第3章　51单片机编程语言

3.1　单片机的编程语言概述

51单片机的编程语言可以是汇编语言，也可以是高级语言，如由C语言演变而成的C51语言等。汇编语言产生的目标代码短，占用的存储空间小，执行速度快，能充分发挥单片机的硬件功能。但对于复杂的应用来讲，使用汇编语言编程复杂，程序的可读性和可移植性不强。高级语言产生的目标代码长，占用的存储空间大，执行速度慢。但这是相对于汇编语言来讲的，其实C语言在大多数情况下的机器代码生成效率和汇编语言相当，但可读性和可移植性却远远超过汇编语言，编程效率也大大高于汇编语言。

可见，汇编语言和高级语言各有优缺点，在应用中应根据实际情况选用。如果应用系统的存储空间比较小，且对实时性的要求很高，则应选用汇编语言编程；如果系统的存储空间比较大，且对实时性的要求不是很高，则C51语言是理想的编程语言。如果系统中有部分模块对实时性的要求很高，而其他模块对实时性的要求不是很高，则可以将两种语言结合，程序的主体部分使用C51编程，对实时性要求高的模块用汇编语言编程，然后将汇编语言程序模块嵌入C51语言程序中。

用高级语言和汇编语言编写的源程序都必须转换为目标程序(机器语言)，单片机才能执行。目前很多公司都将编辑器、汇编器、编译器、连接/定位器、符号转换程序做成了软件包，称为集成开发环境(Integrated Developing Environment，IDE)，如Keil μVision、Silicon Laboratories IDE等。用户进入集成开发环境，编辑好程序后，只需单击相应的菜单或快捷工具按钮就可以完成汇编/编译、连接/定位、程序下载等功能，还可以在线跟踪调试。

可见，汇编语言和高级语言都是开发单片机应用系统必须掌握的编程语言。本书以目前流行的C51为主要编程语言，同时兼顾传统的汇编语言。本章首先简要介绍51单片机的寻址方式和指令系统，然后通过几个编程实例介绍汇编语言程序设计中的几种基本结构程序的设计方法和技巧，最后重点讲述C51语言的基础知识及编程方法。在集成开发环境中调试汇编语言程序和C51语言程序的方法见本书第8章。

3.2　CIP-51指令介绍

CIP-51系统控制器的指令集与标准MCS-51™指令集完全兼容，可以使用标准8051的开发工具开发CIP-51的软件。所有的CIP-51指令共111条，在二进制机器码和功能上

与 MCS-51™产品完全等价(如操作码、寻址方式、对 PSW 标志的影响等),但是指令时序与标准 8051 不同。

在很多的 8051 产品中,机器周期和时钟周期是不同的,机器周期的长度在 2～12 个时钟周期之间,如 Intel 公司的 MSC-51 单片机的机器周期的长度为 12 个时钟周期。但是 CIP-51 只基于时钟周期,所有指令时序都以时钟周期计算,也就是说,CIP-51 的机器周期与时钟周期相等。由于 CIP-51 采用了流水线结构,大多数指令执行所需的时钟周期数与指令的字节数一致。条件转移指令在不发生转移时的执行周期数比发生转移时少一个。附录 A 给出了 CIP-51 指令一览表,包括每条指令的助记符、字节数和时钟周期数。

3.2.1 寻址方式

寻址方式就是根据指令中给出的地址寻找真实操作数地址的方式。8051 单片机的寻址方式有以下 7 种。

1. 寄存器寻址

寄存器寻址时,指令中地址码给出的是某一通用寄存器的编号,寄存器的内容为操作数。例如指令

```
MOV     A,  R0    ;A←(R0)
```

8051 可用寄存器寻址的空间是 R0～R7,ACC,DPTR,B。

2. 直接寻址

直接寻址时,指令中地址码部分直接给出了操作数的有效地址。例如指令

```
MOV     A,  4FH    ;A←(0x4F)
```

可用于直接寻址的空间是内部 RAM 低 128B(包括其中的可位寻址区)、特殊功能寄存器。

3. 寄存器间接寻址

寄存器间接寻址时,指令中给出的寄存器的内容为操作数的地址,而不是操作数本身,即寄存器为地址指针。例如指令

```
MOV  A,@R1  ;A←((R1))
```

8051 中可以用 R0 或 R1 间接寻址片内或片外 RAM 的 256B 范围,可以用 DPTR 或 PC 间接寻址 64KB 外部 RAM 或 ROM。

4. 立即寻址

立即寻址时,指令中地址码部分给出的就是操作数。即取出指令的同时立即得到了操作数。例如指令

```
MOV  A,#6FH  ;A←0x6F
```

5. 变址寻址

变址寻址时,指定的变址寄存器的内容与指令中给出的偏移量相加,所得的结果作为操作数的地址。例如指令

```
MOVC  A,@A+DPTR  ;A←((A)+(DPTR))
```

不论是用 DPTR 还是 PC 来提供基址指针,变址寻址方式都只适用于 8051 的程序存储器,通常用于读取数据表。

6. 相对寻址

相对寻址时,由程序计数器 PC 提供的基地址与指令中提供的偏移量 rel 相加,得到操作数的地址。这时指出的地址是操作数与现行指令的相对位置。例如指令

```
SJMP  rel  ;PC←(PC) + 2 + rel
```

7. 位寻址

位寻址时,操作数是二进制数的某一位,指令中使用位地址指明要操作的位,例如指令

```
SETB  bit  ;(bit)←1
```

8051 可用于位寻址的空间是内部 RAM 的可位寻址区和 SFR 区中的字节地址可以被 8 整除(即地址以 0 或 8 结尾)的寄存器所占空间。

3.2.2 51 指令集

8051 单片机的指令按其功能可分为以下 5 大类。

- 数据传送指令。
- 算术运算指令。
- 逻辑运算指令。
- 布尔运算指令。
- 程序分支指令。

1. 数据传送指令

数据传送指令是将数据由来源地址传送至目的地址,除 XCH 与 XCHD 外,数据传送指令均不会改变来源地址数据内容。8051 的数据传送指令共分以下 7 种,下面分别给予叙述。

MOV(一般传送指令):将来源地址所指定的数据复制到目的地址所指定的单元。此种操作不影响任何标志,这种指令主要用于内部 RAM 及寄存器之间的数据传送。共有以下 16 种寻址格式:

```
MOV    A,Rn            ;A←(Rn)
MOV    A,direct        ;A←(direct)
MOV    A,@Ri           ;A←((Ri))
MOV    A,#data         ;A← data
MOV    Rn,A            ;Rn←(A)
MOV    Rn,direct       ;Rn←(direct)
MOV    Rn,#data        ;Rn← data
MOV    direct, A       ;direct←(A)
MOV    direct, Rn      ;direct←(Rn)
MOV    direct,direct   ;direct←(direct)
MOV    direct,@Ri      ;direct←((Ri))
MOV    direct,#data    ;direct← data
MOV    @Ri,A           ;(Ri)←(A)
MOV    @Ri,direct      ;(Ri)←(direct)
MOV    @Ri,#data       ;(Ri)← data
MOV    DPTR,#data16    ;DPTR← data16
```

MOVC(查表指令):在程序区段复制一字节数据到 A 累加器,此种指令经常用于查表程序中。共有以下 2 种寻址格式:

```
MOVC    A,@A+DPTR       ;A←((A)+(DPTR))
MOVC    A,@A+PC         ;PC←(PC)+1; A←((A)+(PC))
```

MOVX(外部数据存储器读写指令):在外部数据存储器区段复制一字节数据到 A 累加器,或将 A 累加器中的数据复制到被寻址的外部数据存储器区段。共有以下 4 种寻址格式:

```
MOVX    A,@DPTR         ;A←((DPTR))
MOVX    A,@Ri           ;A←((Ri))
MOVX    @DPTR,A         ;(DPTR)←(A)
MOVX    @Ri,A           ;(Ri)←(A)
```

注意,MOVX 指令的寻址地址有 8 位和 16 位两种,16 位时是通过 DPTR 寄存器中的 16 位地址来寻找 64KB 存储器范围的单元。8 位时是通过 Ri 中的内容作为地址的低 8 位,特殊功能寄存器 EMIOCN 的内容作为地址的高 8 位来寻找 64KB 存储器范围的单元。具体见 2.2.5 节。

PUSH(压栈指令):将数据推进堆栈,执行指令时先将 SP 加 1,再将数据压入堆栈。指令格式:

```
PUSH    direct          ;SP←(SP)+1; (SP)←(direct)
```

POP(出栈指令):将数据由堆栈顶端取出,执行指令时先将数据取出,再将 SP 减 1。

```
POP     direct          ;direct←(SP); SP←(SP)-1
```

XCH(字节交换指令):将累加器内数据与指定寻址的内容数据互换。有如下 3 种寻址格式:

```
XCH     A,Rn            ;(A)⟺(Rn)
XCH     A,direct        ;(A)⟺(direct)
XCH     A,@Ri           ;(A)⟺((Ri))
```

XCHD(半字节交换指令):将累加器内低 4 位数据与指定寻址的内容低 4 位数据互换。指令格式:

```
XCHD    A,@Ri           ;(A)3～0 ⟺((Ri))3～0
```

2. 算术运算指令

算术运算指令包括四则运算(加减乘除)、加 1、减 1 以及 BCD 码十进制加法调整指令等,共有 8 种指令。分别叙述如下。

ADD(加指令):将指定寻址的内容加上累加器内容,并将结果存入累加器中,此种运算会影响特殊功能寄存器 PSW。有如下 4 种寻址格式:

```
ADD     A,Rn            ;A←(A)+(Rn) (n: 0～7)
ADD     A,direct        ;A←(A)+(direct)
ADD     A,@Ri           ;A←(A)+((Ri))
ADD     A,#data         ;A←(A)+data
```

ADDC(带进位加指令)：将指定寻址的内容加上累加器内容，再加上进位标志 CY，并将结果存入累加器中。有如下 4 种寻址格式：

```
ADDC    A,Rn            ;A←(A)+(Rn)+(CY)
ADDC    A,direct        ;A←(A)+(direct)+(CY)
ADDC    A,@Ri           ;A←(A)+((Ri))+(CY)
ADDC    A,#data         ;A←(A)+data+(CY)
```

以上两个加运算后，如果位 7 有进位，则特殊功能寄存器 PSW 的进位标志 CY 被设定为 1，否则为 0；如果位 3 有进位，则特殊功能寄存器的辅助进位标志 AC 被设定为 1，否则为 0。此种运算也会影响 OV(溢出标志)，当溢出标志为 1 时，表示两个带符号正数相加变为负数(溢出)或两个带符号负数相加变为正数(溢出)。

DA(十进制调整指令)：将 A 累加器内容作 BCD 码调整。指令格式：

```
DA      A               ;若 AC=1 或 A3～0>9,则 A←(A)+0x06
                        ;若 CY=1 或 A7～4>9,则 A←(A)+0x60
```

DEC(减 1 指令)：将指定寻址的内容减 1。注意，这种运算不影响任何标志，且当递减至 0x0 时再递减变为 0xFF，共有以下 4 种寻址格式：

```
DEC     A               ;A←(A)-1
DEC     direct          ;direct←(direct)-1
DEC     @Ri             ;(Ri)←((Ri))-1
DEC     Rn              ;Rn←(Rn)-1
```

DIV(除法指令)：将累加器 A 和寄存器 B 作无符号数相除。指令格式：

```
DIV     AB              ;(A)/(B),A←商、B←余数
```

INC(加 1 指令)：将指定寻址的内容加 1。注意，此种运算不影响任何标志，且当递增至 0xFF(8b)或 0xFFFF(DPTR)时，再递增变为 0。共有以下 5 种寻址格式：

```
INC     A               ;A←(A)+1
INC     direct          ;direct←(direct)+1
INC     @Ri             ;(Ri)←((Ri))+1
INC     Rn              ;Rn←(Rn)+1
INC     DPTR            ;DPTR←(DPTR)+1
```

MUL(乘法指令)：将累加器 A 和寄存器 B 作无符号数相乘。此结果产生一个 16 位数据。注意，若运算后 B 寄存器不为 0，则溢出标志被设为 1。指令格式：

```
MUL     AB              ;(A)×(B),A←乘积低字节、B←乘积高字节
```

SUBB(带借位减法指令)：将累加器 A 的内容减去指定寻址的内容，再减去进位标志 CY，并将结果存入累加器中。共有 4 种寻址格式：

```
SUBB    A,Rn            ;A←(A)-(Rn)-(CY)
SUBB    A,direct        ;A←(A)-(direct)-(CY)
SUBB    A,@Ri           ;A←(A)-((Ri))-(CY)
SUBB    A,#data         ;A←(A)-data-(CY)
```

如果位 7 有借位，则进位标志 CY 被设定为 1；如果位 3 有借位，则辅助进位标志 AC

被设定为 1。此种运算会影响溢出标志，当溢出标志为 1 时，表示一个带符号正数减一个带符号负数变为负数(溢出)，或一个带符号负数减一个带符号正数变为正数(溢出)。

