



第3章 汇编语言程序设计




3.5 循环程序设计

- ◇ 循环程序结构是在满足一定条件的情况下，重复执行某段程序。
- ◇ 循环结构的程序通常由3个部分组成：
 - ◆ 循环初始化部分：为开始循环准备必要的条件，如循环次数、循环体需要的数值等；
 - ◆ 循环体部分：由需要重复执行的程序语句构成，其中包括对循环控制变量进行修改的语句；
 - ◆ 循环控制部分：判断循环条件是否成立，决定是否继续循环。

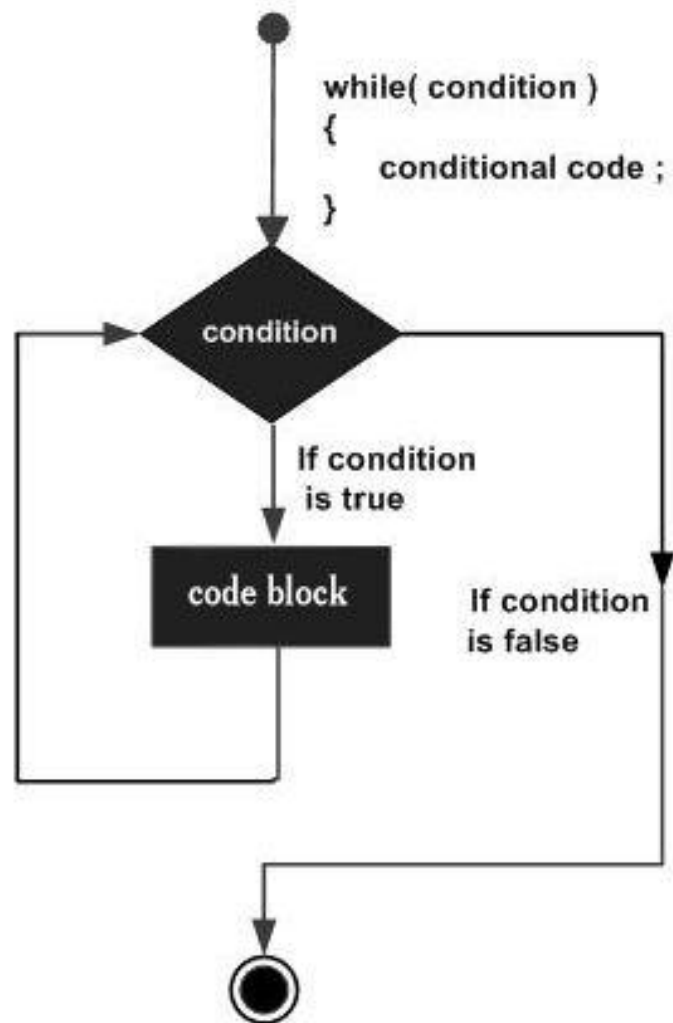
关键是什么？

循环控制



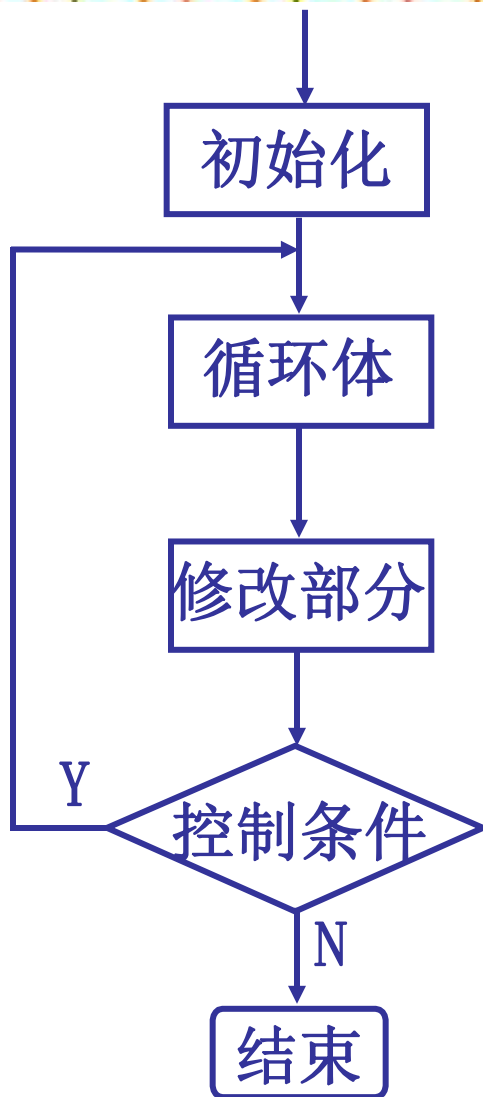
- ◇ 循环结构程序设计的关键是循环控制部分。
 - ◇ 循环控制可以在进入循环之前进行，也可以在循环体后进行，于是形成两种结构：
 - ◆ “先判断、后循环” 结构 (while—do)
 - ◆ “先循环、后判断” 结构 (do—while)
 - ◇ 循环结束的控制可以用循环次数，还可以用特定条件等，于是又有：
 - ◆ 计数控制循环
 - ◆ 条件控制循环
- 

