《软件安全》实验报告

姓名：许洋　学号：2313721 班级：1070

实验名称：

IDE反汇编实验

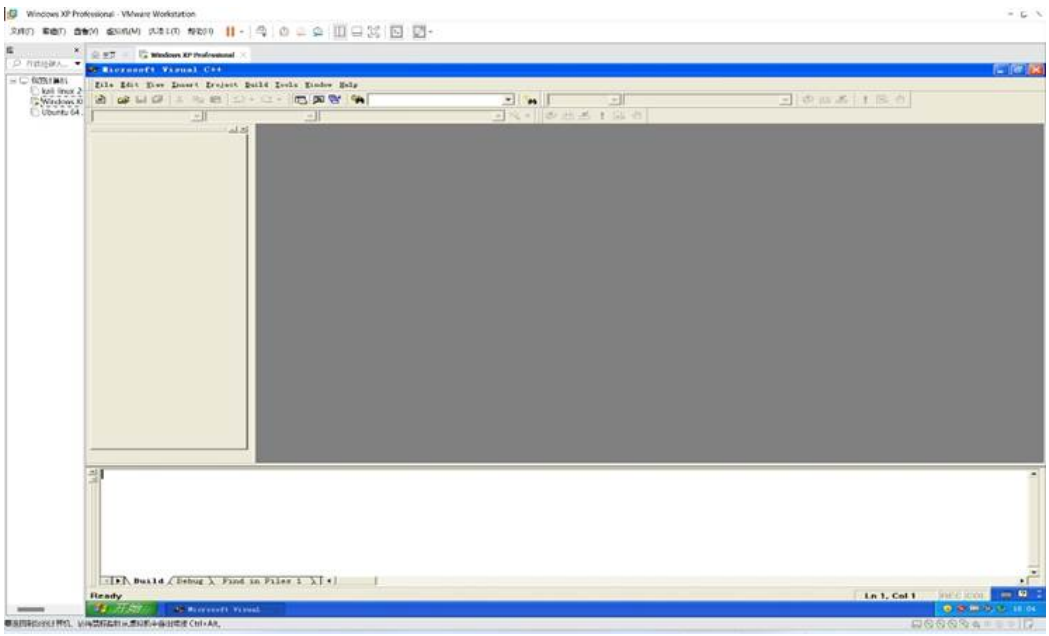
实验要求：

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。