3. 逻辑运算指令

8051 的逻辑运算指令包括 AND、OR、XOR、NOT、左/右移位、清除、高/低 4 位对调等指令。分别叙述如下。

ANL(**逻辑与指令**)：将两个被指定寻址的内容相对应的位分别做逻辑与运算，并将结果存入目的地址中。共有以下 6 种寻址格式：

```
ANL     A,Rn            ;A←(A)∧(Rn)
ANL     A,direct        ;A←(A)∧(direct)
ANL     A,@Ri           ;A←(A)∧((Ri))
ANL     A,#data         ;A←(A)∧data
ANL     direct,A        ;direct←(direct)∧(A)
ANL     direct,#data    ;direct←(direct)∧data
```

逻辑与指令常用于清 0 字节的某些位。欲清 0 的位用“0”去“与”，欲保留的位用“1”去“与”。

ORL(**逻辑或指令**)：将两个被指定寻址的内容相对应的位分别做逻辑或运算，并将结果存入目的地址中。共有以下 6 种寻址格式：

```
ORL     A,Rn            ;A←(A)∨(Rn)
ORL     A,direct        ;A←(A)∨(direct)
ORL     A,@Ri           ;A←(A)∨((Ri))
ORL     A,#data         ;A←(A)∨data
ORL     direct,A        ;direct←(direct)∨(A)
ORL     direct,#data    ;direct←(direct)∨data
```

逻辑或指令常用于置 1 字节的某些位。欲置 1 的位用“1”去“或”，欲保留的位用“0”去“或”。

XRL(**逻辑异或指令**)：将两个被指定寻址的内容相对应的位分别做逻辑异或运算，并将结果存入目的地址中。共有以下 6 种寻址格式：

```
XRL     A,Rn            ;A←(A)⊕(Rn)
XRL     A,direct        ;A←(A)⊕(direct)
XRL     A,@Ri           ;A←(A)⊕((Ri))
XRL     A,#data         ;A←(A)⊕data
XRL     direct,A        ;direct←(direct)⊕(A)
XRL     direct,#data    ;direct←(direct)⊕data
```

逻辑异或指令常用于去取反字节的某些位。欲取反的位用“1”去“异或”，欲保留的位用“0”去“异或”。

CLR(**累加器清 0 指令**)：将累加器 A 内容清除为 0。指令格式：

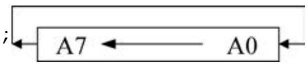
```
CLR     A               ;A←0
```

CPL(**累加器取反指令**)：将累加器 A 内容每位求反。指令格式：

```
CPL     A               ;A←(A)的每位求反，即 0→1，1→0
```

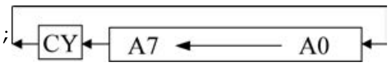
RL(累加器左移指令):将累加器 A 的内容左环移一位。指令格式:

```
RL      A              ;
```



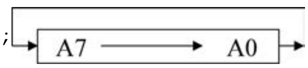
RLC(累加器带进位左移指令):将累加器 A 的内容与进位左环移一位。指令格式:

```
RLC     A              ;
```



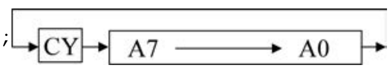
RR(累加器右移指令):将累加器 A 的内容右环移一位。指令格式:

```
RR      A              ;
```



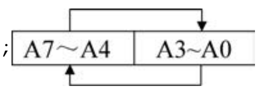
RRC(累加器带进位右移指令):将累加器 A 的内容与进位右环移一位。指令格式:

```
RRC     A              ;
```



SWAP(4 位互换指令):将累加器 A 的高 4 位与低 4 位互换。指令格式:

```
SWAP    A              ;
```



4. 布尔运算指令

8051 的布尔运算指令是比较特殊的一种运算,它可针对可位寻址的内部 RAM、寄存器进行操作。进位标志 CY 就是这种操作运算的累加器。布尔运算指令包括清除、置位、取反、与、或及位传送等指令,共有 6 种。分别叙述如下。

CLR(清除):可清除 CY 标志或任一可位寻址的位。指令格式:

```
CLR     C              ;CY←0
CLR     bit            ;bit←0
```

SETB(置位):可设定 CY 标志或任一可位寻址的位,使其为 1。指令格式:

```
SETB    C              ;CY←1
SETB    bit            ;bit←1
```

CPL(取反):可取反 CY 标志或任一可位寻址的位,使其 1 变 0、0 变 1。指令格式:

```
CPL     C              ;CY←(CY)的反,即 0→1,1→0
CPL     bit            ;bit←(bit)的反,即 0→1,1→0
```

ANL(与):两个位作逻辑与。指令格式:

```
ANL     C,bit          ;CY←(CY)∧(bit)
ANL     C,/bit         ;CY←(CY)∧(bit)的反
```

ORL(或):两个位作逻辑或。

```
ORL     C,bit          ;CY←(CY)∨(bit)
ORL     C,/bit         ;CY←(CY)∨(bit)的反
```

MOV(位传送指令):两个操作数中要有一个为 CY 标志,另一个为可位寻址的直接位

地址。指令格式：

```
MOV    C,bit          ;CY←(bit)
MOV    bit,C          ;bit←(CY)
```

5. 程序分支指令

单片机在执行程序时，程序指令一般是依顺序执行的，而程序分支指令则是用来改变程序指令执行的顺序，使程序设计更加方便。程序分支指令有无条件跳转、有条件跳转、调用子程序、子程序返回、中断返回等指令。分别叙述如下。

JC（有进位跳）：判断 CY 标志等于 1 然后跳转。指令格式：

```
JC     rel            ;若 CY = 1 则 PC←(PC) + 2 + rel
                      ;若 CY = 0 则 PC←(PC) + 2
```

JNC（无进位跳）：判断 CY 标志等于 0 然后跳转。指令格式：

```
JNC    rel            ;若 CY = 0 则 PC←(PC) + 2 + rel
                      ;若 CY = 1 则 PC←(PC) + 2
```

JB（位 1 跳）：判断给出位等于 1 然后跳转。给出的位应该是可位寻址。指令格式：

```
JB     bit,rel        ;若 bit = 1 则 PC←(PC) + 3 + rel
                      ;若 bit = 0 则 PC←(PC) + 3
```

JNB（位 0 跳）：判断给出位等于 0 然后跳转。给出的位应该是可位寻址。指令格式：

```
JNB    bit,rel        ;若 bit = 0 则 PC←(PC) + 3 + rel
                      ;若 bit = 1 则 PC←(PC) + 3
```

JBC（位 1 跳并清除）：判断给出位等于 1 然后跳转，并清除此位为 0。给出的位应该是可位寻址。指令格式：

```
JBC    bit,rel        ;若 bit = 1 则 PC←(PC) + 3 + rel 且 bit←0
                      ;若 bit = 0 则 PC←(PC) + 3
```

JZ（A 为 0 跳）：判断累加器等于 0，然后跳转。指令格式：

```
JZ     rel            ;若(A) = 0 则 PC←(PC) + 2 + rel
                      ;若(A)≠0 则 PC←(PC) + 2
```

JNZ（A 为非 0 跳）：判断累加器不等于 0，然后跳转。指令格式：

```
JNZ    rel            ;若(A)≠0 则 PC←(PC) + 2 + rel
                      ;若(A) = 0 则 PC←(PC) + 2
```

CJNE（比较不相等跳）：比较两个操作数，若不相等就跳转。共有 4 种寻址格式：

```
CJNE   A,direct,rel   ;若(A)≠(direct)则 PC←(PC) + 3 + rel,CY 按规则形成
                      ;若(A) = (direct)则 PC←(PC) + 3,CY = 0
CJNE   A,#data,rel    ;若(A)≠ data 则 PC←(PC) + 3 + rel,CY 按规则形成
                      ;若(A) =  data 则 PC←(PC) + 3,CY = 0
CJNE   Rn,#data,rel   ;若(Rn)≠ data 则 PC←(PC) + 3 + rel,CY 按规则形成
                      ;若(Rn) =  data 则 PC←(PC) + 3,CY = 0
CJNE   @Rn,#data,rel  ;若((Rn))≠ data 则 PC←(PC) + 3 + rel,CY 按规则形成
```

```
                          ;若((Rn)) = data 则 PC←(PC) + 3,CY = 0
```

DJNZ(减 1 不为 0 跳):若寻址内容不等于 0 则跳转。指令格式:

```
DJNZ    Rn,rel            ;Rn←(Rn) - 1
                          ;若(Rn)≠ 0 则 PC←(PC) + 2 + rel
                          ;若(Rn) = 0 则 PC←(PC) + 2
DJNZ    direct,rel        ;direct←(direct) - 1
                          ;若(direct)≠ 0 则 PC←(PC) + 3 + rel
                          ;若(direct) = 0 则 PC←(PC) + 3
```

LJMP(长跳转指令):64KB 内存范围内无条件跳转。指令格式:

```
LJMP    addr16            ;PC← addr16
```

AJMP(短跳转指令):2KB 内存范围内无条件跳转。指令格式:

```
AJMP    addr11            ;PC←(PC) + 2,PC10～0← addr11
```

SJMP(相对跳转指令):在此指令前 128B 或后 128B 范围内跳转。指令格式:

```
SJMP    rel               ;PC←(PC) + 2,PC←(PC) + rel
```

JMP(散转指令):跳转至@A+DPTR 所指定的地址。指令格式:

```
JMP     @A + DPTR         ;PC←(A) + (DPTR)
```

LCALL(长调用指令):64KB 内存范围内子程序调用。指令格式:

```
LCALL   addr16            ;PC←(PC) + 3
                          ;SP←(SP) + 1 ,(SP)←PC7～0
                          ;SP←(SP) + 1 ,(SP)←PC15～8
                          ;PC← addr16
```

ACALL(短调用指令):2KB 内存范围内子程序调用。指令格式:

```
ACALL   addr16            ;PC←(PC) + 2
                          ;SP←(SP) + 1 ,(SP)←PC7～0
                          ;SP←(SP) + 1 ,(SP)←PC15～8
                          ;PC10～0← addr11
```

RET(子程序返回):指令格式:

```
        RET               ;PC15～8←((SP)),SP←(SP) - 1
                          ;PC7～0←((SP)),SP←(SP) - 1
```

RETI(中断返回):指令格式:

```
        RETI              ;PC15～8←((SP)),SP←(SP) - 1
                          ;PC7～0←((SP)),SP←(SP) - 1
                          ;清除相应中断优先级状态位
```

最后还有一条空操作指令:

```
        NOP               ;PC←(PC) + 1
```

该指令仅产生一个机器周期的延时,不进行任何操作。

3.3 汇编语言

3.3.1 伪指令

汇编语言也称符号语言，是使用助记符表示的机器指令进行编程的一种语言。通常把用汇编语言编写的程序称为汇编语言源程序，但机器不能直接识别和执行汇编语言源程序，必须将其翻译成机器语言程序（目标程序），计算机才能执行。这个翻译过程称为汇编。汇编有手工汇编和机器汇编两种方式。手工汇编是通过人工查找指令代码表，得到每条指令的机器代码；机器汇编是通过计算机执行一种系统软件（汇编程序）自动完成的。

当使用机器汇编时，必须在源程序中为汇编程序提供一些辅助信息，以便帮助其完成源程序的翻译并生成目标代码，如源程序中哪些是指令、哪些是数据，数据是字节还是字，代码存放的目的地址在哪里以及程序翻译到哪里结束等。这些为汇编程序提供必要的辅助信息，控制其汇编过程的命令称为伪指令。这里要注意，伪指令不是控制计算机执行某种操作的指令，仅仅是在机器汇编时为汇编程序提供必要的信息。因此，汇编时伪指令并不产生供机器直接执行的机器码，也不会直接影响存储器中代码和数据的分布。

不同的 51 汇编程序对伪指令的定义有所不同，但基本的用法是相似的，下面介绍一些常用的伪指令及其基本用法。

1. 定位伪指令 ORG

格式：

```
ORG  m
```

其中，m 一般为以十进制或十六进制数表示的 16 位地址，用来指定该伪指令后面的指令的汇编地址，即生成机器指令的起始存储地址，也可以用来指定其后的数据定义伪指令所定义的数据的起始存储地址。

在一个汇编语言源程序中允许使用多条定位伪指令，但其值不应和前面生成的机器指令存放地址重叠。在实际应用中，一般仅设置中断服务子程序和主程序的起始存放地址，其他程序或常数依次存放即可。

例 3.1 定位伪指令的用法举例。

```
        ORG     0000H
START:  AJMP    MAIN
          ⋮
        ORG     0100H
MAIN:   MOV     SP,# 30H
          ⋮
```

以 START 开始的程序汇编为机器代码后从 0000H 单元开始存放，以 MAIN 开始的程序机器代码则从 0100H 单元开始连续存放。

从前面的介绍可知，单片机复位后程序计数器 PC 的值为 0000H，即从地址 0 开始执行程序，所以汇编语言源程序一般都以伪指令 ORG 0000H 开始。如果在源程序开始处没有

