CHAPTER 3

MSP430 单片机指令系统　第 3 章

3.1　指令系统概述

指令是使计算机执行某种操作的命令。一条指令通常由操作码和操作数构成。其中操作码指明了该指令要完成的操作类型或性质，如取数、加法等。操作数指明操作对象的内容或所在的存储单元地址。一般来说，操作码部分比较简单，操作数部分则较复杂。

单片机所能执行的全部指令的集合称为指令系统，它描述了单片机内全部的控制信息和“逻辑判断”能力。因此，指令系统是 CPU 的重要性能指标，也是进行 CPU 内部电路设计的基础。不同种类单片机的指令系统包含的指令种类和数目也不同。此外，指令系统有复杂指令集和精简指令集之分。例如，MCS-51 单片机采用的是复杂指令集，具有 111 条指令；PIC16F87x 系列单片机采用的是精简指令集，有 35 条指令。MSP430 单片机的指令系统是 16 位精简指令集系统，共由 27 条指令组成。

3.1.1　指令的表示方法

指令有两种表示方法：一种是机器语言表示法，该方法用一串二进制数码表示一条指令，由于该方法不易书写和记忆，现在基本上不使用它了；另一种方法是汇编语言表示法，它是对机器语言的改进，由于采用了人们便于记忆的一些符号（如简化的英文单词）来表示指令，通常将该方法称为助记符表示法。目前大家见到最多的，就是使用助记符表示的汇编语言指令。

1. 指令格式

MSP430 单片机指令由 4 部分组成，其格式如下。

[标号：]	操作码	操作数	[；注释]

例如，

```
Label:  MOV R0, R5  ;R0→R5
```

可见，一条指令通常由标号、操作码、操作数和注释四部分构成，并且这四部分的相对位置是不能自由更改的。各个部分之间由空格或制表符隔开。其中，标号与注释部分的内容可以没有。现就各部分的含义进行说明。

标号(Label)：为用户自己设定的标记符号，用于指示该指令的起始(即指令第一个字节的)地址，也称为符号地址。汇编时，该符号被赋予该指令的起始地址。书写时，标号一般居左对齐，后面不必用冒号。

操作码(Operation)：规定了指令的功能，因此任何语句都必须具有操作码，不可省略。此部分一般用便于记忆的助记符表示，如上例中的 MOV。

操作数(Operands)：规定了指令要操作的数据信息(主要是数据类型和寻址方式)。一条指令可以具有 0 个、1 个、2 个甚至多个操作数。若具有两个操作数，则第一个为源操作数，第二个为目的操作数。源操作数与目的操作数之间用逗号隔开。若只有一个操作数，则它既是源操作数又是目的操作数。

注释(Comment)：是语句的说明部分，用来指示该指令具体完成什么功能。注释主要是为了方便程序员阅读程序而设定的。它须用分号和前面的部分隔开。在汇编时，该部分内容将被忽略掉。

2. 本章用到的符号

为方便读者阅读与学习，现将指令系统与本书中经常使用的符号进行逐一说明。

(1) *。前面带有 * 符号的指令为仿真指令。

(2) →。表示数据的传输方向，如 A→B 表示将 A 的内容存储到 B 中。

(3) src。表示源操作数；dst 表示目的操作数。

(4) @。表示寄存器间接寻址。

(5) #data。表示 data 为立即数。

(6) &addr。表示 addr 为绝对地址。

(7) Rn。表示 R0～R15。

(8) PC/R0。表示程序寄存器。

(9) SP/R1。表示堆栈指针。

(10) TOS。表示堆栈顶。

(11) &。地址指示符，表明其后面的数据为具体的地址。

(12) MSB。表示最高有效位。

(13) LSB。表示最低有效位。

3.1.2 寻址方式

单片机执行程序的过程，实际上就是不断寻找操作数并对操作数进行操作的过程。寻址方式是指在程序执行过程中，指令寻找操作数所使用的方法。寻址方式是单片机的重要性能指标之一，也是汇编程序设计中最基本的内容。一般情况下，寻址方式越多，功能就越强，灵活性也就越大。操作数的存放不外乎 3 种情况：①操作数包含在指令中，即指令的操

作数字段包含操作数本身，这种操作数为立即数；②操作数包含在内部寄存器中，指令中的操作数字段是内部寄存器的一个编码，这种寻址方式称为寄存器寻址；③操作数在内存数据区，操作数字段包含着此操作数地址。

MSP430 单片机支持立即寻址、绝对寻址、变址寻址、符号寻址、寄存器间接寻址、寄存器寻址、自动增量间接寻址等多种寻址方式。

1. 立即寻址

立即寻址，又称立即数寻址。在这种寻址方式中，操作数在指令中由源操作数直接给出。例如：

```
MOV  #0020H, R4  ;20H→R4
```

注意：

① 立即数前需要加“#”，即 #data；否则将会出错。例如，MOV 0020H，R4 为非法指令。

② 立即寻址只能用于源操作数的寻址。例如，MOV　R4，#0020H 则是错误的用法。

2. 绝对寻址

绝对寻址，又称直接寻址。绝对寻址的操作数位于 RAM 内，操作数的地址直接在指令中给出。例如：

```
MOV R9, &2360H                ;将 R9 中的内容存入 2360 H 地址单元中
MOV &2000H, R7                ;将 2000 H 地址单元中的内容存入 R7 中
```

注意：

① 绝对地址前需要加“&”，即 &addr，否则将会出错。例如，MOV 2000H，R7 为非法指令。

② 绝对寻址方式适用于源操作数寻址和目的操作数寻址。

③ 该寻址方式主要用于访问具有绝对固定地址的外设模块。对它们使用绝对寻址，可以保证软件的透明度。

3. 符号寻址

该寻址方式的操作数在 RAM 中，但操作数的地址却在指令中以标号的形式给出。例如：

```
AAA  MOV  #2036H, R9          ;AAA 表示该指令的起始地址
     MOV  AAA,  R8            ;将 AAA 地址单元的内容存放在 R8 中
```

又如：

```
MOV Tab1, R10                 ;将数据表中第一个字存放在 R10 中
Tab1 DW 10E2H, 2214H, 59A2H   ;定义一块数据区域，标号 Tab1 代表该数据区域的起始地址
```

该语句执行后的结果为 R10＝10E2H。

注意：

① 这里所说的符号是程序员在程序中自己定义的标号，其代表的是此指令的起始地址。

② 绝对寻址中，符号所指代的具体地址可由汇编器计算得到，也可由程序员给出；而符号寻址中的地址是由程序员在编写程序时给出的。

③ 标号前加上“&”，表示的是地址。

④ 该寻址方式适用于源操作数寻址和目的操作数寻址。

⑤ 该寻址方式一般用于随机访问。

4. 寄存器寻址

该寻址方式的操作数存放在寄存器中。例如：

```
MOV R10, R8                    ;将 R10 的内容存放在 R8 中
MOV #1205H, R7                 ;将立即数 1205H 存放在 R7 中
MOV R6, &120H                  ;将 R6 的内容存放到 0120H 单元中
```

注意：

① 该寻址方式将源操作数中内容移动到目的操作数，但源操作数的内容不变。

② 该寻址方式适用于源操作数寻址和目的操作数寻址。

③ 该寻址方式通常用于对时间要求较严格的操作。

5. 变址寻址

该寻址方式的操作数存放在 RAM 中，操作数的地址为寄存器的内容加上前面的偏移量。例如：

```
MOV #0400H,R7                  ;将立即数 0400H 存放在 R7 中
MOV #0530H,R4                  ;将立即数 0530H 存放在 R4 中
MOV 3(R7), 5(R4)               ;将 0400H + 3 地址单元的内容存放到 0530H + 5 地址单元中
```

注意：① 该寻址方式将源操作数或目的操作数涉及的寄存器内容在执行前后不变。

② 该寻址方式适用于源操作数寻址和目的操作数寻址。

6. 寄存器间接寻址

寄存器间接寻址，也称间接寻址。该寻址方式的操作数存放在 RAM 中，但此操作数的地址位于寄存器中。例如：

```
MOV #0453H,R5                  ;将立即数 0453H 存放在 R5 中
MOV @R5, R4                    ;将 0453H 地址单元的内容存放到 R4 中
```

注意：

① 该寻址方式将源操作数或目的操作数涉及的寄存器内容在执行前后不变。

② 该寻址方式只适用于源操作数。

③ 若要在目的操作数中也使用间址寻址，需用变址寻址方式间接实现，即用 0(Rn)代替@Rn。例如，MOV @R5，0(R4)即可实现目的操作数的寄存器间接寻址。

7. 自动增量寄存器间接寻址

该寻址方式的操作数存放在 RAM 中，但此操作数的地址位于寄存器中。可以看出该寻址方式与寄存器间接寻址方式大致相同，唯一不同的是，使用自动增量寄存器间接寻址方式的寄存器在执行完间接寻址之后会自动增加。具体增加的数量视指令类型而定，如对于字节指令自动加 1、对于字指令自动加 2。例如：

```
MOV ＃0201H, R8                    ;将立即数 0201H 存放到 R8 中
MOV ＃3000H,R7                     ;将立即数 3000H 存放到 R7 中
MOV @R8＋, 0(R7)
            ;首先将 0201H 地址单元的内容存放到 3000H 地址单元中,然后 R8 中的内容自动增加 2
```

注意：① 该寻址方式只适用于源操作数。

② 该寻址方式适用于对表进行随机访问，但目的寄存器的内容需程序员手动改变。

MSP430 单片机指令的源操作数可以使用上述 7 种寻址方式，而目的操作数却只能使用其中的 4 种寻址方式，具体见表 3.1。

表 3.1　MSP430 寻址方式

寻址方式	使用场合	
寄存器寻址	源操作数	目的操作数
变址寻址	源操作数	目的操作数
符号寻址	源操作数	目的操作数
绝对寻址	源操作数	目的操作数
间接寻址	源操作数	—
自动增量间接寻址	源操作数	—
立即寻址	源操作数	—

3.2　指令系统

MSP430 单片机的指令系统为 16 位精简指令集系统，共由 51 条指令组成，其中27 条为内核指令(Core Instruction)，24 条为仿真指令(Emulated Instruction)。该指令系统具有寻址方式多、执行速度快、功能强的优点。

内核指令与仿真指令的差别在于，内核指令具有自己的操作码(op-code)，而仿真指令没有。仿真指令的作用就是便于程序员读写。在汇编时，汇编器将把仿真指令转换成具有唯一操作码的内核指令。因此，仿真指令的使用并不会影响指令长度和执行效率。