先判断后循环的循环结构(while—do)



返回

先循环后判断的循环结构(do—while)



循环的初始状态

循环的工作部分
及修改部分

{ 计数控制循环
条件控制循环

返回

3.5.1 计数控制循环

- ◇ 计数控制循环利用循环次数作为控制条件。
- ◇ 易于采用循环指令**LOOP**和**JCXZ**实现。这种循环的程序结构如下：
 - ◆ 初始化，将循环次数或最大循环次数置入**CX**；
 - ◆ 循环体；
 - ◆ 循环控制：用**LOOP**指令对**CX**减1并判断是否为0。

3.5.1 计数控制循环--举例

例3.6 从键盘输入一个字符串，将其中小写字母转换为大写字母，并显示。

;数据段

keynum

keybuf

1B	1B	输入字符的ASCII码存在这里，255个字符以内
----	----	--------------------------

= 255

db keynum ; 键盘输入缓冲区

db 0

db keynum dup(0)

;代码段

mov dx,offset keybuf ;输入字符串

mov ah,0ah

int 21h

mov dl,0ah ;再进行换行

mov ah,2

int 21h


例3.6 大小写字母转换（2/2）

```
mov bx,offset keybuf+1 ;取出字符个数
mov cl,[bx]
mov ch,0                ;作为循环的次数
again: inc bx
      mov dl,[bx]
      cmp dl,'a'         ;小于小写字母a，不需要处理
      jb  disp
      cmp dl,'z'         ;大于小写字母z，不需要处理
      ja  disp
      sub dl,20h          ;是小写字母，则转换为大写
disp:  mov ah,2           ;显示一个字符
      int 21h
      loop again          ;循环，处理完整个字符串
```

转换原理

1B 1B 输入字符的ASCII码存在这里，255个字符以内

255 6 H e l o !



思考：如果上面这个程序执行时，用户未输入任何字符，直接按回车键会出现什么问题？



3.5.1 计数控制循环--举例

例3.8 以二进制形式显示BL中的内容

```
mov cx,8          ; CX←8 (循环次数)
again: shl bl,1     ; 左移进CF,从高位开始显示
mov dl,0          ; MOV指令不改变CF
adc dl,30h         ; DL←0 + 30H + CF
                  ; CF若是0, 则DL←' 0 '
                  ; CF若是1, 则DL←' 1 '
mov ah,2
int 21h           ; 显示
loop again
                  ; CX减1, 如果CX未减至0, 则循环
```

计数控制循环
先循环后判断

3.5.1 计数控制循环--举例

例3.9 编程求数组元素的最大值和最小值。

;数据段

array dw 10

; 假设一个数组，其中头个数据10表示元素个数

dw -3,0,20,900,587,-632,777,234,-34,-56

; 这是一个有符号字量元素组成的数组

maxay dw ? ; 存放最大值

minay dw ? ; 存放最小值

初始化：循环次数 = 元素个数 - 1

循环体：逐个比较求最大、小值

循环控制：比较完所有数据

例3.9 代码段

; 代码段

lea si,array

mov cx,[si] ; 取得元素个数

dec cx ; 减1后是循环次数

add si,2

mov ax,[si]

; 取出第一个元素给AX, AX用于暂存最大值

mov bx,ax

; 取出第一个元素给BX, BX用于暂存最小值

初始化

```
array    dw 10  
          dw -3,0,20,900,587,-632,777,234,-34,-56
```

例3.9 代码段（续）

```
maxck:    add si,2
           cmp [si],ax           ; 与下一个数据比较
           jle minck
           mov ax,[si]          ; AX取得更大的数据
           jmp next

minck:    cmp [si],bx
           jge next
           mov bx,[si]          ; BX取得更小的数据

next:     loop maxck            ; 计数循环
           mov maxay,ax         ; 保存最大值
           mov minay,bx        ; 保存最小值
```