ORG 伪指令,则汇编程序将从自动从 0000H 单元开始存放目标程序。但是我们还知道,程序存储器的 0003H、000BH、0013H 等单元为中断服务程序的入口地址,一般不应该被覆盖,否则相应的中断功能无法实现,所以可以在程序存储器的 0000H 处安排一条转移指令,将主程序转移到程序存储器的其他地方,本例中将程序的真正入口设置在程序存储器的 0100H 处。

2. 汇编结束伪指令 END

格式:

```
END
```

END 是汇编语言源程序的结束标志,表示汇编结束。机器汇编时遇到 END 就认为源程序已经结束,对 END 后面的指令都不再汇编。因此一个源程序只能有一个 END 伪指令,并且必须放在汇编语言源程序的末尾。

3. 定义字节伪指令 DB

格式:

[标号:] DB x_1, x_2, …, x_n

定义字节伪指令 DB(Define Byte)将其右边的数据依次存放到以左边标号为起始地址的存储单元中,x_i 为字节数据,可以采用二进制、十进制、十六进制和 ASCII 码等多种表示形式。DB 通常用于定义一个常数表。

例 3.2 定义字节伪指令的用法举例。

```
     ORG  7F00H
TAB: DB   01110010B,16H,45
     DB   '8','MCS - 51'
```

汇编后存储单元内容为:

```
(7F00H) = 72H    (7F01H) = 16H    (7F02H) = 2DH    (7F03H) = 38H
(7F04H) = 4DH    (7F05H) = 43H    (7F06H) = 53H    (7F07H) = 2DH
(7F08H) = 35H    (7F09H) = 31H
```

4. 定义字伪指令 DW

格式:

[标号:] DW Y_1, Y_2, …, Y_n

定义字伪指令 DW(Define Word)的功能与 DB 类似,但 DW 定义的是一个字(2 字节),主要用于定义 16 位地址表。汇编时,机器自动按高 8 位在前(低地址)、低 8 位在后(高地址)的格式存入存储器,这与 80x86 系列的微处理器正好相反。

例 3.3 定义字伪指令的用法举例。

```
       ORG  6000H
TAB:    DW  1254H,32H,161
       DW  'AB',TAB
```

汇编后存储单元内容为:

```
(6000H) = 12H    (6001H) = 54H    (6002H) = 00H    (6003H) = 32H
(6004H) = 00H    (6005H) = 0A1H   (6006H) = 41H    (6007H) = 42H
(6008H) = 60H    (6009H) = 00H
```

伪指令 DB 和 DW 均是根据源程序的需要，用来定义程序中用到的数据（地址）或数据块的，一般应放在源程序之后，汇编后的数据将紧挨着目标程序的末尾地址开始存放。

5. 定义预留存储空间伪指令 DS

格式：

```
[标号:] DS  数值表达式
```

定义预留存储空间伪指令 DS 从指定的地址开始，保留若干字节的内存空间作为备用。汇编时，将根据表达式的值决定从指定地址开始留出多少字节的存储空间，表达式也可以是一个指定的数值。

例 3.4　定义预留存储空间伪指令的用法举例。

```
ORG  0F00H
DS  10H
DB  20H,40H
```

汇编后，从 0F00H 开始，保留 16(10H)字节的内存单元，然后从 0F10H 开始，按照下一条 DB 伪指令给内存单元赋值，即(0F10H)＝20H，(0F11H)＝40H。保留的空间将由程序的其他部分决定其用处。

DB、DW 和 DS 伪指令都只对程序存储器起作用，不能用来对数据存储器的内容进行赋值或进行其他初始化工作。

6. 等值伪指令 EQU

格式：

```
字符名称  EQU  数据或汇编符号
```

等值伪指令 EQU 将其右边的数据或汇编符赋给左边的字符名称。“字符名称”被赋值后，在程序中就可以作为一个 8 位或 16 位的数据、地址或汇编符来使用了。

使用 EQU 伪指令时应注意，字符名称必须先定义后使用，通常将等值伪指令放在源程序的开头；在同一程序中，用 EQU 伪指令对标号赋值后，该标号的值在整个程序中不能再改变，即不能用 EQU 对同一个标号进行两次或两次以上的赋值。

例 3.5　等值伪指令的用法举例。

```
          ORG     8500H
AA        EQU     R1
A10       EQU     10H
DELAY     EQU     87E6H
          MOV     R0,A10      ;R0←(10H)
          MOV     A,AA        ;A←(R1)
          LCALL   DELAY       ;调用起始地址为 87E6H 的子程序
          END
```

EQU 赋值后，AA 为寄存器 R1，A10 为 8 位直接地址 10H，DELAY 子程序的入口地址为 87E6H。

7. 数据地址赋值伪指令 DATA

格式：

```
字符名称  DATA  表达式
```

数据地址赋值伪指令 DATA 的功能与 EQU 类似，是将其右边“表达式”的值赋给左边的“字符名称”。表达式可以是一个8位或16位的数据或地址，也可以是包含已定义“字符名称”在内的表达式，但不可以是一个汇编符号(如 R0、R7 等)。

DATA 伪指令定义的“字符名称”没有先定义后使用的限制，可以放在源程序的开头或末尾。

8. 位地址定义伪指令 BIT

格式：

```
字符名称  BIT  位地址
```

位地址定义伪指令 BIT 将其右边位地址赋给左边的字符名称。

例 3.6 位地址定义伪指令的用法举例。

```
A1      BIT  ACC.1
USER1   BIT  PSW.5
USER2   BIT  20H
```

这样就把位地址 ACC.1(累加器 ACC 的第1位)赋给了变量 A1，把位地址 PSW.5 赋给了变量 USER1，把位地址 20H 赋给了变量 USER2，在编程中 A1、USER1 和 USER2 就可以作为位地址使用了。

3.3.2 顺序程序设计

顺序程序是指执行顺序与源程序的书写顺序完全一致的一种程序结构，程序中不存在可引起程序流程发生改变的转移类指令。这种结构是所有结构中最简单、最基本的一种，也称为简单程序结构。

例 3.7 编写一个实现两个双字节无符号十进制数相加的程序。

设两个双字节无符号十进制数采用压缩的 BCD 码表示，分别存放在片内 RAM 的 40H、41H 和 50H、51H 单元，结果仍为压缩的 BCD 码，存放在片内 RAM 的 52H、51H 和 50H 单元中。程序如下：

```
DATA0       EQU  12H
DATA1       EQU  34H
DATA2       EQU  56H
DATA3       EQU  78H
ORG         0000H
AJMP        0100H
ORG         0100H
MOV         40H, #DATA0
MOV         41H, #DATA1   ;被加数送 41H,40H
MOV         50H, #DATA2
MOV         51H, #DATA3   ;加数送 51H,50H
MOV         A, 40H
```

```
        ADD       A, 50H        ;(40H) + (50H) →A
        DA        A
        MOV       50H, A        ;保存结果
        MOV       A, 41H
        ADDC      A, 51H        ;(41H) + (51H) + CY→A
        DA        A
        MOV       51H, A        ;保存结果
        MOV       52H, #0
        MOV       A, #0
        ADDC      A, 52H
        MOV       52H, A        ;进位 52H
LOOP0: SJMP      LOOP0
        END
```

程序中首先用 EQU 伪指令定义了 4 个常数(压缩的 BCD 码),这里要注意,常数后面的 H 一定不要漏掉,否则就不是 BCD 码了。如 12H 在计算机中的存储形式为 00010010B,可以表示 BCD 码的 12,而 12 在计算机中的存储形式为 00001100B(0CH),就不是 BCD 码。接着用 4 条 MOV 指令把 4 个常数分别存放到相应的存储单元中。最后进行相加运算,进行相加运算时应注意:

(1) 十进制的加法是在普通加法指令之后用 DA A 指令对结果进行调整实现的;

(2) 低字节相加用 ADD 指令,高字节相加要用 ADDC 指令;

(3) 两个双字节无符号十进制数的和有可能超出双字节的表示范围,要用 3 字节存储,第 3 字节为两数相加产生的进位,即 0 或 1。

本例实现的是十进制数 3412 和 7856 相加,结果为 11268。

程序的最后一条指令 LOOP0: SJMP LOOP0 是让程序原地踏步,不再往前执行。这条指令也可以用 SJMP $ 代替,$ 在指令中表示本条指令的地址。

3.3.3 分支程序设计

分支程序对程序中给定的条件进行判断,然后根据条件的成立与否决定程序的走向。

例 3.8 编写计算函数 $Y=f(X)$ 的程序。

$$Y=\begin{cases}2, & X>0\\ 0, & X=0\\ -2, & X<0\end{cases}$$

该函数有 3 个分支,要做两次判断:第 1 次将 X 取到累加器中并用 JZ 指令判断其是否为 0;第 2 次用位条件转移指令判断 $X>0$ 还是 $X<0$。设 X 和 Y 分别存放在片内 RAM 的 30H 和 31H 单元,程序如下:

```
 $INCLUDE(C8051F020.INC)     ;包含文件
X   EQU   30H
Y   EQU   31H
ORG      0000H
AJMP     0100H
ORG      0100H
MOV      A, X
JZ       DONE               ;若 X = 0,转 DONE
```

```
        JNB     ACC.7, POSI        ;若 X＞0,则转 POSI
        MOV     A, ＃0FEH          ;若 X＜0,则 A = －2(补码为 0FEH)
        LJMP    DONE
POSI:   MOV     A, ＃02H           ;保存结果
DONE:   MOV     Y, A
        SJMP    $
        END
```

.INC 包含文件类似于 C 语言中的.h 头文件,C8051F020.INC 中有对特殊功能寄存器的定义,如开头不加上语句 $INCLUDE(C8051F020.INC),汇编时因为找不到累加器 ACC 的定义而报 ACC.7 是非法位地址的错误信息。文件 C8051F020.INC 要和源文件放在同一个文件夹内。如果程序中只用到个别特殊功能寄存器,或者 C8051F020.INC 缺少所用特殊功能寄存器的定义,也可以在源程序中用 EQU(或 DATA)伪指令直接定义相关特殊功能寄存器,如本例中也可以用 ACC EQU 0E0H 语句替换 $INCLUDE(C8051F020.INC)语句。

3.3.4 循环程序设计

1. 循环程序的结构

顺序结构和分支结构中的指令一般只执行一次。而在一些实际应用系统中,往往同一组操作需要重复执行多次,这种有规律可循又需反复处理的操作可采用循环结构的程序来实现。这样结构可使程序简短,占用内存少,重复次数越多,运行效率越高。

循环程序一般包括以下几个部分:

(1) 初始化部分。程序在进入循环之前,应对各循环变量、其他变量和常量赋初值,为循环操作做必要的准备工作。

(2) 循环体部分。这一部分由重复执行部分和循环控制部分组成。这是循环程序的主体,又称为循环体。值得注意的是,每执行一次循环体后,必须为下一次循环创造条件。如对数据地址指针、循环计数器等循环变量进行修改,还要检查判断循环条件,若符合循环条件,则继续重复循环,不符合则退出循环,以实现对循环的判断与控制。

(3) 结束部分。用于保存和分析循环程序的处理结果。

循环程序设计的一个主要问题是循环次数的控制,一般有两种控制方法:第一种方法是先判断再处理,即先判断是否满足循环条件,如不满足,则不循环。这种结构的循环也称为当型循环,其流程图如图 3-1(a)所示。第二种方法是先处理再判断,即循环先执行一次后,再判断是否还需要下一次循环。这种结构的循环也称为直到型循环,其流程图如图 3-1(b)所示。

例 3.9 片外 RAM 的 BLOCK 单元开始有一个无符号数据块,数据块长度存放在片内 RAM 的 LEN 单元中,编程找出数据块中的最大数存入片内 RAM 的 MAX 单元。

这是基本搜索问题。可以采用两两比较的方法,取两者较大的数再与下一个数进行比较,若数据块长度(LEN)$=n$,则应比较 $n-1$ 次,最后较大的数就是数据块中的最大数。

程序中使用减法指令和借位标志 CY 来判断两数的大小,用 B 寄存器作比较与交换的暂存寄存器,使用 DPTR 作外部 RAM 的地址指针。流程图如图 3-2 所示,程序如下:

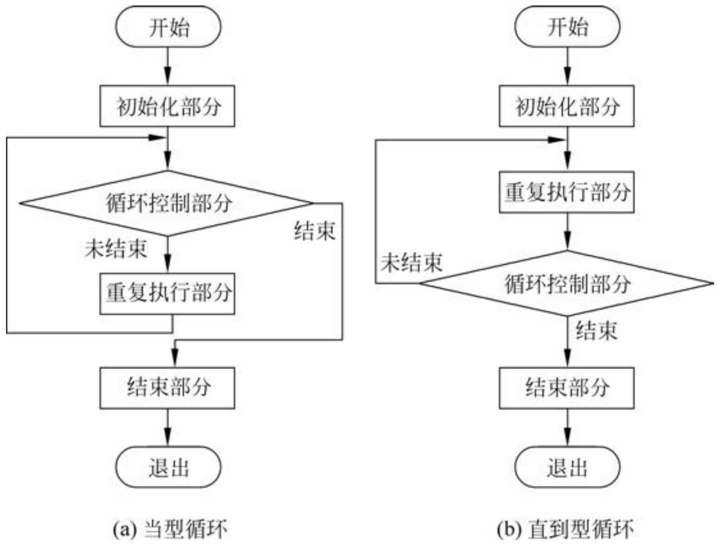


图 3-1　两种循环结构的流程图

```
        $ INCLUDE(C8051F020.INC)    ;包含文件
        BLOCK      DATA 0100H       ;定义数据块首址
        MAX        DATA 31H         ;定义最大数存储单元
        LEN        DATA 30H         ;定义长度计数单元
        ORG        0000H
        AJMP       FMAX
        ORG        0100H
FMAX: MOV DPTR, #BLOCK              ;数据块首址送 DPTR
        DEC        LEN              ;长度减 1
        MOVX       A, @DPTR         ;取数至 A
LOOP:   CLR        C                ;0→CY
        MOV        B, A             ;暂存于 B
        INC        DPTR             ;修改指针
        MOVX       A, @DPTR         ;取数
        SUBB       A, B             ;比较
        JNC        NEXT
        MOV        A, B             ;大者送 A
        SJMP       NEXT1
NEXT:   ADD        A, B             ;(A)>(B),则恢复 A
NEXT1: DJNZ        LEN, LOOP        ;未完继续比较
        MOV        MAX, A           ;存最大数
        SJMP        $               ;程序踏步,若用 RET 指
                                    ;令结尾则可作为子程
                                    ;序调用
        END
```

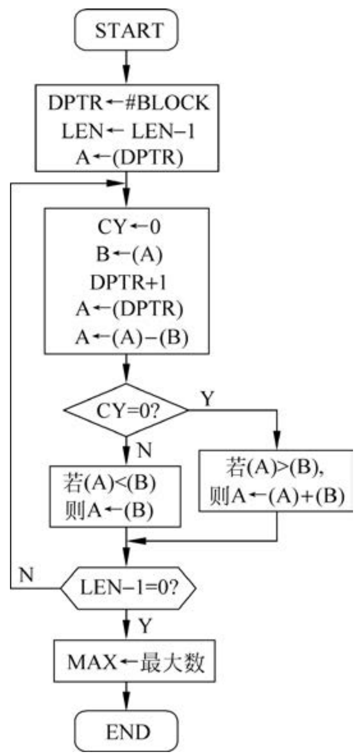


图 3-2　例 3.9 的程序流程图

程序中因为用减法指令 SUBB A, B 比较两数的大小,执行完后 A 中为两数的差,所以当 A 中的数据大于 B 中的数据时,需要对 A 中的数据用 ADD A,B 指令进行恢复。数据的比较也可以用 CJNE 指令实现,该指令可同时完成数据的比较与不相等时的转移功能,效率更高,作为练习请读者自己完成。

例 3.10 片外 RAM 的 BLOCK 单元开始有一个带符号数据块,数据块的长度存放在片内 RAM 的 LEN 单元。试编写程序统计其中正数、负数和零的个数,分别存入片内 RAM 的 PCOUNT、MCOUNT 和 ZCOUNT 单元。

这是一个包含多重分支的单循环程序。数据块中的数据是用补码表示的带符号数,因而首先用 JB ACC.7,REL 指令判断其符号位。若 ACC.7=1,则该数一定是负数,MCOUNT 单元加 1;若 ACC.7=0,则该数可能为正数,也可能为零,再用 JNZ REL 指令进一步判断之,若 A≠0,则一定是正数,PCOUNT 加 1;否则该数为零,ZCOUNT 加 1。当数据块中的所有数据都按顺序判断一次之后,PCOUNT、MCOUNT 和 ZCOUNT 单元中就分别对应正数、负数和零的个数。流程图如图 3-3 所示,程序如下:

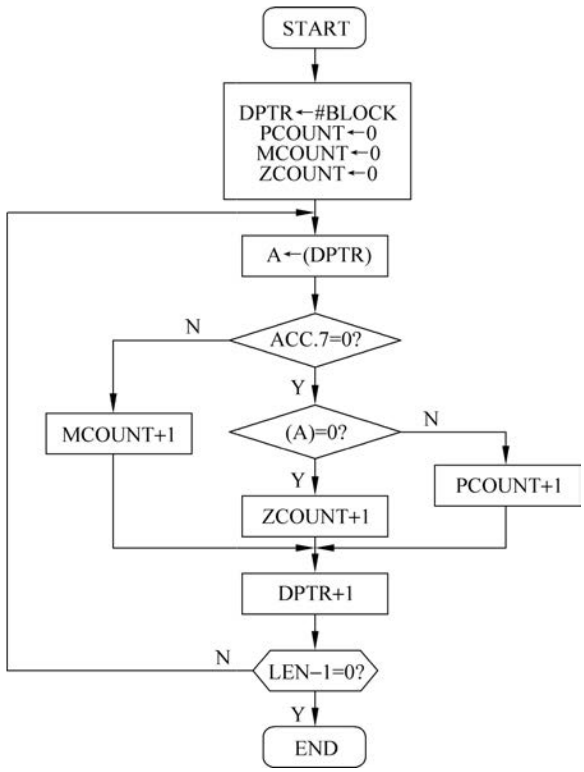


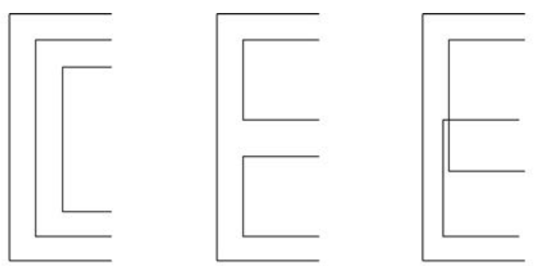
图 3-3 例 3.10 的程序流程图

```
 $ INCLUDE(C8051F020.INC)
        BLOCK       DATA 2000H     ;定义数据块首址
        LEN         DATA 30H       ;定义长度计数单元
        PCOUNT      DATA 31H       ;正数计数单元
        MCOUNT      DATA 32H       ;负数计数单元
        ZCOUNT      DATA 33H       ;零计数单元
        ORG         0000H
        AJMP        START
        ORG         0100H
START:  MOV         DPTR, #BLOCK
        MOV         PCOUNT, #0     ;计数单元清 0
        MOV         MCOUNT, #0
```

```
        MOV             ZCOUNT, #0
LOOP:   MOVX            A, @DPTR        ;取数
        JB              ACC.7, MCONT    ;若 ACC.7 = 1,转负计数
        JNZ             PCONT           ;若(A)≠0,转正计数
        INC             ZCOUNT          ;若(A) = 0,则零的个数加 1
        AJMP            NEXT
MCONT:  INC             MCOUNT          ;负数计数单元加 1
        AJMP            NEXT
PCONT:  INC             PCOUNT          ;正计数单元加 1
NEXT:   INC             DPTR            ;修正指针
        DJNZ            LEN, LOOP       ;未完继续
        SJMP            $
        END
```

2. 多重循环

在前面介绍的两个例子中,程序中都只包含一个循环,这种程序称为单重循环程序。而遇到复杂问题时,采用单重循环往往不够,必须采用多重循环才能解决。所谓多重循环(也称循环嵌套),就是在循环程序中还套有其他的循环程序。应注意,在多重循环中内外循环不能交叉,也不允许从外循环跳入到内循环,只能将内循环完整地包含在外循环当中。图 3-4(a)、图 3-4(b)所示是正确的循环嵌套形式,图 3-4(c)是应避免的不正确的循环嵌套形式。下面通过两个实例说明多重循环程序的设计方法。



(a) 正确嵌套形式1　(b) 正确嵌套形式2　(c) 错误嵌套形式

图 3-4　多重循环的嵌套形式

例 3.11　片外 RAM 的 BLOCK 开始的单元中有一个无符号数据块,其长度存放在片内 RAM 的 LEN 单元中(为方便编程,假设长度<256)。编程将这些无符号数按递增顺序重新排列,并存入原存储区。

处理这个问题要利用双重循环结构,在内循环中将相邻两单元的数进行比较,若符合从小到大的次序则不做任何操作,否则应将两数交换。这样两两比较下去,$n-1$ 次比较之后所有的数都比较并交换完毕,最大数沉底,在下一个内循环中将减少一次比较与交换。为提高程序的执行效率,进行下一次内循环前,先检查上次内循环中有无数据交换发生,若从未交换过,则说明这些数据已按递增顺序排列,程序可提前结束(未执行完的外循环次数不需要再执行);否则将再进行下一次内循环,如此反复比较与交换,每次内循环的最大数都沉底,而较小的数一个个冒上来,因此这种排序方法称为“冒泡排序”。

程序中用 R0、R1 存放相邻两单元地址的低字节,采用 8 位 MOVX 指令访问 XRAM 中的数据,此时需要由外部存储器接口控制寄存器 EMIOCN 提供地址的高字节。因为数据

的个数小于 256,所以整个程序中 EMIOCN 的值不需要改变。用 R7、R6 作为外循环与内循环的循环计数器;用程序状态字 PSW 中的用户标志位 F0 作为交换标志位,内循环中有交换发生时则将 F0 置 1,进入外循环时将 F0 清 0。因为包含文件 C8051F020.INC 中没有 EMIOCN 的定义,所以需要用 DATA 或 EQU 伪指令定义 EMIOCN 的地址,也可以通过修改 C8051F020.INC 文件,加入 EMIOCN 的定义。

流程图如图 3-5 所示,程序如下:

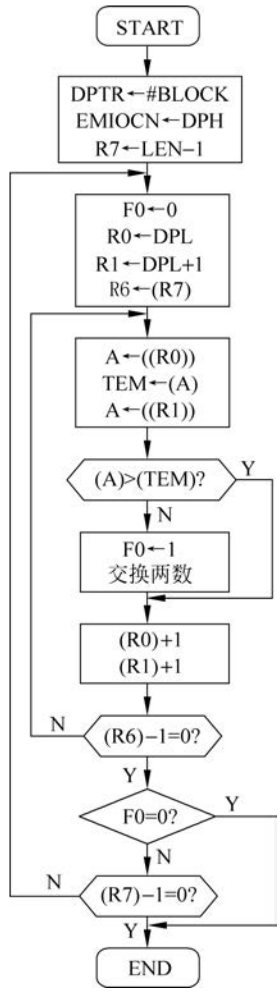


图 3-5　例 3.11 的程序流程图

```
        $ INCLUDE(C8051F020.INC)
        BLOCK     DATA 2200H
        LEN       DATA 51H
        TEM       DATA 50H
        EMIOCN    DATA 0AFH    ;外部存储器接口控制寄存器的地址
        ORG       0000H
        AJMP      START
        ORG       0100H
START:  MOV       DPTR, #BLOCK ;置数据块地址指针
        MOV       EMIOCN, DPH  ;EMIOCN 存放地址的高字节
        MOV       R7, LEN      ;置外循环计数初值
        DEC       R7           ;外循环最多执行 n-1 次
LOOP0:  CLR       F0           ;交换标志清 0
        MOV       R0, DPL
        MOV       R1,DPL       ;置相邻两数地址指针低字节
        INC       R1
        MOV       A, R7
        MOV       R6, A        ;置内循环计数器初值
LOOP1:  MOVX      A, @R0       ;取数
        MOV       TEM, A       ;暂存
        MOVX      A, @ R1      ;取下一个数
        CJNE      A, TEM, NEXT ;两相邻数比较,不等则转
        SJMP      NOCHA        ;相等不交换
NEXT:   JNC       NOCHA        ;CY = 0,不交换
        SETB      F0           ;置位交换标志
        MOVX      @R0, A
        XCH       A, TEM
        MOVX      @R1, A       ;两数交换
NOCHA:  INC       R0
        INC       R1           ;修改指针
        DJNZ      R6, LOOP1    ;内循环未完,则继续
        JNB       F0, HALT     ;若上次内循环中没有交换,则结束
        DJNZ      R7, LOOP0    ;未完,继续
HALT:   SJMP      $
        END
```

例 3.12　设系统时钟频率为 20MHz,试编写延时 2ms 的延时子程序。

软件延时是靠处理器执行一段程序达到的,通常称为延时子程序。延时子程序通常采用双重或多重循环结构。在系统时钟频率确定之后,延时时间主要与两个因素有关:其一

是循环体(内循环)中指令的执行时间;其二是外循环变量(时间常数)的设置。

系统时钟频率为 20MHz,则一个时钟周期为 0.05μs,执行一条 DJNZ Rn,rel 指令需 2 个时钟周期,即 0.1μs。

延时 2ms 的子程序如下:

```
DELAY: MOV R7, #200          ;1 个时钟周期
DLY0: MOV R6, #100           ;1 个时钟周期
DLY1: DJNZ R6, DLY1          ;200 × (100 × 2 × 0.05) = 200 × 10μs = 2ms
      DJNZ R7, DLY0          ;2 个时钟周期
      RET                    ;2 个时钟周期
```

