1. 数据类型和容器

立即数：一个常数，也就是一个普通的数字，不能作为容器。

寄存器：CPU中的临时储存单元，80X86的32位平台平台上CPU寄存器仅有14个，寄存器速度比内存快得多，存储的可以是从0x00000000到0xFFFFFFFF之间的一个整数，一个int类型的存储大小，占32个bit，即一个DoubleWord的长度。

**8086  CPU 中寄存器总共为 14 个，且均为 16 位 。**

即 **AX，BX，CX，DX，SP，BP，SI，DI，IP，FLAG，CS，DS，SS，ES** 共 14 个。

而这 14 个寄存器按照一定方式又分为了通用寄存器，控制寄存器和段寄存器。

**通用寄存器：**

**AX，BX，CX，DX 称作为数据寄存器：**

AX (Accumulator)：累加寄存器，也称之为累加器；

BX (Base)：基地址寄存器；

CX (Count)：计数器寄存器；

DX (Data)：数据寄存器；

**SP 和 BP 又称作为指针寄存器：**

SP (Stack Pointer)：堆栈指针寄存器；

BP (Base Pointer)：基指针寄存器；

**SI 和 DI 又称作为变址寄存器：**

SI (Source Index)：源变址寄存器；

DI (Destination Index)：目的变址寄存器；

**控制寄存器：**

IP (Instruction Pointer)：指令指针寄存器；

FLAG：标志寄存器；

**段寄存器：**

CS (Code Segment)：代码段寄存器；

DS (Data Segment)：数据段寄存器；

SS (Stack Segment)：堆栈段寄存器；

ES (Extra Segment)：附加段寄存器；

而80X86（32位）CPU上的寄存器都从16位扩展到32位，名字也都填了一个E，如EAX，EBX，ECX，EDX分别对应AX，BX，CX，DX，其中FLAG变成EFL，是一个例外。

内存：操作系统分配给程序的虚拟内存，并不是真实内存。80X86的32位平台每个进程都有4G的虚拟内存，也可以说是4GB寻址能力，地址从低位到高位分别是0x00000000到0xFFFFFFFF，占32个bit，即一个DoubleWord的长度，总共4GB。

1. 数据储存方式：

在内存中，一个数比如0x11223344会被分个储存，每个被分成的部分是八个bit，也就是一个byte。所以0x11223344会被分成0x11,0x22,0x33,0x44四个部分，数字的低位存在内存的高位，数据的高位存在内存的低位，这种模式被称为“小端模式”。

从内存条上看，数据是这样的：

Address: 0x00000000 | 0x00000001 | 0x00000002 | 0x00000003

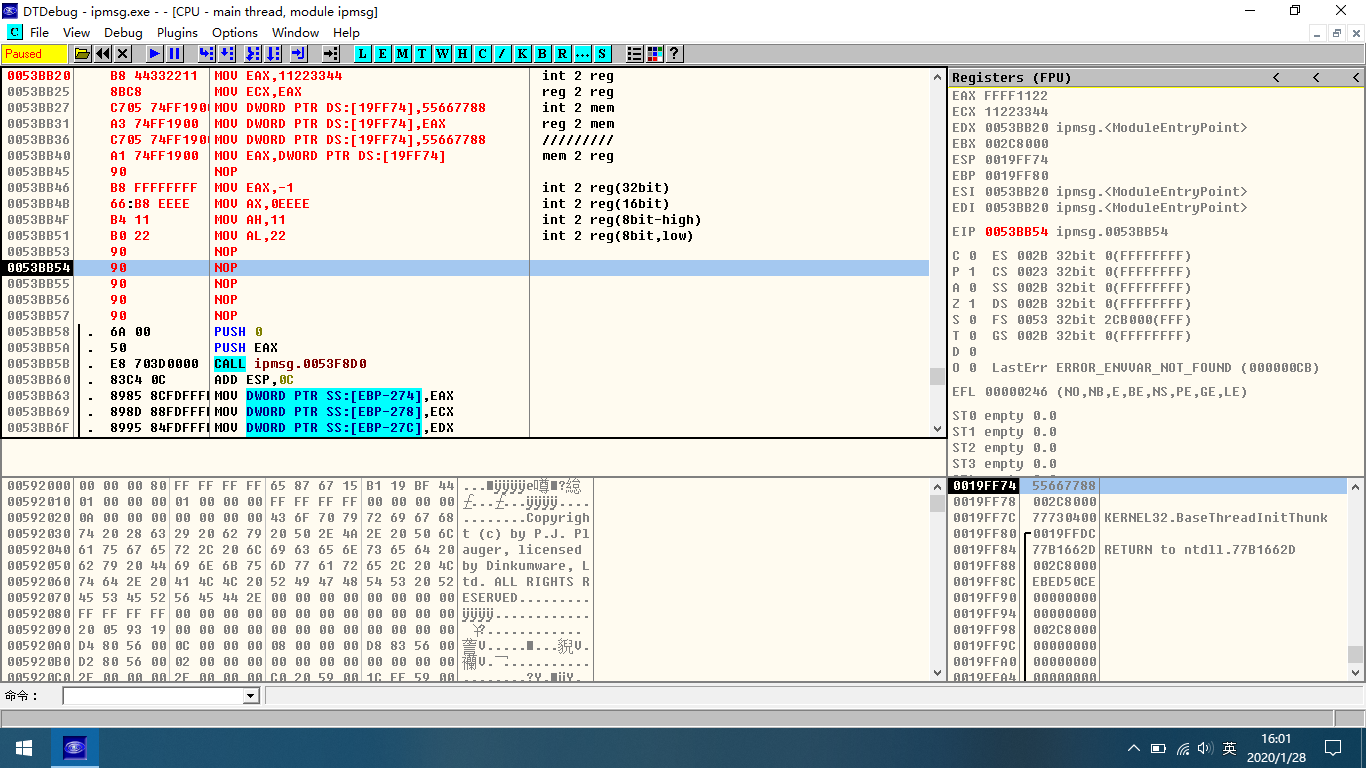
..Value: 0x44.......| 0x33.......| 0x22.......| 0x11......

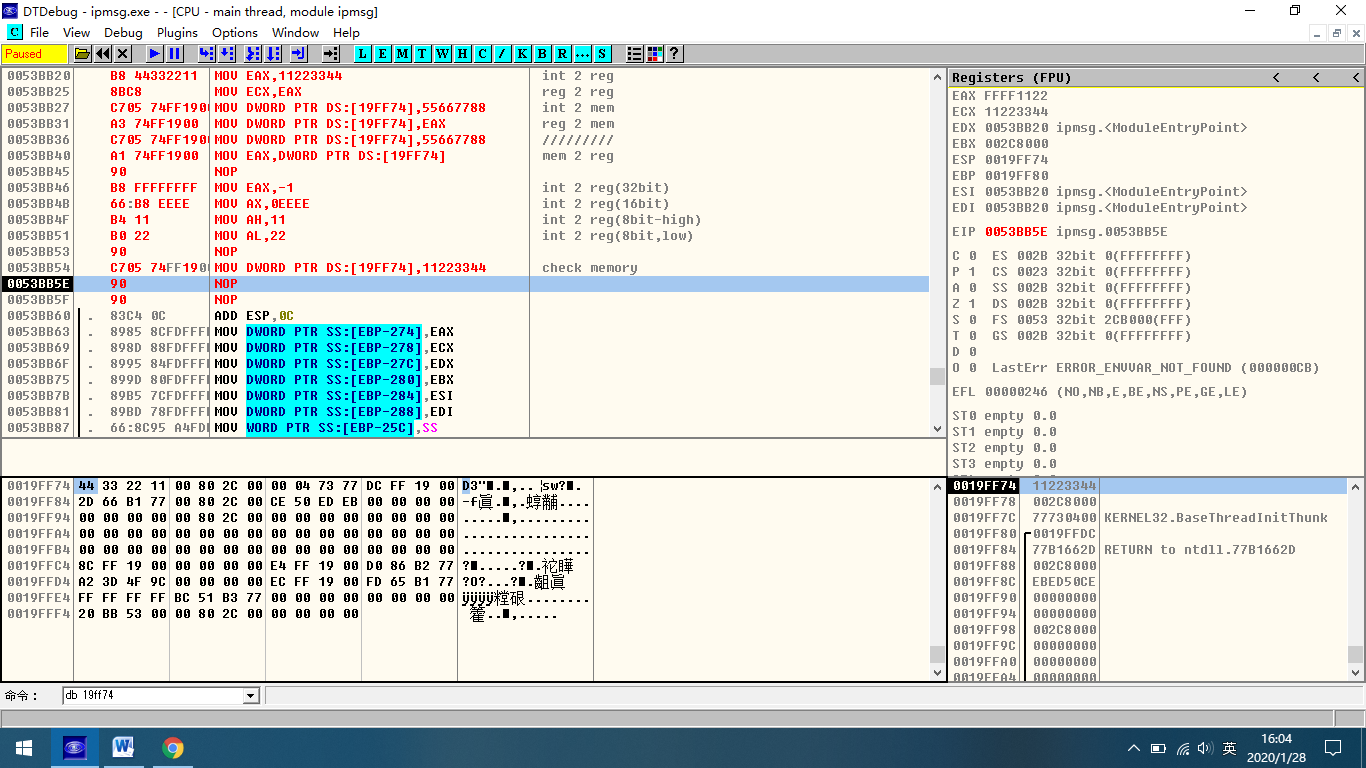
三、80X86汇编指令

*注意：只要没有特别提到，下列这些指令的参数都可以是三十二位立即数、三十二位寄存器、三十二位内存、十六位立即数、十六位寄存器、十六位内存、八位立即数、八位寄存器、八位内存。*