循环体

```
array     dw 10
           dw -3,0,20,900,587,-632,777,234,-34,-56
```

3.5.1 计数控制循环--举例

例3.10 键盘输入N，显示N个“？”

	mov ah,1	;接受键盘输入
	int 21h	
	and al,0fh	;只取低4位
	xor ah,ah	
	mov cx,ax	;作为循环次数
	jcxz done	;次数为0，则结束
again:	mov dl,'?'	;循环体
	mov ah,2	
	int 21h	
	loop again	;循环控制
done:		;结束

“边界”问题

3.5.2 条件控制循环

- ◇ 条件控制循环需要利用特定条件判断循环是否结束。
- ◇ 条件控制循环用条件转移指令判断循环条件。
- ◇ 转移指令可以指定目的标号来改变程序的运行顺序，如果目的标号指向一个重复执行的语句体的开始或结束，便构成了循环控制结构。

3.5.2 条件控制循环--举例

例：显示以0结尾的字符串

```

; 数据段
string db 'Let us have a try !',0
; 代码段
mov bx,offset string
again: mov dl,[bx]
      cmp dl,0
      jz   done      ; 为0结束
      mov ah,2        ; 不为0, 显示
      int  21h
      inc  bx         ; 指向下一个字符
      jmp  again
done:  .....
```

条件控制循环
先判断后循环

3.5.2 条件控制循环--举例

例3.11 记录某个字存储单元数据中1的个数，以十进制形式显示结果。

;数据段

number dw 1110111111100100B

;代码段

mov bx,number

xor dl,dl ; 循环初值: DL←0, 1个数初值

again: test bx,0ffffh ; 也可以用cmp bx,0

jz done

; 全部是0就可以退出循环, 减少循环次数

shl bx,1 ; 用指令shr bx,1也可以


adc dl,0

; 利用ADC指令加CF的特点进行计数

jmp again

条件控制循环
先判断后循环

例3.11 十进制显示（0~16数值）



```
done:  cmp dl,10      ; 判断1的个数是否小于10
       jb  digit      ; 1的个数小于10, 转移
       push dx
       mov dl,'1'     ; 1的个数大于或等于10
       mov ah,2        ; 则要先显示一个1
       int 21h
       pop dx
       sub dl,10
digit:  add dl,'0'      ; 显示个数
       mov ah,2
       int 21h
```

