

《软件安全》实验报告

姓名：陆皓喆 学号：2211044 班级：信息安全

实验名称：

IDE反汇编实验

实验要求：

根据第二章示例2-1，在XP环境下进行VC6反汇编调试，熟悉函数调用、栈帧切换、CALL 和 RET 指令等汇编语言实现，将call 语句执行过程中的EIP 变化、ESP、EBP 变化等状态进行记录，解释变化的主要原因。

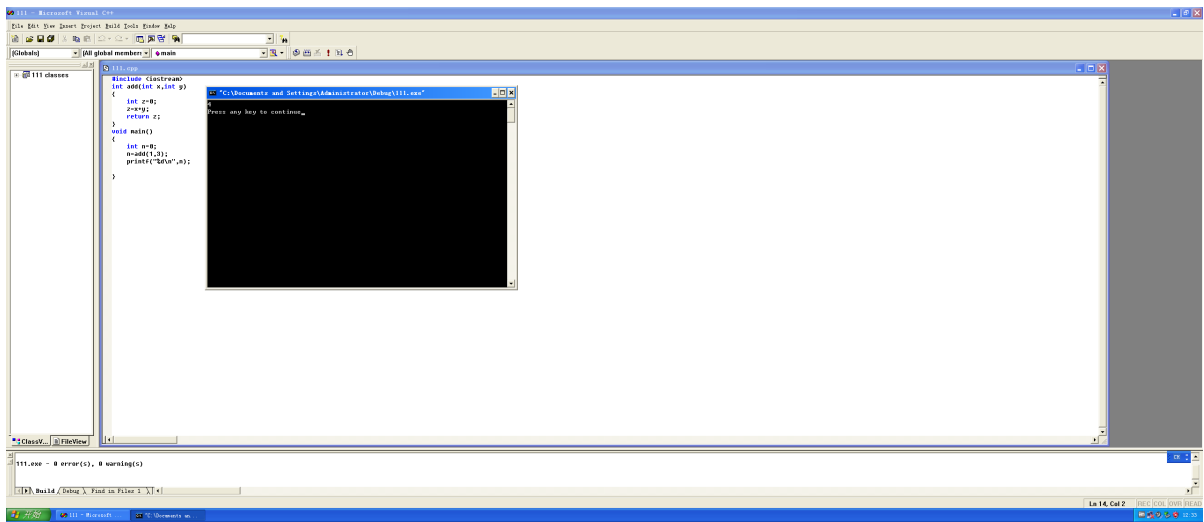
实验过程：

1.进入VC反汇编

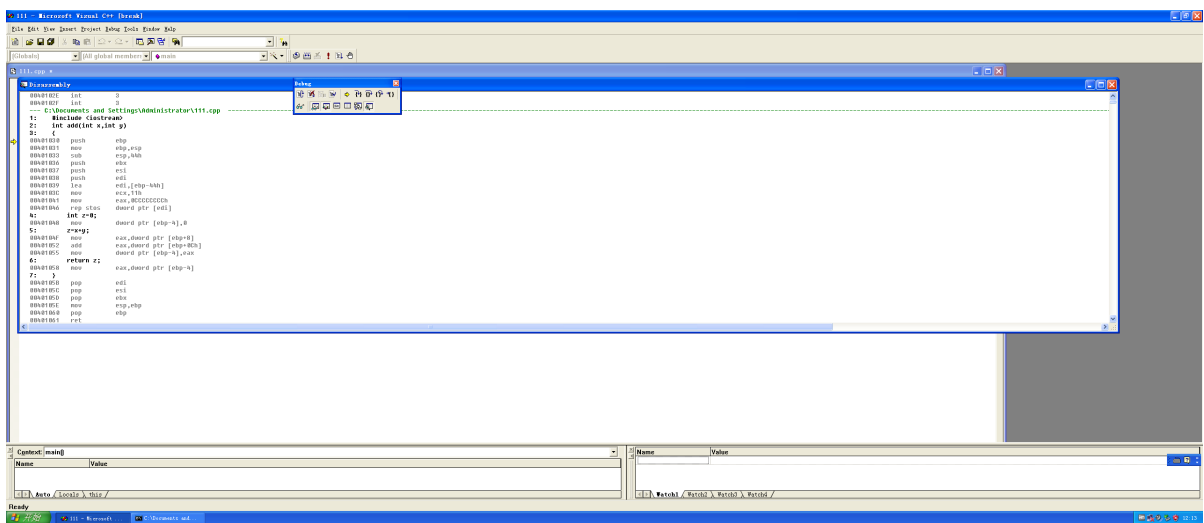
我们打开vmware软件，选择虚拟机“Windows XP Professional”,然后打开XP系统中的Microsoft Visual C++,界面如下所示。



