第二章　基础知识

知识点一：汇编—内存寻址方式
知识点二：汇编—主要指令
知识点三：汇编—函数调用示例

学习之前，回顾两个问题

函数调用的四个步骤

参数入栈　　返回地址入栈
代码区跳转　　栈帧调整

常见寄存器

ESP\EBP　　EIP　　IR
EAX\EBX\ECX\EDX　　ESI\EDI
标志寄存器三个标志位：ZF (零标志)、OF(溢出标志)、CF(进位标志)。

通过寄存器、汇编，看看写的VC程序是如何工作的?

知识点一：
汇编—寻址方式

- **寻址方式就是处理器根据指令中给出的地址信息来寻找有效地址的方式，是确定本条指令的数据地址以及下一条要执行的指令地址的方法。**在存储器中，操作数或指令字写入或读出的方式，有地址指定方式、堆栈存取方式等。
- 几乎所有的计算机，在内存中都采用地址指定方式。
- **当采用地址指定方式时，形成操作数或指令地址的方式称为寻址方式。**

指令寻址

指令的寻址方式有以下两种。

顺序寻址方式。由于指令地址在内存中按顺序安排，当执行一段程序时，通常是一条指令接一条指令地顺序进行。也就是说，从存储器取出第1条指令，然后执行这条指令；接着从存储器取出第2条指令，再执行第二条指令；接着再取出第3条指令。这种程序顺序执行的过程，称为指令的顺序寻址方式。

指令寻址

通常，需要使用指令计数器来完成顺序指令寻址。指令计数器是计算机处理器中的一个包含当前正在执行指令地址的寄存器，在X86架构中称为指令指针IP（Instruction Pointer）寄存器，在ARM或C51架构中也称为程序计数器（PC）。

- 每执行完一条指令时，指令计数器中的地址或自动加1或由转移指针给出下一条指令的地址。

跳跃寻址方式。当程序转移执行的顺序时，指令的寻址就采取跳跃寻址方式。**所谓跳跃，是指下条指令的地址码不是由程序计数器给出，而是由本条指令给出。**注意，程序跳跃后，按新的指令地址开始顺序执行。因此，程序计数器的内容也必须相应改变，以便及时跟踪新的指令地址。

采用指令跳跃寻址方式，可以实现程序转移或构成循环程序，从而能缩短程序长度，或将某些程序作为公共程序引用。指令系统中的各种条件转移或无条件转移指令，就是为了实现指令的跳跃寻址而设置的。注意跳跃的结果是当前指令修改PC程序计数器的值，所以下一条指令仍是通过程序计数器PC给出。

操作数寻址

形成操作数的有效地址的方法称为操作数的寻址方式。由于大型机、小型机、微型机和单片机结构不同，从而形成了各种不同的操作数寻址方式。

为了便于解释，使用汇编语言MOV指令，其用法为

MOV 目的操作数, 源操作数

表示将一个数据从源地址传送到目标地址。

立即寻址

指令的地址字段给出的不是操作数的地址，而是操作数本身，这种寻址方式称为立即寻址。立即寻址方式的特点是指令执行时间很短，因为它不需要访问内存取数，从而节省了访问内存的时间。

如：MOV CL, 05H

表示将05H这个数值存储到CL寄存器中。

直接寻址

直接寻址是一种基本的寻址方法，其特点是在指令中直接给出操作数的有效地址。**由于操作数的地址直接给出而不需要经过某种变换，所以称这种寻址方式为直接寻址方式。**

如：MOV AL,[3100H]

表示将地址[3100H]中的数据存储到AL中

注意：地址要写在括号“[”，“]”内。

直接寻址

如：MOV AL,[3100H]

在通常情况下，操作数存放在数据段中。所以，默认情况下操作数的物理地址由数据段寄存器 DS 中的值和指令中给出的有效地址直接形成。

- 上述指令中，操作数的物理地址应为DS:3100H。
- 但是如果在指令中使用段超越前缀指定使用的段，则可以从其他段中取出数据，如：MOV AL, ES:[3100H]。

间接寻址

间接寻址是相对直接寻址而言的，在间接寻址的情况下，**指令地址字段中的形式地址不是操作数的真正地址，而是操作数地址的指示器**，或者说此形式地址单元的内容才是操作数的有效地址。

