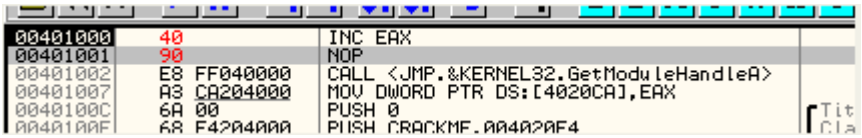
第 5 章-数学指令

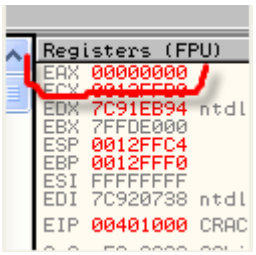
INC 和 DEC

这两个指令分别是执行增加和减少的操作，如果是 INC 指令的话，就加 1，如果是 DEC 指令的话，就减 1。

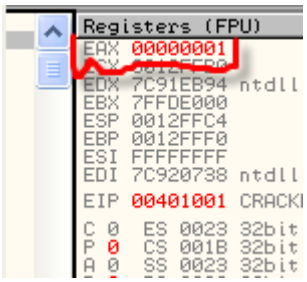
我们跟之前一样用 OD 打开 cruehead 的 CrackMe。



EAX 在我的机器上面初始值是 0，如果你的机器上面不是 0 的话，你可以手动修改。

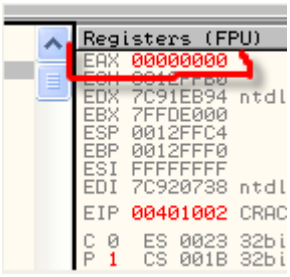


然后按下 F7 键，执行 INC EAX 指令，EAX 就会递增 1。



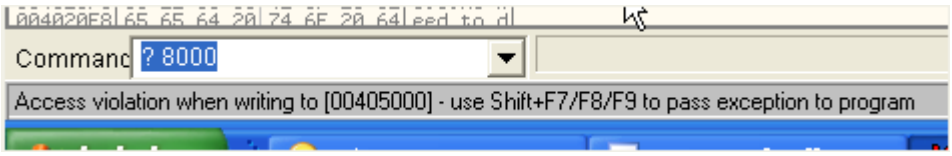
同样的，我们可以使用 DEC 指令替换掉下面的指令。

按 F7 键，执行指令 DEC EAX，EAX 将会由 1 变成 0。

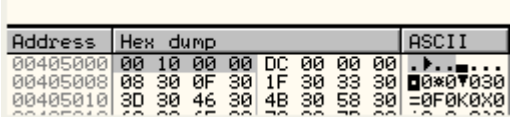


也可以增加或者减少内存单元中的值。

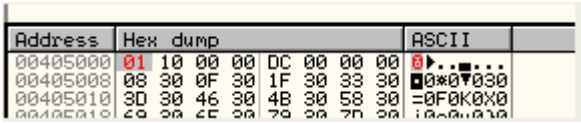
上面的内存单元并不会增加 1，而是引发异常，因为该内存地址我们不具有写权限。



如果这里有写权限的话,结果会是怎么样的呢?我们去数据窗口查看 405000 地址。

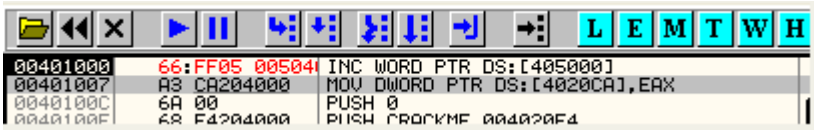


反过来读取,就会是 00001000,增加 1 的话,就变成 00001001。



这是给 DWORD 这个 4 字节的值增加 1 的情况。

对于 WORD 来说增加 1 到最后两个字节中。

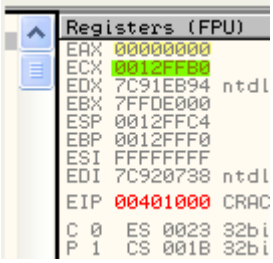


对于 BYTE 来说增加 1 到最后一个字节。

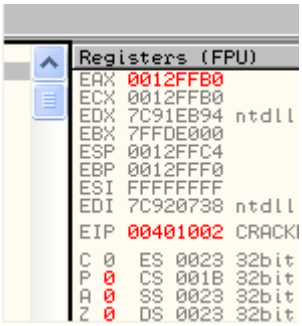
ADD

ADD 指令有两个操作数,相加后的结果存放到第一个操作数中。ADD EAX,1 等价于 INC EAX。ADD 也将两个寄存器相加,我们可以到 OD 里面看一看。

执行该语句之前:

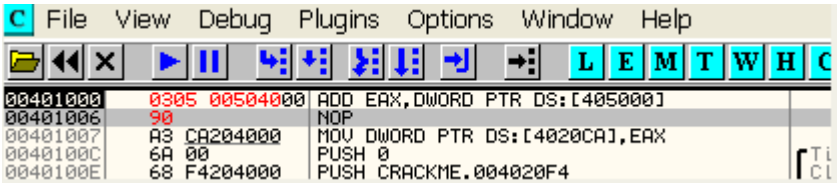


在我的机器上 EAX 的值是 00000000,ECX 的值是 12FFB0。你的机器上可以是其他的值,你可以自己修改它们,当你按下 F7 键,这两个寄存器相加,结果存放到 EAX 中,一起来看看。



