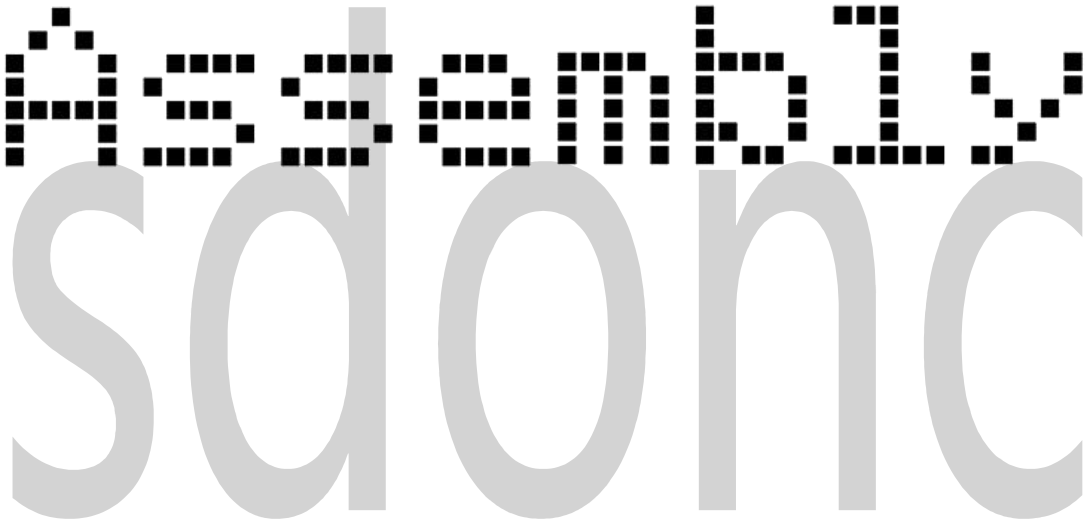


日期：2018 年 7 月

作者：sdonc

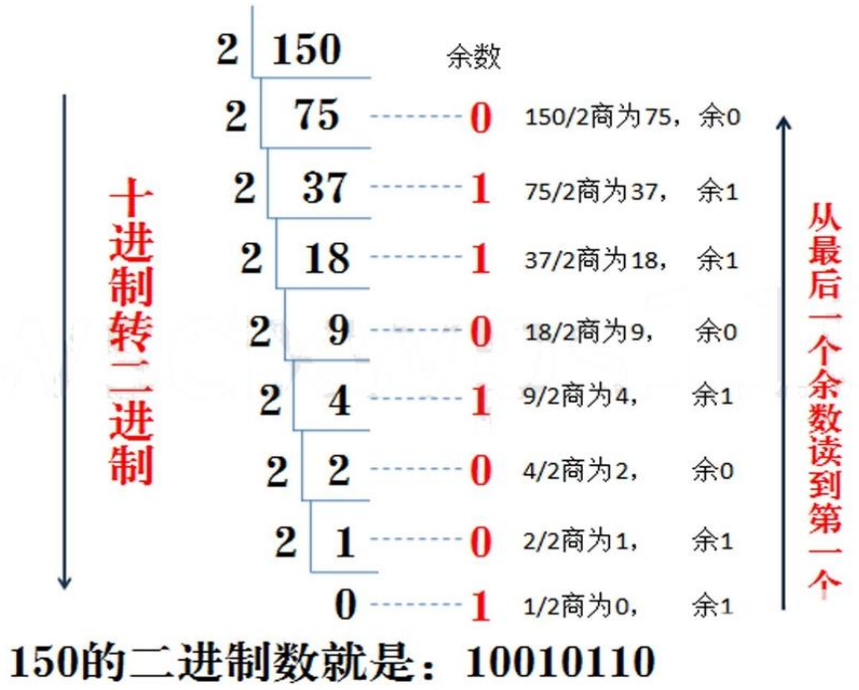
目录

一、基础概念

对应章节：第一章：基本概念、第二章：x86 处理器架构、第三章：汇编语言基础

1、进制转换

1.1 十转二

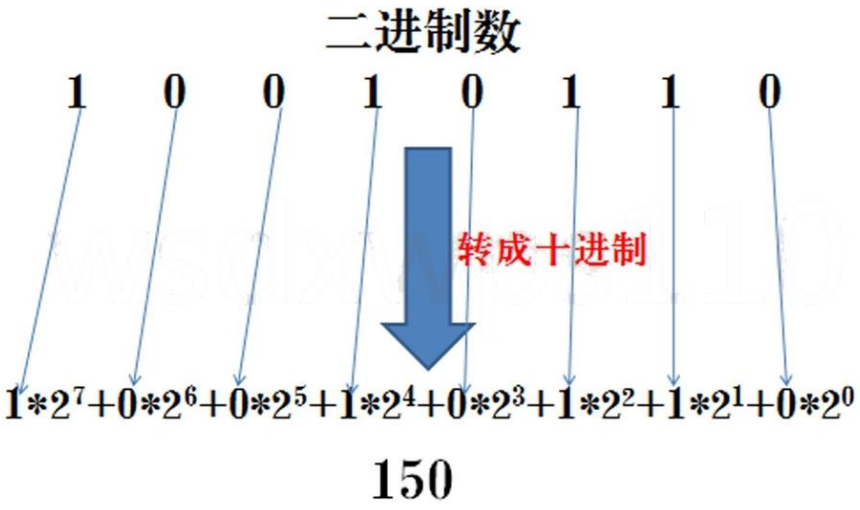


示例：十进制转二进制

(1)23 = 10111

(2)360 = 101101000

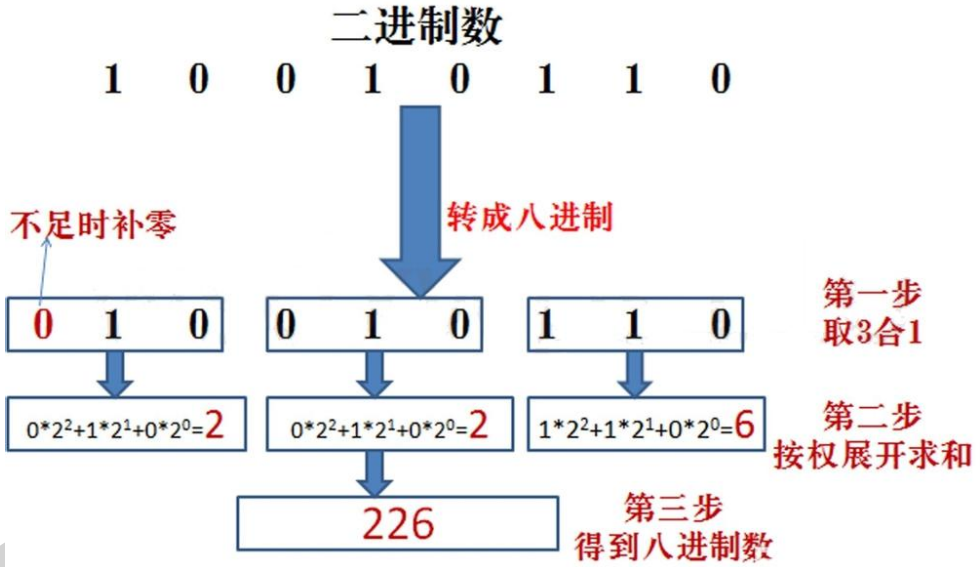
1.2 二转十



示例：二进制转十进制
(1)101100 = 44
(2)1010011010 = 666

1.3 二转八

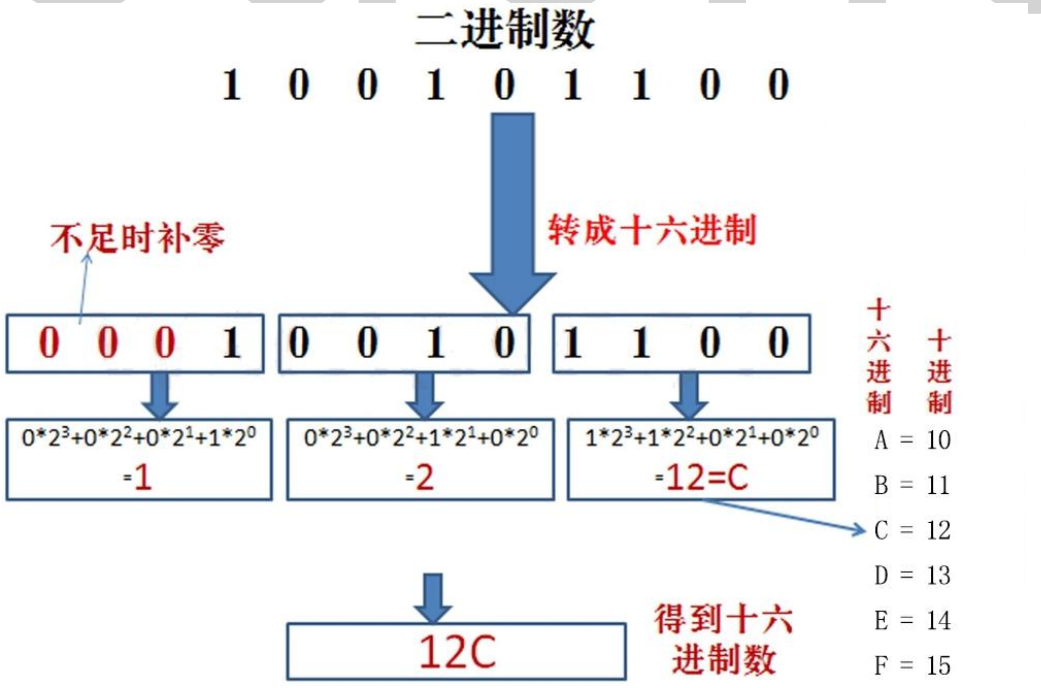
方法：从第到高，每三位一组，高位不足补 0。



示例：二进制转八进制
(1)11111100010 = 3742
(2)1100101001101O1 = 62465

1.4 二转十六

方法：从第到高，每四位一组，高位不足补 0。



示例：二进制转十六进制
(1)11111100010 = 7E2
(2)101010111100 = ABC

2、二进制、十六进制加减法

无符号二/十六进制数加减法与整数加减法类似，此处不再赘述。

有符号二进制数加减法首先要确定这两个有符号数的尺寸（例如 8 位二进制数或 16 位二进制数），然后按照正常的整数加减法来进行计算即可（注：如果有溢出则 CF=1）。

例如：
注：下面的两个数是 8 位二进制有符号数。

①

```
  00110010  (+50)
  01000110  (+70)
+)0 00001100 (进位)
  0 01111000(+120)
```

②

```
  11101100  (-20)