我们新建一个项目，输入主要代码，先进行运行，观察所得结果。



我们在代码 `n=add(1,3)` 按下F9设置断点，再按下F5进行调试，然后按下右键选择“Go To Disassembly”进行反汇编，即可获得汇编代码。



我们需要对该逆向汇编代码进行分析。

2.观察add函数调用前后语句

2.1 调用前

调用前，首先是一条 `MOV` 指令。

```
mov dword ptr [ebp-4],0
```

该条语言是为了局部变量分配了一定的内存空间，`ebp-4` 的意思是，将 `ebp` 寄存器抬高了4字节（低地址为栈的上部）

然后是两条 `PUSH` 指令。

```
push 3
push 1
```

这两条汇编语言的目的是，将两个参数1和3从右到左分别入栈

紧接着，调用了 `CALL` 指令，调用 `ADD` 函数。

```
call @ILT+0(add) (00401005)
```

这句话就是调用了 add 函数，可以发现，是跳转到了 00401005 地址的语句，我们继续跟踪语句，发现跳转到了该语句。

```
@ILT+0(?add@YAHHH@Z):  
00401005 jmp add (00401030)
```

可以发现，语句继续跳转，跳转到 00401030 地址的语句，即为 add 函数的入口处，add 函数调用完成。

```
1:    #include <iostream>  
2:    int add(int x,int y)  
3:    {  
00401030 push ebp
```

2.2 调用后

```
add esp,8
```

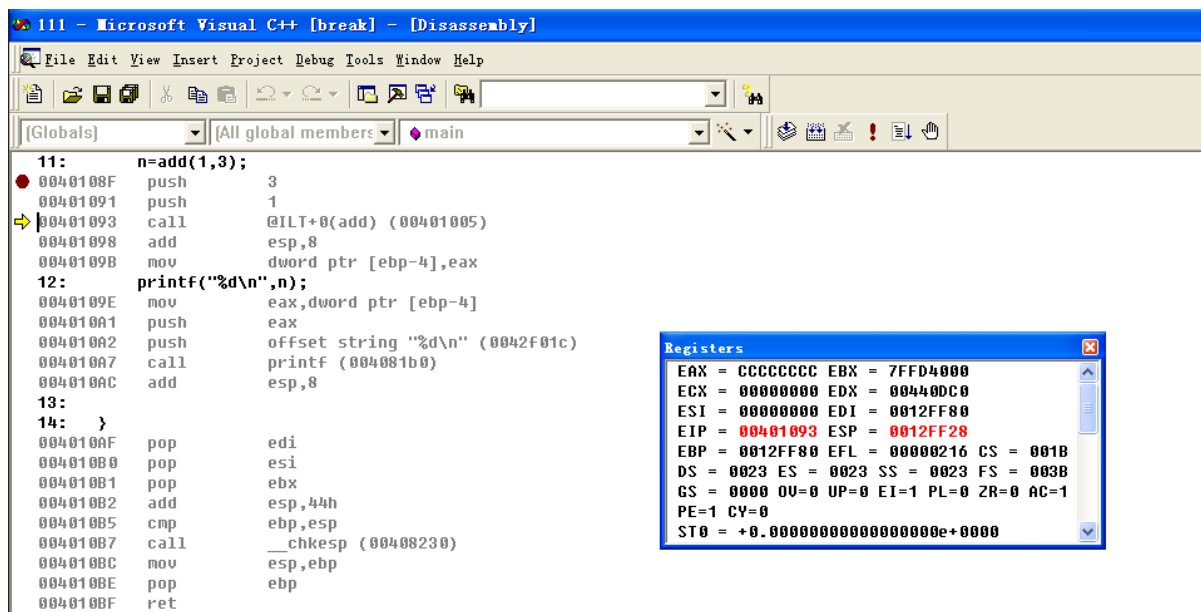
上面的 add 语句，相当于将 esp 寄存器的位置下移8个位置，恢复了调用前所占用的8个字节的空间。

```
mov dword ptr [ebp-4],eax
```