程序的注释部分给出了延时时间的计算,但这种计算不太精确,只计算了内循环一条指令(程序中的第 3 条指令)的执行时间,没有把其他指令的执行时间计算进去。若把所有指令的执行时间计算在内,则延时时间为

$$(10\mu s + 0.15\mu s) \times 200 + 0.15\mu s = 2030.15\mu s = 2.03015ms$$

其中,括号内的 10μs 为内循环执行时间,0.15μs 为第 2、4 条指令的执行时间,两者之和乘以 200 为外循环执行时间,括号外的 0.15μs 为第 1、5 条指令的执行时间。

如果要求延时时间比较精确,可通过修改循环次数和在循环体中增加 NOP 指令的方式实现。本例程序可修改如下:

```
DELAY: MOV  R7, #200
DLY0:  MOV  R6, #98
       NOP
DLY1:  DJNZ R6, DLY1           ;98 × 2 × 0.05 = 9.8μs
       DJNZ R7, DLY0
       RET
```

此程序的实际延时时间为

$$(9.8\mu s + 0.2\mu s) \times 200 + 0.15\mu s = 2000.15\mu s = 2.00015ms$$

如果需要延时更长时间,可以采用多重循环实现。注意,用软件实现延时时,应禁止中断,否则延时过程中若有中断发生,则会严重影响延时时间的准确性。

3.3.5 子程序设计

在大型程序或是多个不同的程序中,往往有许多地方需要执行同样的运算和操作。例如,求三角函数和各种加减乘除运算、代码转换以及延时程序等。如果编程过程中每遇到这样的操作都单独编写一段程序,会使编程工作十分烦琐,也会占用大量程序存储空间。通常将这些能完成某种基本操作的程序段单独编制成子程序,以供不同程序或同一程序的不同地方反复调用。在程序中需要执行这种操作的地方执行一条调用指令,转到子程序中完成规定操作后再返回到原来的程序中继续执行。这就是所谓的子程序结构。

为使子程序能在不同程序或同一程序中反复被调用,子程序应具备以下特征:

(1) 通用性。子程序应设计成能由各种应用程序调用的通用程序,这主要是通过可变参数实现的。

(2) 可浮动性。子程序可以不加任何修改地存放在存储器的任何区域。这要求在子程序中应避免使用绝对转移指令,子程序的首地址应该使用符号地址(即子程序名)。

(3) 可递归性和可重入性。可递归性是指子程序可以自己调用自己,可重入性是指一个子程序可以同时被多个程序调用。这两个特性主要是对大规模复杂系统程序的要求,对一般程序可以不做要求。

1. 子程序的设计要点

1) 子程序的结构

用汇编语言编制子程序时,要注意以下两个问题:

(1) 子程序的第一条指令必须有一个简单明了、见名识义的标号,此标号即为子程序名,代表该子程序的入口地址。在主程序中使用短调用指令 ACALL 或长调用指令 LCALL 和子程序名即可调用子程序。例如调用延时子程序可用:

```
LCALL  DELAY
```

或

```
ACALL  DELAY
```

其中,DELAY 就是子程序名,即延时子程序第一条指令的标号。这两条调用指令属于程序分支(转子)指令,不仅具有寻址子程序入口地址的功能,而且在转入子程序之前,硬件能自动将主程序的断点(调用指令的下一条指令地址)入栈保存,以使返回指令 RET 能正确返回。

(2) 子程序结尾必须有一条子程序返回指令 RET。该指令具有恢复主程序断点的功能,即从堆栈弹出断点至程序计数器 PC,以便继续执行主程序。一般来说,子程序调用指令和子程序返回指令要成对使用。注意,中断服务子程序的返回指令是 RETI,该指令除返回断点外,还会清除相应中断优先级状态位。

2) 参数传递

要使子程序具备通用性,子程序内部就不能使用固定的数据完成相关计算,子程序中使用的数据一般要由主程序以参数的形式提供,子程序的计算结果也应以参数的形式传给主程序。在调用一个子程序时,主程序应先把有关参数(也称入口条件)放到某些约定的位置,子程序在运行时,可以从约定的位置得到有关参数。同样在子程序结束前,也应把处理结果(也称出口条件)送到约定位置。返回后,主程序便可从这些位置中得到需要的结果,这就是参数传递。参数传递的方法有多种,下面将结合具体例子介绍。

3) 现场保护和恢复

进入子程序后,特别是进入中断服务子程序时,要特别注意现场保护和恢复问题。现场保护是指主程序中使用的 RAM 内容、各工作寄存器的内容、累加器 A 的内容和 DPTR 以及 PSW 等寄存器内容,都不应因转入子程序而改变。如果子程序所使用的寄存器和存储单元与主程序中使用的寄存器和存储单元有冲突,则在转入子程序后首先要采取保护措施。现场保护的一般方法是将要保护的单元和寄存器的内容压入堆栈,从而空出这些单元和寄存器供子程序使用。子程序返回主程序之前再将这些内容弹出到原工作单元,恢复主程序原来的状态,此即现场恢复。压栈与出栈应按相反的顺序进行,这样才能保证现场的正确恢复。例如:

```
子程序名: PUSH  ACC        ;保护现场
          PUSH  PSW
```

```
PUSH  DPL
PUSH  DPH
   …           ;子程序体
POP   DPH      ;恢复现场
POP   DPL
POP   PSW
POP   ACC
RET            ;子程序返回
```

对于一个具体的子程序是否要进行现场保护，以及哪些寄存器和存储单元需要保护，要视具体情况而定，不能一概而论。

通过前面的学习，我们知道工作寄存器有 4 个区，每区 8 个(R0～R7)，每个程序一般只使用其中的一个区。那么对工作寄存器的保护就可以通过简单的寄存器区切换来实现。如，通过以下两条指令就可以将工作寄存器切换到第 2 个区。

```
SET  RS1
CLR  RS0
```

4）堆栈设置

调用子程序时，主程序的断点将自动入栈；转入子程序后，现场的保护也要占用堆栈单元，尤其是多重子程序嵌套调用，要求堆栈有更多的空间。因此，恰当地设置堆栈指针 SP 的初值是十分重要的。

C8051F 系列单片机堆栈指针 SP 的复位值为 07H，程序中可将其设置在内部 RAM 的任意单元，但考虑 00H～1FH 为工作寄存器区，20H～2FH 为可位寻址的区域，因此，一般将 SP 设置在 30H 以上的单元。

2. 参数传递方法

1）无须参数传递

这类子程序所需的参数是由子程序本身赋予的，不需要主程序给出。例如，例 3.12 的最后一句为 RET 指令，所以该例是子程序的结构形式。但该子程序延时 2ms 所需要的参数(内、外循环的次数)完全是在子程序内部直接赋值的，调用时只需在主程序中适当位置写入 LCALL DELAY 或 ACALL DELAY 指令即可。

2）用累加器和工作寄存器传递参数

这种方法要求在转入子程序之前把所需的入口参数存入累加器 A 和工作寄存器 R0～R7 中。在子程序中对这些数据进行相关操作，返回时，出口参数也保存在累加器和工作寄存器中。这种参数传递方法最直接、最简单，运算速度也最快。但由于工作寄存器的数量有限，不能传递更多的参数。

例 3.13　编写计算 $c=a^2+b^2$ 的程序，设 a、b 均小于 10。a、b 分别存放在片内 RAM 的 31H、32H 单元，结果 c 存入片内 RAM 的 34H 和 33H 单元(要求和为 BCD 码)。

因该算式两次用到平方值，所以可将求平方运算编写为子程序，主程序中两次调用，再求和即可。求平方值采用查表法实现，主程序和子程序编写如下：

主程序：

```
$ INCLUDE(C8051F020.INC)
ORG    0000H
```

```
        AJMP    START
        ORG     0100H
START:  MOV     SP, #3FH
        MOV     A, 31H      ;取 a
        LCALL   SQR         ;求 a²
        MOV     R1, A
        MOV     A, 32H      ;取 b
        LCALL   SQR         ;求 b²
        ADD     A, R1       ;求和
        DA      A           ;调整
        MOV     33H, A
        MOV     A, #0
        ADDC    A, #0       ;计算和高位
        MOV     34H, A
        SJMP    $
```

子程序：

```
        ORG  0030H
   SQR: INC A               ;累加器 A 增加 1B 变址调整值
        MOVC A, @A+PC
        RET                 ;1B
   TAB: DB   00H, 01H, 04H, 09H, 16H, 25H, 36H, 49H, 64H, 81H
        END
```

主程序和子程序之间使用累加器 A 传递参数。查表指令 MOVC A,@A+PC 使用 A+PC 作为地址访问程序存储器,取出其内容送给累加器 A,执行该指令前,A 中存放的是待查表项数(由主程序设定)。但执行 MOVC A,@A+PC 指令时,PC 指向的是其下一条指令 RET 的首地址,而非表格首地址,因此要能正确查找到表格中的内容,必须使累加器 A 再加上一个变址调整值。这里的变址调整值即为 MOVC A,@A+PC 指令的下一条指令(RET 指令)到表首的间隔,即两处地址之间其他指令所占字节数,这里仅 RET 一条指令,占 1 字节,所以使用 INC A 指令。每条指令所占的字节数都可以在附录 A 中查到。若查表指令与表首之间指令较多,也可以通过指令标号相减的方式让汇编程序自动完成计算,从而省去人工查表的麻烦,如本例的查表子程序也可以按如下方式编写。

```
SQR:   ADD A, #(TAB-XX)     ;累加器 A 变址调整
       MOVC   A, @A+PC
XX:    RET
TAB:   DB     00H, 01H, 04H, 09H, 16H, 25H, 36H, 49H, 64H, 81H
       END
```

用 PC 内容作基址查表只能查距本指令 256B 以内的表格数据,称页内查表指令或短查表指令。查表子程序也可以用指令 MOVC A, @A+DPTR 实现,只要让 DPTR 指向表首,A 中存放待查数据即可,该指令可在 64KB 程序存储器范围内查表,称为长查表指令。本例用 MOVC A, @A+DPTR 实现的查表子程序如下：

```
       ORG  0030H
SQR:   MOV  DPTR, #TAB
       MOVC  A,@A+DPTR
       RET
```

```
TAB:  DB  00H, 01H, 04H, 09H, 16H, 25H, 36H, 49H, 64H, 81H
      END
```

3）通过操作数地址传递参数

该方法中主程序将子程序所需的操作数存入数据存储器中，调用子程序之前将操作数的地址作为入口参数存入R0、R1或DPTR中。子程序以R0、R1或DPTR间接寻址访问数据存储器即可取出所需数据，结束前将结果仍存入数据存储器中，并将其地址作为出口参数存入R0、R1或DPTR中。主程序再以R0、R1或DPTR间接寻址访问数据存储器即可取得运算结果。一般内部RAM由R0、R1作地址指针，外部RAM由DPTR作地址指针。这种参数传递方法可以节省传递数据的工作量，可实现变字长运算。

例3.14 *n*字节求补子程序。

入口参数：(R0)＝待求补数低字节指针，(R7)＝$n-1$

出口参数：(R0)＝求补后的高字节指针

求补运算就是对数据(含符号位)变反加1，程序如下：

```
CPLN: MOV     A, @R0
      CPL     A               ;最低字节取反
      ADD     A, #1           ;加1
      MOV     @R0, A
NEXT: INC     R0
      MOV     A, @R0
      CPL     A               ;高字节取反
      ADDC    A, #0           ;传递进位
      MOV     @R0, A
      DJNZ    R7, NEXT
      RET
```

4）通过堆栈传递参数

堆栈可用于参数传递，在调用子程序前，主程序先把参与运算的操作数用PUSH指令压入堆栈。转入子程序后，用POP指令取出操作数进行相应运算，并把运算结果压入堆栈。返回主程序后，可用POP指令获取运算结果。值得注意的是，转向子程序时，主程序的返回地址也要压入堆栈，占用堆栈2字节，弹出参数时要用两条DEC SP指令修改SP指针，以便使SP指向操作数。另外在子程序返回指令RET之前要增加两条INC SP指令，以便使SP指向返回地址，保证能正确返回主程序。

