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

5.3　如何在实模式下发挥 80386 及其后继机型的优势

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
- 掌握DOS系统功能调用

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；

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；

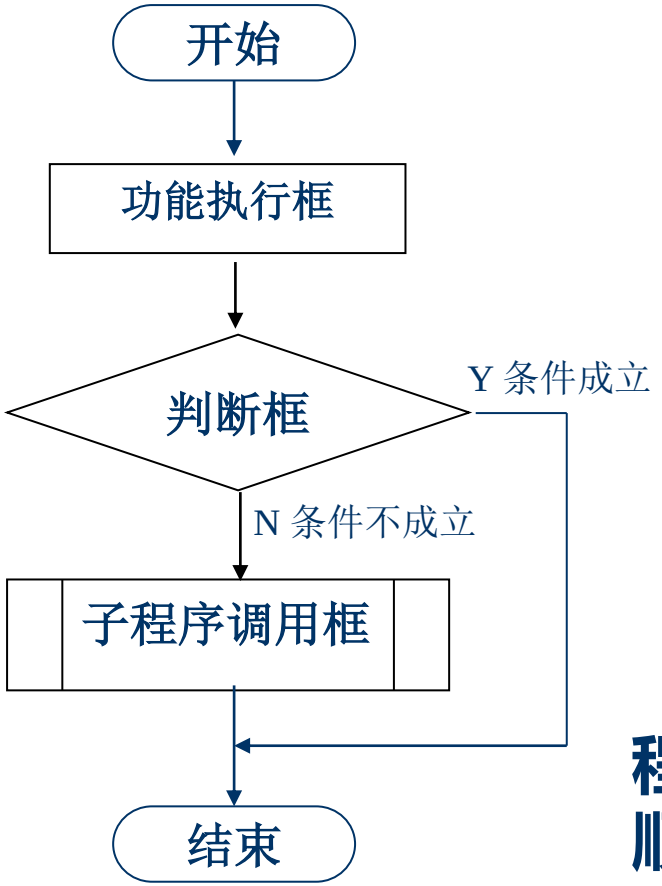
3. **绘制流程图** 设计整个程序处理的逻辑结构，从粗流程到细流程；

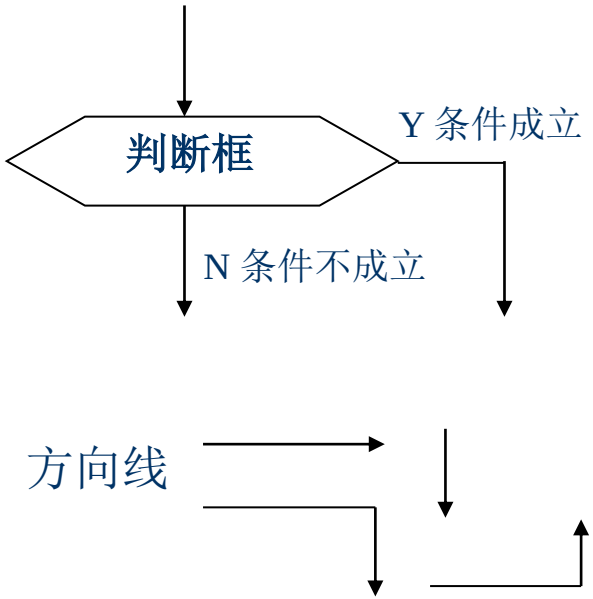
4. **存储空间分配** 分配数据段、堆栈段和代码段的存储空间，分配工作单元，借助数据段、堆栈段和代码段定义的伪操作实现；

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。

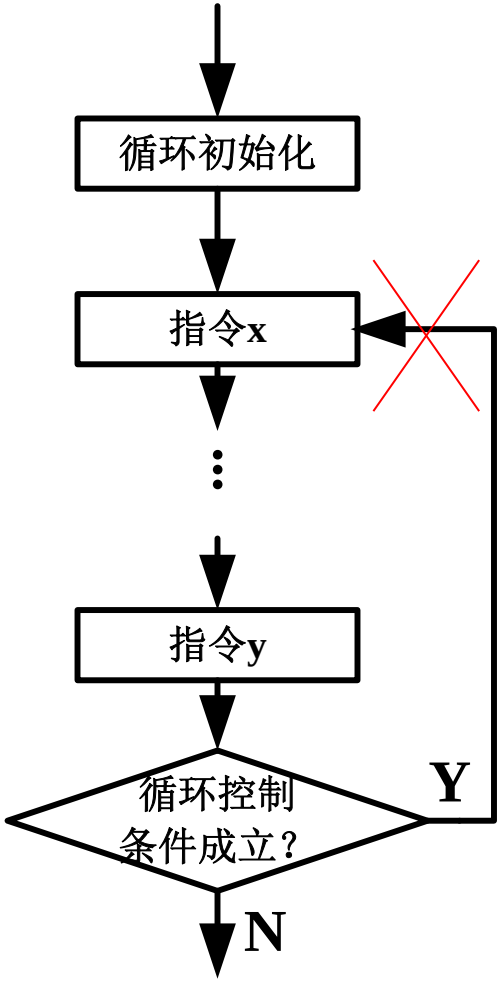
程序流程图画法规定

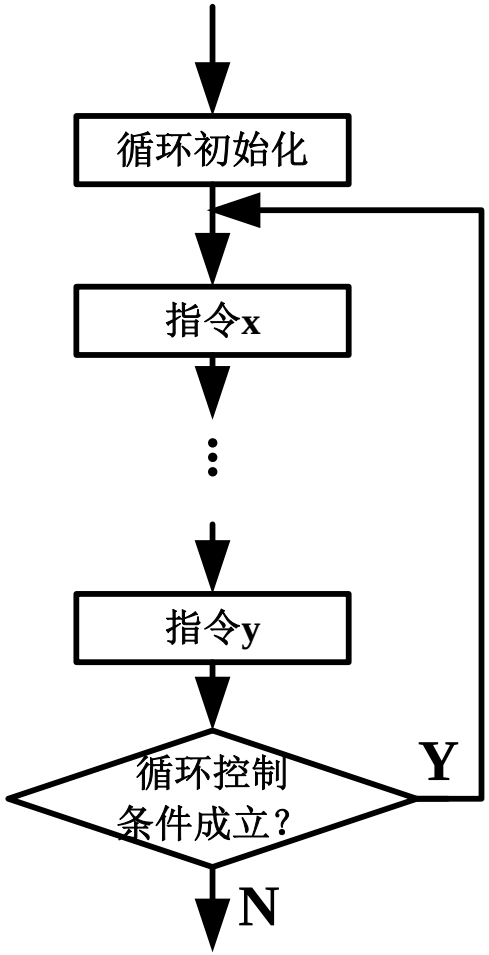




程序结构形式：
顺序、循环、分支和子程序

注意不正确画法

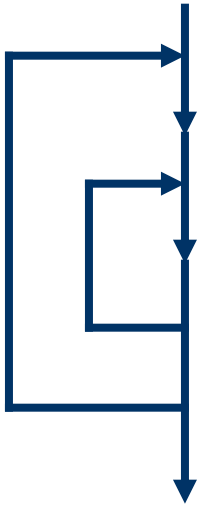




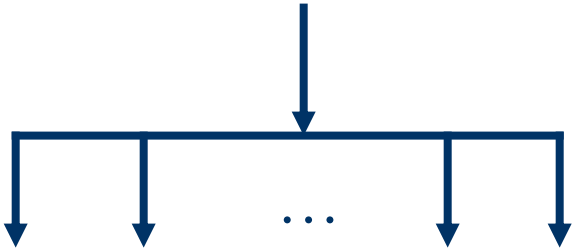
程序结构

顺序结构

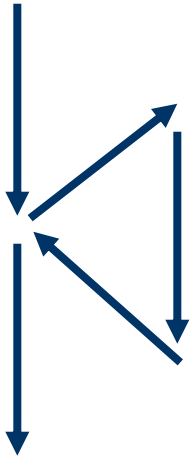
循环结构



分支结构



子程序结构



复合结构：多种程序结构的组合

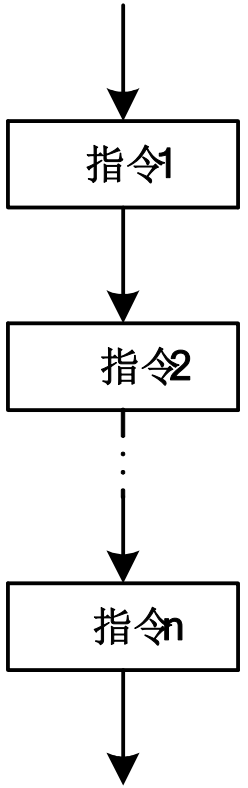
顺序程序设计方法

- 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

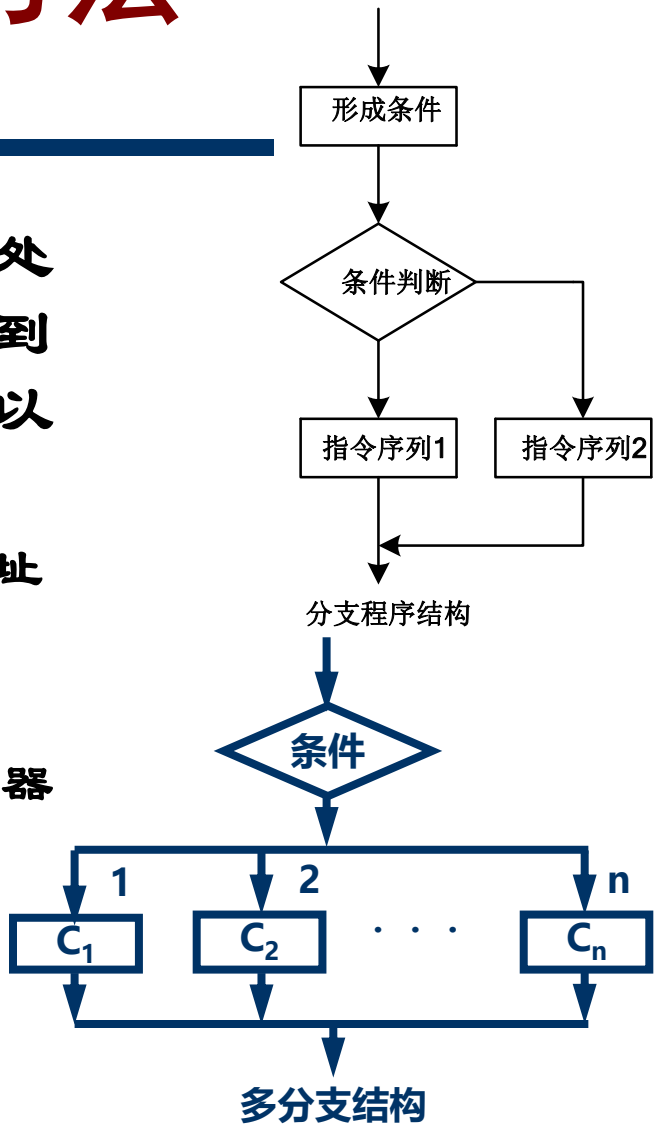
- **程序设计中最基本的结构**



顺序程序结构

分支程序设计方法

- 分支程序是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**：条件转移，标志寄存器，直接寻址
    - 满足条件（条件成立）
    - 不满足条件（条件不成立）
  - **多分支**：无条件转移，变址寄存器，存储器间接寻址
  - 程序每次只能执行其中一个分支