我们可以发现，eax 寄存器中存储的实际上是我们add函数返回的值，即计算结果4。我们将这个计算结果赋值给局部变量 n

3.add函数内部栈帧切换等关键汇编代码

(1) 首先从断点处开始按 F10 步过调试，观察到 esp 从 0012FF30 变成了 001FF28。这表明两个参数入栈成功。因为入栈是由高地址往低地址方向增长，所以 esp 的值减小8。



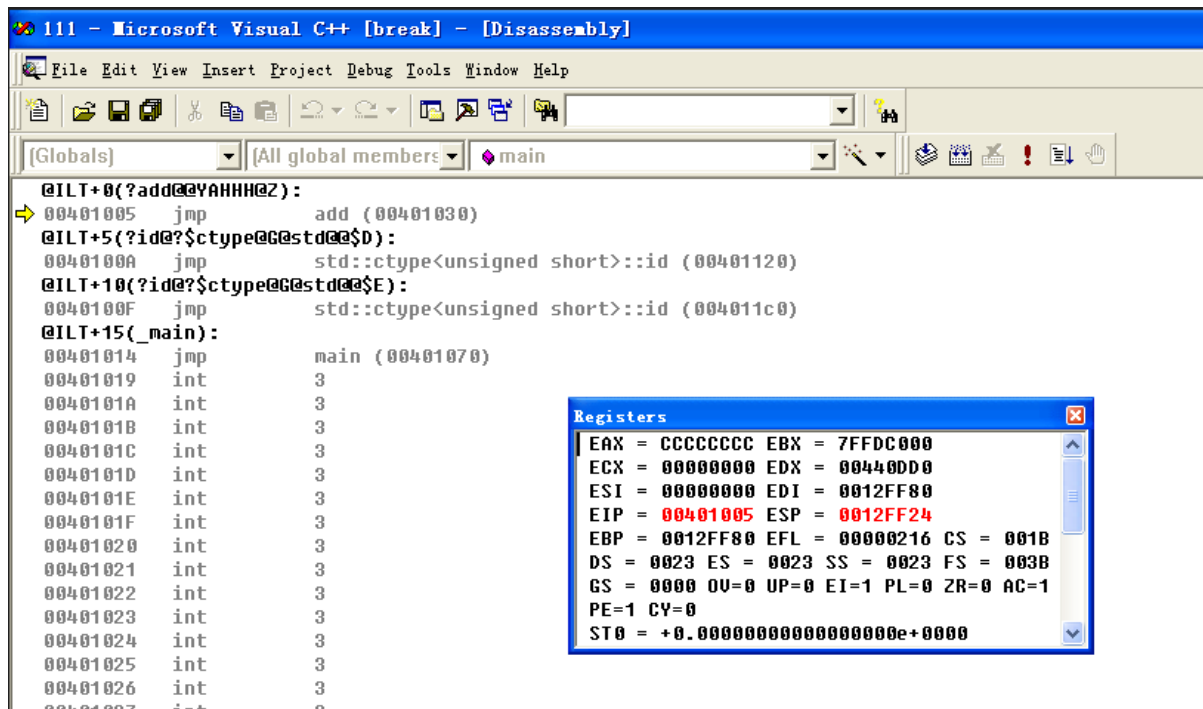
The screenshot shows the Disassembly window of Microsoft Visual C++ with the following assembly code:

```
11:    n=add(1,3);  
0040108F push    3  
00401091 push    1  
00401093 call    @ILT+0(add) (00401005)  
00401098 add     esp,8  
0040109B mov     dword ptr [ebp-4],eax  
12:    printf("%d\n",n);  
0040109E mov     eax,dword ptr [ebp-4]  
004010A1 push    eax  
004010A2 push    offset string "%d\n" (0042F01c)  
004010A7 call    printf (004081b0)  
004010AC add     esp,8  
13:    }  
14:    }  
004010AF pop     edi  
004010B0 pop     esi  
004010B1 pop     ebx  
004010B2 add     esp,44h  
004010B5 cmp     ebp,esp  
004010B7 call    __chkesp (00408230)  
004010BC mov     esp,ebp  
004010BE pop     ebp  
004010BF ret
```

The Registers window on the right shows the following values:

Register	Value
EAX	CCCCCCCC
EBX	7FFD4000
ECX	00000000
EDX	00440DC0
ESI	00000000
EDI	0012FF80
EIP	00401093
ESP	0012FF28
EBP	0012FF80
EFL	00000216
CS	001B
DS	0023
ES	0023
SS	0023
FS	003B
GS	0000
OV	0
UP	0
EI	1
PL	0
ZR	0
AC	1
PE	1
CY	0
ST0	+0.0000000000000000e+0000

