第五章　循环与分支程序

5.1　循环程序设计

5.2　分支程序设计

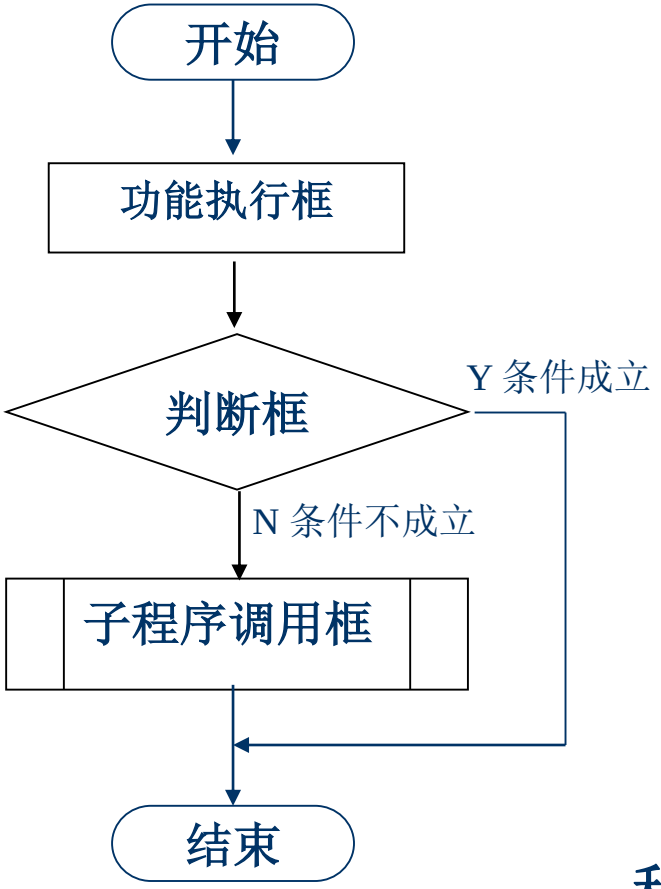
本章目标

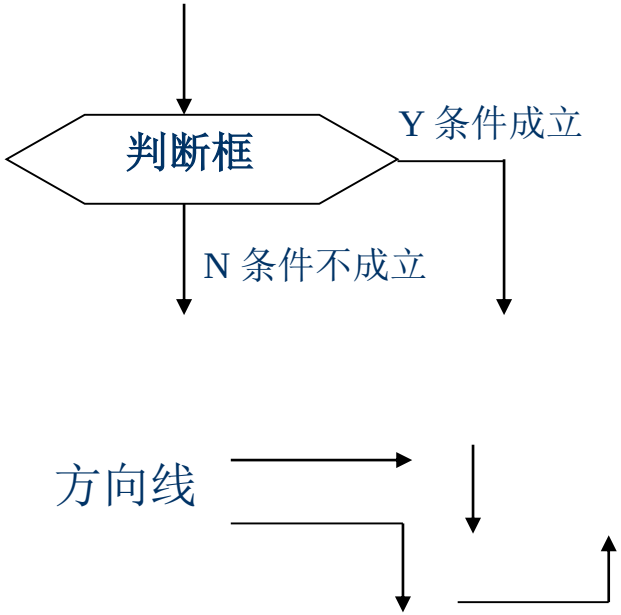
- 掌握汇编语言程序设计的基本步骤
- 熟练掌握顺序、分支和循环程序设计方法
- 掌握汇编语言程序常用的几种退出方法

汇编语言程序设计的基本步骤

1. **分析问题** 根据实际任务（问题）确定任务的数据结构、处理的数学模型或逻辑模型、存储模型；
2. **确定算法** 确定所要解决问题的适当算法(既处理步骤)，如何解决问题，完成任务；
3. **绘制流程图** 设计整个程序处理的逻辑结构，从粗流程到细流程；
4. **存储空间分配** 分配数据段、堆栈段和代码段的存储空间，分配工作单元。借助数据段、堆栈段和代码段定义的伪操作实现；
5. **编写汇编语言源程序** 正确运用80X86CPU提供的指令、伪操作、宏指令以及DOS、BIOS功能调用，同时给出简明的注释；
6. **上机调试** 静态检查后，上机动态调试程序。

程序流程图画法规定





程序结构形式：
顺序、循环、分支和子程序

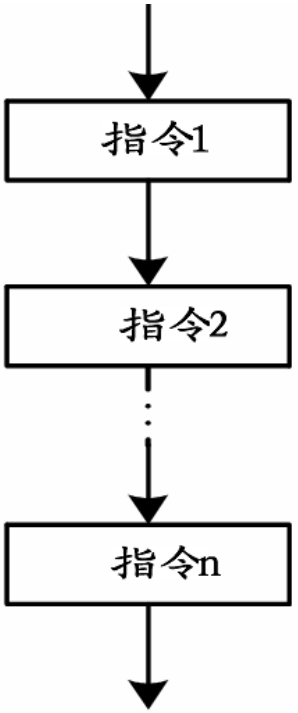
汇编程序的程序结构

- 程序结构形式：
  1. 顺序程序结构
  2. 循环程序结构
  3. 分支程序结构
  4. 子程序结构（第6章）
  5. 中断服务程序（第8章）

1. 顺序程序设计方法

- 程序的执行顺序是从程序的第一条可执行指令开始执行，按照程序编写安排的顺序逐条执行指令，直到最后一条指令为止
- 顺序程序结构所能解决的问题一般属于简单的顺序性处理问题

  如求：Y=（2*X+4*Y）*Z

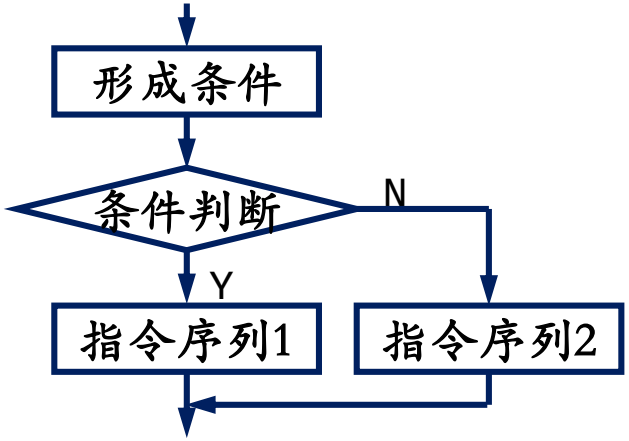


顺序程序结构

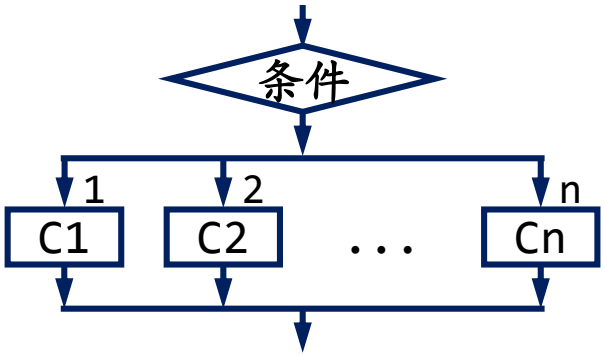
- **程序设计中最基本的结构**

2. 分支程序设计方法

- **根据判断条件转向不同的处理，则要采用分支程序结构。**
- **当执行到条件判断指令时，程序必定存在两个以上分支**
  - **两分支：使用条件转移指令，根据标志位，采用直接寻址方式**
    - **满足条件（条件成立），跳转**
    - **不满足条件（条件不成立），继续**
  - **多分支：使用无条件转移，基于变址寄存器，采用存储器间接寻址方式**
- **程序每次只能执行其中一个分支**



分支程序结构



多分支结构

无条件转移或子程序调用属于哪一类？

例1. 实现符号函数Y的功能。

其中：-128≤X≤+127

$$Y=\begin{cases}1 & \text{当X>0时}\\ 0 & \text{当X=0时}\\ -1 & \text{当X<0时}\end{cases}$$

```
X    DB    ?    ;被测数据
Y    DB    ?    ;函数值单元

     MOV   AL, 0
     CMP   X, AL  ; CMP DS:[X],  AL
     JG    BIG          ; if X>0
     JZ    SAV          ; if X=0
     MOV   AL, 0FFH  ; X<0
     JMP   SHORT  SAV
BIG: MOV   AL, 1    ;大于0
SAV: MOV   Y, AL    ;保存结果
```

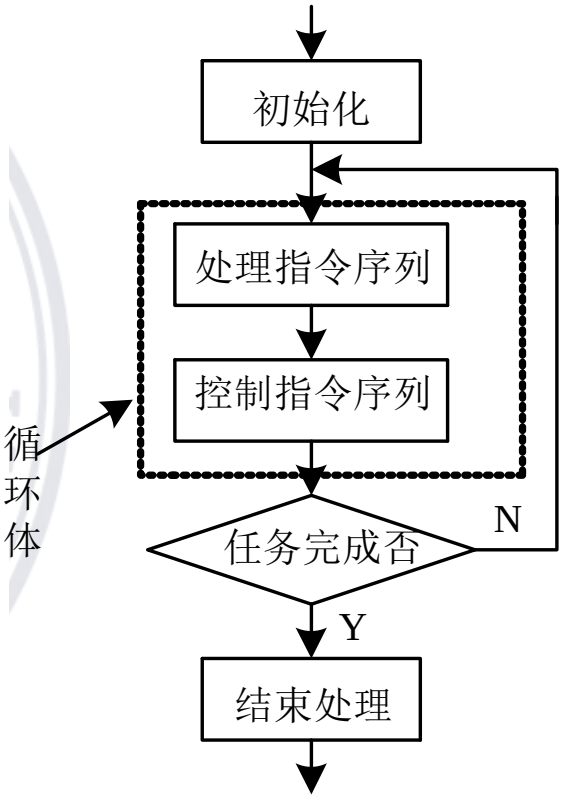
```
DATA    SEGMENT
   X       DB      ?   ;被测数据
   Y       DB      ?   ;函数值单元
DATA    ENDS

CODE    SEGMENT
        ASSUME  CS:CODE,  DS:DATA
START   PROC  FAR

        PUSH    DS
        XOR     AX,   AX
        PUSH    AX

        MOV     AX,   DATA  ;设置段寄存器DS
        MOV     DS,   AX

        MOV     AL, 0
        CMP     X,    AL    ; CMP DS:[X], AL
        JG      BIG
        JZ      SAV
        MOV     AL,   0FFH  ;小于0
        JMP     SHORT  SAV
BIG:    MOV     AL,   1     ;大于0
SAV:    MOV     Y,    AL    ;保存结果

        RET

START   ENDP
CODE    ENDS
        END   START
```

```
DATA    SEGMENT
   X       DB      ?   ;被测数据
   Y       DB      ?   ;函数值单元
DATA    ENDS

CODE    SEGMENT
        ASSUME  CS:CODE,  DS:DATA
START   PROC  FAR

        MOV     AX,   DATA  ;设置段寄存器DS
        MOV     DS,   AX

        MOV     AL, 0
        CMP     X,    AL
        JG      BIG
        JZ      SAV
        MOV     AL,   0FFH  ;小于0
        JMP     SHORT  SAV
BIG:    MOV     AL,   1     ;大于0
SAV:    MOV     Y,    AL    ;保存结果

        MOV     AX，4C00H
        INT     21H

START   ENDP
CODE    ENDS
        END   START
```

3. 循环程序设计方法

- 有一段指令被重复多次执行
  - 被循环多次执行的指令段称为循环体
    - 特例，条件转移指令一般不作为循环体
  - 适应于处理算法相同，每次处理时需要有规律地改变数据或数据地址的问题

- 例如，求内存数据段中存放的N个字节数据（或字数据）的某种运算(加、减、乘、除，移动等等)
  - 循环体是加、减、乘、除，移动等运算指令
  - 设置一个地址指针指向这N个数据的首地址，再设置一个计数器
  - 每次运算之后，修改地址指针使其指向下一个数据，依次循环执行N次



循环程序结构

5.1 循环程序设计

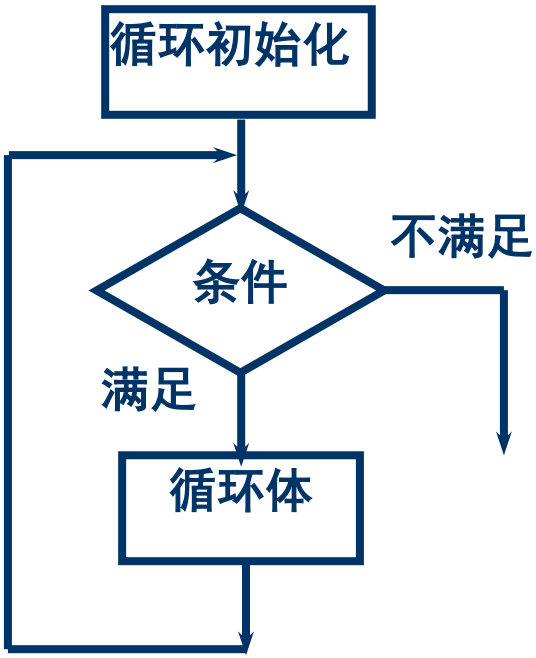
5.1.1 循环程序结构形式

5.1.2 循环程序设计方法

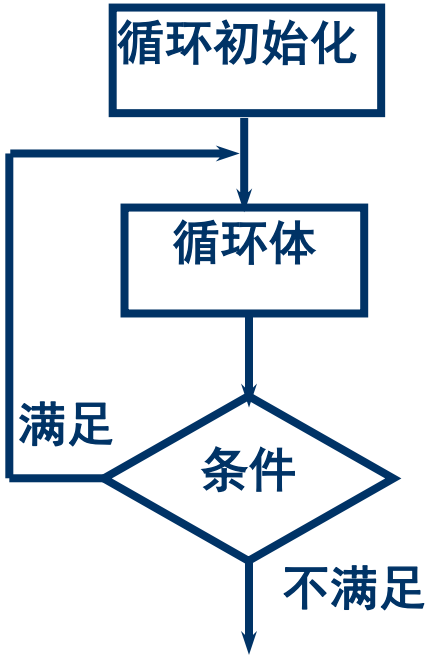
5.1.3 多重循环程序设计

5.1.1 循环程序基本结构

➢ 两种基本结构



DO-WHILE结构



DO-UNTIL结构

■ 三个基本组成部分：

1、初始设置

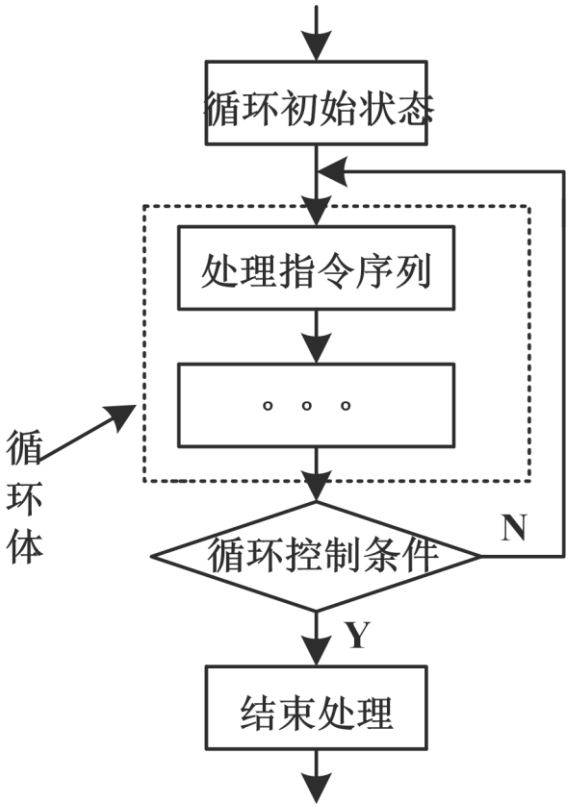
2、循环体

3、循环控制转移

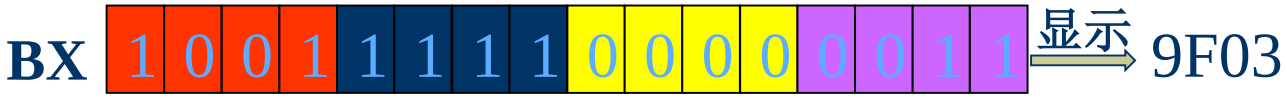
哪种结构性能更好？

5.1.2 循环程序设计方法

- 分析任务，实现三要素：
  1、初始设置
    - 循环次数、数据和变址指针等初始化
  2、循环体
    - 根据任务，选择算法
    - 修改指针等
    - 设置循环控制转移其他条件(标志位)
  3、循环控制转移
    - 正确选择条件转移指令，2要素
      - 循环次数
      - 其他条件，如FLAGS
    - 可在循环体前，也可在循环体后，是程序优化设计的问题，也与CPU的指令预取策略有关。注意会影响初始状态具体设置
- 设计流程框图