例1. 实现符号函数Y的功能。
其中：-128≤X≤+127

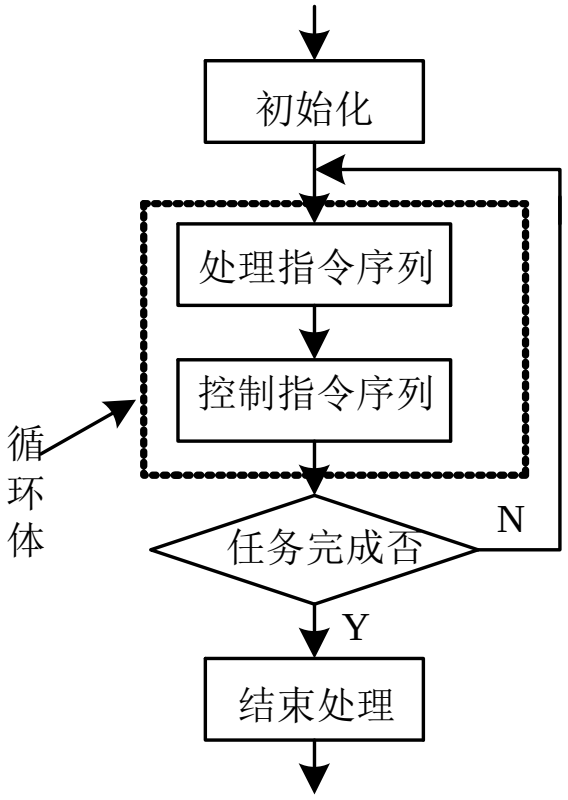
$$Y=\begin{cases}1 & \text{当}X>0\text{时}\\ 0 & \text{当}X=0\text{时}\\ -1 & \text{当}X<0\text{时}\end{cases}$$

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

      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
```

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
  - 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

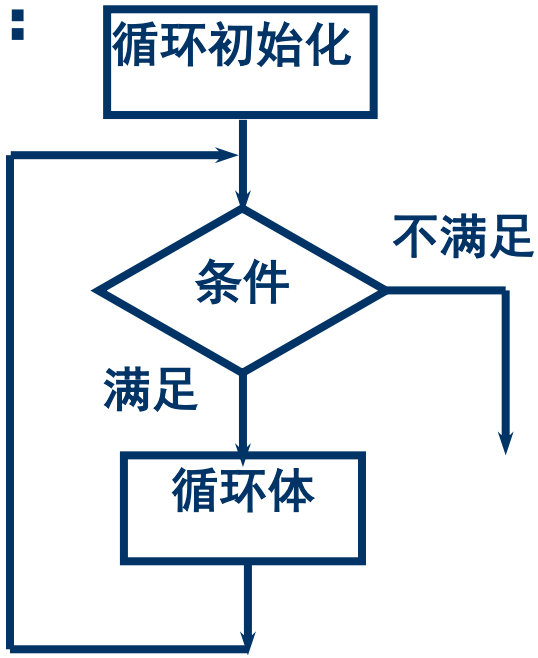
5.1.1 循环程序的结构形式

5.1.2 循环程序的设计方法

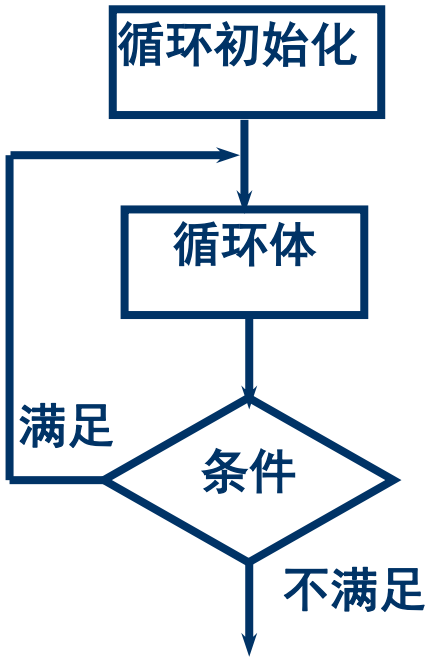
5.1.3 多重循环程序设计

5.1.1 循环程序的基本结构

- 两种基本结构
- 三个基本组成部分：
  - 1、初始设置
  - 2、循环体
  - 3、循环控制转移



DO-WHILE结构



DO-UNTIL结构

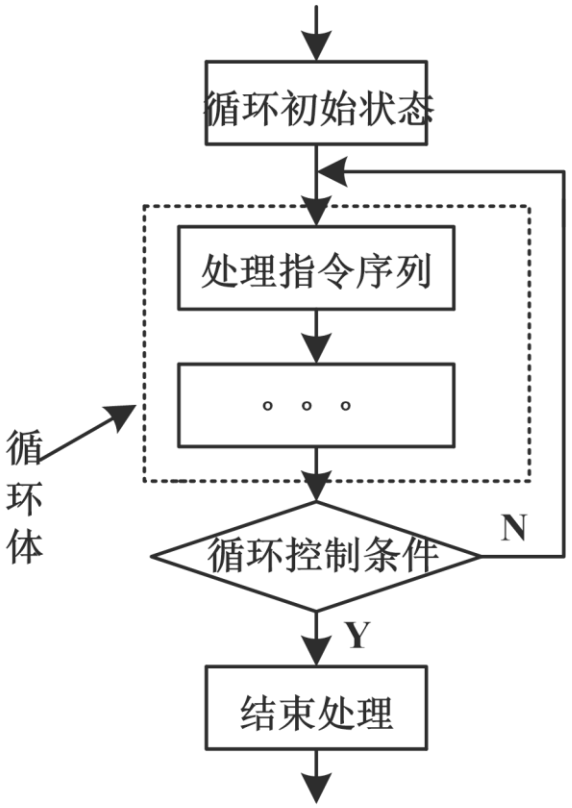
5.1.2 循环程序的设计方法

- 分析任务，实现三要素：

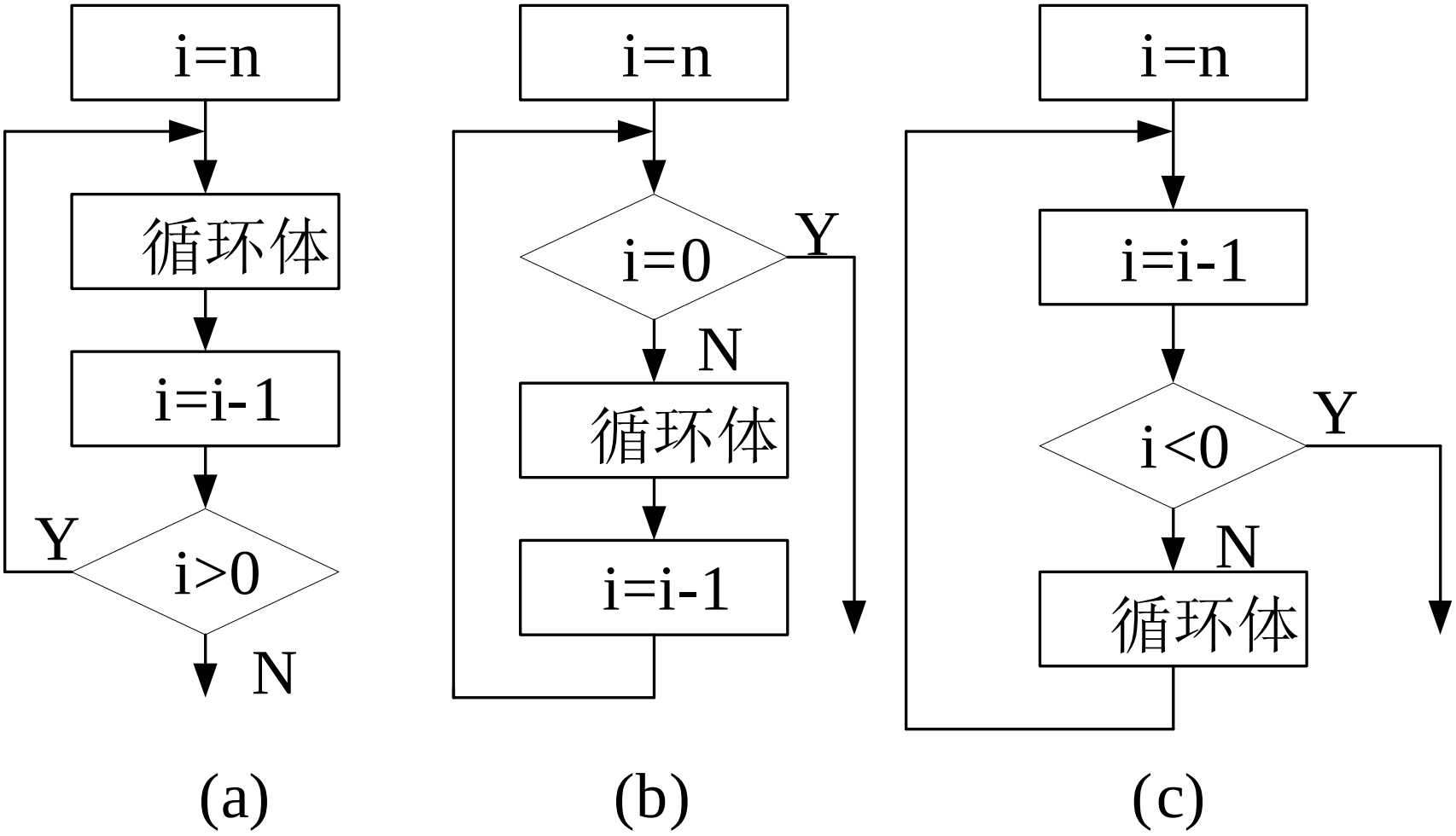
  1、初始设置
  - 循环次数，数据和变址指针初始化等

  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

  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



- 设计流程框图



- 不影响程序执行结果，只是性能优化的问题，有时性能可能差异较大
- 在自己的计算机上试一试(a)(b)两种循环体的运行时间

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

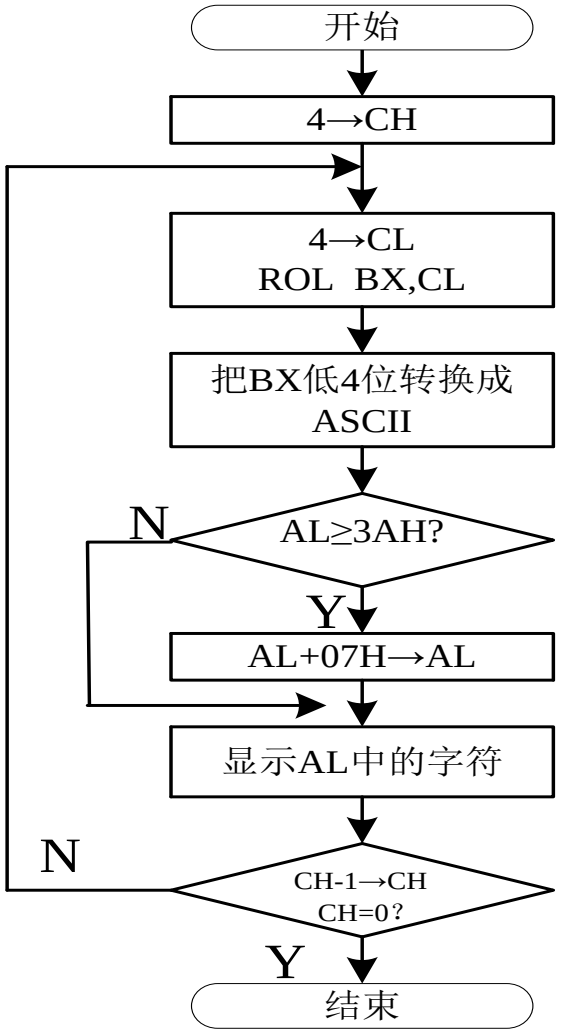
- 循环次数：每次显示1个十六进制数（送显示器相应的ASCII），循环次数=4，CH=4

2、循环体

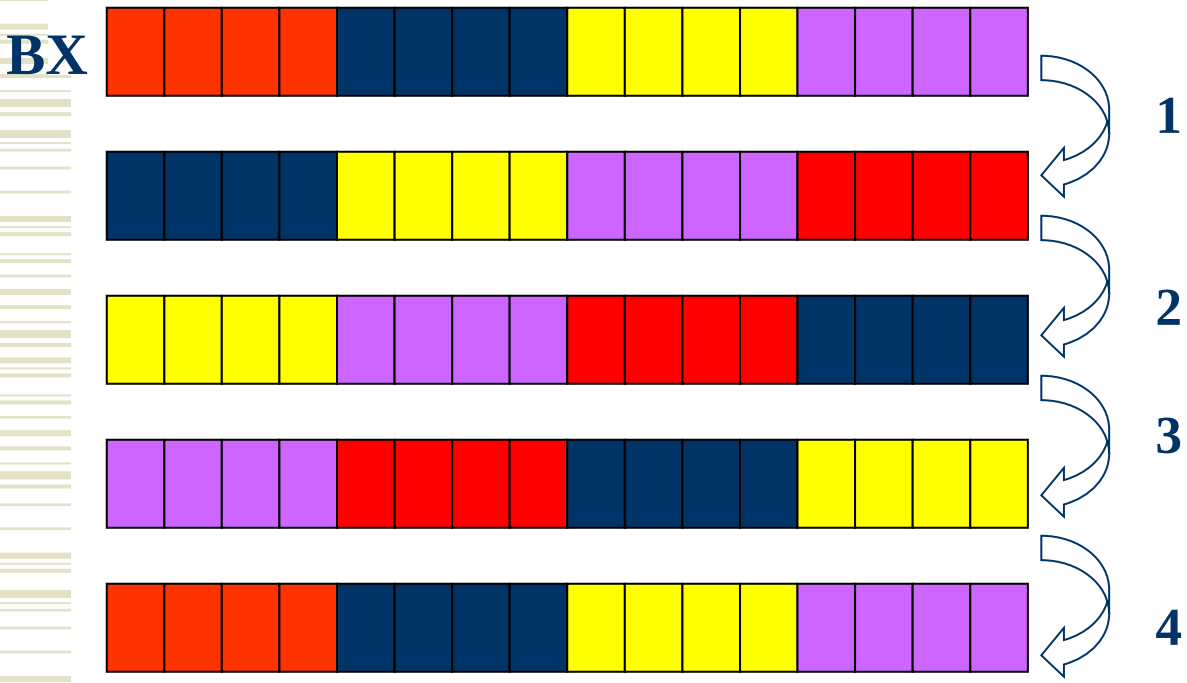
- 根据任务，选择算法
  - 每4位二进制数转换成1位十六进制数的ASCII
  - 调用DOS系统功能在屏幕上显示

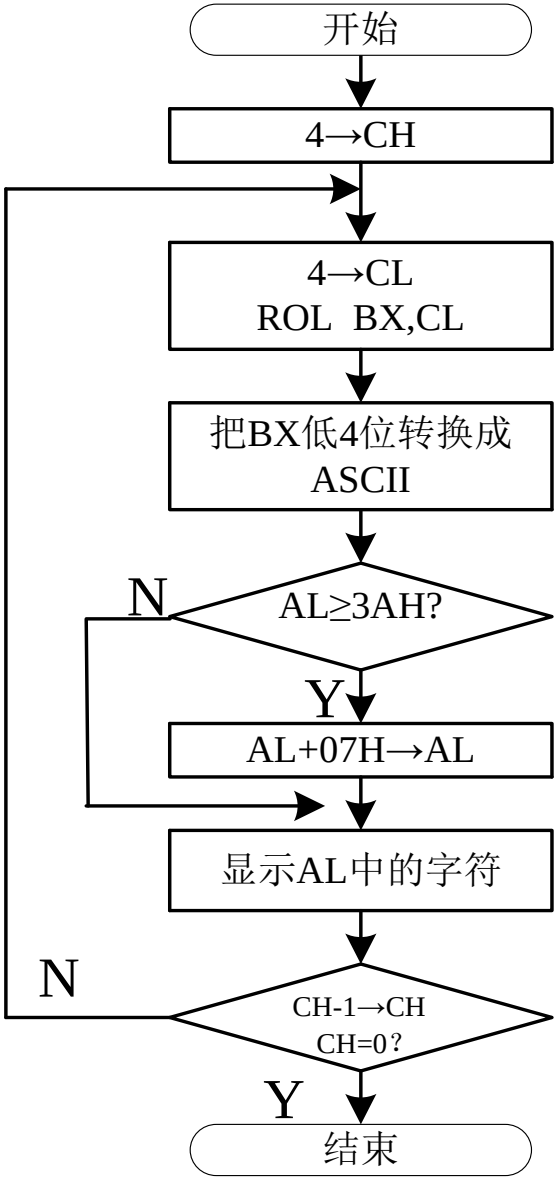
3、循环控制转移

- 根据计数控制循环次数

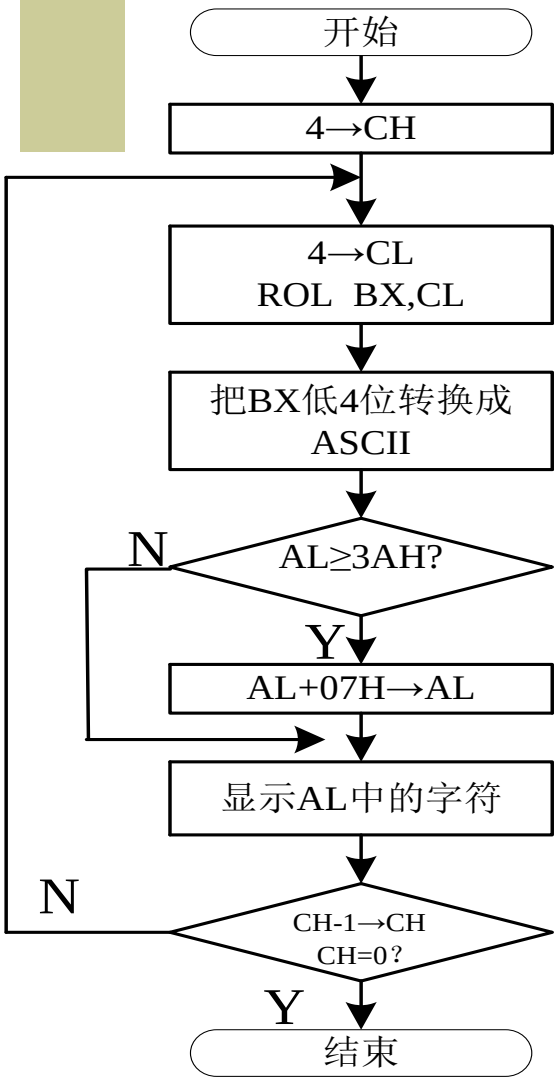


流程图


BX
1
2
3
4


开始
4→CH
4→CL
ROL BX,CL
把BX低4位转换成ASCII
AL≥3AH?
N
Y
AL+07H→AL
显示AL中的字符
CH-1→CH
CH=0?
N
Y
结束

没有数据分配问题，直接编写程序代码段



流程图