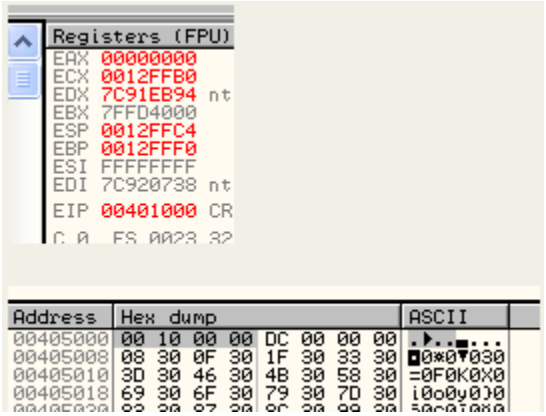
由于 EAX 被修改了，所以变成了红色，相加的结果保存在其中。

也可以将寄存器与内存单元的内容相加。

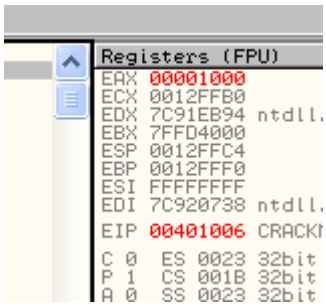


在这种情况下，是否对该内存地址具有写权限都没有问题，因为相加的结果是存放到 EAX 寄存器中的，[405000]内存单元的值并没有改变。因为并不会引发异常。

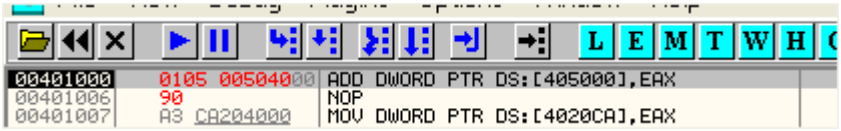
按下 F7 键之前，EAX 的值为 0，405000 内存单元中的值为 00001000。



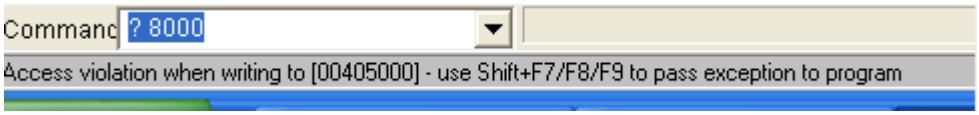
按 F7 键，计算它们的和。



EAX 的初始值是 0，现在变成了 1000。如果你想把结果存放到内存单元中，我们可以这么写：

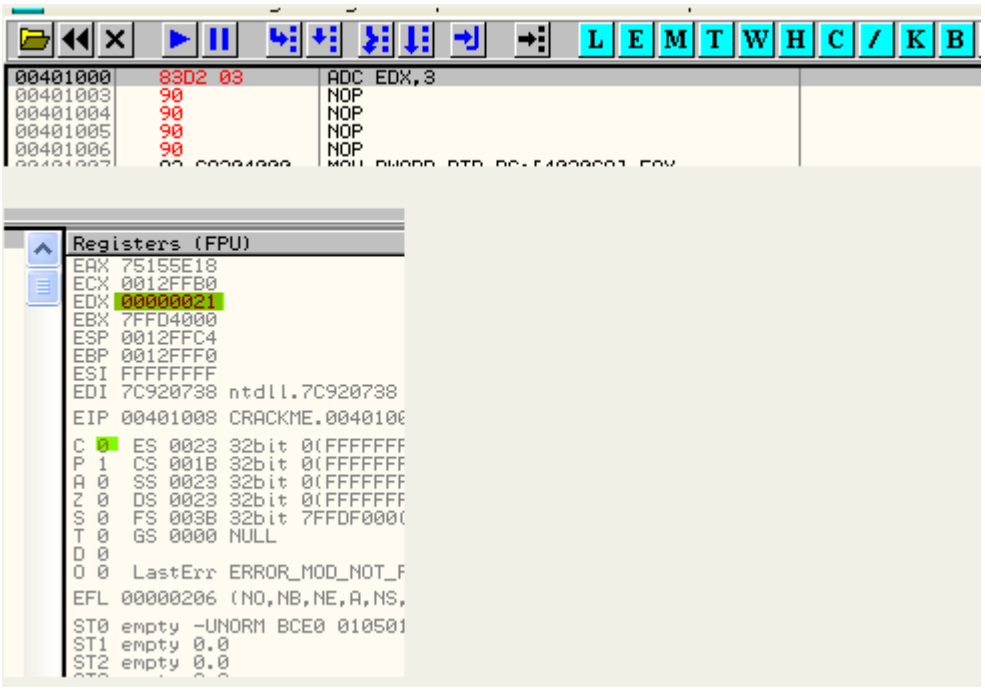


在这种情况下，结果存放到 405000 内存单元中，按下 F7 键，由于我们对该内存单元没有写权限，当尝试修改该内存单元的值时候，发引发异常。

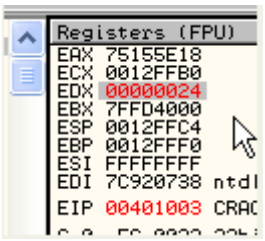


ADC(带进位的加法)

