可用除法运算将二进制数转换为BCD 数。如把AL 中的 8 位无符号二进制数转换为BCD 数放入AX 中的程序段如下：

```
MOV CL，10
MOV AH，0              ；8 位二进制数扩展为 16 位二进制数
DIV CL
MOV CH，AH             ； 暂存BCD 数个位
MOV AH，0
DIV CL
MOV CL，4
SHL AH，CL             ；BCD 数十位移至高 4 位
OR CH，AH              ；BCD 数十位与个位拼合
MOV AH，0
MOV CL，10
DIV CL                 ；AH 中的余数为BCD 数百位
MOV AL，CH             ；BCD 数十位与个位送AL
```

用除 10 取余法将 8 位二进制数FFH 转换为BCD 数 255H 的二进制运算如图 3-3 所示。

```
        00011001          00000010          00000000
1010 ) 11111111   1010 ) 00011001   1010 ) 00000010
       1010               1010              0000
       ----               ----              ----
        1011               101                10
        1010
        ----
         1111
         1010
         ----
          101
```

图 3-3 8 位二进制数FFH 转换为BCD 数 255H 的二进制运算

除法指令对所有的状态标志位均无定义。

3.扩展指令CBW(convert byte to word)和 CWD(convert word to double word)

从除法指令的操作可知，要把一个 8 位二进制数除以一个 8 位二进制数，要有一个 16 位二进制数在AX 中，只是把一个 8 位的被除数放入AL 中是不行的，因为除法指令将把任何在 AH 中的数当作被除数的高8位。所以在做 8 位除以 8 位的除法之前先要把 8 位被除数扩展为 16 位，在做 16 位除以 16 位的除法之前要把 16 位被除数扩展为 32 位，才能保证除法指令的正确操作。这种扩展对于无符号数除法是很容易办到的，只需将AH 或 DX 清 0 即可。对符号整数除法就不能用将被除数的高半部清0 来实现，而要通过扩展符号位来把被除数扩展。例如把－2 的 8 位形式 1111 1110 转换为 16 位形式 1111 1111 1111 1110，即要把高半部全部置 1(－2 的符号位)；而把+3 的 8 位形式 0000 0011 转换成 16 位形式 0000 0000 0000 0011，却要把高半部全部置‘0’(+3 的符号位)。

指令格式 CBW

CWD

将字节扩展为字指令CBW 所执行的操作是把AL 的最高位扩展到AH 的所有位。将字扩展为双字指令CWD把 AX 的最高位扩展到DX 的所有位。在做 8 位除以 8 位和 16 位除以 16 位的除法之前，应先扩展AL 或 AX 中的被除数。

例如，在数据段中，有一符号字数组变量ARRAY，第 1 个字是被除数，第 2 个字是除数，接着存放商和余数，其程序段是：

```
MOV SI，OFFSET ARRAY
MOV AX，[SI]
CWD
IDIV WORD PTR 2 [SI]
MOV 4 [SI]，AX
MOV 6 [SI]，DX
```

一般情况下，都将符号数看作补码数，扩展指令和符号整数除法仅对补码数适用。若特别指出该符号数为原码数，则其扩展和除法运算都要另编程序段实现。

3.1.2 BCD 数调整指令

第 2.3 节介绍的加减指令和本节介绍的乘除指令都是对二进制数进行操作。二进制数算术运算指令对

BCD 数进行运算，会得到一个非BCD 数或不正确的BCD 数。如：

0000 0011B+0000 1001B= 0000 1100B

0000 1001B+0000 0111B= 0001 0000B

第一个结果是非BCD 数；第二个结果是不正确的BCD 数。其原因是BCD 数向高位的进位是逢 10 进 1，而 4 位二进制数向高位进位是逢 16 进 1，中间相差 6。若再加上 6，就可以得到正确的BCD 数：

0000 1100B+0000 0110B= 0001 0010B

0001 0000B+0000 0110B= 0001 0110B

8086/8088 对 BCD 数使用二进制数算术运算指令进行运算，然后执行一条能把结果转换成正确的BCD 数的专用调整指令来处理BCD 数的结果。

1.BCD 数加法调整指令DAA(decimal adjust for add)和 AAA(ASCII adjust for add)

指令格式 DAA

AAA

DAA 指令的意义是将 AL 中的数当作两个压缩 BCD 数相加之和来进行调整，得到两位压缩BCD 数。具体操作是，若(AL & 0FH) >9 或 AF=1， 则 AL 加上 6；若(AL & 0F0H) >90H 或CF=1，则 AL 加 60H。如：

```
MOV AX，3456H
ADD AL，AH          ；AL=8AH
DAA                 ；AL=90H
```

AAA 指令的意义是将AL 中的数当作两个非压缩BCD 数相加之和进行调整，得到正确的非压缩 BCD 数送AX。具体操作是，若(AL&0FH) >9 或AF=1，则(AL+6)& 0FH 送 AL，AH 加 1；否则 AL & 0FH 送 AL，AH 不变。应特别注意，AAA 指令执行前AH 的值。如：

```
MOV AX，0806H
ADD AL，AH          ；AX=080EH
MOV AH，0
AAA                 ；AX=0104H
```

又如：若要将两个 BCD 数的 ASCII 码相加，得到和的 ASCII 码，可以直接用 ASCII 码相加，加后再调整：

```
MOV AL，35H         ；‘5’
ADD AL，39H         ；‘9’，AL=6EH
MOV AH，0
AAA                 ；AX=0104H
OR AX，3030H        ；AX=3134H 即‘34’
```

由调整指令所执行的具体操作可以看到，对结果进行调整时要用到进位标志和辅助进位标志，所以调整指令应紧跟在BCD 数作为加数的加法指令之后。所谓“紧跟”是指在调整指令与加法指令之间不得有改变标志位的指令。

2.BCD 数减法调整指令 DAS(decimal adjust for subtract)和 AAS(ASCII adjust for subtract)

指令格式 DAS

AAS

DAS 指令的功能是将AL 中的数当作两个压缩BCD 数相减之差来进行调整，得到正确的压缩 BCD 数。具体操作是：若(AL & 0FH) >9 或 AF=1， 则 AL 减 6，(AL & 0F0H) >90H 或CF=1，则 AL 减 60H。如：

```
MOV AX， 5634H
SUB AL，AH          ；AL=DEH，有借位
DAS                 ；AL=78H，保持借位即 134-56
```

AAS 指令的功能是将AL 中的数当作两个非压缩BCD 数相减之差进行调整得到正确的非压缩BCD 数。具体操作是：若(AL & 0FH) >9 或 AF=1，则(AL-6) & 0FH 送 AL，AH 减 1；否则 AL & 0FH 送 AL，AH 不变。应特别注意，AAS 指令执行前AH 的值。如：

```
MOV AX，0806H
SUB AL，07H         ；AX=08FFH
AAS                 ；AX=0709H
```

3.非压缩BCD 数乘除法调整指令 AAM(ASCII adjust for multiply)和 AAD(ASCII adjust for divide)

压缩 BCD 数对乘除法的结果不能进行调整，故只有非压缩BCD 数乘除法调整指令。

指令格式 AAM

AAD

AAM 指令的功能是将AL 中的小于 64H 的二进制数进行调整，在AX 中得到正确的非压缩BCD 数。具体操作是AL/0AH 送 AH，AL MOD 0AH 送 AL。如：

```
MOV AL，63H
```

```
        AAM                 ; AX=0909H
```

AAD 指令的功能是将AX 中的两位非压缩BCD 数变换为二进制数。在做二位非压缩 BCD 数除以一位非压缩 BCD 数时，先将 AX 中的被除数调整为二进制数，然后用二进制除法指令 DIV 相除，保存 AH 中的余数后， 再用 AAM 指令把商变回为非压缩的BCD 数。如：

```
        MOV AX，0906H
        MOV DL，06H
        AAD                 ; AX=0060H
        DIV DL              ; AL=10H、AH=0
        MOV DL，AH          ; 存余数
        AAM                 ; AX=0106H
```