内核指令可分为单操作数指令、双操作数指令和跳转指令。而对于单操作数指令与双操作数指令，根据操作数的位数，又可分为字节指令(以.B 为后缀)和字指令(以.W 为后缀)。字节指令即以字节方式[1]访问数据或外设。字指令即以字方式[2]访问数据或外设。若指令不使用后缀(.B 或.W)，则系统默认为字指令。

MSP430 指令集按其功能划分，可分为数据传送指令(6 条)、算术运算指令(14 条)、逻辑操作指令(10 条)、位操作指令(8 条)、控制转移指令(13 条)。

3.2.1　数据传送指令

在单片机程序设计中，数据传送是最基本和最主要的操作，也是最为频繁的操作。

〔1〕 以字节方式就是每读写一次的数据宽度为 8 位，即一个字节。

〔2〕 以字方式就是每读写一次的数据宽度为 16 位，即一个字。

MSP430 单片机提供了 6 条与数据传送有关的指令，可以实现多种数据传送操作。

1. 通用数据传送指令

MOV 指令是最常用的数据传送指令，该指令格式为：

```
MOV[.B]  src, dst                    ;src→dst
```

该指令为内核指令，其功能就是将源操作数复制（或传送）到目的操作数中。期间目的操作数原有数据被覆盖掉，而源操作数不发生任何变化。

由于源操作数具有 7 种寻址方式，而目的操作数只有 4 种寻址方式。根据目的操作数的寻址方式不同，可进行以下划分。

（1）目的操作数使用寄存器寻址时，MOV 指令的格式如下。

```
MOV[.B]        Rm,         Rn        ;Rm→Rn
MOV[.B]        #data,      Rn        ;data→Rn
MOV[.B]        &addr,      Rn        ;(addr)→Rn
MOV[.B]        label,      Rn        ;(label)→Rn
MOV[.B]        X(Rm),      Rn        ;(Rm + X)→Rn
MOV[.B]        @(Rm),      Rn        ;(Rm)→Rn
MOV[.B]        @(Rm) + ,   Rn        ;(Rm)→Rn, Rm + 2[ - 1]
```

（2）目的操作数使用变址寻址时，MOV 指令的格式如下。

```
MOV[.B]        Rm,         X(Rn)     ;Rm→(Rn + X)
MOV[.B]        #data,      X(Rn)     ;data→(Rn + X)
MOV[.B]        &addr,      X(Rn)     ;(addr)→(Rn + X)
MOV[.B]        label,      X(Rn)     ;(label)→(Rn + X)
MOV[.B]        X(Rm),      X(Rn)     ;(Rm + X)→(Rn + X)
MOV[.B]        @(Rm),      X(Rn)     ;(Rm)→(Rn + X)
MOV[.B]        @(Rm) + ,   X(Rn)     ;(Rm)→(Rn + X), Rm + 2[ - 1]
```

（3）目的操作数使用绝对寻址时，MOV 指令的格式如下。

```
MOV[.B]        Rn,         &addr     ;Rn→(addr)
MOV[.B]        #data,      &addr     ;data→(addr)
MOV[.B]        &addr0,     &addr     ;(addr0)→(addr)
MOV[.B]        label,      &addr     ;(label)→(addr)
MOV[.B]        X(Rn),      &addr     ;(Rn + X)→(addr)
MOV[.B]        @(Rn),      &addr     ;(Rn)→(addr)
MOV[.B]        @(Rn) + ,   &addr     ;(Rn)→(addr), Rn + 2[ - 1]
```

（4）目的操作数使用符号寻址时，MOV 指令的格式如下。

```
MOV[.B]        Rn,         Label     ;Rn→(label)
MOV[.B]        #data,      Label     ;data→(label)
MOV[.B]        &addr,      Label     ;(addr)→(label)
MOV[.B]        Label0,     Label     ;(label0)→(label)
MOV[.B]        X(Rn),      Label     ;(Rn + X)→(label)
MOV[.B]        @(Rn),      Label     ;(Rn)→(label)
MOV[.B]        @(Rn) + ,   Label     ;(Rn)→(label), Rn + 2[ - 1]
```

由此可见，MOV 指令结合不同的寻址方式可以实现多种方向的数据传送。另外，该指令对状态标志位(N、Z、C、V)和控制标志位(OSCOFF、CPUOFF、GIE)均不产生任何影响。例如：

```
MOV R6, R8                          ;R6→R8
MOV 4(R8), R9                       ;(R8 + 4)→R9
MOV ＃020H, &2009H                  ;020H→(2009H)
MOV &0326H, R7                      ;(0326H)→R7
MOV R4, &2010H                      ;R4→(2010H)
```

2. 清零指令

该指令为仿真指令，可以实现对目的操作数进行清零处理，其指令格式如下。

```
＊CLR[.B]       dst                 ;0→dst
```

由于该指令只有一个目的操作数，因此该操作数只有 4 种寻址方式，具体如下。

```
＊CLR [.B]      Rn                  ;0→Rn
＊CLR [.B]      X(Rn)               ;0→(Rn + X)
＊CLR [.B]      &addr               ;0→(addr)
＊CLR [.B]      Label               ;0→(label)
```

例如：

```
CLR R5                              ;0→R5
CLR 8(R9)                           ;0→(R9 + 8)
CLR &2012H                          ;0→(2012H)
```

注意：① 该指令是由 MOV ＃0，dst 仿真的指令。其中的＃0 由常数发生器产生。

② 该指令对状态标志位(N、Z、C、V)和控制标志位(OSCOFF、CPUOFF、GIE)均不产生任何影响。

3. 堆栈操作指令

堆栈操作是一种比较特殊的数据传送操作，其特点是根据堆栈指针指示的地址进行数据操作，堆栈操作包括进栈操作和出栈操作。

(1) 进栈指令。进栈指令亦称入栈指令或压栈指令，其功能就是将操作数传送到堆栈中，指令格式如下。

```
PUSH[.B]        src                 ;SP-2→SP, src→(SP)
```

执行进栈操作时，堆栈指针减 2，再把源操作数传送到 SP 所指向的堆栈的顶部。

```
PUSH[.B]        Rn                  ;SP-2→SP, Rn→(SP)
PUSH[.B]        X(Rn)               ;SP-2→SP, (Rn + X)→(SP)
PUSH[.B]        @Rn                 ;SP-2→SP, (Rn)→(SP)
PUSH[.B]        @Rn +               ;SP-2→SP, (Rn)→(SP), Rn + 2[-1]
PUSH[.B]        &addr               ;SP-2→SP, (addr)→(SP)
PUSH[.B]        Label               ;SP-2→SP, (label)→(SP)
PUSH[.B]        ＃data              ;SP-2→SP, data→(SP)
```

注意：① PUSH 指令常用于保护现场或临时保存某一个字或字节数据。

② 对于 PUSH 指令，系统堆栈的指针始终减 2，并与字节后缀无关。主要原因是，系统

堆栈不仅用于 PUSH 指令，还用于中断服务程序。

③ 该指令对状态标志位(N、Z、C、V)和控制标志位(OSCOFF、CPUOFF、GIE)均不产生任何影响。

④ PUSH 指令与 POP 指令一般是成对出现的。

(2) 出栈指令。出栈指令亦称弹出指令，其功能就是将堆栈中操作数传送到目的操作数中，指令格式如下。

```
* POP[.B]      dst                ;(SP)→tmp, SP + 2→SP, tmp→src
```

该仿真指令由 MOV @SP+，dst 仿真。执行出栈操作时，首先将 SP 指向的堆栈顶部的数据存放于临时位置，然后堆栈指针加 2，之后再把存于临时位置的数据传送到目的操作数中。

```
* POP[.B]      Rn                 ;(SP)→tmp, SP + 2→SP, tmp→Rn
* POP[.B]      X(Rn)              ;(SP)→tmp, SP + 2→SP, tmp→(Rn + X)
* POP[.B]      &addr              ;(SP)→tmp, SP + 2→SP, tmp→(addr)
* POP[.B]      Label              ;(SP)→tmp, SP + 2→SP, tmp→(label)
```

注意：① POP 指令常用于恢复现场。

② 对于 POP 指令，系统堆栈的指针始终加 2，并与字节后缀无关。其原因是系统堆栈不仅用于 POP 指令，还用于 RETI 指令。

③ 该指令对状态标志位(N、Z、C、V)和控制标志位(OSCOFF、CPUOFF、GIE)不产生任何影响。

④ PUSH 指令与 POP 指令一般是成对出现的。

例 3.1 已知图 3.1(a)所示为某一部分堆栈空间，试分析：

(1) 执行完指令“PUSH #3456H”后堆栈空间中数据存储情况。

(2) 执行完指令“POP &0208H”后堆栈空间中数据存储情况。

解：由图 3.1(a)可知，当前栈顶指针 SP 正指向地址 0204H 处。首先分析第一种情况，执行完指令“PUSH #3456H”。根据 PUSH 指令将 SP 减 2 再传输数据的特点，数据入栈后，SP 将指向地址 0202H 处。再根据字数据“高位占据高地址”的数据存放规则易知，0202H 处存放 56H，0203H 处存放 34H，具体如图 3.1(b)所示。

对于第二种情况，首先根据 POP 指令先取数据再 SP+2 的特点可知，指令执行完后堆栈指针将指向地址 0206H 处。从堆栈取走的数据将在地址 0208H 处存放，具体可参见图 3.1(c)。这需要注意两点：第一，数据从堆栈取走后，除非有新的数据将该数据覆盖，否则堆栈中就一直保留着原数据；第二，堆栈空间中的数据一定得有规律存放，否则易导致系统不稳定或死机。即一般 PUSH 和 POP 指令是成对出现的。

4. 字操作指令

这里介绍的字操作指令主要包括字节交换指令和符号扩展指令。这两条指令的共同点是两者均是内核指令，并且指令的操作数只能是字。

(1) 字节交换指令。该指令的功能是将低位字节与高位字节进行互换，如图 3.2 所示，其指令格式如下。

```
SWPB          dst                ;第 15～8 位与第 7～0 位互换
```

