本小节内容

汇编常用指令讲解

汇编常用指令讲解

3.1 相关寄存器

通用寄存器

31 16	15 8	7 0
	AH	AL
	BH	BL
	CH	CL
	DH	DL
ESI		
EDI		
EBP		
ESP		

16bit	32bit	说明
AX	EAX	累加器（Accumulator）
BX	EBX	基地址寄存器（Base Register）
CX	ECX	计数寄存器（Count Register）
DX	EDX	数据寄存器（Data Register）
	ESI	变址寄存器（Index Register）
	EDI	
	EBP	堆栈基指针（Base Pointer）
	ESP	堆栈顶指针（Stack Pointer）

除 EBP 和 ESP 外，**其他几个寄存器的用途是比较任意的，也就是什么都可以存。**

3.2 常用指令

汇编指令通常可以分为数据传送指令、逻辑计算指令和控制流指令，下面以 Intel 格式为例(考研考的就是 Intel 的汇编)，介绍一些重要的指令。以下用于操作数的标记分别表示寄存器、内存和常数。

- <reg>：表示任意寄存器，若其后带有数字，则指定其位数，如<reg32>表示 32 位寄存器（eax、ebx、ecx、edx、esi、edi、esp 或 ebp）；<reg16>表示 16 位寄存器（ax、bx、cx 或 dx）；<reg8>表示 8 位寄存器（ah、al、bh、bl、ch、cl、dh、dl）。
- <mem>：表示内存地址（如[eax]、[var+4]或 dword ptr [eax+ebx]）。
- <con>：表示 8 位、16 位或 32 位常数。<con8>表示 8 位常数；<con16>表示 16 位常数；<con32>表示 32 位常数。（也称为立即数）

（1）数据传送指令

1）**mov 指令**。将第二个操作数（寄存器的内容、内存中的内容或常数值）复制到第一个操作数（寄存器或内存）。但不能用于直接从内存复制到内存。

其语法如下：

```
mov <reg>,<reg>
mov <reg>,<mem>
mov <mem>,<reg>
mov <reg>,<con>
mov <mem>,<con>
```

举例：

```
mov eax, ebx                #将 ebx 值复制到 eax
mov byte ptr [var], 5       #将 5 保存到 var 值指示的内存地址的一字节中
```

2）**push 指令**。将操作数压入内存的栈，常用于函数调用。ESP 是栈顶，压栈前先将 ESP 值减 4（栈增长方向与内存地址增长方向相反），然后将操作数压入 ESP 指示的地址。

其语法如下：

```
push <reg32>
push <mem>
push <con32>
```

举例（注意，栈中元素固定为 32 位）：

```
push eax                    #将 eax 值压栈
push [var]                  #将 var 值指示的内存地址的 4 字节值压栈
```

3）**pop 指令**。与 push 指令相反，pop 指令执行的是出栈工作，出栈前先将 ESP 指示的地址中的内容出栈，然后将 ESP 值加 4。

其语法如下：

```
pop edi                     #弹出栈顶元素送到 edi
pop [ebx]                   #弹出栈顶元素送到 ebx 值指示的内存地址的 4 字节中
```

（2）算术和逻辑运算指令

1）**add/sub 指令**。add 指令将两个操作数相加，相加的结果保存到第一个操作数中。sub 指令用于两个操作数相减，相减的结果保存到第一个操作数中。

它们的语法如下：

```
add <reg>,<reg> / sub <reg>,<reg>
add <reg>,<mem> / sub <reg>,<mem>
add <mem>,<reg> / sub <mem>,<reg>
add <reg>,<con> / sub <reg>,<con>
add <mem>,<con> / sub <mem>,<con>
```

举例：

```
sub eax, 10                 #eax ← eax-10
add byte ptr [var], 10      #10 与 var 值指示的内存地址的一字节值相加，并将结果
                             保存在 var 值指示的内存地址的字节中
```

2）**inc/dec 指令**。inc、dec 指令分别表示将操作数自加 1、自减 1。

它们的语法如下：

```
inc <reg> / dec <reg>
inc <mem> / dec <mem>
```

举例：

```
dec eax                     #eax 值自减 1
inc dword ptr [var]         #var 值指示的内存地址的 4 字节值自加 1
```