例5.1、试编制一个程序把BX寄存器内的二进制数用十六进制数的形式在屏幕上显示出来

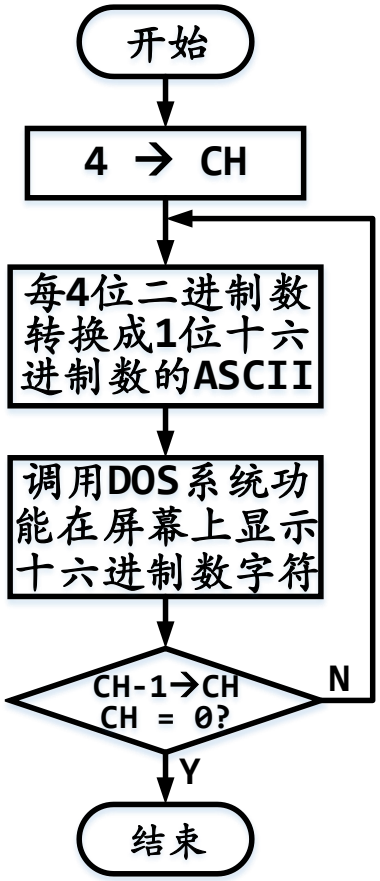


分析问题：把BX寄存器中16位的二进制数用4位十六进制数的形式在屏幕上显示

1、初始设置
- 循环次数：每次显示1个十六进制数字符（送显示器ASCII），循环次数=4，CH=4
- 数据和变址指针初始化等：无

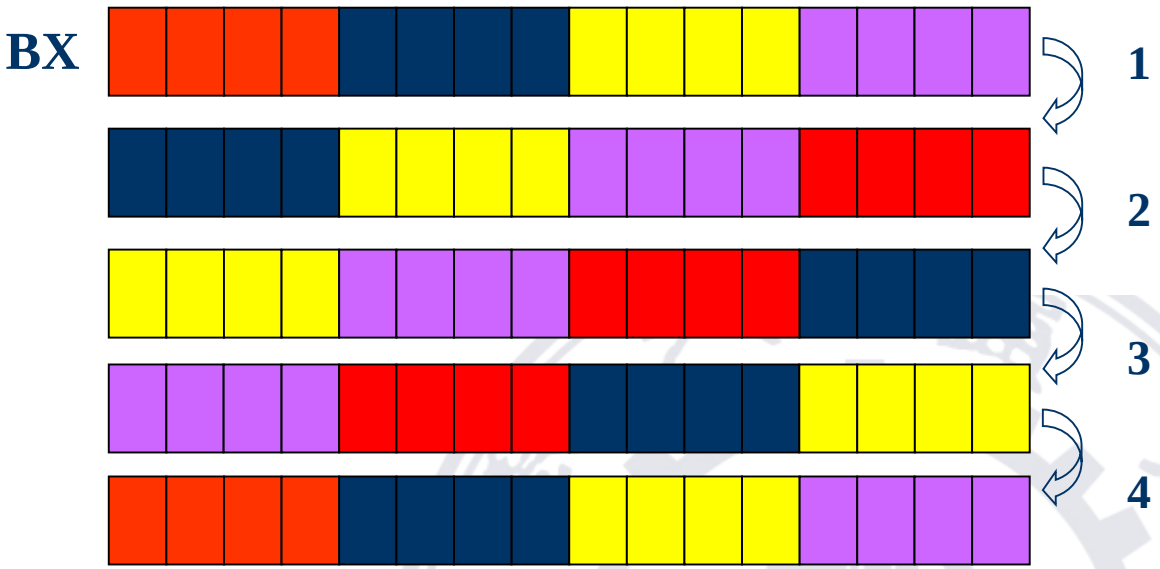
2、循环体
- 根据任务，选择算法
  - 每4位二进制数转换成1位十六进制数的ASCII
  - 调用DOS系统功能在屏幕上显示
- 修改指针：无
- 设置循环控制转移其他条件：无

3、循环控制转移
- 根据计数控制循环次数



设计粗流程图

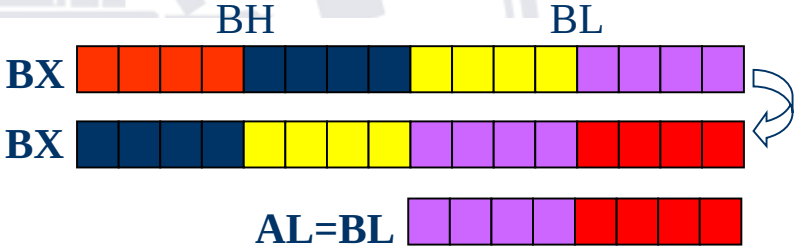
1次显示1位十六进制字符，先显示最高位

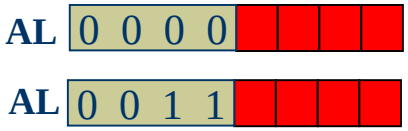


9F03

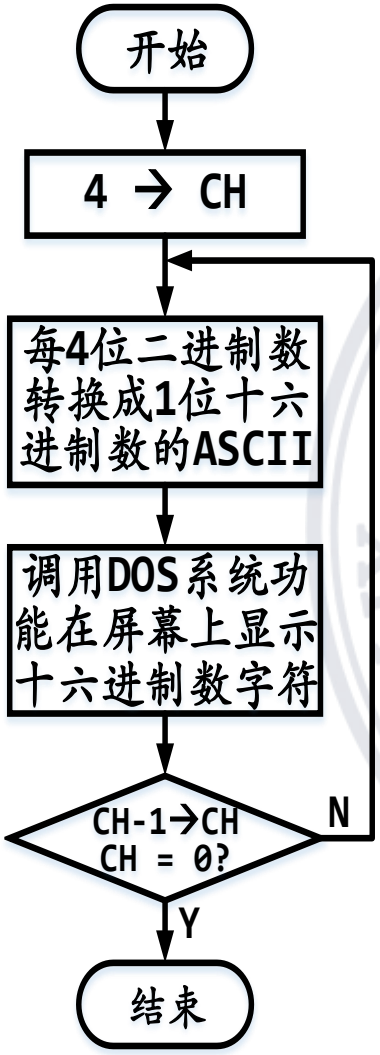
将BX最高4位转换为ASCII码送AL

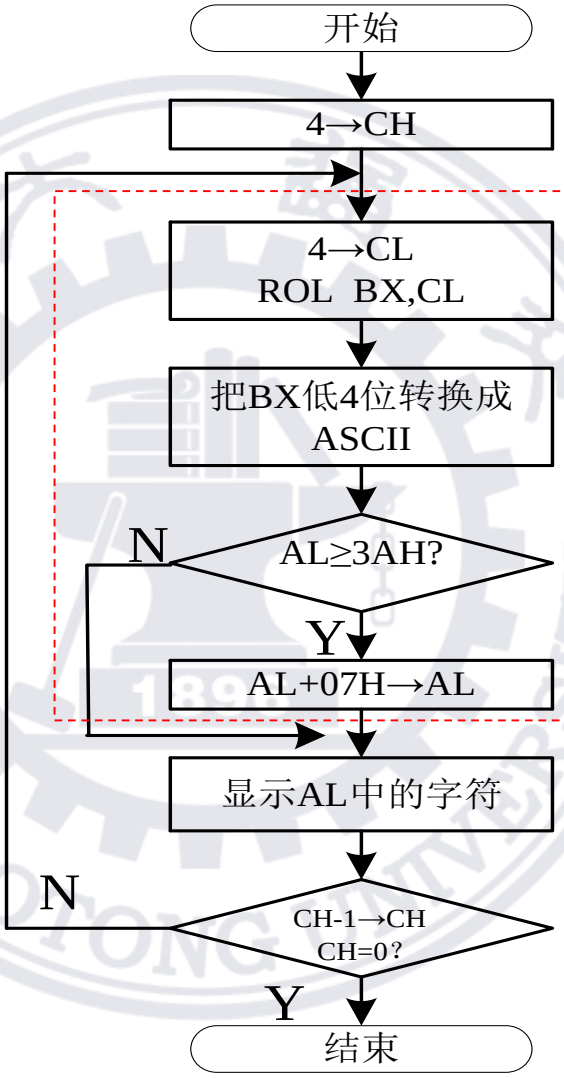
```
rotate:   mov   cl,  4
          rol   bx,  cl
          mov   al,  bl
          and   al,  0fh      ;al的低4位(0-F:0-9、A-F)
          or    al,  30h      ;al=30-39、3A-3F
          cmp   al,  3ah
          jl    printit       ; 是'0'-'9' ASCII 30H-39H，直接送显示器
          add   al,  7h       ;转换为'A'-'F' ASCII 41H-46H
```





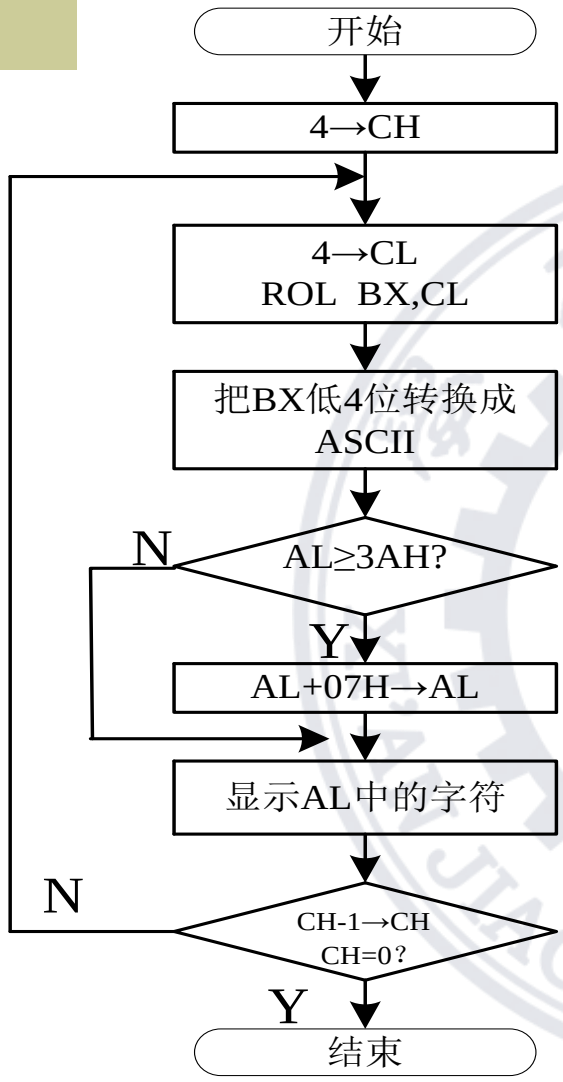
设计详细流程图






```
mov   cl,  4
rol   bx,  cl
mov   al,  bl
and   al,  0fh
or    al,  30h
cmp   al,  3ah
jl    printit
add   al,  7h
```

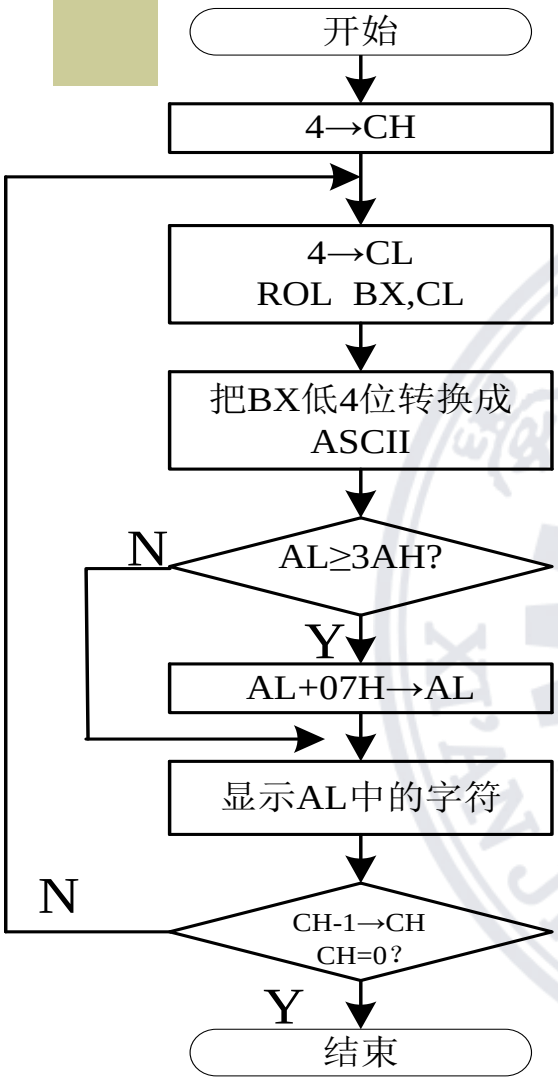
没有数据分配问题，直接编写程序代码段



流程图

```
prognam   segment
          assume cs:prognam
main      proc  far
start:
          push  ds
          sub   ax, ax
          push  ax
初始化    mov   ch, 4
rotate:   mov   cl, 4
          …
printit:
循环体    mov   dl, al
          mov   ah, 2          系统调用
          int   21h            显示1个字符
          dec   ch
          jnz   rotate
循环控制转移
          ret                  DOS环境下返回DOS
main      endp
prognam   ends
          end   start
```

熟记程序框架结构和系统调用约定



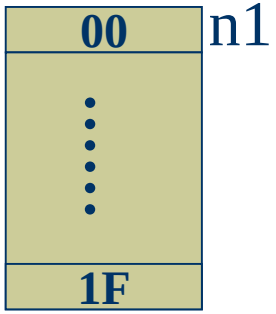

```
prognam   segment
main      proc  far
          Assume cs:prognam
start:    push  ds
          sub   ax, ax
          push  ax

          mov   ch, 4
rotate:   mov   cl,  4
          rol   bx, cl
          mov   al,  bl
          and   al,  0fh ;al低4位(0-F:0-9、A-F)
          or    al,  30h ;al=30-39、3A-3F
          cmp   al,  3ah
          jl    printit  ; '0'-'9' ASCII 30H-39H
          add   al,  7h ;'A'-'F' ASCII 41H-46H
printit:  mov    dl,  al
          mov   ah,  2
          int   21h
          dec   ch
          jnz   rotate

          ret
main      endp
prognam   ends
          end  start
```

mov cl, 4
可放在循环体外优化

例：编制一个数据块移动程序 (已知循环次数)

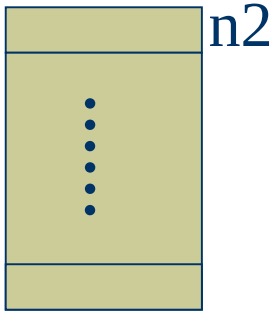
1）任务1：用程序设置数据

给内存数据段（DATA）中偏移地址为n1开始的连续32个字节单元，置入数据00H、01H、02H、……、1FH



2）任务2：移动数据

将内存数据段（DATA）中偏移地址为n1的数据传送到偏移地址为n2开始的连续的内存单元中去



找规律，看有没有重复处理的可能

1）任务1：用程序设置数据

给内存数据段（DATA）中偏移地址为n1开始的连续32个字节单元，置入数据00H，01H，02H，…，1FH

- 多次同样功能的处理，数据处理有规律，采用循环程序结构
- 对有规律（连续）的内存单元操作，地址有规律变化采用变址寻址方式

1、初始设置：
- 变址指针初始化：内存数据段中偏移地址为n1，SI=n1
- 数据初值：数据有规律变化，程序中可以自动生成数据，数据初值=00H
- 循环次数：连续32个字节单元，循环次数=32

2、循环体
- 根据任务，选择算法：一次设置一个字节
- 修改数据，生成新的数据
- 修改指针：变址指针修改
- 设置循环控制转移其他条件：无

3、循环控制转移
- 根据计数控制循环次数

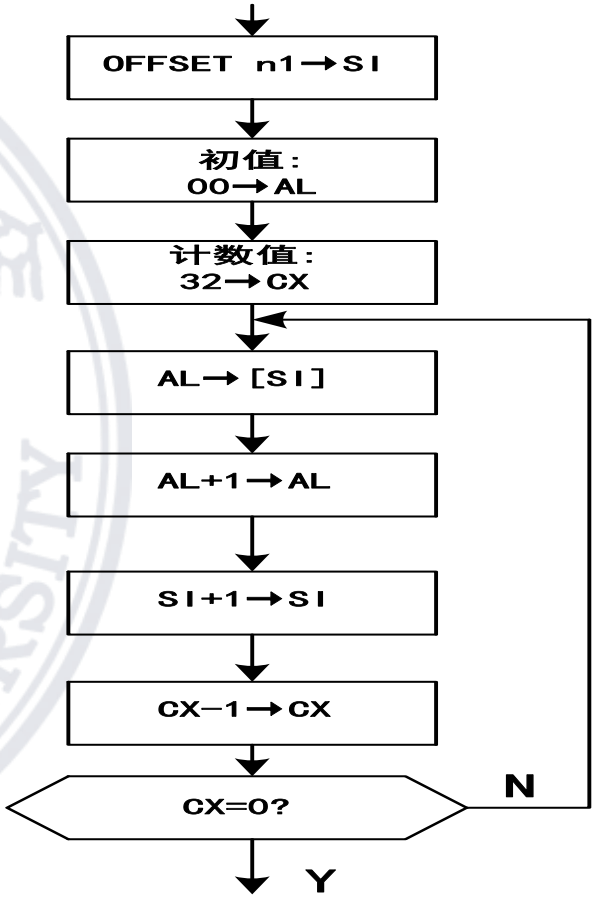
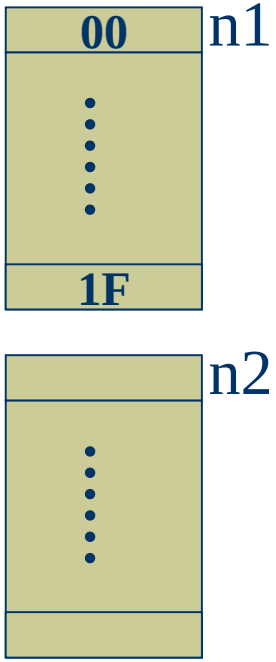


图5.23 置入数据程序流程图

2)任务2：移动数据

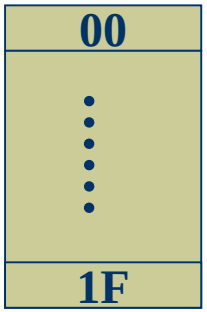
将内存数据段（DATA）中偏移地址为n1的数据传送到偏移地址为n2开始的连续的内存单元中去