例3.15 在片内RAM的HEX单元存放两个十六进制数，编程将它们分别转换为ASCII码并存入片内RAM的ASC和ASC+1单元。

由于要进行两次转换，故可调用查表子程序完成。

主程序：

```
       $INCLUDE(C8051F020.INC)
      ORG     0000H
      HEX     DATA 20H
      ASC     DATA 30H
      AJMP    MAIN
      ORG     0100H
MAIN: MOV     SP, #3FH
```

```
        PUSH    HEX         ;取被转换数
        LCALL   HASC        ;调用子程序
* PC→   POP     ASC         ;ASCL→ASC
        MOV     A, HEX      ;取被转换数
        SWAP    A           ;处理高 4 位
        PUSH    ACC
        LCALL   HASC        ;调用子程序
        POP     ASC + 1     ;ASCH→ASC + 1
        AJMP    $
```

子程序：

```
HASC:   DEC   SP            ;修改 SP 指向 HEX
        DEC   SP
        POP   ACC           ;弹出 HEX
        ANL   A, #0FH       ;屏蔽高 4 位
        ADD   A, #7         ;变址调整
        MOVC  A, @A + PC    ;查表
        PUSH  ACC           ;结果入栈 (2B)
        INC   SP            ;修改 SP 指向断点位置(2B)
        INC   SP            ;(2B)
        RET                 ;(1B)
ASCTAB: DB '0123456789ACBDEF'
        END
```

在主程序中将入口参数 HEX 入栈，即 HEX 被推入堆栈的 40H 单元，当执行 LCALL HASC 指令之后，主程序的返回地址 PC 也被压入堆栈，即 * PCL 被推入 41H 单元，* PCH 被推入 42H 单元，此时 SP=42H，如图 3-6(a)所示。进入子程序 HASC 后，两条 DEC SP 指令使 SP 指向参数 HEX，如图 3-6(b)所示，然后用 POP 指令将其弹出。查表变换的结果通过 PUSH 指令压入到原来 HEX 所在的堆栈单元。返回子程序前用两条 INC 指令使 SP 指向存放返回地址的单元处，如图 3-6(c)所示，以便由 RET 指令正确返回。

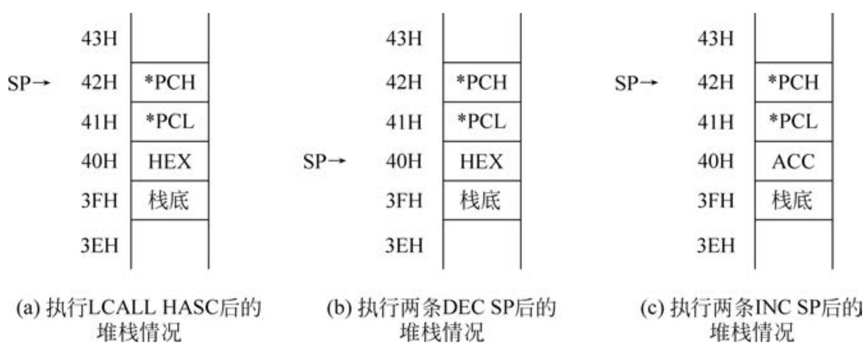


图 3-6　例 3.15 堆栈变化

使用堆栈传递参数，方法简单，能传递大量参数，不必为特定参数分配存储单元。

3.4　C51 语言

目前的嵌入式系统硬件性能和软件规模都有了很大的提高，为了提高程序开发效率和程序质量，开发人员更多采用 C 语言进行嵌入式软件程序设计。

使用 C 语言有以下优点：

- C 语言具有结构性和模块化特点，便于程序的阅读和维护。
- C 语言可移植性好，功能模块可以在不同项目中使用，从而减少了开发时间。
- C 语言程序设计更加简明、清晰，可以减少编程错误，从而提高开发效率。
- C 语言和微控制器是相对独立的，开发者无须详细了解微控制器的内部结构和处理过程，因此可以很快上手。

尽管 C 语言有以上优点，但有时必须采用 C 和汇编语言混合编程才能实现特定功能。在对实时响应时间有很严格要求的应用系统中，使用汇编语言是开发者的唯一选择。

C51 语言是为 8051 单片机编程而设计的一种专用 C 语言，它完全兼容 ANSI 标准 C 语言规范，并针对 8051 单片机的体系结构的特点对 ANSI C 的关键字做了一些扩展。

3.4.1 C51 关键字

关键字(Key Word)是一种具有固定名称和特定含义的标识符，又称为保留字(Reserved Word)。用户自定义的标识符不能和关键字同名。

ANSI C 语言定义了 32 个关键字，还为预处理功能保留了关键字。C51 语言除了支持 ANSI C 标准的关键字以外，还增加了若干关键字，按照其功能划分如下。

(1) 存储器类型相关：用来声明变量存储的内存区域(参见图 2-5)。

code：用来定义位于 8051 程序代码存储区的只读变量。

bdata：用来定义位于 8051 可位寻址的内部数据存储区的变量。

data：用来定义位于 8051 可直接寻址的内部数据存储区的变量。

idata：用来定义位于 8051 可间接寻址的内部数据存储区的变量。

pdata：用来定义位于分页寻址的 8051 外部数据存储区的变量，页大小一般为 256B。

xdata：用来定义位于 8051 外部数据存储区的变量，一般用于外部数据存储区不大于 64KB 的情况。

far：用来定义位于 8051 外部数据存储区的变量，一般用于外部数据存储区大于 64B 的情况。

bit：用来定义位于 8051 可位寻址的内部数据存储区的位变量。

_at_：用来对变量进行存储器绝对空间地址定位。

(2) 特殊功能寄存器相关：用来声明特殊功能寄存器。

sbit：声明可位寻址的特殊功能寄存器的特殊功能位。

sfr：声明 8 位的特殊功能寄存器。

sfr16：声明 16 位的特殊功能寄存器。

(3) 存储模式相关：用来声明没有显式指定存储类型的变量的存储区域。

small：小模式，变量默认存放在内部数据存储区。

compact：紧凑模式，变量默认存放在分页寻址的外部数据存储区。

large：大模式，变量默认存放在外部数据存储区(该区域一般不大于 64KB)。

(4) 函数相关：用来声明函数的实现方法。

interrupt：专门用于中断服务函数的定义与声明。

reentrant：专门用于可重入函数的定义与声明。

using：专门用于指定函数内部使用的 8051 的工作寄存器组。

(5) 其他。

alien：用以声明与 PL/M51 兼容的函数。

priority：规定 RTX51 或 RTX51 Tiny 的任务优先级。

task：定义实时多任务函数。

3.4.2 C51 变量定义

1. 存储器类型

C51 编译器允许在变量声明时使用存储器类型(Memory Type)来指定变量所希望占用的存储区域类型。C51 中的存储器类型修饰符如表 3-1 所示。

表 3-1 C51 中的存储器类型修饰符

类型	存储器类型	存 储 区 域	区域大小	对应的汇编语句	描 述
代码区	code	程序存储区	64KB	MOVC XX,@A+DPTR	用来存储只读变量
内部数据存储区	bdata	可位寻址的内部数据存储区	16B	MOV XX,ADDR	还可使用位寻址来访问的区域
	data	直接寻址的内部数据存储区	128B	MOV XX,ADDR	访问速度快(包含 bdata 区)
	idata	间接寻址的内部数据区	256B	MOV XX,@Rn	可访问整个内部数据区域(包含 data 区)
外部数据存储区	xdata	外部数据存储区	64KB	MOVX XX,@DPTR	使用 DPTR 来访问外部数据
	far	扩充的 RAM 和 ROM			使用用户定义的专用例程或特殊芯片专用指令来访问
	pdata	分页的外部数据存储区	256B	MOVX XX,@Rn	利用 R0,R1 来访问分页存储的外部数据

注意：idata 可用来访问整个内部数据存储区，即 256B，并不是仅局限于内部数据存储区的后 128B。

声明变量时可以说明变量的存储器类型，如下所示：

```
char data var1;                         //内部数据区的字符类型变量
char code text[] = "Hello world!";      //程序区的只读字串变量
unsigned long xdata array[100];         //外部数据区的无符号整型数组变量
float idata x,y,z;                      //间接寻址内部数据区的浮点变量
unsigned int pdata dimension;           //分页寻址外部数据区的无符号整型变量
unsigned char xdata vector[10][4][4];   //外部数据区的无符号字符类型的三维数组变量
char bdata flags;                       //可位寻址内部数据区的字符变量
```

说明：声明变量时存储区修饰符和数据类型修饰符的位置可以互换，即“char data x;”和“data char x;”是完全等效的。本书从一致性考虑，使用前一种格式。

2. 存储模式

如果在变量定义时未显式声明变量的存储器类型，则该变量的存储器类型由程序的存储模式(memory model)来决定。常见的存储模式有以下 3 种。

1）小模式

在小模式（small model）下，所有未显式声明存储器类型的变量，使用内部数据区来存放，即这种方式和用 data 进行显式声明一样。在这种存储模式下，变量的访问是最有效的。但是所有的数据对象（包括堆栈）都必须放在内部数据存储区中，可使用的数据存储区最少。

2）紧凑模式

在紧凑模式（compact model）下，所有未显式声明存储器类型的变量都使用分页寻址外部数据区来存放，即这种方式和用 pdata 显式声明一样。该模式利用 R0 和 R1 寄存器来进行间接寻址（@R0，@R1），此时最大可寻址 256B 的存储区域。这种方式的存取速度比小模式慢，但比大模式快。

使用紧凑模式时，C51 编译器使用@R0 和@R1 来访问外部数据区。R0 和 R1 寄存器的大小为 1 字节，因此只能存放所要访问单元的低 8 位地址。在紧凑模式下如果使用了超过 256B 的外部数据存储区，那么访问单元的高 8 位地址（即页地址）必须由端口 P2 来输出。开发人员必须为分页寻址设置合适的开始地址，编译器会使用这个开始地址在启动代码中对 P2 端口进行设置。

3）大模式

在大模式（large model）下，所有未显式声明存储器类型的变量都使用外部数据区来存放，即和用 xdata 显式说明一样。此时最大可寻址 64KB 的存储区域。此时会使用数据指针寄存器（DPTR）来进行间接寻址来访问相关数据。使用这种寻址方式效率低，生成的代码比小模式和紧凑模式下生成的代码都要长。

注意：小模式可以提供最快和最有效的代码，所以一般选择小模式。当所需数据存储区域较大时，可选择紧凑模式或者大模式。

3. 设置存储模式

有以下几种方式设置存储模式：

1）在 IDE 中的配置选项中设置

在 KEIL IDE 中，可以通过选择 Project→Options for Target 命令进入配置窗口。单击 Target 选项卡，更改 Memory Model 选择框的设定，如图 3-7 所示。

图 3-7　在 IDE 中设置存储模式

该选项的变更实际是通过 C51 的编译参数实现的,可单击 C51 选项卡查看。

小模式是默认模式,无额外选项;紧凑模式使用 COMPACT 选项;大模式使用 LARGE 选项。

2) 在代码中用编译选项设定

在文件开头位置,使用#pragma[SMALL|COMPACT|LARGE]设置。如:

```
#pragma LARGE
```

3) 在函数声明中指定

在定义函数时,使用 large、compact、small 来指定函数内部默认的存储模式。如:

```
void add(int x,int y) large
```

4. 指定变量的绝对地址

开发者有时候希望把变量存储在指定的地址单元中,可用_at_关键词来将变量定位在一个绝对的内存地址单元。使用方法如下:

```
数据类型 存储器类型 变量名_at_变量所在绝对地址;
```

在_at_后面的绝对地址必须符合存储器类型的物理边界限制,即不超过存储区域的最大可寻址范围,该地址必须为常数。

绝对定位的变量遵循以下约束:

(1) 声明绝对定位的变量时,不能同时进行初始化赋值。

(2) 类型为 bit 的变量不能进行绝对地址定位。

(3) 只能对全局变量进行绝对定位,不能对局部变量进行。

下面的例子演示了如何用_at_关键字来定位不同类型的变量。

例 3.16　_at_关键字的使用。

```
int xdata xval _at_ 0x8000;     //全局变量 xval 存储在 xdata 区的地址为 0x8000 和 0x8001 单元
void main(void) {
    xval = 0x1234;              //赋值,不能在声明时进行
}
```

可使用下列语句来在另一个源文件中引用例 3.16 中用_at_修饰的变量。

例 3.17　_at_关键字修饰变量的引用。

```
extern int xdata xval;          //引入例 3.16 中声明的变量
void func(void){
    xval = 0x1000;              //重新赋值
}
```

注意:如果使用_at_关键字声明变量访问定义在 XDATA 区的外设,必须使用 volatile 关键字,以强制生成的代码去内存取硬件数据的实际值,而不是使用以前从硬件读入的保存在寄存器中的旧数据值。原因是外设数据的值可能会被硬件更改,必须确保 C 编译器不会将内存访问语句优化掉。