1. MOV a,b

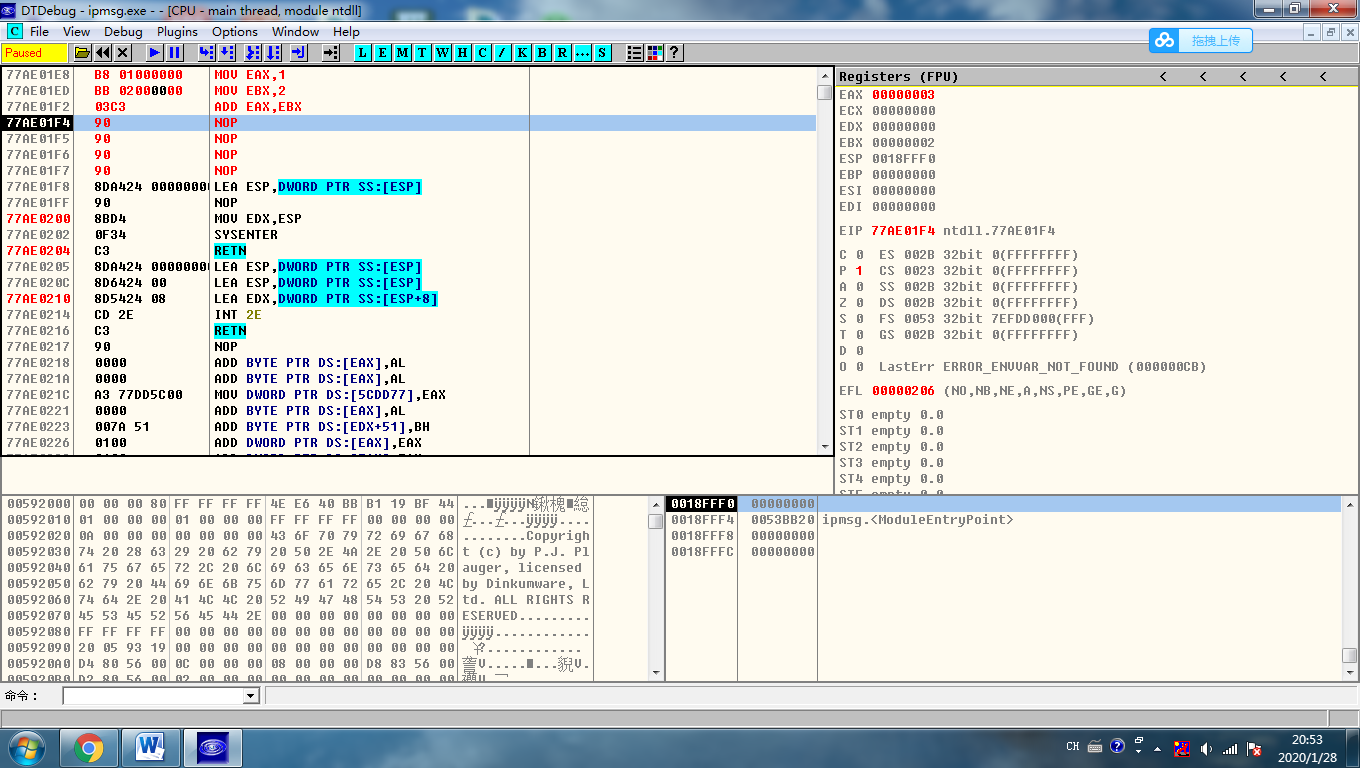
把b写入到a



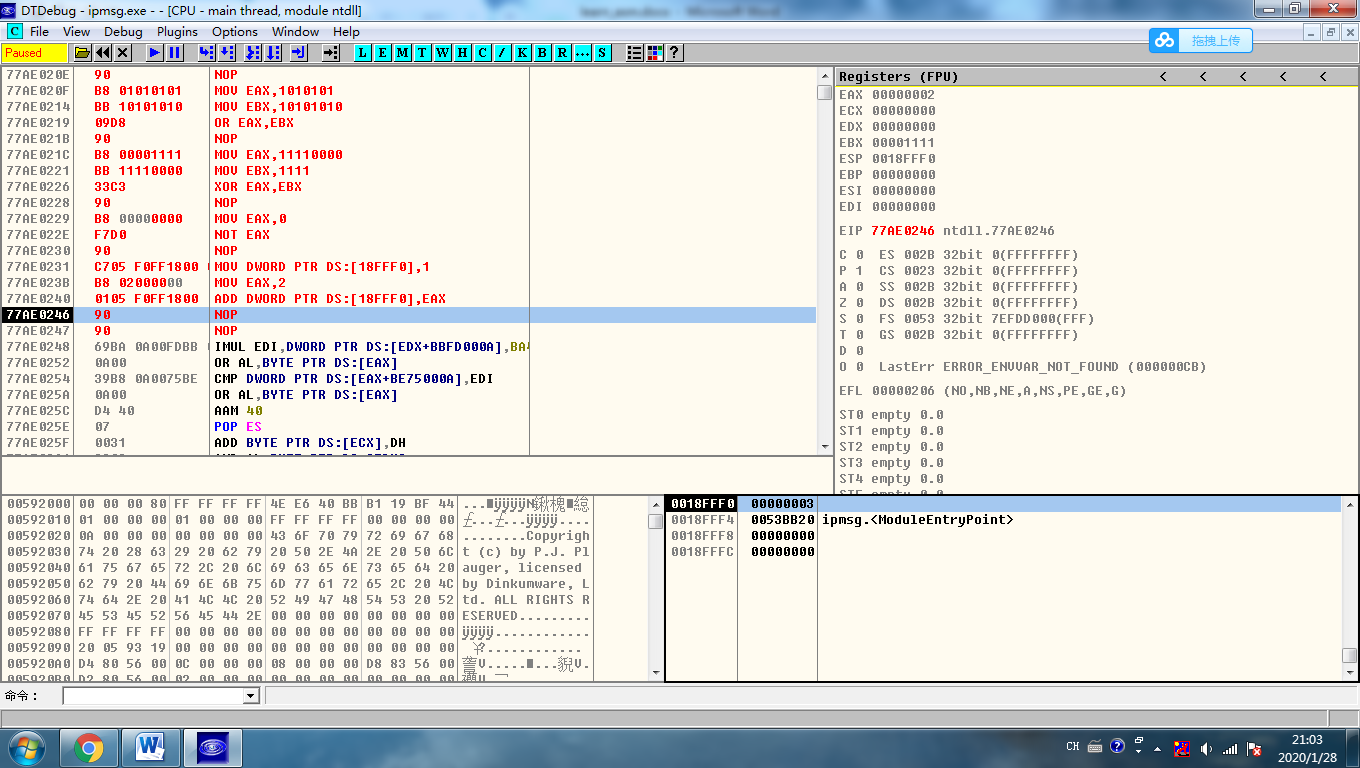


1. ADD a,b

a加b，结果存到a

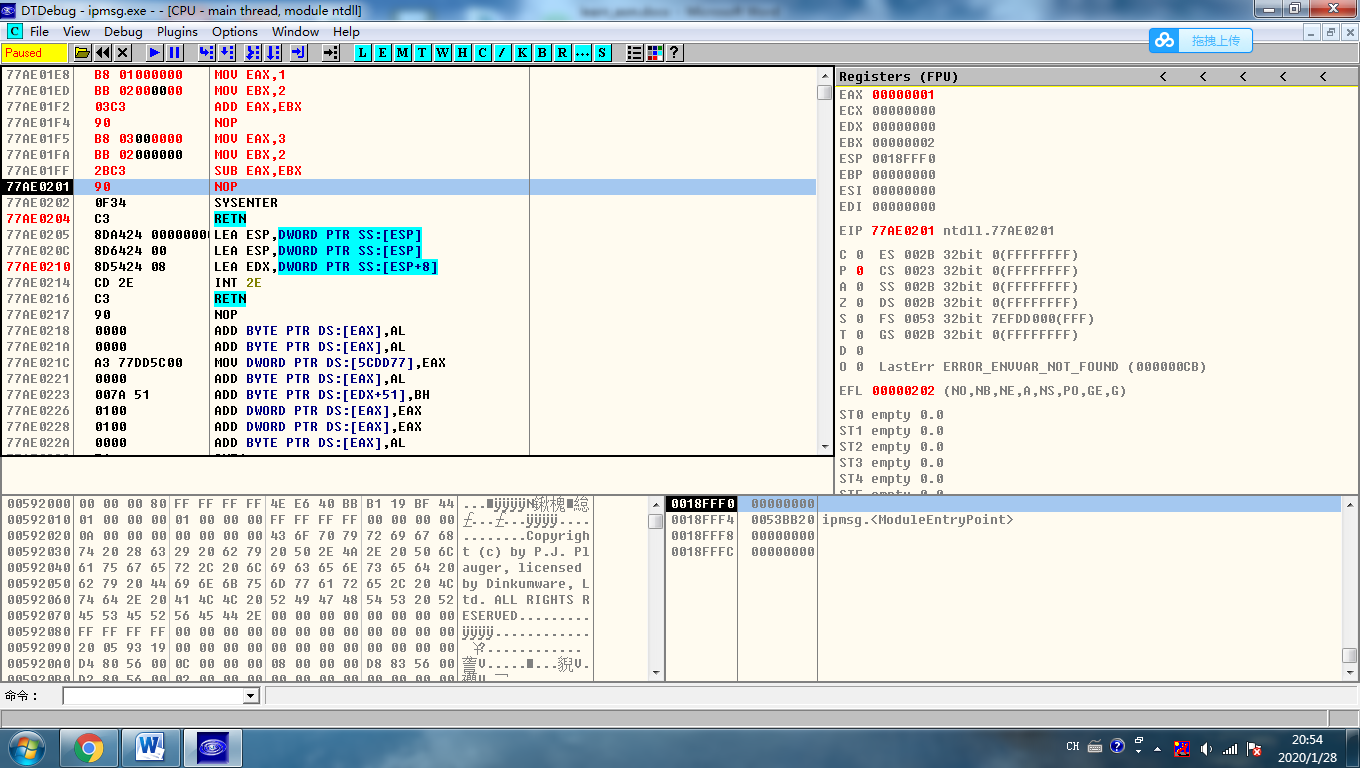


PS:使用内存的ADD。之后的指令如果没有特别说明，也可以有一个参数使用内存地址，但不能两个参数都用内存地址。



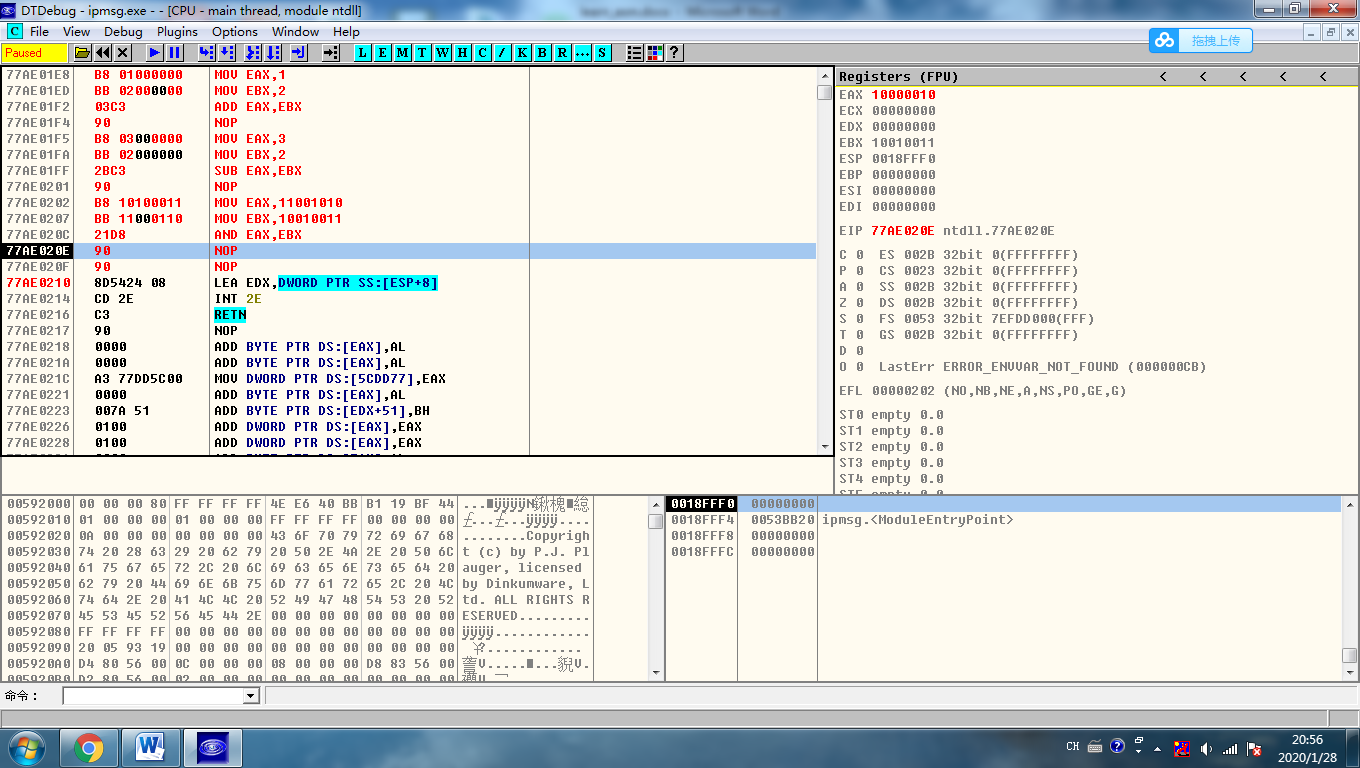
1. SUB a,b （SUB即Subtract的简写）

a减b，结果存到a



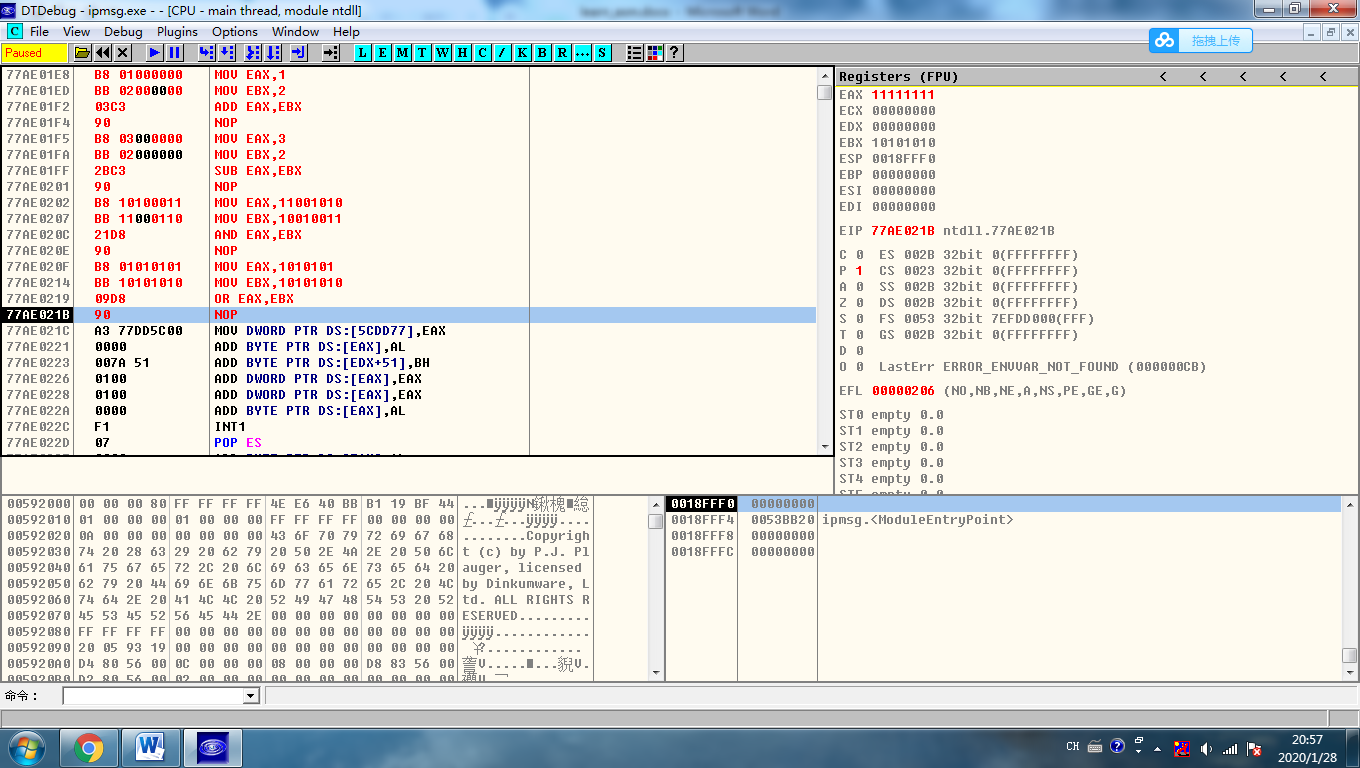
1. AND a,b

a与b，结果存到a



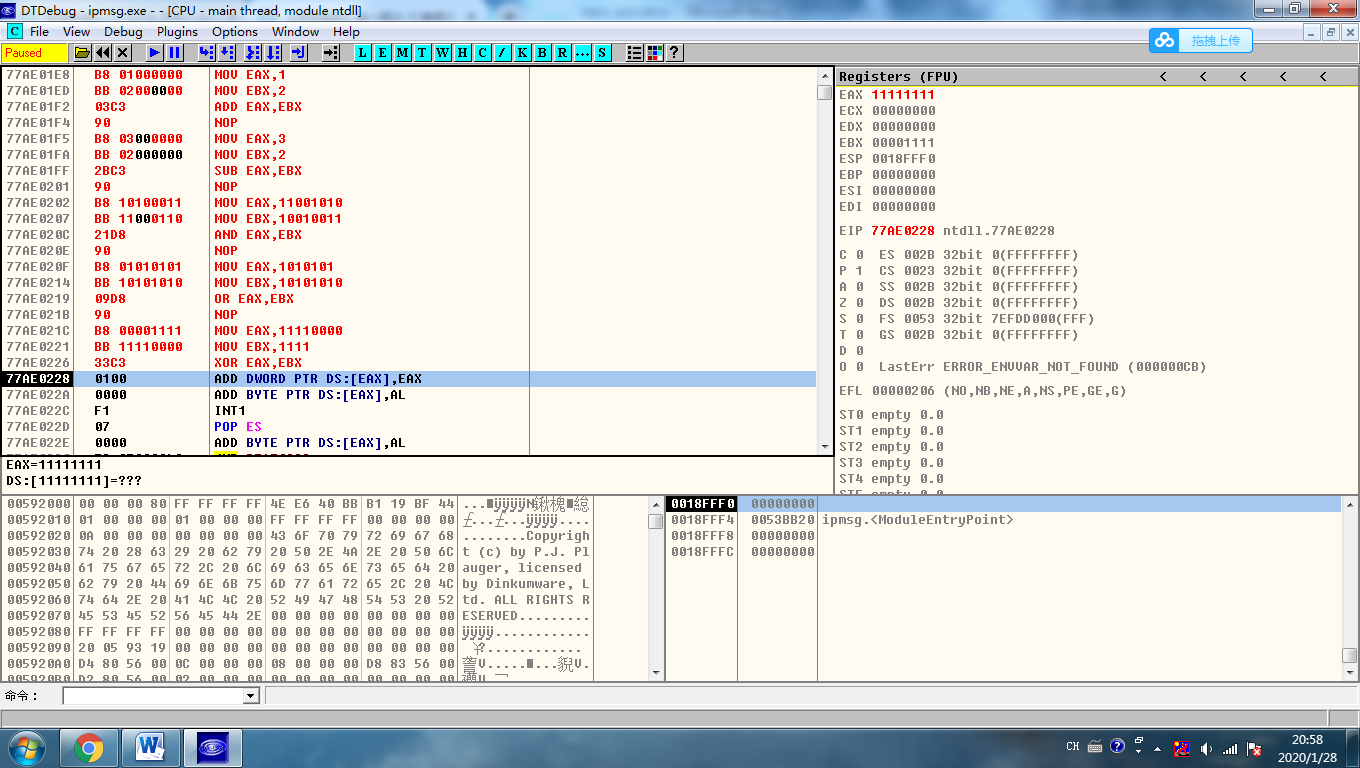
1. OR a,b

a或b，结果存到a



