

数据传送、寻址和算术运算

数据传送指令

操作数类型

- 立即操作数imm
- 寄存器操作数reg
- 内存操作数mem

表 4.1 指令中的操作数表示法

操作数	描 述
r8	8 位通用寄存器: AH, AL, BH, BL, CH, CL, DH, DL
r16	16 位通用寄存器: AX, BX, CX, DX, SI, DI, SP, BP
r32	32 位通用寄存器: EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP
reg	任意的通用寄存器
sreg	16 位段寄存器: CS, DS, SS, ES, FS, GS
imm	8 位、16 位或 32 位立即数
imm8	8 位立即数(字节)
imm16	16 位立即数(字)
imm32	32 位立即数(双字)
r/m8	8 位操作数(可以是 8 位通用寄存器或内存字节)
r/m16	16 位操作数(可以是 16 位通用寄存器或内存字)
r/m32	32 位操作数(可以是 32 位通用寄存器或内存双字)
mem	8 位、16 位或 32 位内存操作数

直接内存操作数

变量名仅仅是对数据段内偏移地址的引用。

指令使用内存操作数实际上使用的是内存操作数的地址。

内存寻址操作

内存操作的最小寻址单位为字节。

- masm32使用方括号表示内存寻址操作, `mov eax, [var1]`
- 通常, 直接内存操作数不使用中括号, `mov eax, var1`
- 涉及到算术表达式时, 使用中括号, `mov eax, [var1+5]`

MOV指令

第一个操作数是目的操作数，第二个操作数是源操作数

注意：

- 两个操作数尺寸必须一致
- 两个操作数不能同时为内存操作数
- 目的操作数不能是CS,EIP,IP
- 立即数不能直接送到段寄存器

内存之间移动

在送至目的操作数之前可以先把源操作数移入一个寄存器中

```
.data
var1 word ?
var2 word ?
.code
mov ax,var1
mov var2,ax
```

LABEL指令

LABEL 常见的用法是，为数据段中定义的下一个变量提供不同的名称和大小属性。如下例所示，在变量 val32 前定义了一个变量，名称为 val16 属性为 WORD：

```
.data
val16 LABEL WORD
val32 DWORD 12345678h
.code
mov ax,val16 ; AX = 5678h
mov dx,[val16+2] ; DX = 1234h
```

val16 与 val32 共享同一个内存位置。LABEL 伪指令自身不分配内存。

有时需要用两个较小的整数组成一个较大的整数，如下例所示，两个 16 位变量组成一个 32 位变量并加载到 EAX 中：

```
.data
LongValue LABEL DWORD
val1 WORD 5678h
val2 WORD 1234h
.code
mov eax,LongValue ; EAX = 12345678h
```

整数的零/符号拓展

把字count（16位无符号）送至eax（32位）中，先把eax清零，然后将count送到ax中

```
count word 1
mov eax,0
mov ax,count
```

MOVZX指令

零扩展传送，仅适用于无符号整数。将源操作数内容复制到目的操作数，并将该值零扩展到16/32位。

```
mov bl,10001111b
movzx ax,bl
mov bx,0A69Bh
movzx eax,bx
movzx edx,bl
movzx cx,bl
```

```
.data
dw_var label word
dd_var dd 12345678h
.code
movzx eax,dw_var
;eax值为00005678h
```

MOVSX指令

符号扩展传送，仅适用于有符号整数。将源操作数内容复制到目的操作数，并将该值零扩展到16/32位。

```
mov bl,10001111b
movsx ax,bl
```

例子

```
.data
var1 byte 90h
var2 byte 31h
eaxnum dword ?
ebxnum dword ?
eaxnum2 dword ?
ebxnum2 dword ?
.code
start:
movzx eax,var1
movsx ebx,var1
mov eaxnum,eax
mov ebxnum,ebx
movzx eax,var2
```

```

movsx ebx,var2
mov eaxnum2,eax
mov ebxnum2,ebx
invoke StdOut,addr eaxnum
end start

