# 目录

# 第 5 章 常用机器指令

Intel x86 的机器指令可以分为多个组，包括通用指令、x87 FPU 指令、MMX 指令、SSE/SSE2/SSE3/SSSE3/SSE4 指令、AVX/AVX2/AVX-512 指令、x86-64 位指令等等。本章主要介绍 x86 的一些通用指令，而其他指令将在本书的后续章节介绍。通用指令较多，它们又可以分成多个小组，每个小组都有一些共同规律。在学习指令时，注意比较指令之间的相同点和相异点，将它们关联起来并发掘它们之间的一些规律，掌握这些规律将降低指令记忆的难度。

## 5．1 通用机器指令概述

通用（General Purpose）机器指令为 Intel 所有的 x86-32 和 x86-64 中央处理器所支持。除了通用指令外，还有 x87 FPU（Float Point Unit，浮点运算单元）指令、MMX（Multi Media eXtension，多媒体扩展）指令、SSE Extensions（Streaming SIMD Extensions，单指令多数据流扩展）、SSE2、SSE3、SSSE3、SSE4、AVX（Advanced Vector Extensions，高级向量扩展）、AVX2、AVX-512、64 位指令、虚拟机扩展指令、安全模式扩展指令、内存保护扩展指令、软件保护扩展指令等等。本章仅介绍部分通用机器指令，其他指令在后续章节介绍。

随着处理器的发展，指令的数目在增加，但是 32 位的 x86 指令系统能很好地兼容 8086 的 16 位的指令系统。特点是：

1. 原有 8086 的 16 位操作指令都可扩展支持 32 位操作数；

2. 原有 16 位存储器寻址的指令都可以使用 32 位的寻址方式；

3. 在实方式和虚拟 8086 方式中段的大小只能为 64KB，只有在保护方式下才使用 32 位段。

x86 微处理器的通用指令可以分为以下几类。

(1) 数据传送指令；

(2) 算术运算指令（二进制算术、十进制算术）；

(3) 逻辑运算指令；

(4) 移位指令；

(5) 位和字节指令

(6) 标志位控制指令；

(7) I/O 指令；

(8) 控制转移指令；

(9) 串操作指令；

(10) 其他指令。

对于控制转移指令，将在第 6 章分支程序设计、第 7 章循环程序设计、第 8 章子程序设计中介绍。串操作指令在第 9 章串处理程序设计中介绍。

在介绍机器指令之前，应注意以下事项。

(1) 大多数指令具有相同的语句格式

双操作数语句的格式为：

[标号:] 操作符 opd，ops [;注释]

单操作数语句的格式为：

[标号:] 操作符 opd/ops [;注释]

其中，方括号中的内容是可以出现也可以不出现的。opd 是目的操作数的寻址方式，ops 是源操作数的寻址方式；

(2) 数据类型的一致性要求

一般的双操作数指令都要求目的操作数与源操作数的类型相同。如果两者的类型都明确，则两者应相同；如果两个都不明确，则一定使用类型定义符（byte ptr、word ptr、dword ptr）使其明确；如果有一个操作数的类型明确而另一个不明确，则采用的是明确的类型。操作数的类型编码在指令前缀或者指令操作码中。

例如：mov ebx,ecx ; 指令机器码为 8B D9

mov bx, cx ; 指令机器码为 66 8B D9

mov bl, cl ; 指令机器码为 8A D9

(3) 源操作数和目的操作数不能同时为存储器操作数

在双操作数指令中，假如一个操作数在数据存储单元中，则另一操作数要么是立即操作数(不能作目的操作数)，要么是寄存器操作数。

(4) 目的操作数不能是立即操作数

(5) 运算结果送入目的地址中(少数指令除外)，而源操作数不改变

(6) 单操作数的运算结果也是送入目的地址对应的单元中

再次强调，在没有指出指令特别规定的情况下，都应遵循上述一般性的要求。

# 5．2 数据传送指令

数据传送指令主要包括：

① 一般数据传送指令（mov、movsx、movzx、xchg、xlat）；

② 带条件的数据传送(cmove、cmovne、cmova、cmovae、cmovb、cmovbe、cmovg、cmovge、cmovl、cmovle、cmovc、cmovnc、cmovo、cmovno、cmovs、cmovns、cmovp、cmovnp)；

③ 堆栈操作指令(push、pusha、pushad、pop、popa、popad)；

④ 标志寄存器传送指令(pushf、pushfd、popf、popfd、lahf、sahf)；

⑤ 地址传送指令(lea、lds、les、lfs、lgs、lss)。

在这些指令中，除了 sahf、popf 两指令外，其它均不影响标志位。其他的杂项指令还有字节交换 bswap、比较并交换 cmpxchg 等指令。

## 5.2.1 一般数据传送指令

### 1. 数据传送指令

语句格式：mov opd, ops

功　　能：将源操作数传送至目的地址中，即(ops)→opd。

用　　途：可以在通用寄存器之间传递数据；　也可以在通用寄存器和存储器之间传递数据；还可以将一个立即数送给一个通用寄存器或存储单元。如果 opd、ops 中有段寄存器，一般会有较多的约束。

### 2. 有符号数传送指令

语句格式：　movsx opd, ops

功　　能：　将源操作数的符号向前扩展成与目的操作数相同的数据类型后再送入目的地址中。

要　　求：① ops 不能为立即操作数；

② opd 必须是 16 位或 32 位的寄存器；

③ 源操作数的位数必须小于目的操作数的位数。

**【例 5.1】** 阅读下列程序段，指出运行结束后，eax、ebx 的值。

```
byte0   db  0F8H, 56H
mov dl, byte0        ; (dl)=0F8H
movsx eax, dl        ; (eax)=0FFFFFFF8H,
movsx ebx, byte0+1   ; (ebx)=00000056H
```

本例中，两条 movsx 指令都是将一个字节的有符号数扩展为 32 位有符号数。

### 3. 有符号数传送指令

语句格式：movzx opd, ops

功　　能：将源操作数的高位全部补 0，扩展成与目的操作数相同的数据类型后再送入目的地址中。

要　　求：该指令的使用规定与 movsx 相同

### 4. 一般数据交换指令

语句格式：xchg opd, ops

功　　能：(opd)→ops，(ops)→opd，即将源地址与目的地址中的内容互换。

要　　求：① ops 不能为立即操作数，当然 opd 也不能为立即操作数；

② ops、opd 不能同时为存储器寻址方式。这是前面已明确的规定，不要只记住了数据交换，而忘了此规定。

**【例 5.2】** 写出交换 x 和 x+4 单元中内容的指令段，即 10 与 20 互换位置

```
x   dd   10,   20
```

方法 1：全部使用 mov 指令

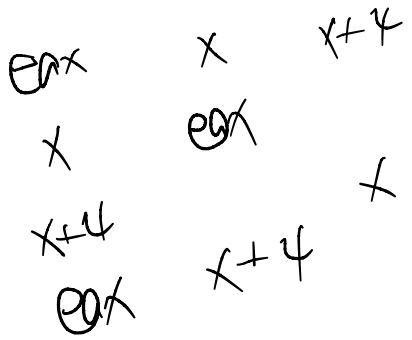
```
mov   eax, x
mov   ebx, x+4
mov   x,   ebx
mov   x+4, eax
```