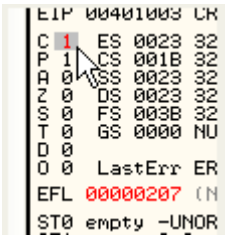
在这种情况下,两个操作数的和加上进位标志的值,结果存放到第一个操作数中。



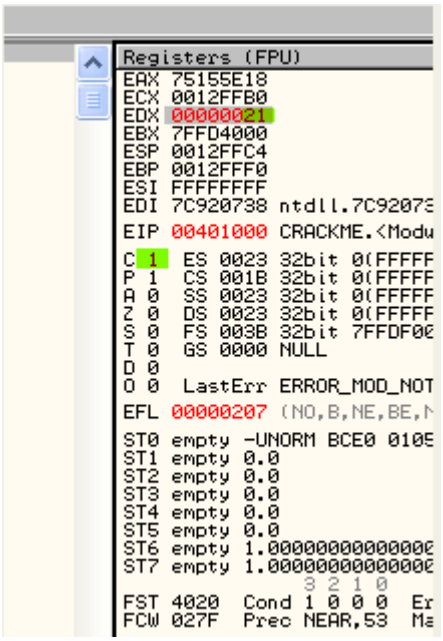
这里我们可以看到,EDX 的值为 21,加上 3,已经进位标志,在这里,进位标志为 0,按下 F7 键。



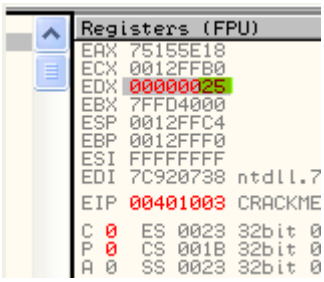
看到结果为 24。现在双击你的鼠标将进位标志修改为 1,然后重复上面的操作。



我们通过修改进位标志,然后重新执行这条指令。



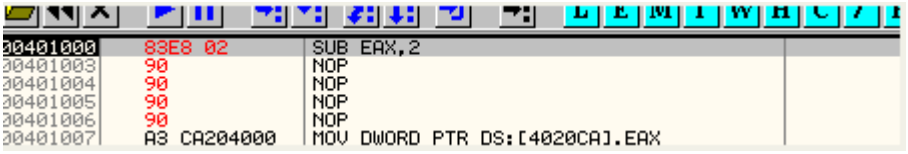
按 F7 键,看,结果是 25。



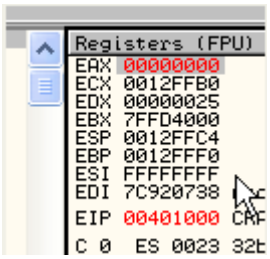
由于 EDX 的值为 21,然后一个数值 3,然后进位标志 1,所以结果为 25。

SUB

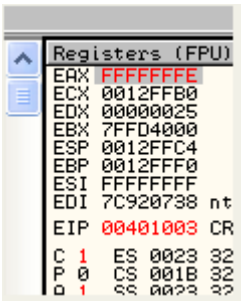
这个指令与 ADD 刚好相反-它将第一个操作数减去第二个操作数的值存放到第一个操作数中。



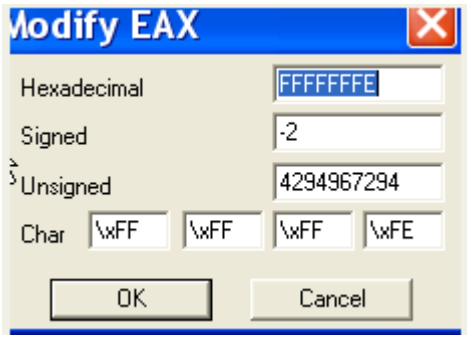
在我的机器上,执行这条指令之前,寄存器的情况如下:



按下 F7 键,EAX 的 0 减去 2。



16 进制的结果为 FFFFFFFE,在这个值上面双击鼠标,你可以看到十进制的值为-2。



我们可以看到十进制为-2。

另外也可以使用该指令来计算寄存器的值来寄存器的值,寄存器的值减去内存单元的值。

SUB EAX,ECX

即 EAX-ECX 的值保存到 EAX 中。

和

SUB EAX,DWORD PTR DS:[405000]

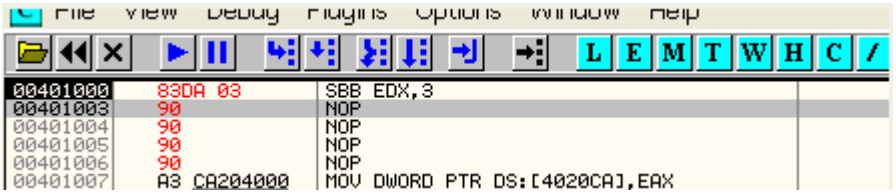
寄存器 EAX 减去 405000 内存单元的值,并将结果保存在 EAX 中。

SUB DWORD PTR DS:[405000],EAX

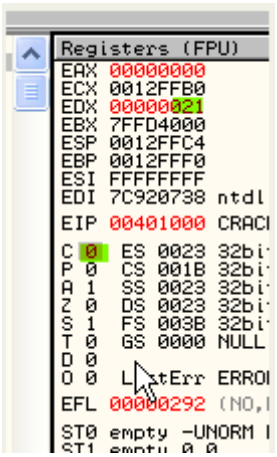
这种情况下,由于我们对 405000 这个内存单元没有写权限,将结果存放到其中的话,会引发一个异常。

