、C、V

3.5 Thumb-2 技术

Thumb-2 是 ARM 架构的指令集扩展，首次于 ARMv6T2 版本中引入并在 ARMv7 版本中全面优化，旨在提升代码密度与执行效率的平衡。其核心设计为混合 16/32 位指令集，允许在单一操作模式下无缝执行不同长度的指令，彻底消除传统 ARM/Thumb 模式切换的开销。该技术广泛应用于嵌入式系统与移动设备，显著降低存储需求并提升性能。

3.5.1 Thumb-2 指令集的组成

1. 指令组成

Thumb-2 指令集在完全兼容传统 16 位 Thumb 指令的基础上，还扩展了大量 32 位指令，形成了 16 位和 32 位指令混合的指令集架构。

(1) 16 位指令：主要包括对低寄存器(R0～R7)的操作，其指令格式受限，立即数范围小、寻址模式有限。典型指令为

```
MOVS R0, #imm5
ADDS Rd, Rn, Rm
LDR Rd, [Rn, #imm5]
```

16 位指令 Thumb 指令集具有代码密度高，适合控制流代码和操作简单的特点。

(2) 32 位指令：消除了对高寄存器的访问约束，能够访问全部寄存器(R0～R15)，并支持大立即数及复杂的寻址方式。其功能得以强化，具备有符号/无符号除法(SDIV/UDIV)、位字段操作(BFC/BFI)、表分支(TBB/TBH)等指令。在性能方面，该指令通常应用于数学密集型计算及复杂内存访问场景。因篇幅所限，具体指令请读者查询 *ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition*(DDI0406C)手册。

说明：Thumb-2 不提供传统 ARM 的全指令条件执行，条件执行主要通过 IT(If-Then)块实现。

2. 混合执行机制

编译器主导：编译器根据代码逻辑静态选择 16/32 位指令，如对循环密集型代码使用 32 位指令优化性能，对控制流、简单操作代码使用 16 位指令压缩体积。

无缝切换：处理器根据指令编码的 bit[15:11]自动判断当前指令是 16 位还是 32 位，例如，bit[15:11]位为 0b11101、0b11110、0b11111 开头的指令是 32 位。在 Thumb-2 状态下，16 位和 32 位指令混合执行，不需要任何模式切换指令。

示例如下：

```
.thumb
    MOVW R0, #0x1234      ;32 位指令(宽立即数(16 位立即数)的传送)
    ADDS R1, R1, #1       ;16 位指令
```

3.5.2　Thumb-2 核心特性

1. 指令扩展

Thumb-2 通过增加新的 32 位指令，增强了指令集的功能，使其能够执行一些在传统 Thumb 指令集中无法高效执行的操作，如数据加载、内存访问、条件分支等。

2. 混合指令集

Thumb-2 允许使用 16 位和 32 位指令的混合模式，提供了灵活的性能优化选项。处理器根据当前代码路径自动选择适当的指令集，以最优化代码执行。

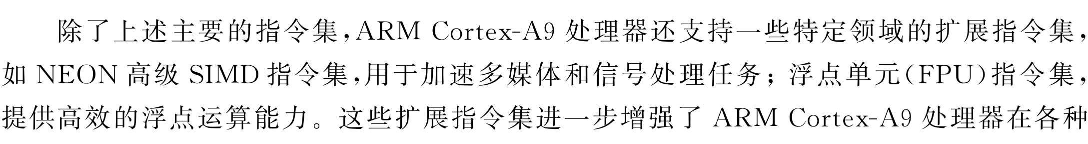
3. 代码压缩

通过使用 16 位 Thumb 指令，程序的代码尺寸显著减小。在代码压缩方面，Thumb-2 比 ARM 指令集具有明显的优势，尤其是在资源受限的嵌入式系统中。

4. 无须频繁切换模式

在早期的 ARM 架构中，切换到 Thumb 模式需要从 ARM 模式进行，并且这种切换会带来一定的性能开销。Thumb-2 消除了这种限制，允许在同一个程序中平滑地混合使用 16 位和 32 位指令，而无须显式切换模式。这不仅简化了编程模型，还减少了切换指令集所带来的性能损失。

3.6　其他扩展指令集



除了上述主要的指令集，ARM Cortex-A9 处理器还支持一些特定领域的扩展指令集，如 NEON 高级 SIMD 指令集，用于加速多媒体和信号处理任务；浮点单元(FPU)指令集，提供高效的浮点运算能力。这些扩展指令集进一步增强了 ARM Cortex-A9 处理器在各种应用中的性能和功能，使其能够更好地满足现代嵌入式系统和移动设备的需求。