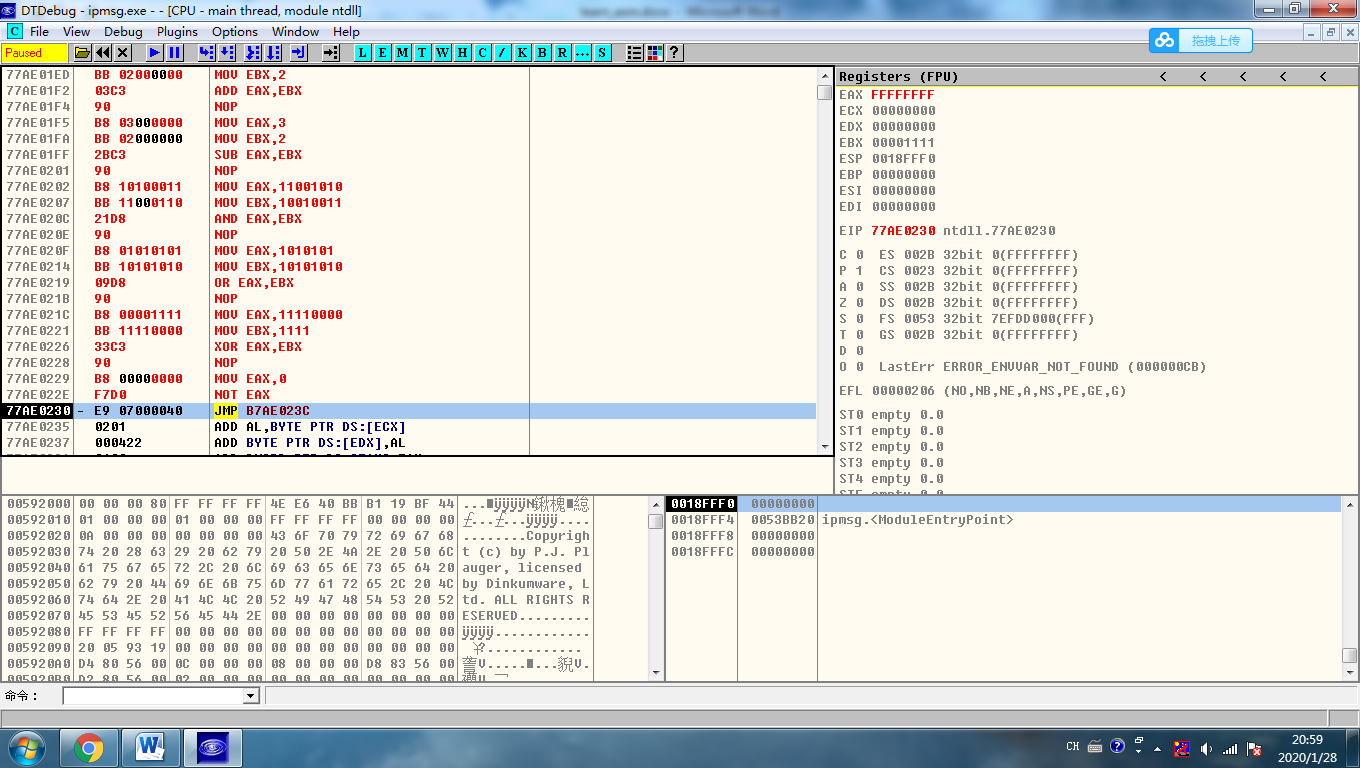
1. XOR a,b

a异或b，结果存到a



1. NOT a

非a，结果存到a



1. MOVS XXX (MOVS全称move string)

MOVS BYTE PTR ES:[EDI],BYTE PTR DS:[ESI] 全等于 MOVSB

MOVS WORD PTR ES:[EDI],WORD PTR DS:[ESI] 全等于 MOVSW

MOVS DWORD PTR ES:[EDI],DWORD PTR DS:[ESI] 全等于 MOVSD

MOVS命令将内存的数据复制到内存的数据。

这个命令很特殊，只有上面这几种写法，也就是说ESI和EDI是不能写成内存地址或寄存器的，只能够写成这两个寄存器。

每执行一句这个命令，ESI和EDI的值都会**增加或减少**一个**BYTE/WORD/DWORD**的长度，也就是**1/2/4**这三个数。