如：MOV [BX], 12H

这是一种寄存器间接寻址，BX寄存器存操作数的偏移地址，操作数的物理地址应该是DS:BX。表示将12H这个数据存储到DS:BX中。

如果操作数存放在寄存器中，通过指定寄存器来获取数据，则称为**寄存器寻址**。如：MOV BX, 12H表示将12H这个数据存储到BX寄存器中。

相对寻址

操作数的有效地址是一个基址寄存器（BX, BP）或变址寄存器（SI, DI）的值加上指令中给定的偏移量之和。

如：MOV AX, [DI + 1234H]
操作数的物理地址应该是DS: DI + 1234H。

与间接寻址相比，可以认为**相对寻址是在间接寻址基础上，增加了偏移量**。

基址变址寻址

将基址寄存器的内容，加上变址寄存器的内容而形成操作数的有效地址。

如：MOV EAX, [EBX+ESI]。

也可以写成MOV EAX, [BX][SI] 或MOV EAX, [SI][BX]

相对基址变址寻址

在基址变址寻址方式融合相对寻址方式，即增加偏移量

如：MOV EAX, [EBX+ESI+1000H]

也可以写成MOV EAX, 1000H [BX][SI]

操作数寻址

例 CPU内部寄存器和存储器之间的数据传送

```
MOV [BX], AX              ;间接寻址                          (16位)
MOV EAX, [EBX+ESI]        ;基址变址寻址                      (32位)
MOV AL, BLOCK             ;BLOCK为变量名，直接寻址    (8位)
```

知识点二：
汇编—主要指令

主要指令

这里对常用的部分指令进行回顾。

大部分指令有两个操作符 (例如：add EAX, EBX)，有些是一个操作符 (例如：not EAX)，还有一些是三个操作符 (例如：IMUL EAX, EDX, 64)。

汇编语言主要指令

MOV: 把源操作数送给目的操作数，其语法为: MOV 目的操作数,源操作数

XCHG: 交换两个操作数的数据

PUSH,POP: 把操作数压入或取出堆栈

PUSHF,POPF,PUSHA,POPA: 堆栈指令群

LEA,LDS,LES: 取地址至寄存器

数据传送指令

MOV语法：MOV 目的操作数,源操作数

mov al,[3100H]; 表示将3100H中的数值写入AL寄存器

LEA 语法：LEA 目的数, 源数

将有效地址传送到指定的寄存器

lea eax, dword ptr [4*ecx+ebx]

源数为”dword ptr [4*ecx+ebx]”，即地址为4*ecx+ebx里的数值，dword ptr是告诉地址里的数值是一个dword型数据。上述lea语句则是将源数的地址4*ecx+ebx赋值给eax。

汇编语言主要指令

位运算指令集

- AND,OR,XOR,NOT,TEST: 执行BIT与BIT之间的逻辑运算
- SHR,SHL,SAR,SAL: 移位指令
- ROR,ROL,RCR,RCL: 循环移位指令

AND (逻辑与)语法: AND 目标数, 源数

AND运算对两个数进行逻辑与运算（当且仅当两操作数对应位都为“1”时结果的相应位为“1”，否则结果相应位为“0”），目标数=目标数 AND 原数。

AND指令会清空OF、CF标记，设置ZF标记

算数运算指令

ADD, ADC	加法指令
SUB,SBB	减法指令
INC, DEC	把OP的值加一或减一
NEG	将OP的符号反相(取二进制补码)
MUL,IMUL	乘法指令
DIV,IDIV	除法指令

ADD语法: ADD 被加数, 加数

加法指令将一个数值加在一个寄存器上或者一个内存地址上

add eax,123; 相当于 eax=eax+123

加法指令对ZF、OF、CF都会有影响

程序流程控制指令集

- **CMP** → 比较OP1与OP2的值
- **JMP** → 跳往指定地址执行
- **LOOP** → 循环指令集
- **CALL,RET** → 子程序调用,返回指令
- **INT,IRET** → 中断调用及返回指令。在执行INT时,CPU会自动将标志寄存器的值入栈,在执行IRET时则会将堆栈中的标志值弹回寄存器
- **REP, REPE, REPNE** → 重复前缀指令集

程序流程控制指令集

CMP语法: CMP 目标数, 源数

CMP指令比较两个值并且标记CF、OF、ZF：

```
CMP     EAX, EBX          ; 比较eax和ebx是否相等，如果相等就设置ZF为1
```