应注意的是，除法的调整不同于加法、减法和乘法，它们的调整是在相应运算操作之后进行，而除法的调整在除法操作之前进行。

调整指令都隐含着AX 或AL，都在AX 或 AL 中进行。下面举几个使用调整指令的例子。

例 3.1 已知字变量 W1 和W2 分别存放着两个压缩 BCD 数，编写求两数之和，并将其和送到 SUM 字节变量中的程序。

此例应注意以下两个问题：

(1)定义字变量 W1 和 W2 的 4 位数应为 BCD 数，其后要加 H，只有这样定义装入内存中的数据才是 4 位BCD 数。

(2)BCD 数的加减运算只能做字节运算，不能做字运算。这是因为加减指令把操作数都当作二进制数进行运算，运算之后再用调整指令进行调整，而调整指令只对 AL 作为目的操作数的加减运算进行调整。

程序如下：

```
stack       segment stack 'stack'
            dw 32 dup(?)
stack       ends
data        segment
W1          DW  8931H
W2          DW  5678H
```

```
SUM             DB 3 DUP(?)
data            ends
code            segment
begin           proc far
assume  ss: stack, cs: code, ds: data
                push ds
                sub ax, ax
                push ax
                mov ax, data
                mov ds, ax
    MOV AL, BYTE PTR W1; AL=31H
    ADD AL, BYTE PTR W2; AL=A9H, CF=0, AF=0
    DAA                ; AL=09H, CF=1
    MOV SUM, AL
    MOV AL, BYTE PTR W1+1  ; AL=89H
    ADC     AL,     BYTE     PTR     W2+1
;AL=E0H, CF=0, AF=1
    DAA                ; AL=46H, CF=1
    MOV SUM+1, AL
    MOV SUM+2, 0              ; 处理向万位的进
位
    RCL SUM+2, 1    ; 也可用指令 ADC SUM+2, 0
                ret
  begin         endp
  code          ends
```

```
                    end   begin
```

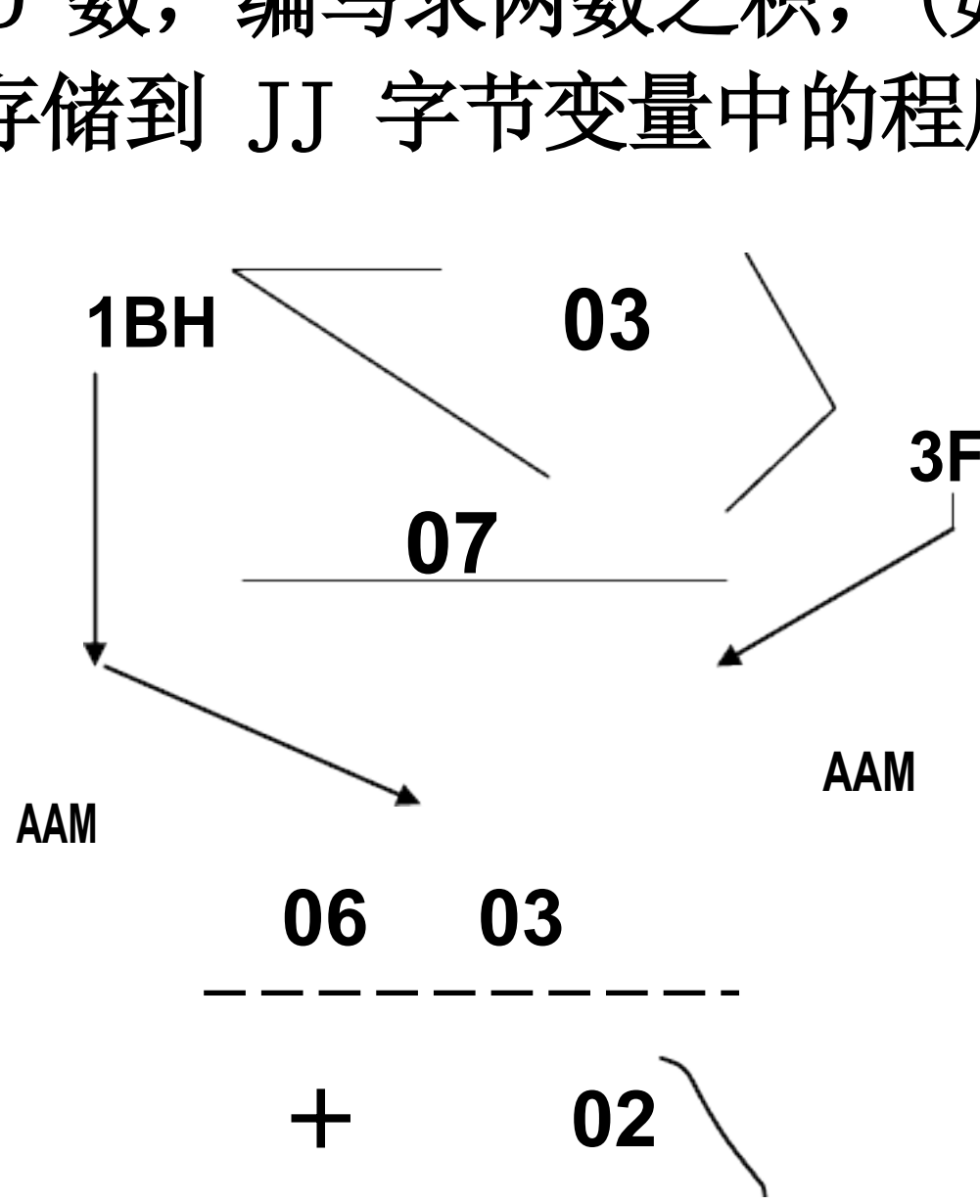
例 3.2 已知字变量 W1 和W2 分别存放着两个非压缩 BCD 数，编写求两数之和，并将其和送到 SUM 字节变量中的程序。

定义字变量W1 和W2 的数应为两位非压缩BCD 数，其后要加 H。程序如下：

```
          stack       segment stack 'stack'
                      dw 32 dup(?)
          stack       ends
          data        segment
          W1          DW  0809H
          W2          DW  0607H
          SUM         DB 3 DUP(?)
          data        ends
          code        segment
          begin       proc  far
                      assume  ss: stack, cs:
code, ds: data
                      push ds
                      sub ax, ax
                      push ax
                      mov ax, data
                      mov ds, ax
        MOV AL, BYTE PTR W1   ; AL=09H
        ADD AL, BYTE PTR W2   ;   AL=10H,
```

```
AF=1
                MOV AH, 0
                AAA                ; AL=06H, AH=01H
                MOV SUM, AL
                MOV AL, AH
                ADD AL, BYTE PTR W1+1   ; AL=09H
              ADD AL, BYTE PTR W2+1   ;      AL=0FH,
AF=0
                MOV AH, 0
                AAA                ; AL=05H, AH=01H
                MOV WORD PTR SUM+1, AX
                        ret
            begin       endp
            code        ends
                        end  begin
```

例 3.3 字变量W 和字节变量 B 分别存放着两个非压缩 BCD 数，编写求两数之积，(如 0307H 和 09H) 并将它存储到 JJ 字节变量中的程序。



```
stack       segment stack 'stack'
            dw 32 dup(?)
stack       ends
            data        segment
            W           DW 0307H
            B           DB 9
            JJ          DB 3 DUP(?)
            data        ends
            code        segment
            begin       proc far
                        assume ss:stack,cs:code,
ds: data
                        push ds
                          sub ax, ax
                          push ax
                          mov ax, data
                          mov ds, ax
                        MOV AL, BYTE PTR W      ;
AL=07H
                        MUL B                   ; AX=003FH
```

```
                    AAM             ;
AX=0603H
                    MOV WORD PTR JJ, AX
                    MOV AL, BYTE PTR W+1;
AL=03H
                    MUL B             ;
AX=001BH
                    AAM             ;
AX=0207H
                        ADD AL, JJ+1      ;
07H+06H=0DH, 即 AL=0DH
                        AAA           ;  调整后
的进位，直接加入 AH!AX=0303H
                    MOV WORD PTR JJ+1, AX
                    ret
          begin     endp
          code      ends
                    end begin
```