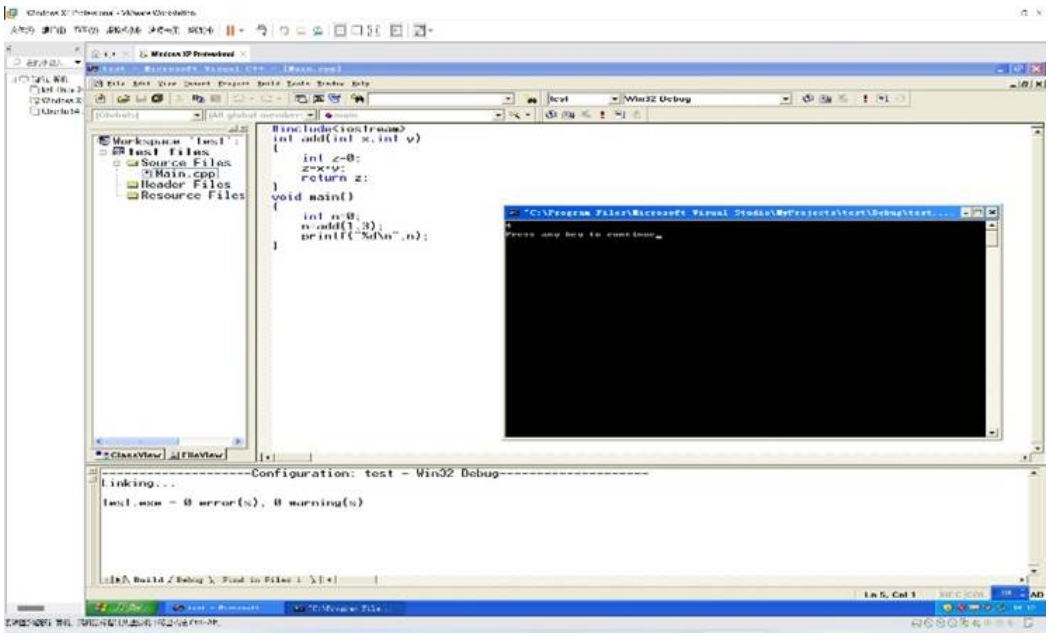
实验过程：

1. 进入VC反汇编

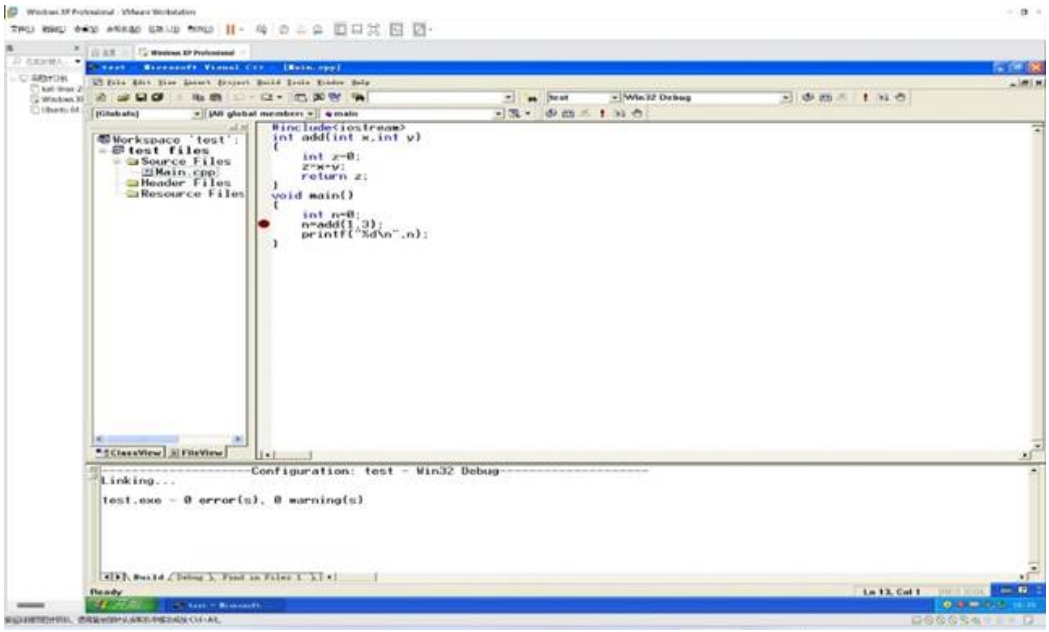
我们打开vmware软件，选择虚拟机“Windows XP Professional”,然后打开XP系统中的Microsoft Visual C++,界面如下。