3）**imul 指令**。带符号整数乘法指令，有两种格式：①两个操作数，将两个操作数相乘，将结果保存在第一个操作数中，第一个操作数必须为寄存器；②三个操作数，将第二个和第三个操作数相乘，将结果保存在第一个操作数中，第一个操作数必须为寄存器。

其语法如下：

```
imul <reg32>,<reg32>
imul <reg32>,<mem>
imul <reg32>,<reg32>,<con>
imul <reg32>,<mem>,<con>
```

举例：

```
imul eax, [var]             #eax ← eax * [var]
imul esi, edi, 25           #esi ← edi * 25
```

乘法操作结果可能溢出，则编译器置溢出标志 OF = 1，以使 CPU 调出溢出异常处理程序。

4）**idiv 指令。**带符号整数除法指令，它只有一个操作数，即除数，而被除数则为 edx:eax 中的内容（64 位整数），操作结果有两部分：商和余数，商送到 eax，余数则送到 edx。

其语法如下：

```
idiv <reg32>
```

```
idiv <mem>
```

举例：

```
idiv ebx
idiv dword ptr [var]
```

5）**and/or/xor 指令**。and、or、xor 指令分别是按位与、按位或、按位异或操作指令，用于操作数的位操作(按位与，按位或，异或)，操作结果放在第一个操作数中。

它们的语法如下：

```
and <reg>,<reg> / or <reg>,<reg> / xor <reg>,<reg>
and <reg>,<mem> / or <reg>,<mem> / xor <reg>,<mem>
and <mem>,<reg> / or <mem>,<reg> / xor <mem>,<reg>
and <reg>,<con> / or <reg>,<con> / xor <reg>,<con>
and <mem>,<con> / or <mem>,<con> / xor <mem>,<con>
```

举例：

```
and eax, 0fH            #将 eax 中的前 28 位全部置为 0，最后 4 位保持不变
xor edx, edx            #置 edx 中的内容为 0
```

6）**not 指令。**位翻转指令，将操作数中的每一位翻转，即 0→1、1→0。

其语法如下：

```
not <reg>
not <mem>
```

举例：

```
not byte ptr [var]      #将 var 值指示的内存地址的一字节的所有位翻转
```

7）**neg 指令**。取负指令。

其语法如下：

```
neg <reg>
neg <mem>
```

举例：

```
neg eax                     #eax ← -eax
```

8）**shl/shr 指令**。逻辑移位指令，shl 为逻辑左移，shr 为逻辑右移，第一个操作数表示被操作数，第二个操作数指示移位的位数。

它们的语法如下：

```
shl <reg>,<con8> / shr <reg>,<con8>
shl <mem>,<con8> / shr <mem>,<con8>
shl <reg>,<cl> / shr <reg>,<cl>
shl <mem>,<cl> / shr <mem>,<cl>
```

举例：

```
shl eax, 1              #将 eax 值左移 1 位，相当于乘以 2
shr ebx, cl             #将 ebx 值右移 n 位（n 为 cl 中的值），相当于除以 2^n
```

9）lea 指令。地址传送指令，将有效地址传送到指定的的寄存器。

```
lea  eax, DWORD PTR _arr$[ebp]
```

lea 指令的作用，是 DWORD PTR _arr$[ebp]对应空间的内存地址值放到 eax 中

（3）控制流指令

x86 处理器维持着一个指示当前执行指令的指令指针（IP），当一条指令执行后，此指针自动指向下一条指令。IP 寄存器不能直接操作，但可以用控制流指令更新。通常用标签 (label) 指示程序中的指令地址，在 x86 汇编代码中，可在任何指令前加入标签。例如，

```
        mov esi, [ebp+8]
begin:  xor ecx, ecx
        mov eax, [esi]
```

这样就用 begin（begin 代表标签名，可以为别的名字）指示了第二条指令，控制流指令通过

标签就可以实现程序指令的跳转。

1）**jmp 指令。**jmp 指令控制 IP 转移到 label 所指示的地址（从 label 中取出指令执行）。

其语法如下：

```
jmp <label>
```

举例：

```
jmp begin            #转跳到 begin 标记的指令执行
```

2）***j*condition 指令**。条件转移指令，依据 CPU 状态字中的一系列条件状态转移。CPU 状态字中包括指示最后一个算术运算结果是否为 0，运算结果是否为负数等。