方法 2：混用 mov 指令和 xchg 指令

```
mov   eax,  x
xchg  eax,  x+4
mov   x,    eax
```

方法 3：全部用 xchg 指令

```
xchg   eax,   x
xchg   eax,   x+4
xchg   eax,   x
```



### 5. 查表转换指令

语句格式：xlat

功　能：([ebx+al])→al 或([ebx+al])→al

即将(ebx)为首址、(al)为位移量的字节存储单元中的数据→al。

**【例 5.3】** 求由(al)指明的十六进制数码（即 0-9、A-F 之一）对应的 ASCII，结果放在(al)中

由(al)中的值是一个十六进制数码，可知(al)=0…15。根据 ASCII 表知道，当(al)=0…9 时，对应的 ASCII 是 30H-39H，即(al)+30H ->al；当(al)= 10…15 时，对应的 ASCII 是 'A'-'F',即 41H-46H，此时(al)+37H->al，也可以写成(al)-10+'A' ->al。如果按此算法写出程序段，无疑是比较麻烦的。下面给出用查表转换指令的实现方式。

```
tab    db   '0123456789ABCDEF'
mov    ebx, offset tab   ; 变量 tab 的地址→ebx
mov    al, 10            ; 查十六进制数码(A)对应的 ASCII
xlat   tab               ; ([ebx+al])=(TAB+10)=41H→al
```

当然，上述功能也可以使用一般的数据传送指令来实现，程序段如下：

```
movzx   eax, al
mov     al,  tab[eax]
```

从该例可看出，通过构造一个信息表有规律的存放数据时，可以极大简化信息编码之间的转换工作。例如对文本数据进行编码和译码，实现简单的加密和解密时，可以构造一个密码信息表，然后指定是查该表中的第几项，可以很容易的得到结果。

## 5.2.2 带条件的数据传输指令

带条件的数据传输指令虽然有很多条，但实际上它们有共同的规律，即根据某一个或某几个标志位的设置情况来判断条件是否成立；如果条件成立，则执行语句中的数据传送；否则不执行本语句中的数据传输。这些语句的语法格式也是相同的。

语句格式：cmov***　r32，r32/m32

功　　能：在条件“***”成立时，将源操作数传送至目的地址中，

即(r32/m32)→r32。cmov 是 Conditional MOVe 的缩写。

要　　求：① 目的地址是一个 32 位的寄存器，记为 r32；

② 源操作数地址是一个 32 位的寄存器或者是 32 位的内存地址（记为 m32）；m32 可以是直接寻址、间接寻址、变址寻址、基址加变址寻址；

③ m32 对应的单元的数据类型是双字，即 32 位。

“***”表示的条件可以用单个标志位的值来判断，例如 cmove/cmovz、cmovc、cmovs、cmovo、cmovp 分别是在 ZF=1、CF=1、SF=1、OF=1、PF=1 时，条件成立，执行后面的数据传送；与此相对的 cmovne/cmovnz、cmovnc、cmovns、cmovno、cmovnp 是在这些标志位为 0 时，

条件成立，执行数据传送。“***”表示的条件也可以用多个标志位的值来判断，如 cmova、cmovb、cmovg、cmovl 分别对应与高于(Above,CF=0 且 ZF=0)、低于(Below,CF=1 且 ZF=0)、大于（Great,SF=OF 且 ZF=0)、小于（Little,SF≠OF 且 ZF=0）的条件。它们还有完全等同的同名词（机器码相同），如高于等价于不低于等于，即 cmova 等价于 cmovnbe。除此之外，还有 cmovae、cmovbe、cmovge、cmovle 对应的条件为高于或等于、低于或等于、大于或等于、小于或等于。

带条件的数据传送语句是转移指令和数据传送指令的结合，它的引入使得程序编写更为简单、执行更为高效。

**【例 5.4】** 设有无符号双字类型变量 x、y、z,将 x 和 y 中间的大者存放到 z 中

```
x    dd   10
y    dd   20
z    dd    0
mov    eax, x
cmp    eax, y   ; 比较指令，根据(eax)-(y) 设置标志位
cmovb  eax, y   ; 使用前一条指令设置的标志位，决定是否要做数据传送
                ; 和前一语句连在一起的功能：若(eax)<(y) ,则（y）->eax;
                ; 否则不做传送
mov    z,     eax
```

注意，带条件的数据传送指令在 16 位指令系统、64 位指令系统下都是支持的，因而也可以写成：cmov***  r16，r16/m16；cmov***  r64，r64/m64。由于在 32 位环境下编程使用的地址是 32 位的，因而不能使用 16 位内存地址的寻址方式。

### 5.2.3 堆栈操作指令

堆栈是在主存中的一片数据存储区，它的访问和其它内存单元并没有什么本质差别，同样由段寄存器(ss)和段内偏移(esp)/(sp)、(ebp)/(bp)计算出待访问单元的物理地址。只不过用 esp/sp 指向栈顶，通过入栈（push）、出栈（pop）将数据压入堆栈和从堆栈中弹出数据，esp/sp 自动的变化。

x86 允许用户建立自己的字或双字堆栈，其存储区位置由堆栈段寄存器 ss 给定，并固定采用 sp（16 位段）或 esp（32 位段）作指针，即 sp 或 esp 的内容为栈顶相对于 ss 的偏移地址。空栈时， esp/sp 指向堆栈段的最高地址即栈底，存入时栈顶均由高地址向低地址变化。在 32 位扁平内存管理模式下，(ss)与(ds)相同，但是数据段中数据与堆栈段中的数据还是在同一大片空间中的不同位置，也可以说两者是分离的。

堆栈是非常重要的数据存储空间。在子程序的学习中，会看到函数参数都是通过堆栈来传递参数。从被调用的函数能够返回到主程序，也是因为将返回的地址放在了堆栈中。函数中定义的局部变量的空间分配同样在堆栈中。当然，这些局部变量和函数参数空间分配地址还是很巧妙的，在子程序设计中会详细介绍。