3.7　ARM 汇编程序设计

汇编语言，作为一种低级的编程语言，与机器语言指令直接对应，是计算机能够直接执行的指令集的符号化表达形式。本节内容将重点介绍 GNU 环境下的 ARM 伪指令集合。伪指令集合是为了辅助编译器而设计的，不同的编译器拥有各自的伪指令集合。例如，ARM C 语言编译器具有特定的伪指令集合，而 GNU 同样具备一套自己的伪指令集合。鉴于 GNU 的开源特性和其广泛的使用，本节将对 GNU 伪指令集合进行详细讲解。

3.7.1　GNU ARM 汇编器的伪操作

所谓伪操作就是没有对应的机器码的指令，它是用于告诉汇编程序如何进行汇编的指令，它既不控制机器的操作也不被汇编成机器代码，只能为汇编程序所识别并指导汇编程序如何进行。所有汇编伪操作的名称都以“.”开始，余下的是字母，通常使用小写字母。

伪操作按照不同的功能可以分为符号定义伪操作、数据定义伪操作、汇编控制伪操作和其他伪操作。

1. 符号定义伪操作

1）全局标号定义伪操作.global 和.globl

伪操作.global 用于声明一个 ARM 程序中的全局变量，使得被声明的符号对连接器(ld)可见，变为整个工程都可使用的全局变量。两种拼写(“.globl”和“.global”)都可以，两种形式是为了兼容其他的汇编器。以上两条伪操作用于定义全局变量，因此在整个程序范围内变量名必须唯一。

格式：

```
.global symbol
.globl  symbol
```

其中，symbol 为全局变量。

示例：

```
.global _start                ;定义了一个全局的符号_start
```

2）局部标号定义伪操作.local

伪操作.local 用于声明一个 ARM 程序中的局部变量，这样它对外部就是不可见的，作用域在本文件内。

格式：

```
.local symbol
```

其中，symbol 为局部变量。

示例：

```
.local loop
```

3）变量赋值伪操作.set

伪操作.set 用于给一个全局变量或局部变量赋值。

格式：

```
symbol .set expr              ;为符号 symbol 赋值 expr
```

其中，expr 为表达式，可以是数值、寄存器值或算术运算结果等。该伪操作在汇编过程中将 expr 的值计算出来，并将其赋给 symbol。使用.set 伪操作可以方便地在汇编代码中进行常量定义和变量赋值，提高代码的可读性和可维护性。

示例：

```
start .set  0x40
start .set  0x50
mov r1, #start                ;寄存器 r1 的值为 0x50
```

4）宏替换伪操作.equ

伪操作.equ 用于给一个全局变量或局部变量赋值。

格式：

```
symbol .equ   expr                ;为符号 symbol 赋值 expr
```

示例：

```
start .equ 0x40
start .equ 0x50
mov r1, #start                    ;寄存器 r1 的值为 0x50
```

2. 数据定义伪操作

数据定义伪操作一般用于为特定的数据分配存储单元，同时对该内存单元中的数据进行初始化。常见的数据定义伪操作有.byte、.short、.word、.long、.quad、.float、.space、.skip、.string、.asciz、.ascii 和.rept。数据定义伪操作如下。

1）.byte

伪操作.byte 的功能是在存储器中分配 1 字节的内存单元，用指定的数据对该存储单元进行初始化。

格式：

```
label: .byte expr
```

其中，label 为程序标号；expr 可以是−128～255 的数字，也可以是字符。

示例：

```
a: .byte   #2                    ;类似于 C 语言中的 char   a = 2
```

2）.short

伪操作.short 的功能是在存储器中分配 2 字节的内存单元，并用指定的数据对该存储单元进行初始化。

格式：

```
label: .short expr
```

其中，label 为程序标号；expr 可以是−32 768～65 535($2^{16}-1$)的数字。

示例：

```
a: .short 0x1111                 ;类似于 C 语言中的 short a = 0x1111
```

3）.word

伪操作.word 的功能是在存储器中分配 4 字节的内存单元，并用指定的数据对该存储单元进行初始化。

格式：

```
label: .word expr
```

其中,label 为程序标号;expr 可以是$-2^{16}\sim2^{32}-1$的数字。

示例:

```
a: .word 0x12345678              ;类似于 C 语言中的 int a = 0x12345678
```

4) .long

.long 的功能等价于.word。

5) .quad

伪操作.quad 的功能是在存储器中分配 8 字节的内存单元,并用指定的数据对该存储单元进行初始化。

格式:

```
label: .quad expr
```

其中,label 为程序标号;expr 可以是$-2^{32}\sim2^{64}-1$的数字。

示例:

```
a: .quad 0x123456789abcd         ;类似于 C 语言中的 long a = 0x123456789abcd
```