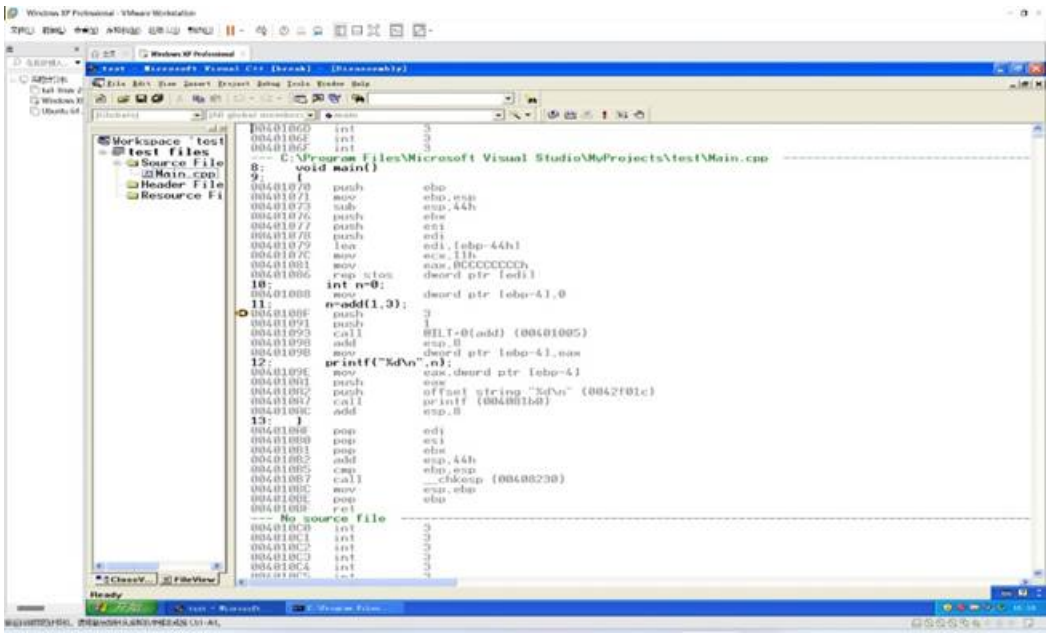


新建一个项目，输入给出的测试代码，构建后运行项目，观察输出。



我们在代码n=add(1,3)按下（Fn+）F9(电脑设置不同，我的电脑需要按下Fn键才可使用，其他可以直接按F9)设置断点，再按下（Fn+）F5进行调试，右键打开菜单，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n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，因为参数入栈是从右向左的。