  11100010  (-30)
+)1 11000000  (进位)
  1 11001110(-50)
```

③

```
   01011001  (+89)
   01101100  (+108)
+) 011110000  (进位)
   011000101  (-59)
```

正溢出

④

```
    10010010  (-110)
    10100100  (-92)
+) 1 00000000  (进位)
   1 00110110  (+54)
```

负溢出

2.1 8 位无符号二进制数加减法

(1)1100 + 101010 = 110110
(2)11100 - 101 = 10111

2.2 8 位有符号二进制数加减法

(1)11011100 - 00101010 = 10110010
(2)00010111 - 01100010 = 10110101
(3)11111100 + 00011010 = 00010110

3、常量

末尾	进制	示例
空/d	十进制（Decimal）	123
h	十六进制（Hexadecimal）	4dfh、0abch
b	二进制（Binary）	101100b
q/o	八进制（Octonary）	523q

注：以字母开头的十六进制数前面要带上 0，例如：0abch、0ffdh 等。

4、变量的定义、类型

无符号整数	缩写	英文	中文	长度（字节）
byte	DB	Byte	字节	1
word	DW	Word	字	2
dword	DD	DoubleWord	双字	4
fword	DF	FarWord	三字	6
qword	DQ	QuadWord	四字	8
tbyte	DT	TenBytes	五字	10

有符号整数	长度（字节）
sbyte	1
sword	2
sdword	4

实数	长度（字节）
read4	4
read8	8
read10	10

变量定义格式：

[变量名] 类型 初始值 [,初始值 1]...