决定到底长度是是**BYTE**、**WORD**还是**DWORD**的因素是MOVS指令后面跟着的是BYTE、WORD还是DWORD；

而决定是增加还是减少的因素，则是EFL（EFlag）寄存器从右往左数的第10个位（也叫DF）的值。如果EFL的第十个位（DF）是0，那就增加；如果第十个位是1，那就减少。

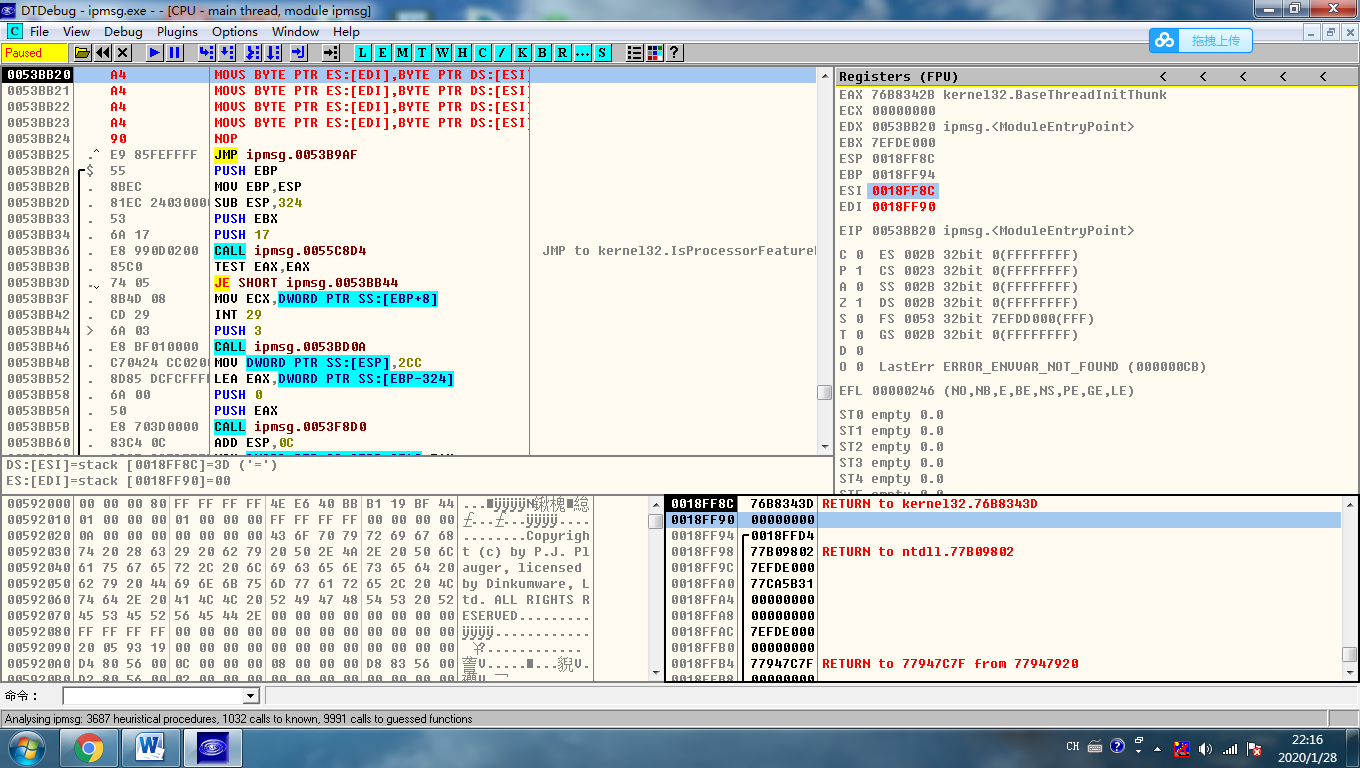
注意！第十个位是要从0开始数的，所以他就是普通人说的第十一个位！！！

Address: 31 30 29 28 …… 11 10 9 8 …… 6 5 4 3 2 1 0

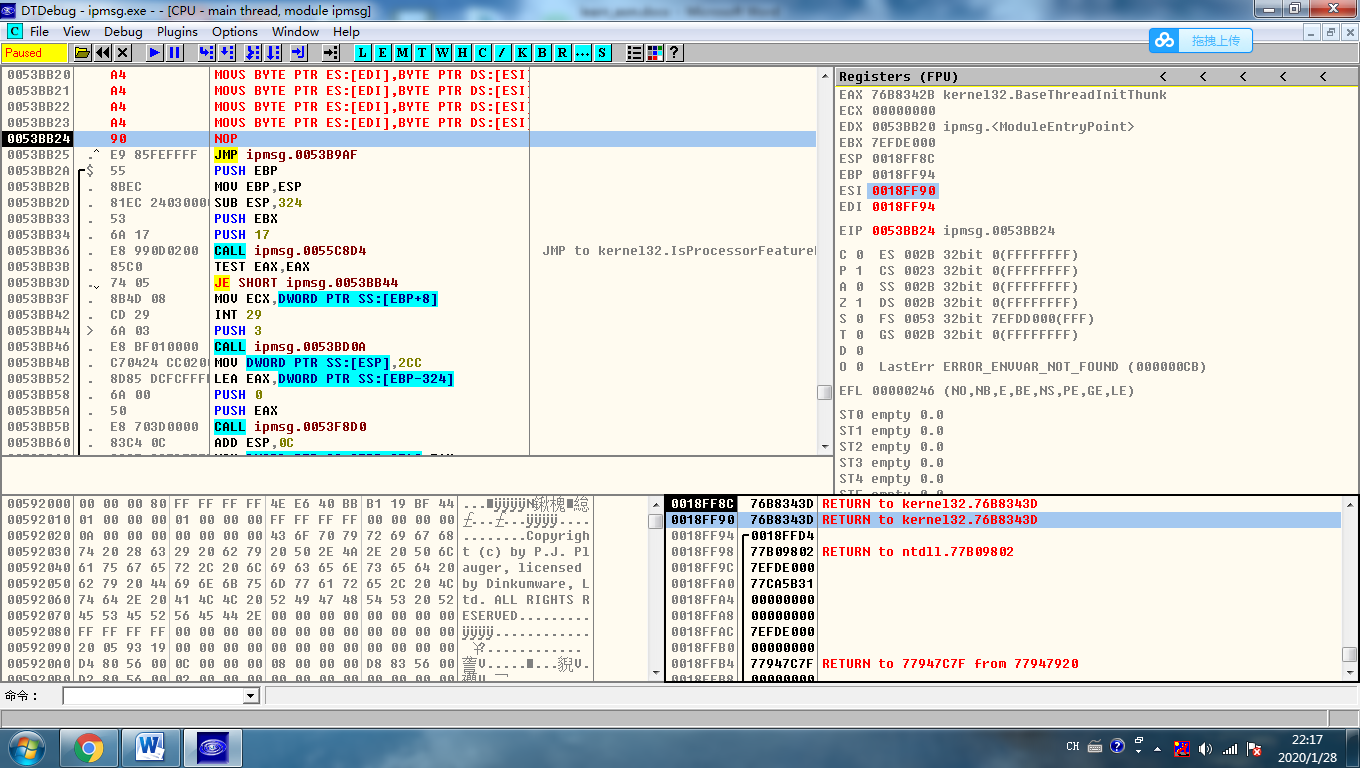
..Value: .0 .0 .0 .0 …… .0 .0 0 0 …… 0 0 0 0 0 0 0

关于MOVSB这样的简写，编译器会自动把他编译成MOVS BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]这句话，他们在编译后的二进制文件中是一样的。

