《漏洞利用及渗透测试基础》实验报告

**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。

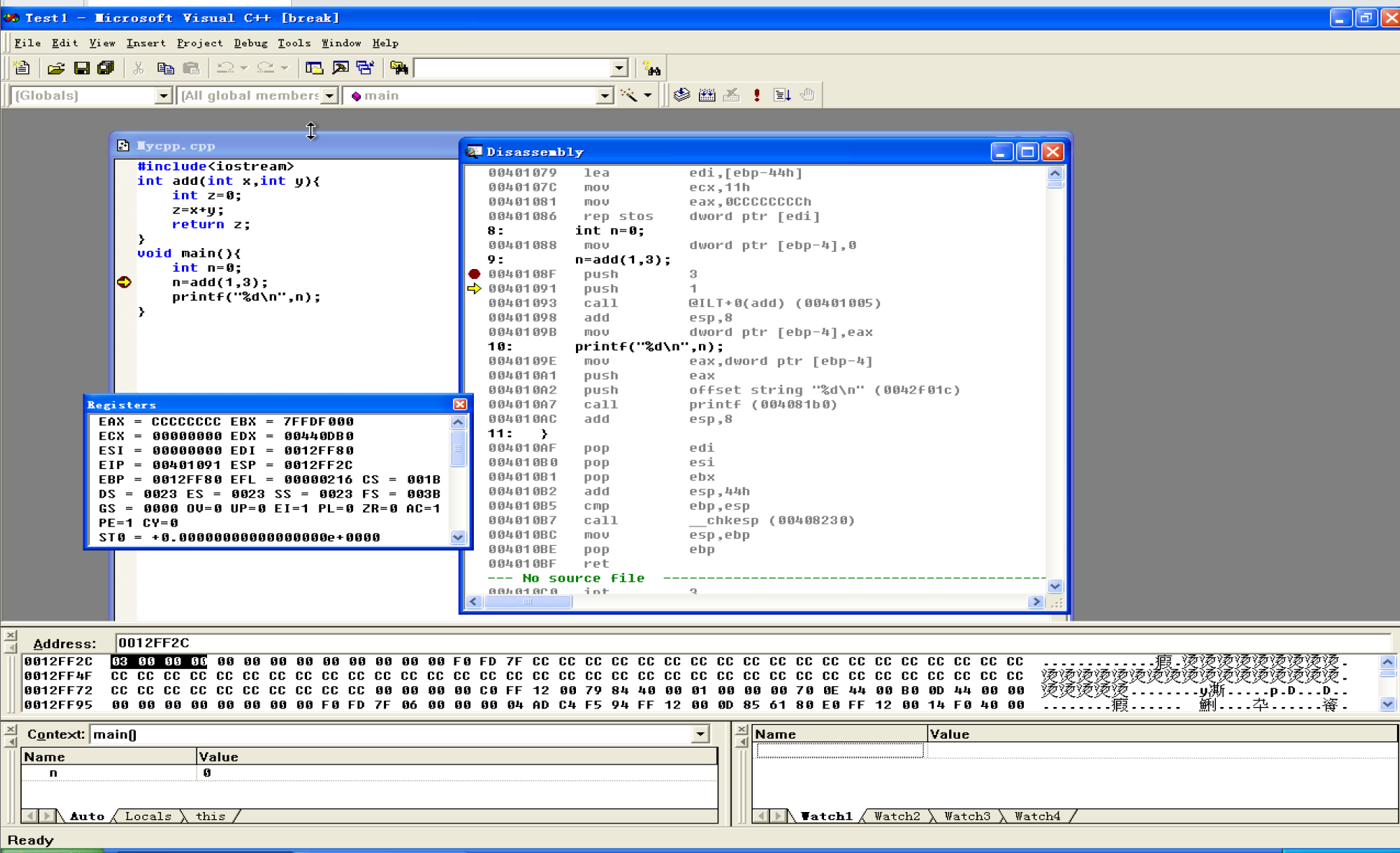
**实验过程：**

1. 进入VC反汇编

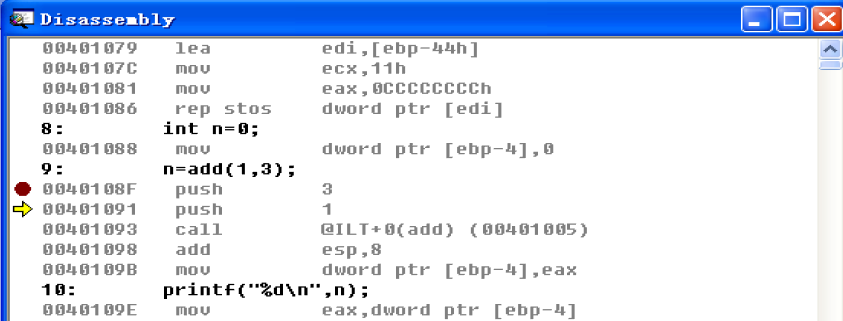
A．利用快捷键F9在add函数前加断点，在按下F5开始执行程序，并跳转到add函数前。

B．在调试模式下，右键进入反汇编模式，可以看到汇编代码以及寄存器的值。（如果没有显示代码以及寄存器窗口可以右键工具栏调整视图的显示）。

C.进入反汇编模式，效果如下：



2. 观察add函数调用前后语句



add函数前后的汇编语句如图所示。

mov dword ptr [ebp-4],0 是将0值压入栈中，也就是赋值给n。