注：

a、带中括号的是可以省略的项，注意汇编里变量名是可以省略的；

b、如果不想赋初值可以使用“?”，例如：“.var1 byte ?”。

初始值使用 dup 伪指令可以按照规定的次数来复制某个操作数，避免多次输入同样一个数据，格式：

[变量名] 类型 次数 dup(重复内容) [,初始值 1]...

例如：

①以 100 个?来初始化 var1
```
var1 byte 100 dup(?)
```

②以 5 个“abc”来初始化 var2，并在末尾补上 0。
```
var2 byte 5 dup("abc"),0
```
相当于：
```
var2 byte "abcabcabcabcabc",0
```

③以 10 个 dword 长度的 0 来初始化 var3
```
var3 dword 10 dup(0)
```

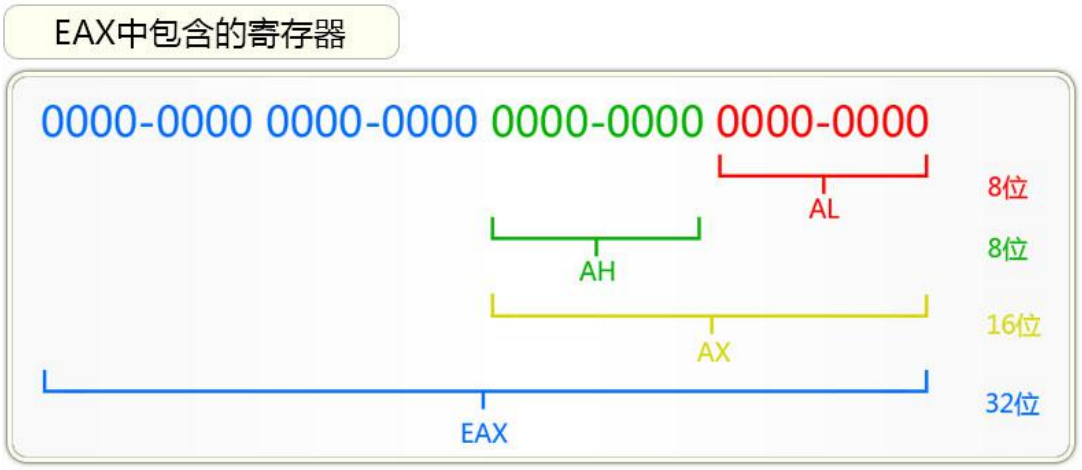
二、基础指令

对应章节：第四章：数据传送、寻址和算术运算、第七章：整数运算、第九章：字符串和数组

1、寄存器

寄存器	英文名	中文名	用途
EAX	Accumulator	累加器	函数的返回值、I/O访问、算术、中断调用
EBX	Base	基地址	内存访问、中断的返回值
ECX	Counter	计数器	循环计数器、获取一些中断值
EDX	Data	数据	I/O访问、算术、函数调用

对于上面所列出来的这四个寄存器，他们都包含了几个大小不同的寄存器：



综合一下，有下面的这些寄存器：

EAX 包含 AX、AH、AL；
EBX 包含 BX、BH、BL；
ECX 包含 CX、CH、CL；
EDX 包含 DX、DH、DL。

寄存器	英文名	中文名	用途
ESI	Source Index	源索引寄存器	字符串操作和内存数组复制
EDI	Destination Index	目标索引寄存器	字符串操作和内存数组复制
EBP	Stack Base	栈底寄存器	指向栈底的指针
ESP	Stack Top	栈顶寄存器	指向栈顶的指针

此外还有 **EIP（Instruction Pointer，指令指针寄存器）**也很常用，该寄存器中存放的是下一条将要指令的汇编指令的地址。

2、标志位

EFLAGS（标志寄存器）

下表列出了一些常用的标志位的相关信息

位置	缩写	英文	中文	=1	=0
0	CF	Carry flag	进位标志	有进位	无进位
2	PF	Parity flag	奇偶标志	偶数	奇数
4	AF	Auxiliary carry flag	辅助进位标志	有进位	无进位
6	ZF	Zero flag	零标志	等于零	非零
7	SF	Sign flag	符号标志	负	正
8	TF	Trap flag	跟踪标志	允许单步调试	禁止单步调试
9	IF	Interrupt enable flag	中断标志	响应可屏蔽中断	禁止可屏蔽中断
10	DF	Direction flag	方向标志	减少	增加
11	OF	Overflow flag	溢出标志	溢出	未溢出

在上面所列出的标志位中，CF、ZF、SF、OF 是比较重要并且常用的，最好能记住他们的中文意思以及置 1 和置 0 所代表的意思。

3、传送指令

3.1 基础传送指令

传送指令	英文	中文	用法/注意点
mov	move	传送	两操作数的尺寸要相等
movzx	extended move with zero data	全零扩展并传送	①用于无符号数 ②高位均设置为 0（即对于 32 位寄存器，其高 16 位均置为 0；对于 16 位寄存器，其高 8 位置为 0） ③操作数 2 的尺寸要比操作数 1 小
movsx	extended move with sign data	符号扩展并转送	①用于有符号数 ②对于正数，和 movzx 效果相同，对于负数，其高位均设置为 1 ③操作数 2 的尺寸要比操作数 1 小
xchg	exchange	交换	