程序流程控制指令集

CALL语法：CALL something

CALL指令将当前EIP中的指令地址压入栈中，并且调用CALL 后的子程序

CALL 可以这样使用：

```
CALL 404000          ;; 最常见: CALL 地址
CALL EAX             ;; CALL 寄存器 - 如果寄存器存的值为404000，那就等同于第一种情况
```

RET语法：RET

RET指令的功能是从一个代码区域中退出到调用CALL的指令处

条件转移命令

JXX: 当特定条件成立则跳往指定地址执行

字符串操作指令集

MOVSB,MOVSW,MOVSD: 字符串传送指令

CMPSB,CMPSW,CMPSD: 字符串比较指令

SCASB,SCASW: 字符串搜索指令

LODSB,LODSW,STOSB,STOSW: 字符串载入或存贮指令

知识点三：
汇编—函数调用示例

一个简单的C语言程序（VC6，Win32控制台程序），


```
#include <iostream>
int add(int x,int y)
{
        int z=0;
        z=x+y;
        return z;
}
void main()
{
        int n=0;
        n=add(1,3);
        printf("%d\n",n);
}
```

在VC6中，使用调试模式，可以通过右键->“转到反汇编”查看所写程序的汇编代码。

具体做法如下：

1、在主函数中设置一个断点，比如在printf该行代码处。

2、按F5进入调试状态。

3.点右键，选择“转到反汇编”。

得到主函数的汇编结果如下：

```
Void main()
{ ********************
            int n=0;
004114OE    mov         dword ptr [n],0
            n=add(1,3);
00411415    push        3
00411417    push        1
00411419    call         add(411096h)
0041141E    add         esp,8
00411421    mov         dwod ptr [n],eax
      printf(“%d\n,n”);
********************
}
```