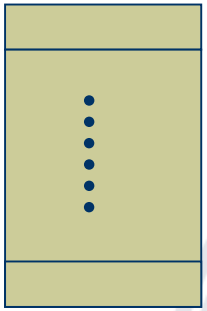
- 程序结构：这个问题是数据块移动问题，可选择循环结构或串传送指令来处理
  - 选择串传送指令MOVSB，或REP MOVSB
- 数据定义：要定义源串和目的串
  - 源串是任务（1）中所设置的数据
  - 目的串要保留相应长度的空间
- 处理方法：MOVSB指令是字节传送指令，它要求事先设置约定寄存器：
  1. 将源串的首偏移地址送SI，段地址为DS
  2. 目的串的首偏移地址送DI，段地址为ES
  3. 串长度送CX寄存器中
  4. 并设置方向标志DF

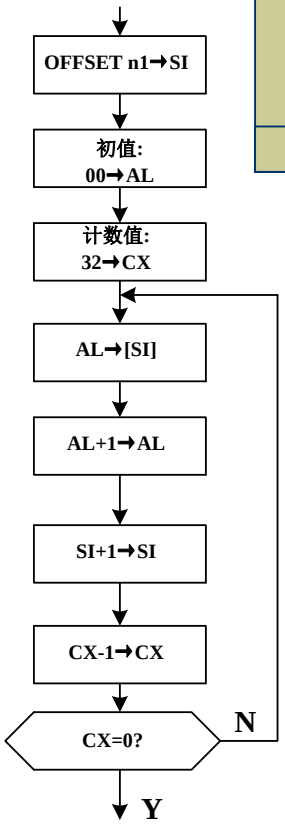


程序源代码请自己编写

隐含、显式循环结构








```
data1     segment
n1        db 32 dup (?)
data1     ends

data2     segment
n2        db 32 dup (?)
data2     ends

prognam   segment
main      proc  far
          assume cs:prognam,
                 ds:data1, es:data2
start:    push  ds
          sub   ax, ax
          push  ax

          mov ax,  data1
          mov ds,  ax
          mov ax,  data2
          mov es,  ax

          mov  si, offset n1
          mov al,  0
          mov cx, 32
rotate:   mov [si],  al
          inc   si
          inc   al
          dec   cx
          jnz  rotate
```

loop rotate

或者

```
data      segment
n1        db 32 dup (?)
n2        db 32 dup (?)
data      ends
assume cs:prognam,
       ds:data, es:data

    mov ax,  data
    mov ds,  ax
    mov es,  ax

          mov si,  offset n1
          mov di,  offset n2
          cld
          mov cx,  32
        rep  movsb es:[di], ds:[si]

          ret

main      endp
prognam   ends
          end  start
```

例子5.2　数“1”的个数

参看p178

方法一：

（1）循环控制条件：计数方式，循环次数一定

（2）数“1”：　移位到进位，测试进位

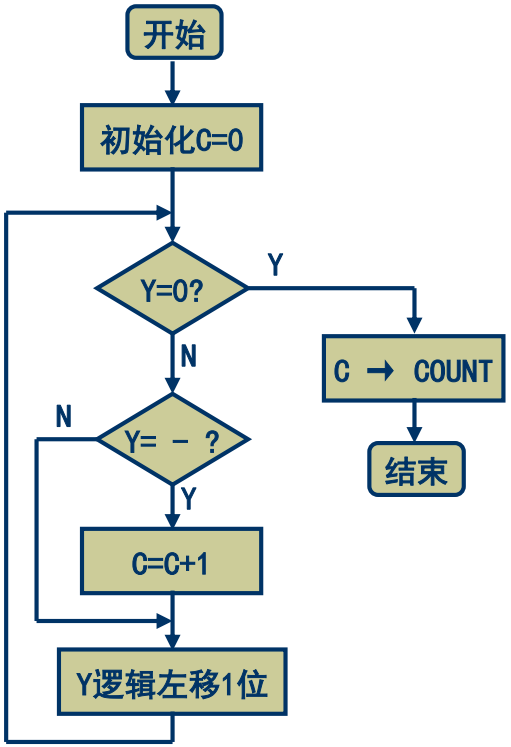
方法二：（优化的方法）

（1）循环控制条件

- 不使用计数方式，而是使用查找完毕，以减少循环次数，提高程序效率
  - 查找完毕，循环次数不定，用条件控制

（2）数“1”的方法

- 判最高位是否为1，测试符号位

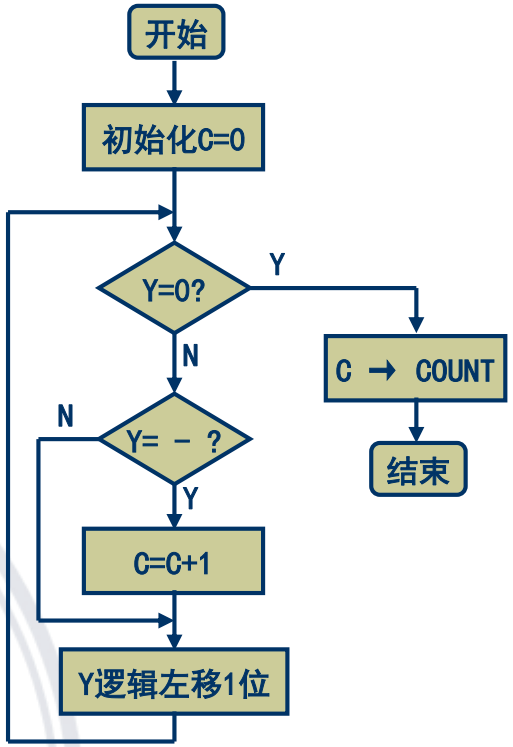


思考：如果要求处理完后源数据值和进位不变，如何处理？

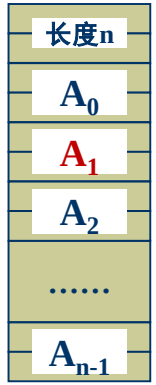
限定条件是不能保存、恢复

```
datarea segment
      addr          dw  number
      number        dw  y
      count         dw  ?
datarea segment

prognam           segment
mian      proc    far
          assume  cs:prognam, ds:datarea
start:    push    ds
          sub     ax, ax
          push    ax

          mov     ax, datarea
          mov     ds, ax

          mov     cx, 0
          mov     bx, addr
          mov     ax, [bx]
repeat:   test    ax, 0ffffh
          jz      exit
          jns     shift   ;最高位是0
          inc     cx      ;最高位是1
shift:    shl     ax, 1
          jmp     repeat

exit:     mov     count, cx
          ret
mian:     endp
prognam   ends
          end     start
```

X X X X X X X X
1 1 1 1 1 1 1 1
X X X X X X X X



例子5.3 删除在未经排序的数组中找到的数（待找的数存放在AX中）



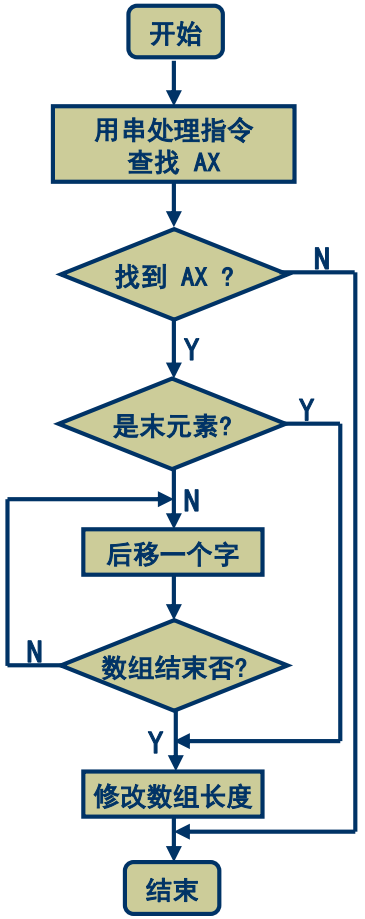
分析题意：

（1）如果没有找到，则不对数组作任何处理

（2）如果找到这一元素，则应把数组中位于高地址中的元素向低地址移动一个字，并修改数组长度值

（3）如果找到的元素正好位于数组末尾，只要修改数组长度值

（4）关键指令是：repnz　scasw

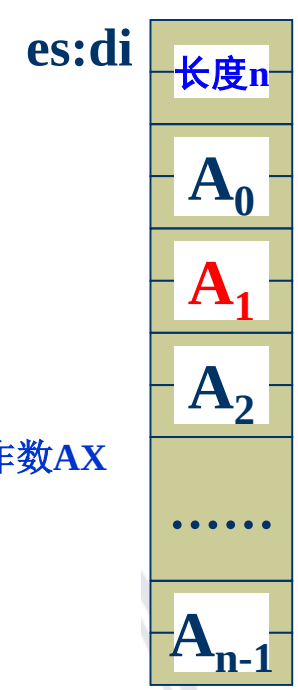


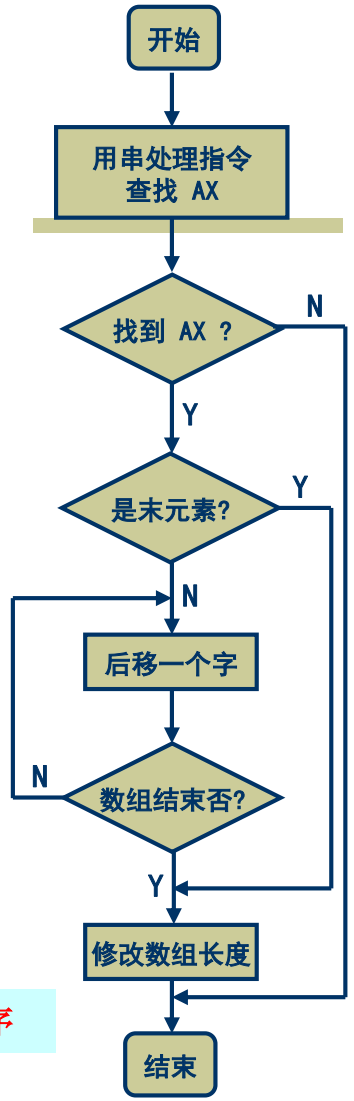
（结合程序p179.asm运行）

ax=0011h　　ax=000bh　　ax=0090h来测试程序

子程序源代码

```
del_ul    proc    near
          cld              ;DF=0
          push    di       ;假设数组的偏移地址在di中
          mov     cx, es:[di]  ;数组长度n送cx
          add     di, 2
          repne   scasw    ;目的操作数es:[di]，源操作数AX
          je      delete   ；找到，跳转
          pop     di       ；没找到
          jmp     short  exit  ;或者ret
delete:   jcxz    dec_cnt  ；  如果CX=0, 在数组末尾，转移
next_el:  mov     bx, es:[di]
          mov     es:[di-2], bx
          add     di, 2
          loop    next_el
dec_cnt:  pop     di
          dec     word ptr es:[di]
exit:     ret
del_ul    endp
```





执行了几次pop操作?

注意repne scasw的具体操作顺序

SCAS指令执行的操作：

1. DST-AL/AX/EAX，但结果不保存，根据结果设置标志位
2. 目标操作数的地址指针（变址寄存器DI）的修改

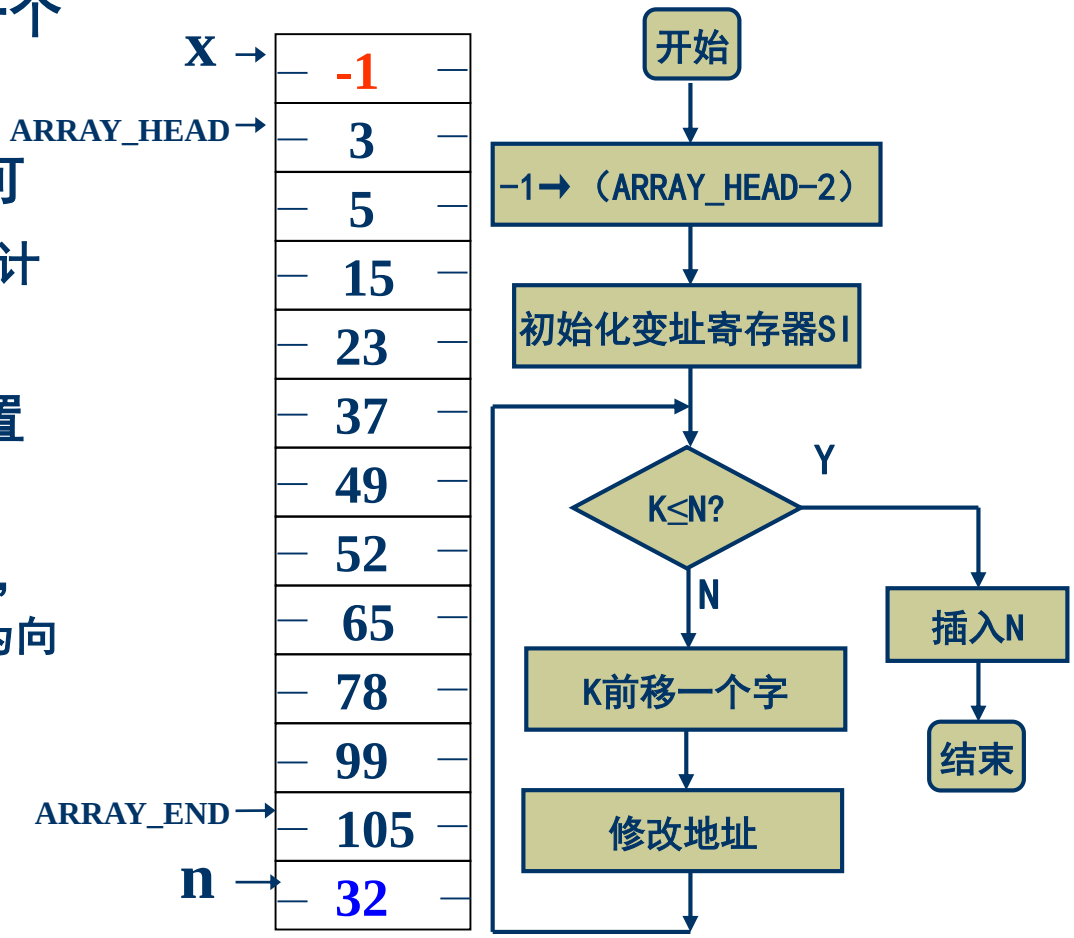
REPNE执行的操作：

1. 修改计数器 CX－1→CX, 执行后跟的串操作指令。
2. 若 CX＝0（计数到)或ZF＝1(相等), 则结束重复
3. 否则， 转1，继续重复上述操作

例子5.4　在已整序的正数字数组中插入正数N

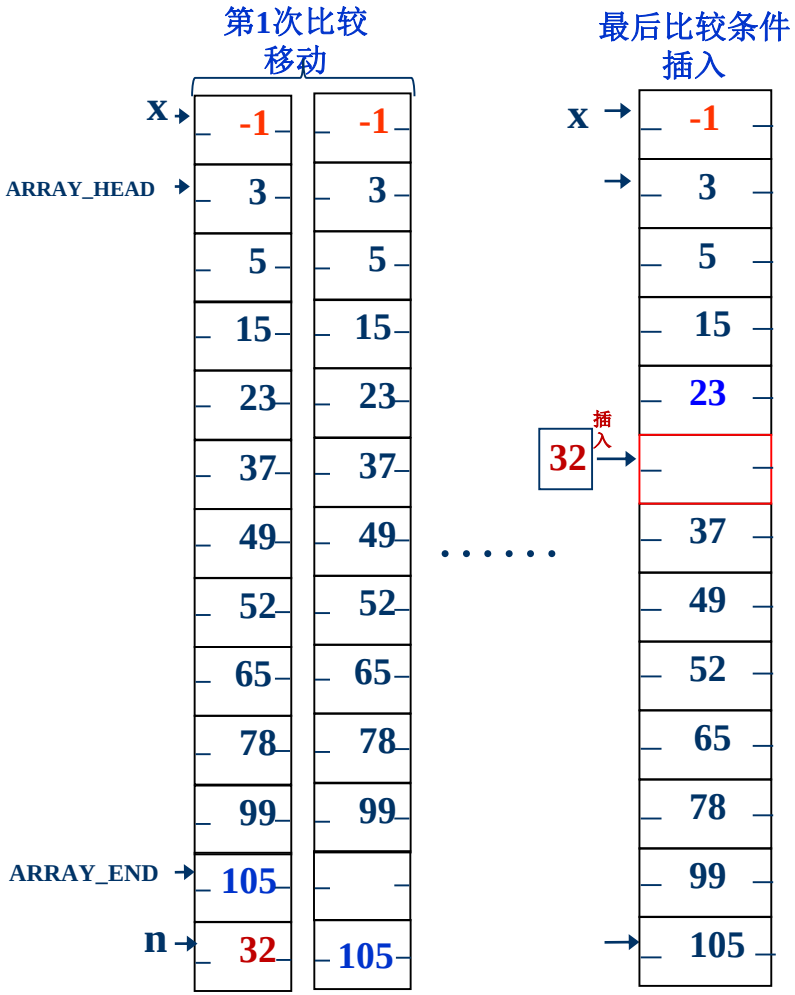
找到位置，将数据向高地址移一个字，插入N, 结束

- 循环控制条件：找到位置即可
  - 位置一定能找到，因此无须计数次数等
- 循环结构的主要任务是找位置和移字数据
  - 将高于N的数向高地址移一个字，边找边移，因此循环体内处理为向高地址移一个字
- 插入N在循环结构外



```
x             dw    ?
array_head    dw    3,5,15,23,37,49,52,65,78,99
array_end     dw    105
n             dw    32      ;需要插入的正数
```

```
        mov   ax, n     ;ax=32
        mov   array_head-2, 0ffffh
        mov   si, 0
compare:
        cmp    ax,  array_end[si]
        jge    insert
        mov   bx, array_end[si]
        mov   array_end[si+2], bx
        sub    si, 2
        jmp    short compare
insert:
        mov   array_end[si+2], ax
```