用一句话简单说 movzx 的作用就是：将操作数 2 的高位全部用 0 来填充，然后赋值给操作数 1,。

举例（下面的横杠和空格只是为了让二进制数更有可读性）：

```
mov BL,111b
movzx AX,BL    ;AX = 0000-0000 0000-0111
```

同样也是一句话概括 movsx 的作用就是：将操作数 2 的高位全部用其符号位来填充，然后赋值给操作数 1。（即：如果操作数 2 是正数，则用 0 来填充高位，如果是负数则用 1 来填充高位）

举例：

```
mov BL,-3       ;BL = 1111-1101
movsx AX,BL     ;AX = 1111-1111 1111-1101

mov BL,3        ;BL = 0000-0011
movsx AX,BL     ;AX = 0000-0000 0000-0011
```

3.2 与标志位有关的传送指令

传送指令	英文	中文
LAHF	load AH from flag	加载状态标志位到 AH
SAHF	save AH to flag	保存 AH 内容到状态标志位

拓展：与标志位置为和清零的指令有如下几个

指令	英文	中文	作用
CLC	Clear Carry Flag	清零进位标志位	CF = 0
STC	Set Carry Flag	置进位标志位	CF = 1
CMC	Complement Carry Flag	进位标志位取反	CF = !CF
CLD	Clear Direction Flag	清零方向标志位	DF = 0
STD	Set Direction Flag	置方向标志位	DF = 1

4、算术运算

4.1 加减法

加法指令	英文	中文	用法/注意点
add	add	相加	
adc	add with carry	带进位加法	两数相加再加 CF（进位标志位）
inc	increase 1	加 1	不影响进位标志位（CF）

减法指令	英文	中文	用法/注意点
sub	substract	相减	
sbb	substract with borrow	带借位减法	两数相减再减 CF（进位标志位）
dec	decrease 1	减 1	不影响进位标志位（CF）

4.2 乘除法

约定：

“被乘数”在前，“乘数”在后，例如“5×3”，5 是被乘数，3 是乘数。

“被除数”在前，“除数”在后，例如“6÷3”，6 是被除数，3 是除数。

4.2.1 乘法

指令	英文	中文
MUL	Multiplication	无符号乘法
IMUL	Integer Multiplication	整数乘法

由于篇幅有限，此处仅列出单操作数格式的乘法，对于双操作数和三操作数格式的乘法请自行查阅相关参考资料。

使用格式：

```
MUL reg/mem8
MUL reg/mem16
MUL reg/mem32
```

注意，

a.不能使用立即数，只能使用寄存器或者内存操作数。

b.在使用 MUL 后要注意对 CF（进位标志位）进行检查，如果 CF=1 则高半部分不为 0。

MUL 的运算

被乘数	乘数	乘积
AL	reg/mem8	AX
AX	reg/mem16	DX:AX
EAX	reg/mem32	EDX:EAX

举例：

①

```
mov al,5h
mov bl,10h
mul bl      ;AX=0050h, CF=0
```

②

```
.data
val1  word  2000h
val2  word  0100h
.code
mov ax,val1         ;AX = 2000h
mul val2            ;DX:AX = 0020 0000h, CF=1
```

IMUL 指令是针对有符号数的乘法，在进行乘法操作后符号位或扩展到高半部分中，其余的与 MUL 指令类似。

4.2.2 除法

使用格式：

```
MUL reg/mem8
MUL reg/mem16
MUL reg/mem32
```

DIV 的运算

被除数	除数	商	余数
AX	reg/mem8	AL	AH
DX:AX	reg/mem16	AX	DX
EDX:EAX	reg/mem32	EAX	EDX

注：在使用 DIV 和 IDIV 的时候如果不使用到高位（DX 或 EDX），应先将被除数的高位清零。

举例：

①

```
mov  ax,0083h      ;被除数
mov  bl,2          ;除数
div  bl            ;AL = 41h,  AH = 01h
```

②

```
mov dx,0           ;清除被除数高 16 位
mov ax,8003h       ;被除数的低 16 位
mov cx,100h        ;除数
div cx             ;AX = 0080h，DX = 0003h
```


IDIV 是针对有符号的除法，在用法上与 DIV 指令只有一个重要区别：在执行除法之前，必须对被除数进行符号扩展（**符号扩展是指将一个数的最高位复制到包含该数的变量或寄存器的所有高位中**）。

5、移位

指令	英文	中文	用法/注意点
SHL	shift left	左移	各个位向左移动一位，最高位存入 CF，最低位补零
SHR	shift right	右移	各个位向右移动一位，最高位补零，最低位存入 CF
SAL	arithmatic shift left	算术左移	符号位不动；其余位向左移动一位，其中最高位存入 CF，最低位补零
SAR	arithmatic shift right	算术右移	符号位不动；其余位向右移动一位，其中最高位补零，最低位存入 CF

ROL	rotate left	循环左移	各个位向左移动一位，最高位存入 CF，最低位用先前的最高位填充
ROR	rotate right	循环右移	各个位向右移动一位，最低位存入 CF，最高位用先前的最低位填充
RCL	rotate left with carry	带进位的循环左移	将 CF 合并到操作数的最高位，然后类似 ROL
RCR	rotate right with carry	带进位的循环右移	将 CF 合并到操作数的最低位，然后类似 ROR

6、常用伪指令

伪指令	中文	作用
offset	取偏移	与 lea 指令作用相同，返回数据标号的偏移量
align	对齐	将一个变量对齐到字节边界、字边界、双字边界或段落边界
ptr	指针	转换变量的长度
type	取类型长度	获取某种类型的长度（字节）
lengthof	取元素个数	获取某个变量元素的个数
sizeof	取变量长度	获取某个变量所占的字节数
lable	标签	常用于用两个较小的整数组成一个较大的整数

示例：

①offset

```
.data
msg byte "Hello World!",0
.code
mov edx,offset msg
call WriteString
```