例 3.4 字变量W 和字节变量 B 中分别存放着两个非压缩 BCD 数，编制程序求二者的商和余数，并分别存放到字变量 QUOT 和字节变量 REMA 中。

定义字变量 W 的数应为两位非压缩 BCD 数，其后要加 H。

由于是 BCD 数的除法，所以要先调整，因此先将 W 中的非压缩 BCD 数取到 AX 中，然后将 AX 中的非

压缩BCD 数调整为二进制数。二进制数的除法之后，又应用 AAM 指令将结果调整为非压缩 BCD 数。AAM 指令是将 AL 中的小于 100 的二进制数调整为非压缩 BCD 数，存入 AX 中，因此，调整前应将除法产生的余数存入 REMA 中。

```
stack       segment stack 'stack'
            dw 32 dup(?)
stack       ends
            data        segment
            W           DW 0909H
            B           DB 5
            REMA        DB ?
            QUOT        DW ?
            data        ends
            code        segment
            begin       proc far
                        assume ss:stack,cs:code,
ds: data
                        push ds
                        sub ax, ax
                        push ax
                        mov ax, data
                        mov ds, ax
                          MOV AX, W
                        AAD     ; 0909H→63H
```

```
                    DIV B     ;
63H÷5=13H,,,,4,  AL=13H, AH=04H
                    MOV REMA, AH
                    AAM       ; 13H→0109H
                  MOV QUOT, AX
                  ret
          begin   endp
          code    ends
                    end begin
```

3.1.3 顺序程序设计举例

例 3.5 编程计算(W1－(W2×W3＋W4－5000))/W5→W6。式中 Wi (i=1～6)均为符号字变量。

```
          stack     segment stack 'stack'
                    dw 32 dup(?)
          stack     ends
          data      segment
          W1        DW 15000
          W2        DW 500
          W3        DW 800
          W4        DW 30000
          W5        DW 4000
          W6        DW 2 DUP(?)
          data      ends
          code      segment
          start     proc far
```

```
                    assume ss:stack, cs:code,
ds:data

                    push ds
                    sub ax, ax
                    push ax
                    mov ax, data
                    mov ds, ax
                    MOV AX, W2
                    IMUL W3
                    MOV BX, AX
                    MOV CX, DX
                    MOV AX, W4
                    CWD
                    ADD BX, AX
                    ADC CX, DX
                    SUB BX, 5000
                    SBB CX, 0
                    MOV AX, W1
                    CWD
                    SUB AX, BX
                    SBB DX, CX
                    IDIV W5
                    MOV W6, AX
                    MOV W6+2, DX
                    ret
```

```
start          endp
code           ends
               end start
```

例 3.6 从键盘上键入 0～9 中任一自然数 N，将其立方值送显示器显示。

求一个数的立方值可以用乘法运算实现，也可以用查表法实现。查表法运算速度比较快，是常用的计算方法。因只要送显示，故将 0～9 的立方值的 ASCII 码按顺序造一立方表。立方值最大值为 729，需三个单元存放它的 ASCII 码，表的每项的单元数相同，再在每项之后加一个‘$’，所以立方表的每项均占 4 个字节。根据这种存放规律可推知，表的偏移首地址与自然数 N 的4 倍之和，正是N 的立方值和$的 ASCII 码的存放单元的偏移首地址。程序如下：

```
stack          segment stack 'stack'
                  dw 32 dup(?)
    stack         ends
          data          segment
          INPUT         DB 'PLEASE INPUT N(0-9):
$'
          LFB                                   DB   '
0$  1$  8$ 27$ 64$125$216$343$512$729$'
          N             DB  ?
          data          ends
```

```
code        segment
start       proc far
            assume ss:stack, cs:code,
ds:data

            push ds
            sub ax, ax
            push ax
            mov ax, data
            mov ds, ax
            MOV DX, OFFSET INPUT   ;显
示提示信息
            MOV AH, 9
            INT 21H
            MOV AH, 1              ;  键
入并回显 N（1 号功能调）
            INT 21H
            MOV N, AL
            MOV AH, 2              ;  换行
（2 号功能调用）
            MOV DL, 0AH
            INT 21H
            MOV DL, N
            MOV DL, 0FH            ;将′N′转
换为 N
            MOV CL, 2              ;  将 N
```

```
乘以 4
                SHL DL，CL
                  MOV DH，0                    ；8 位
4N 扩展为 16 位
                      ADD   DX ，  OFFSET  LFB
        ；4N+表的偏移地址
                      MOV AH，9
                      INT 21H
                      ret
        start         endp
        code          ends
                      end start
```

例 3.7 两个 32 位无符号数的乘法程序。

8086/8088 只有 16 位运算指令，所以该问题无法直接用乘法指令实现，但可以用 16 位乘法指令做四次乘法，然后把部分积相加，如图 3-4 所示，相应的程序如下。

图 3-4 32 位无符号数的乘法

```
        stack         segment stack 'stack'
                      dw 32 dup (?)
        stack         ends
        data          segment
        AB            DD 12345678H
        CD            DD 12233445H
```

```
ABCD        DD 2 DUP(?)
data        ends
code        segment
start       proc far
            assume ss:stack, cs:code,
ds:data

            push ds
            sub ax, ax
            push ax
            mov ax, data
            mov ds, ax
            MOV BX, OFFSET AB
              MOV AX, [BX+4]
        ; d→AX, AX=3445H
              MUL WORD PTR [BX]          ;
d×b
              MOV [BX+8], AX             ;
存 db 的低 16 位
              MOV [BX+10], DX          ;存
db 的高 16 位
              MOV AX, [BX+4]             ;
d→AX; AX=3445H
              MUL WORD PTR [BX+2]        ;
d×a
              ADD [BX+10], AX          ;
```

```
d×a 的低 16 位加上 d×b 的高 16 位
                        ADC DX, 0                    ; 上
述加法若有进位,则加入 d×a 的高 16 位中
                        MOV [BX+12], DX              ;存
d×a 的高 16 位
                        MOV AX, [BX+6]               ;
c→AX; AX=1223H
                        MUL WORD PTR [BX]            ;
c×b
                        ADD [BX+10], AX              ;
c×b 的低 16 位加入
                        ADC [BX+12], DX              ;
c×b 的高 16 位加入存放单元
                        MOV BYTE PTR [BX+14], 0      ;
清 0a×c 的高 16 位的存放单元
                        ADC BYTE PTR [BX+14], 0      ;
c×b 的高 16 位加入时产生的进位存入
                        MOV AX, [BX+6]               ;
c→AX; AX=1223H
                        MUL WORD PTR [BX+2]          ;
a×c
                        ADD [BX+12], AX              ;
a×c 的低 16 位加入
                        ADC [BX+14], DX              ;
a×c 的高 16 位加入
```

```
                ret
start           endp
code            ends
                end start
```

3.2　分支程序设计

顺序程序的特点是从程序的第一条指令开始，按顺序执行，直到最后一条指令。然而，许多实际问题并不能设计成顺序程序，需要根据不同的条件作出不同的处理。把不同的处理方法编制成各自的处理程序段，运行时由机器根据不同的条件自动作出选择判别，绕过某些指令，仅执行相应的处理程序段。按这种方式编制的程序，执行的顺序与指令存储的顺序失去了完全的一致性，称之为分支程序。分支程序是机器利用改变标志位的指令和转移指令来实现的。

转移指令有 JMP 和 Jcond 两类。前者是无条件转移，后者是条件转移。JMP 指令将控制转向其后的目的标号指定的地址。条件转移指令伴随着能改变可测试条件标志的指令之后，根据设置条件决定是否将控 制转向其后的目的地址。

3.2.1 条件转移指令

指令格式 Jcond short-lable

该指令的功能是，若条件满足则short-lable 的偏移地址送IP，否则顺序执行。

条件转移指令是相对转移指令，都是两字节的指令。相对转移指令的转移范围，即从当前地址（执行该指令时的IP 值）到目的标号地址的偏移量为－128～127(从条件转移指令的地址到目的标号的地址则为－126～129)，故条件转移指令只能实现段内转移。