存取堆栈中的数据一般通过专门的指令进行。

#### 1．进栈指令

语句格式：push ops

功能：将立即数或寄存器、段寄存器、存储器中的一个字/双字数据压入堆栈中。

说明：当将立即数压入堆栈时，在 32 位段下，立即数会作为一个 32 位数。

**【例 5.5】** push 指令举例

① push 1234H

执行前：(esp)＝0021FA96

执行： ① (esp)-4 →esp ;(esp)= 0021FA92

② 00001234H→↓(esp)； ([esp])=00001234H

由此可见，esp 始终指向栈顶元素，即(esp)为当前栈顶元素的偏移地址。

执行后的堆栈示意如图 5.1 所示。

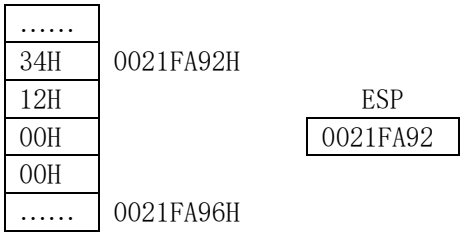



图 5.1 执行 push 1234H 后的堆栈示意图

② push ax

执行前：(esp)＝0021FA96 (ax)=1234H

执行： ① (esp)-2 →esp ;(esp)= 0021FA94

② 1234H→↓(esp) ;([esp])= 1234H

在压入一个字数据时 (esp)减 2；在压入一个双字数据时 (esp)减 4。由此可见，esp 减小量由数据所占的字节数决定。在 32 位段中，默认的立即数占 4 个字节。

③ push x

x 是一个定义在数据段的双字类型变量

执行： ① (esp)-4 →esp

② (x)→↓(esp)

push 指令默认了目的操作数的地址是堆栈栈顶，因此在指令中只需给出源操作数的寻址方式。

## 2．出栈指令

语句格式：pop opd

功能：将栈顶元素弹出送至某一寄存器、段寄存器(除 cs 外)或存储器中。

**【例 5.6】** pop ebx

设执行前 (esp)= 0021FA92，堆栈的双字数据为：([esp])=00001234H。

执行： ① ([esp]) → ebx ;(ebx)= 00001234H

② (esp)+4 →esp

出栈操作可记为：↑(esp)→ebx

注意，执行该操作后，在原来(esp)所指向的数据并没有发生变化，这与我们生活中从一个箱子里取出一个物品是不同的，它实际上做数据传送(拷贝)工作，只是做完传送后，(esp)发生了变化，指向了下一个单元。下面三条语句执行后，(eax)=(ebx)=00001234H。

```
push 1234H
pop  eax
mov  ebx, [esp-4]
```

**【例 5.7】** 下面程序段执行后，(ebx)的值是什么？

```
mov eax, -1
mov [esp], eax
pop ebx
```

我们在阅读分析程序时可以不关心(esp)的值是多少，而只需要知道各指令的功能是什么，各完成了什么操作即可。在执行“mov [esp], eax”时，(esp)是不会变化的，只是将(eax)传送到(esp)指向的单元，即堆栈的栈顶，执行 pop ebx 时，从栈顶取一个双字送给ebx，故 (ebx)=0FFFFFFFFH, 或者写成 (ebx)=-1。

**【例 5.8】**设数据段中定义了 2 个双字类型的变量 x 和 y，写出完成(x)->y 的指令段。

```
x  dd  10
y  dd  20
```

可写的实现方式有多种，但千万不能写成 “mov y,x”。请读者对照双操作数指令的规则，解释为什么。

```
方法①  mov  eax, x
       mov  y,  eax
方法②  push x
       pop  y
方法③  mov  eax, x
       xchg  eax, y
```

### 3．将 8 个寄存器内容顺序入栈指令

该指令分为对 8 个 16 位寄存器入栈指令 pusha 和对 8 个 32 位寄存器入栈指令 pushad 两种。

（1）语句格式：pusha

功能：将 8 个 16 位寄存器按 ax、cx、dx、bx、sp、bp、si、di 顺序入栈保存。

说明：在该指令执行时，压入堆栈的 sp 值是执行 pusha 之前的 sp 值。

（2）语句格式： pushad

功能：将 8 个 32 位寄存器按 eax、ecx、edx、ebx、esp、ebp、esi、edi 顺序入栈保存。

说明：其入栈保存的方式如 pusha，只是栈指针 esp 每次减 4。

### 4．将 8 个寄存器内容顺序出栈指令

相应的出栈指令为 popa 和 popad，出栈顺序与入栈顺序相反。

**补充知识**

严格的说，堆和栈是两个不同的概念。堆（heap）上分配的内存，系统不释放，而且是动态分配的。栈（stack）上分配的内存系统会自动释放，它是静态分配的。运行时栈叫堆栈。栈的分配是从内存的高地址向低地址分配的，而堆则相反。由 malloc 或 new 分配的内存都是从 heap 上分配的内存，从 heap 上分配的内存必须由程序员自己释放，用 free 或 delete 来释放，否则这块内存会一直被占用而得不到释放，就出现了“内存泄露（Memory Leak)”。这样会造成系统的可分配内存越来越少，导致系统崩溃。

## 5.2.4 标志寄存器传送指令

x86 微处理器为标志寄存器专门设置了几条操作指令，它们是：标志寄存器传送指令、标志寄存器进栈和出栈指令。这几条指令均不带操作数且只能对标志寄存器操作，也不能改变标志寄存器中未定义位的值。

### 1．将标志位传送到 ah 中

语句格式：lahf (Load flags into ah register)

功能：将标志寄存器的低 8 位送入 ah 中，即(eflags)$_{7\sim0}$→ah。该指令的执行对标志位

无影响。

说明：eflags 是本书为标志寄存器取的符号名，因而，它不能像寄存器名一样可在机器指令语句中直接引用。

### 2．将(ah)传送到标志寄存器

语句格式：sahf　(Store ah register into flags)

功能：将(ah)送入标志寄存器的低 8 位中，高位保持不变，即(ah)→eflags $_{7\sim0}$。

### 3．32 位标志寄存器进栈指令

语句格式：pushfd

功能：将标志寄存器的内容压入堆栈，记为(eflags)→↓(esp)。

### 4、32 位标志寄存器出栈指令

语句格式：popfd

功能：将栈顶内容弹出送入标志寄存器中，记为↑(esp)→eflags。

说明：该指令不影响标志位：RF、VM、IOPL、VIF、VIP 和未定义位。

**【例 5.9】**　popfd 示例

执行前：([esp])＝00000000H，(eflags)＝00003487H，（esp）＝00FFFF02H

执行：↑(esp)→eflags，（esp）＋4＝00FFFF02H＋4＝00FFFF06H

执行后：(eflags)＝00003002H

说明：由于标志寄存器中第一位总为 1，原 I/O 特权级为 3，都是不能改变的，因此，标志寄存器的内容为 00003002H。由于该例在执行前与执行后标志寄存器高 16 位均为 0，因此，未改变 RF、VM、VIF、VIP 的值。

### 5．16 位标志寄存器进栈指令

语句格式：pushf

功能：将标志寄存器的内容压入堆栈，记为(eflags)$_{15-0}$→↓(esp)。

### 6、16 位标志寄存器出栈指令

语句格式：popf

功能：将栈顶内容弹出送入标志寄存器中，记为↑(esp)→eflags $_{15-0}$。

## 5.2.5 地址传送指令

### 1．传送偏移地址指令

语句格式：lea opd, ops

功能：按 ops 提供的寻址方式计算偏移地址，并将其送入 opd 中。

说明：① opd 一定要是一个 32 位的通用寄存器；

② ops 所提供的一定是一个存储器地址；

③ 在所有的机器指令中，lea 是非常特殊的一条指令。它只计算了源操作数的地址，然后将计算的结果送到 opd 中，而其他指令都是要根据源操作数的地址去取相应的内存单元中的内容。