例子5.5

- 设数组X、Y中分别存有10个字型数据

  试实现以下计算并把结果存入数组Z单元

  Z0＝X0+Y0　　Z1＝X1+Y1

  Z2＝X2-Y2　　Z3＝X3-Y3

  Z4＝X4-Y4　　Z5＝X5+Y5

  Z6＝X6-Y6　　Z7＝X7-Y7

  Z8＝X8+Y8　　Z9＝X9+Y9

有规律处理，不定运算 → 循环+分支

建立逻辑尺：

逻辑尺方法

看左边的流程图

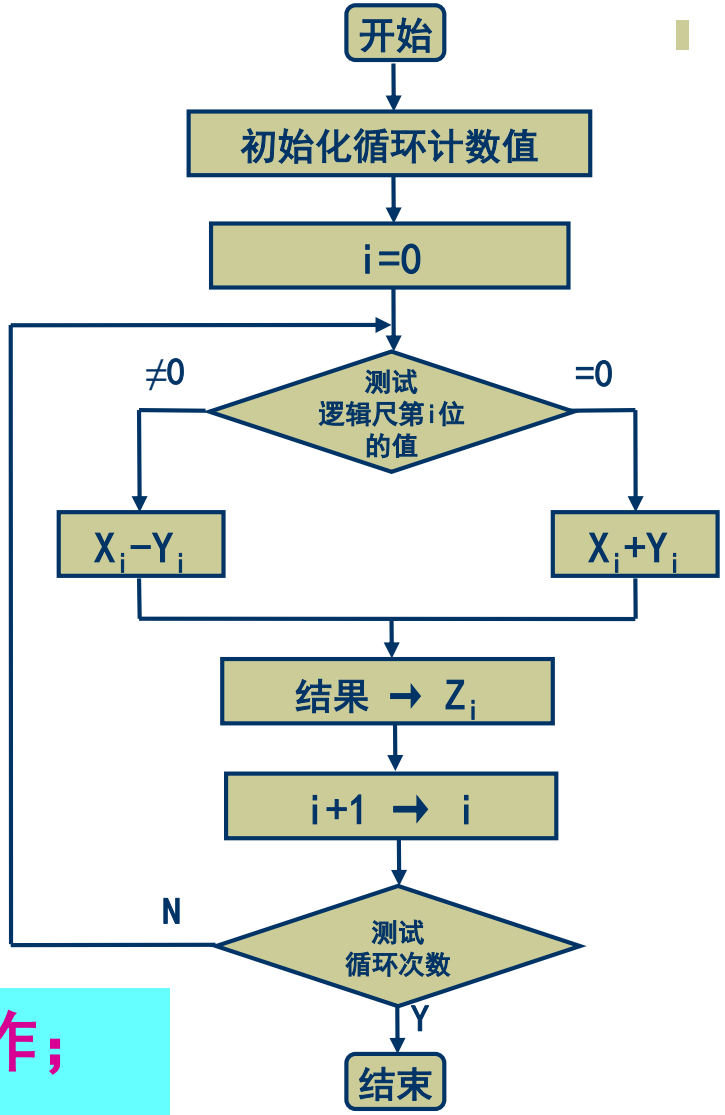
（1）设立标志位（建立逻辑尺），每一位0（加法）、1（减法）可代表两种操作

0000000011011100

$Z_2=X_2-Y_2$　　$Z_0=X_0+Y_0$

（2）进入循环后判断标志位来确定该做的工作

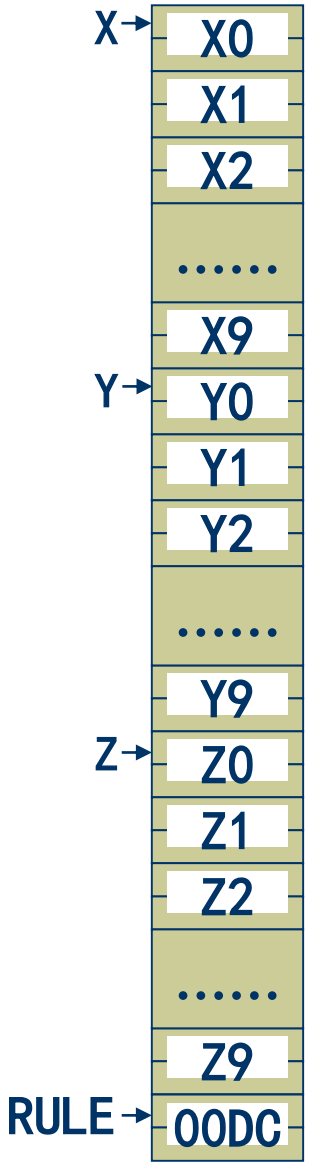
- 适合对有规律的数组进行无规律操作等



逻辑尺：一个二进制数，可表示两种操作；
两位可表示4种操作，…

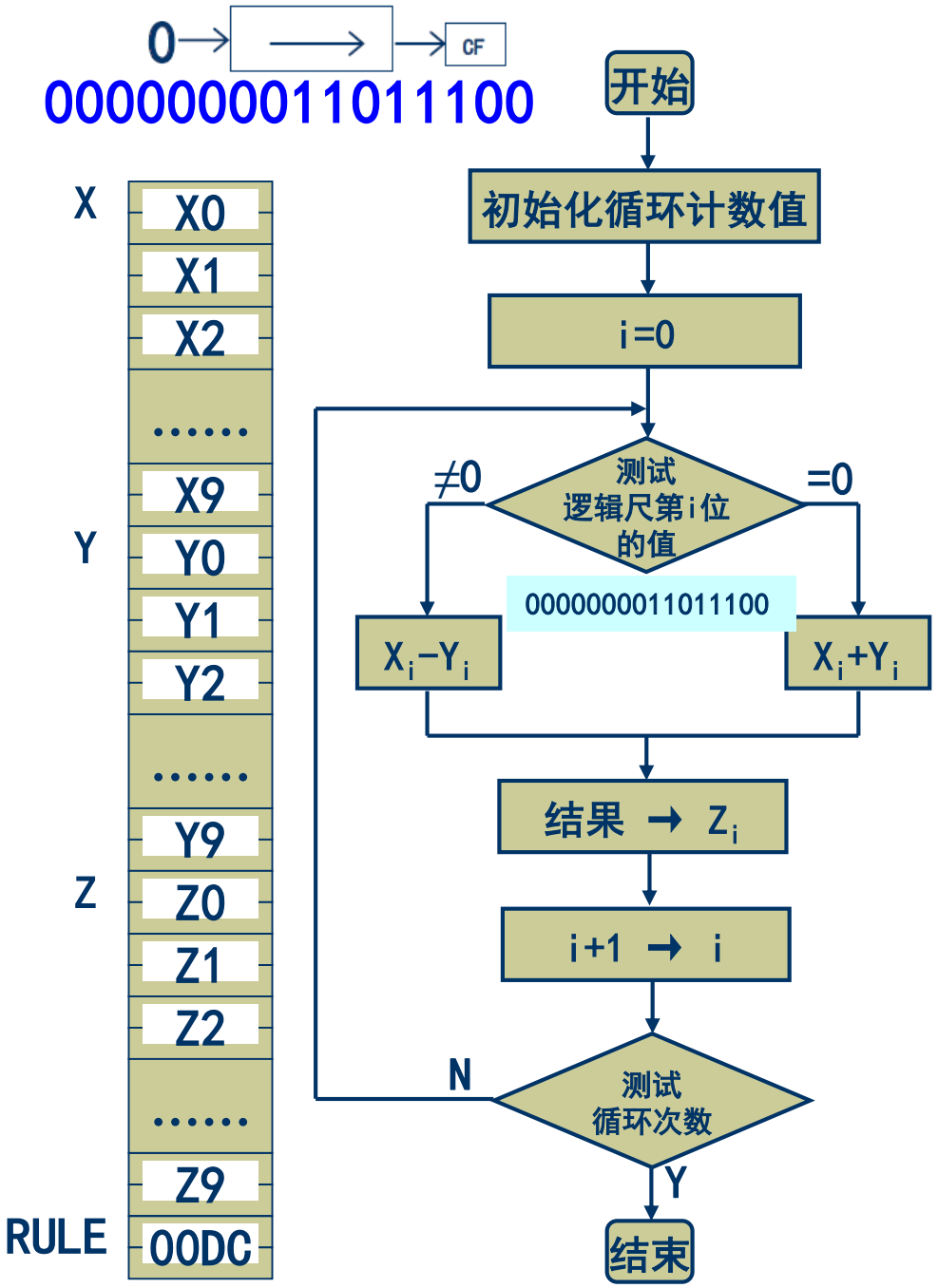
```
DATA    SEGMENT
   X    DW  X0,X1,X2,X3,X4
        DW  X5,X6,X7,X8,X9
   Y    DW  Y0,Y1,Y2,Y3,Y4
        DW  Y5,Y6,Y7,Y8,Y9
   Z    DW  10 DUP(?)
   RULE DW  0000000011011100B;逻辑尺
DATA    ENDS
```

Z0＝X0+Y0　　Z1＝X1+Y1
Z2＝X2-Y2　　Z3＝X3-Y3
Z4＝X4-Y4　　Z5＝X5+Y5
Z6＝X6-Y6　　Z7＝X7-Y7
Z8＝X8+Y8　　Z9＝X9+Y9



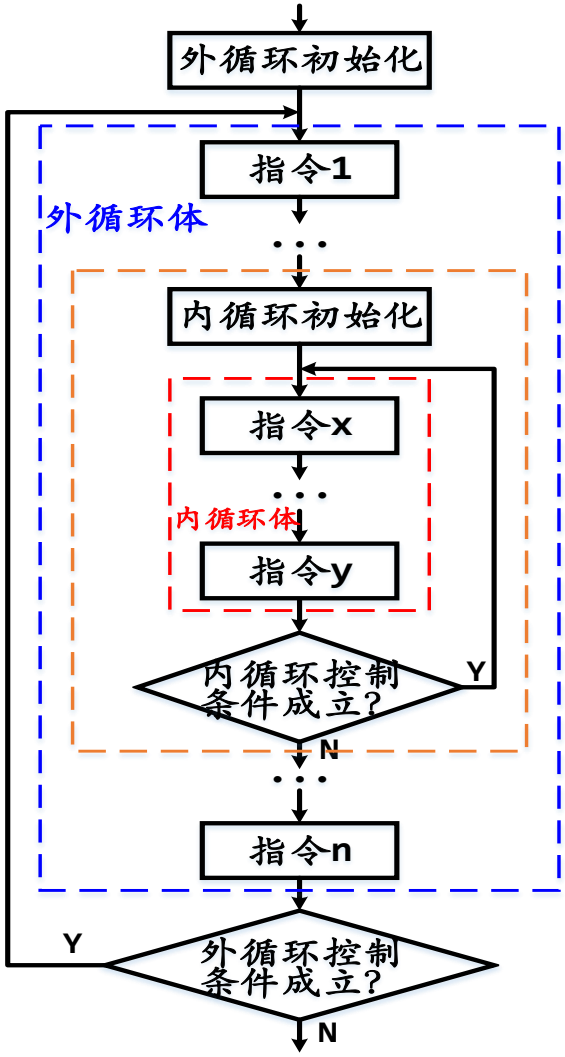
```
CODE   SEGMENT
       ASSUME CS:CODE,DS:DATA
MAIN   PROC   FAR
       MOV    AX,@DATA     ;MOV AX, DATA
       MOV    DS,AX
       MOV    CX,10  ;循环次数
       MOV    DX,RULE   ;逻辑尺
       MOV    BX,0   ;地址指针
NEXT:  MOV    AX,X[BX]   ;取X中的一个数
       SHR    DX,1       ;逻辑尺右移一位
       JC     SUBS       ;CF=1，转移，做减法
       ADD    AX,Y[BX]  ;CF=0，两数加
       JMP    SHORT RESULT
SUBS:  SUB    AX,Y[BX]   ;两数减
RESULT:MOV    Z[BX],AX   ;存结果
       ADD    BX,2   ;修改地址指针
       LOOP   NEXT
       MOV    AX,4C00H
       INT    21H
MAIN   ENDP
CODE   ENDS
       END    MAIN
```

返回DOS



5.1.3 多重循环程序设计

- 多重循环程序结构
  - 循环中有循环
  - 内外层循环都应该具有完整的循环程序结构
  - 注意内循环初始化



例5.7： 有一个首地址为ARRAY的N个字的数组，编制程序使该数组中的数按照从大到小的顺序排序参看p187.asm

- 算法：小数沉底法/起泡排序法
  - 结果：数据由大到小排列，或由小到大排列
- 举例：要求结果是 数据由大到小排列

第1遍：两相邻数据比较，交换，数据按大到小排列，比较N-1次，最后一个是最小数

第2遍：两相邻数据比较，交换，数据按大到小排列，比较N-2次，最后2个数由大到小

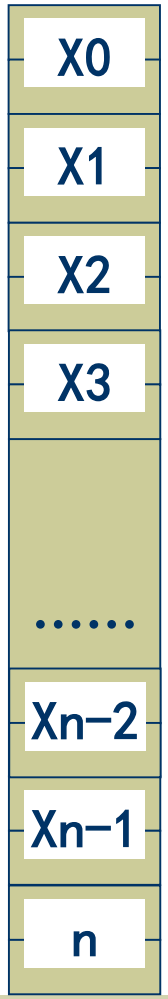
……

第N-1遍：两相邻数据比较，交换，数据按大到小排列，比较1次，得到结果

每遍比较都会在底下得到最小数

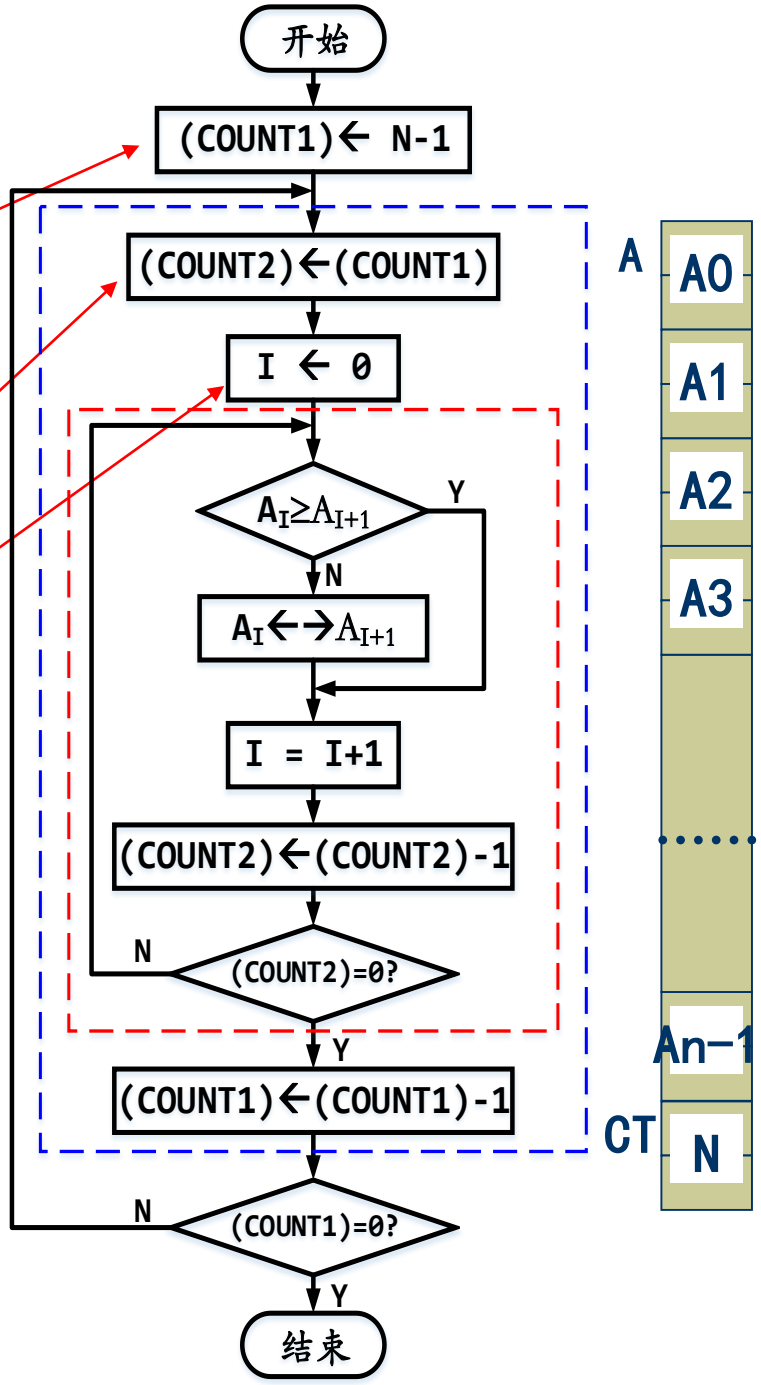
- 总共N-1遍：作为外循环计数，初值=N-1
- 每遍的比较次数：作为内循环计数，初值=外循环计数当前值