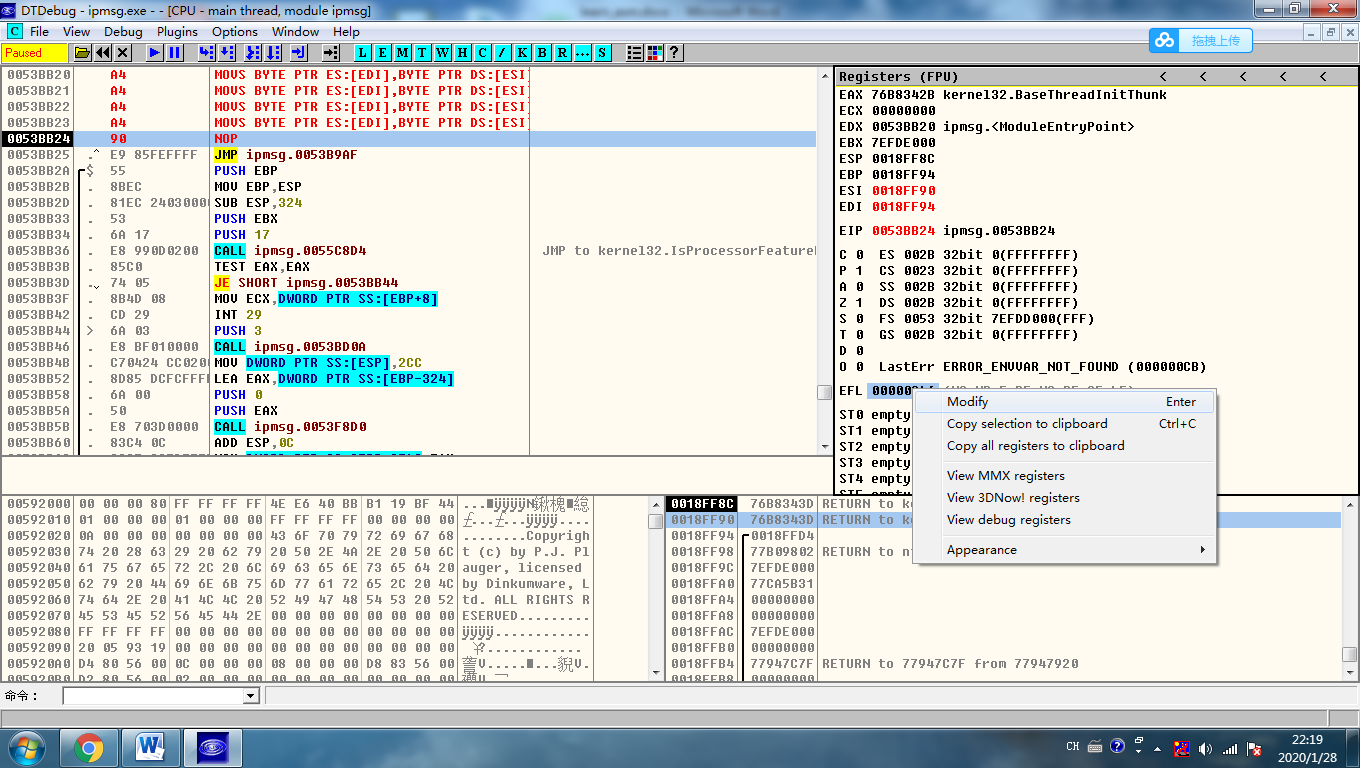
(1).将esi和edi的值设好，输入命令后单步调试



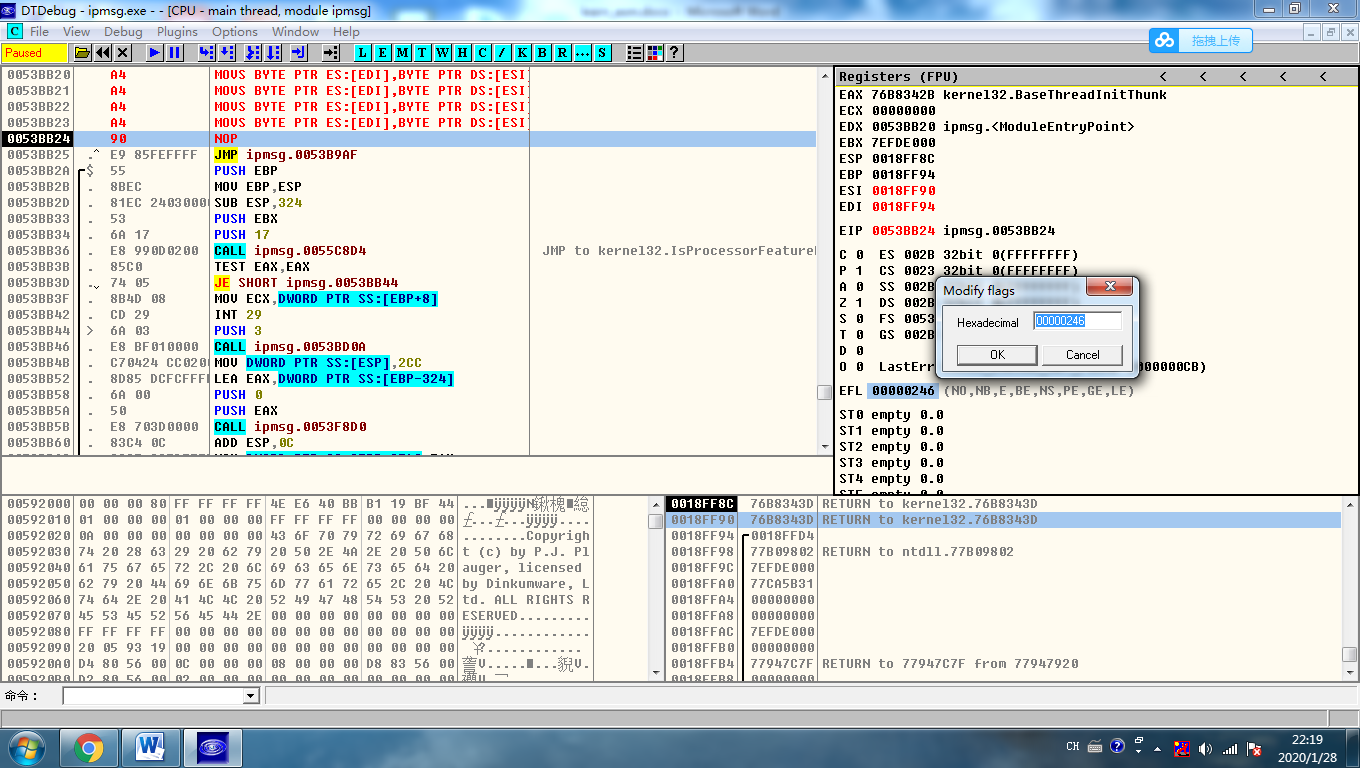
(2).运行完四条命令后，可以看到内存已经成功被复制，ESI和EDI也已经自动增加了四个byte的长度



