代码优化的目标:体积小、速度快,二者通常矛盾，根据需要，选择性优化体积或速度；

最常代码优化:

1.寄存器清0

1) mov eax, 00000000h ;5 bytes

**一样快但更小的优化写法:**

2) sub eax, eax ;2 bytes

3) xor eax, eax ;2 bytes

2.测试寄存器是否为0

简单的比较指令居然要用7/11 bytes：

1) cmp eax, 00000000h ;5 bytes

je \_label\_ ;2/6 bytes (short/near)

[\* 很多指令针对eax作了优化,尽可能多用eax,比如CMP EAX, 1234

5678h (5 bytes) 如果用其他寄存器,就是6bytes \*]

**优化写法:**

2) or eax, eax ;2 bytes

je \_label\_ ;2/6 (short/near)

3) test eax, eax ;2 bytes

je \_label\_ ;2/6 (short/near)

test不改变任何寄存器,且不向任何寄存器写入内容,通常能取得更快的执行速度。

4) xchg eax, ecx ;1 byte

jecxz \_label\_ ;2 bytes

在短跳转情况下2)和3)又节省了1字节

3.测试寄存器是否为0FFFFFFFFh

一些API返回-1,因此如何测试这个值呢?

1) cmp eax, 0ffffffffh ;5 bytes

je \_label\_ ;2/6 bytes

优化写法:

2) inc eax ;1 byte

je \_label\_ ;2/6 bytes

dec eax ;1 byte

可以节省3 bytes并且执行速度会更快。

4.置寄存器为0FFFFFFFFh

看看假如你是Api的作者,如何返回-1?

1) mov eax, 0ffffffffh ;5 bytes

**优化方法1——减少**2 bytes**:**

2) xor eax, eax / sub eax, eax ;2 bytes

dec eax ;1 byte

**优化方法2——减少**2 bytes**:**

3) stc ;1 byte

sbb eax, eax ;2 bytes

有时还可以优化掉1 byte:

jnc \_label\_

sbb eax, eax ;2 bytes only!

\_label\_: ...

5.寄存器清0并移入低字数值

1) xor eax, eax ;2 bytes

mov ax, word ptr [esi+xx];4 bytes

**优化方法1——**减少2 bytes**:**

2) movzx eax, word ptr [esi+xx] ;4 bytes

!

下面的

3) xor eax, eax ;2 bytes

mov al, byte ptr [esi+xx] ;3 bytes

优化为:

4) movzx eax, byte ptr [esi+xx] ;4 bytes

尽可能利用movzx

5) xor eax, eax ;2 bytes

mov ax, bx ;3 bytes

优化为：

6) movzx eax, bx ;3 bytes

执行速度不慢并通常能节省字节

6.关于push,优化代码体积：寄存器操作内存操作要快很多

1) mov eax, 50h ;5 bytes

这样就小了1 word

2) push 50h ;2 bytes

pop eax ;1 byte

当操作数只有1字节时候,push只有2 bytes,否则就是5 bytes,记住!

下一个问题,向堆栈中压入7个0

3) push 0 ;2 bytes

push 0 ;2 bytes

push 0 ;2 bytes

push 0 ;2 bytes

push 0 ;2 bytes

push 0 ;2 bytes

push 0 ;2 bytes

占用14字节,显然不好,优化一下：

4) xor eax, eax ;2 bytes

push eax ;1 byte

push eax ;1 byte

push eax ;1 byte

push eax ;1 byte

push eax ;1 byte

push eax ;1 byte

push eax ;1 byte

可以更紧凑,但会慢一点的形式如下:

5) push 7 ;2 bytes

pop ecx ;1 byte

\_label\_: push 0 ;2 bytes

loop \_label\_ ;2 bytes

可以节省7字节....