SBB

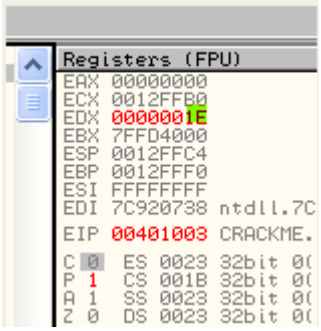
该指令跟 ADC 正好相反,它计算两个操作数的差值,并且还要减去进位标志,结果存放到第一个操作数中。



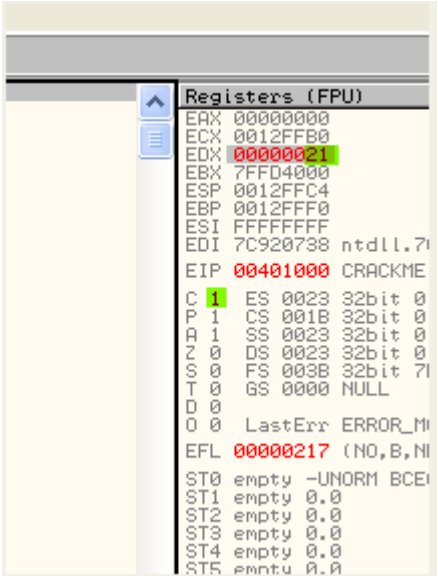
执行该指令之前,EDX 的值为 21,进位标志为 0。



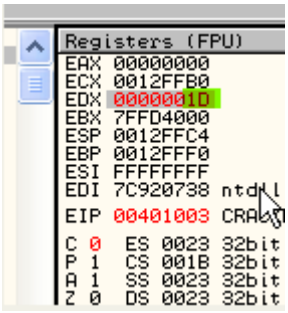
按下 F7 键,EDX 减去 3,然后减去进位标志 0。



现在,将进位标志修改为 1,然后重复上面的操作。



按下 F7 键,EDX 减去 3,然后减去进位标志 1。



这种情况,结果为 1D。

MUL(无符号数的乘法)

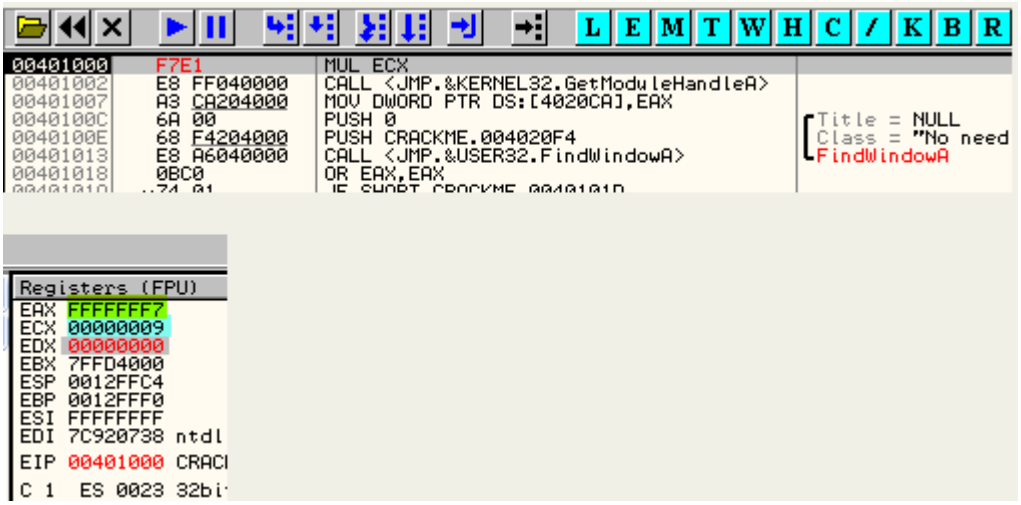
有两种乘法,第一个种是 MUL,这种是无符号数乘法,只有一个操作数,另一个操作数是 EAX,结果存放到 EDX:EAX 中。

例如:

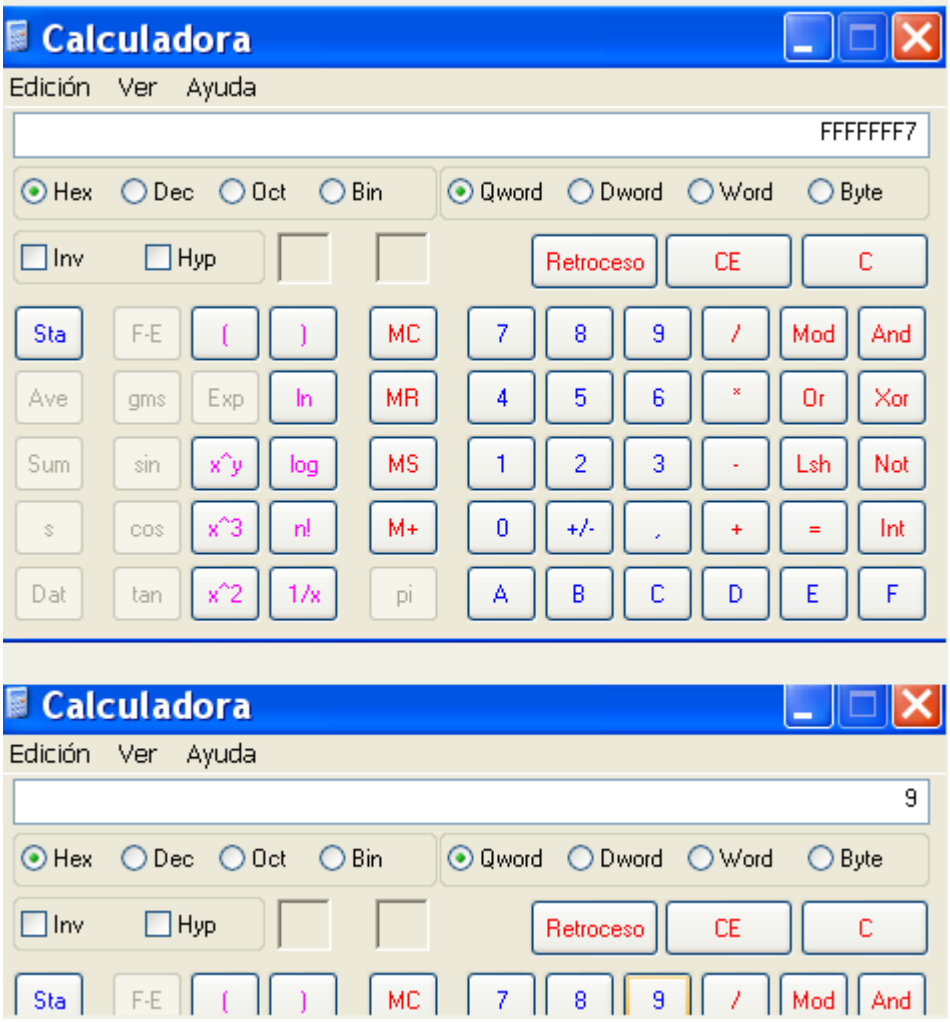
MUL ECX

这里是无符号数 EAX,ECX 相乘,结果存放到 EDX:EAX 中。

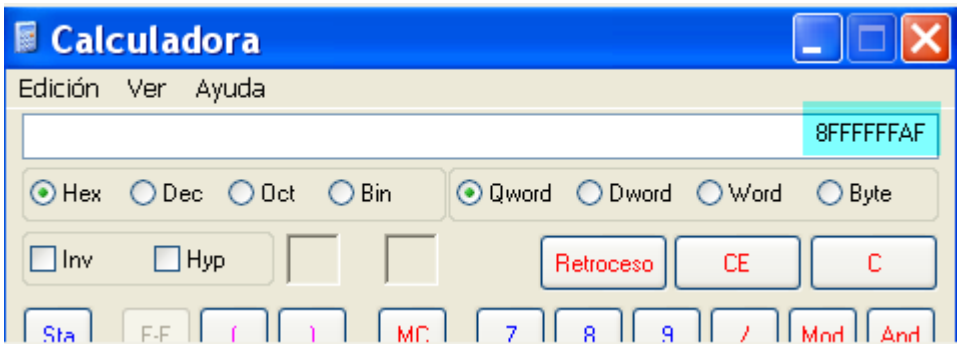
OD 中来看下面的例子:



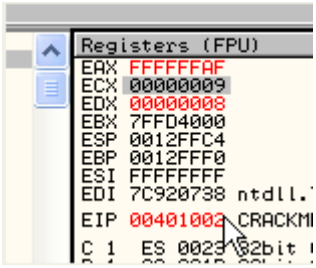
我将 EAX 设置为了 FFFFFFF7,ECX 设置为了 9,使用 Windows 自带的计算器计算的结果如下:



结果为:



EAX 的值不变,按下 F7 键,在 OD 中我们来看看会发生什么。



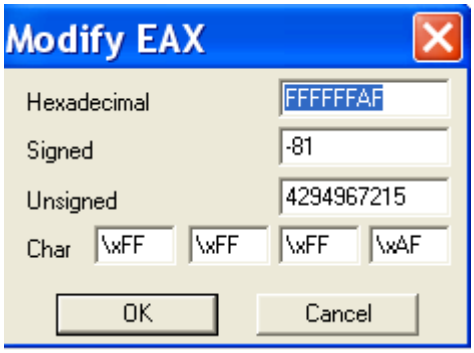
由于 EDX 和 EAX 的值改变了,所以以红色突出显示,而且,我们可以看到,一部分保存在 EAX 中,另一部分保存在 EDX 中,EAX 容纳不下高位的 8,所以存放到 EDX 中了,所以我们说结果存放到 EDX:EAX 中,也就是说,大小是单个寄存器的两倍。

下面的情况:

MUL DWORD PTR DS:[405000]

两个无符号数,405000 内存单元中的值乘以 EAX,结果跟之前的一样,存放到 EDX:EAX 中。

要在 OD 中查看一个数字的无符号 16 进制值的话,可以在任意通用寄存器上面双击鼠标左键。



第二行可以看到有符号数的值(有符号值为-81),但是 MUL 这样的指令,操作数都是无符号的。第三行包含了无符号的十进制值。我们可以看到是 4294967215,是一个无符号数。