```
prognam segment
main proc far
        Assume cs:prognam
start:
        push ds
        sub ax, ax
        push ax
        mov ch, 4
rotate: mov cl, 4
        …
printit:
        mov dl, al
        mov ah, 2
        int 21h
        dec ch
        jnz rotate
        ret
main    endp
prognam ends
        end start
```

初始化

循环体

循环控制转移

系统调用
显示1个字符

DOS环境下
返回DOS

熟记程序框架结构和系统调用约定

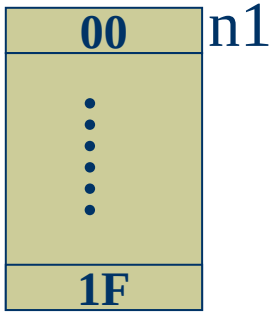
```
        mov   ch, 4
rotate: mov   cl,  4
        rol   bx, cl
        mov   al,  bl
        and   al,  0fh          ;al的低4位0-9，A-F
        add   al,  30h          ;al的低4位30-39，3A-3F
        cmp   al,  3ah
        jl    printit                ;'0'-'9' ASCII 30H-39H
        add   al,  7h          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

例5.1几点说明：

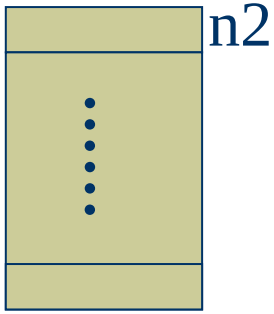
- 程序实现没有使用LOOP指令
  - 解决寄存器使用冲突
  - 用计数值控制循环结束，不是非得用LOOP指令
- 关于循环次数控制
  - 也可以把计数值初始化为0，每循环一次加1，然后比较判断
- 也可用LOOP指令
  - 通过堆栈保存信息解决CX冲突问题