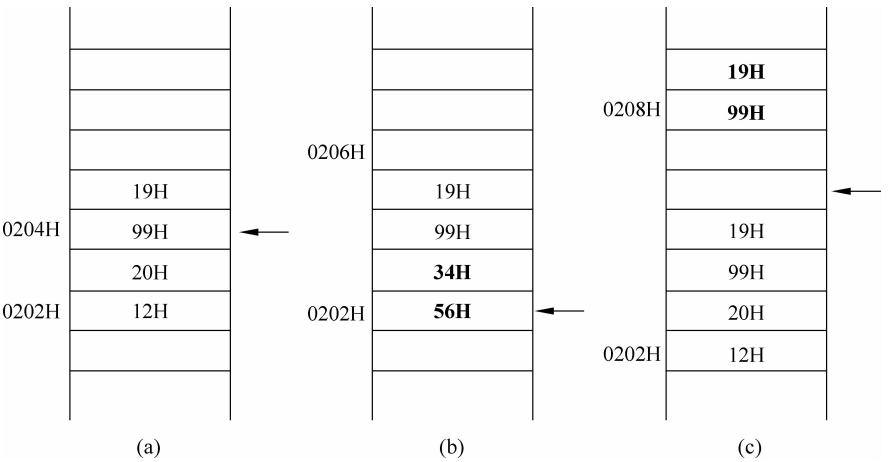


图 3.1　堆栈操作示意图

由指令功能可知，该指令的操作数既是源操作数，也是目的操作数。因此，该操作数具有 4 种寻址方式具体如下。

```
SWPB        Rn
SWPB        X(Rn)
SWPB        &addr
SWPB        Label
```

该指令对状态标志位(N、Z、C、V)和控制标志位(OSCOFF、CPUOFF、GIE)均不产生任何影响。

(2) 符号扩展指令。该指令的功能是将低位字节的符号位扩展到高位字节，也就是将第 0～7 位填充到第 8～15 位中，如图 3.3 所示，指令格式如下。

```
SXT         dst                 ;将第 0～7 位填充到第 8～15 位中
```

由指令功能可知，该指令的操作数既是源操作数，也是目的操作数。因此，该操作数具有 4 种寻址方式具体如下。

```
SXT         Rn
SXT         X(Rn)
SXT         &addr
XT          Label
```

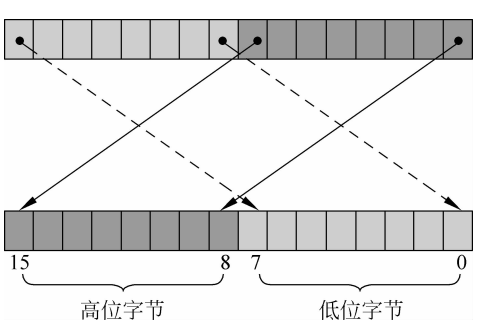


图 3.2　数据交换指令示意图

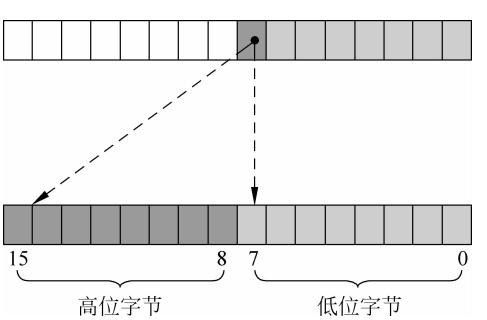


图 3.3　符号扩展指令示意图

该指令对控制标志位(OSCOFF、CPUOFF、GIE)不产生任何影响,但对状态标志位(N、Z、C、V)有影响,具体影响如表 3.2 所示。

表 3.2　字操作指令对状态标志位的影响情况

进位标志(C)	0	其他情况	负标志(N)	0	结果为正
	1	有进位时		1	结果为负
零标志(Z)	0	结果不为零	溢出标志(V)	0	此操作不会引起溢出,故一直为 0
	1	结果为零			

可见 MSP430 共有 6 条数据传输指令,前 4 条指令既可以字操作也可以字节操作,但后两条只能进行字操作。由于 MSP430 的寄存器均是 16 位的,所以对寄存器进行字节操作时,默认是低位字节参与。

例如:

```
MOV #2011H, R6                    ;R6 = 2011H
MOV.B #28H, R6                    ;R6 = 0028H
```

对存储单元进行字操作时,需要格外留意字的起始字节为低位偶数字节。如果所给地址不是偶地址,系统会自动进行偶地址对齐。

例如:

```
MOV #2009H, &0321H                ;(0320H) = 09H, (0321H) = 20H
MOV #2009H, &0322H                ;(0322H) = 09H, (0323H) = 20H
```

3.2.2　算术运算指令

MSP430 单片机具有 14 条算术运算指令,这些指令又分为加法运算指令和减法运算指令,并且这两类指令均会对标志位产生相应的影响。但对控制标志位(OSCOFF、CPUOFF、GIE)不产生任何影响。

1. 加法指令

MSP430 的加法类指令有 7 条,具体如下。

```
ADD[.B]         src, dst          ;src + dst→dst
ADDC[.B]        src, dst          ;src + dst + C→dst
DADD[.B]        src, dst          ;src + dst + C→dst(十进制)

ADC[.B]         dst               ;dst + C→dst
DADC[.B]        dst               ;dst + C→dst(十进制)

INC[.B]         dst               ;dst + 1→dst
INCD[.B]        dst               ;dst + 2→dst
```

(1) ADD 指令。该指令实现源操作数与目的操作数相加的功能,指令格式如下。

```
ADD[.B]  src, dst                 ;src + dst→dst
```

ADD 指令对状态标志位的影响情况如表 3.3 所示。

表 3.3　ADD 指令对状态标志位的影响情况

进位标志(C)	0	其他情况	负标志(N)	0	结果为正
	1	结果不为零		1	结果为负
零标志(Z)	0	结果不为零	溢出标志(V)	0	结果不溢出
	1	结果为零		1	结果溢出

(2) ADDC 指令。该指令为带进位位的加法指令,可以实现源操作数、目的操作数和进位位相加的功能,指令格式如下。

```
ADDC[.B]        src, dst                ;src + dst + C→dst
```

该指令与 ADD 指令类似,其中区别是 ADDC 指令在进行两个操作数相加时,同时把进位位的值也加了上去,结果存放在目的操作数中。需要格外留意的是,这里的进位位是指 ADDC 指令执行前的值。

(3) DADD 指令。该指令为带进位位的 BCD 数加法指令,指令格式如下。

```
DADD[.B]        src(BCD), dst(BCD)      ;src(BCD) + dst(BCD) + C(BCD)→dst(BCD)
```

该指令与 ADDC 指令类似,其中区别是 ADDC 指令进行两个二进制数相加,同时把进位位的值也加了上去,结果存放在目的操作数中。而 DADD 指令的两个操作数均是 BCD 数。注意,这里的进位位是指 DADD 指令执行前的值,并且此时的进位是 BCD 加法的进位情况。

DADD 指令对状态标志位的影响情况如表 3.4。

表 3.4　DADD 对状态标志位的影响情况

进位标志(C)	0	其他情况	负标志(N)	0	其他情况
	1	结果大于 9999		1	MSB为 1
零标志(Z)	0	结果不为零	溢出标志(V)	不确定	
	1	结果为零			

(4) * ADC 指令。该指令为仿真指令,可以实现将进位位加到目的操作数中,指令格式如下。

```
ADC[.B]         dst             ;dst + C→dst
```

该指令可由“ADDC.B #0H, dst”指令仿真实现,用于目的操作数寻址的 4 种寻址方式均可用于 ADC 指令。

(5) * DADC 指令。该指令为仿真指令,可以实现将 BCD 进位位加到目的操作数中,指令格式如下。

```
DADC[.B]        dst(BCD)        ;dst(BCD) + C(BCD)→dst(BCD)
```

该指令可由“DADD.B #0, dst”指令仿真实现,用于目的操作数寻址的 4 种寻址方式均可用于 DADC 指令。

该指令与 ADC 指令类似,唯一不同的是,操作数的性质不一样。ADC 指令的操作数是

二进制数，而 DADC 指令的操作数是 BCD 数。

（6）＊INC 指令。该指令为仿真指令，可以实现目的操作数加 1，指令格式如下。

```
INC[.B]        dst          ;dst + 1→dst
```

该指令可由“ADD.B ＃1，dst”指令仿真实现，一般用在循环程序中修改地址指针和循环次数。可用于目的操作数寻址的 4 种寻址方式均可用于 INC 指令，具体如下。

```
INC[.B]        Rn           ;Rn + 2→Rn
INC[.B]        X(Rn)        ;(Rn + x) + 1→(Rn + x)
INC[.B]        Label        ;(Label) + 1→(Label)
INC[.B]        &addr        ;(addr) + 1→(addr)
```

对于状态位的影响如表 3.5 所示。

表 3.5　＊INC 指令对状态位的影响情况

进位标志(C)	0	其他情况	负标志(N)	0	结果为正
	1	dst＝0FFFFH		1	结果为负
零标志(Z)	0	其他情况	溢出标志(V)	0	其他情况
	1	dst＝0FFFFH		1	dst＝0FFFFH

（7）＊INCD 指令。该指令为仿真指令，可以实现目的操作数加 2，指令格式如下。

```
INCD[.B]       dst          ;dst + 2→dst
```

该指令可由“ADD.B ＃2，dst”指令仿真实现。该指令与 INC 指令基本一致，差别在于 INCD 指令一次完成目的操作数加 2，用于目的操作数寻址的 4 种寻址方式均可用于 INCD 指令，具体如下。

```
INCD[.B]       Rn           ;Rn + 1→Rn
INCD[.B]       X(Rn)        ;(Rn + x) + 2→(Rn + x)
INCD[.B]       Label        ;(Label) + 2→(Label)
INCD[.B]       &addr        ;(addr) + 2→(addr)
```

对于状态位的影响如表 3.6 所示。

表 3.6　＊INCD 指令对状态标志位的影响情况

进位标志(C)	0	其他情况	负标志(N)	0	结果为正
	1	dst＝0FFFFH 或 dst＝0FFFEH		1	结果为负
零标志(Z)	0	其他情况	溢出标志(V)	0	其他情况
	1	dst＝0FFFFH		1	dst＝0FFFFH 或 dst＝0FFFEH

2. 减法指令

MSP430 单片机共有 7 条与减法运算相关的指令，具体如下。

```
SUB[.B]        src, dst              ;dst - src→dst
SUBC[.B]       src, dst              ;dst - src - 1 + C→dst

CMP[.B]        src,dst               ;dst - src
SBC[.B]        dst                   ;dst - 1 + C
TST[.B]        dst                   ;dst - 0

DEC[.B]        dst                   ;dst - 1→dst
DECD[.B]       dst                   ;dst - 2→dst
```

(1) SUB 指令。该指令为不带借位位的减法指令,可以实现两个操作数的相减。指令格式如下。

```
SUB[.B]        src, dst              ;dst - src→dst
```

该指令的功能是将目的操作数减去源操作数的结果存放在目的操作数中。该指令的具体实现方法是先将源操作数取反加 1,然后再与目的操作数相加,再将结果存放在目的操作数中。