②align

下述例子中 bVal 处于任意位置，但其偏移量为 00404000h。

```
bVal byte ?       ;00404000h
align 2
wVal word ?       ;00404002h
bVal2 byte ?      ;00404004h
align 4
dVal dword ?      ;00404008h
dVal2 dowd ?      ;0040400Ch
```

③ptr

```
.data
myDouble dword 12345678h
.code
mov ax,word ptr myDouble
```

④type

```
.data
var1 byte ?
var2 word ?
var3 dword ?
var4 qword ?
.code
mov eax,type var1    ;eax = 1
mov ebx,type var2    ;ebx = 2
mov ecx,type var3    ;ecx = 4
mov edx,type var4    ;edx = 8
```

⑤lengthof

```
.data
byte1      byte   10,20,30
array1     wrod   30 dup(?),0,0
array2     wrod   5 dup(3 dup(?))
array3     dwrod  1,2,3,4
digitStr   byte   “123456789”,0
.code
mov eax,lengthof byte1      ;eax = 3
mov eax,lengthof array1     ;eax = 30 + 2 = 32
mov eax,lengthof array2     ;eax = 5*3    = 15
mov eax,lengthof array3     ;eax = 4
mov eax,lengthof digitStr   ;eax = 9
```

⑥sizeof

```
.data
intArray word 32 dup(0)
.code
mov eax,sizeof intarray     ;eax = 64
```

⑦label

示例 1：

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

示例 2：

```
.data
LongValue label dword
```

```
val1 word 5678h
val2 word 1234h
.code
mov eax,LongValue     ;EAX = 12345678h
```

注：label 伪指令自身不分配内存。

7、寻址方式

7.1 立即寻址方式

操作数直接包含在指令中，紧跟在操作码之后的寻址方式称为立即寻址方式，把该操作数称为立即数。

MOV AL, **6**

MOV AX, **3064H**

7.2 寄存器寻址方式

操作数直接包含在寄存器中,由指令指定寄存器号的寻址方式。

MOV **BX**, **AX**

MOV **DI**, 5678H

7.3 直接寻址方式

操作数的有效地址直接包含在指令中。

MOV EAX, **[78H]**

7.4 寄存器间接寻址方式

操作数的有效地址在寄存器中，而操作数在存储器中。

MOV EAX, **[EBX]**

7.5 寄存器相对寻址方式

操作数的有效地址是一个寄存器和一个位移量之和。

MOV EAX, **8[EBX]** 或 MOV AL, **[EBX＋8]**

7.6 基址变址寻址方式

操作数的有效地址是一个基址寄存器和一个变址寄存器的内容之和。

MOV AL,**[EBX][ESI]** 或 MOV EAX,**[EBX＋ESI]**

7.7 相对基址变址寻址方式

操作数的有效地址是**两个寄存器和一个基址之和**。（通常这两个寄存器一个作为变址寄存器另一个为位移量）

AL,**MASK[BX][SI]** 或 MOV AL,**MASK[BX＋SI]** 或 MOV AL,**[MASK＋BX＋SI]**

注：上面指令中“MASK”是一个立即数，可用来表示基址。

7.8 比例变址寻址方式

操作数的有效地址是**一个基址和一个变址寄存器的内容和指令中给定的一个位移量之和**（操作数地址=[基址寄存器]＋[变址寄存器]×比例因子＋位移量）。简单点说，有比例因子的就是比例变址寻址方式。

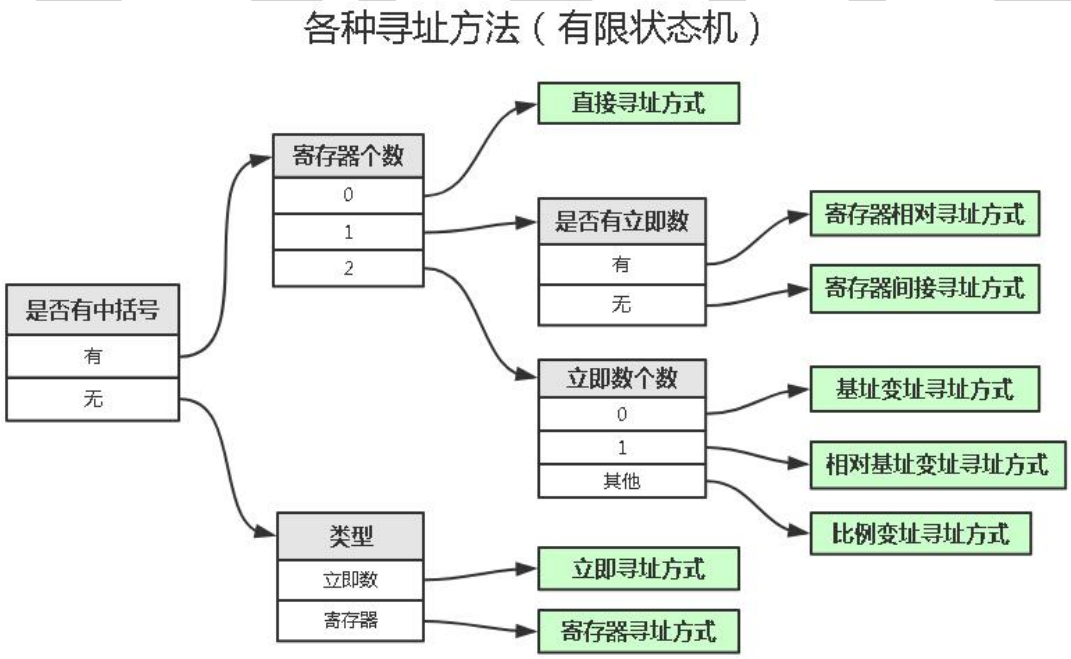
MOV AX, **ARY[BX][4*SI]**

注：上面指令中 ARY 是一个立即数。

7.9 各个寻址方式对比

寻址方式	示例	特征
立即寻址方式	MOV AL,**6**	操作数是立即数
寄存器寻址方式	MOV **DI**,5678H	操作数是寄存器
直接寻址方式	MOV EAX,**[78H]**	中括号、立即数
寄存器间接寻址方式	MOV EAX,**[EBX]**	中括号、一个寄存器
寄存器相对寻址方式	MOV AL,**[EBX＋8]**	中括号、一个寄存器、一个立即数
基址变址寻址方式	MOV EAX,**[EBX＋ESI]**	中括号、两个寄存器
相对基址变址寻址方式	MOV AL,**[MASK＋EBX＋ESI]**	中括号、两个寄存器、一个立即数
比例变址寻址方式	MOV AX,**ARY[EBX][4*ESI]**	有比例因子