有时候可能需要将一个值从一个内存地址转移到另外内存地址,并且要保存所有

寄存器:

6) push eax ;1 byte

mov eax, [ebp + xxxx] ;6 bytes

mov [ebp + xxxx], eax ;6 bytes

pop eax ;1 byte

试试push,pop

7) push dword ptr [ebp + xxxx] ;6 bytes

pop dword ptr [ebp + xxxx] ;6 bytes

7.乘法

当eax已经放入被乘数,要乘28h,如何来写?

1) mov ecx, 28h ;5 bytes

mul ecx ;2 bytes

好一点的写法如下:

2) push 28h ;2 bytes

pop ecx ;1 byte

mul ecx ;2 bytes

更好的写法:

3) imul eax, eax, 28h ;3 bytes

intel在新CPU中提供新的指令并不是摆设,需要你的使用.

8.字符串操作

如何从内存取得一个字节呢?

速度快的方案:

1) mov al/ax/eax, [esi] ;2/3/2 bytes

inc esi ;1 byte

代码小的方案:

2) lodsb/w/d ;1 byte

我比较喜欢lod因为他小,虽然速度慢了点.

如何到达字符串尾呢?

JQwerty's method:

9) lea esi, [ebp + asciiz] ;6 bytes

s\_check: lodsb ;1 byte

test al, al ;2 bytes

jne s\_check ;2 bytes

Super's method:

10) lea edi, [ebp + asciiz] ;6 bytes

xor al, al ;2 bytes

s\_check: scasb ;1 byte

jne s\_check ;2 byte

选择哪一个?Super的在386以下的更快,JQwerty的在486以及pentium上更快,体积一样。

9.复杂一点的...

假设有一个DWORD表,ebx指向表的开始,ecx是数组元素下标,想给数组元素加1：

1) pushad ;1 byte

imul ecx, ecx, 4 ;3 bytes

add ebx, ecx ;2 bytes

inc dword ptr [ebx] ;2 bytes

popad ;1 byte

优化为:

2) inc dword ptr [ebx+4\*ecx] ;3 bytes

一条指令，节省6字节,速度更快,更易读

还可以有立即数:

3) pushad ;1 byte

imul ecx, ecx, 4 ;3 bytes

add ebx, ecx ;2 bytes

add ebx, 1000h ;6 bytes

inc dwor ptr [ebx] ;2 bytes

popad ;1 byte

优化为:

4) inc dword ptr [ebx+4\*ecx+1000h] ;7 bytes

节省了8字节!

10. ★巧用lea指令：

lea eax, [12345678h]

eax的最后结果是什么呢?正确答案是12345678h.

假设 EBP = 1

lea eax, [ebp + 12345678h];6 bytes

结果是12345679h....呵呵比较一下:

mov eax, 12345678h ;5 bytes

add eax, ebp ;2 bytes

5) 看看:

mov eax, 12345678h ;5 bytes

add eax, ebp ;2 bytes

imul ecx, 4 ;3 bytes

add eax, ecx ;2 bytes

6) 用lea计算:

lea eax, [ebp+ecx\*4+12345678h] ;7 bytes

一条lea指令速度更快!不影响标志位...记住下面的格式,在许多地方善用他们

你可以节省时间和空间.

OPCODE [BASE + INDEX\*SCALE + DISPLACEMENT]

11.下面是关于病毒重定位优化的,惧毒人士请绕行...

下面的代码你不应该陌生

1) call gdelta

gdelta: pop ebp

sub ebp, offset gdelta

在以后的代码中我们这样使用delta来避免重定位问题

lea eax, [ebp + variable]

这样的指令在应用内存数据的时候是不可避免的,如果能优化一下,我门将会得到

数倍收益,打开你的sice或者trw或者ollydbg等调试器,看看:

3) lea eax, [ebp + 401000h] ;6 bytes

假如是下面这样

4) lea eax, [ebp + 10h] ;3 bytes

也就是说如果ebp后面变量是1字节的话,总的指令就只有3字节

修改一下最初的格式变为:

5) call gdelta

gdelta: pop ebp

在某些情况下我们的指令就只有3字节了,可以节省3字节,让我们看看:

6) lea eax, [ebp + variable - gdelta] ;3 bytes

和上面的是等效的,但是我们可以节省3字节,看看CIH...