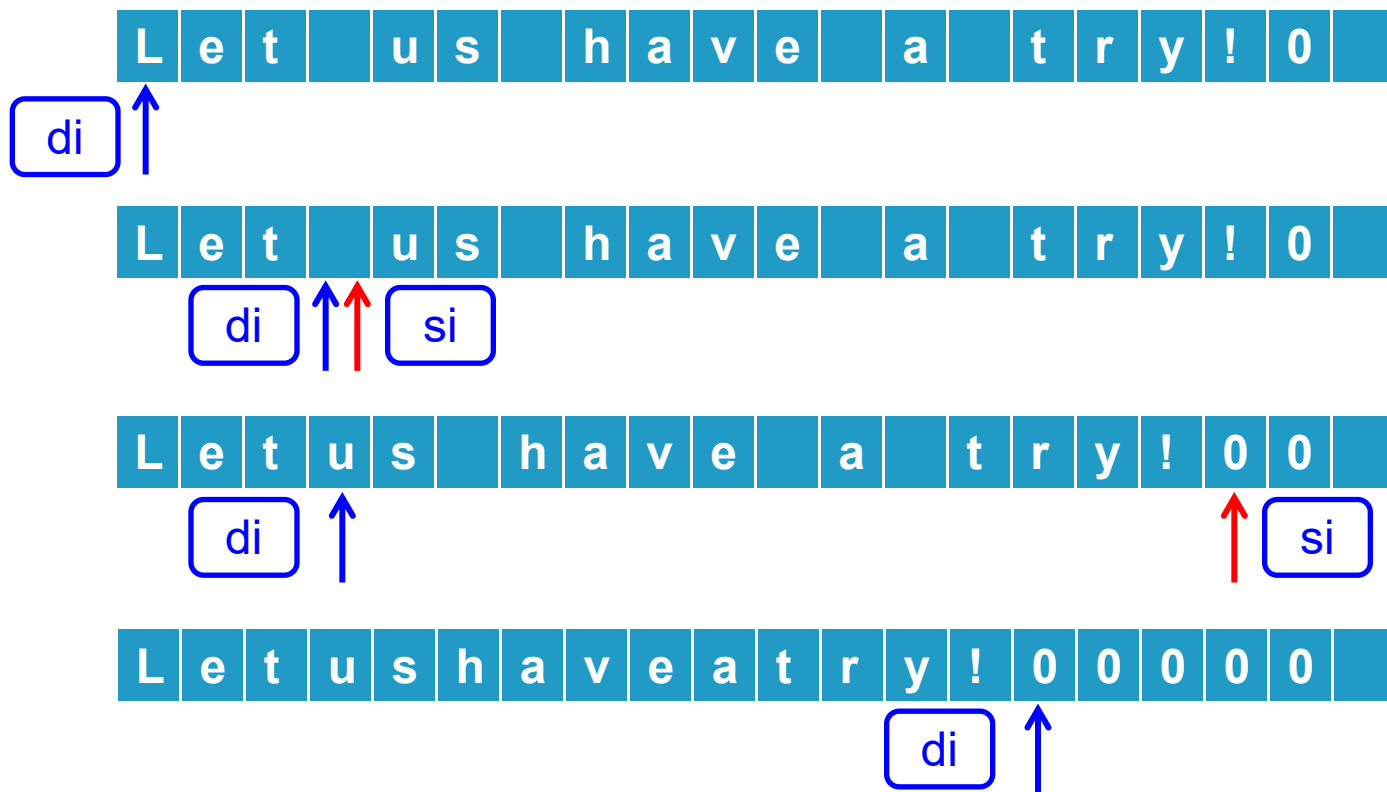
单分支结构



双重循环程序设计

例3.12 剔除字符串中的空格字符

现有一个以“0”结尾的字符串，要求剔除其中的空格字符。



例3.12 剔除字符串中的空格字符

;数据段

string db 'Let us have a try !',0 ;字符串以"0"结尾

;代码段

mov di,offset string

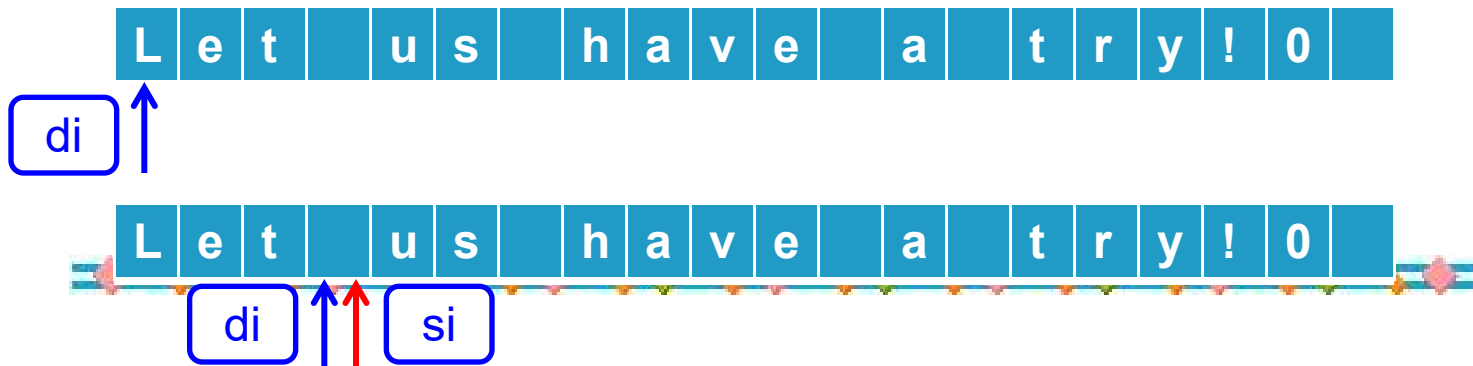
outlp: cmp byte ptr [di],0 ;外循环，先判断后循环

jz done ;为0结束

again: cmp byte ptr [di],' ' ;检测是否是空格

jnz next ;不是空格检查下一字符

mov si,di ;是空格，进入剔除空格分支



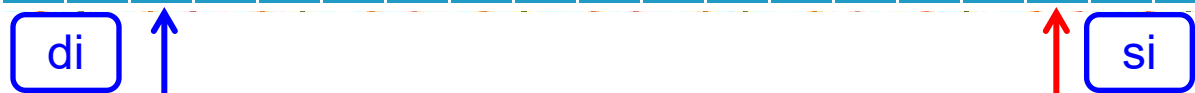