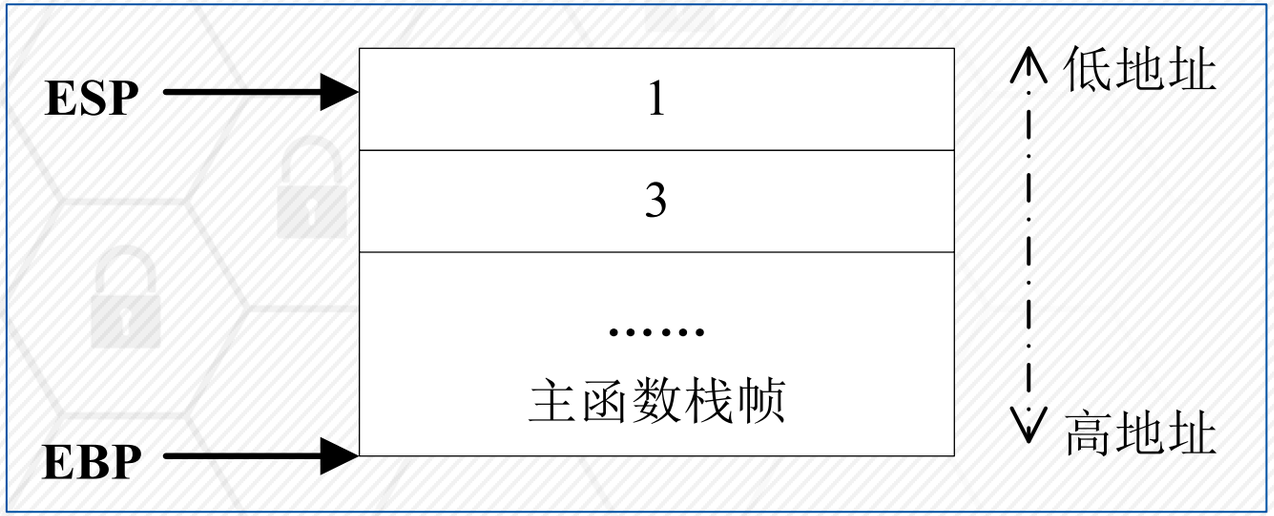
函数调用前：

参数入栈

00411415 push 3

00411417 push 1

将参数入栈，此时栈区状态为：



函数调用时：

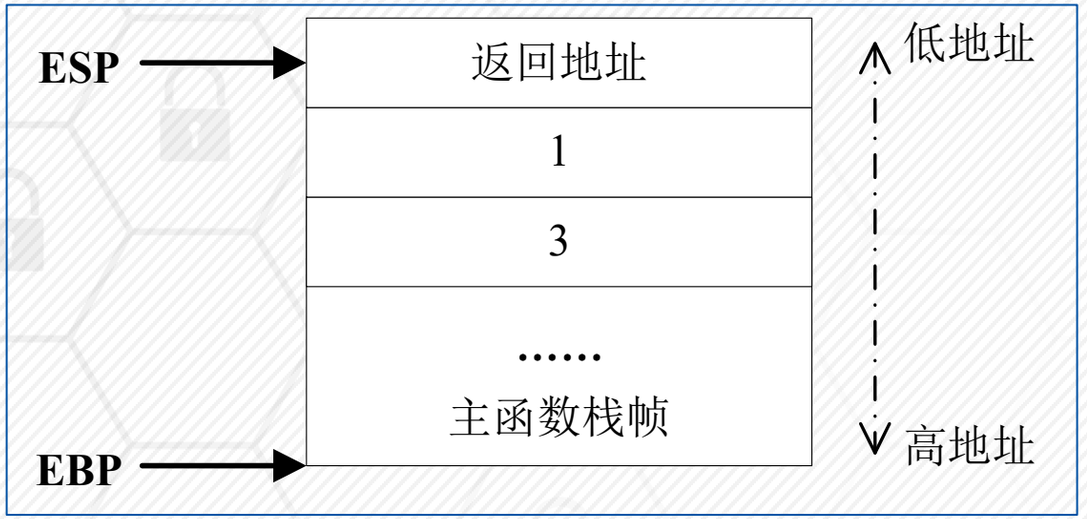
返回地址入栈

00411419 call add (411096h)

函数调用call语句完成两个主要功能：

a）**向栈中压入当前指令在内存中的位置**，即**保存返回地址**；

b）**跳转到所调用函数的入口地址**，即函数入口处。



此时栈区状态为：

Add函数的汇编代码

```
Int  add (int  x,int  y)
{
004113A0    push         ebp
004113A1    mov          ebp,esp
004113A3    sub          esp, 0CCh
004113A9    push         ebx
004113AA    push         esi
004113AB    push         edi
004113AC    lea          edi,[ebp-0CCh]
004113B2    mov          ecx,33h
004113B7    mov          eax,0CCCCCCCCh
004113BC    rep stos     dword ptr  es:[edi]
        int z=0;
004113BE    mov          dword ptr [z],0
        z=x+y;
004113C5    mov          eax,dword ptr [x]
004113C8    add          eax,dword ptr [y]
004113CB    mov          dword ptr [z],eax
        return z;
004113CE    mov          eax,dword ptr[z]
}
```