12.其他技巧:

如果EAX小于80000000h,edx清0:

1) xor edx, edx ;2 bytes, but faster

换种写法：

2) cdq ;1 byte, but slower

我一直使用cdq,为什么不呢?体积更小...

下面这种情况一般不要使用esp和ebp,使用其他寄存器.

1) mov eax, [ebp] ;3 bytes

2) mov eax, [esp] ;3 bytes

3) mov eax, [ebx] ;2 bytes

交换寄存器中4个字节的顺序?用bswap

mov eax, 12345678h ;5 bytes

太长了，这样：

bswap eax ;2 bytes

;eax = 78563412h now

Wanna save some bytes replacin' CALL ?

1) call \_label\_ ;5 bytes

ret ;1 byte

改为如下：

2) jmp \_label\_ ;2/5 (SHORT/NEAR)

如果仅仅是优化,并且不需要传递参数,请尽量用jmp代替call

比较 reg/mem 时如何节省时间:

1) cmp reg, [mem] ;slower

这样写：

2) cmp [mem], reg ;1 cycle faster

乘2除2如何节省时间和空间?

1)mov eax, 1000h

mov ecx, 4 ;5 bytes

xor edx, edx ;2 bytes

div ecx ;2 bytes

移位：

2)shr eax, 2 ;3 bytes

除法呢？

3)mov ecx, 4 ;5 bytes

mul ecx ;2 bytes

当然也一样

4) shl eax, 2 ;3 bytes

loop指令

1) dec ecx ;1 byte

jne \_label\_ ;2/6 bytes (SHORT/NEAR)

分特，loop没学过？

2) loop \_label\_ ;2 bytes

再看:

3) je $+5 ;2 bytes

dec ecx ;1 byte

jne \_label\_ ;2 bytes

记住loop

4) loopXX \_label\_ (XX = E, NE, Z or NZ) ;2 bytes

loop体积小,但486以上的cpu上执行速度会慢一点...

交换寄存器:

1)push eax ;1 byte

push ebx ;1 byte

pop eax ;1 byte

pop ebx ;1 byte

明明有xchg指令：

2) xchg eax, ebx ;1 byte

3) xchg ecx, edx ;2 bytes

如果仅仅是想移动数值,用mov,在pentium上会有较好的执行速度:

4) mov ecx, edx ;2 bytes

比较以下程序:

1) 未优化 —— 共14 bytes:

lbl1: mov al, 5 ;2 bytes

stosb ;1 byte

mov eax, [ebx] ;2 bytes

stosb ;1 byte

ret ;1 byte

lbl2: mov al, 6 ;2 bytes

stosb ;1 byte

mov eax, [ebx] ;2 bytes

stosb ;1 byte

ret ;1 byte

2) 优化——11 bytes:

lbl1: mov al, 5 ;2 bytes

lbl: stosb ;1 byte

mov eax, [ebx] ;2 bytes

stosb ;1 byte

ret ;1 byte

lbl2: mov al, 6 ;2 bytes

jmp lbl ;2 bytes

要善于利用代码

读取常数变量,试试在指令中直接定义:

mov eax, [ebp + variable] ;6 bytes

...

mov [ebp + variable], eax ;6 bytes

...

...

variable dd 12345678h ;4 bytes

2) 优化为:

mov eax, 12345678h ;5 bytes

variable = dword ptr $ - 4

...

...

mov [ebp + variable], eax ;6 bytes

呵呵,好久没看到这么有趣的代码了,前提是编译的时候支持代码段的写入属性

最后介绍未公开指令SALC,现在的调试器都支持...什么含义呢:就是CF位置1的话

就将al置为0xff

1) jc \_lbl1 ;2 bytes

mov al, 0 ;2 bytes

jmp \_end ;2 bytes

\_lbl: mov al, 0ffh ;2 bytes

\_end: ...

当然用SALC啦

2) SALC db 0d6h ;1 byte ;)