这里目的操作数既可以是寄存器也可以是存储单元。源操作数可使用 7 种所有的寻址方式。该指令对状态的影响如表 3.7 所示。

表 3.7　SUB 指令对状态标志位的影响情况

进位标志(C)	0	借位或其他情况	负标志(N)	0	结果为正
	1	无借位或结果的 MSB 产生进位		1	结果为负
零标志(Z)	0	结果不为零	溢出标志(V)	0	结果不溢出
	1	结果为零		1	结果溢出

(2) SUBC(SBB)指令。该指令为带借位的减法指令,指令格式如下。

```
SUBC[.B]       src, dst              ;dst - src - 1 + C→dst
```

该指令与 SUB 类似,不同的是,该指令在进行两操作数相减时考虑了进位标志位的状态。因此,它经常用于多字节(字)的运算中。该指令对状态标志位的影响如表 3.8 所示。

表 3.8　SUBC 指令对状态标志位的影响情况

进位标志(C)	0	借位或其他情况	负标志(N)	0	结果为正
	1	无借位或结果的 MSB 产生进位		1	结果为负
零标志(Z)	0	结果不为零	溢出标志(V)	0	结果不溢出
	1	结果为零		1	结果溢出

(3) CMP 指令。该指令为比较指令,其功能实现两个数相减,但是不保存结果。也就是说,该指令仅仅影响状态标志位,指令执行前后操作数内容不变,指令格式如下。

```
CMP.[B]        src, dst    ;dst - src
```

该指令常用于比较两个数的大小关系。若相等，则 Z 标志位为 1；若不相等，则可以根据其他标志位进行判断谁大谁小。具体判断方法如下：假设有两个数 X 与 Y，执行 CMP X，Y 后：

① 当 Z=0 时，$X=Y$；

② 若 X、Y 均为无符号数，则有：

➢ 当 C=0 时，则 $Y>X$；否则 $Y<X$。

➢ 当 N=0 时，则 $Y>X$；否则 $Y<X$。

③ 若 X、Y 均为有符号数，则有：

➢ 结果溢出时(V=1)，当 N=0，$X>Y$；否则 $X<Y$。

➢ 结果不溢出时(V=0)，当 N=0，$Y>X$；否则 $Y<X$。

(4) * SBC 指令。该指令是仿真指令，其功能可实现从目的操作数中减去借位，指令格式如下。

```
SBC[.B]        dst                    ;dst-1 + C→dst
```

该指令是由“SUBC ＃0，dst”仿真实现的。该指令对状态标志位的影响如表 3.9 所示。

表 3.9　* SBC 指令对状态标志位的影响情况

进位标志(C)	0	其他情况	负标志(N)	0	结果为正
	1	dst 从 0000 减到 0FFFF 时		1	结果为负
零标志(Z)	0	结果不为零	溢出标志(V)	0	结果不溢出
	1	结果为零		1	结果溢出

(5) * TST 指令。该指令是仿真指令，其功能是对目的操作数进行测试，指令格式如下。

```
TST[.B]        dst          ;dst-0
```

实质上，该指令的作用就是影响状态标志位，对目的操作数的内容没有影响。该指令通常放于条件转移指令之前。该指令由比较指令“CMP ＃0，dst”仿真实现，对状态标志位的影响如表 3.10。

表 3.10　* TST 指令对状态标志位的影响情况

进位标志(C)	0	—	溢出标志(V)	0	—
零标志(Z)	0	结果不为零	负标志(N)	0	结果为正
	1	结果为零		1	结果为负

(6) * DEC 指令。该指令为仿真指令，其功能是使目的操作数减 1，指令格式如下。

```
DEC            dst          ;dst-1→dst
```

目的操作数既可以是寄存器也可以是存储器。指令一般用在循环程序中，实现自动修改地址指针和循环次数。该指令由“SUB ＃1，dst”仿真实现，对状态标志位的影响如表 3.11。

表 3.11　* DEC 指令对状态标志位的影响情况

进位标志(C)	0	dst=0 时	负标志(N)	0	结果为正
	1	其他情况		1	结果为负
零标志(Z)	0	结果不为零	溢出标志(V)	0	结果不溢出
	1	结果为零		1	结果溢出或 dst 的初始值是 08000h 或 080h

(7) * DECD 指令。该指令为仿真指令,其功能是使目的操作数减 2,指令格式如下。

```
DECD        dst        ;dst - 2→dst
```

该指令由“SUB #2, dst”仿真实现,对状态标志位的影响如表 3.12 所示。

表 3.12　* DECD 指令对状态标志位的影响情况

进位标志(C)	0	dst=0 或 1 时	负标志(N)	0	结果为正
	1	其他情况		1	结果为负
零标志(Z)	0	结果不为零	溢出标志(V)	0	结果不溢出
	1	结果为零		1	结果溢出或 dst 的初始值是 08000h 或 08001h; 或 080h 或 081h

3.2.3　逻辑操作指令

逻辑操作指令共 10 条,其中逻辑运算指令 6 条,逻辑移位指令 4 条。这些指令对控制标志位(OSCOFF、CPUOFF、GIE)不产生任何影响,部分指令对状态标志位有影响。

```
AND[.B]    src, dst      ;src∧dst→dst
BIC[.B]    src, dst      ;src∧dst→dst
BIS[.B]    src, dst      ;src∨dst→dst
BIT[.B]    src,dst       ;src∧dst
XOR[.B]    src,dst       ;src⊕dst→dst
RLA[.B]    dst           ;C←MSB...LSB←0
RLC[.B]    dst           ;C←MSB...LSB←C
RRA[.B]    dst           ;MSB→MSB...LSB→C
RRC[.B]    dst           ;C→MSB...LSB→C
INV        dst           ;dst→dst
```

1. 逻辑运算指令

MSP430 单片机只有 6 条逻辑运算指令,其中 AND、XOR、INV、BIT 指令对状态标志位有影响,而 BIC、IS 对状态标志位不产生任何影响。

(1) AND 指令。该指令为逻辑与操作,可以实现按位“与”运算,指令格式如下。

```
AND[.B]    src, dst      ;src∧dst→dst
```

该指令通常用于对目的操作数的部分位进行置零操作。

例如，使目的操作数的 0、3 位置零，其他位不变。可使用指令“AND.B　＃F6h，dst”实现。即根据目的操作数中需要置零的位置信息，只需要对源操作数中的相应位置 0，其他位置 1，即可达到目的。需要注意的是，MSP430 单片机没有提供专门的位操作指令来实现对特定位的操作，因此只能通过使用字节或字模板和逻辑操作，来实现对特定位的操作。这一点初学者需要格外注意。

该指令对状态标志位的影响如表 3.13 所示。

表 3.13　AND 指令对状态标志位的影响情况

进位标志(C)	0	其他情况	负标志(N)	0	MSB=0
	1	结果不为零		1	MSB=1
零标志(Z)	0	结果不为零	溢出标志(V)	0	—
	1	结果为零			

(2) XOR 指令。该指令为异或操作指令，即实现目的操作数与源操作数的异或运算，其指令格式如下。

```
XOR src,dst                ;src⊕dst→dst
```

常用于目的操作数中的某些位保持不变，某些位取反的情况。具体做法是，根据目的操作数中需要取反位的位置信息，只需要对源操作数的相应位置 0，其他位置 1，即可达到目的。例如：

```
XOR        R5,  R5         ;R5 清零
XOR.B      ＃00FFh, R9     ;低字节取反,高字节不变
```

该指令对状态标志位的影响如表 3.14 所示。

表 3.14　XOR 指令对状态标志位的影响情况

进位标志(C)	0	其他情况	负标志(N)	0	MSB=0
	1	结果不为零		1	MSB=1
零标志(Z)	0	结果不为零	溢出标志(V)	0	其他情况
	1	结果为零		1	两个操作数均为负数时

(3) ＊INV 指令。该指令为仿真指令，由指令“XOR[.B] ＃0FF[FF]h，dst”仿真实现。其功能是对目的操作数逐位取反，指令格式如下。

```
INV[.B]    dst             ;dst→dst
```

该指令对状态标志位的影响如表 3.15 所示。

表 3.15　＊INV 指令对状态标志位的影响情况

进位标志(C)	0	其他情况	负标志(N)	0	结果为正
	1	结果不为零		1	结果为负
零标志(Z)	0	结果不为零	溢出标志(V)	0	其他情况
	1	结果为零		1	初始目的操作数为负

(4) BIT 指令。该指令为位测试指令,指令格式如下。

```
BIT[.B] src, dst     ;src∧dst
```

该指令与 AND 指令类似,唯一不同的是该指令并不将与运算的结果保存到目的操作数中,但其结果影响状态标志位。该指令对状态标志位的影响如表 3.16 所示。

表 3.16　BIT 指令对状态标志位的影响情况

进位标志(C)	0	其他情况	负标志(N)	0	MSB=0
	1	结果不为零		1	MSB=1
零标志(Z)	0	结果不为零	溢出标志(V)	0	—
	1	结果为零			

本指令通常应用在条件检测中。一般放在条件转移指令之前。例如:

```
BIT        #0200h,  R8  ;检测第 9 位是否为 1
JNZ        TOM          ;若是 1,转到 TOM 处继续执行
…                       ;否则,接着往下执行
```

(5) BIC 指令。该指令可对目的操作数的各位清零操作,指令格式如下。

```
BIC[.B] src, dst          ;(src)∧dst→dst
```

该指令在进行与运算前,先进行取反操作,然后再进行“与”运算。与 AND 指令类似,BIC 指令亦用于目的操作数中的某些位保持不变,某些位置零的场合中。注意,该指令对于状态标志位不产生任何影响。例如:

```
BIC        #0FC00h, R8   ;R8 高字节的高 6 位置零,其他位不变
```

(6) BIS 指令。该指令为“或”运算指令,其功能是对目的操作数的各位置 1,指令格式如下。

```
BIS[.B] src, dst          ;src∨dst→dst
```

该指令用于目的操作数中的某些位保持不变,某些位置 1 的场合中。具体做法是,根据目的操作数中需要置 1 的位置信息,只需要对源操作数的相应位置 0,其他位置 1 即可。注意,该指令对于状态标志位不产生任何影响。例如:

```
BIS        #003FH, R7    ;R7 低字节的低 6 位置 1,其他位不变
```

2. 逻辑移位指令

(1) *RLA 指令。该指令为仿真指令,由指令“ADD[.B] dst, dst”仿真实现,该指令的功能是实现目的操作数的算术左移,即每执行一次,目的操作数所有位向左移动一位,其中最高位移到进位标志位中,最低位补零,如图 3.4 所示,指令格式如下。

```
RLA  dst                  ;C←MSB…LSB←0
```