6) .float

伪操作.float 的功能是在存储器中分配 4 字节的内存单元,并用指定的浮点数据对该存储单元进行初始化。

格式:

```
label: .float expr
```

其中,label 是程序标号;expr 可以是 4 字节之内的浮点数值。

示例:

```
a: .float 1.11                   ;类似于 C 语言中的 float a = 1.11
```

7) .space

.space 伪操作用于分配一片连续的存储区域并将其初始化为指定的值。如果后面的填充值省略不写则默认在后面填充为 0。

格式:

```
label: .space size,expr
```

其中,label 为程序标号;expr 为要将该内存区域初始化成的值。

示例:

```
a: .space 8,0x1          ;在当前的内存空间中申请了 8 字节的空间,并全部初始化为 1
```

8) .skip

类似于.space。

9）.string/.ascii/.asciz

伪操作.string/.ascii/.asciz定义一个字符串。

（1）.string伪指令。

.string用于在汇编程序中定义一个字符串。字符串以空字符(\0)结尾，即自动添加终止符。

格式：

```
label: .string "str"
```

其中，label为程序标号；str为一个字符串。

示例：

```
string_str: .string "CString"     ;string_str 字符串长度为 8 字节,包含空字符\0
```

（2）.ascii伪指令。

.ascii用于定义一个字符串，但不自动添加终止符，需要手动添加空字符以确保字符串正确终止。

格式：

```
label: .ascii "str"
```

其中，label为程序标号；str为一个字符串。

示例：

```
ascii_str: .ascii "RawData"       ;ascii_str 字符串为 7 字节,无终止符
```

（3）.asciz伪指令。

.asciz代表了.ascii与zero的结合，它用于设定一个字符串，并在末尾自动追加空字符终止符。这样可以确保字符串以\0结尾，方便C语言等依赖终止符的编程语言进行处理。

格式：

```
label: .asciz "str"
```

其中，label为程序标号；str为一个字符串。

示例：

```
asciz_str: .asciz "CString"       ;asciz_str 字符串为 8 字节,有空字符(\0)
```

10）.rept

伪操作.rept的功能是重复执行后面的指令，以.rept开始，并以.endr结束。

格式：

```
.rept count
...
.endr
```

其中,count 为程序后面要执行的次数。

示例:

```
.rept 3
mov R0, #0x01
.endr
```

展开后的代码如下:

```
mov R0, #0x01
mov R0, #0x01
mov R0, #0x01
```

3. 汇编控制伪操作

汇编控制伪操作在 ARM 汇编语言中扮演着至关重要的角色,它们用于控制汇编过程,实现条件汇编、宏定义、重复汇编等功能,为程序员提供了极大的灵活性和便利性。

1) 条件汇编伪操作

通过条件汇编伪操作,程序员能够基于特定条件选择性地编译代码段。这在生成适应不同配置或环境的程序版本时显得尤为实用。典型的条件汇编伪操作包括.if/.else/.endif。

.if:开始一个条件汇编块,如果条件为真,则汇编其后的代码。

.else:与.if 配合使用,如果.if 后面的逻辑表达式为假,则汇编.else 后的代码。

.endif:结束一个条件汇编块。

通过这些伪操作,程序员可以根据需要灵活地包含或排除某段代码,从而生成不同版本的程序。

指令的语法格式有两种。

(1) 格式一。

```
.if logical - expression
...
.else
...
.endif
```

其中,logical-expression 为用于决定指令编译流程的逻辑表达式。

(2) 格式二。

```
.if logical - expression1
...
.elseif logical - expression2
...
.endif
```

其中,logical-expression1 和 logical-expression2 为用于决定指令编译流程的逻辑表达式。

说明:使用.elseif 避免了 if-else 形式的嵌套,使得程序结构更加清晰、易读。

2) 宏定义伪操作

宏定义中常见的伪操作包括.macro、.endm 和.exit。通过使用伪操作.macro 和.endm,可以将代码片段整合成一个单元,便于在程序中多次调用。而.exitm 伪操作则用于终止当前

宏指令的执行。

宏操作的运用和子程序类似，子程序支持模块化编程、节省内存并提升执行效率。然而，调用子程序时需保存现场，这会增加系统负担。因此，在代码较短且参数较多的情况下，宏操作是子程序的良好替代品。