栈帧切换

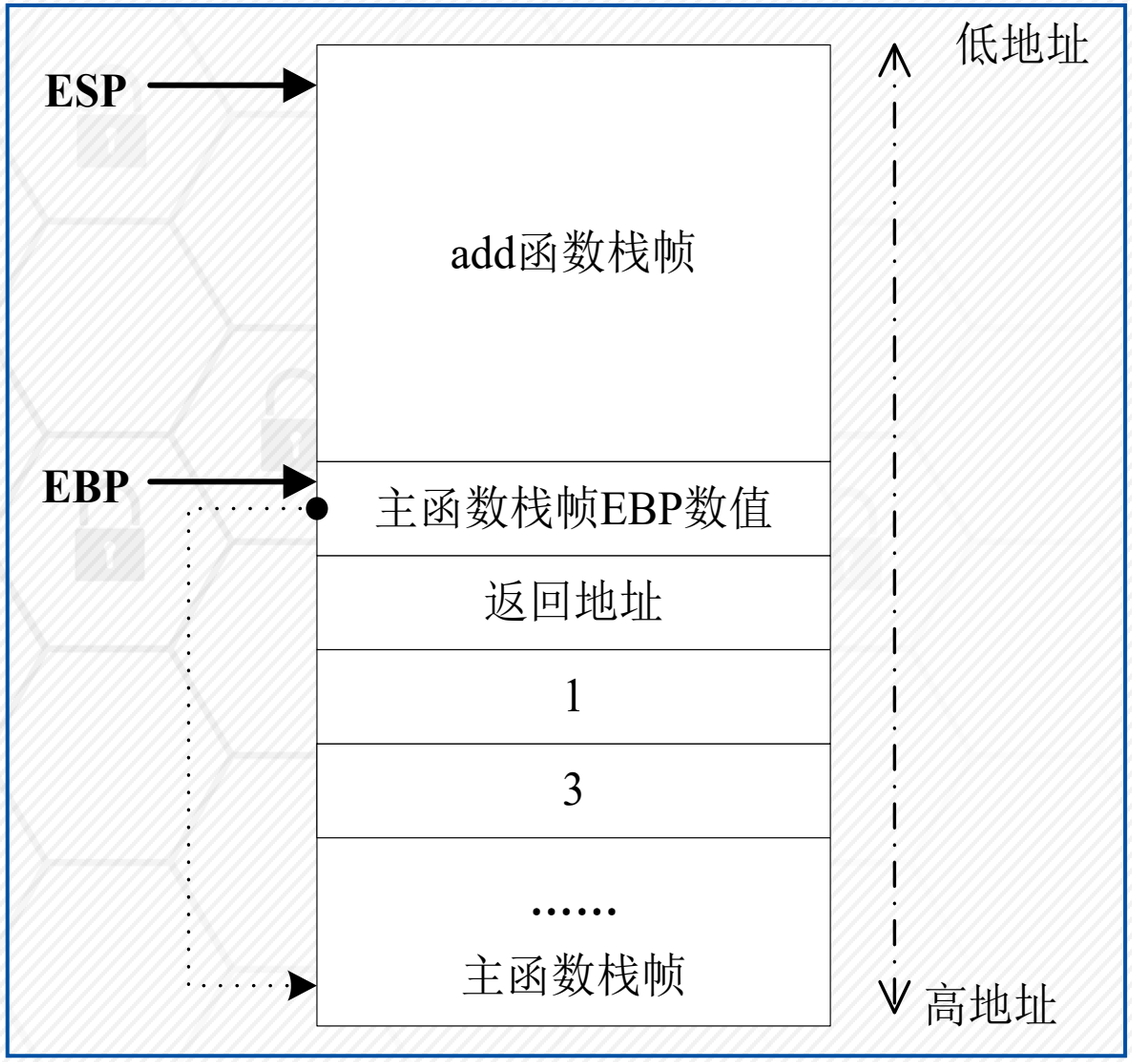
004113A0 push ebp； 将EBP的值入栈。

004113A1 mov ebp,esp； 将ESP的值赋值给EBP。

004113A3 sub esp, 0CCh； 将ESP抬高。

上面三行汇编代码完成了栈帧切换，即保存了主函数栈帧的EBP的值，也通过改变EBP和ESP寄存器的值，为add函数分配了栈帧空间。

此时栈区状态为：



函数状态保存

004113A9 push ebx;
/用于保存现场 ebx作为内存偏移指针使用。

004113AA push esi;
用于保存现场 esi是源地址指针寄存器。

004113AB push edi;
用于保存现场 edi是目的地址指针寄存器。

004113AC lea edi,[ebp-0CCh] ;
将ebp-0CCh地址装入EDI。

栈帧切换

004113B2 mov ecx,33h;
设置计数器数值，即将ECX寄存器赋值为33h

004113B7 mov eax,0CCCCCCCCh;
向寄存器EAX赋值

004113BC rep stos dword ptr es:[edi];
循环将栈区数据都初始化为CCh。其中：

rep指令的目的是重复其上面的指令.ECX的值是重复的次数.

STOS指令的作用是将eax中的值拷贝到ES:EDI指向的地址.

执行函数体

```
 int z=0;
004113BE  mov     dword ptr [z],0 ；  将z初始化为0
          z=x+y;
004113C5  mov     eax,dword ptr [x];
                  将寄存器EAX的值设置为形参x的值
004113C8  add     eax,dword ptr [y] ；
                        将寄存器EAX累加形参y的值
004113CB  mov     dword ptr [z],eax;
                              将EAX的值复制给z
          return z;
004113CE  mov     eax,dword ptr [z];
                        将z的值存储到EAX寄存器中
```