IMUL(有符号数的乘法)

IMUL 指令用法类似于 MUL。

IMUL ECX

该指令将有有符号数 ECX 乘以 EAX,结果存放到 EDX:EAX 中。

除了上面一条指令外,IMUL 还允许使用多个操作数,这是与 MUL 不同的地方。

补充:

尽管在默认情况下是使用 EAX 和 EDX 寄存器,但是我们还可以指定其他的数据源以及目标多达三个操作数。第一个操作数,用来乘以两个数,并且保存相乘的结果,结果必须保存在寄存器中。下面我们将看到两个和三个操作数的例子。

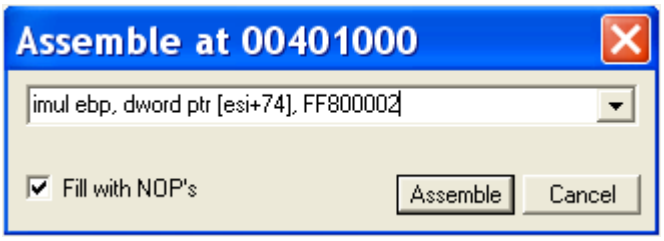
F7EB IMUL EBX EAX x EBX -> EDX:EAX

这是第一个例子,类似于 MUL,但是是有符号的乘法。

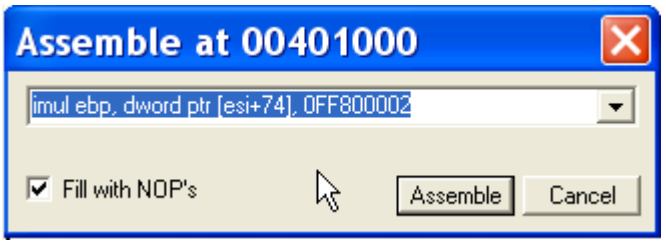
696E74020080FF　　IMUL EBP, DWORD PTR [ESI+74],FF800002 [ESI+74] x FF800002 -> EBP

这是第二个例子,其中有三个操作数: ESI+74 内存单元的值乘以 FF800002,并且将结果存储到 EBP 中。我们可以在 OD 中执行看看。

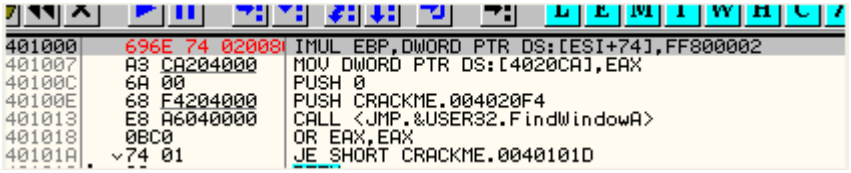
把 IMUL EBP, DWORD PTR [ESI+74], FF800002 拷贝到 OD 中。



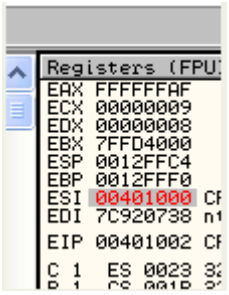
当我们按下 Assemble 按钮的时候会提示错误,因为在 OD 中,数字开头不能为字符,我们可以在开头添加 0 来解决这个问题。



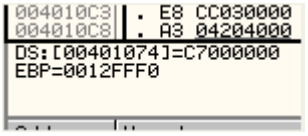
现在按下 Assemble 按钮就不会提示错误了。



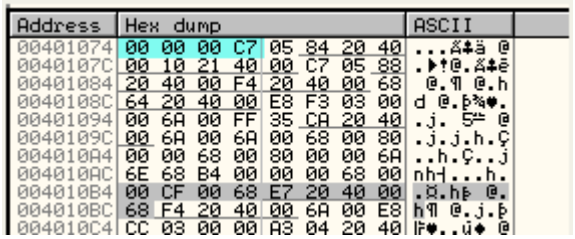
为了确保能够读取 ESI+74 这个地址的值,我们将 ESI 的值修改为 401000。



见注释:



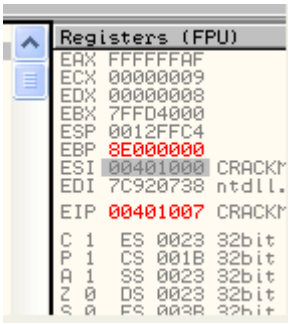
这里表示 ESI+74 的值为 401074,该内存单元中的值为 C7000000。在数据窗口中查看一下。



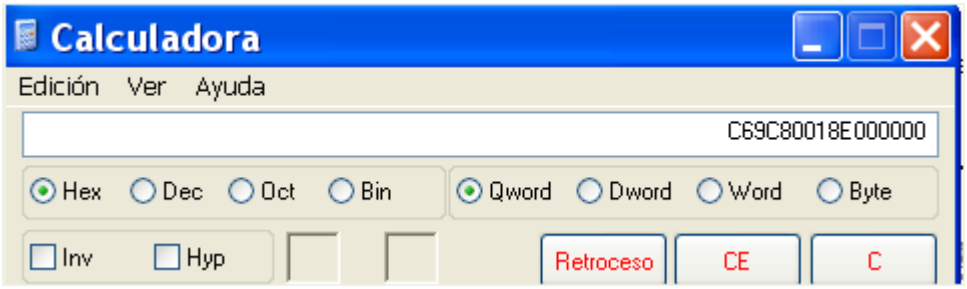
这里,ESP+74 是指向 401074 这个地址,里面存放的值为 C7000000。乘以 FF800002,结果存放到 EBP 中。