(2) 运行到 `call` 指令时，按 `F11` 进行步入，进入到 `add` 函数内部。这时我们观察到，`esp` 的值又减小了4变为 `0012FF24`，这是因为 `call` 指令分两步，第一步是将调用点的下一条指令地址入栈，作为返回地址；第二步是修改 `EIP` 的值，截图中可以看出下一条要执行的指令地址为 `00401005`，这是个 `jump` 指令，跳转到 `add` 函数。这在上面的调用过程中也有所提及。



(3) 按一下 `F11`，我们根据代码行 `00401030` 的内容，找到 `add` 函数的地址，下面是主要代码部分。

```
00401030 push ebp
00401031 mov  ebp,esp
00401033 sub  esp,44h
00401036 push ebx
00401037 push esi
00401038 push edi
00401039 lea  edi,[ebp-44h]
0040103C mov  ecx,11h
00401041 mov  eax,0CCCCCCCch
00401046 rep stos dword ptr [edi]
```

(4) 再按一下 `F11`，我们进入 `add` 函数的第一行，发现 `esp` 寄存器的值又减少了4，说明我们成功将 `ebp` 寄存器入栈了，这是为了方便后期恢复栈帧。

111 - Microsoft Visual C++ [break] - [Disassembly]

File Edit View Insert Project Debug Tools Window Help

[Globals] [All global members] main

```

0040102C int 3
0040102D int 3
0040102E int 3
0040102F int 3
--- C:\Documents and Settings\Administrator\111.cpp -----
1: #include <iostream>
2: int add(int x,int y)
3: {
00401030 push ebp
00401031 mov ebp,esp
00401033 sub esp,44h
00401036 push ebx
00401037 push esi
00401038 push edi
00401039 lea edi,[ebp-44h]
0040103C mov ecx,11h
00401041 mov eax,0CCCCCCCCh
00401046 rep stos dword ptr [edi]
4: int z=0;
00401048 mov dword ptr [ebp-4],0
5: z=x+y;
0040104F mov eax,dword ptr [ebp+8]
00401052 add eax,dword ptr [ebp+0Ch]
00401055 mov dword ptr [ebp-4],eax
6: return z;
00401058 mov eax,dword ptr [ebp-4]
7: }
0040105B pop edi
0040105C pop esi
0040105D pop ebx
0040105E mov esp,ebp
00401060 pop ebp
00401061 ret

```

Registers

EAX	=	CCCCCCCC	EBX	=	7FFDC000
ECX	=	00000000	EDX	=	004400D0
ESI	=	00000000	EDI	=	0012FF80
EIP	=	00401031	ESP	=	0012FF20
EBP	=	0012FF80	EFL	=	00000216
CS	=	001B	DS	=	0023
ES	=	0023	SS	=	0023
FS	=	003B	GS	=	0000
OV	=	0	UP	=	0
EI	=	1	PL	=	0
ZR	=	0	AC	=	1
PE	=	1	CY	=	0
ST0	=	+0.0000000000000000e+0000			

(5) 下面两条语句，通过把 `ebp` 赋值为 `esp`，实现了栈帧的切换，然后 `esp` 减去 `44h`，开辟了局部变量的内存空间。执行后 `esp` 变为 `0012FEDC`。

```

mov ebp,esp
sub esp,44h

```

```

--- C:\Documents and Settings\Administrator\111.cpp -----
1: #include <iostream>
2: int add(int x,int y)
3: {
00401030 push ebp
00401031 mov ebp,esp
00401033 sub esp,44h
00401036 push ebx
00401037 push esi
00401038 push edi
00401039 lea edi,[ebp-44h]
0040103C mov ecx,11h
00401041 mov eax,0CCCCCCCCh
00401046 rep stos dword ptr [edi]
4: int z=0;
00401048 mov dword ptr [ebp-4],0
5: z=x+y;
0040104F mov eax,dword ptr [ebp+8]
00401052 add eax,dword ptr [ebp+0Ch]
00401055 mov dword ptr [ebp-4],eax