沉底由上向下比较，气泡由下向上比较



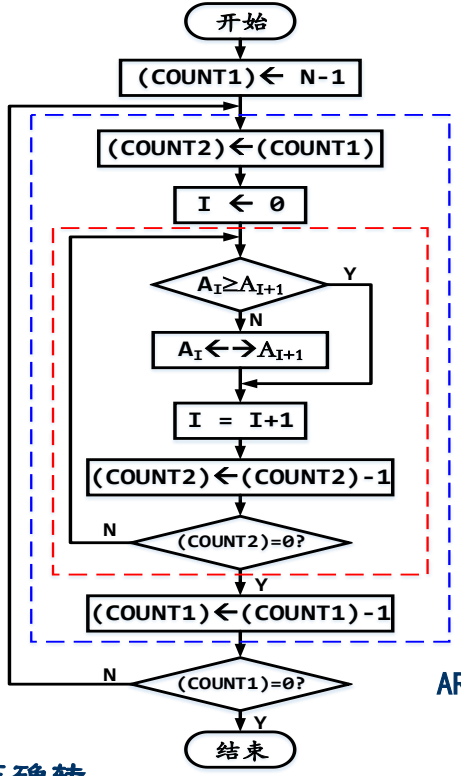
- 总共N-1遍：作为外循环计数，初值=N-1
- 每遍的比较次数：作为内循环计数，初值=外循环计数当前值
- 每次从最低地址单元开始

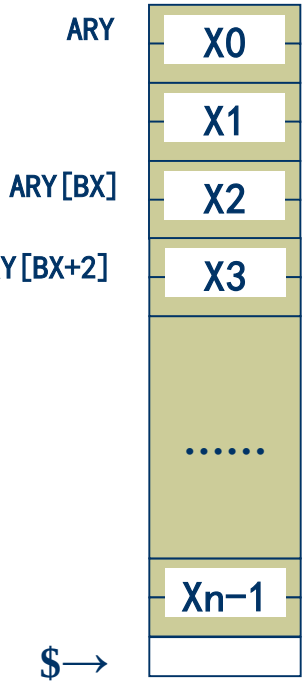
- 必须N-1遍，但有时实际中不需N-1遍就已经得到结果，程序效率不高



```
DATA    SEGMENT
ARY     DW    n DUP(?)
CT      EQU   ($-ARY)/2          ;元素个数
DATA    ENDS

CODE    SEGMENT
        ASSUME CS:CODE,DS:DATA,SS:STACKSG
MAIN    PROC  FAR
        MOV   AX,DATA
        MOV   DS,AX
;
        MOV   DI,CT-1            ;初始化外循环次数
LOP1:   MOV   CX,DI              ;置内循环次数
        MOV   BX,0               ;置地址指针
LOP2:   MOV   AX,ARY[BX]
        CMP   AX,ARY[BX+2]       ;两数比较
        JGE   CONT                        ;Xi≥Xi+1,次序正确转
        XCHG  AX,ARY[BX+2]                ;次序不正确互换位置
        MOV   ARY[BX],AX
CONT:   ADD   BX,2               ;修改地址指针
        LOOP  LOP2               ;内循环控制
        DEC   DI                 ;修改外循环次数
        JNZ   LOP1               ;外循环控制
;
        MOV   AX,4C00H
        INT   21H
MAIN    ENDP
CODE    ENDS
        END   MAIN
```

CT=n





$是汇编程序
地址计数器值

CT EQU ($-ARY)/2 什么意思？
只是定义了一个常数，不分配存储单元

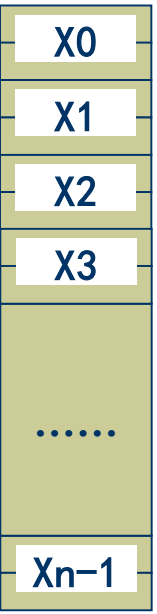
CT db ($-ARY)/2 什么意思？
分配一个字节存储单元，并赋初值，变量名为CT

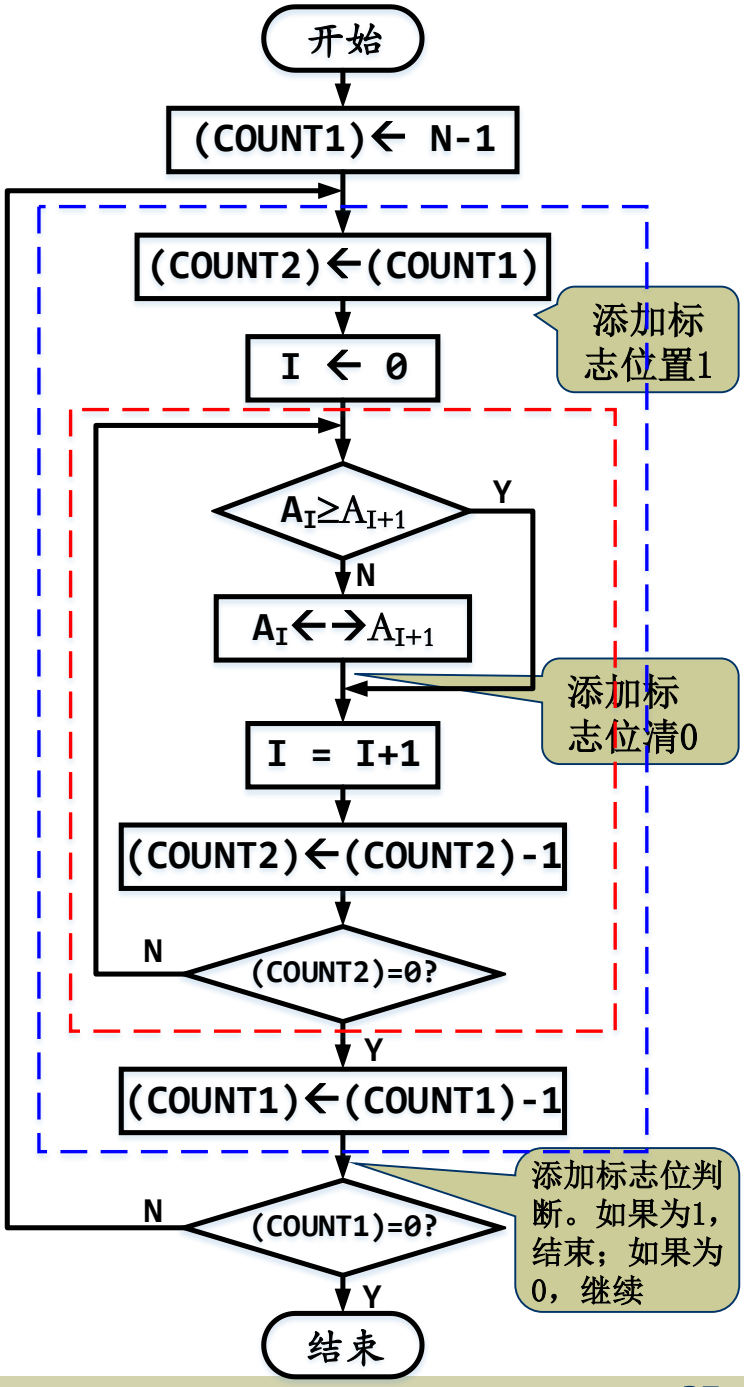
MOV DI,CT-1
执行结果？

多重循环时，最内循环最好使用计数寄存器CX

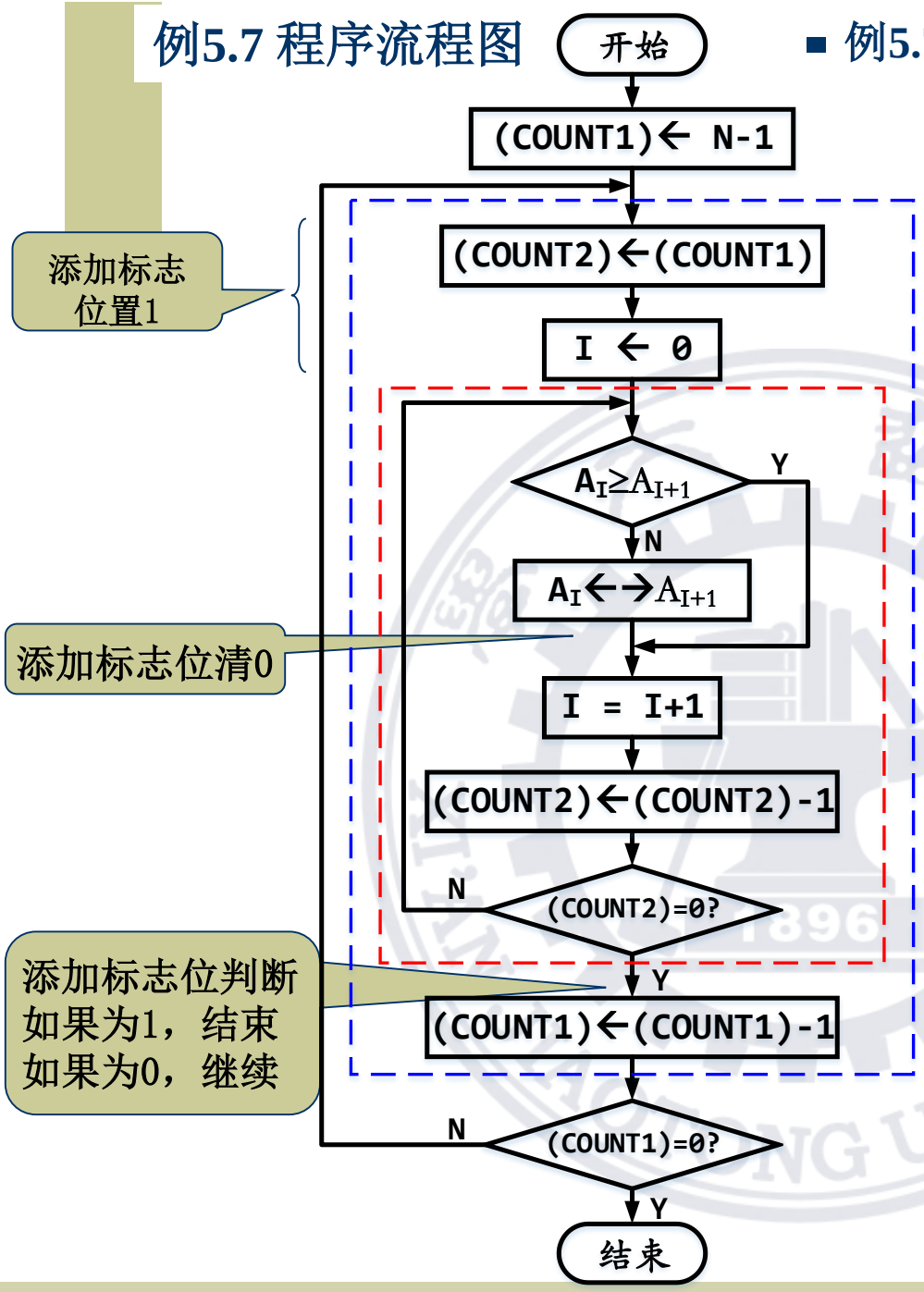
如何提高算法性能？

- 对例5.7中算法进行优化，提高程序效率(参考例5.8)
  - 能否提前结束？
    - 如果某遍没有交换操作，说明排序已经符合要求，可以提前结束
    - 设置有无交换操作标志位
      - 每次进入内循环之前将交换操作标志位初始化为1，如果内循环中有数据交换则将标志位清0
    - 内循环的循环次数同例5.7
  - 外循环的结束条件：循环次数计数=0 或者 交换操作标志位=1
  - 参看p189框图和p190.asm

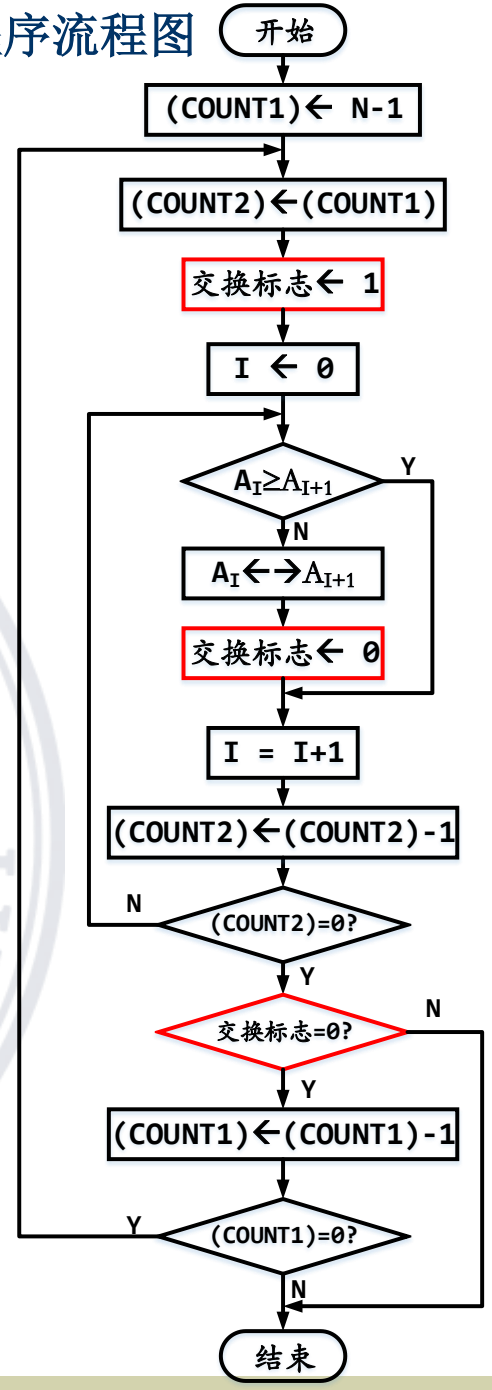




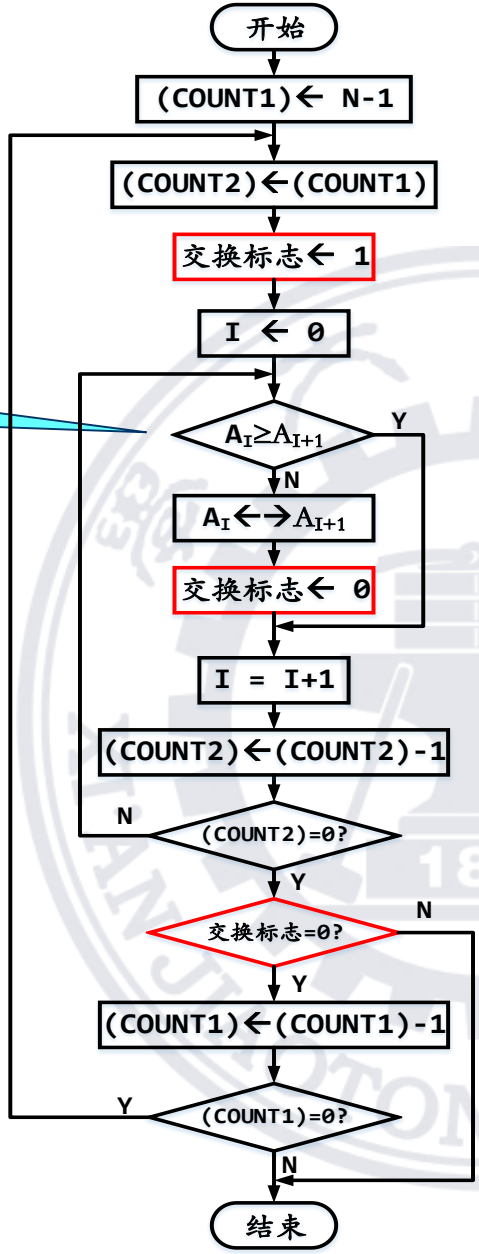
例5.7 程序流程图



例5.7 优化后的程序流程图

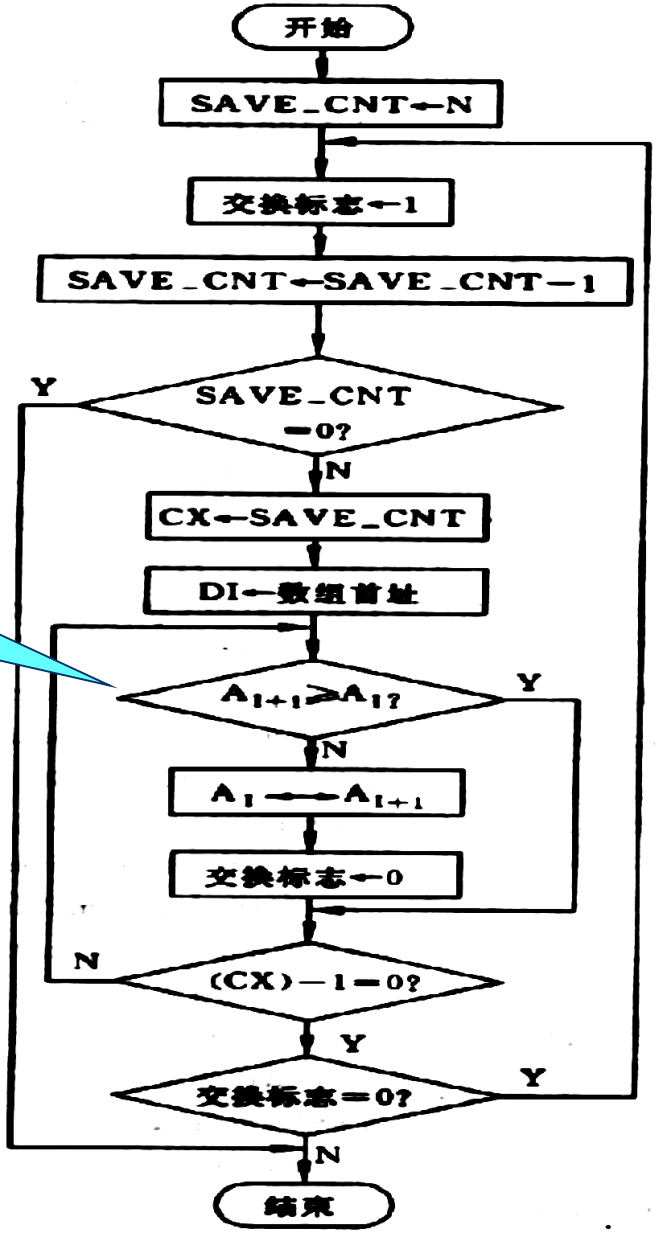


例5.7优化程序流程图



数由大到小排列

例5.8的程序流程图



数由小到大排列

例. 编制用软件延时200ms的程序

- 设用以下程序片段可以延时10ms：

```
DELAY10:   MOV   CX,2801   ;置循环次数，根据机器主频定
WT:        LOOP  WT
```