例3.12 剔除字符串中的空格字符（续）

inlp: `inc si` ;剔除空格的循环程序段
 `mov ah,[si]` ;前移一个位置
 `mov [si-1],ah`
 `cmp byte ptr [si],0` ;内循环，先循环后判断
 `jnz inlp`
 `jmp again`
next: `inc di` ;继续处理后续字符
 `jmp outlp`
done: ;结束

L e t u s h a v e a t r y ! 0



L e t u s h a v e a t r y ! 0 0



3.5.3 串操作类指令

◇8088的串操作类指令能对内存中一个连续区域的数据（如数组、字符串等）进行传送、比较等操作，指令有：

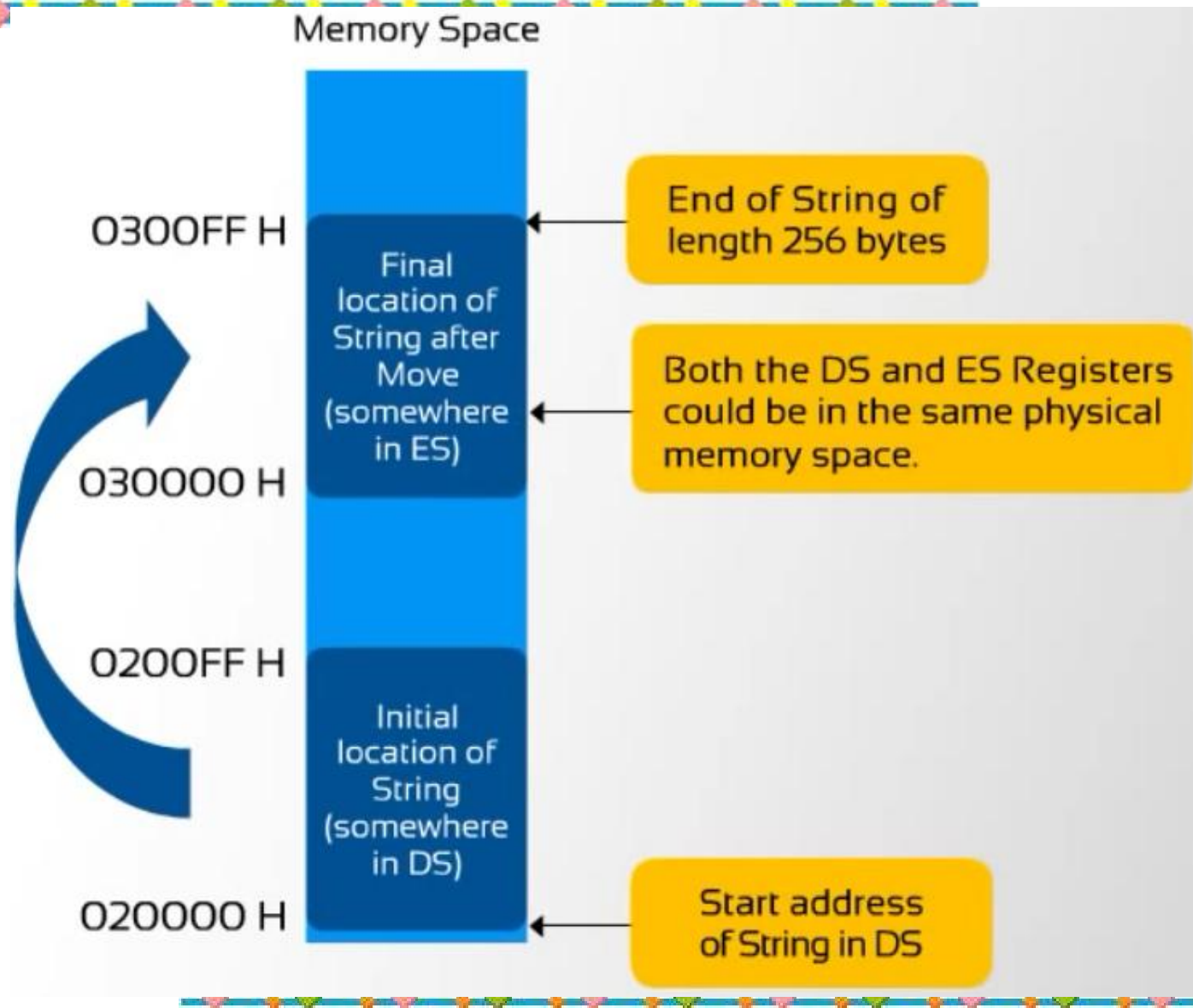
1. 传送数据串：MOVSB, STOSB, LODSB
2. 检测数据串：CMPSB, SCASB
3. 重复前缀：REP, REPZ, REPNZ