1.简单的条件转移指令

简单的条件转移指令是仅根据一个可测试标志位实现转移的指令。简单的条件转移指令如表3-1 所列。

表 3-1 简单的条件转移指令

指令助记符	功能	标志设置
JE/JZ	相等/等于 0 转移	ZF=1
JNE/JNZ	不相等/不等于 0 转移	ZF=0
JC	有进(借)位转移	CF=1
JNC	无进(借)位转移	CF=0
JS	为负转移	SF=1
JNS	为正转移	SF=0
JO	溢出转移	OF=1
JNO	无溢出转移	OF=0
JP/JPE	偶转移	PF=1
JNP/JPO	奇转移	PF=0

2.无符号数条件转移指令

条件转移指令常根据比较指令比较两个数的关系的结果来实现转移。两个数的关系除了相等与否外，还有两个数中哪一个比较大。但这就有一个有趣的问题，如8 位二进制数 11111111 大于 00000000 吗? 答案既可肯定又可否定。因为若视这两个二进制数为无符号数，11111111 当然大于 00000000；若视这两个二进制数为符号数（补码），11111111 为－1，就比0 小了。为此要使用两种术语来区分无符号数和符号数的这种关系。如果把数作为符号数来比较，就使用术语“小于”和“大于”；如果把数作为无符号数来比较，就使用术语“低于”和“高于”。因此8 位二进制数 11111111 高于 00000000，小于00000000，而00000001既高于又大于 00000000。所以 8086/8088 设置了符号数的条件转移指令和无符号数的条件转移指令。

无符号数条件转移指令有 4 条，如表 3-2 所列。

表 3-2 无符号数条件转移指令

指令助记符	功能
JB/JNAE	低于/不高于等于转移
JNB/JAE	不低于/高于等于转移
JA/JNBE	高于/不低于等于转移
JNA/JBE	不高于/低于等于转移

已知 AL 中有一十六进制数的ASCII 码，将它转换为十六进制数需判别该ASCII 码是 0～9 的 ASCII 码还是 A～F 的ASCII 码，可以用无符号数条件转移指令来判别，其程序段为：

```
CMP AL，'A'
```

```
            JB NS7              ；AL 低于A 的ASCII 码去NS7
            SUB AL，7
NS7：       SUB AL，30H
            ┇
```

也可以用简单的条件转移指令

```
            JC NS7
```

代替无符号数条件转移指令。因为AL 低于A 的 ASCII 码，AL 与‘A’比较（即相减）后有借位。

1. 符号数条件转移指令

符号数条件转移指令有 4 条，如表 3-3 所列。

表 3-3 符号数条件转移指令

指令助记符	功能
JL/JNGE	小于/不大于等于转移
JNL/JGE	不小于/大于等于转移
JG/JNLE	大于/不小于等于转移
JNG/JLE	不大于/小于等于转移

3.2.1 无条件转移指令

条件转移指令的转移范围为-128～127，若超过这个范围时，就要在这个范围的某处放一条无条件转移指令来实现转移。无条件转移指令没有范围限制。在分支程序中还要用无条件转移指令将各分支又重新汇集到一起。

无条件转移指令有直接转移和间接转移两类。

1.无条件直接转移指令

格式　JMP target

功能　将控制转向目的标号 target，即 target 的偏移地址送 IP，target 的段首址送 CS(若 target 与该指令在同一段，则无此操作，只改变IP)。

2.无条件间接转移指令

格式　JMP dest

目的操作数可为寄存器和存储器。若为寄存器，则是将寄存器的内容送 IP。存储器操作数若为字变量，则是将字变量送IP（仅能实现段内转移)；若为双字变量，则是将双字送CS 和 IP（可实现段间转移)。如：

JMP W（W 为字变量）的操作是将W 的内容送IP；

JMP DUW（DUW 为双字变量）的操作是将WORD PTR DUW 的内容送 IP、WORD PTR DUW+2 的内容送 CS。又如：

JMP WORD PTR ［BX］的操作是将［BX+1］和［BX］送 IP；

JMP DWORD PTR ［BX］的操作是将［BX+1］和［BX］送 IP；［BX+3］和［BX+2］送 CS。

3.2.2 分支程序设计举例

分支的实现有多种方法，这里仅介绍两种基本方法：利用比较转移指令实现分支和利用跳转表实现分支。

例 3.8　编制计算下面函数值的程序(X、Y 均为字节符号数)。

$$Z=\begin{cases}1 & X\geq 0,\ Y\geq 0\\ -1 & X<0,\ Y<0\\ 0 & X,Y\text{异号}\end{cases}$$

根据题意，先判 x、y 是否异号，异号 Z 赋 0 后结束；若不异号即同号，则只需再判其中任一数的符号即可得知 X 和 Y 是大于等于 0 还是小于 0。使用 XOR 指令判别 X、Y 是否异号，XOR 指令执行 X、Y 按位加，X 和Y 的符号位按位加，若 X、Y 同号则按位加结果为 0 即 SF=0，若 X、Y 异号则按位加结果为 1 即 SF=1。为了减少分支，采用先赋值后判的方法。赋 0 和 1 是用 MOV 指令完成的，赋-1 是用对 1 求补即用求补指令 NEG 完成的。

```
stack       segment stack 'stack'
            dw 32 dup(?)
stack       ends
data        segment
X           DB -5
Y           DB 20
Z           DB ?
```

```
data        ends
code        segment
start       proc far
            assume ss:stack, cs:code, ds:data
            push ds
            sub ax, ax
            push ax
            mov ax, data
            mov ds, ax
            MOV AL, X
            XOR AL, Y              ; 根据X、Y 的符号置S 标志，相同为 0
            JS DIFF                ; 相异为 1，X、Y 相异去DIFF
            MOV Z, 1
            CMP X, 0               ;  相同后，判断其中某数的符号
            JNS NOCHA              ; 大于等于 0 结束
            NEG Z                  ; 小于 0；Z 赋-1 结束
NOCHA:      RET
DIFF:       MOV Z, 0
            ret
start       endp
code        ends
            end start
```

例 3.9　某工厂的产品共有 8 种加工处理程序 P0～P7，而某产品应根据不同情况，作不同的处理，其选择由键入的值 0～7 来决定。若键入 0～7 以外的键，则退出该产品的加工处理程序。

```
stack       segment stack 'stack'
            dw 32 dup(?)
stack       ends
data        segment
INPUT       DB 'INPUT(0～7):$ '
data        ends
code        segment
start       proc far
            assume ss:stack, cs:code, ds:data
            push ds
            sub ax, ax
            push ax
            mov ax, data
            mov ds, ax
AGAIN:      MOV DX, OFFSET INPUT
            MOV AH, 9
            INT 21H
            MOV AH,1
            INT 21H
            CMP AL, '0'
            JE  P0
            CMP AL, '1'
            JE P1
            ⋮
            CMP AL, '7'
            JE P7
            RET
P0:
```

```
                ⋮
                JMP AGAIN

                ⋮
P7:
                ⋮
                JMP AGAIN
start           endp
code            ends
                end start
```

该程序的编制方法是利用比较转移指令实现分支，每次比较转移实现二叉分支。这种方法编程条理清楚，容易实现。但各处理程序不能太长，且分支不能太多。因为分叉进入各处理程序所用的指令均是条件转移指令(此例为 JE Pi)，条件转移指令所允许的转移范围为－128～127，若各处理程序较长或者分支再多一些，就会超过条件转移指令所允许的范围。为了解决这个问题，可以利用跳转表法来实现这种多叉分支。