如果主频高，延迟时间太短，可加空操作

```
WT:  NOP
     LOOP WT
```

- 则延时200ms只需把以上程序片段循环20次即可

```
DELAY          PROC   NEAR
               MOV    BL,20          ;置外循环次数
DELAY10:       MOV    CX,2801        ;置内循环次数
WT:            LOOP   WT
               DEC    BL             ;修改外循环次数
               JNZ    DELAY10        ;外循环控制
               RET
DELAY          ENDP
```

5.2 分支程序设计

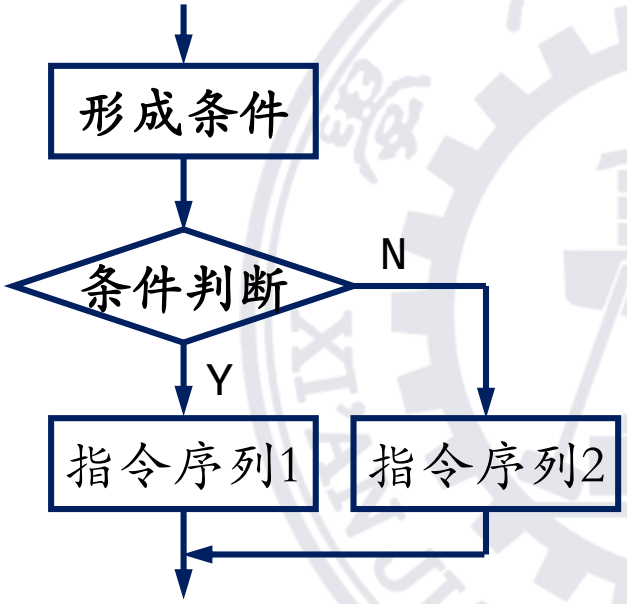
5.2.1 分支程序的结构形式

5.2.2 分支程序的设计方法

5.2.3 跳跃表法

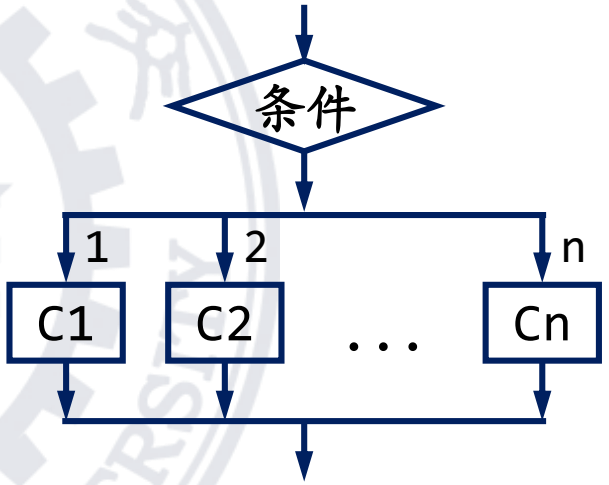
5.2.1 分支程序的结构形式

程序有两条以上执行路径，但每次只能执行一个指令序列



分支程序结构

实现：条件转移指令JZ等



多分支结构

实现：①基于逻辑尺的条件转移指令
②基于跳转表的间接寻址转移指令

5.2.2 分支程序的设计方法

- **两分支程序：**基本的分支程序，程序分支一般用条件转移指令来产生
- **多分支程序：**
  - 利用转移指令不影响条件码的特性，连续使用条件转移指令（宏观上属于多分支程序，微观上属于两分支程序）
  - 基于逻辑尺的条件转移指令（宏观上属于多分支程序，微观上属于两分支程序）
  - 跳跃表法的无条件间接寻址转移指令

```
    jl  C
    je  B
A: …
B: …
C: …
```

$$Y=\begin{cases}1 & \text{当X>0时}\\ 0 & \text{当X=0时}\\ -1 & \text{当X<0时}\end{cases}$$

```
      CMP   X,AL
      JG    BIG
      JZ    SAV
      MOV   AL,0FFH
      JMP   SHORT SAV
BIG:  MOV   AL,1
SAV:  MOV   Y,AL
```

```
    cmp ax, 00
    jz  D
    cmp ax, 01
    jz  C
    cmp ax, 02
    jz  B
A: …
B: …
C: …
D: ….
```

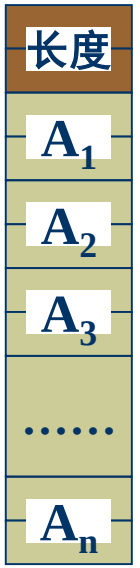
存储器单元

目标地址1
目标地址2
……
目标地址n

```
MOV  SI, AX
SAL  SI
JMP  WORD  PTR  [BX][SI]
```

例子5.9：有一个从小到大顺序排列的无符号数的数组。

- DI=数组首地址，数组中第一个单元存放数组长度，AX中有一个无符号数。
- 要求：在数组中查找与AX相等的数
  - 如找到，则使标志位CF=0，并在SI中给出该元素在数组中的相对位置；
  - 如未找到，则使CF=1



- 查找时：
  - 如果数据大小排序不定，只能用顺序查找方法；
  - 如果数据大小排序规整，也可用折半查找法，以提高查找效率

折半查找算法

在一个长度为n的有序数组r中，查找元素k的折半查找算法如下：

(1)初始化被查找数组的首尾下标：1→low，n→high;

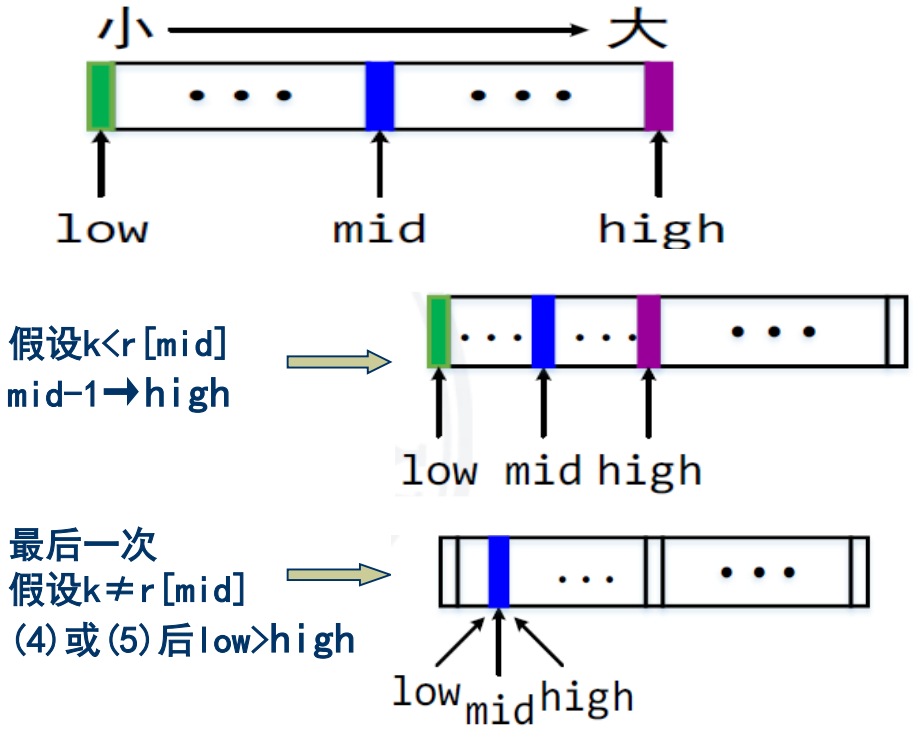
(2)若low>high，则查找失败，置CF=1，退出程序。

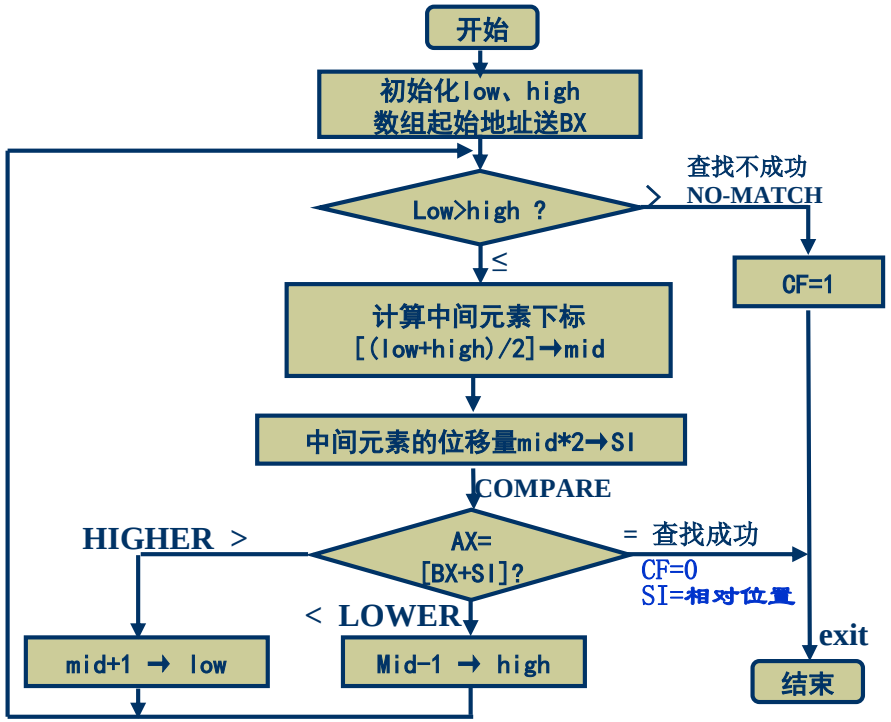
否则，计算中点： (low+high)/2 → mid;

(3)k与中点元素r[mid]比较。若k=r[mid]，则查找成功，程序结束；若k<r[mid]，则转步骤(4);若k>r[mid]，则转步骤(5)；（假设数据由小到大排列）

(4)低半部分查找(lower)，mid-1 → high，转步骤(2)；

(5)高半部分查找(lower)，mid+1 → low，转步骤(2) ；





```
dseg    segment
    low   dw   ?
    high  dw   ?
dseg    ends
cseg    segment
    assume  cs:cseg
    assume  ds:dseg, es:dseg
b_search  proc  near
    push ds      ; 保存主程序的DS内容
    push ax
    mov  ax, dseg
    mov  ds, ax
    pop  ax

chk_first:
    cmp  ax, es:[di+2]
    ja   chk_last     ;ax > A1
    mov  si, 2
    je   exit      ;ax = A1,CF=0
    jmp  no_match     ;ax < A1
chk_last:
    mov  si, es:[di]
    shl  si, 1
    mov  bx,di
    cmp  ax, es:[bx+si]
    jb   search       ;ax < An
    je   exit      ;ax = An,CF=0
    jmp  no_match     ;ax > An

search:
    mov  low_idx, 1
    mov  bx, es:[di]
    mov  high_idx, bx
    mov  bx, di

mid:
    mov  cx, low_idx
    mov  dx, high_idx
    cmp  cx, dx
    ja   no_match
    add  cx, dx
    shr  cx, 1  ;(cx+dx)/2
    mov  si, cx
    shl  si,1   ;SI*2→SI

compare:
    cmp  ax, es:[bx+si]
    je   exit   ;ax = [mid]
    ja   higher

lower:
    dec  cx
    mov  high_idx, cx
    jmp  mid

higher:
    inc  cx
    mov  low_idx, cx
    jmp  mid

no_match:
    stc      ;CF置1
exit:
    pop ds   ;恢复主程序的DS内容
    ret
b_search  endp
cseg    ends
    end
```

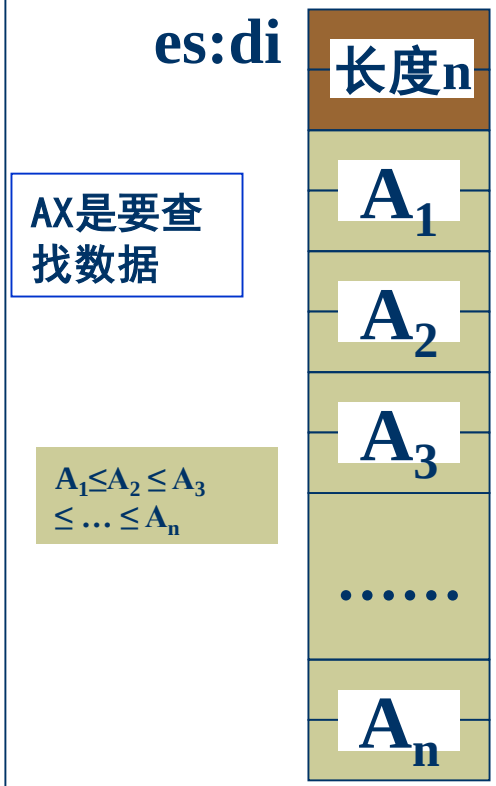
用存储单元做指针

查找子程序

子程序中保存、恢复哪些寄存器视具体情况定

si*2，数组元素为字

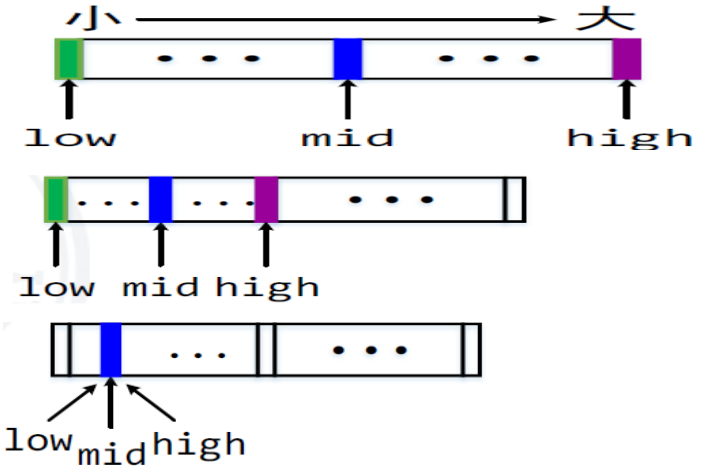
可以没有，能否减少程序执行时间？

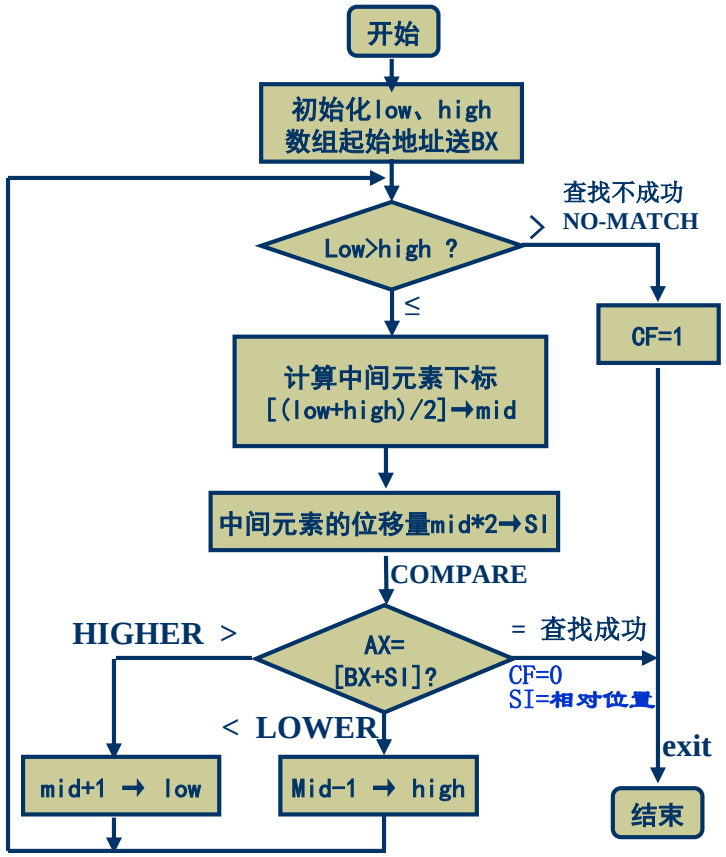


AX是要查找数据

$A_1 \le A_2 \le A_3 \le \ldots \le A_n$

找到：CF=0，SI=相对位置
未找到：CF=1






```
search:
    mov   low, 1
    mov   bx, es:[di]
    mov   high, bx
    mov   bx, di

mid:
    mov   cx, low
    mov   dx, high
    cmp   cx, dx
    ja    no_match
    add   cx, dx
    shr   cx, 1 ;(cx+dx)/2
    mov   si, cx
    shl   si,1   ;SI*2→SI

compare:
    cmp   ax, es:[bx+si]
    je    exit   ;ax = [mid]
    ja    higher

lower:
    dec   cx
    mov   high, cx
    jmp   mid

higher:
    inc   cx
    mov   low, cx
    jmp   mid
```

数组元素为字

[mid]=[bx+si]

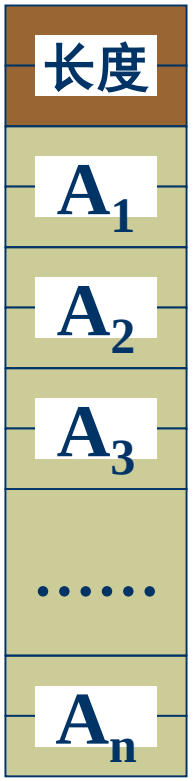
```
no_match:
    stc     ;CF置1
exit:
    pop ds
    ret
b_search  endp
cseg      ends
    end
```