7.10 拓展：利用有限状态机记忆寻址方式



8、二维数组

暂不总结！

9、字符串基本指令

指令	中文	功能
MOVSB、MOVSW、MOVSD	传送字符串数据	将 ESI 寻址的内存数据复制到 EDI 寻址的内存位置
CMPSB、CMPSW、CMPSD	比较字符串	比较分别由 ESI 和 EDI 寻址的内存数据
SCASB、SCASW、SCASD	扫描字符串	比较累加器（AL、AX、EAX）与 EDI 寻址的内存数据
STOSB、STOSW、STOSD	保存字符串数据	将累加器内容保存到 EDI 寻址的内存位置
LODSB、LODSW、LODSD	从字符串加载到累加器	将 ESI 寻址的内存数据加载到累加器

字符串基本指令只能处理一个或一对内存数值。如果记上重复前缀，指令就可以用 ECX 作计数器重复执行。重复前缀使得单条指令能够处理整个数组。下面为可用的重复前缀：

重复前缀	功能
REP	ECX > 0 时重复
REPZ、REPE	ZF = 1 且 ECX > 0 时重复
REPNZ、REPNE	ZF = 0 且 ECX > 0 时重复

根据方向标志位的状态，字符串基本指令增加或减少 ESI 和 EDI 可以使用 CLD 和 STD 指令显式修改 CF（方向标志位）的值：

DF 的值	对 ESI 和 EDI 的影响	地址顺序
0	增加	低到高
1	减少	高到低

示例：

①复制字符串

```
.data
string1 byte "123456789",0
string2 byte 10 dup(0)
.code
cld                         ;清除方向标志
mov esi,offset string1      ;ESI 指向源串
mov edi,offset string2      ;EDI 指向目的串
mov ecx,10                  ;计数器赋值为 10
rep movsb                   ;传送 10 个字节
```

②扫描是否有匹配字符

```
.data
alpha byte "ABCDEFGH",0
.code
mov edi,offset alpha     ;EDI 指向字符串
mov al,'F'               ;检索字符 F
mov ecx,lengthof alpha   ;设置检索计数器
cld                      ;方向为正向
repne scasb              ;不想动则重复
jnz quit                 ;若未发现字符则退出
dec edi                  ;发现字符：EDI 减 1
```

三、分支与循环

对应章节：第六章：条件处理

1、布尔和比较

逻辑运算符	中文	用法/注意点
and	按位与	按位进行运算
or	按位或	
not	按位非	
xor	异或	①“相异为 1”，两个二进制数对应位置，若不相同则该位的运算结果为 1。 ②常用于寄存器的清零操作，例如：xor eax,eax
neg	求补	可用于有符号数的取相反数
test	测试（按位与）	与 and 类似，但不修改操作数，仅影响标志位

2、跳转

方法：记住那几个字母的英文，然后就很容易记住各种跳转指令了。

2.1 类型一：无条件跳转以及针对数值的条件跳转

2.2 类型二：针对寄存器的条件跳转

		英文	中文	第一列跳转时机
JZ	JNZ	Z:zero	零	ZF=1（等于零）
JC	JNC	C:carry	进位	CF=1（有进位）
JO	JNO	O:overflow	溢出	OF=1（有溢出）
JS	JNS	S:sign	符号	SF=1（负数）
JP	JNP	P:parity	奇偶	PF=1（偶数）

3、循环

循环指令	功能
loop	CX 减 1,CX 不等于 0,则转移至标号处循环执行， 直至(CX)=0,继续执行后继指令.
loope/loopz	当 ZF=1 时,CX/ECX 减 1,当 CX/ECX 不等于 0 时,跳转到标号 Label 指定的目的操作数,否则执行下一条指令.
loopne/loopnz	当 ZF=0 时,CX/ECX 减 1,当 CX/ECX 不等于 0 时,跳转到标号 Label 指定的目的操作数,否则执行下一条指令.

4、If...else 、while 、fo...while 、for

注：这几个高级语言翻译成汇编代码结果不唯一，此处仅给出其中一种结果的参考。

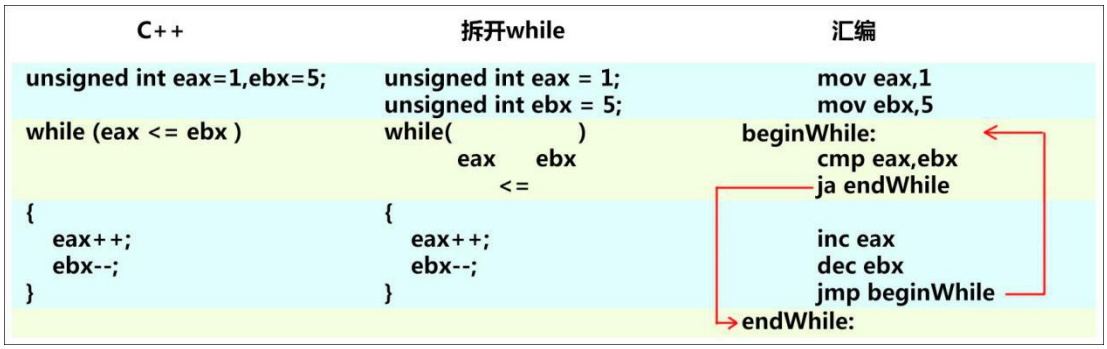
4.1 if...else

C++	拆开if	汇编
unsigned int eax=1,ebx=2;	unsigned int eax = 1; unsigned int ebx = 2;	mov eax,1 mov ebx,2
if(eax > ebx)	if (eax > ebx) ——取反变“≤”即“jbe”——>	cmp eax,ebx jbe L1