跳转表法实现分支的具体做法是，在数据区中开辟一片连续存储单元作为跳转表，表中顺序存放各分支处理程序的跳转地址。跳转地址在跳转表中的位置，即它们在表中的偏移始地址等于跳转表首地址加上它们各自的序号与所占字节数的乘积。要进入某分支处理程序只须查找跳转表中相应的地址即可。用跳转表法编制的程序如下。

```
stack           segment stack 'stack'
                dw 32 dup(?)
stack           ends
data            segment
INPUT           DB 'INPUT(0-7)：$'
PTAB            DW P0, P1, P2, P3, P4, P5, P6, P7
data            ends
code            segment
start           proc far
                assume ss:stack, cs:code, ds:data
                push ds
                sub ax, ax
                push ax
                mov ax, data
                mov ds, ax
AGAIN:          MOV DX, OFFSET INPUT
                MOV AH, 9
                INT 21H
                MOV AH, 1
                INT 21H
                CMP AL, '0'
                JB EXIT
                CMP AL, '7'
                JA EXIT
                AND AX, 0FH
                ADD  AL, AL
                MOV  BX, AX
                JMP PTAB [BX]
EXIT:           RET
P0:
                ⋮
                JMP AGAIN
```

```
                ⋮
        P7:
                ⋮
                JMP AGAIN
        start   endp
        code    ends
                end start
```

3.3 循环程序设计

顺序程序和分支程序中的指令，最多只执行一次。在实际问题中重复地做某些事的情况是很多的，用计算机来做这些事就要重复地执行某些指令。重复地执行某些指令，最好用循环程序实现。

例如，例 2.4（将 AX 的小于 255 大于 0 的 3 位 BCD 数转换为二进制数，存入字节变量SB 中）可由下面的循环程序实现：

```
stack   segment stack 'stack'
        dw 32 dup(?)
stack   ends
data    segment
SB      DB ?
data    ends
code    segment
start   proc far
        assume ss:stack, cs:code, ds:data
        push ds
        sub ax, ax
        push ax
        mov ax, data
        mov ds, ax
        MOV CX, 8                ; 循环准备
AGAIN:  SHR AX, 1                ; 循环体
        RCR SB, 1
        MOV DL, AL
        AND DL, 88H
        SHR DL, 1
        SHR DL, 1
        SUB AL, DL
        DEC CX                   ; 循环的控制与修改
        JNZ AGAIN
        ret
start   endp
code    ends
        end start
```

从此例可以看出循环程序一般有 4 部分：

（1）循环准备　亦称循环初始化，它为循环作必要的准备。这部分的主要工作是建立地址指针，置计数器，设置些必要的常数，将工作寄存器或工作单元清0 等。

（2）循环体　完成循环的基本操作，是循环程序的实质所在。

（3）循环的修改　修改或恢复某些内容，为下一轮循环做好必要的准备。修改的内容一般包括计数器、寄存器和基址或变址寄存器。有的循环还要恢复某些计数器，寄存器和基址或变址寄存器。

（4）循环的控制　修改计数器，判断控制循环的继续或终止。

任何循环程序一般都应有这 4 部分，但各部分的界限并不是很清楚的。有时为设计方便或为了节省存储空间或为了控制简单等原因，这4 部分形成相互包含、相互交叉的情况，很难分出某条或某几条指令究竟属于哪一部分。

3.3.1 循环程序的基本结构

循环程序的基本结构有两种，如图3-5 所示。一种（a）是“先执行，后判断”，这种结构的循环先执行一次循环体，后判断循环是否结束。这种结构的循环至少执行一次循环体，上面所举程序属于这种结构。另一种（b）是“先判断，后执行”，这种结构的循环首先判断是否进入循环，再视判断结果，决定是否执行循环体。这种结构的循环，如果一开始就满足循环结束的条件，会一次也不执行循环体，即循环次数可 以为 0，若能确保一个循环程序在任何情况下都不会出现循环次数为0 的情况，采用以上任一种结构都可以；当不能确保时，用后一种结构为好。

图 3-5 循环程序的基本结构

例如，编程统计字变量 W 中有多少位 1。

这个程序最好采用“先判断，后执行”的结构。先将W 送 AX，判 AX 是否为 0，如果 AX=0，则不必做统计工作了；如果AX≠0，则将AX 左移或者右移 1 位，通过判移出位是 1 还是 0，决定字节变量N 是否加 1 来统计W 中1 的位数。其程序如下：

```
stack       segment stack 'stack'
            dw  32 dup(?)
stack       ends
data        segment
W           DW 1999H
N           DB ?
data        ends
code        segment
start       proc far
            assume ss:stack, cs:code, ds:data
            push ds
            sub ax, ax
            push ax
            mov ax, data
            mov ds, ax
            MOV N, 0
            MOV AX, W
LOP:        AND AX, AX
            JZ DONE
            SHL AX, 1
            JNC NOINC
            INC N
NOINC:      JMP LOP
DONE:       ret
start       endp
code        ends
            end start
```

3.3.2 重复控制指令

循环程序必须要有指令来控制循环，重复控制指令在循环的首部或尾部确定是否进行循环。确定是否循环的方法通常是在计数寄存器CX 中预置循环次数，重复控制修改CX，再判断 CX。当 CX 不等于 0 时，循环至目的地址；否则顺序执行该重复控制指令的下一条指令。重复控制指令同条件转移指令一样，也是 相对转移指令，重复控制指令的目的地址必须在本指令地址的－126～129 字节的范围之内。这些指令对串操作和数据块操作是很有用的。重复控制指令有下述4 条。

1.LOOP 指令

指令格式 LOOP short-lable

指令的意义是将CX 减 1，然后判断 CX 是否等于 0。若 CX≠0，则控制程序转移到short-lable 所指的指令，否则顺序执行。

使用 LOOP 指令之前，必须把循环次数送入CX 寄存器中，一条 LOOP short-lable 指令，相当于 DEC CX 和 JNZ short-lable 两条指令。使用LOOP 指令实现‘0’次循环必须使用图 3-5(b)的结构，且在循环准备时将 CX 置 1。若将 CX 置 0，则循环要进行 65536 次。其原因是执行LOOP 指令时，CX 先减 1，后判断 CX 是否为 0。

2.LOOPZ / LOOPE 指令

指令格式　LOOPZ short-lable 或 LOOPE short-lable

指令意义是先将 CX 减 1，然后判断 CX 的内容和 ZF 标志的状态。若 CX≠0，且 ZF=1 时，将程序转移到 short-lable 所指的指令，否则顺序执行。

3.LOOPNZ / LOOPNE 指令

指令格式　LOOPNZ short-lable 或 LOOPNE short-lable

指令意义是先将 CX 减 1，然后判断 CX 的内容和 ZF 标志的状态。若 CX≠0，且 ZF=0 时，将程序转移到 short-lable 所指的指令，否则顺序执行。

4.JCXZ 指令

指令格式　JCXZ　short-lable

指令意义是若CX=0，则将程序转移到short-lable 所指的指令，否则顺序执行。

3.3.3 单重循环程序设计举例

1.计数控制的循环程序

此类循环程序的特点是循环次数已知，故可用某个寄存器或存储单元作为计数器，用计数器的值来控制循环的结束。

例 3.10 计算 Z=X+Y 其中X 和Y 是双字变量。

双字变量占 4 个字节，故其和可能占 5 个字节，字节相加的程序如下：

```
stack       segment stack 'stack'
            dw 32 dup (?)
stack       ends
data        segment
X           DD 752028FFH
Y           DD 9405ABCDH
Z           DB 5 DUP(?)
data        ends
code        segment
start       proc far
            assume ss:stack, cs:code, ds:data
            push ds
            sub ax, ax
            push ax
            mov ax, data
            mov ds, ax
            MOV CX, 4
            MOV SI, 0
            AND AX, AX                          ; 清 CF，即 CF 为 0
AGAIN:      MOV AL, BYTE PTR X [SI]
            ADC AL, BYTE PTR Y [SI]
            MOV Z [SI], AL
            INC SI
            LOOP AGAIN
            MOV Z [SI], 0
            RCL Z [SI], 1
            ret
start       endp
code        ends
            end start
```

例 3.11 编写将某数据区中的十六进制数加密的程序，每个数字占一个字节。

在实际的应用中为了对某些信息保密，通常可通过硬件或软件的方法对信息进行加密，使用时再进行解密。软件的加密和解密是通过运行加密和解密程序实现的。加密程序是用与原数字对应的加密表中的信息代替原数字。解密程序则是通过解密表将加密信息还原。表3-4 是任意设计的十六进制数字的加密数和相应的解密数。