IMUL EBP, DWORD PTR [ESI+74], FF800002

按下 F7 键,可以看到 EBP 变成红色了,相乘的结果存放在其中。



在计算器中我们计算 C7000000*FF800002 的结果如下:



由于 EBP 容纳不下结果,所以只能保存能够容纳的部分,其余的部分丢弃。

第三个例子,只有两个操作数,两者相乘,并将结果存放到第一个操作数中。

0FAF55E8 IMUL EDX, DWORD PTR [EBP-18] EDX x [EBP-18] -> EDX

正如你所看到的,这些方法中,最好的方式就是一个操作数的方式,结果存放到 EDX:EAX 中,这意味着拥有两倍的大小,通过两个操作数或者三个操作数无法完成的操作可以通过一个操作数的方式来完成。

DIV(无符号除法)/IDIV(有符号除法)

这两个指令反别与 MUL 和 IMUL 相反。

DIV 只有一个操作数,该操作数必须是无符号数,结果存放到 EDX:EAX 中。

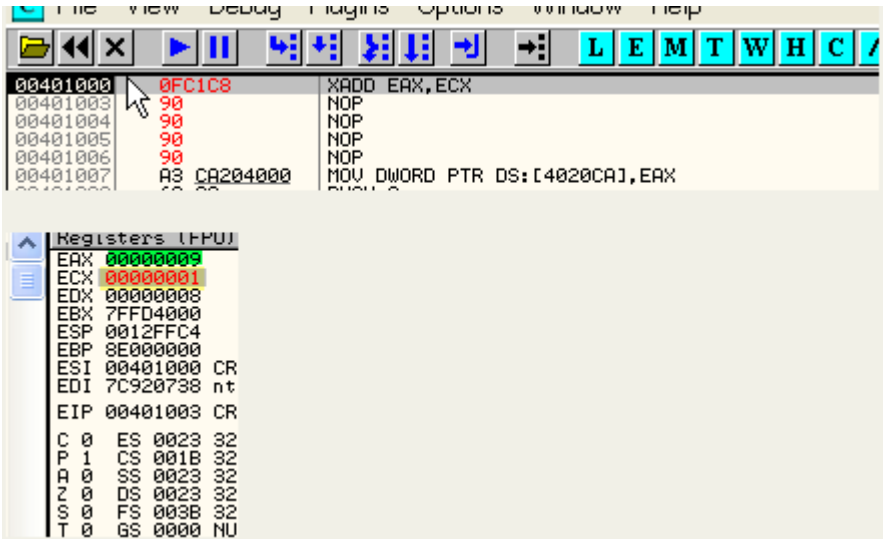
IDIV 指令经常被使用。如果是一个操作数的话,那么它和 DIV 类似,只不过操作数是有符号的,结果依然保存在 EDX:EAX 中。两个操作数的情况,第一个操作数除以第二个操作数, 结果存放到第一个操作数中。三个操作数的情况,第二个操作数除以第三个操作数,结果存放到第一个操作数中。

由于跟 MUL 和 IMUL 类似,这里就不提供例子了。

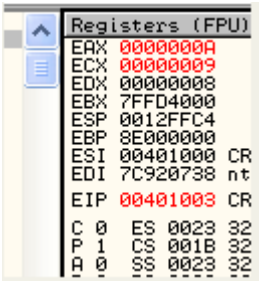
XADD(交换并相加)

正如你所猜想的一样,这个指令其实就是 XCHG 和 ADD 两个简单指令的组合。

XADD EAX,ECX



我们可以看到 ECX 的值为 1，EAX 的值为 9。按 F7 键,它们交换数值。也就是,EAX 的值变为了 1,ECX 的值变为了 9,然后相加。



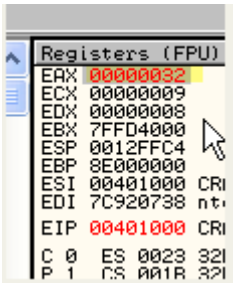
可以看出,相加的结果是存放到第一个操作数中的。ECX 的值现在为 9,这是执行这条指令之前 EAX 的值。

NEG

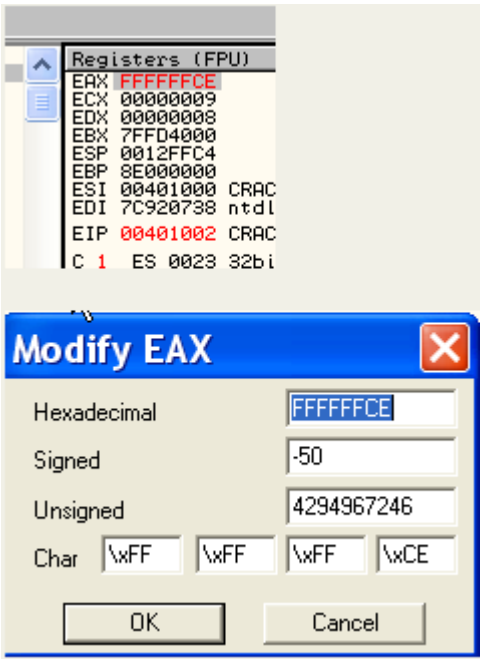
该指令的目的是将操作数的符号取反,即如果我们有一个 32 位的 16 进制数,用 NEG 操作以后,结果就会取反。