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
```

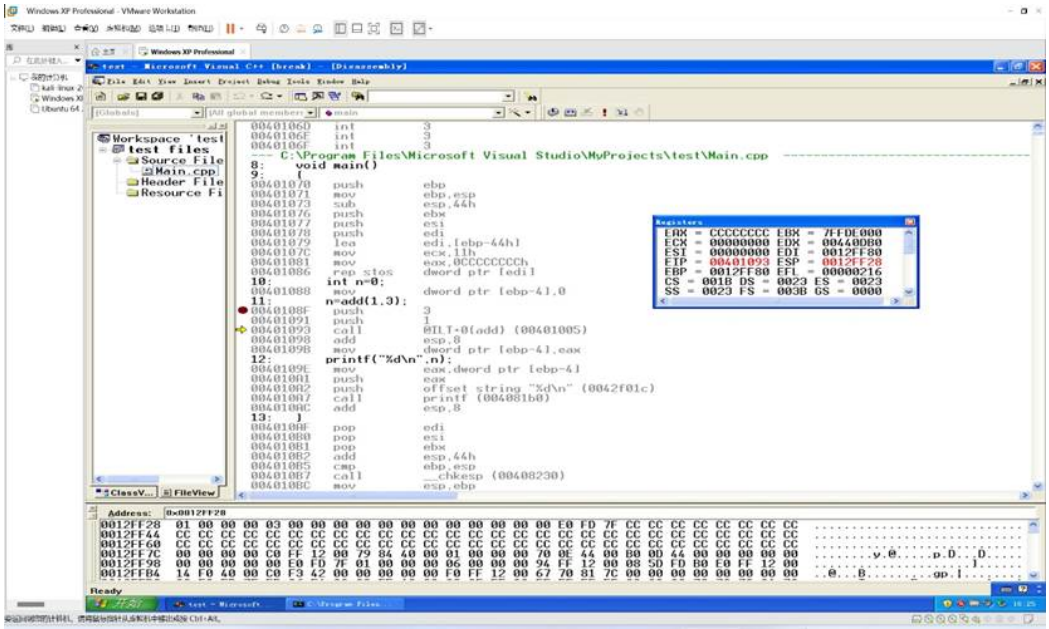
上面的add语句，相当于将esp寄存器的位置下移8个位置，恢复了调用前所占用的8个字节的空间。

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

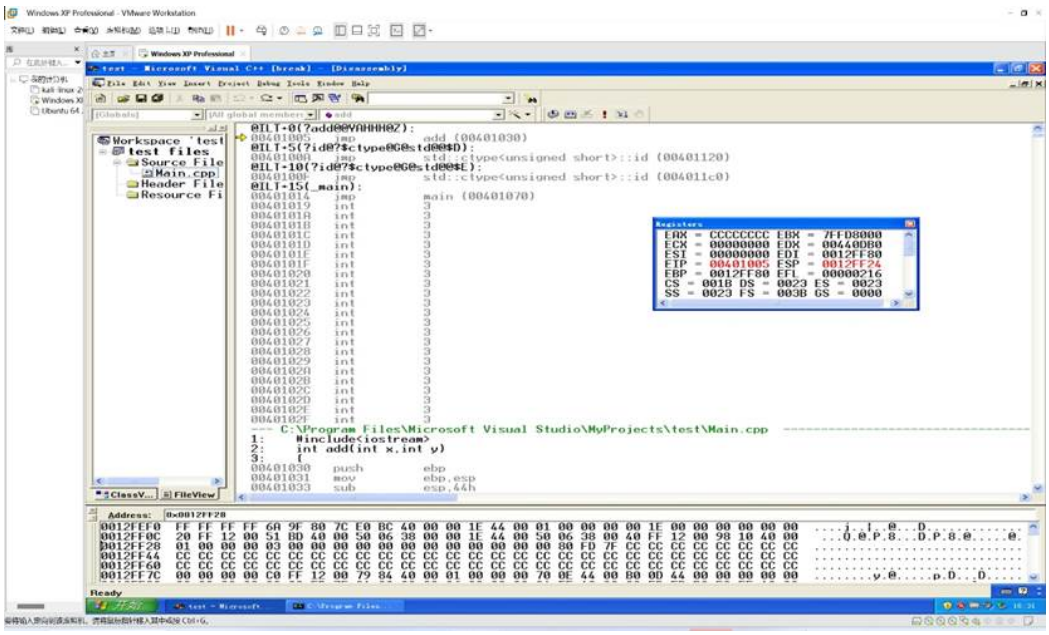
我们可以发现，eax寄存器中存储的实际上是我们add函数返回的值，即计算结果4。我们将这个计算结果赋值给局部变量n。

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（Fn+）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。