表 3-4 十六进制数字的加密数和相应的解密数表

十六进制数	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

加密数	A	9	8	E	F	1	0	B	2	5	D	3	7	4	6	C
解密数	6	5	8	B	D	9	E	C	2	1	0	7	F	A	3	4

加密程序如下：

```
stack       segment stack 'stack'
            dw 32 dup (?)
stack       ends
data        segment
HEXS        DB 1,2,,,,,0EH
NUMBER      EQU $－HEXS
JMB         DB 0AH,9,8,0EH,0FH,1,0,0BH,2,5,0DH,3,7,4,6,0CH
JMHEX       DB NUMBER DUP(?)
data        ends
code        segment
start       proc far
            assume ss:stack, cs:code, ds:data
            push ds
            sub ax, ax
            push ax
            mov ax, data
            mov ds, ax
            MOV BH, 0                   ; JMB 表中的位移量的高 8 位为 0
            MOV SI, 0                   ; HEXS 和 JMHEX 两个数据区的位移量
            MOV CX, NUMBER
AGAIN:      MOV BL, HEXS [SI]           ; 取十六进制数
            MOV AL,JMB[BX]              ; AL← [BX+JMB]（十六进制数的加密数）
            MOV JMHEX [SI], AL
            INC SI
            LOOP AGAIN
            ret
start       endp
code        ends
            end start
```

例 3.12 将字节变量SB 中的 8 位二进制数送显示器显示。

先将字节变量中的 1 位二进制数移入AH 中，再将移入的二进制数变为ASCII 码。为了避免通过CF 来传递二进制数，先将SB 中的 8 位二进制数送入AL 中，再左移AX，将 1 位二进制数直接移入AH 中。程序如下：

```
stack segment stack 'stack'
            dw 32 dup(?)
stack       ends
data        segment
SB DB 9AH
OBUF DB 9 DUP(?)
data        ends
code        segment
start       proc far
            assume ss:stack, cs:code, ds:data
            push ds
            sub ax, ax
            push ax
            mov ax, data
            mov ds, ax
            MOV CX, 8
            MOV BX, 0
            MOV AL, SB
```

```
AGAIN:      MOV AH, 0
            SHL AX, 1
            ADD AH, 30H
            MOV OBUF[BX], AH
            INC BX
            LOOP AGAIN
            MOV OBUF [BX], '$'
            MOV DX, OFFSET OBUF          ; 将输出数据区的偏移首地址送DX
            MOV AH, 9
            INT 21H
            ret
start       endp
code        ends
            end start
```

例 3.13 编写将键入的十进制数(-32768～32767)转换为二进制数的程序。

将 i 位十进制整数转换为二进制数的方法有多种，其中之一是使用算法（ 0×10+a)×10+,,)×10+a 。

i-1 0

例如，将 548 转换为二进制数，计算机执行的二进制运算如图3-6 所示。

```
   5×10          (5×10)+4      (5×10+4)×10     (5×10+4)×10+8
  00000101       00110010        00110110        1000011100
×     1010     +      100     ×     1010      +       1000
-----------    ----------    ------------    ------------
      101         00110110        110110        1000100100
+   101                       +110110
-----------                   ------------
  00110010                    1000011100
```

图 3-6 将 548 转换为二进制数的二进制运算

图 3-6 中的计算机运算的结果是：10 0010 0100B=224H，即 548=224H。

非计算机计算即将十进制数转换为十六进制数的计算如下：

548=512+32+4=200H+20H+04H=224H

用该方法将键入的十进制数转换为二进制数比较适宜，因为键入的十进制数是一个单元一位按高位到低位的顺序存放在数据存储区中的，且十进制数的位数也是已知的。在转换之前，先判别该数是正数还是负数。为简化设计，正数则按习惯不键入+号。若是负数，则十进制数的位数要比键入的字符数少一位； 另外在转换完后，还要将转换的结果进行求补。

数据段中定义两个变量：IBUF 和 BINARY。IBUF 共计定义 9 个单元用来存放键入的十进制字符串，因为键入的字符串连同负号最多 6 个字符，加 1 个回车，共计 7 个。另外，根据 10 号功能调用的入口参数的要求，还要在第 1 单元装入允许键入的字符数，并预留第 2 单元给 10 号功能调用存放实际键入的字符数。字变量BINARY 用来存放转换的结果。程序如下：

```
stack       segment stack 'stack'
            dw 32 dup(?)
stack       ends
data        segment
IBUF        DB 7, 0, 7 DUP(?)
BINARY      DW 0
data        ends
code        segment
start       proc far
            assume ss:stack, cs:code, ds:data
            push ds
            sub ax, ax
            push ax
            mov ax, data
            mov ds, ax
            MOV DX, OFFSET IBUF     ; 键入十进制数（10 号功能调用）
            MOV AH, 10
```

```
                INT 21H
                MOV CL, IBUF+1          ; 十进制数（含“-”号）的位数送CX
                MOV CH, 0
                MOV SI,OFFSET IBUF+2    ; 指向键入的第一个字符
                CMP BYTE PTR [SI], '-'  ; 判是否为负数
                PUSHF                   ; 保护零标志，供转换之后再判别
                JNE SININC              ; 正数跳转，去SININC
                INC SI                  ; 越过“-”号指向数字
                DEC CX                  ; 实际字符数少 1（“-”号）
SININC:         MOV AX, 0               ; 开始将十进制数转换为二进制数
AGAIN:          MOV DX, 10              ; ((0×10+a4)×10+„)×10+a0
                MUL DX
                AND BYTE PTR [SI], 0FH  ; 将十进制数的ASCII 码转换为BCD 数
                ADD AL, [SI]
                ADC AH, 0
                INC SI
                LOOP AGAIN
                POPF                    ; 恢复判断是否为负数时的零标志ZF
                JNZ NNEG                ; 非 0 即为正数,则不求补
                NEG AX                  ; 负数对其绝对值求补
NNEG:           MOV BINARY, AX          ; 存放结果
                ret
start           endp
code            ends
                end start
```

例 3.14 多位非压缩BCD 数与一位非压缩BCD 数相乘。

定义 3 个字节变量分别存放被乘数即多位非压缩BCD 数、乘数即一位非压缩BCD 数和积。被乘数和积的存放按高位在高地址、低位在低地址顺序存放。从被乘数的最低位开始，逐位与乘数相乘得到部分积， 将部分积相加即可得最终结果。部分积的相加不是在求得所有的部分积后再进行，而是逐次累加。求得部 分积后，将它的两位非压缩BCD 数中的低位与上一次部分积的高位相加后保存到积的一个存放单元中，应注意的是这是非压缩BCD 数的相加，故加后要进行调整，调整所产生的进位正好加入部分积的高位AH 中； 部分积的高位暂存DH 中，待下一次乘法后，与部分积的低位相加。程序如下：

```
stack       segment stack 'stack'
            dw 32 dup(?)
stack       ends
data        segment
FIRST       DB 08H, 06, „ , 02H             ; 非压缩BCD 数即十进制数 2,,68
COUNT       EQU $－FIRST
SECOND      DB 05H
THIRD       DB COUNT+1 DUP(?)
data        ends
code        segment
start       proc far
            assume ss:stack, cs:code, ds:data
            push ds
            sub ax, ax
            push ax
            mov ax, data
            mov ds, ax
            MOV CX, COUNT
            MOV SI, 0
            MOV DH, 0                       ; 部分积高位清 0
AGAIN:      MOV AL, FIRST [SI]              ; 取一位被乘数
            MUL SECOND
```

```
                AAM
                ADD AL, DH                          ; 部分积的低位加上一次部分积的高位
                AAA
                MOV DH, AH                          ; 暂存部分积高位
                MOV THIRD [SI], AL                  ; 存入一位积
                INC SI
                LOOP AGAIN
                MOV THIRD [SI], AH                  ; 存入被乘数最高位的部分积高位
                ret
start           endp
code            ends
                end start
```