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



1）任务1：用程序设置数据

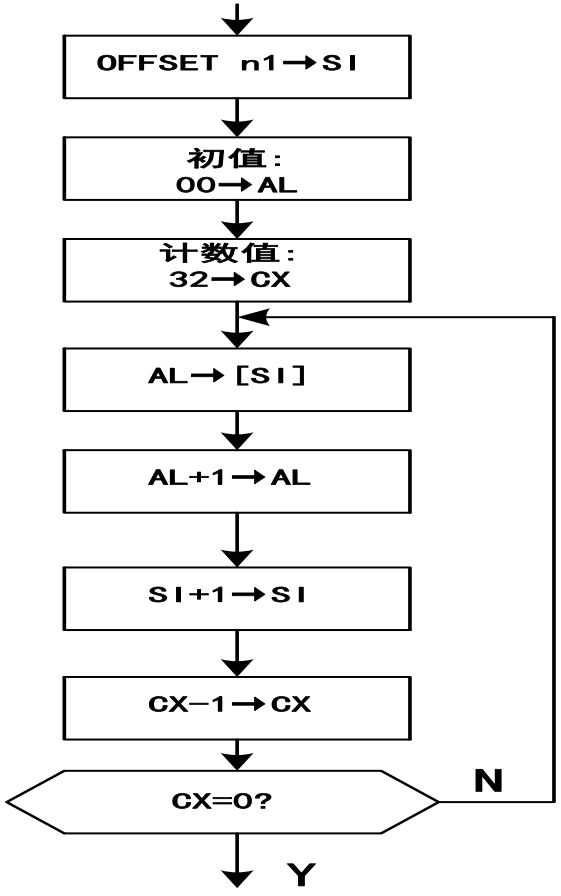
给内存数据段（DATA）中偏移地址为n1开始的连续32个字节单元，置入数据00H，01H，02H，……，1FH

- 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
- 多次同样功能的处理，数据处理有规律，
- 采用循环程序结构

1、初始设置：
- 循环次数：连续32个字节单元，循环次数=32
- 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
- 变址指针初始化：内存数据段中偏移地址为n1

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

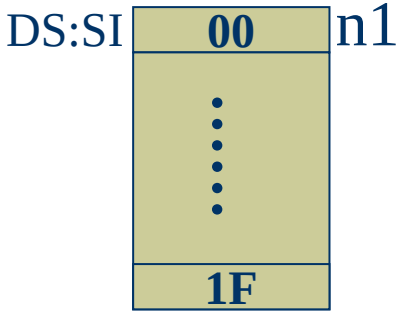
3、循环控制转移
- 根据计数控制循环次数

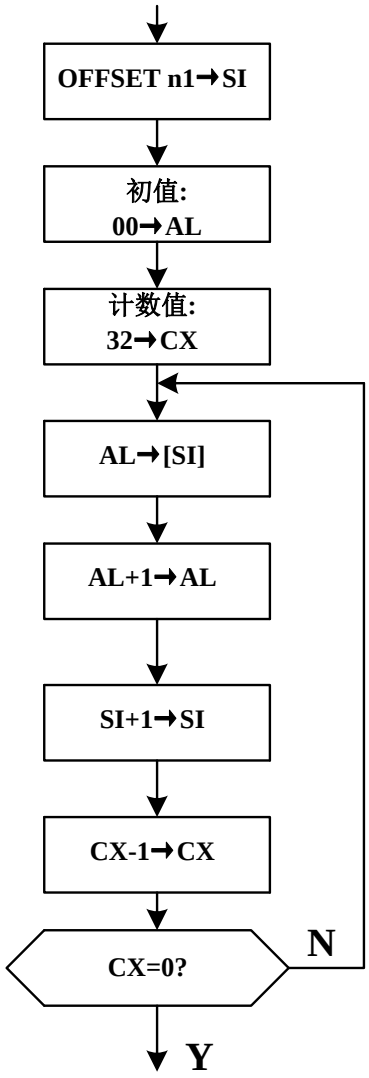


置入数据程序流程图

选择寄存器
进一步细化程序流程

- 循环的初始化部分完成
  - (1) 将n1的偏移地址置入变址寄存器SI，这里的变址寄存器作为地址指针用
  - (2) 待传送数据的起始值00H 送 AL
  - (3) 循环计数值32（或20H）送计数寄存器CX
- 循环体中完成
  - (4) AL中内容送地址指针所指存储器单元
  - (5) AL加1送AL；依次得到01H，02H，03H，…，1FH
  - (6) 地址指针SI加1，指向下一个地址单元
- 循环结束判断
  - (7) 计数器CX-1→CX
  - (8) 若CX≠0继续循环；否则结束循环
- 程序源代码请自己编写



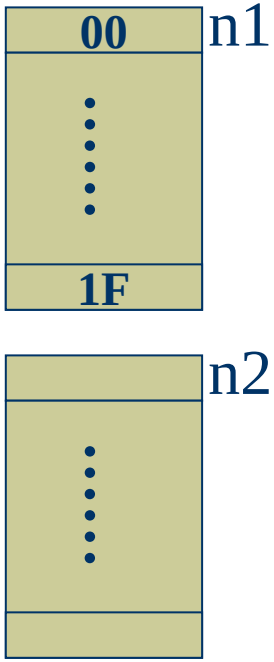


置入数据程序流程图

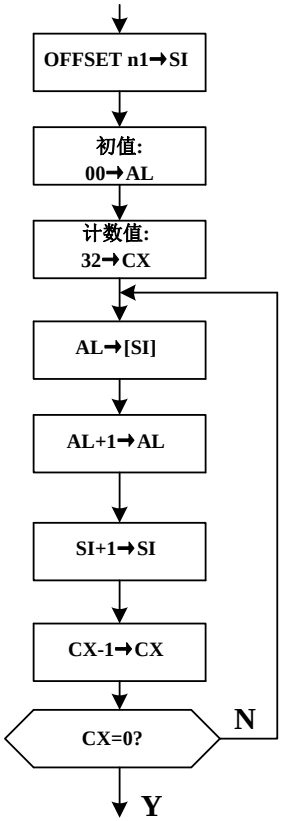
2)任务2：移动数据

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

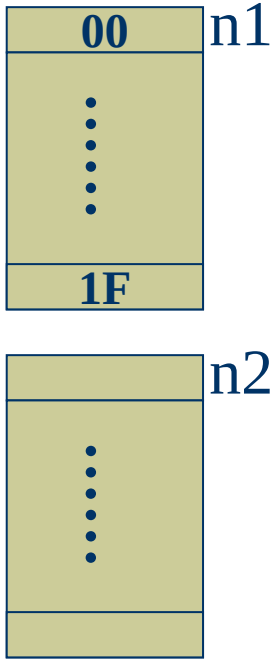


程序源代码请自己编写



置入数据程序流程图

```
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

          mov si,  offset n1
          mov di,  offset n2
          cld
          mov cx,  32
          rep  movsb

          ret

main      endp
prognam   ends
          end  start
```



例子5.2 数“1”的个数

在addr单元中存放着数 Y 的地址，把 Y 中1的个数存入 COUNT 单元中

参看p178

（1）数“1”的方法

a） 移位到进位，测试进位；测试符号位

b） 判最高位是否为1

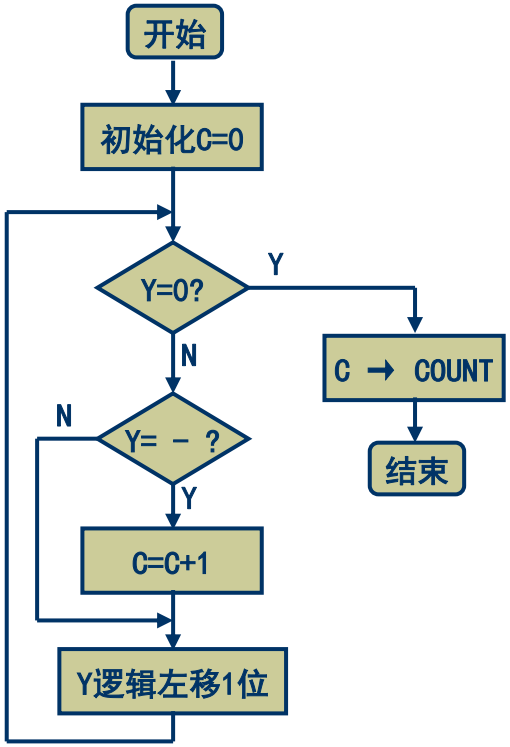
（2）循环控制条件

a） 简单思路：计数值以16控制

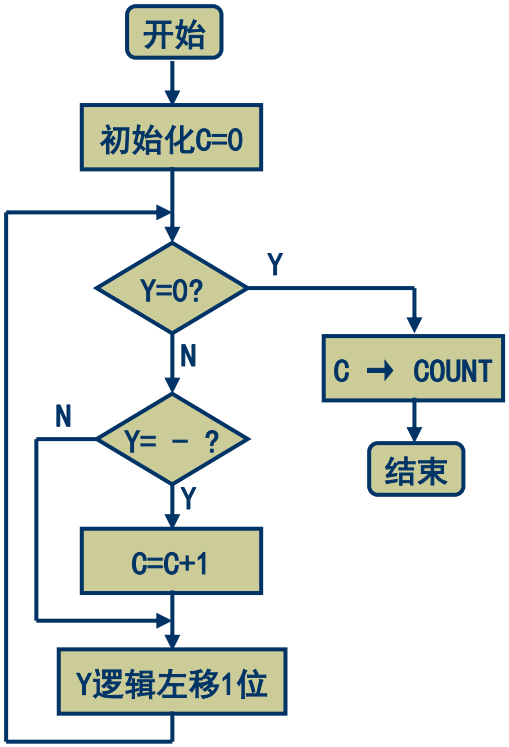
b） 比较好的方法：简单思路结合测试数是否为0

（3）DO_WHILE 结构

Y本身为0的可能性



```
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
        inc         cx
shift:  shl         ax, 1
        jmp         repeat
exit:   mov         count, cx
        ret
mian:   endp
prognam ends
        end         start
```