◇串操作指令采用了特殊的寻址方式

◇利用循环程序也可以实现串操作指令的功能

图示

3.5.3 串操作类指令



3.5.3 串操作类指令的寻址方式

- ◇ 源操作数默认在数据段中，用寄存器**SI**间接寻址，即由**DS:SI**给出其内存地址，允许段超越；
- ◇ 目的操作数默认在附加段中，用寄存器**DI**间接寻址，即由**ES:DI**给出其内存地址，不允许段超越；
- ◇ 每执行一次串操作，源地址指针**SI**和目的地址指针**DI**将自动修改： ± 1 或 ± 2
 - ◆ 对于以字节为单位的数据串（指令助记符用**B**结尾）操作，地址指针应该 ± 1
 - ◆ 对于以字为单位的数据串（指令助记符用**W**结尾）操作，地址指针应该 ± 2
 - ◆ 当**DF=0**（执行**CLD**指令），地址指针应该 $+1$ 或 $+2$
 - ◆ 当**DF=1**（执行**STD**指令），地址指针应该 -1 或 -2

1. 传送数据串

◇ 串传送指令

MOVS **B** ;ES:[DI]←DS:[SI]; SI←SI ± 1, DI←DI ± 1

MOV **S** **W** ;ES:[DI]←DS:[SI]; SI←SI ± 2, DI←DI ± 2

◇ 串存储指令

STO **S** **B** ;ES:[DI]←AL; 然后: DI←DI ± 1

STO **S** **W** ;ES:[DI]←AX; 然后: DI←DI ± 2

◇ 串读取指令

LOD **S** **B** ;AL←DS:[SI]; SI←SI ± 1

LOD **S** **W** ;AX←DS:[SI]; SI←SI ± 2

◇ 重复前缀指令

REP ;执行一次串指令, CX减1; 直到CX=0

数据块传送MOVS

mov cx,400h

; 设置循环次数

mov si,offset sbuf

; SI指向源缓冲区

mov di,offset dbuf

; DI指向目的缓冲区

cld

rep movsb

again: mov al,[si]
 mov es:[di],al
 inc si
 inc di
 loop again

数据块存储

数据块存储指令的典型应用是初始化某一缓冲区。
例：将附加段全部**64KB**初始化为**0**，可用如下代码实现：

```
mov di,0  
mov ax,0  
mov cx,8000h    ;CX←传送次数 (32 × 1024)  
rep stosw       ;重复字传送：ES:[DI]←0
```

3.5.3 串操作类指令

例3.13 利用串操作指令编程，挑出数组中的正数（不含0）和负数，分别形成正数数组和负数数组。

假设数组**array**具有**count**个字数据，正数数组为**ayplus**，负数数组为**ayminus**，它们都在数据段中。

Wj0313.asm



si  → AX lodsw



di  ← AX stosw



bx 



例3.13 挑出数组中的正数（不含0）和负数

;数据段

```
count      equ 10
array      dw 23h,9801h,8000h, ...
ayplus     dw count dup(0)
ayminus    dw count dup(0)
```

;代码段

```
mov si,offset array
mov di,offset ayplus
mov bx,offset ayminus
mov ax,ds
mov es,ax      ;ES = DS
mov cx,count   ;CX←字节数
cld
```

例3.13 挑出数组中的正数（不含0）和负数（续）



```
again:      lodsw                ;从array取出一个数据
            cmp ax,0            ;判断是正是负
            jl minus           ;小于0，转向minus
            jz next            ;等于0，继续下一个数据
            stosw               ;大于0，存入ayplus
            jmp next

minus:      xchg bx,di
            stosw               ;把负数存入ayminus
            xchg bx,di

next:      loop again           ;继续进行
```