例 3.15 从键盘上输入两个加数N1 和 N2（1～8 位十进制数），求和并送显示器显示。

该程序分为 3 部分：输入两个加数N1 和N2、相加并将结果以ASCII 码形式存入输出数据区OBUF、输出结果。输入两个加数部分和输出结果部分有3 个数据区：N1、N2 和OBUF。用 BX 做数位较多加数的指针， 用 SI 做数位较少加数的指针，用DI 做存放和数的 ASCII 码的输出数据区 OBUF 的指针。相加并将结果以ASCII 码形式存入输出数据区部分是本程序的主要部分。N1 和 N2 两个加数的数位不一定相等，哪个的数位多也未作规定。因此，应先按较少数位的位数进行两数的相加运算，然后进行数位较多加数的剩余位与 进位的相加，最后再处理两数相加后的进位。例如99567+768，先做 567+768 3 位的相加运算，再做 99 与进位的相加，最后再处理进位。程序框图如图3-7 所示。程序如下。

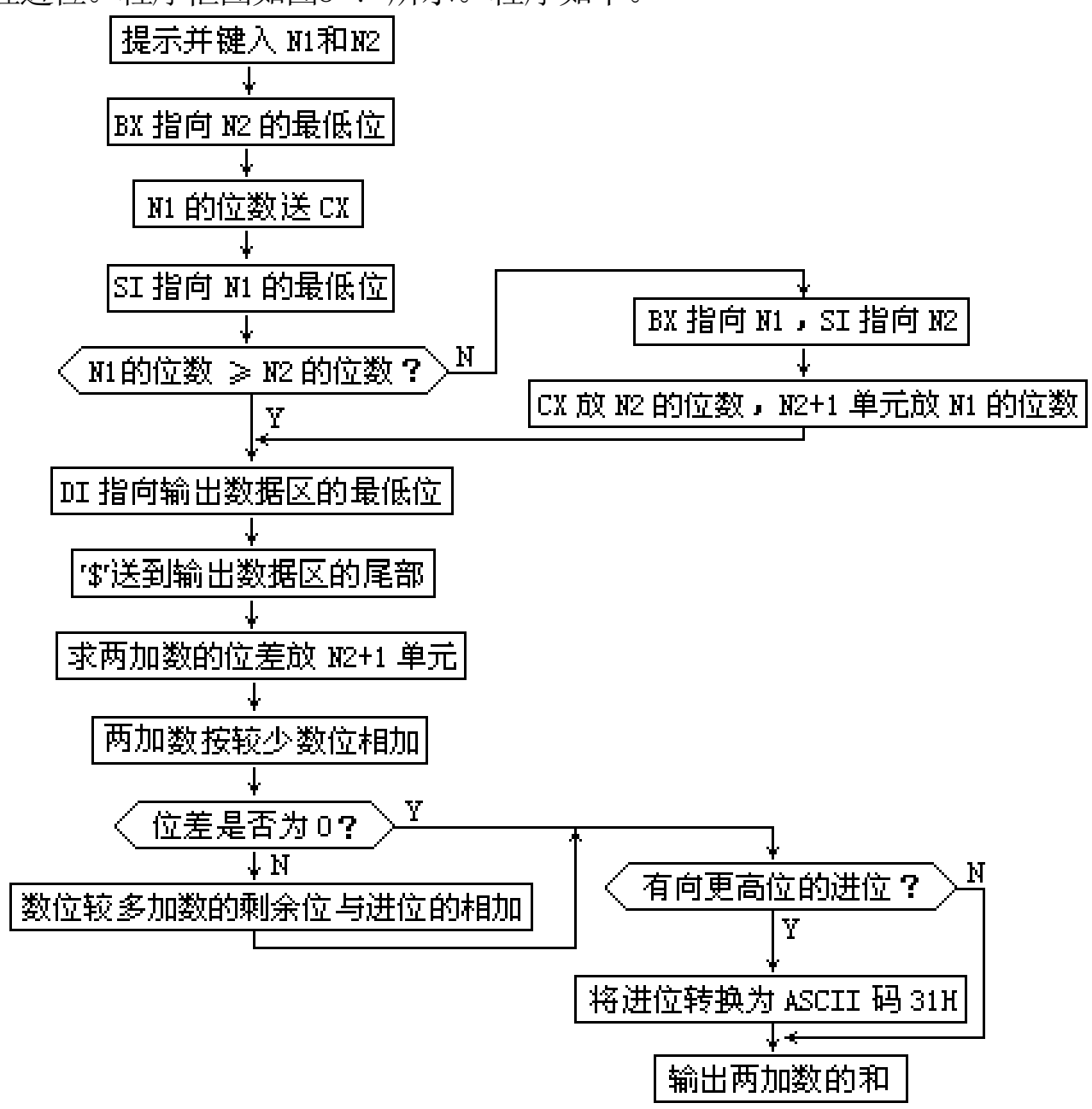


图 3-7　例 3.15 的程序框图

```
stack           segment stack 'stack'
                dw 32 dup(?)
stack           ends
data            segment
OBF1            DB 'PLEASE INPUT  N1: $'
OBF2            DB 'PLEASE INPUT  N2: $'
N1              DB 9, 0, 9 DUP(?)
N2              DB 9, 0, 9 DUP(?)
```

```
OBUF        DB 10 DUP(?)
data        ends
code        segment
start       proc far
            assume ss:stack, cs:code, ds:data
            push ds
            sub ax, ax
            push ax
            mov ax, data
            mov ds, ax
            MOV DX, OFFSET OBF1              ; 提示并键入N1
            MOV AH, 9
            INT 21H
            MOV DX, OFFSET N1
            MOV AH, 10
            INT 21H
            MOV DL, 0AH                      ; 换行
            MOV AH, 2
            INT 21H
            MOV DX, OFFSET OBF2              ; 提示并键入N2
            MOV AH, 9
            INT 21H
            MOV DX, OFFSET N2
            MOV AH, 10
            INT 21H
            MOV DL, 0AH                      ; 换行
            MOV AH, 2
            INT 21H
            MOV BL, N2+1                     ; N2 的位数送BX
            MOV BH, 0
            ADD BX, OFFSET N2+1              ; BX 指向N2 的最低位
            MOV CL, N1+1                     ; N1 的位数送CX 和 SI
            MOV CH, 0
            MOV SI, CX
            ADD SI, OFFSET N1+1              ; SI 指向N1 的最低位
            CMP CL, N2+1                     ; N1 与 N2 的位数相比
            JC  NXCHG
            XCHG CL, N2+1                    ; N1 大，CX 放 N2 的位数，N2+1 单元放N1 的位数
            XCHG BX, SI                      ; BX 指向N1，SI 指向N2
NXCHG:      MOV DI, WORD PTR N2+1            ; DI 指向输出数据区的最低位
            AND DI, 00FFH
            ADD DI, OFFSET OBUF
            MOV BYTE PTR[DI+1], '$'          ; '$'送到输出数据区的尾部
            SUB N2+1, CL                     ; 求两加数的位差放N2+1 单元
            MOV AL, 0                        ; 清AL（进位）
AGAIN:      MOV AH, 0                        ; 清 AH，以便AAA 指令放进位
            ADD AL, [BX]                     ; 将某加数的 1 位与低位的进位相加
            ADD AL, [SI]                     ; 加另一加数的 1 位
            AAA
            ADD AL, 30H                      ; 一位和数转换为ASCII 码
            MOV [DI], AL                     ; 存入输出数据区
            MOV AL, AH                       ; 向高位的进位放AL 中
            DEC BX                           ; 调整指针BX、SI 和DI
            DEC SI
```

```
            DEC DI
            LOOP AGAIN                  ；按较少数位的两数相加完否?
            MOV CL，N2+1                ；两加数的位差送CL
            AND CL，CL                  ；判位差是否为 0
            JZ DONE
AGAIN1：    MOV AH，0                   ；数位较多加数的剩余位与进位的相加
            ADD AL，[BX]
            AAA
            ADD AL，30H
            MOV [DI]，AL
            MOV AL，AH
            DEC BX
            DEC DI
            LOOP AGAIN1
DONE：      AND AL，AL                  ；判是否有向更高位的进位
            JNZ DONE1
            INC DI                      ；无向更高位的进位，调整指针
            JMP DONE2
DONE1：     ADD AL，30H                 ；有则将进位转换为ASCII 码 31H
            MOV [DI]，AL
DONE2：     MOV DX，DI                  ；将输出数据的偏移首地址送DX
            MOV AH，9
            INT 21H
            ret
start       endp
code        ends
            end start
```