AX是要查找数据

r[mid]=[bx+si]

$A_1 \le A_2 \le A_3 \le \ldots \le A_n$

es:di



找到：CF=0，SI=相对位置

未找到：CF=1

- 为了提高程序效率
  - 将最常用的数据尽量放在寄存器中
    - 例子5.9中，可用BP、DX替代变量low、high
  - 尽量合理分配使用寄存器

分支程序小结

（１）形成条件：CMP指令、运算类指令、TEST指令等

（２）实现转移：条件转移指令

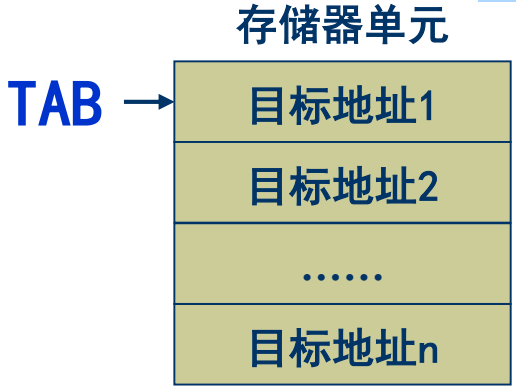
（３）逻辑尺方法实现2个以上分支

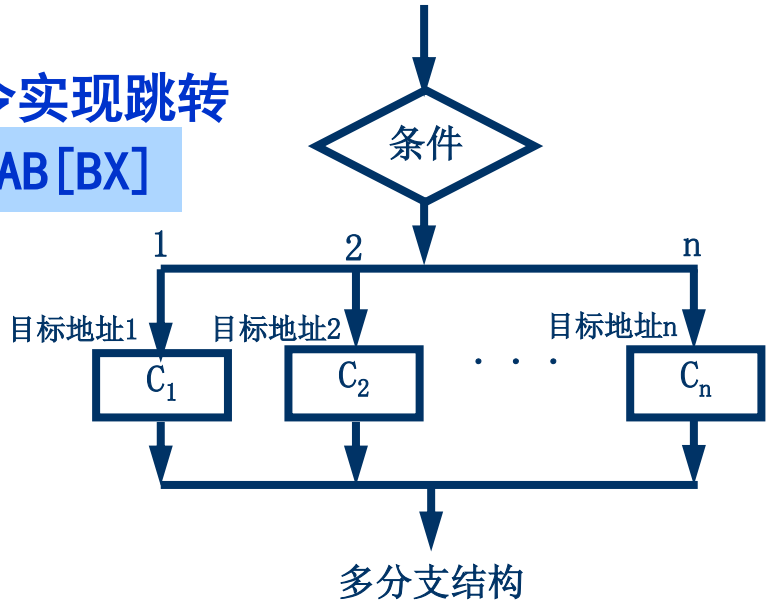
（４）循环结构可以认为是分支结构的一个特例，分支结构是循环结构的基本组成部分

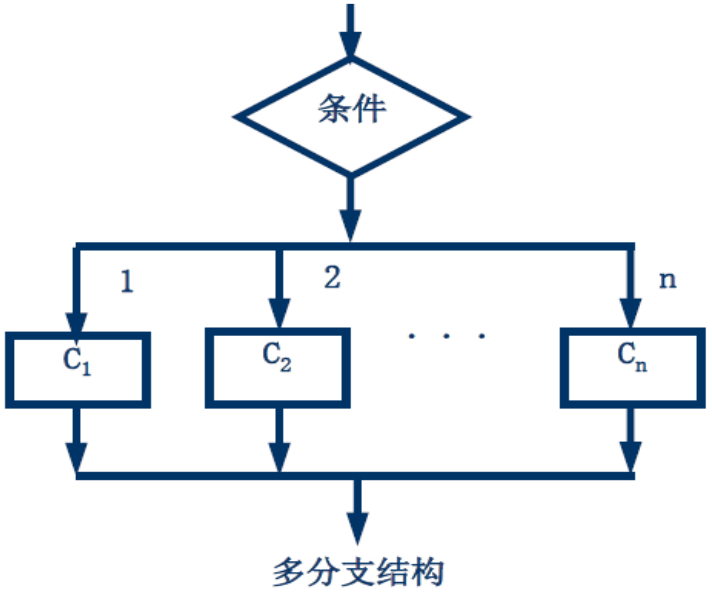
5.2.3 跳跃表法

- 分支程序的两种结构形式都可以用上面所述的基于判断跳转方法来实现。此外，在**实现CASE结构**时，还可以使用跳跃表法，使程序根据不同的条件转移到多个程序分支中去
- **利用跳跃表法实现多路分支，关键是：**
  - ① **构建跳转表**（分支转移目标地址表）
  - ② 灵活、正确使用**无条件间接转移指令实现跳转**

```
JMP word ptr TAB[BX]
```







JMP指令(参看P48)：

(1) JMP word ptr TAB[BX] ;(DS*16+TAB+BX) →IP

(2) JMP word ptr[BX][SI] ; (DS*16+BX+SI) →IP

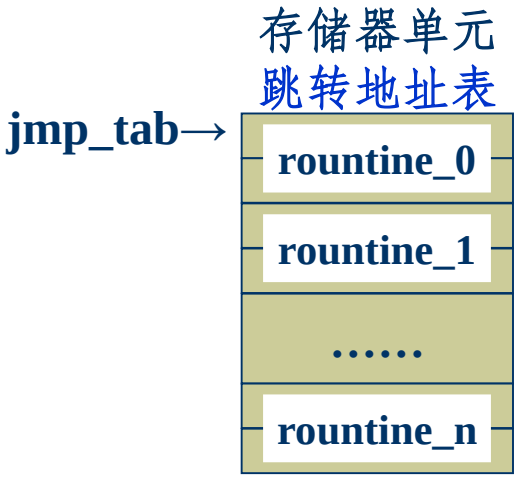
用子程序调用间接存储器寻址语句（CALL mem)可以实现什么？

灵活使用各种寻址方式，获得各种编程技巧。

下面举例说明跳跃表法编程

用无条件间接转移指令实现CASE转移

- 跳转表的转移目标地址数据定义：
  - 用DW定义 ←→ 段内间接转移指令（IP）
  - 用DD定义 ←→ 段间间接转移指令（IP, CS）



- 段内间接转移的几种格式：
  - JMP JMP_TAB[SI]　　;SI中为位移量
  - LEA BX, JMP_TAB /MOV BX, OFFSET JMP_TAB
    JMP WORD PTR [BX][SI] ;SI中为位移量
  - LEA BX,JMP_TAB
    ADD BX, 位移量
    JMP WORD PTR [BX]

段间时：jmp dword ptr [bx]

```
branch_addresses  segment
jmp_tab           dw  routine_0
                  dw  routine_1
                  ……
branch_addresses  ends
```

```
branch_addresses  segment
jmp_tab           dd  routine_0
                  dd  routine_1
                  ……
branch_addresses  ends
```

例5.10：利用跳转表实现程序的分支转移

设有一组选择项目（程序中有多个分支）（项目0、项目1、项目2、项目3、……、项目9），其执行标志分别存于累加器AX中的低10位中

AX	0	0	0	0	0	0	ACC.9	ACC.8	ACC.7	ACC.6	ACC.5	ACC.4	ACC.3	ACC.2	ACC.1	ACC.0

ACC.0（累加器的第0位）存放项目0的标志，ACC.1存放项目1的标志，…，ACC.9存放项目9的标志

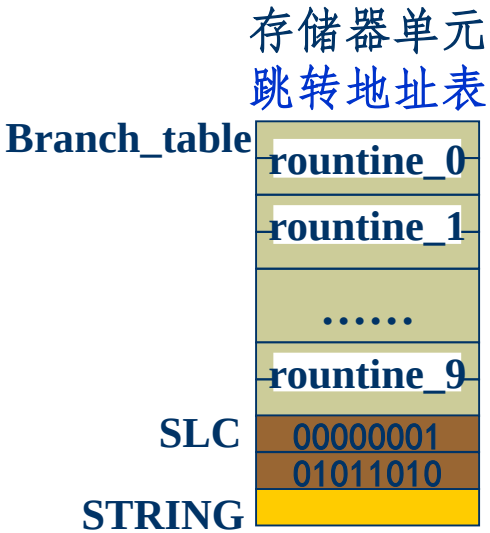
编程完成根据AX中哪一位为“1”，把程序分支转移到相应的项目中去执行

建立跳转表

```
Branch_addresses   segment     ;define data segment
Branch_table       dw   routine_0    ;分支程序0入口地址偏移量
                   dw   routine_1
                   dw   routine_2
                        …
                   dw   routine_9
SLC                dw    ?      ;项目选择标志
STRING             db    ?      ;处理项目的“提示”
Branch_addresses   ends
```

目标地址标号或子程序名

一般在程序中生成条件，条件也可放在寄存器中，这里不定义，程序中可灵活生成



如果段间远跳转时如何定义？
DW改成DD就行

跳转处理方法

- 根据间接转移寻址有多种转移方法：寄存器间接、基址变址、变址跳转法，跳转指令允许的间接寻址方式都可以

(1) 寄存器间接寻址方式实现

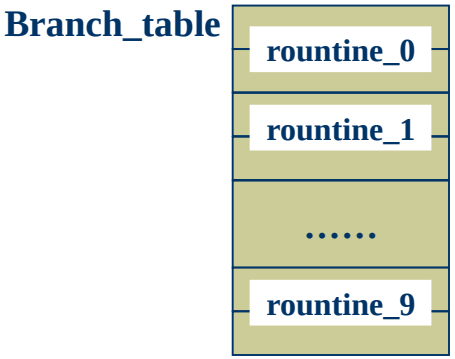
- BX指向跳转表开始地址

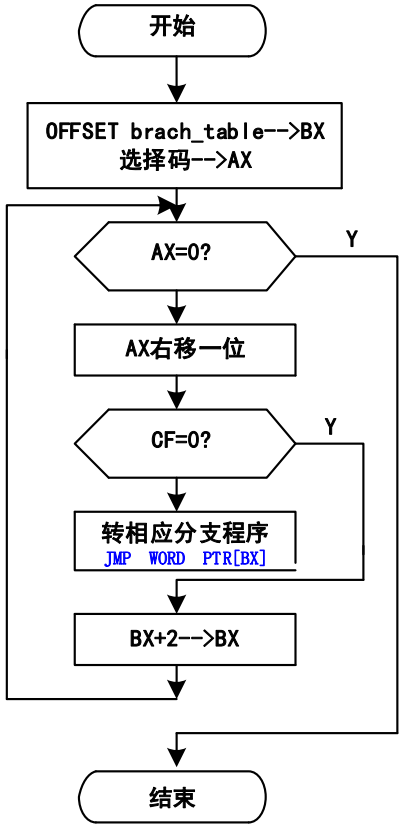
当ACC.0=1时，置BX指针指向routine_0

当ACC.1=1时，置BX指针指向routine_1

……

存储器单元
跳转地址表

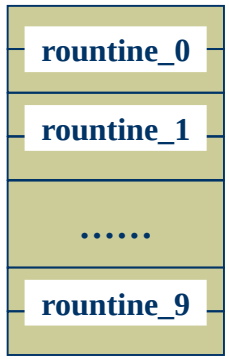




- 用add bx, type branch_table指向分支地址
- 用JMP WORD PTR[BX]实现转移

(2) 基址变址间接寻址方式实现

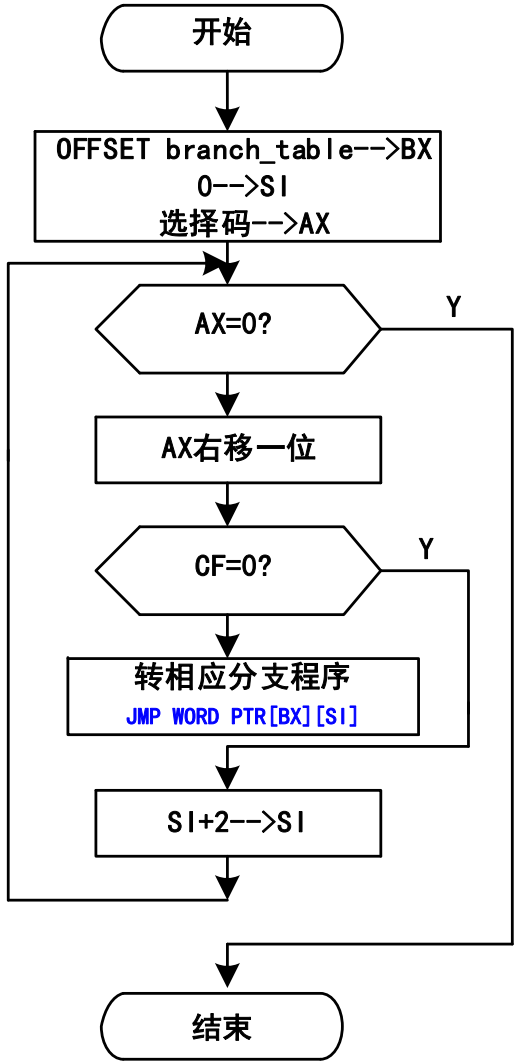
存储器单元
跳转地址表



- BX指向跳转表开始地址
- 0→SI

当ACC.0=1时，置[BX][SI]指针指向routine_0
当ACC.1=1时，置[BX][SI]指针指向routine_1
……

- 用add si, type branch_table指向分支地址相对位置
- 用JMP WORD PTR [BX][SI]实现转移

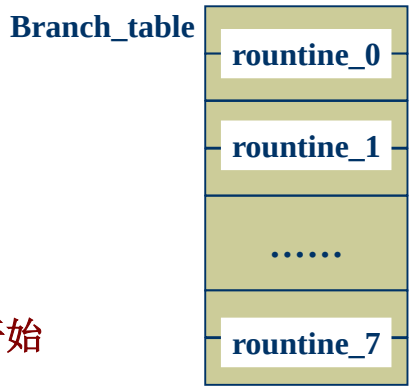


AX	0	0	0	0	0	0	ACC.9	ACC.8	ACC.7	ACC.6	ACC.5	ACC.4	ACC.3	ACC.2	ACC.1	ACC.0

```
procedure_select  segment
        ASSUME CS: procedure_select, DS:branch_addresses
main            proc   far
start:
      push  ds
      sub         bx,  bx
      push  bx
;
      mov   bx,  branch_addresses
      mov   ds,   bx
;
      cmp   al,   0
      je       continue_main_line
      lea     bx,   branch_table
      mov   si,    7*type branch_table  ;从ACC.7开始
      mov   cx,  8
l:    shl     al,   1
      jnb    not_yet  ;cf=0?
      jmp   word ptr[bx][si]
not_yet:
      sub  si,  type branch_table
      loop l
continue_main_line:
      ……
      ret
main          endp
;
routine_0:  ……   ;分支0
      ……
;
routine_1:  ……   ;分支1
      ……
      ……
;
routine_7:  ……    ;分支7
      ……
;
procedure_select  ends
      end   start
```

```
branch_addresses  segment
branch_table      dw  routine_0
                  dw  routine_1
                  dw  routine_2
                      …
                  dw  routine_7
branch_addresses  ends
```

存储器单元
跳转地址表

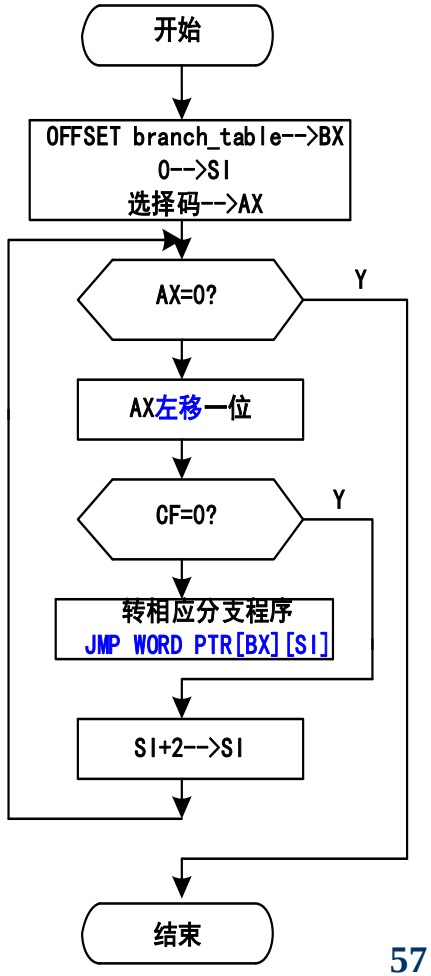



```
call   word ptr[bx][si]
```

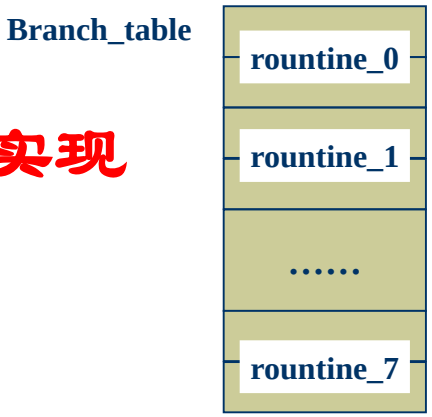
al

ACC.7	ACC.6	ACC.5	ACC.4	ACC.3	ACC.2	ACC.1	ACC.0

```
routine_0   proc   near   ;子程序0
            ……
            ret
routine_0   endp
;
routine_1   proc near     ;/子程序1
            ……
            ret
routine_1   endp;
            ……
```