其语法如下：

```
je <label> (jump when equal)
jne <label> (jump when not equal)
jz <label> (jump when last result was zero)
jg <label> (jump when greater than)
jge <label> (jump when greater than or equal to)
jl <label> (jump when less than)
jle <label> (jump when less than or equal to)
```

举例：

```
cmp eax, ebx
jle done  #如果 eax 的值小于等于 ebx 值，跳转到 done 指示的指令执行，否则执行下一
          条指令。
```

3）**cmp/test 指令**。cmp 指令用于比较两个操作数的值， test 指令对两个操作数进行逐位与运算，这两类指令都不保存操作结果，仅根据运算结果设置 CPU 状态字中的条件码。

其语法如下：

```
cmp <reg>,<reg> / test <reg>,<reg>
cmp <reg>,<mem> / test <reg>,<mem>
cmp <mem>,<reg> / test <mem>,<reg>
cmp <reg>,<con> / test <reg>,<con>
```

cmp 和 test 指令通常和 *j*condition 指令搭配使用，举例：

```
cmp dword ptr [var], 10   #将 var 指示的主存地址的 4 字节内容，与 10 比较
jne loop                  #如果相等则继续顺序执行；否则跳转到 loop 处执行
test eax, eax             #测试 eax 是否为零
jz xxxx                   #为零则置标志 ZF 为 1，转跳到 xxxx 处执行
```

4）**call/ret 指令。**分别用于实现子程序（过程、函数等）的调用及返回。

其语法如下：

```
call <label>
ret
```

call 指令首先将当前执行指令地址入栈，然后无条件转移到由标签指示的指令。与其他简单的跳转指令不同，call 指令保存调用之前的地址信息（当 call 指令结束后，返回调用之前的地址）。ret 指令实现子程序的返回机制，ret 指令弹出栈中保存的指令地址，然后无条件转移到保存的指令地址执行。call 和 ret 是程序（函数）调用中最关键的两条指令。

3.3 条件码

编译器通过条件码（标志位）设置指令和各类转移指令来实现程序中的选择结构语句。

（1）条件码（标志位）

除了整数寄存器，CPU 还维护着一组条件码（标志位）寄存器，它们描述了最近的算术或逻辑运算操作的属性。可以检测这些寄存器来执行条件分支指令，最常用的条件码有：

- **CF**：进（借）位标志。最近无符号整数加（减）运算后的进（借）位情况。有进（借）位，CF=1；否则 CF=0。如 (unsigned) t < (unsigned) a ，因为判断大小是相

减。

- **ZF**：零标志。最近的操作的运算结算是否为 0。若结果为 0，ZF=1；否则 ZF=0。如（t == 0）。
- **SF**：符号标志。最近的带符号数运算结果的符号。负数时，SF=1；否则 SF=0。
- **OF**：溢出标志。最近带符号数运算的结果是否溢出，若溢出，OF=1；否则 OF=0。

可见，**OF 和 SF 对无符号数运算来说没有意义，而 CF 对带符号数运算来说没有意义。**

如何判断溢出，简单的就是正数相加变负数为溢出，负数相加变正数溢出，但是考研不这么考，考研往往给你十六进制的两个数考溢出，通过如下手法判断即可。

- 数据高位进位，符号位进位未进位，溢出。
- 数据位高位未进位，符号位进位，溢出。
- 数据位高位进位，符号位进位，不溢出。
- 数据位高位未进位，符号位未进位，不溢出。

简单一句话就是数据位高位和符号位高位进位不一样的时候会溢出!

常见的算术逻辑运算指令（add、sub、imul、or、and、shl、inc、dec、not、sal 等）会设置条件码。但有两类指令只设置条件码而不改变任何其他寄存器，即 **cmp 和 test** 指令，**cmp 指令和 sub 指令的行为一样，test 指令与 and 指令的行为一样，但它们只设置条件码，而不更新目的寄存器。**

3.2 中的控制流指令中的 J*condition* 条件转移指令，就是根据条件码 ZF 和 SF 来实现转跳。

例如 后面小节代码转汇编实战中的 if 比较，就是通过 SF 为 1，来判断跳转。

注意：乘法溢出后，可以跳转到“溢出自陷指令”，例如 int 0x2e 就是一条自陷指令，但是考研只需要掌握溢出，可以跳转到“溢出自陷指令”即可，不需要记自陷指令有哪些。