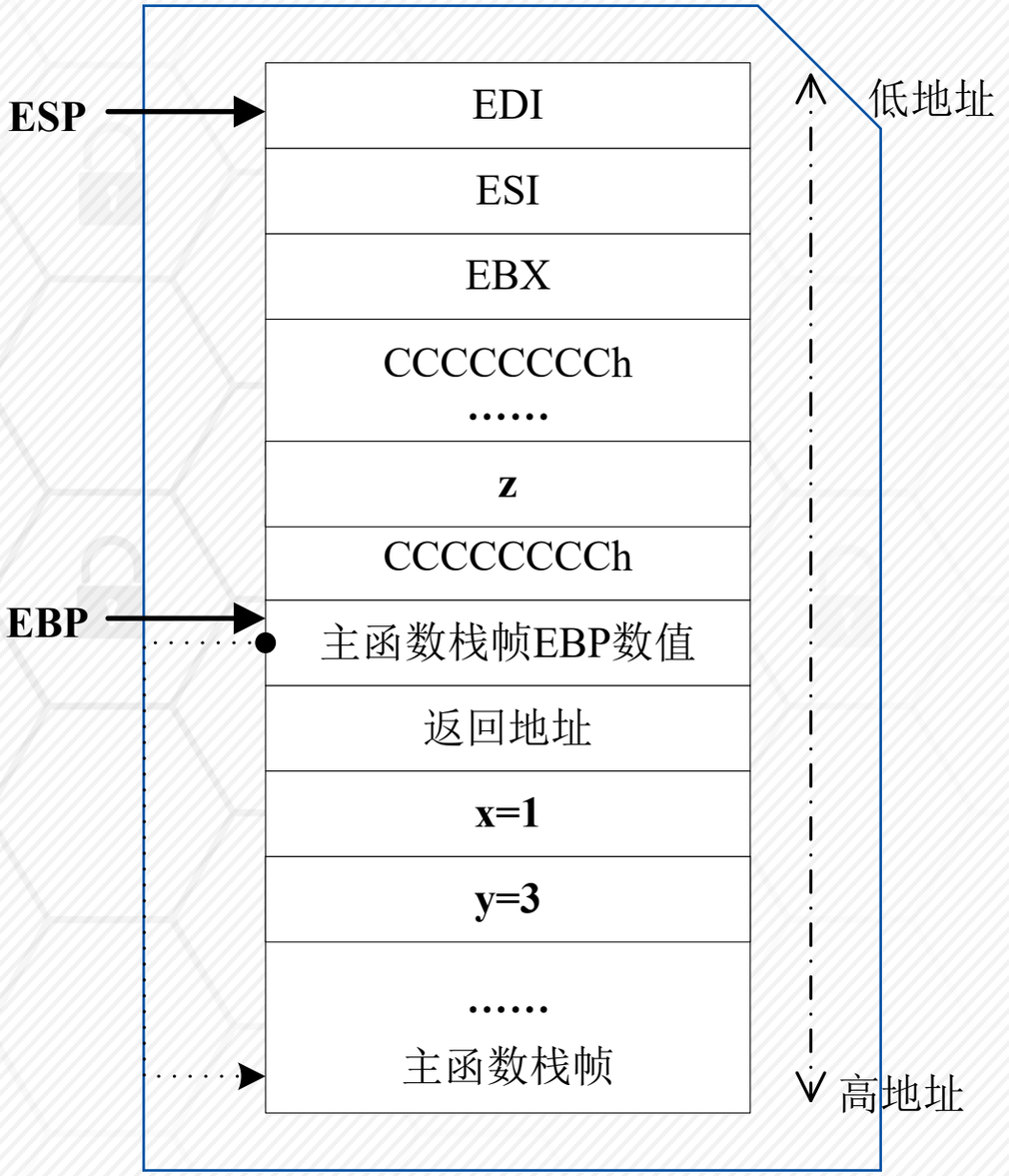
此时栈帧如下：

[x]=[ebp+8]

[y]=[ebp+0ch]

[z]=[ebp-8]？？？

注意，这个是基于VS2005编译的



恢复状态

函数调用完毕，而函数的返回值将存储在EAX寄存器中。

之后，函数调用完毕后，将恢复栈状态到main函数：

```
004113D1  pop       edi ；恢复寄存器值
004113D2  pop       esi ；恢复寄存器值
004113D3  pop       ebx ；恢复寄存器值0
004113D4  mov       esp,ebp；恢复寄存器值
004113D6  pop       ebp ；恢复寄存器值
004113D7  ret             ；根据返回地址恢复EIP值，相当于pop EIP。
```