宏定义体是指包含在.macro和.endm之间的指令集。在宏定义体的首行，应明确宏的原型（包括宏名称和所需参数）。之后，便能在汇编程序中通过宏名称来执行该指令集。编译时，汇编器会将宏调用展开，用定义体中的指令替换程序中的宏调用，并将实际参数值传递给定义体内的形式参数。

在程序中，程序员可以通过宏的名称和参数来调用宏，汇编器会将宏展开为实际的代码。

4. 其他伪操作

GNU 汇编中还有一些其他的伪操作，在汇编程序中经常会被使用。

(1) .align 用于使程序当前位置满足一定的对齐方式。

(2) .section 用于定义一个段的伪操作。

(3) .data 用于定义一个数据段。

(4) .text 用于定义一个代码段。

(5) .include 用于包含一个头文件。

(6) .arm 定义代码使用 ARM 指令集编译。

(7) .code 32 的作用同.arm。

(8) .code 16 的作用同.thumb。

(9) .thumb 定义代码使用 Thumb 指令集编译。

(10) .extern 用于声明一个符号是引用的外部的符号，常被省略。

(11) .weak 用于声明一个符号是弱符号，如果这个符号没有定义，编译就忽略，而不会报错。

(12) .end 代表汇编程序的结束。

3.7.2 伪指令

ARM 伪指令不是 ARM 指令集中的指令，只是为了编程，方便编译器定义的指令，使用时可以像其他 ARM 指令一样，但在编译时这些指令将被等效的 ARM 指令代替。ARM 伪指令有 ADR、ADRL、LDR、NOP 这 4 条。

(1) ADR 伪指令将基于 PC 相对偏移的地址值加载到寄存器中。

格式：

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

其中，register 为加载的目标寄存器；expr 为地址表达式。当地址值是非字对齐地址时，取值范围为−255～255 字节；当地址是字对齐地址时，取值范围为−1020～1020 字节。

示例：

```
Start: MOV R0,#10
ADR R4,Start                ;将 Start 的地址加载到 R4
```

(2) ADRL 伪指令将程序相对偏移或寄存器相对偏移地址加载到寄存器中。在汇编编译源程序时,ADRL 伪指令被编译器替换成 2 条合适的指令。若不能用 2 条指令实现 ADRL 伪指令功能,则产生错误,编译失败。

格式:

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

其中,register 为加载的目标寄存器;expr 为地址表达式。当地址值是非字对齐地址时,取值范围为−64～64KB;当地址值是字对齐地址时,取值范围为−256～256KB;当地址值是 16 字节对齐时,其取值范围更大。在 32 位的 Thumb-2 指令中,地址取值范围达到−1～1MB。

示例:

```
Start: MOV R0,#10
ADRL R4,Start + 60000
```

以上 ADRL R4,Start+60000 在编译后会被替换为以下两条指令。

```
ADD R4,PC,#59392
ADD R4,R4,#596
```

(3) LDR 伪指令用于加载 32 位的立即数或一个地址值到指定寄存器。在汇编编译源程序时,LDR 伪指令被编译器替换成一条合适的指令。若加载的常数未超出 MOV 或 MVN 的范围,则使用 MOV 或 MVN 指令代替该 LDR 伪指令;否则,汇编器将产生文字常量放入文字池,并使用一条程序相对偏移的 LDR 伪指令从文字池读出常量。