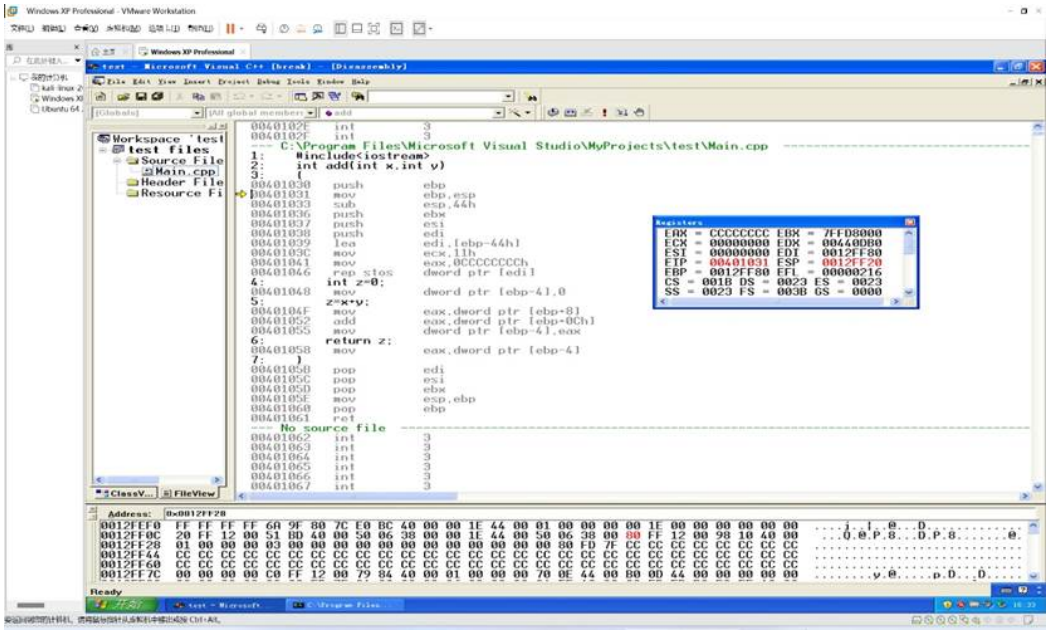
（2）运行到call指令时，按（Fn+）F11进行步入，进入到add函数内部。这时我们观察到，esp的值又减小了4变为0012FF24，这是因为call指令分两步，第一步是将调用点的下一条指令地址入栈，作为返回地址即00401098；第二步是修改EIP的值，截图中可以看出下一条要执行的指令地址为00401005，这是个jump的跳转指令，跳转到add函数。



（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。

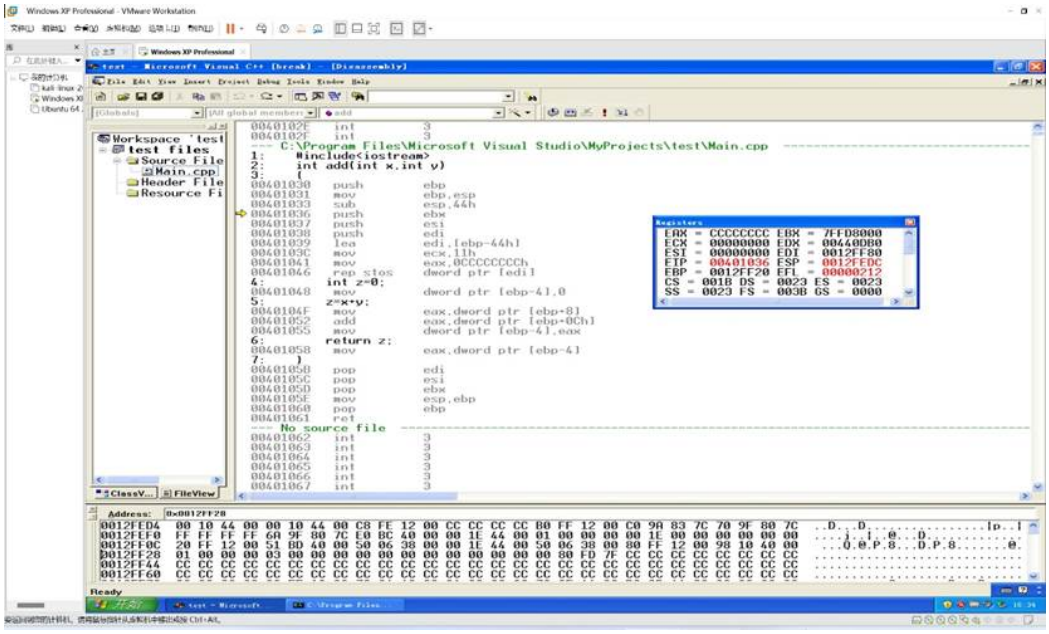
```
00401030 push ebp