该指令对状态标志位的影响如表 3.17 所示。

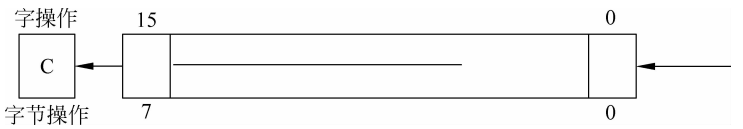


图 3.4　算术左移运算示意图

表 3.17　* RLA 指令对状态标志位的影响情况

进位标志(C)	dst 的最高位		负标志(N)	0	结果为正
				1	结果为负
零标志(Z)	0	其他情况	溢出标志(V)	0	其他情况
	1	结果为零		1	发生算术溢出时，如 03FFFh ＜ dst ＜ 0C000h 或 03Fh ＜ dst ＜ 0C0h

(2) RRA 指令。该指令为算术右移指令，指令格式如下。

```
RRA   dst      ;MSB→MSB…LSB→C
```

该指令与 RLA 移动的方向相反，即每执行一次该指令，目的操作数的所有位向右移动一位，其中，最低位移至进位标志位中，最高位保持不变(符号位不变)，如图 3.5 所示。

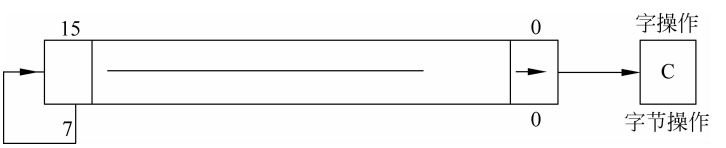


图 3.5　算术右移运算示意图

该指令对状态标志位的影响如表 3.18 所示。

表 3.18　RRA 指令对状态标志位的影响情况

进位标志(C)	dst 的最低位		溢出标志(V)	0	—
零标志(Z)	0	其他情况	负标志(N)	0	结果为正
	1	结果为零		1	结果为负

(3) * RLC 指令。该指令为仿真指令，由指令“ADDC[.B] dst，dst”仿真实现，该指令的功能是实现目的操作数带进位位的循环左移，如图 3.6 所示，指令格式如下。

```
RLC   dst      ;C←MSB…LSB←C
```

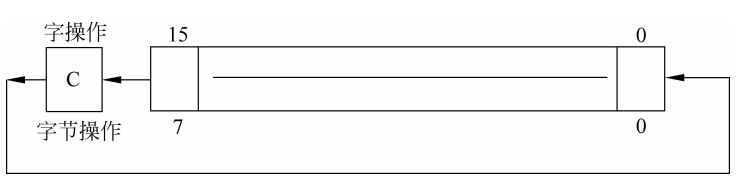


图 3.6　带进位位左移运算示意图

该指令对状态标志位的影响如表 3.19 所示。

表 3.19　＊RLC 指令对状态标志位的影响情况

进位标志(C)	dst 的最高位		负标志(N)	0	结果为正
				1	结果为负
零标志(Z)	0	其他情况	溢出标志(V)	0	其他情况
	1	结果为零		1	发生算术溢出时，如 03FFFh ＜ dst ＜ 0C000h 或 03Fh ＜ dst ＜ 0C0h

(4) RRC 指令。该指令为带进位位循环右移指令，如图 3.7 所示，指令格式如下。

```
RRC   dst      ;C→MSB…LSB→C
```

该指令对状态标志位的影响如表 3.20 所示。

表 3.20　RRC 指令对状态标志位的影响情况

进位标志(C)	dst 的最低位		溢出标志(V)	0	—
零标志(Z)	0	其他情况	负标志(N)	0	结果为正
	1	结果为零		1	结果为负

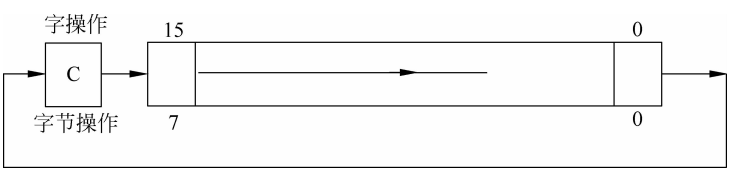


图 3.7　带进位位右移运算示意图

3.2.4　位操作指令

此类指令共 8 条，属于无操作数指令，只能对状态寄存器中的位进行操作。也就是说，MSP430 单片机的寄存器是不具备位寻址功能的。要想实现位操作，只能通过字或字节掩膜的办法实现。

MSP430 的位操作指令只有 8 条且均为仿真指令，分别对进位位、零标志位、负标志位及中断使能标志位进行设置，具体如下。

(1) ＊CLRC。该指令为进位位清零指令，由指令“BIC ＃1，SR”仿真实现。

(2) ＊SETC。该指令为进位位置 1 指令，由指令“BIS ＃1，SR”仿真实现。

(3) ＊CLRZ。该指令为零标志位清零指令，由指令“BIC ＃2，SR”仿真实现。

(4) ＊SETZ。该指令为零标志位置 1 指令，由指令“BIS ＃2，SR”仿真实现。

(5) ＊CLRN。该指令为负标志位置零指令，由指令“BIC ＃4，SR”仿真实现。

(6) ＊SETN。该指令为负标志位置 1 指令，由指令“BIS ＃4，SR”仿真实现。

(7) * DINT。该指令为禁止中断指令,即控制标志位 GIE 位清零。由指令“BIC #8, SR”仿真实现。

(8) * EINT。该指令为开中断指令,即控制标志位 GIE 位置 1。由指令“BIS #8, SR”仿真实现。

注意:前 6 条指令只对操作的位产生影响,并不对其他状态标志位和控制标志位(OscOff、CPUOff、GIE)产生影响。后两条指令只对控制标志位 GIE 产生影响,对其他标志位不产生任何影响。

3.2.5 控制转移指令

此类指令共 13 条,大致可分为无条件转移、条件转移、子程序调用、返回指令、空操作指令。该类指令(除 RETI 外)对状态标志位不产生任何影响。

1. 无条件转移指令

(1) * BR(BRANCH)指令。该指令为长转移指令 * BR(BRANCH),由指令“MOV dst, PC”仿真实现。指令格式如下。

```
BR    dst          ;dst→PC
```

该指令可以转移到 64KB 空间的任何一个地方,并且可以使用所有的 7 种源操作数寻址方式。该指令为单字指令。

(2) JMP。该指令为短转接指令,指令格式如下。

```
JMP   label        ;PC + 2 offset→PC
```

该指令的转移范围为−511～512。PC 寄存器的低 10 位为带符号的偏移量。

2. 条件转移指令

条件转移指令共有 7 条指令,由于它们的转移范围都位于−511 至 512 之间,所以条件转移指令都是短转移指令。条件转移就是根据状态标志位(C、Z、V、N)的值进行转移。

(1) JC/JHS 指令。该指令的功能是大于和等于时转移,指令格式如下:

```
JC    label        ;若 C = 0 则转移 label 处执行,否则继续往下执行
JHS   label        ;同上
```

(2) JNC/JLO 指令。该指令的功能是小于时转移,指令格式如下:

```
JNC   label        ;若 C = 0 则转移 label 处执行,否则继续往下执行
JLO   label        ;同上
```

(3) JEQ/JZ 指令。该指令的功能是相等/为零时转移,指令格式如下:

```
JZ    label        ;若 Z = 1 则转移 label 处执行,否则继续往下执行
JEQ   label        ;同上
```

(4) JNE/JNZ 指令。该指令的功能是不等/不为零时转移,指令格式如下:

```
JZ    label        ;若 Z = 0 则转移 label 处执行,否则继续往下执行
JEQ   label        ;同上
```

(5) JGE 指令。该指令的功能是大于等于时转移,指令格式如下:

```
JGE    label        ;若(N .XOR. V) = 0 则转移 label 处执行,否则继续往下执行
```

(6) JL 指令。该指令的功能是小于时转移,指令格式如下:

```
JL     label        ;若(N .XOR. V) = 1 则转移 label 处执行,否则继续往下执行
```

(7) JN 指令。该指令的功能是为负时转移,指令格式如下:

```
JN     label        ;若 N = 1 则转移 label 处执行,否则继续往下执行
```

3. 子程序调用

该指令用于调用子程序,其跳转的范围为 64KB。

```
CALL   dst          ;dst→tmp
                    ;SP - 2→SP,PC→@SP
                    ;tmp→PC
```

4. 返回指令

(1) * RET 指令。该指令为仿真指令,用于从子程序返回到主程序,由指令“MOV @SP+, PC”仿真实现。

```
RET                 ;@SP→PC, SP + 2→SP
```

(2) RETI 指令。该指令为中断程序返回指令,用于从中断服务程序返回到主程序。

```
RETI                ;(SP)→SR,SP + 2→SP
                    ;(SP)→PC,SP + 2→SP
```

5. 空操作指令

空操作指令主要用于填充存储器字和软件定时,指令形式如下。

```
* NOP
```

该指令也是仿真指令,由指令“MOV #0, R3”仿真实现。当然,其他指令亦可以仿真 NOP 指令,具体如下。

```
MOV    #0,     R3          ;1 个指令周期, 1 个字长度
MOV    0(R4),  0(R4)       ;6 个指令周期, 3 个字长度
MOV    @R4,    0(R4)       ;5 个指令周期, 2 个字长度
BIC    #0,     EDE(R4)     ;4 个指令周期, 2 个字长度
JMP    $ + 2               ;2 个指令周期, 1 个字长度
BIC    #0,     R5          ;1 个指令周期, 1 个字长度
```

需要注意的是,上述指令在执行过程中,可能会带来一些附加的影响,使用时务必要谨慎。

3.3　指令格式与指令周期

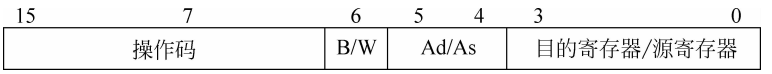
尽管 MSP430 单片机的指令系统具有 51 条指令,但真正意义上的指令也就是 27 条核心指令。所以在本节讨论指令格式与指令周期时所涉及的指令均是指内核指令。

3.3.1 指令格式

由于内核指令由单操作数指令、双操作数指令和跳转指令组成，这里分别对其指令代码格式的构成做一简单介绍。指令格式中，B/W 是字节操作标识符。该位为 1 时表示该指令为字节操作，为 0 时表示字操作；Ad 表示目的操作数使用的寻址方式(2 位)；As 表示源操作数使用的寻址方式(1 位)。