格式:

```
LDR{cond}register,[expr|label_expr]
```

其中,register 为加载的目标寄存器;expr 为 32 位立即数;label_expr 为程序相对偏移或外部表达式。

示例:

```
LDR R3, = 0xff0                //编译后生成 MOV R3,#0xff0
```

(4) NOP 为空操作伪指令。在汇编时将会被代替成 ARM 中的空操作,比如“MOV R0,R0”指令等。NOP 可用于延时操作。

格式:

```
NOP
```

3.7.3 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]
```

（1）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，不能既有大写字母也有小写字母。

（2）所有标号在一行的最左侧书写，后面添加“:”。

（3）注释内容从“;”开头到本行结束。

（4）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。

（5）变量的设置，常量的定义，其标识符必须在一行顶格书写，以便识别。

3.7.4 汇编语言的程序结构

1. 汇编语言的程序格式

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、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。

在 ARM（Thumb）汇编语言程序中可以使用.section 来进行分段，其中每一个段用段名或者文件结尾为结束，这些段使用默认的标志，如 a 为允许段，w 为可写段，x 为执行段。

在一个段中，可以用如下伪操作定义下列的子段：

.text：代码段，包含程序的指令代码；

.data：已初始化的数据段（全局/静态变量），包括固定的数据，如常量、字符串；

.bss：未初始化数据段，包含未初始化的变量、数组等；

.sdata：已初始化的小数据变量，如字符、短整型；

.sbss：包含未初始化的小数据变量，如字符、短整型。

当程序较长时，可以分割为多个代码段和数据段，多个段在程序编译链接时最终形成一个可执行的映像文件。

2. 汇编语言的子程序调用

在 ARM 汇编语言程序中，子程序的调用一般是通过 BL 指令来实现的。在程序中，使用指令“BL 子程序名”即可完成子程序的调用。

该指令在执行时完成如下操作：将子程序的返回地址存放在连接寄存器 LR 中，同时将程序计数器 PC 指向子程序的入口点。当子程序执行完毕需要返回调用处时，只需要将存放在 LR 中的返回地址重新复制给程序计数器 PC 即可。

以下是使用 BL 指令调用子程序的汇编语言源程序的基本结构：

```
.text
.global_start
_start:
   ldr r0, = 0x3ff5000
   ldr r1, = 0xff
   str r1,[r0]
   ldr r0, = 0x3ff5008
   ldr r1, = 0x01
   str r1,[r0]
```

```
    bl print_text
...
print text:
...
    mov  pc, lr
```

3. 汇编语言程序设计举例

通过组合使用条件执行和条件标志设置,可简单地实现分支语句,不需要任何分支指令。改善性能,减小代码尺寸,提高代码密度和执行速度。

下面是一段C语言程序,该程序实现了著名的Euclid最大公约数算法。

```
int gcd(int a,int b)
{
    while(a!= b)
    {
        if(a > b)
            a = a - b;
        else
            b = b - a;
    }
return a;
}
```

用ARM汇编语言重写C语言例子:

```
Code1:
gcd:
    CMP  r0,r1
    BEQ  end
    BLT  Less
    SUB  r0,r0,r1
    B    gcd
Less:
    SUB  r1,r1,r0
    B    gcd
```

充分地利用条件执行修改上面的例子:

```
Code2:
gcd:
    CMP  r0,r1
    SUBGT    r0,r0,r1
    SUBLT    r1,r1,r0
    BNE  gcd
```

从上述两段代码可以看出,Code1仅使用了分支指令,Code2充分利用了ARM指令条件执行的特点,仅使用了4条指令就完成了全部算法。这对提高程序的代码密度和执行速度十分有帮助。

由此可见,分支指令十分影响处理器的速度。每次执行分支指令,处理器都会清空流水线,重新装载指令。

3.8　ATPCS 介绍

ATPCS(ARM-Thumb Procedure Call Standard)是 ARM 程序和 Thumb 程序中子程序调用的基本规则,旨在确保不同编译器和汇编器生成的代码能够相互兼容,从而方便 C 语言与 ARM 汇编语言之间的混合编程。具体来说,ATPCS 规定了以下与混合编程相关的规则。

3.8.1　寄存器的使用规则

寄存器的使用必须满足下列规则。

(1) 子程序间通过寄存器 R0～R3 来传递参数,记为 A1～A4。被调用的子程序在返回前无须恢复寄存器 R0～R3 的内容。

(2) 子程序中使用寄存器 R4～R11 来保存局部变量,记为 V1～V8。如果在子程序中使用了寄存器 V1～V8 中的某些寄存器,则子程序进入时必须保存这些寄存器的值,在返回前必须恢复这些寄存器的值;对于子程序中没有用到的寄存器,则不必进行这些操作。在 Thumb 程序中,通常只能使用寄存器 R4～R7 来保存局部变量。

(3) 寄存器 R12 用作子程序间的 scratch 寄存器,常用于子程序间的连接代码段中,记作 IP。

(4) 寄存器 R13 用作数据栈指针,记作 SP。在子程序中寄存器 R13 不能用作其他用途。寄存器 SP 在进入子程序时的值和退出子程序时的值必须相等。

(5) 寄存器 R14 称为链接寄存器,记作 LR。它用于保存子程序的返回地址。如果在子程序中保存了返回地址,则寄存器 R14 可作其他用途。

(6) R15 是程序计数器,记作 PC。不能作其他用途。

3.8.2　数据栈的使用规则

ATPCS 规定数据栈为满递减类型,并且对数据栈的操作是 8 字节对齐的。异常中断的处理程序可使用中断程序的数据栈。ATPCS 规定数据栈特性及数据栈相关概念如下。

1. 数据栈类型

ATPCS 明确规定,数据栈采用满递减(Full Descending,FD)类型。在此类型栈中,数据从高地址向低地址压入,栈指针(Stack Pointer)随每次数据压入而递减,指向当前栈顶位置。

2. 数据栈对齐要求

为确保数据访问的效率和程序的稳定性,ATPCS 要求数据栈的操作必须保持 8 字节对齐。这种对齐方式有助于优化内存访问,减少因未对齐访问而产生的性能损耗。

3. 异常中断处理程序的栈使用

在异常中断发生时,处理程序可以使用专门的中断程序数据栈。这一栈区域用于保存和恢复中断前的上下文信息,确保中断处理完毕后能够正确返回到原程序执行点。

4. 数据栈指针(Stack Pointer)

数据栈指针是指向栈顶数据的内存地址的指针。在 FD 型栈中,栈指针随数据的压入

而递减,始终指向最新的栈顶位置。

5. 数据栈基地址(Stack Base)

数据栈的基地址是栈的最高地址,即栈的起始位置。在 ATPCS 中,由于栈是满递减的,因此最早入栈的数据将占用基地址之后的内存单元(若按字节计址,则为基地址减一的位置)。

6. 数据栈界限(Stack Limit)

数据栈界限定义了栈中可使用的最低内存单元地址。当栈指针递减至接近或等于栈界限时,表示栈空间即将耗尽,此时可能触发栈溢出错误,需进行相应的错误处理。

7. 数据栈中的数据帧

在数据栈中,每个子程序调用都会分配一个独立的数据帧。数据帧用于保存子程序执行过程中所需的寄存器和局部变量,确保子程序的独立性和数据的隔离性。子程序执行完毕后,其数据帧将被弹出栈,释放相应的内存空间,以便后续子程序使用。

综上所述,ATPCS 对数据栈的详细规定确保了不同编译器和汇编器生成的代码在栈使用上的兼容性。了解并遵循这些规定,对于开发者正确管理和使用数据栈,提高程序的稳定性和性能具有重要意义。在 C 语言与 ARM 汇编语言的混合编程中,更应注重数据栈的正确管理和使用。

3.8.3 参数的传递规则

1. 参数个数固定的子程序参数传递规则

对于参数个数固定的子程序,如果系统包含浮点运算的硬件部件,浮点参数将按照下面规则传递。

(1) 各个浮点参数按顺序处理。

(2) 为每个浮点参数分配 FP 寄存器。方法是:满足该浮点参数需要的且编号最小的一组连续的 FP 寄存器。

(3) 第一个整数参数,通过寄存器 R0～R3 来传递。其他参数通过数据栈传递。

2. 参数个数可变的子程序参数传递规则

(1) 当参数不超过 4 个,使用 R0～R3 传递参数,当参数超过 4 个时,使用数据栈来传递参数。

(2) 在参数传递时,所有参数看作是存放在连续的内存字单元中的字数据。然后,依次将各字数据传递到寄存器 R0～R3 中,如果参数多于 4 个,将剩余的字数据传送到数据栈中,入栈的顺序与参数顺序相反,即最后一个字数据先入栈。

3. 子程序结果返回规则

(1) 结果为一个 32 位整数,可通过寄存器 R0 返回。

(2) 结果为一个 64 位整数,可通过寄存器 R0,R1 返回,以此类推。

(3) 结果为一个浮点数时,可通过运算部件的寄存器 F0、D0 或者 S0 来返回。

(4) 结果为复合型的浮点数(如复数)时,可通过寄存器 F0～Fn 或者 D0～Dn 来返回。

(5) 对于位数更多的结果,需通过内存来传递,如通过数据栈来传递。

3.9 C 语言与汇编语言的混合编程

在开发实践中，通常会将 C 语言和汇编语言结合起来进行混合编程。在 C 语言代码中加入汇编代码，主要有两种方式：内联汇编和嵌入式汇编。内联汇编是在 C 语言代码里直接插入汇编指令，这通常借助关键字 asm 来实现。嵌入式汇编是将汇编代码作为一个单独的模块，通过链接外部文件或使用特定的语法规则，将其嵌入 C 语言程序中。

通过混合编程的方式，可以在 C 语言程序中执行一些 C 语言本身无法实现的功能。例如，在 C 语言程序中，若要操作程序状态寄存器，就必须用到内联汇编或者嵌入式汇编。

3.9.1 GNU 内联汇编

1）格式。

GNU 内联汇编语言的格式如下：