```

Address	Hex dump
00403000	90 31 90 00 00 00 90 FF
00403008	FF FF 31 00 00 00 31 00
00403010	00 00 00 00 00 00 00 00

LAHF和SAHF指令

LAHF指令将EFLAGS寄存器的低字节（8个字节）复制到AH寄存器，可以将标志值保存在变量中。被复制的标志包括：符号标志、零标志、辅助进位标志、奇偶标志、进位标志。

SAHF指令复制AH寄存器的值至EFLAGS寄存器的低字节。

```

.data
saveflags byte ?
.code
lahf ;状态标志送ah
mov saveflags,ah ; 状态标志存变量
mov ah,saveflags
sahf

```

XCHG指令

交换两个操作数的内容，不接受立即数操作数，不能交换两个内存操作数

若交换两个内存操作数

```

mov ax,val1
xchg ax,val2
mov val1,ax

```

直接偏移操作数

访问没有显式标号的内存

```

arrayb byte 10h,20h,30h,40h
mov al,arrayb
mov al,[arrayb+1]
arrayd dword 1000h,2000h
mov eax arrayd+1

```

```

.data
    var1 DWORD 1000h, 2000h, 3000h, 4000h
.code
start:
    mov eax, var1
    mov eax, [var1+1]
    mov eax, [var1+2]
    invoke ExitProcess, 0

```

窗口 - 主线程, 模块 hello

00	A1 00304000	MOV EAX, DWORD PTR DS:[403000]
05	A1 01304000	MOV EAX, DWORD PTR DS:[403001]
0A	A1 02304000	MOV EAX, DWORD PTR DS:[403002]
0F	6A 00	PUSH 0
11	E8 00000000	CALL <JMP.&kernel32.ExitProcess>

地址	十六进制数据	多
00403000	00 10 00 00 00 20 00 00 00 30 00 00 00 40 00 00	
00403010	00 20 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00403020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	
00403030	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	

加法和减法

inc和dec指令

- inc 加一
- dec 减一
- 可跟reg和mem

add和sub指令

add/sub 目的操作数，源操作数
相加/相减的结果储存在目的操作数中

neg指令

通过将数字转换成对应的补码而求其相反数。
可跟reg和mem.

```

.data
    var1 dd 1000h
.code
start:
    neg var1
end start

```

Address	Hex dump	ASCII
00403000	00 F0 FF FF 00 00 00 00	??
00403008	00 00 00 00 00 00 00 00

加法和减法影响的标志

- 进位标志CF用于表示**无符号**整数运算是否发生了溢出：若结果超出目的操作数的大小，则置位
- 溢出标志OF用于表示**有符号**整数运算是否发生了溢出：若结果小于十进制数-32768，则置位
- 零标志ZF用于表示运算结果是否为0
- 符号标志SF用于表示运算结果是否为负
- 奇偶标志PE用于表示目的操作数的最低有效字节内1的个数是否为偶数
- 辅助进位标志AC在运算结果的最低有效字节的第三位向高位产生进位1时置位

进位标志

```
mov ax,00FFh
add ax,1
;AX=0100h,CF=0
mov ax,0FFFFh
add ax,1
;AX=0000h,CF=1
mov al,1
sub al,2
;al=FFh,CF=1
```