1. 单操作数指令

单操作数指令的格式如下。



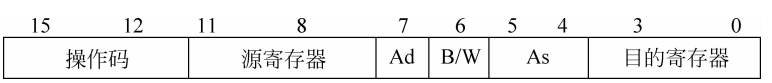
说明：虽然操作码部分占有 9 位(第 15～7 位)，但实际上第 15～10 位固定为 000100，所以只剩下 3 位(第 9～7 位)，因此单操作数指令只有 7 条指令，具体如表 3.21 所示。

表 3.21 单操作数指令及其相应的操作码

操作码									相应指令
15	14	13	12	11	10	9	8	7	
0	0	0	1	0	0	**0**	**0**	**0**	RRC(.B) dst
0	0	0	1	0	0	**0**	**0**	**1**	SWPB dst
0	0	0	1	0	0	**0**	**1**	**0**	RRA(.B) dst
0	0	0	1	0	0	**0**	**1**	**1**	SXT dst
0	0	0	1	0	0	**1**	**0**	**0**	PUSH(.B) src
0	0	0	1	0	0	**1**	**0**	**1**	CALL dst
0	0	0	1	0	0	**1**	**1**	**0**	RETI

2. 双操作数指令

双操作数指令的格式如下。



双操作数指令共计 12 条。操作码部分占有 4 位(第 15～12 位)，共有 16 种组合。但由于 0000～0011 这 4 种组合未使用，故有 12 条，具体如表 3.22 所示。

表 3.22 双操作数指令及其相应的操作码

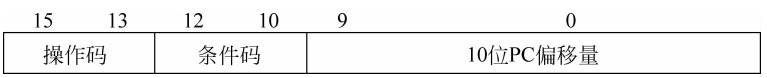
操作码				相应指令
15	14	13	12	
0	1	0	0	MOV(.B) src，dst
0	1	0	1	ADD (.B) src，dst
0	1	1	0	ADDC (.B) src，dst
0	1	1	1	SUBC (.B) src，dst
1	0	0	0	SUB (.B) src，dst

续表

操作码				相应指令	
15	14	13	12		
1	0	0	1	CMP (. B)	src, dst
1	0	1	0	DADD (. B)	src, dst
1	0	1	1	BIT (. B)	src, dst
1	1	0	0	BIC (. B)	src, dst
1	1	0	1	BIS (. B)	src, dst
1	1	1	0	XOR (. B)	src, dst
1	1	1	1	AND (. B)	src, dst

3. 跳转指令

跳转指令的格式如下。



跳转指令共 8 条，具体如表 3.23 所示。

表 3.23　跳转指令及其相应的操作码

操作码			条件码			相应指令	
15	14	13	12	11	10		
0	0	1	0	0	0	JEQ/JZ	label
0	0	1	0	0	1	JNE/JNZ	label
0	0	1	0	1	0	JC	label
0	0	1	0	1	1	JNC	label
0	0	1	1	0	0	JN	label
0	0	1	1	0	1	JGE	label
0	0	1	1	1	0	JL	label
0	0	1	1	1	1	JMP	label

3.3.2　指令周期

指令周期是执行一条指令所需要的时间，即从取指令、分析指令到执行完所需的全部时间。可见，指令周期就是指令的执行周期，亦称指令执行周期。指令的执行周期取决于指令字格式和寻址方式，而不是指令本身。通常，指令的执行周期是以主时钟(MCLK)作为参考的。

1. 寻址方式对指令周期的影响

同一条指令的指令周期会因所采用的寻址方式不同而存在较大的差异。现以最常见的数据传输指令 MOV 为例来说明这个问题。

(1) 寄存器组之间的数据传输，如“MOV Rn, Rm”，即源操作数与目的操作数均采用的是寄存器寻址方式。在该寻址方式下，指令执行效率最高。此时 MOV 指令的周期为一个 MCLK 周期。

(2) 寄存器与立即数之间的数据传输,如“MOV #data, Rn”,即源操作数采用立即数寻址,目的操作数采用寄存器寻址。在该寻址方式下,指令执行效率依然较高。此时 MOV 指令的周期为两个 MCLK 周期。

(3) 寄存器与存储器之间的数据传输,如“MOV Rn, EDE”,即源操作数采用寄存器寻址,目的操作数采用符号寻址。在该寻址方式下,指令执行效率相比前者就低得多了。此时 MOV 指令的周期为 4 个 MCLK 周期。

(4) 存储器与存储器之间的数据传输,如“MOV @Rn(@Rn+), EDE”,即源操作数采用间接(自动增量间接)寻址,目的操作数采用符号寻址。在该寻址方式下,指令执行效率相比前者更低。此时 MOV 指令的周期为 5 个 MCLK 周期。

指令执行周期最长的是类似“MOV &EDE, EDE”这样的数据传输。在该类寻址方式下,指令执行效率最低。此时 MOV 指令的周期为 6 个 MCLK 周期。

2. 指令格式对指令周期的影响

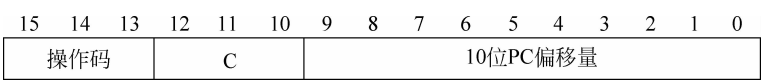
由上可知,影响指令周期长短的主要因素是寻址方式。但随着指令中操作数的多少也对指令周期产生一定影响。MSP430 单片机的指令按照指令格式共分为 3 种指令,即跳转指令、单操作数指令与双操作数指令。对于跳转指令而言,其指令周期是固定的,均是两个 MCLK。通常,单操作数指令一般比双操作数指令要快。

3.3.3 指令长度

指令的长度与指令格式、指令的寻址方式也有很大关系。由于 MSP430 的指令格式有 3 种分别是跳转指令、单操作数指令与双操作数指令。现分别介绍这 3 种指令格式与指令长度之间的关系。

1. 跳转指令

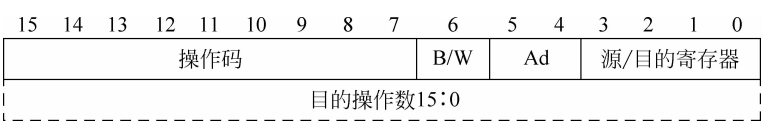
对于跳转指令,其指令格式如下。



可见,该类指令的长度只有一个字。

2. 单操作数指令

对于单操作数指令,其指令格式如下。



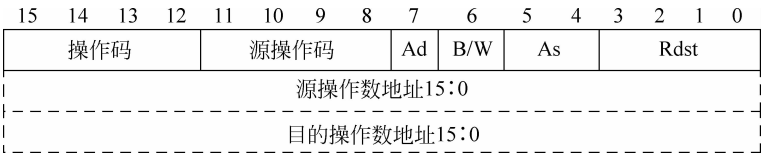
由单操作数指令格式可知,该类指令的指令长度只有两种情况,要么是一个字,要么是两个字。具体长度视该指令的寻址方式而定。

(1) 若采用寄存器寻址、寄存器间接寻址或自动增量寄存器间接寻址,指令长度为 1 个字。此时源/目的寄存器的位置实际为寄存器编号。

(2) 若采用立即数寻址、变址寻址、符号寻址、绝对寻址,则指令长度为两个字。

3. 双操作数指令

对于双操作数指令,其指令格式如下。



由双操作数指令格式可知，该类指令长度介于 1～3 个字之间。而指令采用的寻址方式则直接决定了该指令的长度。

(1) 若源操作数与目的操作数均采用了寄存器寻址、寄存器间接寻址或自动增量寄存器间接寻址，则该指令的长度为 1 个字。

(2) 若两操作数中只有一个操作数采用了寄存器寻址、寄存器间接寻址或自动增量寄存器间接寻址，则该指令的指令长度为 2 个字。

(3) 若源操作数与目的操作数均采用变址寻址、绝对寻址、符号寻址或立即数寻址，则该指令的指令长度为 3 个字。

需要说明的是，指令长度与指令周期之间没有什么必然的联系。并不是说指令长度越短，其指令周期也就越短。比如，指令 PUSH @Rn+，该指令长度为 1 个字长，但其指令周期是 5 个 MCLK。类似地，还有中断返回指令 RETI 其指令长度为 1 个字长，但其指令周期却是 5 个 MCLK。

3.4　MSP430X 指令系统

MSP430X 是对 MSP430 指令系统的扩展，以适应存储器空间扩大的需要。此前所讲的 MSP430 指令集是针对最大为 64KB 存储空间设计的。而 MSP430X 指令集中的指令可以直接访问 1MB 空间。图 3.8 所示为扩展前后存储空间的变化情况。为了与存储空间小于 64KB 的单片机程序做到兼容，扩展后原 64KB 存储空间的功能区分配不变，超出 64KB 范围的用作程序存储空间。

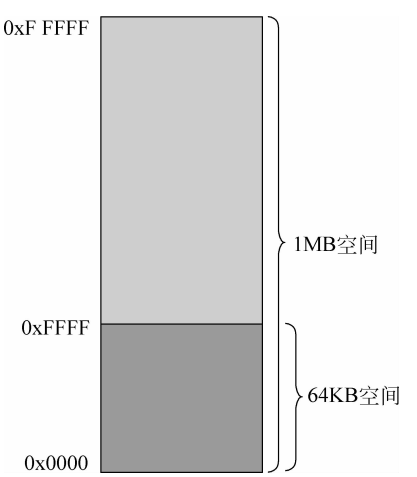


图 3.8　存储器扩展示意图

当存储空间扩展到 1MB 时，其地址位也由 16 位扩展到 20 位。即访问 64KB 空间外的数据时，使用的绝对地址必然大于 16 位。因此，用于存放地址的寄存器必须扩充。这是 MSP430X 产生的重要原因。

为了更好、更快地处理地址空间扩展后指令的执行性能，在原来指令集的基础上，增加了一些新指令。寻址方式上也有不同程度的扩展。

这里仅简要介绍指令集的扩展情况，而有关 MSP430X 其他扩展情况，请参阅 MSP430 相关系列的用户手册。

3.4.1　指令集的扩展

MSP430X 对单操作数指令、双操作数指令和仿真指令都进行了部分扩充，以满足寻址

范围扩大后的需要。

1. 扩展的单操作数指令

在原先的基础上，单操作数指令扩充了 16 条指令，如表 3.24 所示。

表 3.24　扩展单操作数指令

序号	扩展指令	操作数	说　明
1	CALLA	dst	调用子程序指令
2	POPM.A	#n,Rdst	重复出栈指令
3	POPM.W	#n,Rdst	重复出栈指令
4	PUSHM.A	#n,Rsrc	重复压栈指令
5	PUSHM.W	#n,Rsrc	重复压栈指令
6	PUSHX[.B,.A]	src	压栈指令
7	RRCM[.A]	#n,Rdst	重复带进位位循环右移指令
8	RRUM[.A]	#n,Rdst	重复无符号数算术右移指令
9	RRAM[.A]	#n,Rdst	重复算术右移指令
10	RLAM[.A]	#n,Rdst	重复算术左移指令
11	RRCX[.B,.A]	dst	带进位位循环右移指令
12	RRUX[.B,.A]	dst	无符号数算术右移指令
13	RRAX[.B,.A]	dst	算术右移指令
14	SWPBX[.A]	dst	字节交换治疗
15	SXTX[.A]	Rdst	位扩展指令
16	SXTX[.A]	dst	位扩展指令