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）P179

分析题意：

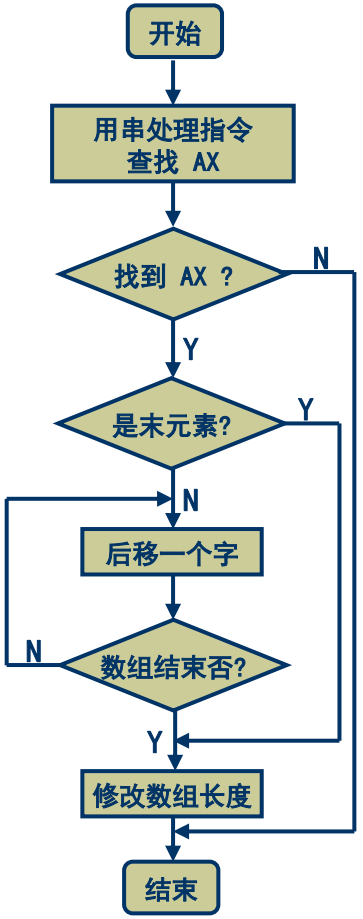
（1）如果没有找到，则不对数组作任何处理

（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

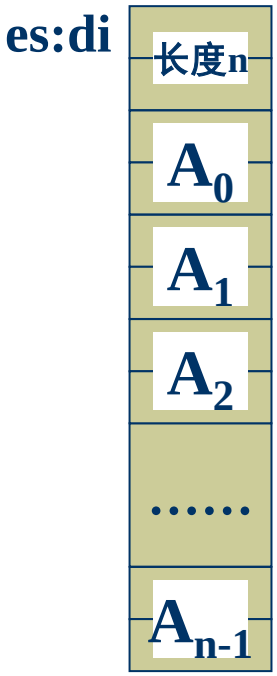
（4）查找元素用串处理指令（repnz　scasw）

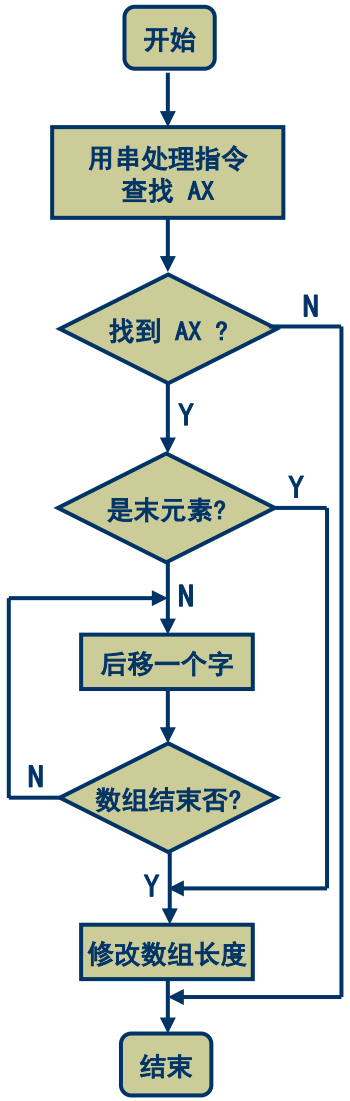
（5）删除元素用循环结构



子程序源代码

```
del_ul    proc     near
          cld
          push     di
          mov      cx, es:[di]
          add      di, 2
          repne    scasw
          je       delete
          pop      di
          jmp      short  exit
delete:   jcxz     dec_cnt     ;  如果CX=0, 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?

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

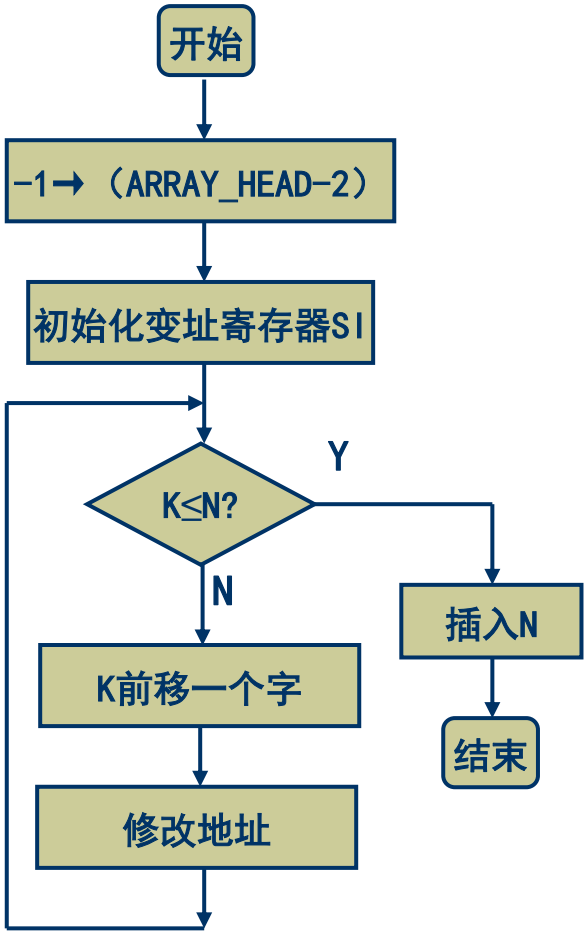
REPNE执行的操作：

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

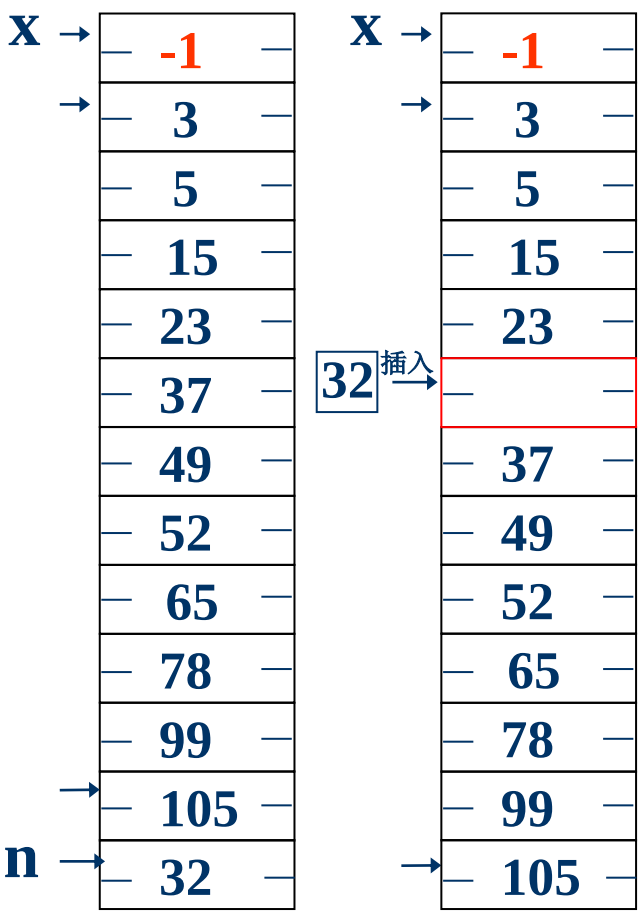
找到位置，将数据向高地址移一个字，插入N, 结束

- 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
- 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
- 循环结构的主要任务是找位置和移字数据



```
x           dw   ?
array_head  dw   3,5,15,23,37,49,52,65,78,99
array_end   dw   105
n           dw   32
```

```
        mov   ax, n
        mov   array_head-2, 0ffffh
        mov   si, 0
compare:
        cmp   array_end[si], ax
        jle   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

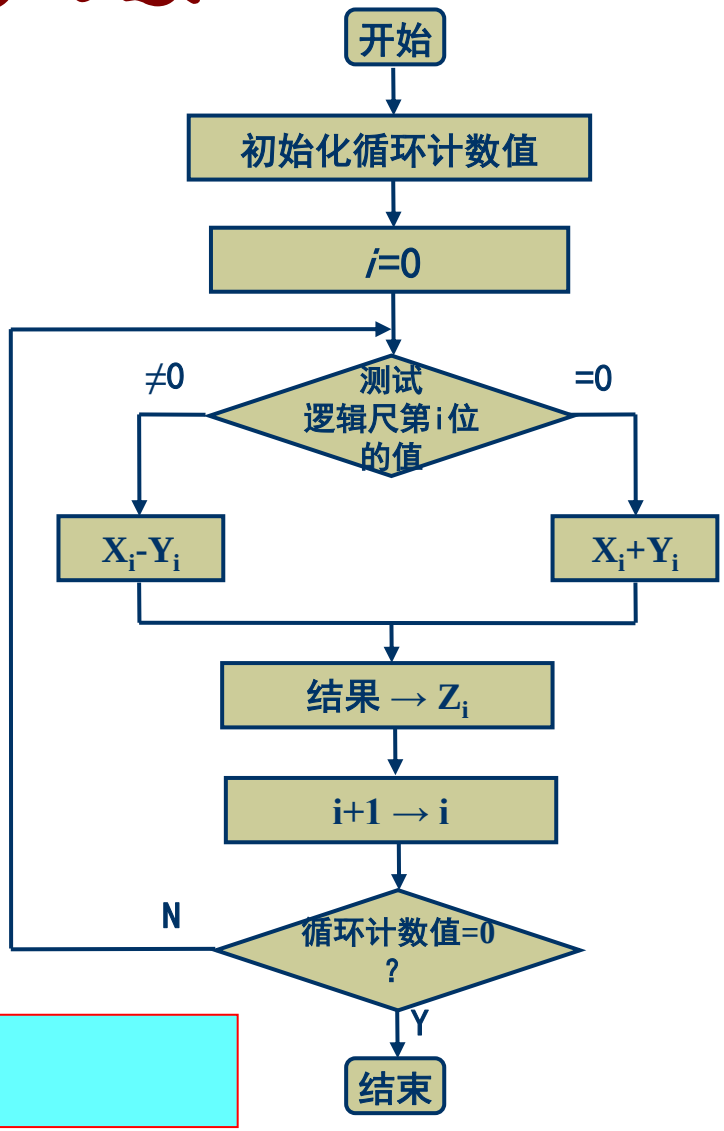
逻辑尺方法

（1）设立标志位

0000000011011100

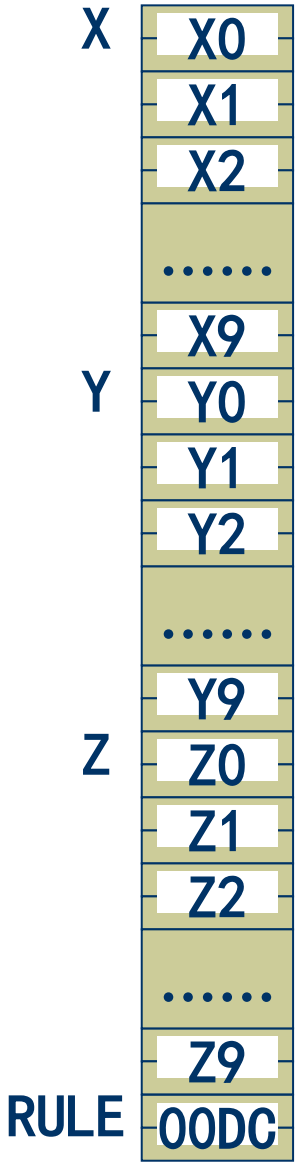
$Z_2=X_2-Y_2$　　$Z_0=X_0+Y_0$

（2）进入循环后判断标志位来确定该做的工作

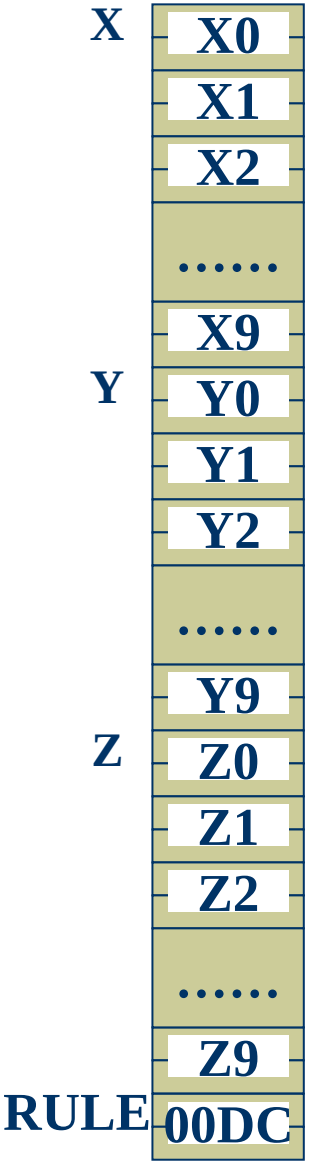


建立逻辑尺：++--+---++ 0000000011011100

```
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
```



```
CODE   SEGMENT
       ASSUME CS:CODE,DS:DATA
MAIN   PROC   FAR
       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           ;分支判断并实现转移
       ADD    AX,Y[BX]       ;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       } 返回DOS
       INT    21H            }
MAIN   ENDP
CODE   ENDS
       END    MAIN
```