inc和dec指令不影响进位标志，对非0操作数执行neg操作总是设置进位标志

辅助进位标志

```
mov al,0Fh
add al,1
;AC=1
```

符号标志

```
mov al,1
sub al,2
;bl=FFh,SF=1
```

溢出标志

目的操作数无法容纳

```
mov al,+127
add al,1
;OF=1
mov al,-128
sub al,1
;OF=1
```

发生以下情况说明溢出：两正数相加为负数；两负数相加为正数

乘法和除法（无符号）

除法指令div

如果参数是r8/m8,把AX做被除数;商->AL,余数->AH

如果参数是r16/m16,把DX:AX做被除数;商->AX,余数->DX

如果参数是r32/m32,把EDX:EAX做被除数;商->EAX,余数->EDX

乘法指令mul

如果参数是 r8/m8, 把AL做乘数, 结果放在AX

如果参数是 r16/m16, 把AX做乘数, 结果放在EAX

如果参数是 r32/m32, 把EAX做乘数, 结果放在EDX:EAX

结果存储到寄存器 AX、寄存器对 DX:AX 或 EDX:EAX（取决于操作数大小），乘积的高位分别保存在寄存器 AH、DX 或 EDX。如果乘积的高位均为 0，则清除 CF 与 OF 标志；否则将它们设置为 1

逻辑移位shr与shl指令

shl是逻辑左移指令，它的功能为：

- 将一个寄存器或内存单元中的数据向左移位；
- 将最后移出的一位写入CF中；
- 最低位用0补充。

指令：

```
mov al,01001000b
shl al,1 ;将al中数据左移一位
;执行后 (al)=10010000b, CF=0
```

shr（右移）指令使目的操作数逻辑右移一位，最高位用 0 填充。在多位移操作中，最后一个移出位 0（LSB）的数值进入进位标志位。

数值移位

SAL	算术左移 (Shift Arithmetic Left)							
	<div> <div>O</div> <div>D</div> <div>I</div> <div>S</div> <div>Z</div> <div>A</div> <div>P</div> <div>C</div> </div> <div> <div>*</div> <div></div> <div></div> <div>*</div> <div>*</div> <div>?</div> <div>*</div> <div>*</div> </div>							
	<p>目的操作数中的每一位左移，源操作数决定移位数目。最高位复制到进位标志中，最低位以0填充。使用 8086/8088 处理器时，imm8 操作数必须是 1。</p> <p>指令格式：</p> <div> <div>SAL reg, imm8</div> <div>SAL mem, imm8</div> <div>SAL reg, CL</div> <div>SAL mem, CL</div> </div>							

SAR	算术右移 (Shift Arithmetic Right)							
	<div> <div>O</div> <div>D</div> <div>I</div> <div>S</div> <div>Z</div> <div>A</div> <div>P</div> <div>C</div> </div> <div> <div>*</div> <div></div> <div></div> <div>*</div> <div>*</div> <div>?</div> <div>*</div> <div>*</div> </div>							
	<p>目的操作数中的每一位右移，源操作数决定移位数目。最低位复制到进位标志中，最高位保持原值。SAR 指令通常用于有符号数操作，因为它保留了符号位的值。使用 8086/8088 处理器时，imm8 操作数必须是 1。</p> <p>指令格式：</p> <div> <div>SAR reg, imm8</div> <div>SAR mem, imm8</div> <div>SAR reg, CL</div> <div>SAR mem, CL</div> </div>							

shr是逻辑右移指令，它和shl所进行的操作刚好相反。

和数据相关的操作符与伪指令

offset操作符

返回数据标号的偏移地址（4字节）

```
.data
myarray word 1,2,3,4,5
.code
mov esi,offset myarray+4
;获取myarray的地址，然后加4并把其和所表示的地址处的内容送esi寄存器
```

```
.data
str_hello db "hello world",0
var dd 1000h
.code
start:
mov eax,offset str_hello
end start
```