00401031 mov ebp,esp
00401033 sub esp,44h
00401036 push ebx
00401037 push esi
00401038 push edi
00401039 lea edi,[ebp-44h]
0040103C mov ecx,11h
00401041 mov eax,0CCCCCCCCh
00401046 rep stos dword ptr [edi]
```

（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。



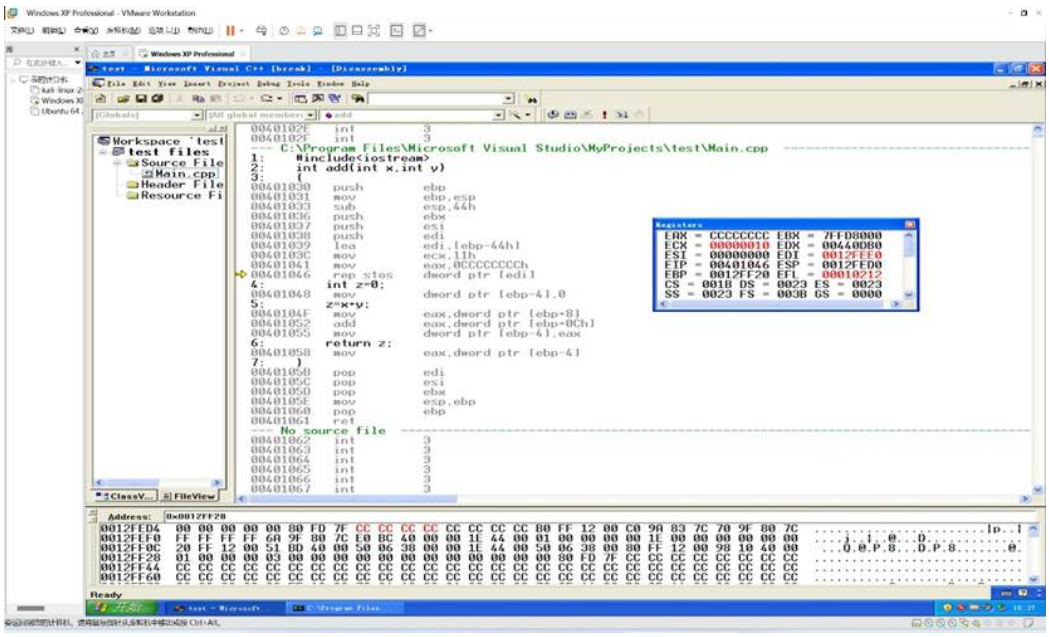
（5）下面两条语句，通过把esp的值赋给ebp，实现了栈帧的切换，然后esp减去44h，开辟了局部变量的内存空间。执行后esp变为0012FEDC。