```
asm volatile(
            "asm code"              //汇编代码部分
            :output                 //指定汇编结果写入C语言变量的位置
            :input                  //指定汇编指令的输入来源
            :changed                //告知编译器哪些资源被修改
            );
```

说明：

（1）内联汇编语言必须以“;”结尾，内联汇编不管有多长对 C 语言都只是一条语句；

（2）volatile：告诉编译器不要优化内联汇编，如果想优化可以不加；

（3）如果后面部分没有内容“：”可以省略，前面或中间的不能省略；

（4）没有 asm code 也不可以省略双引号，没有 changed 必须省略“:”。

2）汇编代码“asm code”

汇编代码必须放在一个字符串内，但是字符串中间不能直接按 Enter 键换行，可以写成多个字符串，只要字符串之间不加任何符号，编译完后就会变成一个字符串。

示例：

```
"mov r0,r0\n\t"
"mov r1,r1\n\t"
```

说明：\n 和\t 是汇编代码中的格式控制符，\n 保证指令独立成行，\t 实现代码对齐。它们在 GNU 内联汇编中广泛使用，使代码结构清晰且兼容性强。

字符串内除了能放指令，也可以放一些标签、变量、循环、宏等，还可以把内联汇编放在 C 语言函数的外面，用内联汇编定义函数、变量、段等，与直接写汇编文件一样。

3）使用占位符

占位符是内联汇编的核心机制，通过%数字语法连接汇编和 C 语言代码。

占位符%+数字的编号规则（编号顺序）如下：

%0 为第一个输出约束；

%1 为第二个输出约束；

%2 为第三个输出约束；

⋮

%N－1 为第 N 个输出约束；

%N 为第一个输入约束；

%N＋1 为第二个输入约束。

示例：

```
int a = 100,b = 200;
int result;
asm  volatile (
             "mov  % 0, % 3\n\t"
             "ldr  r0, % 1\n\t"
             "ldr  r1, % 2\n\t"
             "str  r0, % 2\n\t"
             "str  r1, % 1\n\t"
             :" = r"(result)," + m"(a)," + m"(b)      //output1:result 是 % 0,
                                                      //output2:a 是 % 1,output3:b 是 % 2
             :"i"(123)                                //input:i = 123 是 % 3
             );
```

4）输出操作数列表(可选)(ASM→C)

在将汇编代码转换为 C 语言代码的过程中，输出操作数列表是一个可选的步骤。

格式：

```
:"constraint"(variable)
```

其中，constraint 用于明确指定变量 variable 在汇编代码中的存放位置和访问方式，具体如下：

r：表示该变量可以使用任何可用的通用寄存器进行存储。

m：表示该变量应当使用其内存地址进行存储。

修饰符的输出规则如下：

＋：表示输出操作数既可以被读取也可以被写入。

＝：表示输出操作数仅限于被写入，不允许读取。

&：表示输出操作数不得占用输入操作数已经占用的寄存器，此修饰符仅能与"＋&"或"＝&"组合使用。

示例：

```
int a = 10,result;
asm volatile(
           "add  % 0, % 0, #1\n\t"              //将 % 0 的值加 1
           :" + r"(a)                           //输出:a 是可读可写的
           );                                   //执行后 a 的值变为 11
```

在这个例子中，使用了“＋r”修饰符，这表示变量 a 在汇编代码中既可读也可写，并且使用了一个通用寄存器来存储其值。

示例：

```
int num;
 asm   volatile (
//mov     r3,#123                          //编译器自动加的指令
"mov    %0,%1\n\t"      //mov     r3,r3       输入和输出使用相同的寄存器
:"=r"(num)
:"r"(123)
);
int num;
 asm      volatile_(
//mov r3,#123
"mov  %0,%1\n\t"        //mov   r2,r3         加了 & 后输入和输出的寄存器不一样了
:"=&r"(num)             //mov    r3,r2,       编译器自动加的指令
:"r"(123)
);
```

在上述例子中，说明了在变量 num 的约束中添加了“&”修饰符后，输入和输出使用的寄存器不一样了。

5）输入操作数列表（可选）(C→ASM)

格式：

```
:"constraint"(variable/immediate)
```

其中，constraint 用于明确指定变量 variable/立即数 immediate 的存放位置：

r：使用任何可用的通用寄存器（变量和立即数都可以）。

m：使用变量的内存地址（不能用立即数）。

i：使用立即数（不能用变量）。

示例：