CPU - main thread, module test

00402002	6C	INS BYTE PTR ES:[EDI],DX	I/O command
00402003	6C	INS BYTE PTR ES:[EDI],DX	I/O command
00402004	6F	OUTS DX,DWORD PTR ES:[EDI]	I/O command
00402005	2077 6F	RND BYTE PTR DS:[EDI+6F],DH	I/O command
00402008	72 6C	JB SHORT test.00402076	
0040200A	64:0000	ADD BYTE PTR FS:[EAX],AL	
0040200D	1000	ADC BYTE PTR DS:[EAX],AL	
0040200F	0000	ADD BYTE PTR DS:[EAX],AL	
00402011	0000	ADD BYTE PTR DS:[EAX],AL	
00402013	0000	ADD BYTE PTR DS:[EAX],AL	
00402015	0000	ADD BYTE PTR DS:[EAX],AL	
00402017	0000	ADD BYTE PTR DS:[EAX],AL	
00402019	0000	ADD BYTE PTR DS:[EAX],AL	
0040201B	0000	ADD BYTE PTR DS:[EAX],AL	
0040201D	0000	ADD BYTE PTR DS:[EAX],AL	
0040201F	0000	ADD BYTE PTR DS:[EAX],AL	
00402021	0000	ADD BYTE PTR DS:[EAX],AL	
00402023	0000	ADD BYTE PTR DS:[EAX],AL	
00402025	0000	ADD BYTE PTR DS:[EAX],AL	
00402027	0000	ADD BYTE PTR DS:[EAX],AL	
00402029	0000	ADD BYTE PTR DS:[EAX],AL	
0040202B	0000	ADD BYTE PTR DS:[EAX],AL	
0040202D	0000	ADD BYTE PTR DS:[EAX],AL	
0040202F	0000	ADD BYTE PTR DS:[EAX],AL	
00402031	0000	ADD BYTE PTR DS:[EAX],AL	
00402033	0000	ADD BYTE PTR DS:[EAX],AL	
00402035	0000	ADD BYTE PTR DS:[EAX],AL	
00402037	0000	ADD BYTE PTR DS:[EAX],AL	
00402039	0000	ADD BYTE PTR DS:[EAX],AL	
0040203B	0000	ADD BYTE PTR DS:[EAX],AL	
0040203D	0000	ADD BYTE PTR DS:[EAX],AL	
0040203F	0000	ADD BYTE PTR DS:[EAX],AL	
00402041	0000	ADD BYTE PTR DS:[EAX],AL	

DX=1000
ES:[EDI]=00401000=B8

Address	Hex dump	ASCII
00402000	68 65 6C 6C 6F 20 77 6F	hello wo
00402008	72 6C 64 00 00 00 00 00	rd... ..
00402010	00 00 00 00 00 00 00 00
00402020	00 00 00 00 00 00 00 00
00402028	00 00 00 00 00 00 00 00
00402030	00 00 00 00 00 00 00 00
00402038	00 00 00 00 00 00 00 00
00402040	00 00 00 00 00 00 00 00
00402048	00 00 00 00 00 00 00 00
00402050	00 00 00 00 00 00 00 00
00402058	00 00 00 00 00 00 00 00
00402060	00 00 00 00 00 00 00 00
00402068	00 00 00 00 00 00 00 00
00402070	00 00 00 00 00 00 00 00

Registers (FPU)

EAX 00402000 ASCII "hello world"
ECX 00401000 test.<ModuleEntryPoint
EDX 00401000 test.<ModuleEntryPoint
EBX 00299000
ESP 0019FF74
EBP 0019FF00
ESI 00401000 test.<ModuleEntryPoint
EDI 00401000 test.<ModuleEntryPoint
EIP 00402002 test.00402002
C 0 ES 002B 32bit 0(FFFFFFFF)
P 0 CS 0023 32bit 0(FFFFFFFF)
A 0 SS 002B 32bit 0(FFFFFFFF)
Z 0 DS 002B 32bit 0(FFFFFFFF)
S 0 FS 0053 32bit 29C000(FFF)
T 0 GS 002B 32bit 0(FFFFFFFF)
D 0
O 0 LastErr ERROR_FILE_NOT_FOUND (EFL 00010202 (NO,NB,A,NS,PO,GE,6
ST0 empty 0.0
ST1 empty 0.0
ST2 empty 0.0
ST3 empty 0.0
ST4 empty 0.0
ST5 empty 0.0
ST6 empty 0.0
ST7 empty 0.0
FST 0000 Cond 3 2 1 0 Err 0 0 0 0
FCW 027F Prec NEAR,53 Mask 1 1