3.4.3　C51 数据类型

表 3-2 列出了 C51 语言支持的数据类型。

表 3-2　C51 语言支持的数据类型

数据类型	C51 专用	长　度	取值范围
signed char		单字节	－128～＋127
unsigned char		单字节	0～255
signed short		2 字节	－32 768～＋32 767
unsigned short		2 字节	0～65 535
signed int		2 字节	－32 768～＋32 767
unsigned int		2 字节	0～65 535
signed long		4 字节	－2 147 483 648～＋214 746 483 647
unsigned long		4 字节	0～4 294 967 295
float		4 字节	－3.402 823E＋38～－1.175 494E－38 或 1.175 494E－38～3.402 823E＋38
*		1～3 字节	对象的地址
enum		1 字节或 2 字节	－128～＋127 或－32 768～＋32 767
bit	专用	1 位	0 或 1
sbit	专用	1 位	0 或 1
sfr	专用	1 字节	0～255
sfr16	专用	2 字节	0～65 535

注意：在未使用 signed/unsigned 关键字定义整数型变量时，变量是有符号数，还是无符号数由使用的编译器和相关设置共同决定。

C51 中有几种 ANSI C 所没有的特殊数据类型，这些数据类型是和存储区域和存储器类型的概念密切相关的。

1. 特殊功能寄存器

8051 系列的微控制器提供了一个独立的内存区，用来存放特殊功能寄存器（Special Function Register，SFR）。

SFR 用来进行定时器、计数器、串行 I/O、I/O 端口等内部资源的工作控制。SFR 驻留在 0X80 到 0XFF 内部地址空间，可按字节寻址，某些寄存器还可以按位寻址或按字寻址。

由于 8051 系列的微控制器所拥有的 SFR 的数量、名称和类型是不完全相同的，因此 C51 使用 sfr、sfr16 和 sbit 关键字对 SFR 进行声明。

C51 编译器为常用的微控制器提供预先定义的 SFR 头文件（.h 文件）。编程时用户可以通过引用对应的头文件，来获取 SFR 定义信息。例如，对标准的 8052 芯片，可使用 #include <reg52.h>语句。当然用户也可自行定义 SFR 头文件，甚至为 SFR 进行不同的命名。

2. 8 位特殊功能寄存器（sfr）

sfr 关键字可以用来定义 8051 单片机的 8 位特殊功能寄存器。格式如下：

```
sfr 特殊功能寄存器名 = 特殊功能寄存器的地址;
```

SFR 的声明和 C 变量的声明格式是一样的，只不过使用的修饰符不是 char 或 int，而是 sfr。

例如：

```
sfr P0  =  0x80;                //Port - 0,对应地址为 80H
sfr P1  =  0x90;                //Port - 1,对应地址为 90H
```

```
sfr P2 = 0xA0;              //Port-2,对应地址为 0A0H
sfr P3 = 0xB0;              //Port-3,对应地址为 0B0H
```

P0、P1、P2、P3 是 sfr 声明的特殊功能寄存器的名称。等号后的常量表示 SFR 所在的内存地址。地址必须是数值常量,不允许使用带运算符的表达式。

特殊功能寄存器名称只要是一个合法的 C 标识符即可,但一般使用大写名称,并和芯片手册中的 SFR 的名称一致。

3. 16 位特殊功能寄存器(sfr16)

8051 芯片可以将两个 8 位 SFR 作为一个 16 位寄存器来访问。条件是这两个 SFR 必须处在相邻地址上,并且是低字节在高字节地址的前面。

C51 提供了 sfr16 关键字来进行 16 位特殊功能寄存器的声明,声明时低字节地址被用来作为 16 位特殊功能寄存器的地址。定义格式如下:

```
sfr16 特殊功能寄存器名 = 特殊功能寄存器的低字节地址;
```

例如:

```
sfr16 T2 = 0xCC;            //TL2 0CCH, TH2 0CDH
sfr16 RCAP2 = 0xCA;         //RCAP2L 0CAH, RCAP2H 0CBH
```

在这个例子中,T2 和 RCAP2 被声明为 16 位的特殊功能寄存器。

sfr16 声明和 sfr 声明的规则相同,等号后的地址是低字节所对应的地址。

4. 普通位变量(bit)

位变量(Bit Type)是指用一个二进制位表示的变量。位数据类型可以用来声明变量、参数表、函数返回值等。位数据变量声明和基本的数据类型声明一样,格式如下:

```
[存储种类] bit 变量名表;
```

所有的位变量都存储在内部数据区的可位寻址段中。因为该段只有 16 字节,所以在一个作用域内最多只能声明 128 个位变量。

位变量定义或声明时必须遵循以下规则:

(1) 禁止中断的函数(#pragma disable)和显式指定寄存器组(using n)的函数不能使用位变量返回值,否则编译器将产生一个错误信息。

(2) 不能将指针声明为指向一个位类型,同样也不能获取位变量的地址。

```
bit *ptr;                   //非法语句
```

(3) 不能声明位变量类型的数组。

```
bit ware[5];                //非法语句
```

例 3.18 位变量使用示例。

```
bit done_flag = 0;          //全局位变量
bit testfunc (              //函数返回值为位类型
    bit flag1,              //位类型参数
    bit flag2)
{
    bit ret;                //局部位变量
```

```
    ret = flag1&flag2;          //位变量运算
    return (ret);               //返回位类型值
}
```

5. 特殊位变量(sbit)

sbit 关键字有如下两种使用方式:

(1) 用来引用已经声明的可位寻址的对象的某一位。

```
sbit 位变量名 = 可位寻址变量名 ^指定的可寻址位的序号;

int bdata ibase;                //可位寻址的整型变量
char bdata cbase;               //可位寻址的字符型变量
long bdata lbase;
sbit mybit0 = ibase ^ 0;        //和 ibase 变量的位 0 (最低位)实现关联
sbit mybit15 = ibase ^ 15;      //和 ibase 变量的位 15 (最高位)实现关联
sbit bitc7 = cbase^ 7;          //和 cbase 变量的位 7 (最高位)实现关联
sbit bitl31 = lbase^ 31;        //和 lbase 变量的位 31 (最高位)实现关联
```

在上面的例子中的语句不是赋值语句,而是对 ibase、cbase、lbase 变量的特定位进行声明。表达式中在"^"符号后的表达式定义了位的位置。该表达式必须是一个常量。

注意:表达式的取值范围由变量声明中的基变量的数据类型来决定。对 char 和 unsigned char 类型,范围为 0～7;对 int、unsigned int、short、unsigned short 类型,为 0～15;对 long 和 unsigned long 为 0～31。

可以使用 bdata 来定义全局可位寻址变量和局部可位寻址变量。但由于 sbit 声明的变量必须为全局变量,因此 sbit 声明所使用的可位寻址变量必须为全局变量。

声明可位寻址对象的可寻址位时,必须使用 sbit 关键字,而不能使用 bit 关键字。

例 3.19 sbit 与 bit 的区别。

```
int bdata iData = 1;
sbit sbTest = iData ^0;         //位寻址变量必须为全局变量
//sbTest 和 iData 的末位绑定, 此时 sbTest = 1,即 iData 末位的值
void main(void)
{
    bit bTest = iData^0;        //运行结果: iData = 1(不变), sbTest = 1(不变),bTest = 1
    //bTest 的值是 iData 的值和 0 按位异或,并取最后一位的值
    iData = 32;                 //运行结果: iData = 32,sbTest = 0, bTest = 1(不变)
    bTest = 0;                  //运行结果: iData = 32(不变), sbTest = 0(不变), bTest = 0
    sbTest = 1;                 //运行结果: iData = 33,sbTest = 1, bTest = 0(不变)
}
```

(2) 用来引用已经声明的特殊功能寄存器对象的某一位。

在 8051 应用中,经常需要对 SFR 中的可寻址位(特殊功能位)进行独立访问。可以用 sbit 数据类型将 SFR 中的可寻址位声明为特殊功能位。

```
sbit 位变量名 = 可寻址位的位地址;
```

注意:不是所有的 SFR 都是可位寻址的。只有那些 SFR 地址能被 8 整除的特殊功能寄存器的是可位寻址的,即 SFR 二进制地址表示的低 3 位应全为 0。例如,在地址为 0XA8(IE)和 0XD0(PSW)的 SFR 是可以位寻址的,而地址为 0X81(SP)和 0X89(TMOD)的 SFR 不能

位寻址。

任何合法标识符均可用在 sbit 声明中。等号右边的表达式定义了标识符的绝对位地址。有 3 种方法来声明位地址：

方法一：sfr_name ^ int_constant，即 SFR 寄存器名^整型常量。

这种方法使用已经定义的 SFR 作为 sbit 的基地址。该 SFR 必须可位寻址，^符号后的表达式定义了可寻址位的位编号。位编号必须是 0～7 的数。

```
sfr PSW = 0xD0;              //声明寄存器名
sbit OV = PSW ^ 2;           //声明特殊功能位 OV
sbit CY = PSW ^ 7;           //声明特殊功能位 CY
sfr IE = 0xA8;               //声明寄存器名
sbit EA = IE ^ 7;            //声明特殊功能位 EA
```

为了计算 SFR 中位的地址，用位的编号加上 SFR 寄存器的字节地址。上例中 sbit EA＝IE ^ 7 等效于 sbit EA＝0xAF，EA 的位地址等于 IE 的 SFR 寄存器地址 0xA8 加上位编号 7，等于 0XAF。

注意：由于 sfr16 所对应的寄存器是两个相邻寄存器，因此不可能两个地址均是 8 的倍数，从而 sfr16 定义的寄存器一般不能位寻址。

方法二：int_constant^int_constant，即整型常量^整型常量。

这种方法使用整型常数作为基地址。该地址必须地址值在 0X80～0XFF 之间，并且可以被 8 整除。^符号后的表达式定义了可寻址位的位编号。位编号必须是 0～7 的数。

```
sbit OV = 0xD0 ^ 2;
sbit CY = 0xD0 ^ 7;
sbit EA = 0xA8 ^ 7;
```

方法三：int_constant，即用绝对位地址来声明 sbit。

```
sbit OV = 0xD2;
sbit CY = 0xD7;
sbit EA = 0xAF;
```

注意：sbit、bit 和 ANSI C 语言中的位域(Bit Field)是 3 种不同的数据类型。使用 sbit 声明时，基对象必须可位寻址变量或者是可以位寻址的特殊功能寄存器。

3.4.4 C51 指针类型

C51 的指针和标准 C 中的指针功能相同。但是由于 8051 体系结构的不同存储区域的地址有重叠(如 data 区和 xdata 区均从地址 0 开始编址)，因此必须要在指针中保存额外的存储区域信息。

根据存储区域信息保存方式的不同，C51 提供了两种不同类型的指针：通用指针(Generic Pointer)和具体指针(Memory-specific Pointer)。

1. 通用指针

通用指针可以用来保存位于不同存储区域中的相同数据类型变量的地址。通用指针的声明和标准 C 中的指针声明是相同的，例如：

```
char  * s;                   //指向字符类型的指针
```

```
int   * numptr;                //指向整型类型的指针
long   * state;                //指向长整型类型的指针
```

由于一般情况下,8051 存储区域的最大寻址范围不大于 64KB,因此通用指针总是占用 3 字节。第 1 字节保存存储器类型编码值(见表 3-3),第 2 字节保存地址的高字节,第 3 字节保存地址的低字节。许多 C51 的库例程使用这种指针类型,通用指针类型可以访问任何存储区域内的变量。

表 3-3　存储器类型编码值

存储器类型	idata/data/bdata	xdata	pdata	code
编码值	0x00	0x01	0xFE	0xFF

下列代码表示了不同存储区的通用指针变量的赋值过程。

例 3.20　通用指针的使用。

```
void main(void)
{
    char * c_ptr;              //通用字符指针
    char data dj;              //data 区字符变量
    char xdata xj;             //xdata 区字符变量
    char code cj = 9;          //code 区字符变量

    c_ptr = &dj;
    c_ptr = &xj;
    c_ptr = &cj;
}
```

2. 具体指针

具体指针是在声明时指定了存储器类型的指针,仅用于保存指定存储区域中的指定数据类型变量的地址。

```
char data * str;               //指向 data 区的字符变量的指针
int  xdata  * numtab;          //指向 xdata 区的整型变量的指针
long  code  * powtab;          //指向 code 区的长整型变量的指针
```

因为存储器类型在编译时就已经指定,所以和通用指针不同,具体指针不需要保存存储器类型字节。具体指针可以保存在 1 字节(idata、data、bdata、pdata 类型指针,这些区域的最大寻址范围不大于 256B)或 2 字节(code 和 xdata 类型指针,这些区域的最大寻址范围不大于 64KB)中。

例 3.21　具体指针的使用。

```
void main(void)
{
    char data * c_ptr;         //指向 data 区的字符变量的指针
    char data dj;              //data 区字符变量
    char xdata xj;             //xdata 区字符变量

    c_ptr = &dj;
```

```
    c_ptr = &xj;            //非法语句
}
```

定义具体指针变量时可以使用两个存储器类型，“*”前的存储器类型修饰指针指向的数据，“*”后的存储器类型修饰指针本身，即指针所占据的存储区域类型。例如：

```
char data * xdata str;        //指向 data 区的字符变量的指针,指针变量本身存储在 xdata 区
int xdata * data numtab;      //指向 xdata 区的整型变量的指针,指针变量本身存储在 data 区
```

