

## 1. 32 位系统下的远指针

`mov eax, ds:[ebx]`; 这里的 `ebx` 表示 32 位偏移地址  
在 32 位系统下, 由于偏移地址的范围是  $[0, 2^{32}-1]$ , 即可以在段地址不变只变化偏移地址的情况下可以访问 4G 空间, 所以段寄存器 `cs=ds=es=ss` 并且是不变的。因此在 C 语言中, 平时接触到指针均为近指针。

```
char *p;  
short int *q;  
long int *r;  
double *s;
```

对上述变量求 `sizeof()`, 则 `sizeof(p)`、`sizeof(q)`、`sizeof(r)`、`sizeof(s)` 均等于 4。

32 位汇编中, 远指针是指 16 位段地址+32 位偏移地址。  
若变量 `p` 定义成 48 位的远指针, 则它的类型修饰为:

```
fword ptr
```

例如:

```
main:
```

```
    jmp begin
```

```
p dd 12345678h; p 里面存放了一个 48 位的远指针
```

```
    dw 18h          ; 18h:12345678h
```

```
begin:
```

```
    les ebx, fword ptr cs:[p]; es=18h
```

```
                                ; ebx=12345678h
```

```
    jmp fword ptr cs:[p]; jmp 18h:12345678h
```

在 32 位系统下, 16 位的段地址不再通过后面补一个十六进制的 0 来得到段首地址, 而是要通过查表得到段首地址。

这里提到的表称为全局描述符表 (Global Descriptor Table), 简称 `gdt` 表。`gdt` 表其实是一个数组, 该数组的首地址存放在 `gdtr` 寄存器内, 数组中每个元素的宽度均为 8 字

节。

设 gdt 表首地址为 t，从地址 t 开始存放以下数据：

t+0      xx,xx,xx,xx,xx,xx,xx,xx

t+8      xx,xx,xx,xx,xx,xx,xx,xx

t+10h   xx,xx,00h,00h,00h,xx,xx,80h

t+18h   xx,xx,40h,30h,20h,xx,xx,10h

把段地址 18h 与 t 相加得到 gdt 表内第 3 个元素的偏移，再取出 t+18h 指向的元素的值 (共 8 字节)，其中第 2、3、4、7 个字节逆向排列得到段首地址=10203040h。把段首地址与偏移地址 12345678h 相加就得到了物理地址：

18h:12345678h = 10203040h + 12345678h

逻辑地址

物理地址

8 字节中余下的 4 个 xx 计 32 位，其中 20 位用来表示段长度 (长度单位可以是字节也可以是页，其中 1 页=4K)，剩余的 12 位中有一部分位用来表示段的 ring 级别 (0、1、2、3 共 4 级) 及权限 (读 Read、写 Write、执行 eXecute)。

系统代码与用户代码用 ring 分级：其中系统代码是 ring0，而用户代码是 ring3。ring 级别保存在 cs 低 2 位中。当用户代码要访问某个段如 18h 时，cpu 会检查用户代码的 (cs & 0x0003) 是否小于等于 18h 对应段描述中的 ring 级别，若条件为真则允许访问，否则拒绝访问。

保护模式 (段有权限) ↔ 实模式 (段无权限)

## 2. PUSHF, POPF

把标志寄存器 FL 压入堆栈/弹出堆栈

不能写成 push FL , pop FL

寄存器 FL 及 IP 在编程时都是不能直接引用的：

mov ip, ax;\

`mov ax, ip; \ 都是错误的`

`mov bx, fl; /`

`mov fl, bx; /`

如何间接改变 ip:

`cs:0123 jmp 1000h; ip=1000h`

`...`

`cs:1000 call 1234h; ip=1234h`

`cs:1003 mov ah, 4Ch`

`cs:1005 int 21h`

`cs:1234 ...`

`cs:1235 ret; ip=1003h`

标志寄 FL

`stc` 指令可以使 CF=1;

`clic` 指令可以使 CF=0;

`cld` 指令可以使 DF=0;

`std` 指令可以使 DF=1;

`cli` 指令可以使 IF=0;

`sti` 指令可以使 IF=1;

要让 TF=1

`ODIT SZ xAxPxC`

`pushf`

`pop ax; AX=FL`

`or ax, 100h; 1 0000 0000B`

`push ax`

`popf; FL=AX`

`sub ax, ax; ZF=1`

`or ax, 1; ZF=0`

但并不存在指令 `clt`、`stt` 来改变 **TF** 的值。要改变 **TF** 的值必须通常 `pushf` 和 `popf` 实现：

```
pushf
```

```
pop ax; AX=FL
```

```
or ax, 100000000B; 或 or ax, 100h
```

```
push ax
```

```
popf; FL=AX, 其中第 8 位即 TF=1
```

```
...
```

```
pushf
```

```
pop ax; AX=FL
```

```
and ax, not 100000000B; 或 and ax, 0FEFFh
```

```
push ax
```

```
popf; FL=AX, 其中第 8 位即 TF=0
```

在常数表达式中除了 `+` `-` `*` `/` 运算符外，还可以使用 `not`, `or`, `and`, `xor` 等运算符。

`pushf`/`popf` 配合起来除了可以刻意改变 **FL** 中的某些位外，也可以用来保护/恢复 **FL** 的值：

```
pushf; 保护 FL 的当前值
```

```
...
```

```
popf; 恢复 FL 的值
```

在 32 位系统中，**EFL** 是一个 32 位寄存器，对应的指令是 `pushfd` 和 `popfd`；若在 32 位系统使用 `pushf`/`popf` 则控制的仅是 **EFL** 的低 16 位即 **FL**。

### 3. 符号扩充指令：CBW, CWD, CDQ

`cbw`:convert byte to word

`cwd`:convert word to double word

cdq: convert double word to quadruple word  
mov al, 0FEh  
cbw; 把 AL 扩充成 AX, AX=0FFFEh  
mov ax, 8000h  
cwd; 把 AX 扩充成 DX:AX, DX=FFFFh, AX=8000h  
mov eax, 0ABCD1234h  
cdq; 把 EAX 扩充成 EDX:EAX  
; EDX=0FFFFFFFFh, EAX=0ABCD1234h

零扩充指令: movzx

movzx ax, al; zx: zero extension  
movzx eax, al;  
movzx ebx, cx;

新的符号扩充指令: movsx

movsx ax, al; sx: sign extension 符号扩充  
; 效果等同于 cbw

#### 4. 乘法指令: mul

8 位乘法: 被乘数一定是 AL, 乘积一定是 AX

例如 mul bh 表示 AX=AL\*BH

16 位乘法: 被乘数一定是 AX, 乘积一定是 DX:AX

例如 mul bx 表示 DX:AX=AX\*BX

**32 位乘法：被乘数一定是 EAX，乘积一定是 EDX:EAX**

**例如 mul ebx 表示 EDX:EAX=EAX\*EBX**

**利用乘法指令把十进制字符串转化成 32 位整数的例子：**

<http://10.71.45.100/bhh/dec2v32.asm>

**利用乘法指令把十进制字符串转化成 16 位整数的例子：**

<http://10.71.45.100/bhh/dec2v16.asm>