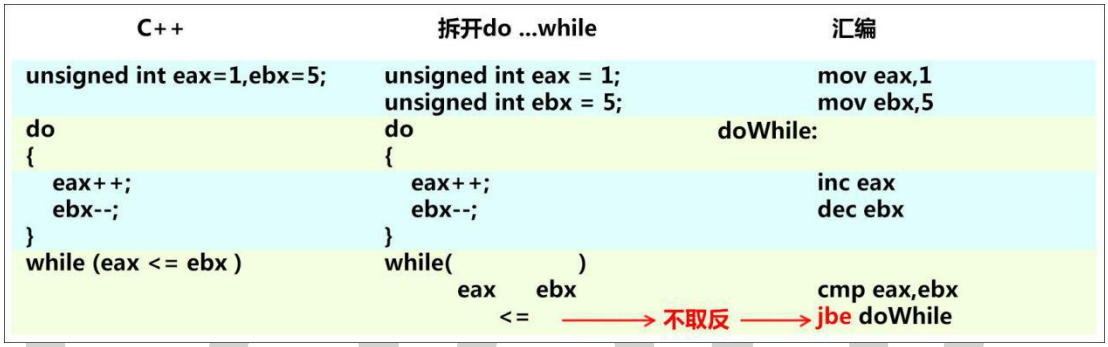
{
eax++;
ebx--;
}
	{
	eax++;
	ebx--;
	}
		inc eax
		dec ebx
		jmp L2

else
{
eax--;
ebx++;
}
	else
	{
	eax--;
	ebx++;
	}
		L1:
		dec eax
		inc ebx
		L2:

4.2 while

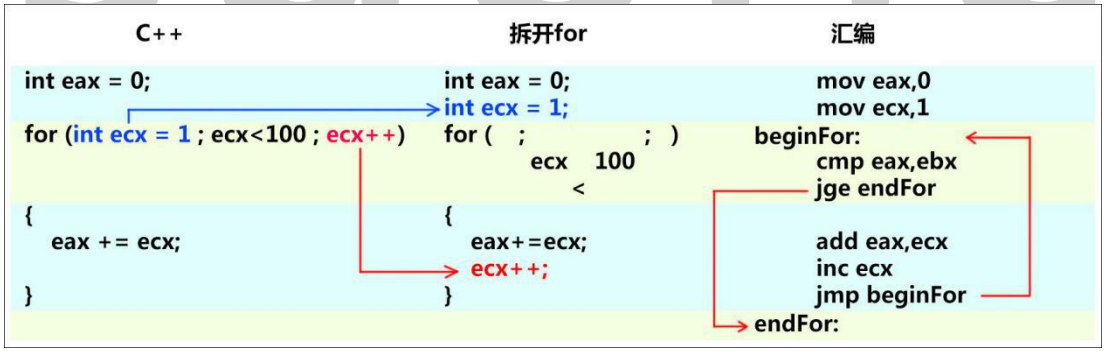


4.3 do...while



4.4 for

思路：将 for 语句按照语法修改成 while 循环即可。



四、过程

对应章节：第五章：过程、第八章：高级过程

1、出入栈

入栈	出栈	英文	用法/注意点
push	pop	push/pop	出/入栈
pushf	popf	push/pop flags	将 16 位标志寄存器 FLAGS 出/入栈
pushfd	popfd	push/pop eflags	将 32 位标志寄存器 EFLAGS 出/入栈
pusha	popa	push/pop all	将 8 个 16 位通用寄存器出/入栈 （入栈顺序：AX,CX,DX,BX,SP,BP,SI,DI）
pushad	popad	push/pop all data	将 8 个 32 位通用寄存器出/入栈 （入栈顺序：EAX,ECX,EDX,EBX,ESP,EBP,ESI,EDI）

2、堆栈帧

堆栈帧创建步骤：
(1)参数 push 入栈，如果没有参数则跳过
(2)当子程序被调用时，使该子程序的返回值入栈
(3)子程序开始执行时，EBP 入栈
(4)EBP = ESP，EBP 变成该子程序所有参数的引用基址
(5)如果有局部变量，修改 ESP，为变量预留空间
(6)如果要保存寄存器，将它们也入栈

3、lea、enter、leave、local、uses、invoke

指令/伪指令/运算符	功能
lea	返回数据标号的偏移量
enter	创建堆栈帧
leave	结束堆栈帧
local	声明一个或多个变量名，并定义其大小属性
uses	在过程开始前对使用寄存器进行保存，然后在过程结束后恢复之前保存的寄存器的值
invoke	call 指令一个方便的替代品

3.1 lea

```
.data
msg byte "Hello World!",0
.code
lea edx,msg
call WriteString
```

3.2 enter

语法：

enter *numbytes,nestinglevel*

numbytes ：**局部变量保存的堆栈空间字节数**
nestinglevel ：**词法嵌套等级**

示例：
①一个没有局部变量的过程

```
MySub proc
    enter 0,0
```

等效于：

```
MySub proc
    push ebp
    mov ebp,esp
```

②enter 为局部变量保留了 8 个字节的堆栈空间

```
MySub proc
    enter 8,0
```

等效于：

```
MySub proc
    push ebp
    mov ebp,esp
    sub esp,8
```


注：如果使用 enter 指令，建议在同一个过程的结尾处同时使用 leave 指令。否则，为局部变量保留的堆栈空间就可能无法释放。这将会导致 ret 指令从堆栈中弹出错误的返回地址。