存储器单元
跳转地址表



(3) 变址间接寻址方式实现

- 0→SI

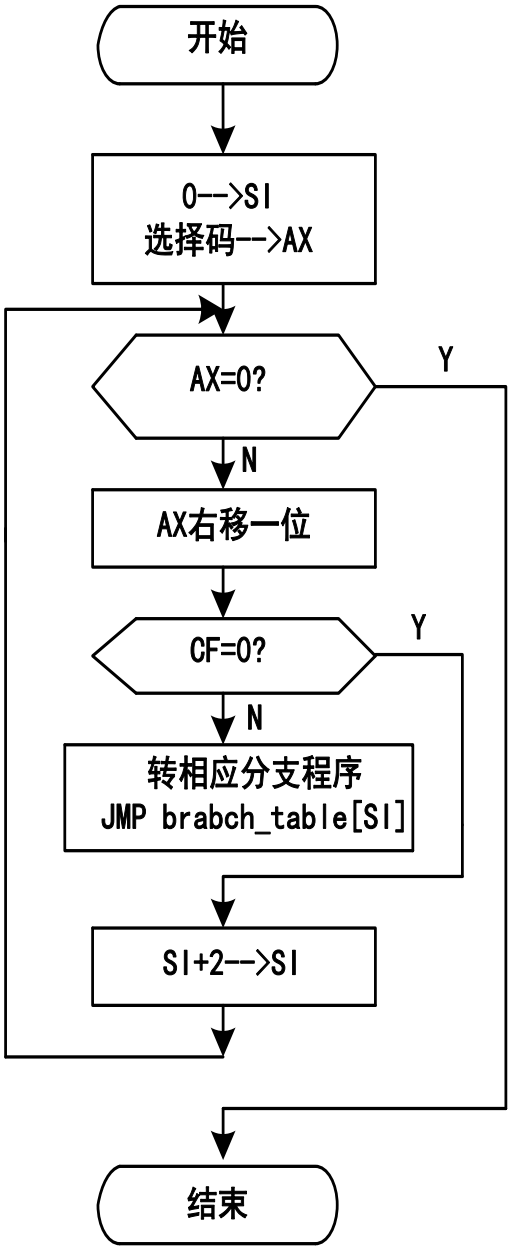
当ACC.0=1时，置BRANCH_TABLE[SI]指针指向routine_0

当ACC.1=1时，置BRANCH_TABLE[SI]指针指向routine_1

……

- 用add si, type branch_table指向分支地址相对位置
- 用JMP BRANCH_TABLE[SI]实现转移

CALL BRANCH_TABLE[SI]?
根据需要灵活使用指令即可



用ARM指令实现分支和循环

- 分支实现
- 循环实现

例1、用ARM指令实现一个分支

例1.实现符号函数Y的功能。

其中：-128≤X≤+127

$$Y=\begin{cases}1 & \text{当X＞0时}\\ 0 & \text{当X＝0时}\\ -1 & \text{当X＜0时}\end{cases}$$

常用指令：

- cmp 比较2个数的大小
- bgt 大于跳转
- blt 小于跳转
- beq 等于跳转

```
data segment
    X    db   ?    ; 被测数据
    Y    db   ?    ; 函数值单元
data ends
code segment
assume cs:code, ds:data
main proc far
    push  ds
    xor   ax, ax
    push  ax
    mov   ax, data ;设置段寄存器DS
    mov   ds, ax
    mov   al, 0
    cmp   X,  al
    jg    big
    jz    sav
    mov   al, 0FFH  ; 小于0
    jmp   short sav
big:mov   al, 1     ; 大于0
sav:mov   Y,  al    ; 保存结果
    ret
main endp
code ends
end main
```

x86

```
.data
    X: .dword 0x1    // 被测数据
    Y: .dword 0x0    // 函数值单元

.text
.type main, %function
.global main  //和C语言的main函数一样

main:
    eor   x0, x0, x0
    ldr   x2, X
    cmp   x2, x0
    bgt   BIG
    beq   SAV
    mov   x0, #-1     ; 小于0
    b     SAV
BIG:mov   x0, #1      ; 大于0
SAV:adr   x1, Y
    str   x0, [x1]    ; 保存结果

    mov   x0, #0
    ret
```

ARM64

例2、把Xn寄存器内的二进制数用十六进制数的形式在屏幕上显示

分析问题：把Xn寄存器中64位的二进制数用4位十六进制数的形式在屏幕上显示

1、初始设置

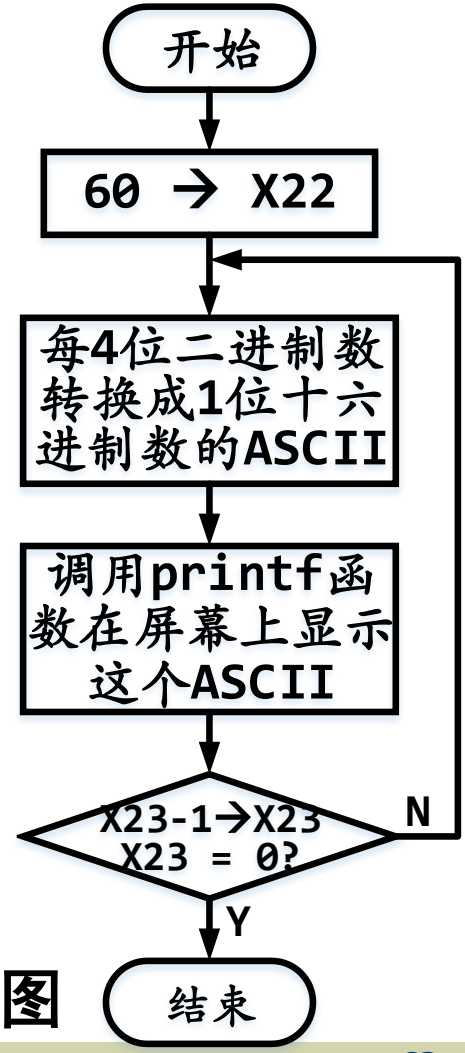
- 循环次数：每次显示1个十六进制数（送显示器相应的ASCII），循环次数=16

2、循环体

- 根据任务，选择算法
  - 每4位二进制数转换成1位十六进制数的ASCII
  - 调用libC库的printf函数，或者Linux的系统功能在屏幕上显示
- 修改指针：无
- 设置循环控制转移其他条件：无

3、循环控制转移

- 根据计数控制循环次数



设计粗流程图

用16进制字符显示寄存器中的值

```
.data
var: .dword 0x12Ab34cD56Ef7890
hexfmt: .asciz  "%X"
newline:.asciz "\n"

.text
.type main, %function
.global main


main:   //GCC链接
    stp  x29, x30, [sp, #-16]!
    ldr  x22, var
    // 64位寄存器，包含16个hex
    mov  x23, 16
```

子程序调用时将返回地址自动保存在X30.入栈保存

```
LOOP:
    ror  x22, x22, #60
    and  x1, x22, 0x0F
//  用16进制显示低4位
    adr  x0, hexfmt
    bl   printf // libC库提供的功能

    subs x23, x23, 1
    bne  LOOP //不等于0继续循环

// 输出回车换行符
    adr  x0, newline
    bl   printf

    ldp  x29, x30, [sp], #16
    mov  x0, 0
    ret   //X30内容送PC返回
```

字符显示

- 使用printf函数显示字符：
  - printf函数是libC库提供的功能，通过汇编调用时
    - X0中包含的是指向格式化显示字符串的符号地址(指针)
    - X1包含的是要显示数据值，这里就是寄存器的低4位
  - libC库函数调用时，要遵循程序调用中的寄存器使用规则
    - X0-X7 传递函数的参数，返回值放在X0中
    - X19-X28寄存器的值会由被调用的函数保存，但其它寄存器的值需要由调用者保存。
  - 包含C库函数时，用gcc进行程序链接比较简单

字符显示

- 使用Linux的系统功能显示字符：
  - printf函数也是对操作系统（这里是Linux）提供的write系统调用的封装
  - write系统调用的参数为：
    - X8：0x40，表示调用sys_write系统调用
    - X0: 输出的文件句柄，标准输出（屏幕）值为1
    - X1：要显示的字符串的首地址（指针）
    - X2：要显示的字符串的长度
  - 系统调用的方式
    - svc #0

```
.data
var: .dword 0x12Ab34cD56Ef7890
hexfmt: .dword 0x0     //字符串
newline:.asciz "\n"

.text
.global _start

_start:    //用LD链接
    ldr   x22, var
    mov   x23, 16
LOOP:
    ror   x22, x22, #60
    and   x0, x22, 0x0F //低4位
    // 把字符串的首地址放在X1中
    orr   x0, x0, 0x30 //ASCII
    cmp   x0, 0x3a   // A-F
    blt   print
    add   x0, x0, #7 // A-F
```

```
print:
 //一次显示1个字符，只用第1个字节
    adr   x1, hexfmt
    str   x0, [x1]
    mov   x8, 0x40 // sys_write
    mov   x0, #1   // 输出到屏幕
    mov   x2, #1   // 显示1个字符
    svc   #0

    subs x23, x23, 1
    bne   LOOP
    // 显示回车换行符号
    adr   x1, newline
    mov   x8, 0x40 // sys_write
    mov   x0, #1   // 输出到屏幕
    mov   x2, #1   // 显示1个字符
    svc   #0
    // 返回Linux
    mov   x0, 0
    mov   x8, #0x5D    // exit
    svc   #0
```

返回Linux系统的两种方式

函数方式返回

```
.text
    .global  main
    .type    main, %function
    .global main
main:
    stp  x29, x30, [sp, #-16]!
    ...
    ...
    ...
    ldp  x29, x30, [sp], #16
    mov  x0, 0
    ret

// 用gcc进行链接
```

- 函数返回
  - 定义main函数
  - 进入main函数后，首先保存X29和X30寄存器的内容，包含了返回地址
  - 在函数结束后，使用X0保存返回代码，0表示正常返回，通过ret指令返回Linux OS

返回Linux系统的两种方式

系统调用返回

```
.text
.global _start

_start:
    ...
    ...
    ...
    mov   x0, 0
    mov   x8, #0x5D    // exit
    svc   #0

//用 ld 进行链接
//若使用了glibC中的库函数，需要使用 ld -lc进行链接
```

- 系统调用返回
  - 使用Linux的系统调用exit返回
  - X8为0x5D，表示exit系统调用
  - X0保存返回代码， 0表示正常返回，通过软中断 svc #0 指令返回Linux OS

例3、数“1”的个数

方法一：

（1）循环控制条件：计数方式，循环次数一定

（2）数“1”：使用左移，由于移位操作不影响NZCV标志位，需要测试最高位是否为1

方法二：（优化的方法）

（1）循环控制条件

- 不使用计数方式，而是使用查找完毕，以减少循环次数，提高程序效率
  - 查找完毕，循环次数不定，用条件控制（连续0位串长度为64）

（2）数“1”的方法

- 联合使用clz、cls和左移操作进行0，1位串查找

方法一：统计“1”的个数

```
.data
    var: .dword 0x1234a000
    // 用十进制显示1的个数
    mesg: .asciz "%d\n“

.text
.global _start

_start:
    ldr  x0, var
    //设置X1的最高位为1，其余为0
    movz x1, #0x8000, lsl #48
    mov  x2, #64  // 循环64次
    // 1的总数保存在x3中
    eor  x3, x3, x3
```

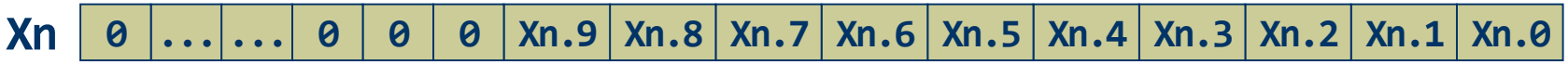
```
LOOP:
    tst  x0, x1
    bpl  NEXTBIT     // 为0跳转
    add  x3, x3, #1 // 为1计数
NEXTBIT:
    lsl  x0, x0, #1 // 左移1位
    subs x2, x2, #1 // 次数减1
    bgt  LOOP  // 大于0继续循环
    // 字符串首地址保存在X0中
    adr x0, mesg
    mov x1, x3    // 1的个数
    bl  printf    // 输出1的个数
    mov x0, #0    // 返回值为0
    mov x8, 0x5d // exit
    svc #0
```

方法二：统计“1”的个数

```
.data
    var: .dword 0x1234a0f1
    mesg: .asciz "%d\n"
.text
.global _start
_start:
    ldr  x0, var
    eor  x3, x3, x3  //1的个数
LOOP:
    clz  x2, x0  // 先统计0位串
    cmp  x2, #64 // 是否全为0
    beq  FINISH  // 全0，完成
    //左移0位串长度后最高位必为1
    lsl  x0, x0, x2
    cls  x2, x0  // 再统计1位串
    //除最高位外，1位串长度为0?
    cmp  x2, #0
    beq  NEXTPART
    // 1位串长度为x2
    add  x3, x3, x2  // 计数
    lsl  x0, x0, x2  // 左移
NEXTPART:   // 记录最高位，左移
    add  x3, x3, #1
    lsl  x0, x0, #1
    b    LOOP   // 继续处理位串

FINISH:
    // 使用printf显示1的数量
    adr x0, mesg
    mov x1, x3  // 1的数量
    bl  printf

    mov x0, #0
    mov x8, 0x5d   // exit
    svc #0
```

例4、利用跳转表实现程序的分支转移

设有一组选择项目（程序中有多个分支）（项目0、项目1、项目2、项目3、……、项目9），其执行标志分别存于存储单元或寄存器Xn中的低10位中



Xn.0（寄存器Xn的第0位）存放项目0的标志，Xn.1存放项目1的标志，…，Xn.9存放项目9的标志

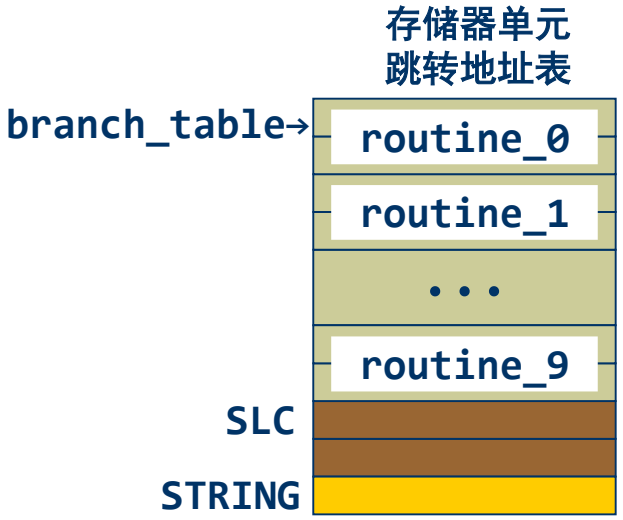
编程完成根据寄存器Xn中哪一位为“1”，把程序分支转移到相应的项目中去执行

建立跳转表

```
branch_table: .dword  routine_0   ;分支程序0入口地址偏移量
              .dword  routine_1
              .dword  routine_2
              …
              .dword  routine_9
SLC           .dword   ?    ;项目选择标志
STRING        .asciz   ?    ;处理项目的“提示”
```

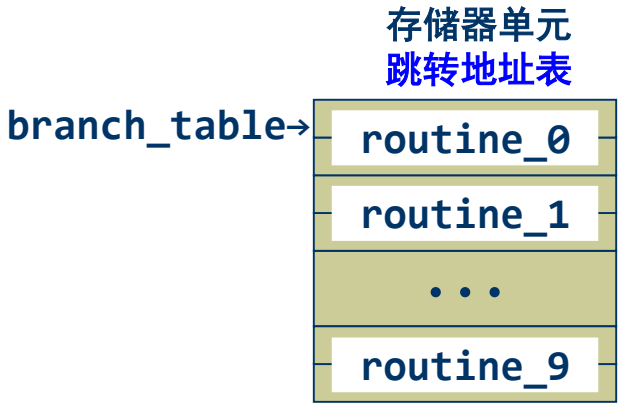
一般在程序中生成条件，条件也可放在寄存器中，这里不定义，程序中可灵活生成

ARM64是64位地址



跳转处理方法

使用寄存器寻址方式实现



假设用X21指向跳转表开始地址

当Xn.0=1时，置X21指针指向routine_0

当Xn.1=1时，置X21指针指向routine_1

...

- 用ldr　X22，[X21] 指向分支地址
- 用blr　X22 实现转移

跳转表和处理函数定义

```
.data
.balign 8
        branch_table:  .dword routine_0, routine_1
                       .dword routine_2, routine_3
                       .dword routine_4, routine_5
                       .dword routine_6, routine_7
                       .dword routine_8, routine_9
        SLC:           .dword 0xa3   // 支持调用多个函数
        mesg:          .asciz "Call function %d...\n"
.text
.global _start
routine_0:  // routine_1, routine_2, ..., 的定义类似
        stp  x29, x30, [sp, #-16]!
        adr  x0, mesg     // 要输出的字符串首地址
        mov  x1, #0       // 表示调用第0个函数进行处理
        bl   printf       // 用输出字符串表示函数执行过程
        ldp  x29, x30, [sp], #16
        ret
```

主程序处理逻辑

```
_start: adr   x21, branch_table     // 得到跳转表首地址
        ldr   x22, SLC              // 得到选择标志
        cmp   x22, #0               // 是否所有位均为0?
        beq   FINISH
        eor   x23, x23, x23  // 保存 正在处理第几位
LOOP:   tst   x22, #1
        beq   NEXTBIT       // 该位为0跳转
        // 该位为1, 得到该位对应的函数地址
        ldr   x25, [x21, x23, lsl #3]  //函数地址占8字节
        blr   x25           // 调用对应的函数
NEXTBIT: lsr  x22, x22, #1  // 准备处理下一位
        add   x23, x23, #1
        cmp   x23, #10      // 跳跃表仅支持10个函数
        blt   LOOP          // 继续处理更多的函数调用
FINISH:
        mov   x0, #0
        mov   x8, #0x5D     //exit
        svc   #0
```

谢谢！