2. 扩展的双操作数指令

在原先的基础上，双操作数指令扩充了 12 条指令，如表 3.25 所示。

表 3.25　扩展双操作数指令

序号	扩展指令	操作数	说　明
1	MOVX[.B,.A]	src,dst	数据传送指令
2	ADDX[.B,.A]	src,dst	加法指令
3	ADDCX[.B,.A]	src,dst	带进位的加法指令
4	SUBX[.B,.A]	src,dst	减法指令
5	SUBCX[.B,.A]	src,dst	带借位的减法指令
6	CMPX[.B,.A]	src,dst	比较指令
7	DADDX[.B,.A]	src,dst	带进位的十进制加法指令
8	BITX[.B,.A]	src,dst	位测试指令
9	BICX[.B,.A]	src,dst	位清零指令
10	BISX[.B,.A]	src,dst	或运算指令
11	XORX[.B,.A]	src,dst	异或运算指令
12	ANDX[.B,.A]	src,dst	与运算指令

3. 扩展的仿真指令

在原先的基础上，仿真指令扩充了 19 条指令，如表 3.26 所示。

表 3.26　扩展仿真指令

序号	扩展指令	操作数	说　明	用于仿真的指令
1	ADCX[.B,.A]	dst	加进位位指令	ADDCX[.B,.A] #0,dst
2	BRA	dst	无条件转移指令	MOVA dst,PC
3	RETA		子程序返回指令	MOVA @SP+,PC
4	CLRA	Rdst	清零指令	MOV #0,Rdst
5	CLRX[.B,.A]	dst	清零指令	MOVX[.B,.A] #0,dst
6	DADCX[.B,.A]	dst	十进制加进位位指令	DADDX[.B,.A] #0,dst
7	DECX[.B,.A]	dst	减 1 指令	SUBX[.B,.A] #1,dst
8	DECDA	Rdst	减 2 指令	SUBA #2,Rdst
9	DECDX[.B,.A]	dst	减 2 指令	SUBX[.B,.A] #2,dst
10	INCX[.B,.A]	dst	加 1 指令	ADDX[.B,.A] #1,dst
11	INCDA	Rdst	加 2 指令	ADDA #2,Rdst
12	INCDX[.B,.A]	dst	加 2 指令	ADDX[.B,.A] #2,dst
13	INVX[.B,.A]	dst	取反指令	XORX[.B,.A] #-1,dst
14	RLAX[.B,.A]	dst	算术左移指令	ADDX[.B,.A] dst,dst
15	RLCX[.B,.A]	dst	带进位位循环左移指令	ADDCX[.B,.A] dst,dst
16	SBCX[.B,.A]	dst	减借位位指令	SUBCX[.B,.A] #0,dst
17	TSTA	Rdst	测试指令	CMPA #0,Rdst
18	TSTX[.B,.A]	dst	测试指令	CMPX[.B,.A] #0,dst
19	POPX	dst	出栈指令	MOVX[.B,.A] @SP+,dst

3.4.2　指令集扩展对程序设计的影响

尽管在对 MSP430 的指令集扩展时，充分考虑到与原有指令集的兼容性，但对于程序编写人员来说，还是需要了解一下对具体程序设计带来的影响。

1. 对汇编语言程序的影响

由于汇编语言是面向底层的低级语言，对硬件结构及其指令集的改动最为敏感。因此，指令集扩展对汇编程序设计影响最大。

从指令层次来看，MSP430 指令集是 MSP430X 指令集的真子集。在访问 64KB 以内的存储空间时，优先使用 MSP430 指令集中的指令进行程序设计，这样做是为了增强程序的兼容性。当超过 64KB 时再使用扩展的指令。

在程序设计层面，程序的启动地址应安排在低 64KB 空间。主程序原则上可以存在程序存储空间的任何位置，但考虑到兼容性问题，一般优先将其置于低地址空间。同时务必保证所有中断服务程序都处在低 64K 存储区内；否则将会出现错误。当然，这一过程一般默认是由汇编器或编译器完成的。

2. 对 C 语言程序的影响

相对汇编语言，对于 C 语言程序设计的影响微乎其微。这是因为，指令集的扩展主要影响指令的使用以及寻址方式和寄存器间的操作。而这些操作对于 C 语言来说都是透明的，也就是说，C 编译器可以对所使用的指令自动识别。因此，对 C 语言程序的编写并无影响。

指令集扩展后，对 C 语言程序带来一些好处。如 C 语言的执行周期数减少，中断响应时间进一步缩短，编译生成的代码体积更小。

3.5 MSP430 单片机汇编语言基础

汇编语言(Assembly Language)是面向机器的程序设计语言。相对枯燥难懂的机器语言,汇编语言更易于读写、易于调试和修改。优秀的汇编语言程序,在汇编后的代码执行速度比高级语言更快,占内存空间少。

3.5.1 伪指令

在汇编语言程序设计中,指令系统中的指令是程序的主体。前面已对 MSP430 的指令集进行了较为详细的介绍,这里不再赘述。除了指令系统中的指令外,为了便于编写程序,还定义了一些辅助指令,它们既不控制机器的操作,也不被汇编成机器代码,只能被汇编程序识别并指导汇编如何进行。伪指令的主要作用有初始化存储器、汇编条件块、定义全局变量、将代码和数据汇编到规定的段中等。下面介绍 IAR 汇编编译器中常用的汇编伪指令。

1. 模块控制伪指令

该类伪指令主要说明一个程序模块的开始与结束,并给模块命名和指示其类型。

NAME(PROGRAM)表示一个程序模块的开始; MODULE(LIBRARY)表示一个库模块的开始; ENDMOD 表示当前汇编模块的结束; END 表示一个汇编文件的结束。

2. 段控制伪指令

该类伪指令主要说明程序代码与数据在存储器中是如何分配和管理的。ASEG 表示一个绝对段的开始; RSEG 表示一个可重定位段(相对段)的开始; STACK 表示定义一个堆栈段; COMMON 表示一个定义公共段; ORG 表示设置特定的定位指针; ALIGN 表示通过插入一些填充字节校准 PC; EVEN 表示通过插入一些填充字节使 PC 对准偶地址。

3. 数值分配伪指令

SET(ASSIGN,VAR)表示给一个变量赋予一个临时值; EQU(=)表示在当前模块内赋予一个永久有效的值; DEFINE 表示定义一个在该文件内有效的值。

4. 数据定义伪指令

DB 表示定义的是字节数据; DW 表示定义的是字数据; DL 表示定义的是 32 位数据; DF 表示定义的是 32 位浮点型数据; DS n 表示分配 n 个连续的字节空间。

5. 符号控制伪指令

EXTERN(IMPORT)表示引入外部符号; PUBLIC(EXPORT)表示输出符号。

3.5.2 汇编语言程序设计基础

汇编语言程序设计就是在既定汇编语言书写规则下,根据某种算法或规则在伪指令的辅助下将相关单片机指令有机地组织在一起。经检查无误后经过机器汇编成机器语言,并下载到单片机内执行的过程。在进行单片机汇编语言程序设计时,由于单片机自身资源相对匮乏,执行效率显得尤为重要,一般要求编写程序时要尽量节省数据单元,减少程序代码的长度,提高执行速度。

在汇编语言程序设计中，模块化程序设计是十分必要的。为实现模块化程序设计，下面的 4 种程序结构是经常用到的。

1. 顺序结构

顺序结构是一种最简单、最基本的程序结构。以该结构编写的程序按照既定顺序一条一条地执行指令，整个过程程序流向不变，如图 3.9 所示。顺序结构可以独立使用构成一个简单的完整程序，常见的输入、计算、输出三部曲的程序就是顺序结构。不过大多数情况下顺序结构都是作为程序的一部分，与其他结构（如分支结构中的复合语句、循环结构中的循环体）等一起构成一个复杂的程序。

例 3.2　已知某 16 位负数 x 以补码形式存放于 0x0400 处，试编程求该数的绝对值 $|x|$，并存放到 0x0410 处。

解　由题意知，程序如下。

```
#include <msp430x26x.h>

X          EQU    0400H          ;定义 X
Y          EQU    0410H          ;定义 Y
           MOV    X, R8
           DEC    R8             ;负值
           INV    R8
POS        MOV    R8,Y
           JMP    $
           END
```

2. 分支结构

顺序结构的程序虽然简单易读，但不能处理判断再选择的情况。对于要先做判断再选择的问题就要使用分支结构。分支结构的执行是根据一定的条件选择执行路径，并不是严格按照语句出现的物理顺序执行，如图 3.10 所示。在汇编语言中分支结构中条件选择是通过条件转移指令实现的。编写分支结构程序的关键在于正确使用转移指令。分支结构适合于带有逻辑或关系比较等条件判断的计算场合，还可将分支结构细分为单分支结构和多分支结构。

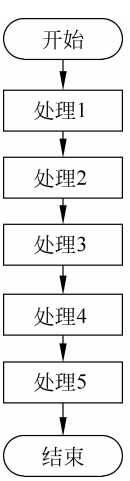


图 3.9　顺序结构示意图

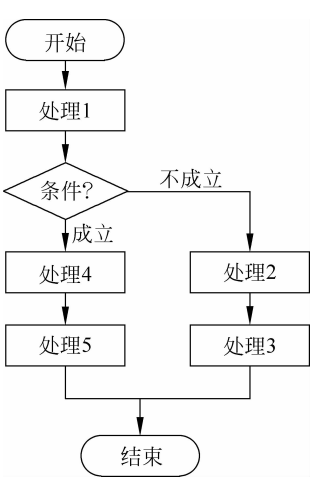


图 3.10　分支结构示意图

例 3.3 已知 RAM 中 0x0400 处变量 x，变量 y 与 x 的关系式如下。

$$y=\begin{cases}10, & x>0\\0, & x=0\\-10, & x<0\end{cases}$$

试编程计算变量 y 的值，并存入 0x0500 处。

解 由题意知，程序如下。

```
#include <msp430x26x.h>

X          EQU    0400H           ;定义 X
Y          EQU    0500H           ;定义 Y
           MOV    X, R8
           MOV    #0,R9
           CMP    R9, R8
           JZ     DONE            ;等于零
           JL     NEG             ;负值
           MOV    #000AH, R8      ;正值
           JMP    DONE
NEG        MOV    #0FFF6H, R8
DONE       MOV    R8, Y
           JMP    $
           END
```