0019FF74 75BEFA29 RETURN to KERNEL32.75BEFA29
0019FF78 00299000
0019FF7C 75BEFA10 KERNEL32.BaseThreadInitThunk
0019FF80 0019FFDC
0019FF84 77847BBE RETURN to ntdll.77847BBE
0019FF88 00299000
0019FF8C 0AE342DA
0019FF90 00000000
0019FF94 00000000
0019FF98 00299000
0019FF9C 00000000
0019FFA0 00000000
0019FFA4 00000000
0019FFA8 00000000
0019FFAC 00000000
0019FFB0 00000000

align伪指令

将变量的位置按字节、字、双字或段边界对齐

边界值为2，地址为偶数；边界值为4，地址为4的倍数...

```

bval byte? ;00404000
align 2
wval word?;00404002
bval2 byte?;00404004
align 4
dval dword?00404008
dval2 dword?0040400C

```

```

.data
var1 BYTE 10h, 20h
var2 DWORD 0AAAAAAAAh
ALIGN 4
var3 DWORD 0BBBBBBBBh

```

地址	十六进制数据
00403000	10 20 AA AA AA AA 00 00 BB BB BB BB 00 00 00 00
00403010	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403030	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00403040	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

cpu指令对齐，填充nop指令

ptr操作符

重载操作数声明的默认尺寸

可以使用 PTR 操作符来重载操作数声明的默认尺寸，这在试图以不同于变量声明时所使用的尺寸属性来访问变量的时候非常有用。

例如，假设要将双字变量 myDouble 的低 16 位送 AX 寄存器，由于操作数大小不匹配，编译器将不允许下面的数据传送指令：

```
.data
myDouble  DWORD  12345678h
.code
mov ax,myDouble           ; 错误
```

但是 WORD PTR 操作符使得将低字 (5678h) 送 AX 成为可能：

```
mov ax,WORD PTR myDouble
```

为什么不是 1234h 被送到 AX 寄存器了呢？这与 3.4.9 节中讨论的 Intel CPU 使用的小尾顺序存储格式有关。在下图中，我们列出了 myDouble 变量在内存中以三种方式显示的布局：双字、两个字 (5678h, 1234h) 和 4 个字节 (78h, 56h, 34h, 12h)。

双字	字	字节	偏移	
12345678h	5678	78	0000	myDouble
		56	0001	myDouble + 1
	1234	34	0002	myDouble + 2
		12	0003	myDouble + 3

CPU 能够以这三种方式中的任意一种访问内存，与变量定义的方式无关。例如，如果 myDouble 开始于偏移 0000，存储在该地址的 16 位值是 5678h，那么还可以使用下面的语句返回地址 myDouble+2 处的字 1234h：

```
mov ax,WORD PTR [myDouble+2] ; 1234h
```

类似地，可以使用 BYTE PTR 操作符把 myDouble 处的一个字节送到 BL：

```
mov bl,BYTE PTR myDouble ; 78h
```

注意，PTR 必须和汇编器的标准数据类型联合使用：BYTE，SBYTE，WORD，SWORD，DWORD，SDWORD，FWORD，QWORD 或 TBYTE。

将较小值送较大的目的操作数中：有时候，或许需要把内存中两个较小的值送到较大的目的操作数中。在下例中，第一个字将复制到 EAX 的低半部分，第二个字将复制到 EAX 的高半部分，DWORD PTR 操作符使这一切成为可能：

```
.data
wordList WORD 5678h,1234h
.code
mov eax,DWORD PTR wordList ; EAX = 12345678h
```

```
.data
var dword 12345678h
.code
start:
movzx eax,byte ptr var
```

```
movzx ebx,byte ptr [var+2]
end start
```

```
Registers (FPU)
EAX 00000078
ECX 00401000 test.<ModuleEntryPoint>
EDX 00401000 test.<ModuleEntryPoint>
EBX 00000034
ESP 0019FF74
```

```
data
var word 1234h
var2 word 5678h
.code
start:
mov eax,dword ptr var
end start
```

```
Registers (FPU)
EAX 56781234
ECX 00401000 test.<ModuleEntryPoint>
EDX 00401000 test.<ModuleEntryPoint>
```

type操作符

返回变量的字节数

lengthof操作符

计算数组中元素的个数，元素由出现在同一行的值定义

```
.data
byte1 BYTE 10,20,30
array1 WORD 30 DUP(?),0,0
array2 WORD 5 DUP(3 DUP(?))
array3 DWORD 1,2,3,4
digitStr BYTE "12345678",0
```