在 8086 和分段内存管理模式中，opd 可以是 16 位的通用寄存器。在 32 位段扁平内存管理模式下，由于寄存器间接寻址等都要求使用 32 位寄存器，因而给 16 位的寄存器赋值无实际意义。

**【例 5.10】**　取变量 x 的地址送给寄存器 ebx

设有如下数据段定义：

```
.DATA
 x db  '12345'
 p dd  x
```

方法① lea ebx，x ；与变量 x 定义的类型无关。

方法② mov ebx，offset x

注意，只有当 x 是全局变量时，能够在编译时确定其地址，才能用 offset 事先就得到其地址。当 x 是定义在子程序（函数）的局部变量时，则只能用 “lea ebx，x”。这与局部变量的空间分配方法有关，详见子程序设计中的介绍。

方法③ mov ebx，p

p 是一个双字类型的变量，其后的表达式是变量 x，存放的就是 x 的偏移地址。

**【例 5.11】** 编写程序段实现 (ebx)+(ecx)->eax

千万不要写 “mov eax，ebx+ecx”。请读者根据六种寻址方式的语法格式，判断其错误的原因。

方法 ①
```
mov eax, ebx
add eax, ecx
```

方法 ②
```
lea eax, [ebx+ecx]
```

请读者一定要注意“mov eax，[ebx+ecx]”和“lea eax，[ebx+ecx]”两语句的本质区别。前者要根据源操作数地址，取相应单元的内容送给 eax；而后者是将计算出的源操作数的有效地址直接送给 eax。另外，“lea eax，[ebx+ecx]”的操作码只有 lea，源操作数地址是一个基址加变址的表达形式。该指令的机器码为：8D 04 0B，有兴趣的读者可以按照 4.9 节中介绍的机器指令编码规则予以解读。

### 2．传送偏移地址及数据段首址指令

在 8086 和分段管理模式下，用户完全可以根据自己的需要，在一个程序中定义多个代码段、多个堆栈段和多个数据段。如果 CPU 需要访问 6 个段以外的存储区，只要改变相应段寄存器内容即可。当程序中使用多个数据段时，lds、les、lfs、lgs、lss 指令为随时改变段寄存器的内容提供了极大的方便。它们的格式是一样的。

语句格式：lds opd，ops

功　能：(ops)→opd，(ops＋2)→ds

说　明：① opd 一定要是一个 16/32 位的通用寄存器。

② ops 所提供的一定要是一个存储器地址，且类型为 dd/df。

在 32 位扁平内存管理模式下，不再需要使用这些指令。

## 5．3 算术运算指令

算术运算指令分为二进制数算术运算指令和 BCD 码算术运算调整指令，本节将对二进制数算术运算指令作详细的介绍，而 BCD 码算术运算调整指令只需读者作一般性了解。

二进制算术运算指令是指对二进制数进行加、减、乘、除运算的指令，主要包括：

加法类指令(inc、add、adc)；

减法类指令(dec、sub、sbb、neg、 cmp)；

乘法类指令(imul、mul)；

除法类指令(idiv、div)

符号扩展指令(cbw、cwd、cwde、cdq)。

除此之外，还有带进位的无符号整数加法 adcx、带溢出位的无符号整数加法 adox 等指令。在这些指令中，除符号扩展指令之外，均会不同程度地影响标志寄存器中的标志位。在第二章中介绍标志寄存器时，已介绍了常见的根据运算结果设置标志位的规则。

十进制算术指令有 daa(Decimal Adjust after Addition)、das(Decimal Adjust after Subtraction)、aaa/aas/aam （ASCII Adjust after Addition/Subtraction/ Multiplication)、aad（ASCII Adjust before Division)。有兴趣的读者可以自己去查阅资料，了解其用法。

### 5.3.1 加法指令

#### 1. 加 1 指令

语句格式：inc opd

功　能：(opd)＋1→opd

#### 2. 加法指令

语句格式：add opd, ops

功　能：(opd)＋(ops)→opd

将目的操作数与源操作数相加，结果存入目的地址中，而源地址中的内容并不改变。

#### 3. 带进位的加法指令

语句格式：adc opd, ops

功　能：(opd)＋(ops)+(cf）→opd

即将目的操作数、源操作数及标志位 cf 相加，结果存入目的地址中，而源地址中的内容并不改变。

**【例 5.12】** 用 32 位寄存器实现 64 位数的加法

```
x   dd   0F2345678H, 30010205H ; 从(x)取到的 64 位数是 30010205F2345678H
y   dd   10000002H, 55667780H ; 从(y)取到的 64 位数是 5566778010000002H
z   dd   0, 0
mov   eax, x
add   eax, y        ; (eax)=0234567AH, CF=1,ZF=0,SF=0, OF=0
mov   z,   eax      ; mov 指令不影响标志位
mov   eax, x+4
adc   eax, y+4      ; (eax)=85677986H, CF=0,ZF=0,SF=1,OF=1
mov   z+4, eax      ; z 中的 64 位数是 856779860234567AH
```

### 5.3.2 减法指令

1. 减 1 指令

语句格式： dec opd

功　　能：(opd)-1→opd

2. 求补指令

语句格式：neg opd

功　　能： 将目的操作数的每一位取反(包括符号位)后加 1→opd。

求补指令是求(opd)中相反数的补码，这与求(opd)的绝对值是有差别的。若要求(opd)的绝对值，首先需要判断(opd)的最高二进制位是为 1，若为 1，则再求补；若为 0，则直接返回。

3. 减指令

语句格式：sub　opd, ops

功　　能：(opd)－(ops)→opd

4. 带借位的减指令

语句格式：sbb　opd, ops

功　　能：(opd)－(ops)-(cf)→opd

用法与 adc 类似，用 32 位寄存器实现 64 位数的减法时，先做低 32 位的减法，然后做高 32 位的减法时，就要减去前面产生的借位。

5. 比较指令

语句格式： cmp opd, ops

功　　能： (opd)-(ops)

cmp 指令根据(opd)-(ops)的差来设置标志位，但该结果并不存入目的地址，该语句执行结束后，源地址和目的地址中的内容均不改变。在一般情况下，此语句的后面常常是条件转移语句，用来根据比较的结果实现程序的分支。

**【例 5.13】** 分析下面两条语句的功能

```
cmp　eax, 0
jne　L　　; 等价语句是 jnz　L　即 ZF=0 转移
```

第一条语句将(eax)与 0 比较，第二条语句是转移语句，根据前面一条语句的比较结果确定转移方向。如果(eax)≠0，则转至标号 L 处执行；否则顺序执行。

**【例 5.14】** 阅读下列程序段，指出它所完成的运算。

```
cmp　eax, 0
jge　exit　　; 如果(eax)≥0，则转 exit
neg　eax　　; 如果(eax)＜0，则 (eax)求补→eax
exit: ……
```