2. 检测数据串

◇ 串比较指令

CMPSB ;DS:[SI] – ES:[DI]; SI←SI ± 1, DI←DI ± 1

CMPSW ;DS:[SI] – ES:[DI]; SI←SI ± 2, DI←DI ± 2

◇ 串扫描指令

SCASB ;AL – ES:[DI]; DI←DI ± 1

SCASW ;AX – ES:[DI]; DI←DI ± 2

◇ 重复前缀指令

REPE|REPZ ;执行一次串指令，CX减1；只要CX = 0
或ZF = 0，重复执行结束

◇ 重复前缀指令

REPNE|REPNE ;执行一次串指令，CX减1；只要CX = 0
或ZF = 1，重复执行结束



3.5.3 串操作类指令

例3.14 比较两个等长的字符串

假设一个字符串**string1**在数据段，另一个字符串**string2**在附加段，都具有**count**个字符，比较的结果存入**result**单元，相等用**0**表示，不等用**-1**表示。

用不带前缀的串操作指令和带前缀的串操作指令分别编程。

例3.14 比较两个等长的字符串（不使用重复前缀）



```
mov si,offset string1
mov di,offset string2
mov cx,count
cld
again:  cmpsb           ;比较两个字符
        jnz unmat      ;出现不同的字符，转移
        loop again     ;进行下一个字符的比较
        mov al,0       ;字符串相等
        jmp output     ;转向output
unmat:  mov al,-1
output: mov result,al   ;输出结果标记
```

Wj0314.asm

例3.14 比较两个等长的字符串（使用重复前缀）

Wj0314d.asm

```
mov si,offset string1
mov di,offset string2
mov cx,count
cld
```

```
repz cmpsb
```

;比较两个字符

```
jnz unmat
```

;出现不同的字符，转移

```
mov al,0
```

;字符串相等

```
jmp output
```

;转向output


unmat:

```
mov al,-1
```

output:

```
mov result,al
```

;输出结果标记



循环结构程序设计部分结束
谢谢！




表3.1 存储模式

存储模式	特点
TINY (微型模式)	COM 类型程序，只有一个小于 64KB 的逻辑段 (MASM 6.x 支持)
SMALL (小型模式)	小应用程序，只有一个代码段和一个数据段 (含堆栈段)，每段不大于 64KB
COMPACT (紧凑模式)	代码少、数据多的程序，只有一个代码段，但有多多个数据段
MEDIUM (中型模式)	代码多、数据少的程序，可有多多个代码段，只有一个数据段
LARGE (大型模式)	大应用程序，可有多多个代码段和多个数据段 (静态数据小于 64KB)
HUGE (巨型模式)	更大应用程序，可有多多个代码段和多个数据段 (对静态数据没有限制)
FLAT (平展模式)	32位 应用程序，运行在 32位80x86CPU 和 Windows 9x 或 NT 环境