```

Registers

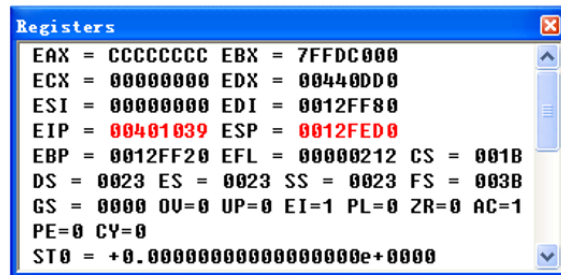
EAX	=	CCCCCCCC	EBX	=	7FFDC000
ECX	=	00000000	EDX	=	004400D0
ESI	=	00000000	EDI	=	0012FF80
EIP	=	00401036	ESP	=	0012FEDC
EBP	=	0012FF20	EFL	=	00000212
CS	=	001B	DS	=	0023
ES	=	0023	SS	=	0023
FS	=	003B	GS	=	0000
OV	=	0	UP	=	0
EI	=	1	PL	=	0
ZR	=	0	AC	=	1
PE	=	0	CY	=	0
ST0	=	+0.0000000000000000e+0000			

(6) 接下来三条指令，是把三个保存寄存器的值入栈进行保存。执行后 `esp` 的值变为 `0012FED0`。

```

--- C:\Documents and Settings\Administrator\111.cpp -----
1:  #include <iostream>
2:  int add(int x,int y)
3:  {
00401030  push    ebp
00401031  mov     ebp,esp
00401033  sub     esp,44h
00401036  push    ebx
00401037  push    esi
00401038  push    edi
00401039  lea     edi,[ebp-44h]
0040103C  mov     ecx,11h
00401041  mov     eax,0CCCCCCCch
00401046  rep stos dword ptr [edi]
4:      int z=0;
00401048  mov     dword ptr [ebp-4],0

```



(7) 接下来的四条指令是把局部变量最顶部的地址赋值给 `edi`，然后利用 `ecx` 作为计数器，循环 11h 次填充操作，将刚才为局部变量开辟出的 44h 的空间全部赋值为 0CCCCCCCch。

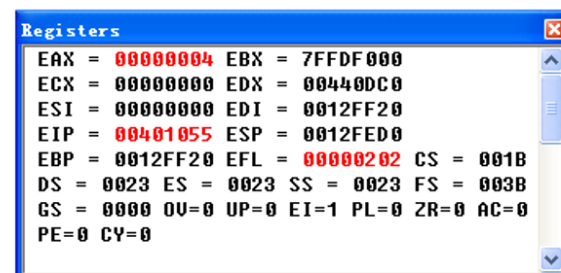
0012FF3A	FD	7F	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC
0012FF45	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC
0012FF50	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC
0012FF5B	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC
0012FF66	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC
0012FF71	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC

(8) 接下来就是进行加法运算，`ebp+8` 访问的是参数 `x`，`ebp+0Ch` 访问的是参数 `y`。最后结果保存在 `eax` 中。保存返回值是再把 `[ebp-4]` 中存的加和值赋给 `eax`。在下图中可以发现，`EAX` 寄存器的值被赋值成了 4，就是 1+3 的结果。

```

3:  {
00401030  push    ebp
00401031  mov     ebp,esp
00401033  sub     esp,44h
00401036  push    ebx
00401037  push    esi
00401038  push    edi
00401039  lea     edi,[ebp-44h]
0040103C  mov     ecx,11h
00401041  mov     eax,0CCCCCCCch
00401046  rep stos dword ptr [edi]
4:      int z=0;
00401048  mov     dword ptr [ebp-4],0
5:      z=x+y;
0040104F  mov     eax,dword ptr [ebp+8]
00401052  add     eax,dword ptr [ebp+0Ch]
00401055  mov     dword ptr [ebp-4],eax
6:      return z;

```

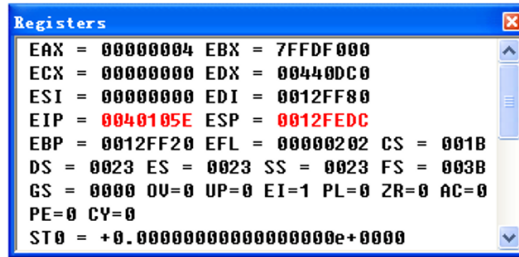


(9) 下一步就是要进行栈帧的恢复，把三个寄存器给弹出，可以看出 `ESP` 寄存器的值变成了 0012FEDC。

```

3:  {
00401030 push    ebp
00401031 mov     ebp,esp
00401033 sub     esp,44h
00401036 push    ebx
00401037 push    esi
00401038 push    edi
00401039 lea     edi,[ebp-44h]
0040103C mov     ecx,11h
00401041 mov     eax,0CCCCCCCCh
00401046 rep stos dword ptr [edi]
4:  int z=0;
00401048 mov     dword ptr [ebp-4],0
5:  z=x+y;
0040104F mov     eax,dword ptr [ebp+8]
00401052 add     eax,dword ptr [ebp+0Ch]
00401055 mov     dword ptr [ebp-4],eax
6:  return z;
00401058 mov     eax,dword ptr [ebp-4]
7:  }
0040105B pop     edi
0040105C pop     esi
0040105D pop     ebx
→ 0040105E mov     esp,ebp
00401060 pop     ebp