- 返回DOS操作系统有两种方法：

通用方法：

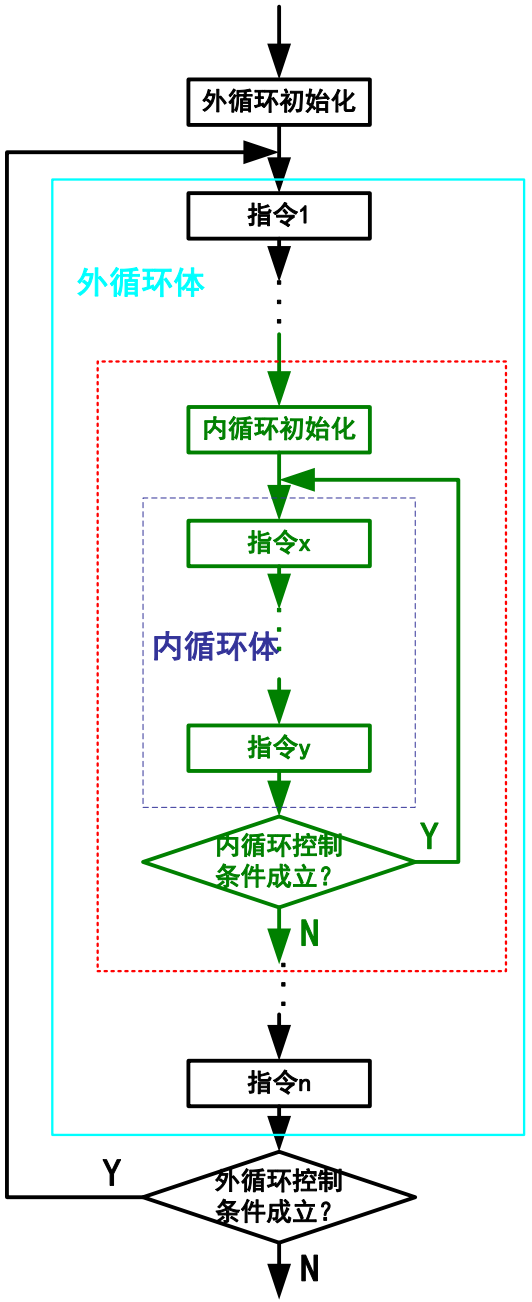
```
start:
        push  ds
        sub  ax, ax
        push  ax
        ……
        ret
main  endp
```

高级DOS版本可用方法：

```
start:
        ……
        mov  ax,  4c00h
        int  21h
main  endp
```

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



例5.7： 有一个首地址为ARRAY的N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

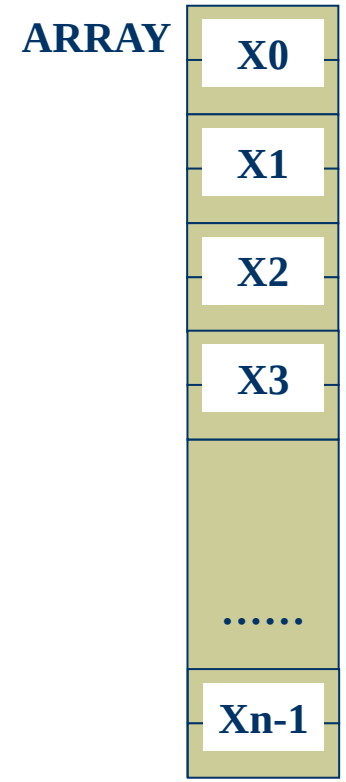
第1遍：两相邻数据比较，交换，数据按大到小排列， 比较N-1次，最后一个是最小数