```
mov ebp,esp
sub esp,44h
```

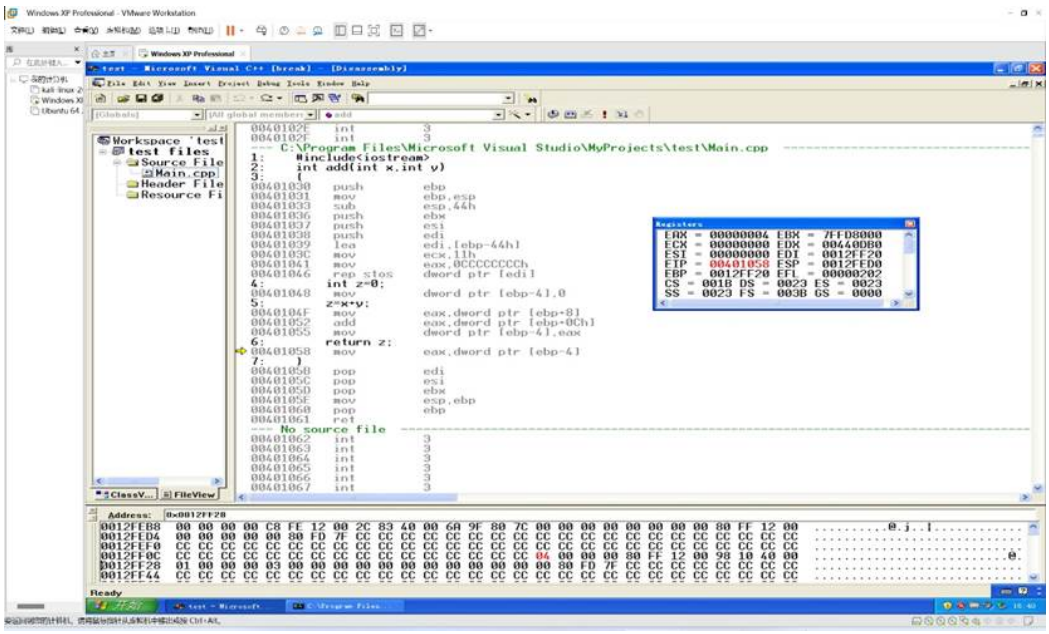


（6）接下来三条指令，是把三个保存寄存器的值入栈进行保存。执行后esp的值变为0012FED0。

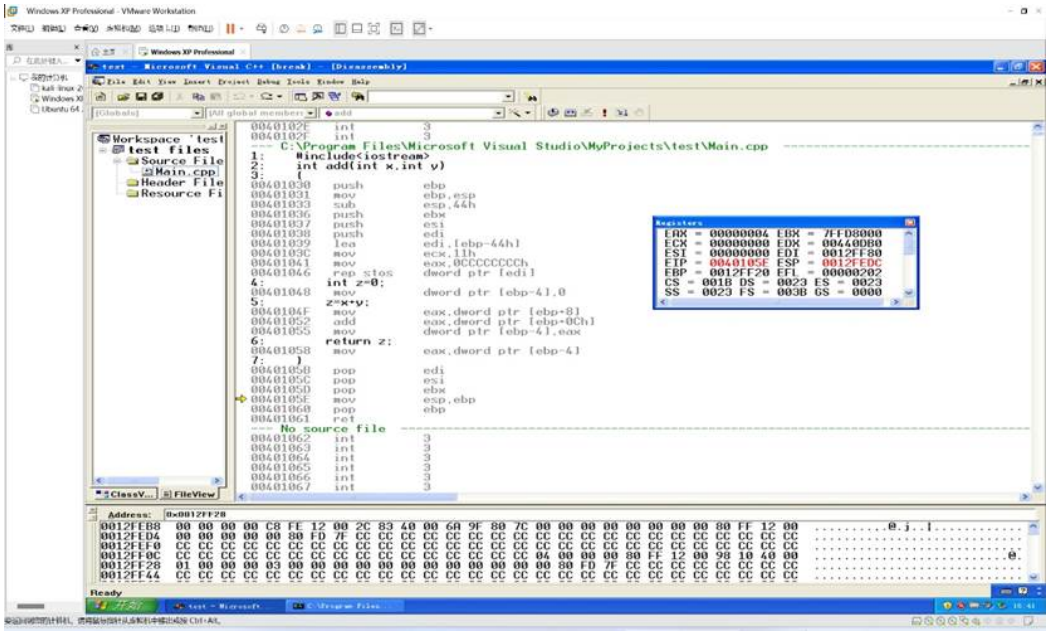
（7）接下来的四条指令是把局部变量最顶部的地址赋值给edi，然后利用ecx作为计数器，循环11h次填充操作，将刚才为局部变量开辟出的44h的空间全部赋值为0CCCCCCCCh。然后进入对z的初始化。



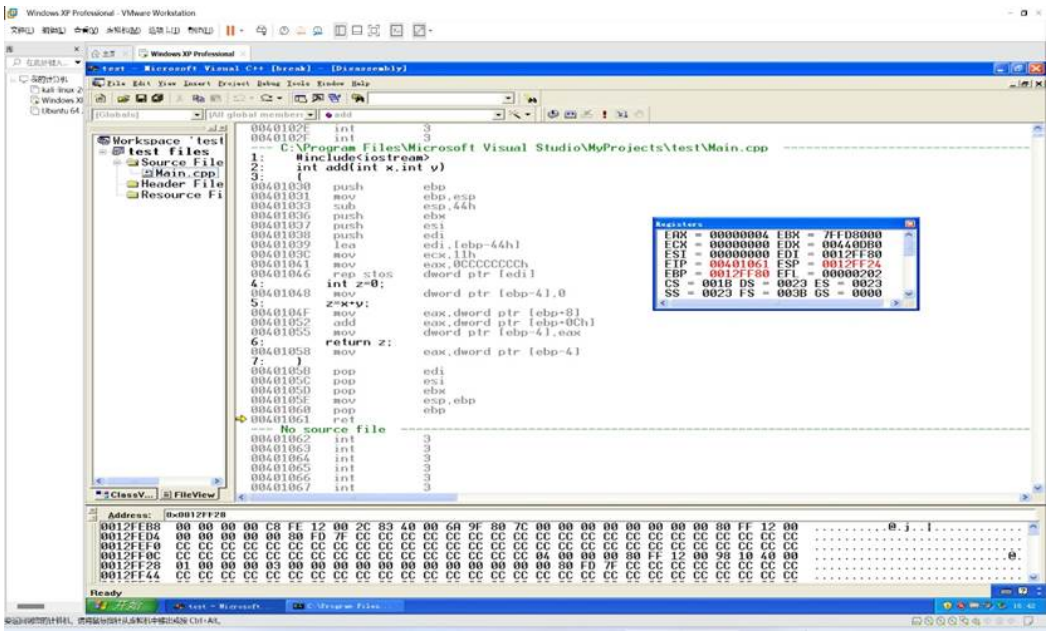
（8）接下来就是进行加法运算，ebp+8访问的是参数x赋值给eax，ebp+0Ch访问的是参数y加到eax上，然后将eax赋值给ebp-4即z的位置。保存返回值是再把[ebp-4]中存的加和值赋给eax。在下图中可以发现，eax寄存器的值被赋值成了4，就是1+3的结果。



（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。