该程序段可实现求(eax)绝对值->eax。

本例中可以用 jns 来代替 jge。但是两个语句使用的标志位并不相同，详见第 6 章分支程序设计中有关转移指令的介绍。

下面给出另一种将(eax)绝对值送给 eax 的方法。

```
        cmp eax, 0
        jge exit        ; 如果(eax)≥0，则转 exit
        mov ebx, eax
        mov eax, 0
        sub eax, ebx
exit: ……
```

当然，还可以用乘法，当(eax)<0 时，(eax)*(-1)->eax。

### 5.3.3 乘法指令

在 x86 中，前面介绍的加法、减法运算都不区分运算对象是有符号数还是无符号数。有符号数在机内均采用补码表示，其最高位为符号位，计算机在进行运算时，并不单独处理符号，而是将符号作为数值一起参加运算。当然，若将两个运算数都当成有符号数或者都当成无符号数进行比较，结果是不同的，但是用 cmp 指令比较时，不存在有/无符号数的说法，标志位的设置是按规则设定的，只是条件转移指令使用的标志位不同而已。详见第 6 章中转移指令的介绍。

在乘法、除法运算中，要区别有符号数与无符号数的乘除法运算。x86 微处理器提供了有符号乘、除指令和无符号乘、除指令。

#### 1. 双操作数的有符号乘指令

语句格式：imul  opd, ops

功　　能：（opd）*（ops）→opd

说明：opd 可为 16/32 的寄存器，ops 可为同类型的寄存器、存储器操作数或立即数。

将两个操作数都当成有符号数，然后看计算的结果

```
例如：  imul eax, ebx                ;(eax)*(ebx)→eax
        imul eax, dword ptr [edi]    ;(eax)*([edi]) →eax
        imul eax, 5                  ;(eax)*5 →eax
        imul ax, bx                  ;(ax)*(bx)→ax
```

#### 2. 三个操作数的有符号乘指令

语句格式：imul  opd, ops, n

功　　能：（ops）* n →opd

说　　明：opd 可为 16/32 的寄存器，ops 可为同类型的寄存器、存储器操作数，n 为立即数

```
例如：imul eax, ebx, -10             ;(ebx)* (-10)→eax
      imul eax, dword ptr [edi], 5   ;([edi])*5→eax
      imul ax, bx, -10               ;(bx)*(-10) →ax
```

#### 3. 单操作数的有符号乘指令

语句格式：  imul ops

功　　能：字节乘法：  (al)*(ops)→ax

字　乘法：　(ax)*(ops)→dx,ax

双字乘法：(eax)*(ops)→edx,eax

说明：

(1) 字节/字/双字乘法，到底选哪一个？这是由 ops 的类型决定的。当 ops 是字节类型的地址时（包括字节类型的寄存器），则是字节乘法，依此类推；

(2) 该格式的乘运算指令只需指定源操作数(乘数)，而另一个操作数是隐含的，被乘数和乘积都在规定的寄存器中(不可使用其它寄存器)。源操作数只能是存储器操作数或寄存器操作数而不能是立即数，操作结束后，(ops)不变；

(3) 如果乘积的高位(字节相乘指结果在 ah 中的部分,字相乘指 dx,双字相乘指 edx)不是低位的符号扩展，即在 ah(或 dx、edx)中包含有乘积的有效位，则 CF＝1、OF＝1；否则，CF＝0，OF＝0。系统未定义乘法指令影响 SF、ZF、AF 和 PF 标志位。

单操作数的乘法指令比多操作数的乘法指令的使用方法要复杂一些，建议初学者在编程时尽量选用多操作数的乘法指令。

#### 4. 无符号乘指令

语句格式：mul ops

功　　能：字节乘法：(al)*(ops)→ax

字　乘法：　(ax)*(ops)→dx,ax

双字乘法：(eax)*(ops)→edx,eax

说明：

(1) 该指令的使用格式与单操作数的 imul 相同，只是参与运算的操作数及相乘后的结果均是无符号数。

(2) 如果乘积的高位不为 0，即在 AH(或 dx/edx)中包含有乘积的有效位，则 CF＝1、OF＝1；否则，CF＝0、OF＝0。系统也未定义该乘法指令影响 SF、ZF、AF 和 PF 标志位。

**【例 5.15】**　有符号乘法和无符号乘法的比较

```
mov  al,10H
mov  bl, -2        ;  (bl)=0FEH
imul bl
```

执行后(ax)=0FFE0H，结果高字节无有效位，有 NC，NV

若将 imul bl 换成 mul bl,执行后 (ax)= 0FE0H，结果高字节有有效位,有 CY(CF=1)，OV(OF=1)。

### 5.3.4 除法指令

#### 1. 无符号除指令

语句格式：　div ops

功　　能：字节除法：(ax)/(ops)→al(商) 、ah(余数)

字　除法：(dx、ax)/(ops)→ax(商)、dx(余数)

双字除法：(edx、eax)/(ops)→eax(商)、edx(余数)

说　　明：

(1) 除法类型由 ops 的类型决定；

(2) ops 不能是立即操作数，且指令执行后，(ops)不变；

(3) 如果除数为 0，即(ops)=0，产生异常 Integer division by zero；

(4) 如果被除数太大，而除数小，使得商在相应的寄存器中存放不下，则产生溢出异常 Integer overflow。

例如,(ax)=1234H ,(bl)=1,执行 div bl， 按理论结果，商为 1234H，将其送入到 al 中，无疑超出了一个字节的表示范围，故溢出。

### 2. 有符号除指令

语句格式： idiv ops

功　　能： 字节除法：(ax)/(ops)→al(商)、ah(余数)

字　除法：(dx，ax)/(ops)→ax(商)，dx(余数)

双字除法：(edx、eax)/(ops)→eax(商)、edx(余数)

说明：(1)-(4)与 div 语句相同；

(5)相除后，所得商的符号与数学上规定相同，但余数与被除数同号。

为了避免除法产生溢出错误，应该使用更长的数据类型参与运算。

例如，要实现(ax)=1234H 和(bl)=1 的无符号数除法，可以写如下指令

```
mov    dx，0
movzx bx，bl
div    bx      ；用(dx,ax)作为被除数，(bx)为除数，商 1234H->ax。
```

在记忆字节除法指令时，是用(al)存放商还是余数，是有诀窍的。假设要将（ax）中的一个十六进制数码转换成 2 进制串输出，例如，将(ax)=000AH 转换成“31H 30H 31H 30H”，即 1010B 对应的 ASCII 串输出。我们可以用除 2 的方法，(ax)/2 的余数->ah,是最后的一位的二进制码，然后用商(al)继续除 2，直到商为 0。从前面的除法指令的规则可知，字节除法的被除数为(ax)，这就需要将商(al)扩展为(ax)。由符号扩展的方法知，是将低位的符号向前扩展，应而将商放在 al 中扩展起来就很方便。