下表列出了每个 LENGTHOF 表达式的返回值。

表 达 式	值
LENGTHOF byte1	3
LENGTHOF array1	30+2
LENGTHOF array2	5*3
LENGTHOF array3	4
LENGTHOF digitStr	9

注意当在数组的定义中使用嵌套 DUP 定义时，LENGTHOF 将返回两个计数器的乘积。

如果声明了一个跨多行的数组，LENGTHOF 只把第一行的数据作为数组的元素。在下例中，LENGTHOF myArray 的返回值是 5：

```
myArray BYTE 10,20,30,40,50
          BYTE 60,70,80,90,100
```

然而，也可以在第一行的最后加一个逗号，以连接下一行的初始值。在下例中，LENGTHOF myArray 的返回值是 10：

```
myArray BYTE 10,20,30,40,50,
          60,70,80,90,100
```

sizeof操作符

返回值等于lengthof和type返回值的乘积

label伪指令

允许插入一个标号并赋予其尺寸属性而无需分配任何实际的储存空间

label后面的数据类型不能写缩写

LABEL 伪指令允许插入一个标号并赋予其尺寸属性而无需分配任何实际的存储空间。**LABEL** 伪指令可使用 **BYTE**、**WORD**、**DWORD**、**QWORD** 或 **TBYTE** 等任意的标准尺寸属性。**LABEL** 伪指令的一种常见用法是为数据段内其后定义的变量提供一个别名以及一个不同的尺寸属性。下例中在 **val32** 前面声明了一个名为 **val16** 的标号并赋予其 **WORD** 属性：

```
.data
val16 LABEL WORD
val32 DWORD 12345678h
.code
mov ax,val16           ; AX = 5678h
mov dx,[val16+2]       ; DX = 1234h
```

val16 是名为 **val32** 的存储地址的一个别名。**LABEL** 伪指令本身并不占用实际存储空间。

有时需要用两个较小的整数构造一个较大的整数。在下例中，由两个16位变量构成的32 位值被装入了 **EAX** 中：

```
.data
LongValue LABEL DWORD
val1 WORD 5678h
val2 WORD 1234h
.code
mov eax,LongValue      ; EAX = 12345678h
```

```
.data
dd_val label dword
dw_var1 dw 1234h
dw_var2 dw 5678h
dw_val label word
dd_v dd 12345678h
.code
start:
mov eax,dd_val
movzx ebx,dw_val
end start
```

Address | Hex dump

00402000 34 12 78 56 78 56 34 12
00402001 00 00 00 00 00 00 00 00

Registers (FPU)

EAX 56781234
ECX 00401000 test.<ModuleEntryPoint>
EDX 00401000 test.<ModuleEntryPoint>
EBX 00005678

间接寻址

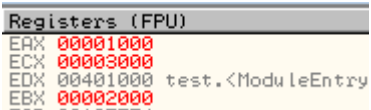
用寄存器作为指针并控制该寄存器的值称为间接寻址。

间接操作数

如果一个操作数使用的是间接寻址，就成为间接操作数。任何一个32位通用寄存器加上方括号就构成一个间接操作数。

32位整数相加

```
.data
array dword 1000h,2000h,3000h
.code
start:
mov esi,offset array
mov eax,[esi]
add esi,4
mov ebx,[esi]
add esi,4
mov ecx,[esi]
end start
```



Register	Value
EAX	00001000
ECX	00003000
EDI	00401000 test.<ModuleEntry
EBX	00002000