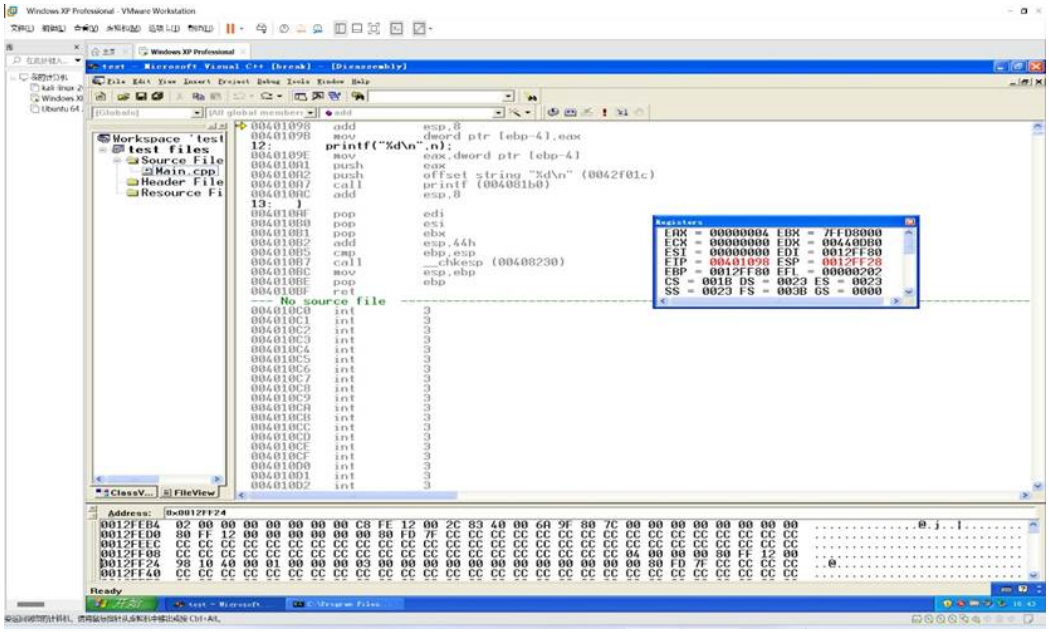
（10）接下来的两步操作，先把ebp赋值给esp，再把旧栈帧的ebp值弹出，这样就恢复到调用前的栈帧位置。这个时候栈顶指针esp所指的内容就是返回地址。



```
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。



心得体会：

通过实验，掌握了RET指令的用法；

RET指令实际就是执行了Pop EIP

此外，通过本实验，掌握了多个汇编语言的用法

通过这次实验，我学到了很多新的知识与理论。

1. 我学会了如何配置虚拟机，以及在虚拟机的VC6上进行创建项目编译运行与设置断点，反汇编。

2. 学会了RET的使用方法。从上面的过程中，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. 通过该次实验，我学到了多种汇编语言的方法。
4. 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。