## 5.3.5 符号扩展指令

### 1. 将字节转换成字指令

语句格式：cbw

功　　能：将 al 中的符号扩展至 ah 中。若(al)的最高二进制位为 1,则(ah)=0FFH；

若(al)的最高二进制位为 0，则(ah)=00H；

说　　明：cbw 是 Convert Byte to Word 的缩写

该语句等效于“movsx ax, al”。

### 2、 将字转换成双字指令

语句格式：cwd

功　　能：将 ax 中的符号扩展至 dx 中

说　　明：cwd 是 Convert word to doubleword 的缩写

在使用字除法指令时被除数是(dx,ax)，可以用该指令将(ax)扩展成(dx,ax)。

在 8086 时代，没有 32 位的寄存器，(ax)符号扩展到(dx)。

### 3、 将字转换成双字指令

语句格式：cwde

功　　能：将 ax 中的有符号数扩展为 32 位数→eax

说　　明：cwde 是 Convert Word to Doubleword in Eax register 的缩写。

该语句等效于“movsx eax, ax”。

### 4、 将双字转换成四字指令

语句格式：cdq

功　　能：将 eax 中的有符号数扩展为 64 位数→edx,eax

说　　明：cdq 是 Convert Doubleword to Quadword 的缩写

## 5．4 逻辑运算指令

x86 微处理器提供逻辑运算指令是一种按位操作指令，即对应的二进制位上的操作结果与其他位置上的二进制位无关，所有二进制位采用相同的操作规则。现分述如下。

### 1．求反指令

语句格式： not opd

功　能：　将目的地址中的内容逐位取反后再送入目的地址中，即$\overline{(opd)}$→opd

注意和 neg 指令的差别，neg 是求补。

### 2．逻辑乘指令

语句格式： and opd，ops

功　能：　(opd)∧(ops)→opd

目的操作数和源操作数按位做逻辑乘运算，其结果存入目的地址中。

说明：逻辑乘的运算法则为：1∧1＝1，1∧0＝0，0∧1＝0，0∧0＝0

逻辑乘可以用于清除某些二进制位。源操作数与目的操作数逻辑乘后，在结果中对应源操作数二进制位置为 0 的那些位一定为 0。例如，保留(ax)的最低 4 位，而让其他位为 0，即第 15～4 位清 0，可以写 “and ax，0FH”。如果要保留(ax)的最高二进制位，可以写“and ax，8000H”，若该运算的结果为 0（ZF=1），说明原来(ax)的最高位为 0；若该运算的结果不为 0（ZF=0），说明原来(ax)的最高位为 1。

### 3．测试指令

语句格式： test opd，ops

功　　能：　(opd)∧(ops)

目的操作数与源操作数做逻辑乘运算，并根据结果置标志位 SF、ZF、PF，操作结束后，源操作数地址和目的操作数地址中的内容并不改变。

该指令主要用来检测与源操作数中为 1 的位相对应的目的操作数中的那几位是否为 0(或为 1)，它的后面往往跟着转移指令，根据测试的结果决定转移方向。

例如，如果要测试(ax)中第 7 位和第 15 位是否同时为 0，为 0 转 L，可用以下语句实

现：

```
test  ax，8080H
jz    L
```

### 4．逻辑加指令

语句格式： or opd，ops

功　　能： (opd)∨(ops)→opd

目的操作数与源操作数做逻辑加运算，结果存入目的地址中。

说明：逻辑加的运算法则为：1∨1＝1，1∨0＝1，0∨1＝1，0∨0＝0。

逻辑加可以将某些二进制位置为 1。例如，要使(ax)中的第 0、2、4、6 位均置 1，其余位仍保持不变,则可以使用指令“or ax，55H”，55H 对应 01010101B。

### 5．按位加指令

语句格式： xor opd，ops

功　　能： (opd) ⊕ (ops)→opd

目的操作数与源操作数做按位加运算，其结果送入目的地址中。

说明：按位加的运算法则为：1⊕1＝0，1⊕0＝1，0⊕1＝1，0⊕0＝0。

该指令主要用于将目的地址中与源操作数置 1 的对应位取反。如果操作数自身作按位加，其结果为 0。例如“xor ax，ax”等价于语句“mov ax，0”。

## 5．5 移位指令

移位指令包括算术移位指令(sal,sar)、逻辑移位指令(shl,shr)、循环移位指令(rol,ror,rcl,rcr)和双精度移位指令(shld,shrd)。前三种指令的语句格式相同：

**操作符 opd，n**

其功能为：将目的操作数中的所有位按操作符所规定的方式移动 n 所规定的次数，然后将结果送入目的地址中。n 超过目的操作数的长度。

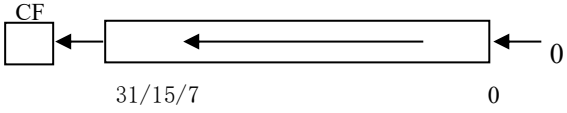
下面将分别介绍各指令的移动方式。

### 1．算术左移和逻辑左移指令

语句格式：

```
sal opd，n    ；Shift Arithmetic Left
shl opd，n    ；SHift logical Left
```

功能：将(opd)向左移动 n 指定的位数，而低位补入相应个数的 0。CF 的内容为最后移入位的值。




“sal　opd，n” 等价于“sal　opd，1”连续做 n 次。

使用 sal 和 shl 可以很方便地实现有符号数和无符号数乘 $2^n$ 的运算。不过在使用时应注意是否会发生溢出。

### 2．逻辑右移指令

语句格式：shr opd，n　；SHift logical Right

功能：将(opd)向右移动 n 规定的次数，最高位补入相应个数的 0，CF 的内容为最后移入位的值。

使用 shr 指令可以很方便地实现对无符号数除 $2^n$ 的运算。

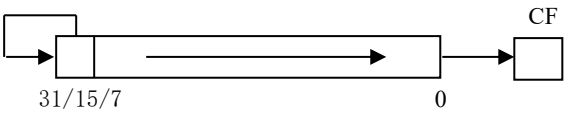
除此之外，逻辑移位指令还有一个很重要的用途，就是可以将一个字(或字节)中的某一位(或几位)移动到指定的位置，从而达到分离出这些位的目的。

例如，执行 shr　ah,4 后，原(ah)的高 4 位就移动到了低 4 位，新的高 4 位为 0，从而使得新的(ah)为原(ah)的高 4 位的值，即原(ah)高 4 位的十六进制数码。