与ptr联合使用：指明操作数的尺寸大小

```
inc byte ptr[esi]
```

变址操作数

变址操作数把常量和寄存器相加以得到一个有效地址。任何一个32位通用寄存器都可以作为变址寄存器。

```
.data
array_dw DWORD 1000h, 2000h, 3000h
.code
mov esi, 0
mov eax, array_dw[esi]
add esi, 4
add eax, array_dw[esi]
add esi, 4
add eax, array_dw[esi]
```

```
.data
array_dw DWORD 1000h, 2000h, 3000h
.code
mov esi, OFFSET array_dw
mov eax, [esi]
```

```
add eax, [esi+4]
add eax, [esi+8]
```

```
.data
array byte 10h,20h,30h
.code
mov esi,0
mov al,array[esi*type array]
mov esi,1
add al,array[esi*type array]
mov esi,2
add al,array[esi*type array]
```

基址变址操作数

基址变址操作数把两个寄存器的值相加，得到一个偏移地址 `mov eax,[ebx+esi]`

相对基址变址操作数

相对基址变址操作数把偏移、基址、变址以及可选的比例因子组合起来，产生一个偏移地址。

```
.data
table dd 10000h, 20000h, 30000h
row_size = ($ - table)
dd 40000h, 50000h, 60000h
dd 70000h, 80000h, 90000h
.code
mov ebx, row_size
mov esi, 2
mov eax, table[ebx + esi * type table]
```

```
Registers (FPU)
EAX 00060000
ECX 00401000 test.<ModuleEntryPoint>
EDX 00401000 test.<ModuleEntryPoint>
EBX 0000000C
ESP 0019FF74
EBP 0019FF80
ESI 00000002
EDI 00401000 test.<ModuleEntryPoint>
EIP 00401011 test.00401011
```

指针

near指针，储存在双字变量中，相对数据段开始的16位偏移地址

```
arrayb byte 10h,20h,30h
arrayd word 1000h,2000h,3000h
ptrb dword arrayb
ptrw dword offset arrayd
```

typedef操作符：自定义类型

```
PBYTE TYPEDEF PTR BYTE
```

这样的声明一般被置于程序的开始处，通常在数据段之前。此后，就可以使用 PBYTE 来定义变量了：

```
.data
arrayB BYTE 10h,20h,30h,40h
ptr1 PBYTE ? ; 未初始化
ptr2 PBYTE arrayB ; 指向数组
```

例子程序：指针。下面的程序（pointers.asm）使用 TYPDEF 创建了三种指针类型（PBYTE、PWORD 和 PDWORD）。程序使用这些指针类型创建了几个指针变量并用数组的初始地址进行了初始化。在运行时析取这些指针的值用以访问数据：

```
TITLE Pointers (Pointers.asm)

INCLUDE Irvine32.inc

; 创建用户自定义类型
PBYTE TYPEDEF PTR BYTE ; 字节指针
PWORD TYPEDEF PTR WORD ; 字指针
PDWORD TYPEDEF PTR DWORD ; 双字指针

.data
arrayB BYTE 10h,20h,30h
arrayW WORD 1,2,3
arrayD DWORD 4,5,6

; 创建一些指针变量
ptr1 PBYTE arrayB
ptr2 PWORD arrayW
ptr3 PDWORD arrayD

.code
main PROC
; 使用指针变量访问数据
mov esi,ptr1
mov al,[esi] ; 10h
mov esi,ptr2
mov ax,[esi] ; 1
mov esi,ptr3
mov eax,[esi] ; 4
exit
main ENDP
END main
```

```
.data
arrayB BYTE 60h,61h,62h
arrayW WORD 6364h,6566h,6768h
arrayD DWORD 696A6B6Ch,6D6E6F70h,71727374h
ptr1 APBYTE arrayB
ptr2 APWORD arrayW
ptr3 APDWORD arrayD

.code
start:
mov eax,0
mov ebx,0
mov esi,ptr1
mov al,[esi]
mov esi,ptr2
mov bx,[esi+4]
mov esi,ptr3
mov ecx,[esi+type arrayD]
end start
```