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

……

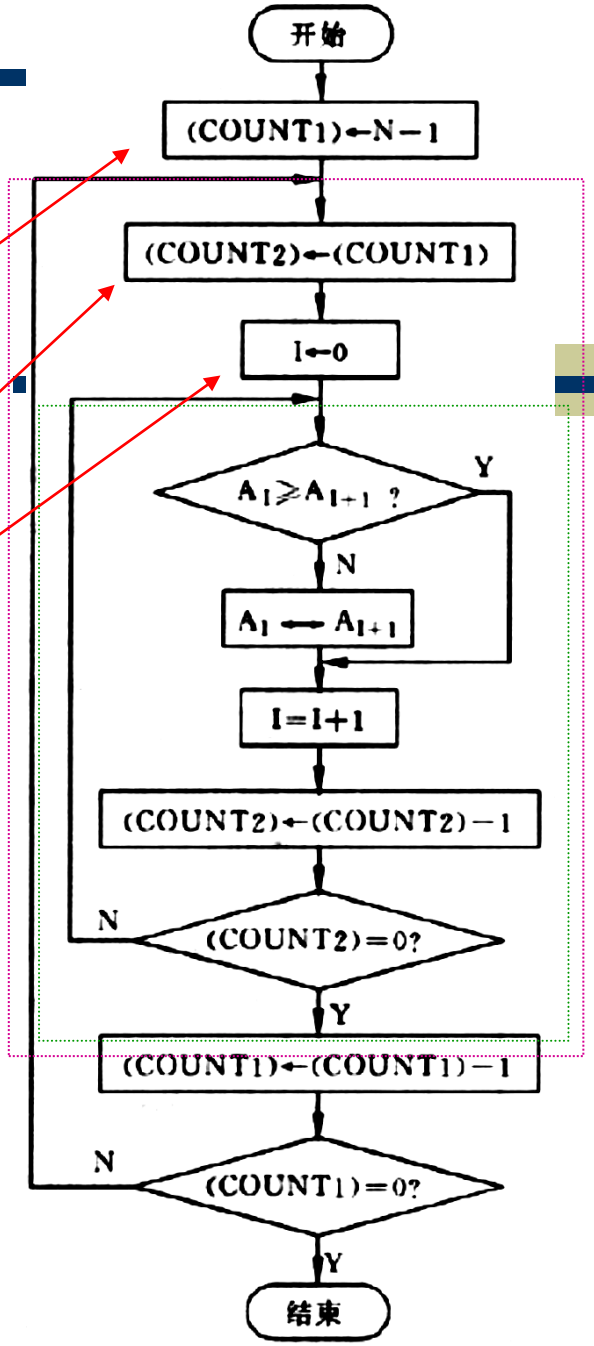
第N-1遍：两相邻数据比较，交换，数据按大到小排列，比较1次，得到结果

- 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

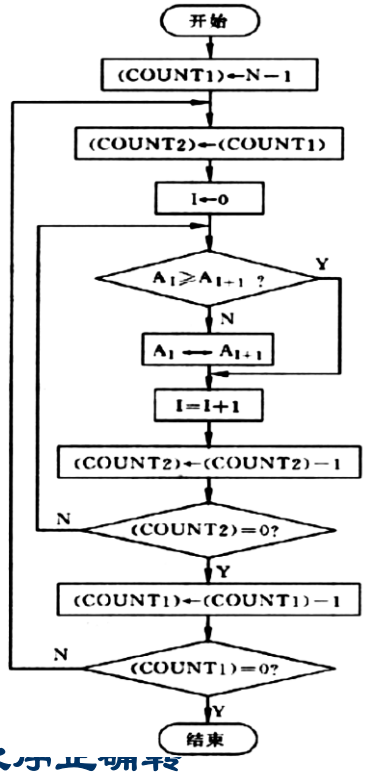


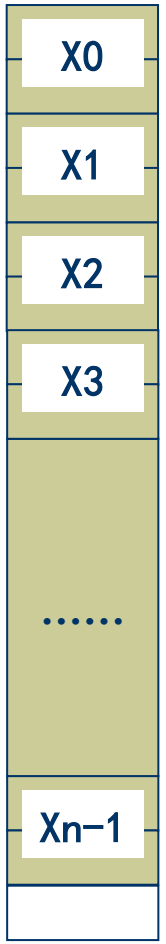
沉底由上向下比较，气泡由下向上比较

- 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



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

CODE    SEGMENT

        ASSUME CS:CODE,DS:DATA
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+2,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
```



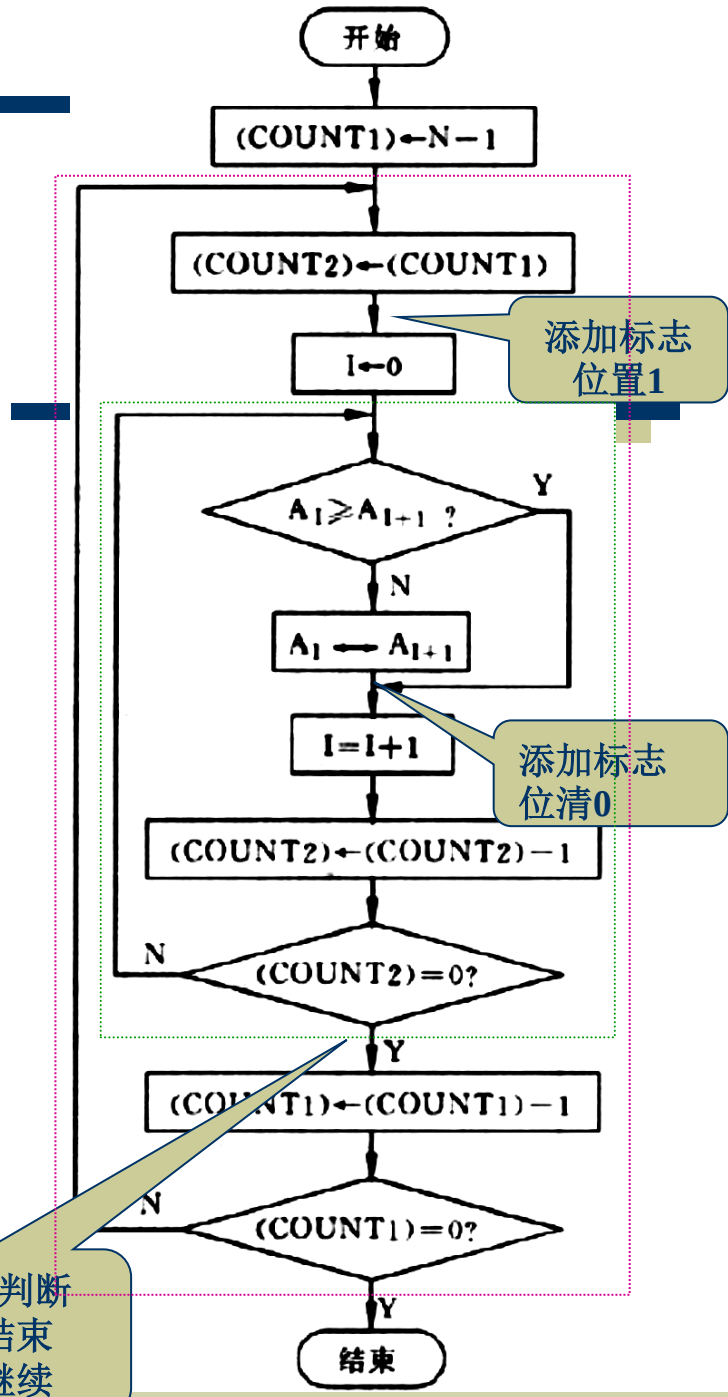


汇编程序
地址计数器值 $→

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

- 对例5.7中算法进行优化，提高程序效率(参考例5.8)
  - 提前结束，设置交换操作标志位
    - 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
    - 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



容易出错点

- 特别要注意进入循环体时，循环次数、各寄存器、标志位、存储单元的初始值一定要正确，必要时自己可以模拟执行一次
- 循环判定条件的确定
- 指针及循环控制条件的修改等

5.2 分支程序设计

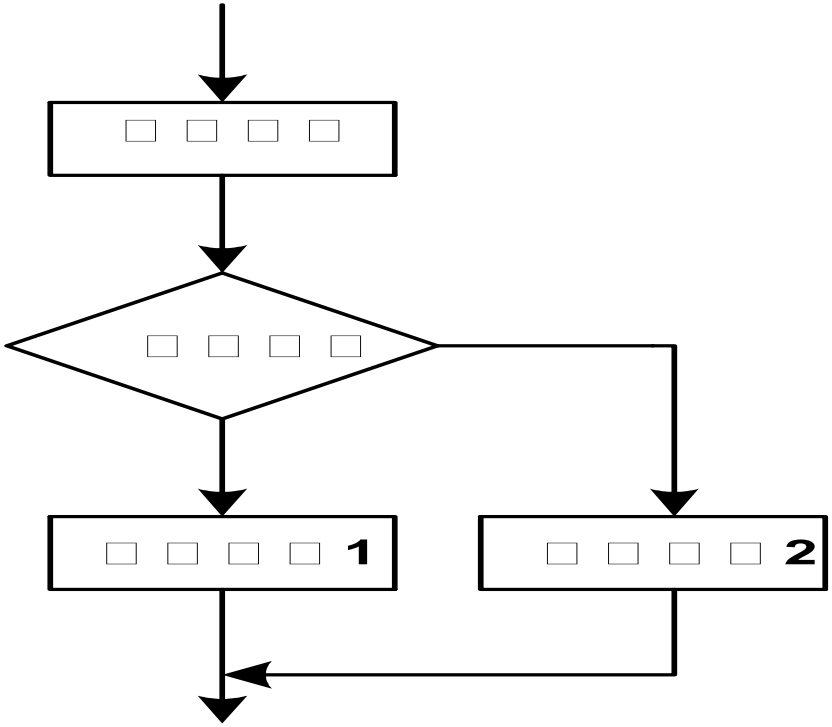
5.2.1 分支程序的结构形式

5.2.2 分支程序的设计方法

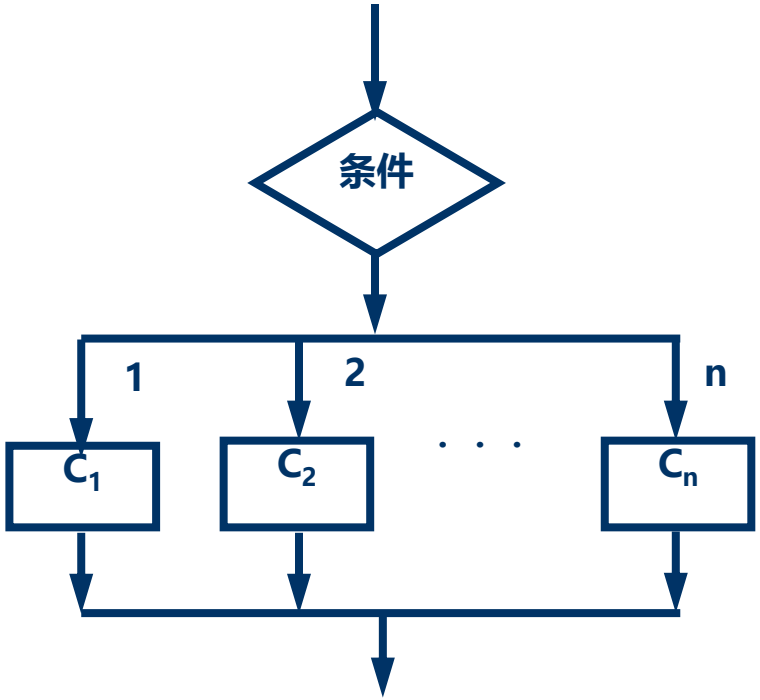
5.2.3 跳跃表法

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



IF-THEN-ELSE 结构



CASE 结构

5.2.2 分支程序的设计方法

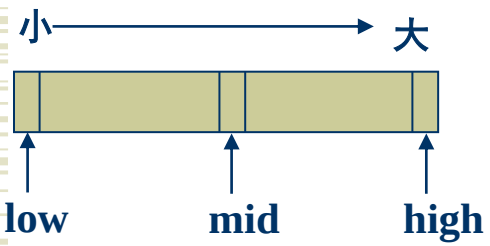
- 一般使用条件转移指令产生分支
  - 利用转移指令不影响条件码的特性，连续使用条件转移指令
- 基于逻辑尺的条件转移指令
- 跳跃表法的无条件间接寻址转移指令

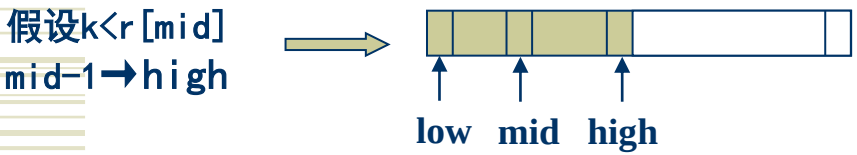
例子5.9：有一个从小到大顺序排列的无符号数数组，DI=数组首地址，数组中第一个单元存放数组长度，AX中有一个无符号数。要求在数组中查找与AX相等的数，如找到，则使CF=0，并在SI中给出该元素在数组中的相对位置；如未找到，则使CF=1，SI给出最后一个比较元素的偏移地址。

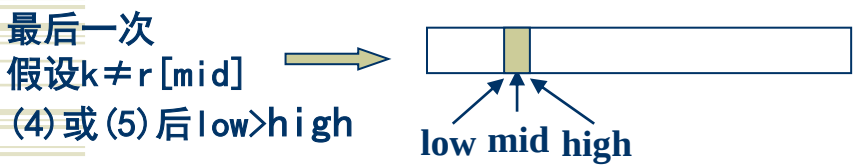
- 查找时：
  - 如果数据大小排序不定，只能用顺序查找方法；
  - 如果数据大小排序规整，也可用折半查找法，以提高查找效率

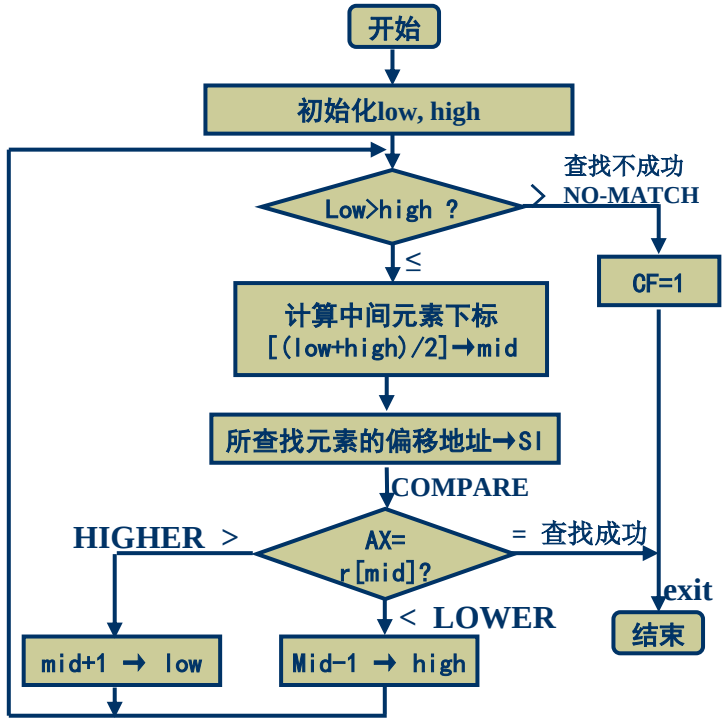
折半查找算法

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，返回步骤(2)，继续查找；
(5)高半部分查找(higher)，mid+1 → low，返回步骤(2)，继续查找；





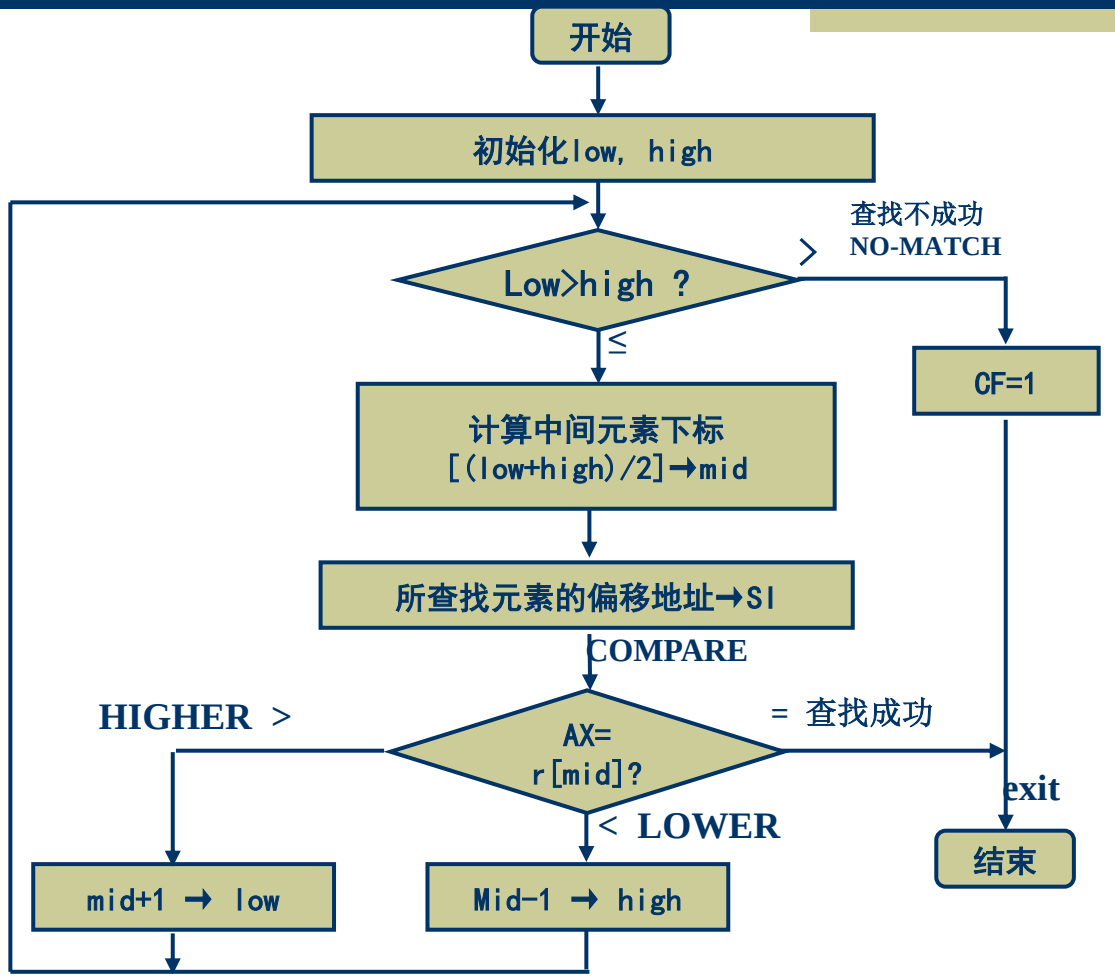




折半查找法

流程：图5.11
(参看核心程序段p194search～no_match)

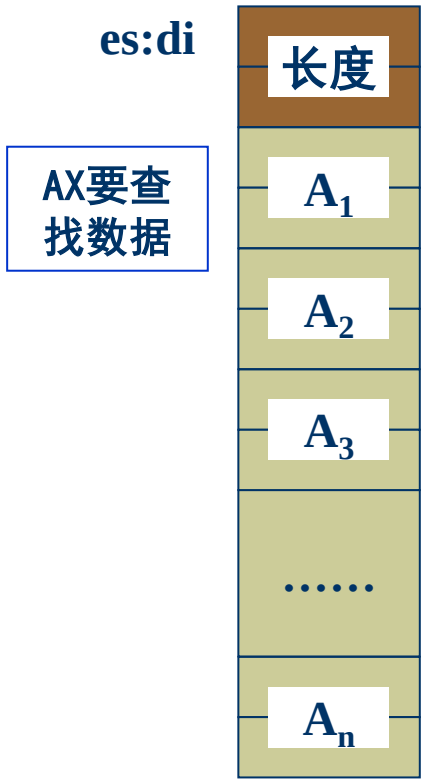
参看p192.asm