注意：使用通用指针类型的代码和具体指针类型的代码相比，完成相同的功能代码的运行速度要慢很多。这是因为通用指针类型只有在程序运行时才能知道实际的变量存储区类型，所以编译器就不能对内存访问进行优化，从而只能生成可以访问任意存储区的通用代码。如果必须优先考虑程序的运行速度，那么只要有可能就应该使用具体指针来替代通用指针。

3.4.5 C51 函数定义

1. C51 函数完整声明

综上所述，完整的函数声明如下：

```
[return_type] funcname([args]) [{small|compact|large}]
[reentrant][interrupt x][using y]
```

2. 指定存储模式

C51 定义函数时可使用 small、compact 或 large 这 3 个 C51 关键字来指明函数内部使用的存储模式。

3. 可重入函数

一个可重入函数可以在同一个时刻由多个进程共享。即当一个进程正在执行一个可重入函数，另一个进程可以中断该进程，然后执行同一个可重入函数，而不会影响函数的运行结果。

ANSI C 调用函数时会把函数的调用参数和函数中使用的局部变量存入堆栈。而 C51 使用固定的存储空间(称为局部数据区)来存放相关数据。所以在递归调用仅使用局部变量的函数时，ANSI C 函数总是可重入的，C51 中的函数是不能重入的(局部数据区存储的数据会被覆盖)。

为此必须使用 reentrant 函数属性来声明函数是可重入的，以便 C51 编译器对函数进行特殊处理。格式如下：

```
函数类型 函数名(形式参数列表) reentrant
```

C51 编译器为可重入函数创建一个模拟堆栈(软件方式实现)来完成参数传递和局部变量存储，从而解决数据信息覆盖问题。可重入函数一般占用较大的内存空间，运行起来也比较慢，并且不允许传递 bit 类型的变量，也不能定义 bit 类型的局变量。

可重入函数经常在实时应用系统中应用，也可在中断响应函数和非中断响应函数同时调用同一个函数时使用。

下面以斐波那契数列(Fibonacci Sequence)为例看一下可重入函数的应用。斐波那契数列定义：$F(0)=0, F(1)=1, F(n)=F(n-1)+F(n-2)(n\geqslant 2, n\in \mathrm{N}^*)$。

例 3.22 可重入函数的使用。

```
int fib(int num) reentrant
{
    int ret = 0;
    if(num == 0)
        ret = 0;
    else if(num == 1)
        ret = 1;
    else
        ret = fib(num - 1) + fib(num - 2);
    return ret;
}
void main(void)
{
    int x;
    x = fib(3);
    return;
}
```

使用 reentrant 关键字,变量 x 的值为 2,正确。

不使用 reentrant 关键字,编译时有警告,变量 x 的值为 1,错误。

4. 中断响应函数

传统的 8051 处理器的中断有 5 种,某些 8051 兼容类型可以有更多的中断,C51 最大支持 32 个中断。

中断响应函数的定义如下:

```
函数类型 函数名(形式参数列表) interrupt n
```

它将函数定义为中断响应函数。中断属性带一个值为 0~31 的整型参数,用来表示中断响应函数所对应的中断号,该参数不能是带运算符的表达式。

注意:仅能在函数定义时使用 interrupt 函数属性,不能在函数声明时使用 interrupt 函数属性。

中断响应过程如下:

(1) 当中断产生时,首先由硬件实现返回地址(PC 值)压入堆栈操作。

(2) 然后中断响应函数被调用,用软件方式实现以下处理:

- 使用语句将 ACC、B、DPH、DPL、PSW 这些特殊功能寄存器的值将保存在堆栈中(如果对应寄存器在中断处理函数中未被使用则不保存,由编译器自动判断)。
- 如果中断响应函数未使用 using 属性进行修饰,中断响应函数中所使用的通用寄存器的值保存到堆栈中。
- 对中断进行处理。
- 恢复保存寄存器的值,退出中断响应函数(其对应的汇编代码使用 RETI 指令退出,普通函数使用 RET 指令退出)。

(3) 执行 RETI 语句,硬件实现返回地址的载入,并跳转到对应语句执行。

例 3.23 中断响应函数定义。

```
int alarm_count = 0;            //中断计数
```

```
void falarm(void) interrupt 1   //中断号 1 对应中断源 T0
{
        alarm_count++;          //每产生一次 T0 中断,计数值增加 1
}
```

中断响应函数应遵循以下规则:

- 中断响应函数不能进行参数传递。
- 中断响应函数没有返回值。
- 不能在其他函数中直接调用中断响应函数。
- 如果在中断中调用了其他函数,则必须保证这些函数和中断响应函数使用了相同的寄存器组,并且这些函数应为可重入函数。
- C51 编译器将在绝对地址 8n+3 中存放一个绝对跳转指令,实现对中断响应函数的调用,其中 n 为中断号。

5. 指定寄存器组

可使用 using 函数说明属性来规定函数所使用的寄存器组。格式如下:

```
函数类型 函数名(形式参数列表)  using n
```

using 属性使用一个值为 0~3 的整型参数,这个参数表示使用的寄存器组的编号,这个参数不能使用带运算符的表达式。using 属性只能在函数定义中使用,不能在函数原型声明中使用。

使用 using 属性的函数将完成以下操作:

- 进入函数前,将当前使用的寄存器组的标号保存在堆栈中。
- 更改 PSW 的寄存器组选择位,选择设定的寄存器组作为当前的寄存器组。
- 函数退出时,将寄存器组恢复成进入函数前的寄存器组。

例 3.24 寄存器组的使用。

```
void test1(void)
{         char idata x = 0x10;      }
void test2(void) using 1
{         char idata x = 0x11;      }
void main(void)
{
    test1();
    test2();
}
```

反汇编代码:

```
…
; FUNCTION test1(BEGIN)
MOV     R0,#LOW x
MOV     @R0,#010H
RET
; FUNCTION test1(END)

…
; FUNCTION test2(BEGIN)
```

```
PUSH    PSW
MOV     PSW,#08H
MOV     R0,#LOW x
MOV     @R0,#011H
POP     PSW
RET
; FUNCTION test2(END)
```

可以看出,test2 使用 using 关键字,会在函数内部对 PSW 寄存器的 RS1、RS0 位进行更改,上例中改成了 01,对应 using 1,从而使用 BNAK 1 的相关寄存器。

注意:使用 using 属性就不能通过寄存器来返回值了。必须很小心地使用 using 属性,以避免出现错误。另外,即使用相同的寄存器组,使用 using 属性声明的函数也不能返回 bit 值(bit 值是通过 CF 标志来返回的,使用 using 属性的函数在退出时,将恢复 PSW 字,CF 是 PSW 字中的一位)。

使用寄存器组切换技术可以提高程序的运行速度,若使用不当则会导致参数传递错误,代码维护也比较麻烦,建议不要使用。

3.4.6 C51 程序设计的注意事项

C51 编译器能对 C 程序源代码进行处理,产生高度优化的代码。注意下面一些问题,可以获取性能更好的代码。

- 采用短变量。减小变量的数据宽度提高代码效率的最基本的方法。使用 C 语言编程时,用户习惯于对循环控制变量使用 int 类型,这对 8 位的单片机来说是一种极大的浪费,应该仔细考虑变量值可能的范围,然后选择合适的变量类型。很明显,经常使用的变量应该是 unsigned char,只占用 1 字节。
- 避免使用浮点运算。在 8 位操作系统上进行 32 位浮点数运算速度是很慢的,所以如果需要使用浮点数,可以考虑是否使用整型运算来替代浮点运算。整型(长整型)的运算速度要比浮点数的运算速度要快得多。另外两个浮点数比较是否相等时,一般不使用==运算符,而是采用两个数的差小于一个极小值来判断。
- 使用位变量。对于逻辑值应使用位变量,这将节省内存的使用,提高程序的运行速度。
- 用局部变量代替全局变量。全局变量始终占用内存空间,因此使用全局变量会占用更多的内存空间。而且在中断系统和多任务系统中,可能会出现几个过程同时使用全局变量的情况,因而必须对全局变量进行保护,才能确保不会出现错误的运行结果。
- 尽量使用内部数据存储区。应把经常使用的变量放在内部数据存储区中,这可使程序的运行速度得到提高,缩短代码长度。考虑存储速度,应按下面的顺序使用存储器:DATA、IDATA、PDATA、XDATA。
- 使用具体指针。程序中使用指针时,应指定指针的类型,确定它们指向的存储区域,这样程序代码会更加紧凑,运行速度更快。
- 使用库函数。常用的和汇编指令对应的库函数有循环左移和循环右移(字符类型)_crol_、_cror_,(int 类型)_irol_、_iror_,(long 类型)_lrol_、_lror_ 以及空操作

nop。这些例程直接对应着汇编指令,因而速度更快。

- 使用宏替代函数。对于小段代码,如使能某些电路或从锁存器中读取数据,可通过使用宏来替代函数。这使得程序有更好的可读性。编译器在碰到宏时,用事先定义的代码去替代宏。当需要改变宏时,只要修改宏的定义。这可以提高程序的可维护性。
- 存储器模式。C51 提供了 3 种存储器模式,应该尽量使用小模式。小模式下编译出的代码运行速度较快,但可以使用的内存空间较小。如果既希望可以使用较大的内存空间,又希望部分函数有较快的运行速度,此时可以使用混合的存储模式。例如,将项目设置为大模式,将部分经常执行的函数显式声明为小模式。这样编译器将该函数的局部变量存储在内部数据区中,因而可以较快地执行。

习　题　3

1. 片外 RAM 1000H～10FFH 单元有一个数据块,用汇编语言编写程序将其传送到片外 RAM 的 2500H 单元开始的区域中。

2. 用汇编语言编写将片内 RAM 的 31H、30H 单元中的 16 位二进制数(31H 中为高位)求补码后放回原单元的程序。

3. 用汇编语言编写将累加器 A 中的一位十六进制数(A 的高 4 位为 0)转换为 ASCII 码的程序,转换结果仍存放在累加器 A 中,要求用查表和非查表两种方式实现。

4. 用汇编语言编程实现函数

$$y=\begin{cases}x+1, & x>10\\ 0, & 5\leqslant x\leqslant 0\\ -1, & x<5\end{cases}$$

设 x 的值存放在片内 RAM 的 35H 单元,y 的值存放在片内 RAM 的 36H 单元。

5. 假设累加器 A 中的内容为 0～5,编写根据累加器 A 的不同内容,转向不同分支进行处理的汇编语言程序。

6. 用汇编语言编写程序,将 R0 中的 8 位二进制数的各位用其 ASCII 码表示,结果保存放到片内 RAM 的 30H 开始的单元中。

7. 片内 RAM 的 HEXR 开始的单元中存放着一组十六进制数(一个单元放两位),数据的个数放在片内 RAM 的 LEN 单元中,用汇编语言编写程序将这些十六进制数转换为 ASCII 码,并存入片内 RAM 中 ASCR 开始的单元。

8. 程序存储器中有一个 5 行×8 列的表格,用汇编语言编程把行下标为 I、列下标为 J 的元素读入到累加器 A 中。

9. 用汇编语言编写程序,将累加器 A 中的 8 位二进制数转换为十进制数存放在片内 RAM 的 21H(百位)和 20H(十位和个位)单元中。

10. 用汇编语言编写程序实现图 3-8 所示的硬件逻辑功能。其中 P1.1、P1.2 和 P1.3 分别是端口线上的信息,IE0、IE1 为外部中断请求标志,25H 和 26H 为两个位地址。

11. 用汇编语言编程求两个无符号数据块中最大值的乘积。数据块的首地址分别为片内 RAM 的 60H 和 70H,每个数据块的第 1 字节用来存放数据块的长度。结果存入片内

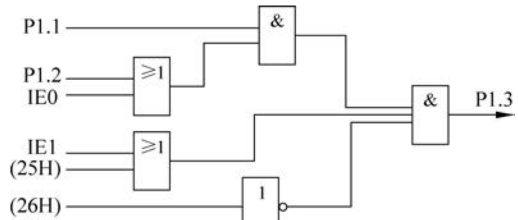


图 3-8　习题 10 电路图

RAM 的 5FH 和 5EH 单元中，要求求数据块中的最大值，用子程序实现。

12. 编写多字节无符号数加法子程序，入口参数为 R0 为被加数低位地址指针、R1 为加数低位地址指针、R2 为字节数。出口参数为 R0 为和的高位地址指针。

13. C51 有哪几种存储区域？如何将变量定义在这些区域中？如何进行绝对定位？

14. 存储种类、存储器类型各指什么？各自分为哪几类？

15. C51 的存储模式有哪几种？各有什么特点？

16. C51 有哪些特殊数据类型？

17. 什么是通用指针？什么是具体指针？两者各有什么优缺点？

18. C51 语言对函数定义进行了哪些扩展？