3. 循环结构

在程序设计中常会遇到反复执行某些指令，如数据的传输中就需要反复执行数据传输指令，这时可以使用循环结构。循环结构最大的优点是，减少源程序重复书写的工作量，使程序更加简洁易读。因此它十分适合用来描述重复执行某段算法的场合。

循环结构一般由循环初始化、循环处理、循环控制和循环结束四部分组成。循环初始化部分主要负责与循环过程有关的工作单元的初始化，如循环次数计数器初始化、地址指针的初始化等；循环处理部分位于循环体的内部，负责循环结构所要做的处理工作。该部分需要循环执行。一般要求尽可能简练，以提高执行速度；循环控制部分负责控制循环体循环的次数，如循环计数器的修改及条件判断与转移。

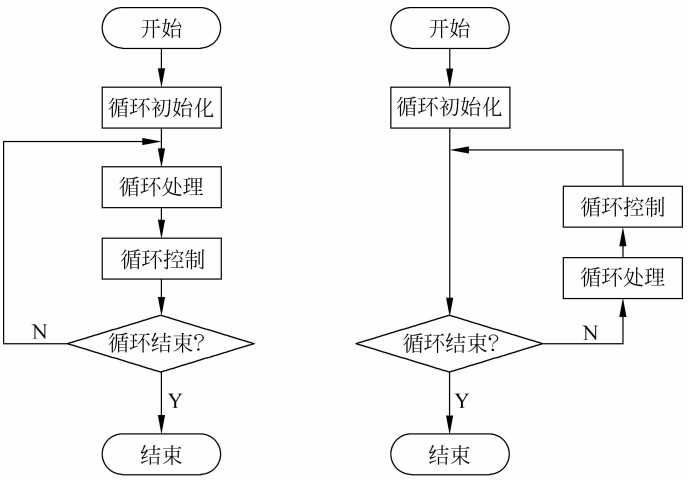
循环结构有两种形式：一种是先判断循环条件后进行循环处理，如图 3.11(a)所示；另一种是先进行一次循环处理再进行条件判断，如图 3.11(b)所示。这两种形式均有自己的适合场合。

例 3.4 将 Flash 中 0x4090 处的 20 个 16 位数移至 RAM 中的 0x0200 处。

解：由题意知，程序如下。

```
#include <msp430x26x.h>

           MOV    #0014H,R5       ;循环次数
           MOV    #4090H,R6       ;源地址
           MOV    #0200H,R7       ;目标地址
LOOP       MOV    @R6+,0(R7)
           INCD   R7
           DEC    R5
           JNZ    LOOP
           JMP    $
           END
```



(a) 先执行后判断　　(b) 先判断后执行

图 3.11　循环结构的两种形式

4. 子程序设计

通常将程序设计中重复的程序段单独列出来，并按一定的格式编写成子程序。子程序是指完成某一确定任务，并能被其他程序反复调用的程序段。使用子程序可以有效地缩短整个程序的长度，有利于实现模块化程序结构。子程序设计方便开发、调试和调用，子模块可反复被调用，避免重复输入。

在模块化程序设计中，子程序的划分应遵循高独立性、低耦合性、长度控制的原则，具体如下。

(1) 子程序编写应做到功能简单、明确，内容简明、易懂，任务清楚、明确，以便易于修改。

(2) 子程序之间应尽可能独立，即两者之间的联系及互相影响尽可能地减少，尽可能减少调用关系和数据交换关系。

(3) 子程序的长度要适中。一般长度为 20～100 条的范围较合适。程序太长时，分析和调试比较困难，失去了模块化程序结构的优越性；过短则程序间的连接太复杂，信息交换太频繁。

例 3.5　现有两个 16 位无符号数 a 和 b，它们分别存在 0x0300 和 0x0310 处。请编写子程序将 a、b 中最大者存入 0x0320 处。要求子程序的输入参数分别存入 R8、R9 中，比较的结果存入 R10 中。

解　由题意知，程序如下。

```
#include <msp430x26x.h>

A          EQU     0300H           ;定义 A
B          EQU     0310H           ;定义 B
D          EQU     0320H           ;定义 D
           MOV     A,R8            ;(A)→R8
           MOV     B,R9            ;(B)→R9
           CALL    #GET_MAX
           MOV     R10, D          ;R10→(D)
           JMP     $
```

```
GET_MAX   CMP    R8,R9
          JL     Label1          ; (R9)<(R8)
          MOV    R9, R10
          RET
Label1    MOV    R8, R10
          RET
          END
```

3.5.3 汇编语言与高级语言

汇编语言程序是基于单片机指令系统编写的,是最接近机器语言的。具有占用资源少、程序执行效率高的优点。但现在使用汇编语言进行程序设计的人却越来越少,其原因是多方面的,大致可归纳:①需要掌握的与硬件相关的底层知识过多,如需要熟练掌握指令的寻址方式和指令系统中指令的使用方法和技巧;②开发难度和工作量大,汇编语言程序书写量较大,尤其是对于精简指令集系统。以 MSP430 系列单片机为例,它使用的是 16 位精简指令系统。尽管需要记忆的指令少了,但是对于汇编程序设计来说,设计难度和代码的编写量都增加了。例如,对于乘法运算,原本需要一个指令就能完成的事,现在便需要多条语句组合完成。在编写复杂程序时,局限性更加明显。由于难度和工作量大,势必会增加开发成本,延长开发周期,而这些恰恰是大家不愿看到的;③汇编语言依赖于具体处理器,移植性较差。由于不同 CPU 间的指令系统有所差异,其汇编语言程序也会有区别,所以不利于不同 CPU 间的程序移植;④高级语言的进步,但随着编译技术不断提高,部分高级语言(如C 语言)在编译后代码已逼近汇编的执行效率,使得汇编语言程序在执行效率方面的优点正在被逐渐弱化。

尽管如此,汇编语言仍然是计算机底层设计程序员必须了解的语言,在某些行业与领域,汇编语言是必不可少的,有些地方甚至非它不可。对于嵌入式系统开发初学者来说,掌握汇编语言基础知识也是必要的。因为汇编语言中一条指令就对应一个机器码,每一步执行什么动作都很清楚,并且程序大小和堆栈调用情况都容易控制,调试起来也比较方便。更重要的是,汇编语言可以从更深层次了解单片机程序的运行机制与工作原理。

相对于汇编语言,以 C 语言为代表的高级语言却大行其道,越来越受到人们的欢迎。

C 语言是一种编译型程序设计语言,它兼顾了多种高级语言的特点,并具备汇编语言的功能。C 语言有功能丰富的库函数、运算速度快、编译效率高、有良好的可移植性,而且可以直接实现对系统硬件的控制。作为一种结构化程序设计语言,C 语言支持当前程序设计中广泛采用的自顶向下结构化程序设计技术。C 语言程序还具有完善的模块程序结构,为软件开发中采用模块化程序设计方法提供了有力的保障。

因此,利用 C 语言编写软件尤其大型软件,会大大缩短开发周期,且明显地增加软件的可读性,便于改进和扩充。从单片机程序设计的发展来看,用 C 语言进行单片机程序设计是单片机开发与应用的必然趋势。正因为这种趋势,目前大多数单片机均有自己的 C 编译器以支持各自单片机的 C 语言程序开发。目前 C 语言已成为软件开发的一个主流语言。

当然,这不是说 C 语言是完美的。实际上,C 语言在用于单片机程序设计时也存在一定缺点。最为突出的是,利用 C 语言生成的机器代码体积大,很容易出现程序空间不够、堆栈溢出等问题。但随着技术进步和工艺改进,单片机中存储资源越来越大,C 语言在这方面的劣势也逐步被削弱。

总之,汇编语言与 C 语言两者各有优、缺点,在进行小型程序开发时两者均可采用。若

进行较大程序开发时，C 语言的优势更加明显。

习　　题

3-1　为什么说 MSP430 指令集属于精简指令集？

3-2　简述仿真指令与内核指令的区别。

3-3　简述仿真指令的作用。

3-4　MSP430 指令集中有哪几种指令格式？它们各自的特点分别是什么？

3-5　MSP430 共有几种寻址方式？它们在使用时有何不同？

3-6　简述寄存器寻址与寄存器间接寻址的不同。

3-7　常数发生器产生的常数与立即数在进行数据传输时在执行效率上有什么不同？

3-8　简述符号寻址与绝对寻址之间的区别。

3-9　列出下面指令中源操作数的寻址方式。

(1) MOV R4，R8　　(2) MOV　&2036H，R9

(3) MOV 2(R5)，7(R6)　　(4) MOV　@R5，R4

(5) MOV @R5＋，5(R3)　　(6) MOV　＃2011，R9

3-10　MOV ＃2012H，&2011H，执行指令后，(2010H)、(2011H)、(2012H)中的内容发生何种变化？

3-11　MSP430 是否支持位地址寻址？如何进行位操作？

3-12　若 SP＝1002H，(SP)＝1234H，R5＝5678h，则 POP　R5 执行后，SP、R5 中的内容有何变化。

3-13　影响指令周期的因素有哪些？并举例说明。

3-14　简述 MSP430X 指令集的产生背景及其特点。

3-15　指令集扩展对程序设计有何影响？

3-16　什么是伪指令？伪指令的作用是什么？

3-17　列出常见的汇编程序结构，并指出各自的特点。

3-18　已知 RAM 内 0x0300 处连续存放某班 60 名学生成绩(百分制)，试利用汇编语言编程完成以下功能：

(1) 求出平均成绩，并存入 0x400 处。

(2) 将成绩分成 A(＞89)、B(80～89)、C(70～79)、D(60～69)和 E(0～59)5 个等级，并计算每一等级内的学生个数，并将其依次存放到 0x0410～0x0414 处。

3-19　已知 RAM 内有一块连续存放的数据，起始地址为 0x0332，数据块大小为 50。试利用汇编语言编程实现块内数据降序排列。

3-20　已知 RAM 内 0x0600 处存放有一个 16 位无符号数，试利用汇编语言编程实现将其二进制数转换成十进制数，并将该十进制数转换成相应的 ASCII 码。

3-21　已知系统时钟主频为 1MHz，利用汇编语言编程实现能延时 10ms 的子程序。

3-22　在汇编语言程序中，叙述指令 RET 与指令 RETI 的用途。

3-23　汇编语言程序的优点和缺点是什么？

3-24　为什么 C 语言会成为目前单片机开发设计中的主要使用语言之一？