例如:

在 OD 中写入 NEG EAX 指令,并将 EAX 的值改为 32。



按下 F7 键后,将会得到一个-32。一起来看看:



在第二行中,我们看到的结果是十进制的-50(对应的十六进制的-32)。

如果你再命令栏中输入? -32,就可以得到其十进制的值。这个值就是-50,还有 16 进制的值 FFFFFFCE。

所以 NEG 指令是将操作数符号取反。

逻辑指令

逻辑指令有两个操作数,两操作数按位运算,并将结果存放到第一个操作数中。

AND

只有两个二进制位都为 1 的时候结果才为 1,其他情况,结果都为 0。

1 and 1 = 1

1 and 0 = 0

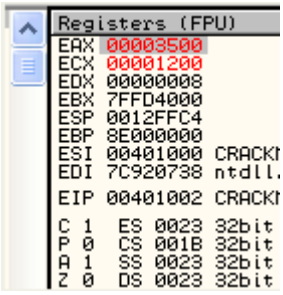
0 and 1 = 0

0 and 0 = 0

让我们来看看 OD 中的例子:

AND ECX,EAX

将 ECX 设置为 0001200,EAX 设置为 3500。

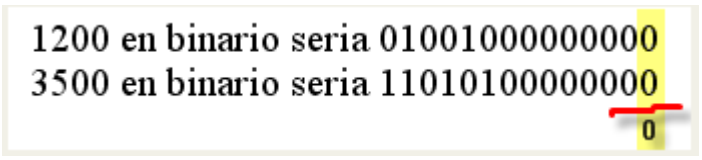


手工计算的话,我们需要将两个数字转化为二进制:

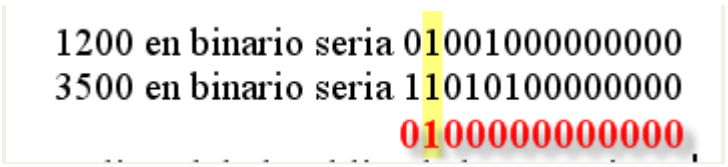
1200 二进制为 01001000000000

3500 二进制为 11010100000000

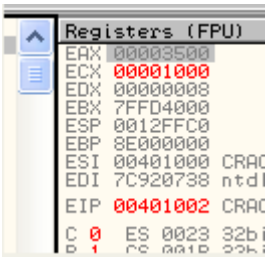
我们将两数二进制按位与,例如:最后一位数字:



两个 0 取 0。对每一位重复此过程,当两位都为 1 的时候才取 1,这两个数中只有一次,两位都是 1 的情况。



按下 F7 键,我们将看到 ECX 的值为 1000。相当于二进制的 01000000000000。



OR

该指令 AND 的不同之处在于,两位中只要有一位为 1,结果就取 1。

1 or 1 = 1

1 or 0 = 1

0 or 1 = 1

0 or 0 = 0

XOR

该指令时异或运算,当两位不同时取 1,相同时取 0。

1 xor 1 = 0

1 xor 0 = 1

0 xor 1 = 1

0 xor 0 = 0

NOT

该指令是简单的按位取反。

not 1 = 0

not 0 = 1

例如:not 0110 = 1001

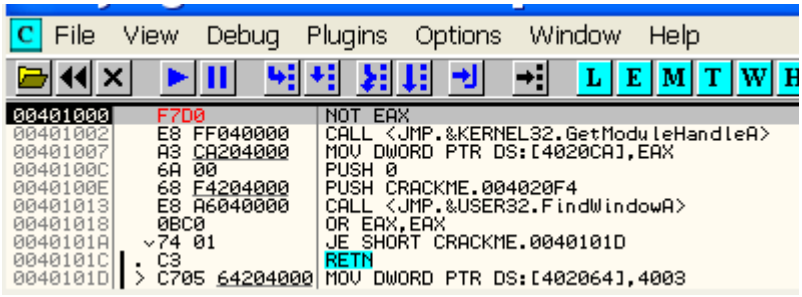
例如:EAX=1200,其二进制位 1001000000000,我们将其填充为 32 位,不足的时候补 0。

00000000000000000001001000000000

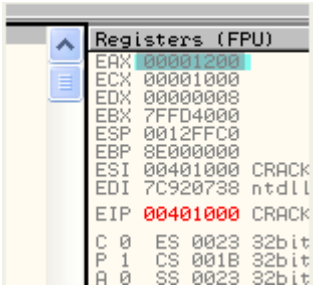
NOT 后得到:

11111111111111111110110111111111

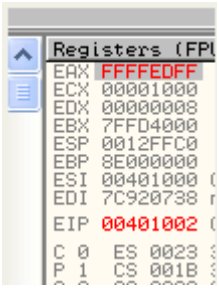
在 windows 自带的计算器中十六进制显示为 FFFFEDFF。



将 EAX 设置为 1200。



按 F7 键。



可以看到与计算器中的结果是一直的。

这里我们第 5 章就结束了,下一章我们将介绍比较,跳转,函数调用,以及函数返回指令。