### 3. 算术右移指令

语句格式：sar opd，n　；Shift Arithmetic Right

功能：将(opd)向右移动 n 指定的次数且最高位保持不变。CF 的内容为最后移入位的值。

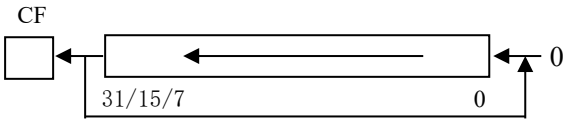



使用 sar 可以很方便地实现对有符号数除 $2^n$ 的运算。

### 4. 循环左移指令

语句格式：　rol opd，n　；ROtate Left

功能：将目的操作数的最高位与最低位连接起来，组成一个环，将环中的所有位一起向左移动 n 所规定的次数。CF 的内容为最后移入位的值。




### 5.　循环右移指令

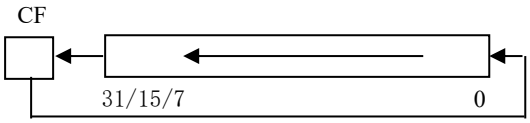
语句格式：ror opd，n　；ROtate Right

功能：该指令的移动方式完全同 rol，只是向右移动。

### 6. 带进位的循环左移指令

语句格式：rcl opd，n　；Rotate through Carry Left

功能：将目的操作数连同 CF 标志一起向左循环移动所规定的次数。




### 7. 带进位的循环右移指令

语句格式：rcr opd，n　；Rotate through Carry Right

功能：该指令的移动方式完全同 rcl，只是向右移动。

### 8. 双精度左移指令

语句格式：shld　opd,ops,n　；SHift Left Double

功能：　将 ops 的最高 n 位移入 opd 低 n 位中，ops 保持不变，opd 最后移出的一位保存在 CF 中。

说明：ops 只能为与 opd 类型相同的 16 位或 32 位的寄存器。

**9.双精度右移指令**

语句格式：shrd　opd,ops,n

功能：　　将 ops 的最低 n 位移入 opd 高 n 位中，ops 保持不变，opd 最后移出的一位保存在 CF 中。

## 5．7 位和字节指令

位指令实现的功能是测试和修改一个字或者双字中的某一位。字节操作指令根据某种条件是否成立设置一个字节的值，条件成立将该字节的值设为 01H，否则置为 00H。位操作指令有 bt、bts、btr、btc、bsf、bsr；字节操作指令有 sete、sets、seto、setp、seta、setb、setg、setl、setae 等等。下面仅以 bt 和 sete 为代表进行介绍。

（1）bt 指令

bt 指令的功能是根据位编号来对目的操作数中的位进行测试。对于字节类型，从最低为到最高位的编号是从 0 至 7；字是从 0 至 15；双字是从 0 至 31。

语句格式：bt opd，ops

功　能：将目的操作数中的指定位的值→CF，该位的编号由源操作数指定。操作结束后，opd、ops 的内容并不改变。

说　明：

① opd 必须为 16/32 位的寄存器或存储单元；

② ops 只可为立即数或与 opd 同类型的寄存器操作数，不能是存储器操作数；

③ ops 的值应在-128-+127 之间。当 ops 的绝对值大于 opd 的位数时，系统则取 ops 除 16 或 32（由 opd 的类型决定）后的余数作为 opd 的位号。

（2）sete 指令

语句格式：sete r8/m8

功　能：当 ZF=1 时；将目的操作数设置为 01H，否则将目的操作数设置为 00H。

说　明：

① r8 表示一个字节寄存器；

② m8 表示一个数据为字节的地址表达式，可以是直接寻址、寄存器间接寻址、变址寻址、基址加变址寻址。

在指令指令中，set***中的“***”与带条件的数据传送指令的条件表达是相同的。

## 5．8 标志位控制指令和杂项指令

涉及到标志寄存器（EFLAGS）中标志位的指令除了前面已介绍过的 lahf、sahf、pushf、popf、pushfd、popfd 之外，还有：stc、clc、cmc、std、cld、sti、cli。这些指令后面都没有操作数，它们都隐含规定了操作对象。

stc、std、sti：分别置（SeT）进位标志位（Carry flag）CF、方向标志位（Direction flag)、中断允许标志位(Interrupt flag)为1。

clc、cld、cli 分别清除（CLear）CF、DF、IF，即将相应的标志位设为0。

cmc 是对 CF 进行求反。

杂项执行包括 nop、cpuid、rdrand 等等。

① nop

该指令后无操作数,仅仅占一个字节,CPU 执行该指令时,不做任何操作(No OPeration)。

② cpuid

该指令用于获取当前 CPU 的信息，包括 CPU 的系列代号(family，如 PentiumIII 为第 6 代)、型号(model)、CPU 步进(Stepping ID)、CPU 字串、CPU 的缓存等信息。该指令是一个无操作数的指令，但是用（eax）来指明要获取什么信息（相对于函数 CPUID 的入口参数)；获取结果在 eax、ebx、ecx、edx 中，这些寄存器中各个位代表什么含义都比较较繁杂。使用获取的信息可用于判明当前 CPU 是否具有某种处理能力。

③ rdrand R32

产生一个 32 位的随机数，结果放在 32 位的寄存器 R32 中。

## 5．9 I/O 指令

计算机主机（CPU 和内存）从外部设备获取数据称为输入（Input)，向外部设备传送数据称为输出（Output)，简称 I/O。外部设备通过外部设备寄存器与主机之间交换数据。外部设备寄存器也称为端口（Port)。

按外部设备寄存器的作用不同，可分为设备状态寄存器（状态端口)、设备控制寄存器（控制端口）和数据寄存器（数据端口）三大类。每种寄存器的数量由具体外部设备决定。CPU 访问内存需要知道被访问单元的地址。同样的，访问外部设备需要知道外部设备的地址。每个设备寄存器均分配了一个唯一的编号，即端口地址。这些地址形成了 I/O 空间，其集合不应超过 64K（如果所需寄存器容量超过 64KB，则必须通过其它机制转换地址空间)。

在 X86 系统中， I/O 空间的编址方式与主存相同，地址范围 0000H～0FFFFH，共 64K。。I/O 空间与主存空间不同的是，I/O 空间只能由 I/O 指令访问（I/O 指令可使 I/O 读写控制线有效)，而对其它任何指令的访问一律视为无效。

### 1. 输入指令

语句格式： in al/ax/eax，dx/n

功　　能：从端口号为（dx）或 n 的设备寄存器中取数据送入 al、ax 或者 eax 中。

说　　明：① n 为一个立即数,取值范围是 0～255;

② 外设寄存器的地址大于 255 时，要使用 dx 存放待访问的端口号;

③ 从端口中输入一个字节、字、双字数据分别存放在 al、ax、eax 中;