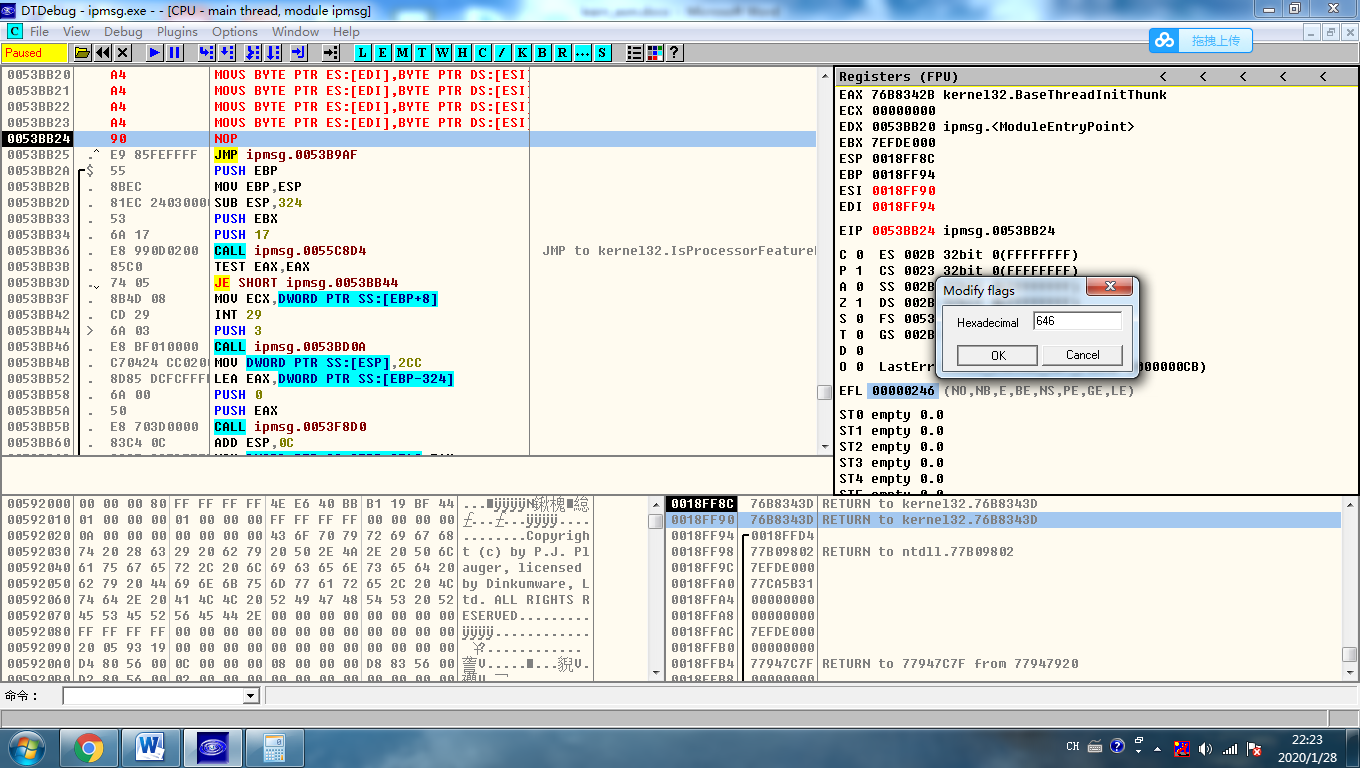
(3).对EF进行Modify



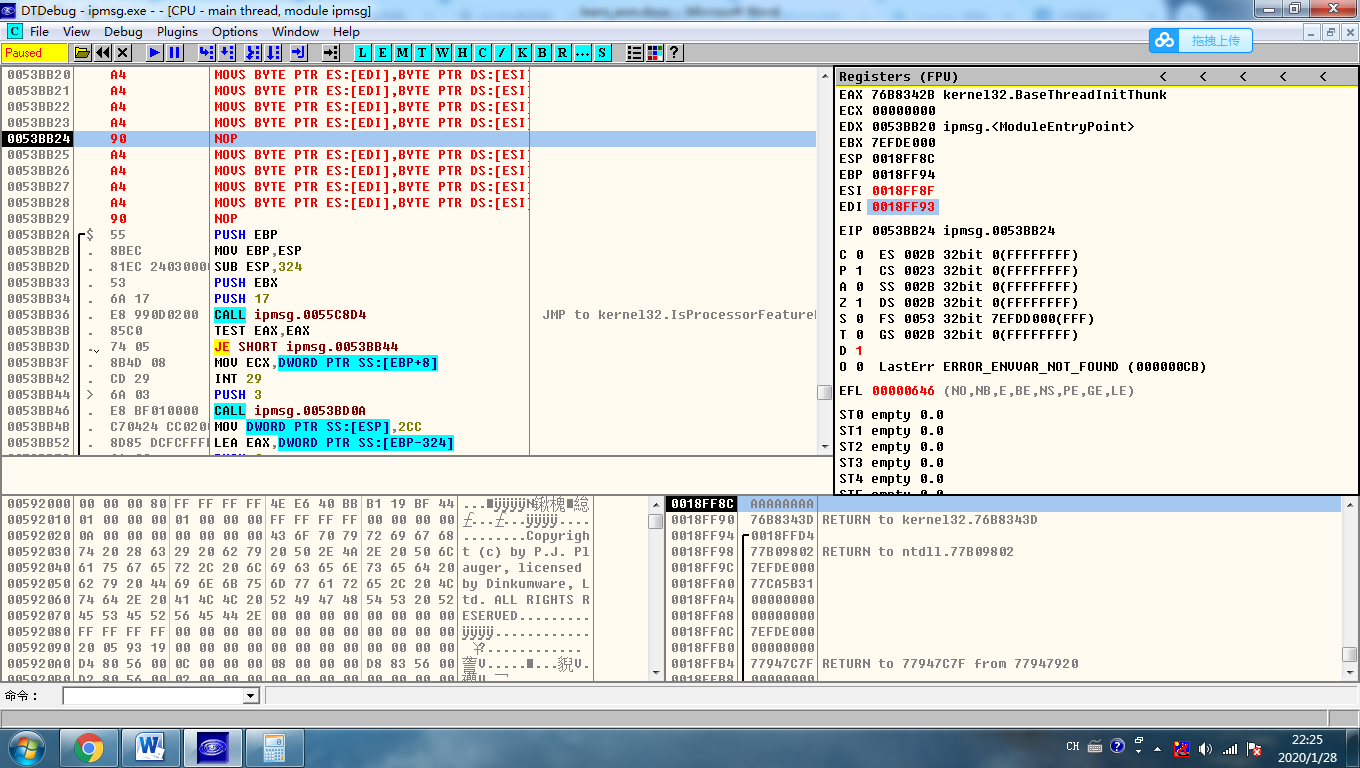
(4).可以看到当前的值是0x00000246，转换成二进制是0000 0000 0000 0000 0000 0010 0100 0110



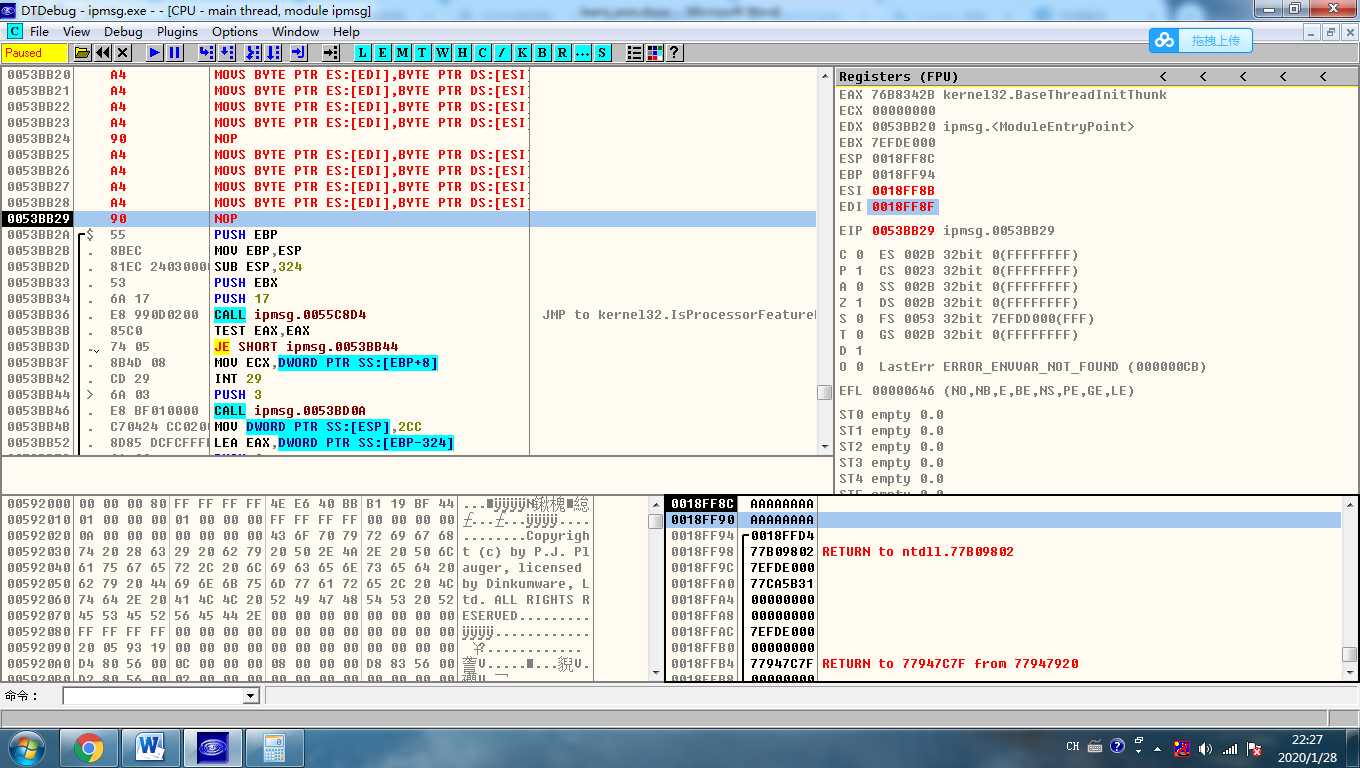
(5). 0000 0000 0000 0000 0000 0010 0100 0110从右往左数十个的位是0，把它改成1得到0000 0000 0000 0000 0000 0110 0100 0110，转换成十六进制是646，把它填进去：



(6).在把ESI和EDI各减1，这是他们指向刚才那块内存的最后一个byte的位置，把ESI所在内存的值设为0xAAAAAAAA，观察MOVS复制的效果



(7).MOVS把内存倒着复制过去了，ESI和EDI也都减了4个byte的长度，搞定！



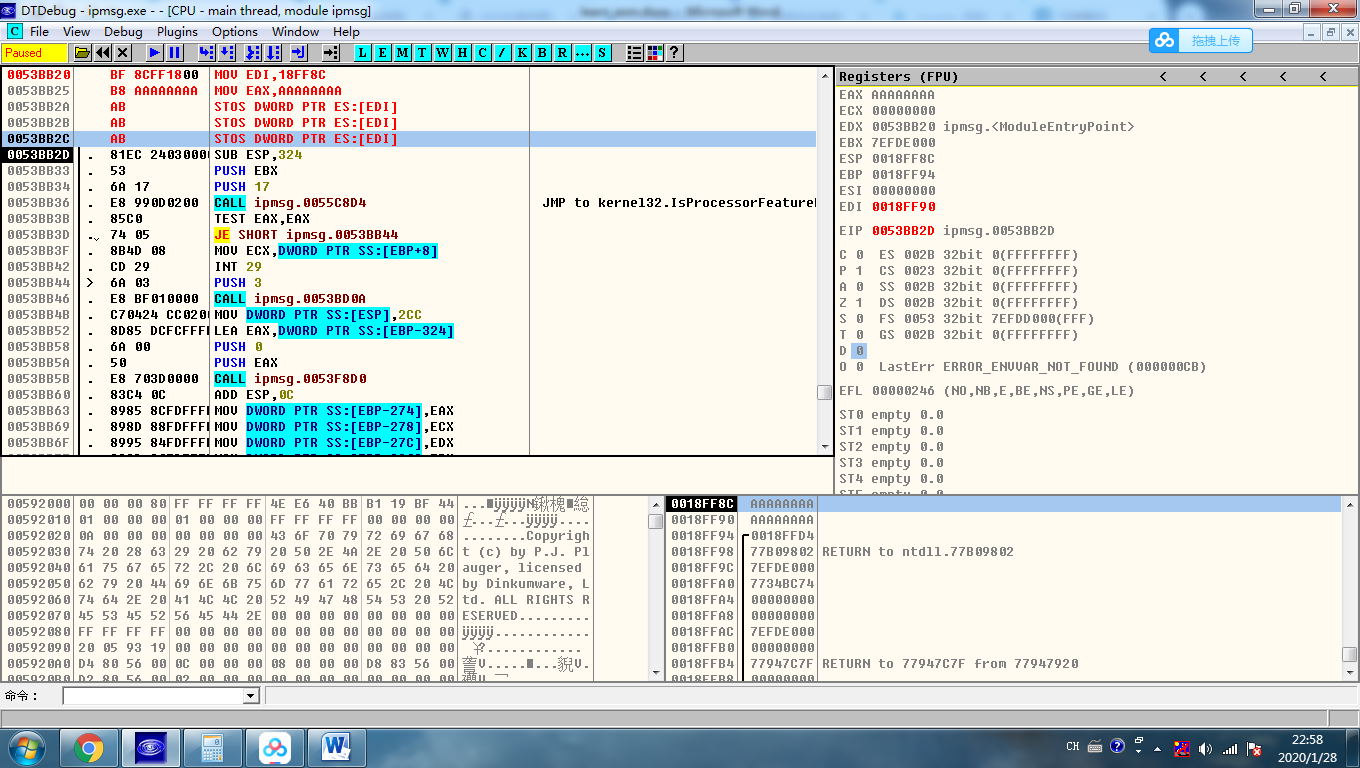
1. STOS XXX (STOS全称store string)