3.3 leave

Leave 指令结束一个过程的堆栈帧，它反转了之前的 enter 指令操作；恢复了过程被调用时 esp 和 ebp 的值。

用法；

```
MySub proc
    enter 8,0
    ...
    leave
    ret
MySub endp
```

上面的指令等效于

```
MySub proc
    push ebp
    mov ebp,esp
    sub esp,8
    ...
    mov esp,ebp
    pop ebp
    ret
MySub endp
```

3.4 local

Local 伪指令是作为 enter 指令的高级替补，local 声明一个或多个变量名，并定义其大小属性。如果要使用 Local 伪指令，它必须紧跟在 proc 伪指令的后面。

语法：

Local *varlist*

varlist ：变量定义列表，用逗号分隔表项，可选为跨越多行。每个变量定义采用如下格式：

label : *type*

其中，*label* 可以为任意有效标识符，*type* 既可以是标准类型（word、dword 等），也可以是用户定义类型。

示例：

①

```
MySub proc
    local var1:byte
```

②

```
MySub proc
    local var1:dword,var2:byte
```

③

```
MySub proc
    loacl var1:ptr word
```

④

```
MySub proc
    local arr[10]:dword
```

3.5 uses

Uses 让汇编器做两件事情：第一，在过程开始时生成 push 指令，将寄存器保存到堆栈第二，在过程结束时生成 pop 指令，从堆栈恢复寄存器的值。

示例：

```
ArraySum proc uses  esi,ecx
    ...
    ret
ArraySum endp
```

上面的代码等效于：

```
ArraySum proc
    Push esi
    Push ecx
    ...
    Pop ecx
    Pop esi
    ret
ArraySum endp
```


3.6 invoke

Invoke 伪指令，将参数入栈并调用过程。Invoke 是 call 指令一个方便的替代品，因为，他用一行代码就能传递过个参数。

语法：

Invoke *procedureName [, argumentList]*

ArgumentList 是可选项，它用逗号分隔传递给过程的参数。

示例：

```
Invoke DumpArray, OFFSET array, LENGTHOF array, TYPE array
```

上面的代码等效于：

```
push   type array
push   lengthof array
push   offset array
call   DumArray
```

4、访问堆栈参数

使用**基址 - 偏移量寻址**方式来访问堆栈参数。其中，ebp 是基址寄存器，偏移量是常数。

示例：

```
AddTwo proc
    push ebp
    mov ebp,esp             ;堆栈帧的基址
    mov eax,[ebp + 12]      ;第二个参数
    add eax,[ebp + 8]       ;第一个参数
    pop ebp
    ret
AddTwo endp
```

注：

a. 上面代码中的“[ebp + 12]”和“[ebp + 8]”就是使用了**基址 - 偏移量寻址**的方式来访问堆栈参数。

b. 在 32 位汇编中，访问参数至少要加 8 个字节，因为[ebp]和[ebp+4]分别存放的是 ebp 的值以及 call 调用返回的地址。

5、局部变量

示例（C 调用规则）：

```
MySub proc
    push    ebp
    mov     ebp,esp
    sub     esp,8                       ;创建局部变量
    mov     dword ptr [ebp - 4],10
    mov     dword ptr [ebp - 8],20
    mov     esp,ebp
    pop     ebp
    ret
MySub endp
```


注：

a. 在 32 位汇编中，给局部变量分配空间需要是 4 的倍数。

b. 对于局部变量的访问，也是使用**基址 - 偏移量寻址**方式。

c. 相对于访问堆栈参数，**局部变量是 ebp 减去某个值，而访问堆栈参数是加上某个值。**

6、函数调用规范

语言调用规则	参数入栈顺序	平衡堆栈	主要用途
C	右到左	调用者	C 程序
stdcall	右到左	子程序	API

注：写纯汇编程序建议使用 stdcall，如果需要 C/C++和汇编混合编程建议使用 C 调用规则。

4.1 stdcall 调用示例

```
Example proc
    push    ebp
    mov     ebp,esp     ;堆栈帧基址
    mov     eax,[ebp+12]  ;第二个参数
    add     eax,[ebp+8]   ;第一个参数
    pop     ebp
    ret     8       ;清除堆栈
Example endp
```

4.2 C 调用示例

注：假设有一个函数为：AddTwo(A,B)

```
Example proc
    Push    6
    Push    5
    Call    AddTwo
    Add     esp,8   ;从堆栈移除参数
Example endp
```

要点：从上面的代码可以看出，C 调用是由**调用者**来进行平衡堆栈的。

五、其他

1、一个最基础的汇编程序框架

```
.386
.model flat,stdcall
ExitProcess PROTO,dwExitCode:dword
.data
.code
main proc

    ;你的代码

    invoke ExitProcess,0
main endp
end main
```

注：如果你的代码要使用 Irvine 库，请在“.model”的下一行加上“include Irvine32.inc”

2、零散知识点

① INC、DEC 不影响 CF（进位标志位）。

② MOV 不能修改 EIP，要修改 EIP 通常使用跳转指令或 call、ret 指令。

③ Irvine 库常用函数

函数名	功能	输入	返回
Crlf	换行	无	无
WriteString	输出字符串	EDX = 字符串偏移量	无
ReadDec	读无符号整数	无	EAX = 数字
ReadInt	读有符号整数	无	EAX = 数字
WriteInt	输出有符号整数	EAX = 数字	无
WriteDec	输出无符号整数	EAX = 数字	无

④ 在对二进制数进行乘 2 或除 2 的时候，使用左移或右移更高效便捷。

⑤ 符号扩展是指将一个数的最高位复制到包含该数的变量或寄存器的所有高位中，例如 movsz、imul、idiv 指令都有有使用到符号扩展。

⑥ 汇编语言指令不区分大小写。