④ 当输入一个字或双字时，数据存放在连续的两个或四个端口中，指令中

给出的端口号是最低字节对应的端口的编号。

例 1　in　al，60H

执行前：（60H）=41H，（al）＝56H

执行后：（al）＝41H，（60H）不变

说明：60H 是键盘的端口地址，该端口中存放当前按键的键码。

当（dx）=60H 时，指令“in al，dx” 与本指令的功能相同。

### 2. 输出指令

语句格式：　out　dx/n，　al/ax/eax

功　　能：将(al)、(ax)或者(eax)输出到端口号为（dx）或 n 的设备寄存器中。

### 3. 串输入指令

语句格式：　insb　　；输入一个字节

insw　　；输入一个字

insd　　；输入一个双字

ins　　opd，dx

功　　能：① insb、insw、insd 是无操作数的指令；隐含（[dx]）→es:[edi]；

② opd 的作用是为了确定输入是字节、字、还是双字；

③ “ins opd，dx”将被翻译成 insb、insw 或 insd。

当 DF＝0 时，(edi)增量 1(字节操作)或 2(字操作) 或 4(双字操作)。

当 DF＝1 时，(edi) 减量 1(字节操作)或 2(字操作) 或 4(双字操作)。

如果同 rep 前缀连用,ins 可以从一输入端口传送信息块到一连续的存储器空间。例如，通过磁盘控制器的端口读入一个文件，通过网卡适配器的端口读入数据帧等。

### 4. 串输出指令

语句格式：　outs　dx，ops

outsb　；输出一个字节

outsw　；输出一个字

outsd　；输出一个双字

功　　能：（ds:[esi]）→[dx]

当 DF＝0 时，(esi)增量 1(字节操作)或 2(字操作) 或 4(双字操作)。

当 DF＝1 时，(esi) 减量 1(字节操作)或 2(字操作) 或 4(双字操作)。

如果同 rep 前缀连用，outs 可将一连续的主存储器的内容传送到一输出端口中（ops 为源串的符号首址））。

值得注意的是，在 Windows 系统中，应用程序不能直接使用 I/O 指令。尽管程序中使用 I/O 指令后，编译和链接正常，但执行程序时会出现异常，指出 I/O 指令是特权指令（Privileged instruction）。 标志寄存器 EFLAGS 中的 IOPL 位定义了使用 I/O 指令的权力，一般特权级为 0 和 1 的程序具有对 I/O 指令具有访问权，而应用程序的特权级是最低的，为 3 级，无权利使用 I/O 指令。

当然，每个程序的任务状态段 TSS 中用二进制序列存放着每个任务可以访问的端口地址映像关系，称为允许图，每个端口对应着一个二进制位，用 0 或者 1 表示能否访问该端口。在不满足特权级要求的情况下，CPU 会检查 I/O 允许图。如果 I/O 允许图中不允许访问该端口，CPU 就发出一般保护异常信号。如果允许访问，则 I/O 操作可以继续执行。

在 Windows 的保护机制之下，应用程序要访问外部设备就要通过 Windows 操作系统提供的 API 来实现。本书的例子程序中使用 printf、scanf 之类的函数实现输入输出。 printf 通过调用 windows API 函数 GetStdHandle 来得到标准设备 Std 的句柄，然后使用 WriteFile 这个 API 来写该设备，从而实现在屏幕上显示信息。

# 习题 5

5.1 设有如下数据段，要求将 (x)和(x+2) 中的字数据交换位置。按要求写出实现该功能的语句或语句片段。

```
.data
x  dw  10,  20
```

(1) 只使用 mov 指令
(2) 只使用 xchg 指令
(3) 混合使用 mov 和 xchg 指令
(4) 用 push、pop 指令
(5) 用循环左移指令
(6) 用循环右移指令

5.2 对于 5.1 所示的数据段，写出将(x+2) 中的字数据送入 ax 中的指令或指令段，要求访问(x+2)时使用指定的寻址方式。

(1) 直接寻址
(2) 寄存器间接寻址
(3) 变址寻址
(4) 基址加变址寻址

5.3 编写程序段，判断(ax)的最高位是否为 1，若是则转移到 L 处。在判断时要求用到指定的指令。

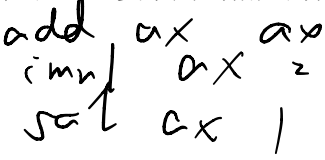
(1) cmp 指令
(2) test 指令
(3) or 指令
(4) shl 指令
(5) rol 指令

5.4 编写程序段，求 (ax)的绝对值，结果仍保留在 ax 中，要求用到指定的指令。

(1) 求补指令
(2) 乘法指令

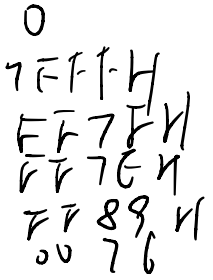
(3) 减法指令
(4) 求反指令

5.5 编写程序段，求 (ax)*2->ax，不考虑溢出，要求用到指定的指令。
(1) 加法指令
(2) 乘法指令
(3) 移位指令



5.6 运行如下程序段，给出每条语句执行后 ax 中的十六进制内容是什么？

```
mov  ax,0
add  ax,7FFFH
xchg ah, al
dec  ax
add  ax,0AH
not  ax
sub  ax,0FFFFH
or   ax, 0ABCDH
and  ax, 0DCBAH
sal  ax, 1
rcl  ax, 1
```



## 上机实践 5

5.1 设有如下的 C 语言程序段

```
int  x=10;
int  y=20;
x = x - y;
y = x + y;
x = y - x;
```

该段程序的功能是什么？用反汇编观察其生成的代码。用汇编语言写出实现同等功能的优化代码。

5.2 设有如下的 C 语言程序段

```
int  x=10;
int  y=20;
x = x^y;
y = x^y;
x = x^y;
```

该段程序的功能是什么？用反汇编观察其生成的代码。用汇编语言写出实现同等功能的优化代码。

5.3 编写程序，统计(ax)的二进制数中数码 1 出现的次数。
例如 （ax）=7000H， 数码 1 出现 3 次。

(3) 减法指令
(4) 求反指令

5.5 编写程序段，求 (ax)*2->ax，不考虑溢出，要求用到指定的指令。
(1) 加法指令
(2) 乘法指令
(3) 移位指令

5.6 运行如下程序段，给出每条语句执行后 ax 中的十六进制内容是什么？

```
mov  ax,0
add  ax,7FFFH
xchg ah, al
dec  ax
add  ax,0AH
not  ax
sub  ax,0FFFFH
or   ax, 0ABCDH
and  ax, 0DCBAH
sal  ax, 1
rcl  ax, 1
```

# 上机实践 5

5.1 设有如下的 C 语言程序段

```
int  x=10;
int  y=20;
x = x - y;
y = x + y;
x = y - x;
```

该段程序的功能是什么？用反汇编观察其生成的代码。用汇编语言写出实现同等功能的优化代码。

5.2 设有如下的 C 语言程序段

```
int  x=10;
int  y=20;
x = x^y;
y = x^y;
x = x^y;
```

该段程序的功能是什么？用反汇编观察其生成的代码。用汇编语言写出实现同等功能的优化代码。

5.3 编写程序，统计(ax)的二进制数中数码 1 出现的次数。
例如 （ax）=7000H， 数码 1 出现 3 次。