STOS BYTE PTR ES:[EDI] 全等于 STOSB

STOS WORD PTR ES:[EDI] 全等于 STOSW

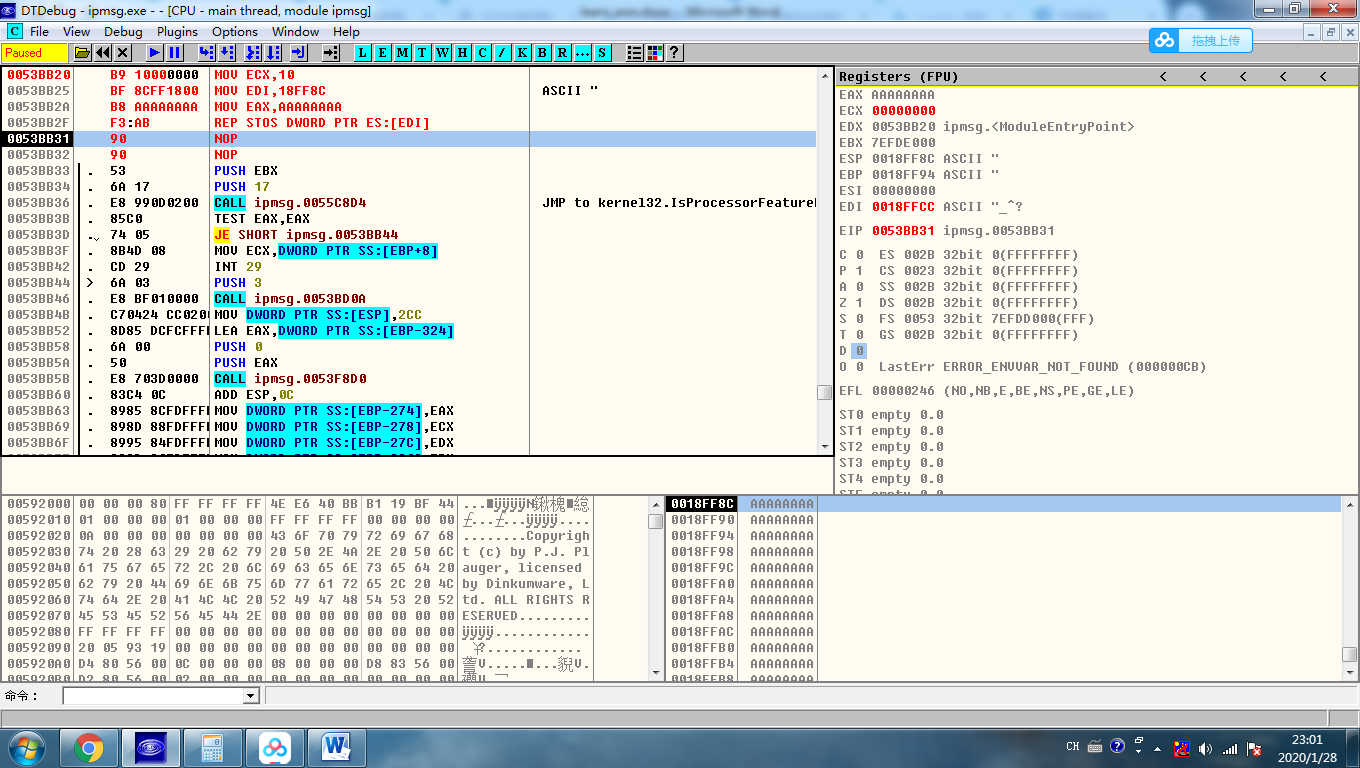
STOS DWORD PTR ES:[EDI] 全等于 STOSD

把寄存器EAX的值存到地址为EDI的内存中，并将EDI加或减一个BYTE/WORD/DWORD的值（参见MOVS）。



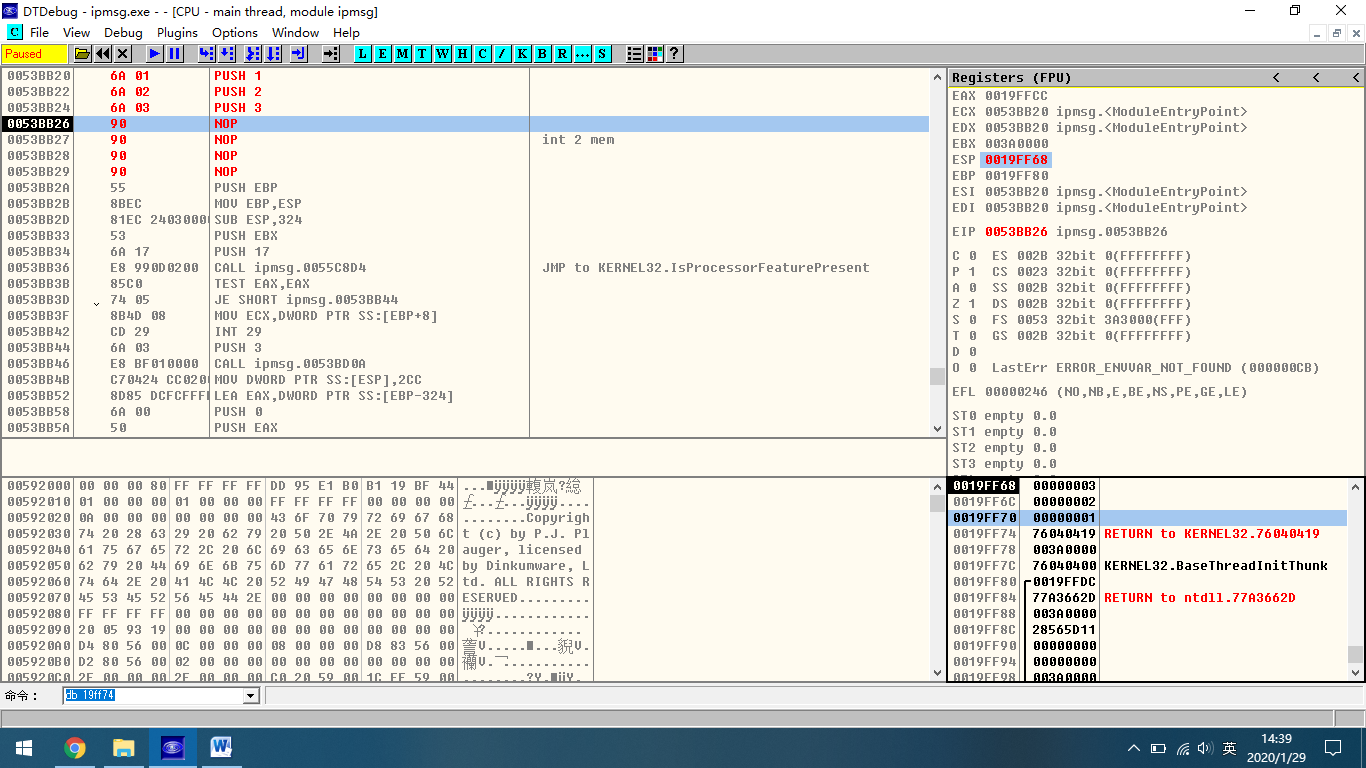
1. REP XXX (REP全称repeat)