```

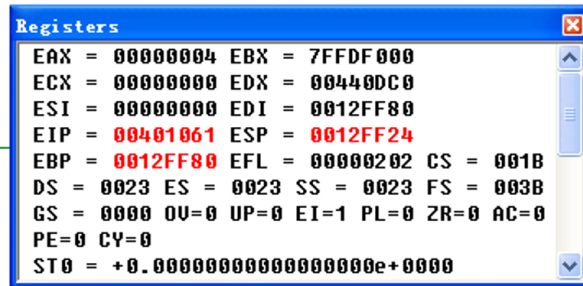


(10) 接下来的两步操作，先把 `ebp` 赋值给 `esp`，再把旧栈帧的 `ebp` 值弹出，这样就恢复到调用前的栈帧位置。这个时候栈顶指针 `esp` 所指的内容就是返回地址。

```

6:  return z;
00401058 mov     eax,dword ptr [ebp-4]
7:  }
0040105B pop     edi
0040105C pop     esi
0040105D pop     ebx
0040105E mov     esp,ebp
00401060 pop     ebp
→ 00401061 ret
--- No source file ---
00401062 int      3
00401063 int      3
00401064 int      3
00401065 int      3
00401066 int      3
00401067 int      3
00401068 int      3
00401069 int      3
0040106A int      3
0040106B int      3
0040106C int      3
0040106D int      3
0040106E int      3
0040106F int      3

```



0012FF24 98 10 40 00 01 00 00 00 03 00 00

这行代码的意思就是 `esp` 寄存器中存储的内容。可以知道，`eip` 寄存器在执行 `ret` 指令后，就会赋值为 `00401098`。

(11) `ret` 指令结束，跳出 `call` 指令，到下一句 `add` 函数，可以看出 `EIP` 寄存器的值变成了 `00401098`。整个 `call` 指令结束。

```
00401086 rep stos dword ptr [edi]
10: int n=0;
00401088 mov dword ptr [ebp-4],0
11: n=add(1,3);
0040108F push 3
00401091 push 1
00401093 call @ILT+0(add) (00401005)
00401098 add esp,8
0040109B mov dword ptr [ebp-4],eax
12: printf("%d\n",n);
0040109E mov eax,dword ptr [ebp-4]
004010A1 push eax
004010A2 push offset string "%d\n" (0042F01C)
004010A7 call printf (004081B0)
004010AC add esp,8
```

Registers
EAX = 00000004 EBX = 7FFDF000
ECX = 00000000 EDX = 00440DC0
ESI = 00000000 EDI = 0012FF80
EIP = 00401098 ESP = 0012FF28
EBP = 0012FF80 EFL = 00000202 CS = 001B
DS = 0023 ES = 0023 SS = 0023 FS = 003B
GS = 0000 OV=0 UP=0 EI=1 PL=0 ZR=0 AC=0
PE=0 CY=0
ST0 = +0.0000000000000000e+0000

心得体会：

通过这次实验，我学到了很多新的知识与理论。

1. 我学会了如何在虚拟机的VC6上进行编程与设置断点，反汇编，懂得了虚拟机的使用。
2. 学会了RET的使用方法。从上面的过程（11）中，我们可以看出，ret指令的用处实际上是pop eip，就是说将eip寄存器进行弹出，我们发现经过ret后，eip的地址变成了00401098，就跟上面的esp存储的内容是一样的。
3. 此外，通过该次实验，我学到了多种汇编语言的方法。
4. 通过此次实验，我发现软件安全的上机实验与上学期所学习的《汇编语言与逆向技术》有所不同，在汇编语言课中，我只学习了栈地址的一些理论知识而没有进行具体的操作；而在软件安全实验中的IDE反汇编实验中，我具体进行了在逆向语言中的实际操作，让我受益匪浅。
5. 通过本次实验，我对函数调用过程中，参数、局部变量的内存空间分配有了更深入的了解，对各个寄存器的使用也有了掌握。