Registers (FPU)	
EAX	00000060
ECX	6D6E6F70
EDX	00401000 test.<ModuleE
EBX	00006768
ECB	00000000

JMP和LOOP指令

控制转移：

- 无条件转移：将CPU控制权直接转移到指定的汇编语句。修改EIP为指定的内存地址；CPU从EIP指定的内存地址读取下一条机器指令。
- 条件转移

jmp指令

无条件转移

创建循环

```
top:
...
    jmp top ;无限循环
```

loop指令

loop指令重复执行一块语句，执行的次数是特定的，ecx被自动用作计数器，在每次循环后减一

执行包括两步：ecx-1，与0比较，如果不为0，跳转到目的地址处；如果等于0，不发生跳转。

```
    mov ax,0
    mov ecx,5
L1:
    inc ax
    loop L1
```

ecx初始化为0，减1得到FFFFFFFFh，循环4294967296次！

循环的嵌套

```
.data
count  DWORD 0
.code
MOV ECX, 100 ;L1 循环100次
L1:
MOV count, ECX
MOV ECX, 10 ; L2 循环10次
L2:
... ....
LOOP L2
MOV ECX, count
LOOP L1
```


数组求和

```
.data
array DWORD 100h, 200h, 300h, 400h
.code
MOV ECX, LENGTHOF array ; 循环次数
MOV EDI, OFFSET array
MOV EAX, 0
L1:
ADD EAX, [EDI]
ADD EDI, TYPE array
LOOP L1
```

字符串赋值

```
.data
src BYTE "Hello World", 0Dh, 0Ah, 0
dst BYTE SIZEOF src DUP(0), 0
.code
MOV ECX, SIZEOF src
MOV ESI, 0
L1:
MOV AL, BYTE PTR src[ESI]
MOV BYTE PTR dst[ESI], AL
INC ESI
LOOP L1
```

数组元素的访问

直接偏移寻址：

变量名称后加偏移值，访问没有显式标号的内存地址。如：

```
mov al,[array+1]
```

寄存器间接寻址：

使用间接操作数，间接操作数可以是任何用方括号括起来的任意32位通用寄存器，寄存器里存放着数据的偏移。

```
val byte 10h
mov esi,offset val
mov al,[esi]
```

寄存器相对寻址：

使用变址操作数：把常量和寄存器相加以得到一个有效地址。

constant[reg]

[constant+reg]

把变量名和寄存器结合在一起，变量名是表示变量偏移地址的常量。如：

```
mov esi,0
mov ax,[array+esi]
add esi,type array
add ax,array[esi]
```

把变址寄存器与常量偏移结合，用变址寄存器存放数组或结构的首地址，用常量标识各个数组元素。如：

```
mov esi, offset array
mov ax, [esi]
add ax, [esi+type array]
```

基址变址寻址：

基址变址操作数把两个寄存器的值相加，得到一个偏移地址。

[base + index]

如：

```
mov ebx,offset array
mov esi,2
mov ax,[ebx+esi]
```

相对基址变址寻址：

相对基址变址操作数把偏移、基址、变址以及可选的比例因子组合起来，产生一个偏移地址。

[base+index+displacement]

displacement[base+index]

displacement可以是变量的名字或常量表达式。如：

```
mov ax, 0x1000[bx][si]
mov ax, [bx + si + 0x1000]
mov ax, 0x1000[si][bx]
mov ax, 0x1000[bx + si]
```

使用指针

创建指针并用数组的初始地址进行初始化，运行时析取指针的值用以访问数据：

```
arraya dword 10h,20h,30h,40h
arrayb byte 10h,20h,30h,40h
```

```
ptra dword offset arraya
pbyte typedef ptr byte
ptrb pbyte arrayb
mov esi,ptra
mov eax,[esi]
mov esi,ptrb
mov bl,[esi]
```