```
dseg        segment
    low_idx         dw  ?
    high_idx        dw  ?
dseg        ends
cseg        segment
b_search  proc      near
    assume  cs:cseg
    assume  ds:dseg, es:dseg
    push    ds
    push    ax
    mov     ax, dseg
    mov     ds, ax
    mov     es, ax
    pop     ax

    cmp     ax, es:[di+2]
    ja      chk_last        ;ax>A1
    mov     si, 2
    je      exit            ;ax=A1
    jmp     no_match        ;ax<A1
chk_last:
    mov     si, es:[di]
    shl     si, 1
    add     si, di
    cmp     ax, es:[si]
    jb      search          ;ax<An
    je      exit            ;ax=An
    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:
    add     si, bx      [mid]=[bx+si]
    cmp     ax, es:[si]
    je      exit
    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
    ret
b_search  endp
cseg        ends
    end
```



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
  - 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 （*置条件码类指令*）

（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：
  - 构建跳转表（分支转移目标地址表）
  - 灵活、正确使用无条件间接转移指令实现跳转

存储器单元

目标地址1
目标地址2
……
目标地址n

例5.10：根据 AL 寄存器中哪一位为 1（从低位到高位），把程序转移到 8 个不同的程序分支

```
branch_table  dw  routine1
              dw  routine2
              dw  routine3
              dw  routine4
              dw  routine5
              dw  routine6
              dw  routine7
              dw  routine8
```

（寄存器间接寻址）

```
        ……
        cmp   al, 0                          ;AL为逻辑尺
        je    continue
        lea   bx, branch_table
L:      shr   al, 1                          ;逻辑右移
        jnc   add1
        jmp   word ptr [bx]                  ;段内间接转移
add1:   add   bx, type branch_table  ;add bx,2
        jmp   L
continue:
        ……
routine1:
        ……
routine2:
        ……
```

（寄存器相对寻址）

```
        ……
        cmp   al, 0
        je    continue
        mov   si, 0
L:      shr   al, 1                    ;逻辑右移
        jnc   add1
        jmp   branch_table[si]         ;段内间接转移
add1:
        add   si, type branch_table
        jmp   L
continue:
        ……
routine1:
        ……
routine2:
        ……
```

```
        ……
        cmp   al, 0                          （基址变址寻址）
        je    continue
        lea   bx, branch_table
        mov   si, 7 * type branch_table
        mov   cx, 8
L:      shl   al, 1                          ;逻辑左移
        jnc   sub1
        jmp   word ptr [bx][si]    ;段内间接转移
sub1:   sub   si, type branch_table ;(si)-2
        loop   L
continue:
        ……
routine1:
        ……
routine2:
        ……
```

用ARM实现分支和循环

- 分支实现
- 循环实现

用ARM实现一个分支

例1.实现符号函数Y的功能。

其中：-128≤X≤+127

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

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
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5.1、试编制一个程序把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出来

Xn 1001 1111 0000 0011 显示⇒ 9F03

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

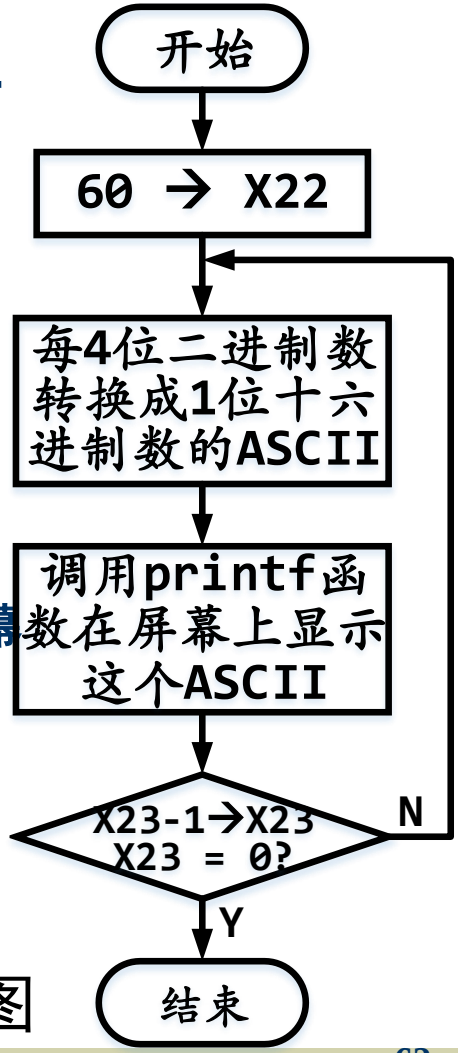
- 循环次数：每次显示1个十六进制数（送显示器相应的ASCII），循环次数=16

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
.global main

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

X29: frame pointer

X30: procedure link register

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

    subs x23, x23, 1
    bne  LOOP  // 大于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
```

用16进制字符显示寄存器中的值

- 几点说明：
  - **printf函数是libC库提供的功能**，通过汇编调用时，X0中包含的是指向字符串的地址(指针)，X1包含的是格式化字符串中要显示的第1个数据值，这里就是寄存器的低4位。
  - 函数调用时，要遵循程序调用规则中的寄存器使用
    - **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。（安全）
  - **包含C库函数时，用gcc进行程序链接比较简单**

用16进制字符显示寄存器中的值

- 几点说明：
  - 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: 输出文件，标准输出（屏幕）是为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:
    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
    adr  x1, hexfmt
    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
    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，通过ret指令返回
- 系统调用返回
  - 使用Linux系统调用exit返回
  - X8为0x5D，表示exit系统调用
  - X0保存返回代码，通过svc #0 指令返回

返回Linux系统的两种方式

```
.text
    .global  main
    .type    main, %function

main:
    stp  x29, x30, [sp, #-16]!
    ...
    ...
    ...
    mov  x0, 0
    ldp  x29, x30, [sp], #16
    ret

// 用gcc进行链接
```

函数返回

```
.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

例子5.2 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（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

思考：如果要求处理完后源数据值和进位不变，如何处理？附加条件是不能保存、恢复

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  // 再统一1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

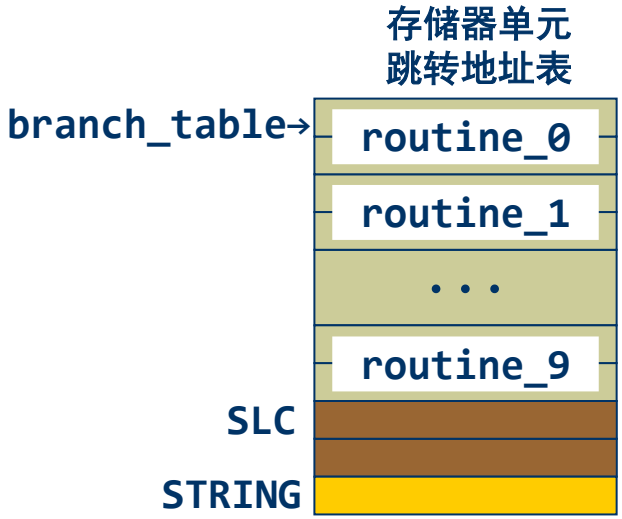
Xn.0（寄存器Xn的第0位）存放项目0的标志，Xn.1存放项目1的标志，…，Xn.9存放项目9的标志

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”
```

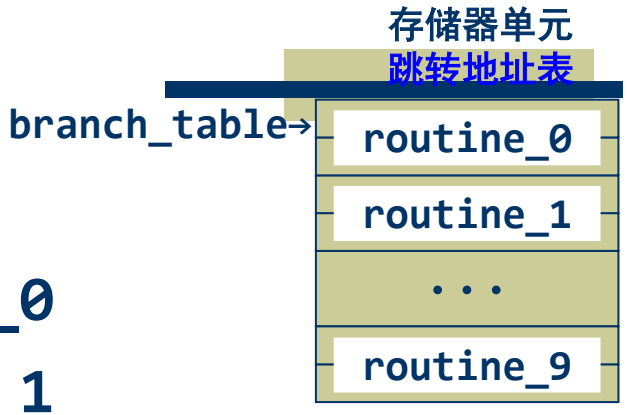
一般在程序中生成条件，条件也可放在寄存器中，这里不定义，程序中可灵活生成



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 0x2    // 选择执行的函数
        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
         b   FINISH
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
```

第5章结束语

- 如何使用循环、多重循环、分支、多分支解决实际问题