```
int a = 10, b = 20, result;
asm volatile (
            "add %0, %1, %2\n\t"        //%0 = %1 + %2,即完成 result = a + b
            "=r"(result)                //输出:result 使用寄存器
            :"r"( a), "r"(b)            //输入:a 和 b 使用寄存器
            );                          //执行后 result = 30
```

3.9.2 汇编程序与C语言程序的相互调用

汇编程序、C 语言程序相互调用时，要特别注意遵守相应的 ATPCS 规则。下面一些例子具体说明了在这些混合调用中应注意遵守的 ATPCS 规则。

1. 从C语言程序中调用汇编语言

下面的程序示例显示了如何在 C 语言程序中调用汇编语言子程序，该段代码实现了将一个字符串复制到另一个字符串。

```
#include <stdio.h>
extern void strcopy(char *d,const char *s);
```

```
int main()
{
    const char * srcstr = "First string - source";
    char dststr[ ] = "Second string - destination";
    /* 下面将 dststr 作为数组进行操作 */
    printf("Before copying:\n");
    printf(" %s\n, %s\n",srcstr, dststr);
    strcopy(dststr,srcstr);
    printf("After copying:\n"):
    printf(" %s\n, %s\n",srcstr,dststr);
    return(0);
}
```

调用的汇编程序：

```
.global strcopy
strcopy:                        ;R0 指向目的字符串,R1 指向源字符串
    LDRB R2, [R1], #1           ;加载字节并更新源字符串指针地址
    STRB R2. [R0], #1           ;存储字节并更新目的字符串指针地址
    CMP R2, #0                  ;判断是否为字符串结尾
    BNE strcopy                 ;如果不是,程序跳转到 strcopy 继续复制
    MOV pc, Ir                  ;程序返回
```

2. 汇编程序调用 C 语言程序

通过 BL 指令调用 C 语言函数，参数通过寄存器 R0～R3 传递，超出部分用堆栈，返回前恢复寄存器状态，遵循 AAPCS/ATPCS 标准。

```
int g (int a, int b, int c. int d, int e)
{
    return a + b + c + d + e;
}
```

下面的程序段显示了汇编语言调用。假设程序进入 f 时，R0 中的值为 i。

```
.text
.global   _start
_start:
    STR 1r,[sp, # - 4]!          //保存返回地址 lr
    ADD R1, R0, R0               //计算 2*i(第 2 个参数)
    ADD R2, R1, R0               //计算 3*i(第 3 个参数)
    ADD R3, R1, R2               //计算 5*i
    STR R3, [sp, # - 4]!         //第 5 个参数通过堆栈传递
    ADD R3, R1, R1               //计算 4*i(第 4 个参数)
    BL   g                       //调用 C 程序
    ADD sp, sp, #4               //从堆栈中删除第 5 个参数
.end
```

3.10 本章小结

本章详细介绍了 ARM 微处理器的指令系统，包括指令的基本格式、条件码、寻址方式、ARM 指令集、Thumb 指令集、Thumb-2 技术、其他扩展指令集、汇编语言程序设计、

ATPCS 介绍以及 C 语言与汇编语言的混合编程。本章通过具体的指令格式、案例和应用场景，旨在帮助读者理解 ARM 微处理器指令系统，提升读者在 ARM 编程方面的实践能力。

习题

1. 请描述 ARM 指令的基本格式包括哪些部分，并解释每部分的作用。

2. 列出 ARM 指令集中常用的条件码，并给出一个使用条件码的实际指令例子，说明其在程序中的作用。

3. 解释 ARM 微处理器中的立即寻址、寄存器寻址、寄存器间接寻址和基址变址寻址等几种寻址方式，并举例说明每种寻址方式的应用场景。

4. 给出一条 ARM 指令（如 MOV R1，#0x12），要求识别该指令属于 ARM 指令集中的哪一类指令（如数据处理指令、加载/存储指令等），并解释其功能。

5. 简述 Thumb 指令集相对于 ARM 指令集的主要特点，以及它在嵌入式系统中的应用优势。

6. 解释 Thumb-2 技术是如何结合 16 位和 32 位指令来优化代码密度和性能的，并给出一个 Thumb-2 指令的例子。

7. 列举一个 ARM 架构中的扩展指令集（如 NEON、VFP 等），并简述它们的主要用途和在特定应用领域（如图形处理、浮点运算）中的重要性。

8. 编写一个简单的 ARM 汇编语言程序，实现两个寄存器中值的相加，并将结果存储到第三个寄存器中的功能。

9. 解释 ATPCS 中关于函数调用约定（如参数传递、返回值处理、堆栈管理）的基本规则，并分析其对编写可移植 ARM 代码的意义。