重复执行XXX指令ECX次。例子：REP STOSD



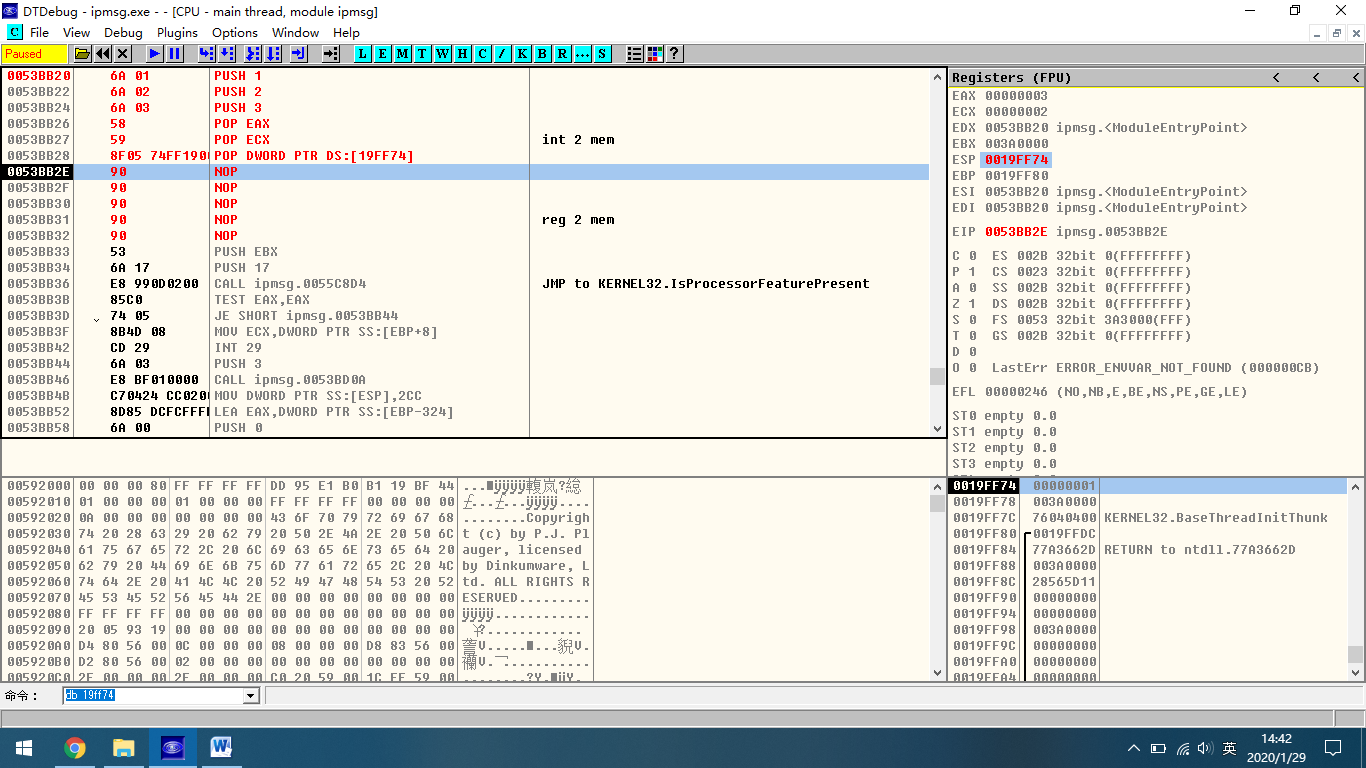
1. PUSH a

将a压入堆栈，并且前移栈顶指针ESP的地址。



1. POP a

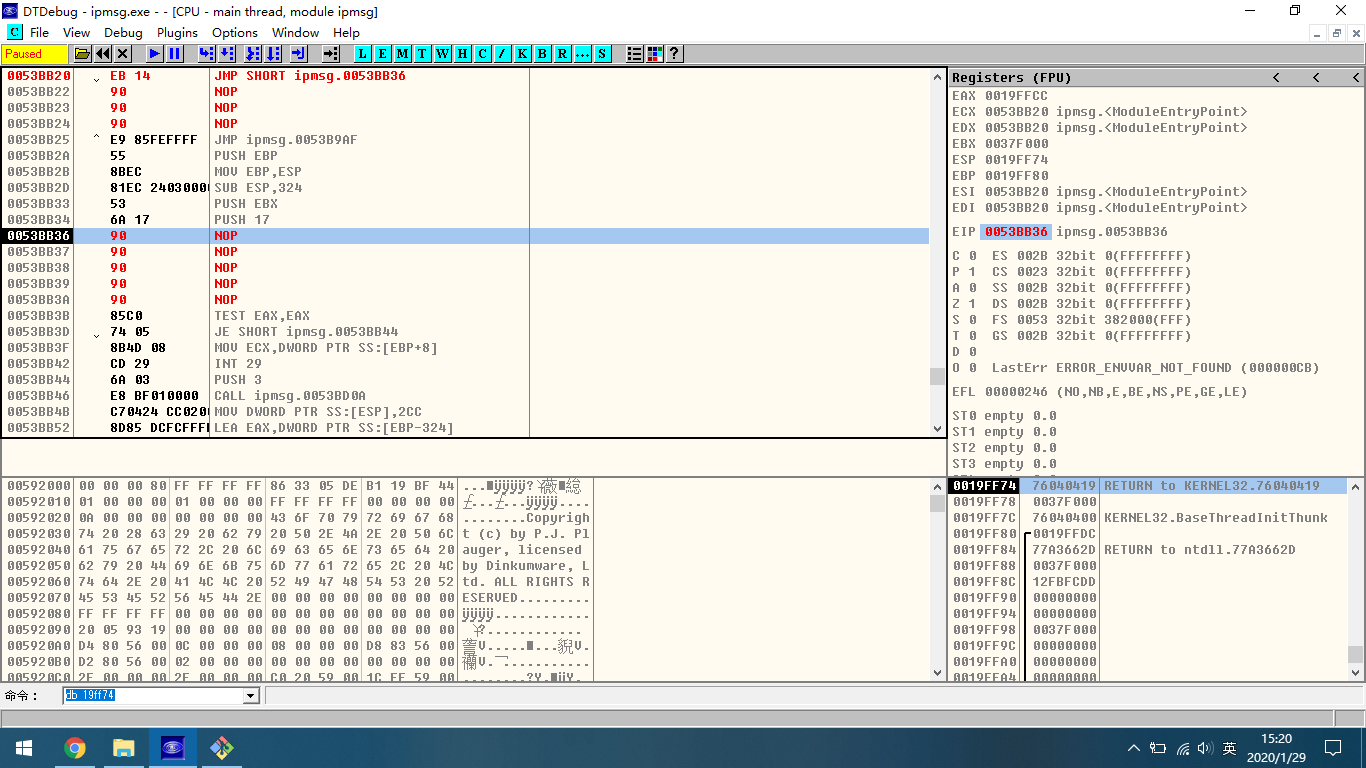
弹出栈顶到寄存器或内存a，并后移栈顶指针ESP的地址。



1. JMP a

跳转到第a行代码。

EIP寄存器记录着下一行要执行的代码的地址，JMP a实际上就是MOV EIP,a。只是Intel不提供这种MOV指令，所以就要使用JMP指令。

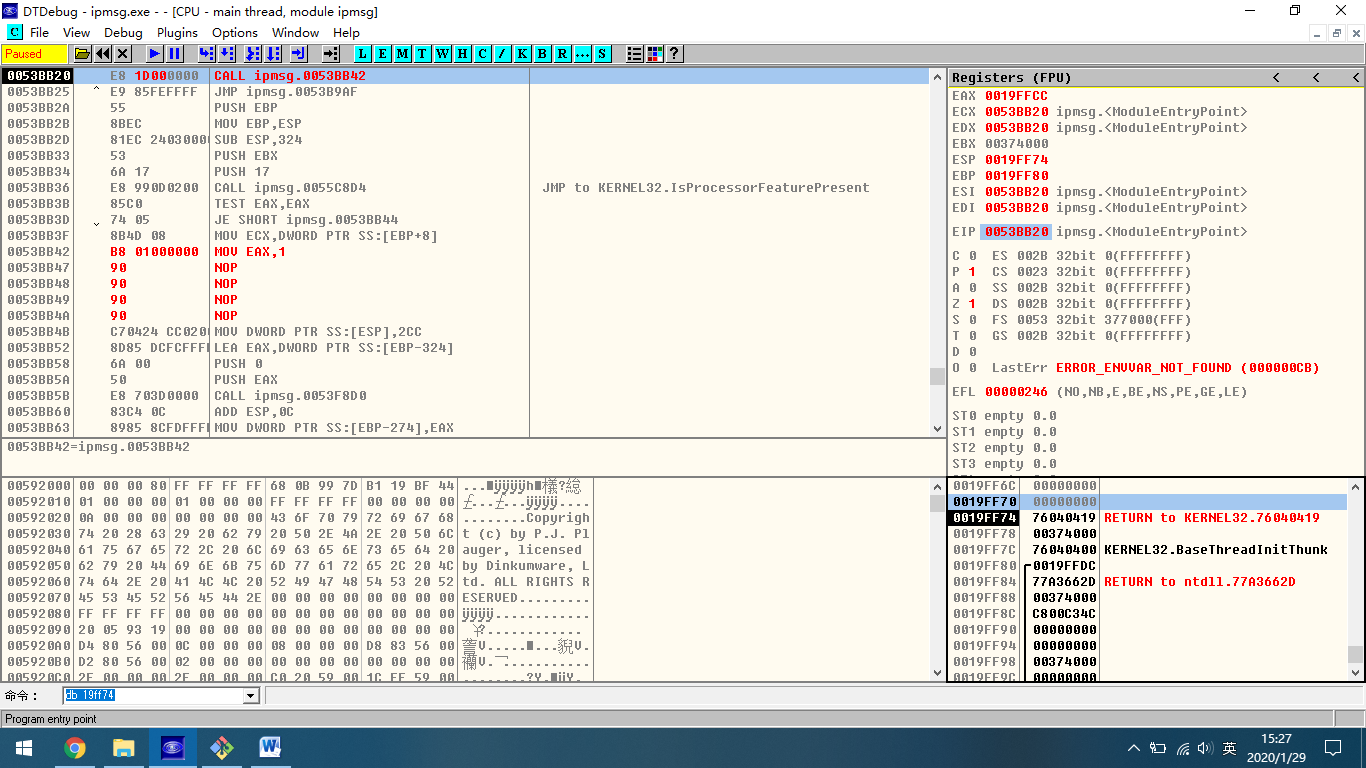


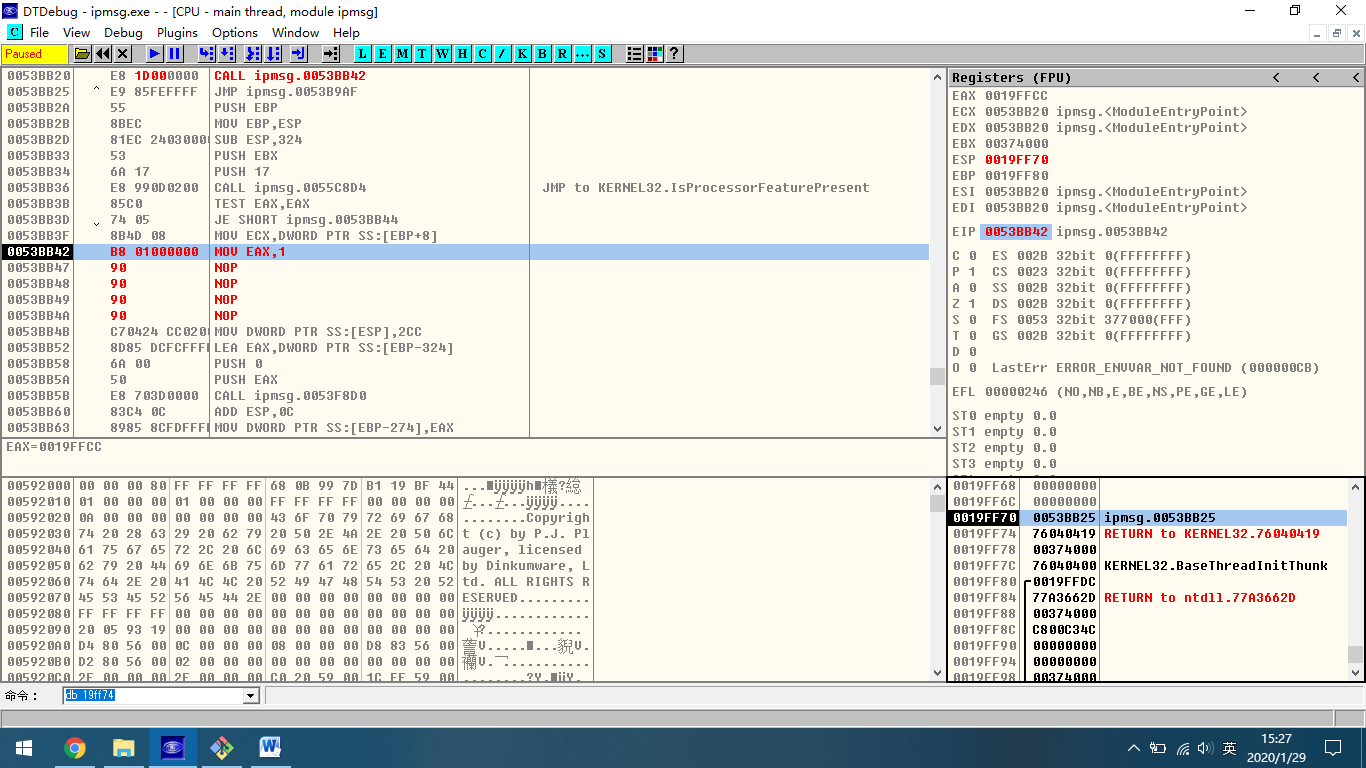
1. CALL a（调试时用F7单步执行，而不是其他命令的F8）

保存当前状态并跳转到代码的第a行。

CALL指令实际上是JMP+PUSH指令。

他会JMP跳转到代码的第a行，然后PUSH CALL指令的下一行的地址。





1. RET（RET全程Return）

返回CALL之前的状态。

RET实际上是JMP+POP。

他会跳转到之前CALL指令保存在堆栈中的“下一行”的地址，然后弹出这个“下一行”地址。