XLAT指令的功能

XLAT

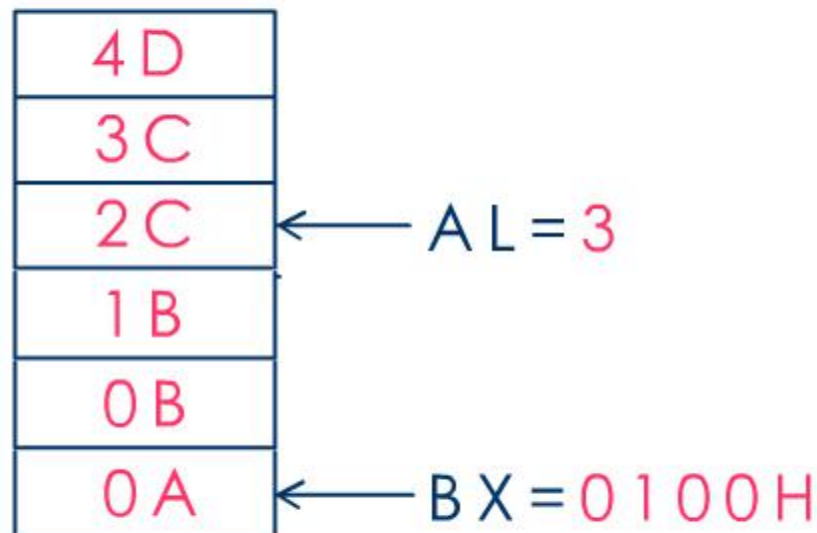
执行前：

AL

BX

执行后：

AL



大小写字母的比较和转换

'A' = 41H = 01000001B	'B' = 42H	... 'Z' = 5AH = 01011001B
'a' = 61H = 01100001B	'b' = 62H	... 'z' = 7AH = 01111001B

结论1: 大小写字母的**ASCII**码值相差**20H**

结论2: 大小写字母的**ASCII**码值仅**D5**位不同

方法1（加减指令）：“**ADD DL,20H**” “**SUB DL,20H**”

方法2（逻辑指令）：“**OR DL,20H**” “**AND DL,0DFH**”

大小写互换（异或指令）：“**XOR DL,20H**”



十进制	二进制	符号	十进制	二进制	符号	十进制	二进制	符号	十进制	二进制	符号
0	0000 0000	NUL	32	0010 0000	[空格]	64	0100 0000	@	96	0110 0000	`
1	0000 0001	SOH	33	0010 0001	!	65	0100 0001	A	97	0110 0001	a
2	0000 0010	STX	34	0010 0010	"	66	0100 0010	B	98	0110 0010	b
3	0000 0011	ETX	35	0010 0011	#	67	0100 0011	C	99	0110 0011	c
4	0000 0100	EOT	36	0010 0100	\$	68	0100 0100	D	100	0110 0100	d
5	0000 0101	ENQ	37	0010 0101	%	69	0100 0101	E	101	0110 0101	e
6	0000 0110	ACK	38	0010 0110	&	70	0100 0110	F	102	0110 0110	f
7	0000 0111	BEL	39	0010 0111	'	71	0100 0111	G	103	0110 0111	g
8	0000 1000	BS	40	0010 1000	(72	0100 1000	H	104	0110 1000	h
9	0000 1001	HT	41	0010 1001)	73	0100 1001	I	105	0110 1001	i
10	0000 1010	LF	42	0010 1010	*	74	0100 1010	J	106	0110 1010	j
11	0000 1011	VT	43	0010 1011	+	75	0100 1011	K	107	0110 1011	k
12	0000 1100	FF	44	0010 1100	,	76	0100 1100	L	108	0110 1100	l
13	0000 1101	CR	45	0010 1101	-	77	0100 1101	M	109	0110 1101	m
14	0000 1110	SO	46	0010 1110	.	78	0100 1110	N	110	0110 1110	n
15	0000 1111	SI	47	0010 1111	/	79	0100 1111	O	111	0110 1111	o
16	0001 0000	DLE	48	0011 0000	0	80	0101 0000	P	112	0111 0000	p
17	0001 0001	DC1	49	0011 0001	1	81	0101 0001	Q	113	0111 0001	q
18	0001 0010	DC2	50	0011 0010	2	82	0101 0010	R	114	0111 0010	r
19	0001 0011	DC3	51	0011 0011	3	83	0101 0011	S	115	0111 0011	s
20	0001 0100	DC4	52	0011 0100	4	84	0101 0100	T	116	0111 0100	t
21	0001 0101	NAK	53	0011 0101	5	85	0101 0101	U	117	0111 0101	u
22	0001 0110	SYN	54	0011 0110	6	86	0101 0110	V	118	0111 0110	v
23	0001 0111	ETB	55	0011 0111	7	87	0101 0111	W	119	0111 0111	w
24	0001 1000	CAN	56	0011 1000	8	88	0101 1000	X	120	0111 1000	x
25	0001 1001	EM	57	0011 1001	9	89	0101 1001	Y	121	0111 1001	y
26	0001 1010	SUB	58	0011 1010	:	90	0101 1010	Z	122	0111 1010	z
27	0001 1011	ESC	59	0011 1011	;	91	0101 1011	[123	0111 1011	{
28	0001 1100	FS	60	0011 1100	<	92	0101 1100	\	124	0111 1100	
29	0001 1101	GS	61	0011 1101	=	93	0101 1101]	125	0111 1101	}
30	0001 1110	RS	62	0011 1110	>	94	0101 1110	^	126	0111 1110	~
31	0001 1111	US	63	0011 1111	?	95	0101 1111	_	127	0111 1111	DEL

AX