例 3.16 在屏幕中部四处分别显示黑桃、红心方块和草花，如图3-8 所示。

图 3-8 屏幕显示的黑桃、红心、方块和草花

将草花、方块、黑桃和红心的ASCII 码、显示属性、显示的行和列按顺序排成数据表，用计数循环调用 BIOS 中的显示器服务程序。

```
stack       segment stack 'stack'
            dw 32 dup(?)
stack       ends
data        segment
CDSH        DB 5，70H，10，40
            DB 4，74H，13，37
            DB 6，70H，13，43
            DB 3，74H，16，40
data        ends
code        segment
begin       proc far
            assume ss：stack，cs：code，ds：data
            push ds
            sub ax，ax
            push ax
            mov ax，data
            mov ds，ax
            MOV AH，0               ；设置 80×25 彩色文本方式
            MOV AL，3
            INT 10H
            MOV AH，15              ；读显示方式
            INT 10H
```

```
            MOV SI, OFFSET CDSH
            MOV CX, 4
AGAIN:      PUSH CX
            MOV AH, 2
            MOV DH, [SI+2]      ; 置光标位置
            MOV DL, [SI+3]
            INT 10H
            MOV AH, 9           ; 写字符和属性
            MOV AL, [SI]
            MOV BL, [SI+1]
            MOV CX, 1
            INT 10H
            ADD SI, 4
            POP CX
            LOOP AGAIN
            ret
  begin     endp
  code      ends
            end begin
```

例 3.17 在屏幕中部画一条红线。

```
  stack     segment stack 'stack'
            dw 32 dup(?)
  stack     ends
  code      segment
  begin     proc far
            assume ss: stack, cs: code
            push ds
            sub ax, ax
            push ax
            MOV AH, 0           ; 设置 320×200 彩色图形方式
            MOV AL, 5
            INT 10H
            MOV AH, 0BH         ; 设置背景色为黑色
            MOV BH, 0
            MOV BL, 0
            INT 10H
            MOV AH, 0BH         ; 设置彩色组 0
            MOV BH, 1
            MOV BL, 0
            INT 10H
            MOV CX, 320         ; 设置 320 个象点计数器
            MOV BP, 0           ; 设置象点列号初值
AGAIN:      PUSH CX
            MOV CX, BP          ;  写象点，行号为 100，列号从 0 至
            MOV AH, 0CH         ; 319 象点为红色(彩色值为 2)
            MOV AL, 2
            MOV DX, 100
            INT 10H
            INC BP              ; 象点列号加 1
            POP CX
            LOOP AGAIN
            ret
begin       endp
code        ends
```

```
                    end   begin
```

2.条件控制的循环程序

例 3.18 将存储器中的 16 位无符号二进制数转换成十进制数，送显示器显示出来。

将二进制数转换为十进制数的方法也有多种。可以采用除 10 取余法将二进制转换为十进制数，每除一次得到一位十进制数，最先得到最低位，最后得到最高位。由于仅由显示器显示，所以得到一位十进制 数后即将它转换为它的 ASCII 码存入输出数据区中。输出数据区的尾部存入‘$’供 9 号功能调用作结束符用。本例虽是 16 位二进制数，但不能用16 位除以 8 位的除法，而要用 32 位除以 16 位的除法，这是因为16 位二进制数除以 10 所得的商仅在AL 中，AL 有可能装不下所得商！如32768/10=3276,,8，即商为3276，余数为 8，AL 装不下商 3276（0CCCH）。32768/10=3276,,8 的二进制运算为： 8000H/0AH=0CCCH,,08H（0CCCH=800H+400H+80H+40H+0CH=2048+1024+128+64+12=3276）。16 位无符号二进制数的最大值为 5 位即十进制数 65535，再加上 9 号功能调用的结束符‘$’，输出数据区 0BUF 最多只需 6 个单元。程序如下：

```
stack segment stack 'stack'
            dw 32 dup(?)
stack ends
data segment
BINARY DW 55H
OBUF DB 6 DUP(?)
data        ends
code        segment
start       proc far
            assume ss:stack, cs:code, ds:data
            push ds
            sub ax, ax
            push ax
            mov ax, data
            mov ds, ax
            MOV BX, OFFSET OBUF+5
            MOV BYTE PTR  [BX], '$'
            MOV AX, BINARY
            MOV CX, 10                  ; 做 32 位除以 16 位的除法，故将 10 送 CX
AGAIN:      MOV DX, 0                   ; 无符号数扩展将 16 位扩展为 32 位
            DIV CX
            ADD DL, 30H                 ; 将DL 中的一位十进制数转换为ASCII 码
            DEC BX                      ; 调整指针
            MOV [BX], DL
            OR AX, AX                   ; 根据商是否为 0，设置ZF
            JNZ AGAIN                   ; 判商是否为 0，不为 0 继续除以 10
            MOV DX, BX                  ; 将输出数据区的偏移首地址送DX
            MOV AH, 9
            INT 21H
            ret
start       endp
code        ends
            end start
```

例 3.19 编制程序，反复从键盘输入字符，并将其送显示器和打印机输出。当键入 ctrl+←(BACK SPACE)时，结束程序运行返回调用程序。

```
      stack       segment stack 'stack'
                  dw 32 dup(?)
      stack       ends
      code        segment
      begin       proc far
                  assume ss:stack, cs:code
                  push ds
                  sub ax, ax
```

```
            push ax
AGAIN:      MOV AH, 0                   ; 接收键入字符
            INT 16H
            MOV AH, 14                  ; 送显示器
            INT 10H
            MOV DX, 0                   ; 送 0 号打印机
            MOV AH, 0
            INT 17H
            CMP AL, 0DH                 ; 键入字符是否回车?
            JNE NEXT                    ; 键入字符不是回车，接收下一个字符
            MOV AL, 0AH                 ; 显示器和打印机换行
            MOV AH, 14
            INT 10H
            MOV AH, 0
            INT 17H
NEXT:       CMP AL, 7FH                 ; 判断键入字符是不是ctrl+←
            JNE AGAIN                   ; 不是，循环执行
            ret
begin       endp
code        ends
            end begin
```

3.双重控制的循环程序

例 3.20 已知字节变量 BUF 存储区中存放着以 0DH(回车的 ASCII 码)结束的十进制数的 ASCII 码。编程检查该字节变量存储区中有无非十进制数，若有显示‘ERROR’；若无则统计十进制数的位数(小于 100)并送显示器显示。

结束本程序有两种情况：存储区中有无非十进制数或者统计工作完毕。程序执行后，显示器显示 ERROR 或者显示统计的十进制数的位数（00～99）。程序如下：

```
stack       segment stack 'stack'
            dw 32 dup(?)
stack       ends
data        segment
BUF         DB '345678,,,,', 0DH
OBUF        DB 3 DUP(?)
ERR         DB 'ERROR$'
data        ends
code        segment
start       proc far
            assume ss:stack, cs:code, ds:data
            push ds
            sub ax, ax
            push ax
            mov ax, data
            mov ds, ax
            MOV AX, 0                   ; 统计十进制数的位数（ASCII BCD 数）
            MOV BX, 0                   ; 存储区的位移量
AGAIN:      CMP BUF [BX], 0DH
            JE DONE
            CMP BUF [BX], '0'
            JB ERROR
            CMP BUF [BX], '9'
            JA ERROR
            INC AL                      ; AAA 不判 CF, 所以可不用ADD 指令
            AAA
            INC BX
```