随后的两条push语句是将函数调用的参数入栈，值得注意的是参数入栈的顺序是从后向前入栈的，后面的参数先入栈，这样符合栈后进先出的特点。

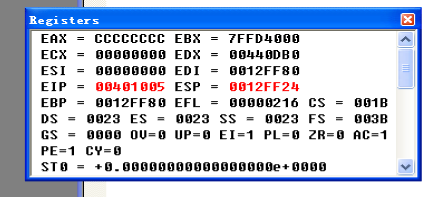
在call语句执行完毕以后的add语句是将栈顶元素弹出，然后的mov语句是将保存在寄存器eax里的函数返回值赋值给n。

3. add函数内部栈帧切换等关键汇编代码

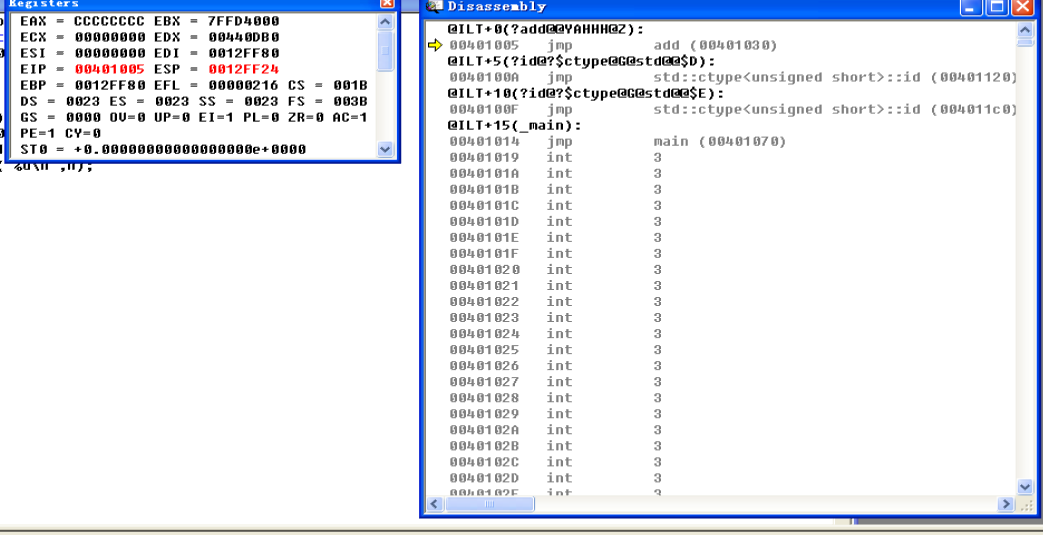
在执行到call语句时我们按下F11进入add函数。

在进入add函数时，首先会将call语句的下一条指令入栈，因此可以看到栈顶元素的值为，由于机器采用小端编码，所以栈顶所存储的数值为0x00401098，就是call语句后add语句的地址，这样处理便会记录下add函数执行完程序要返回的地址。

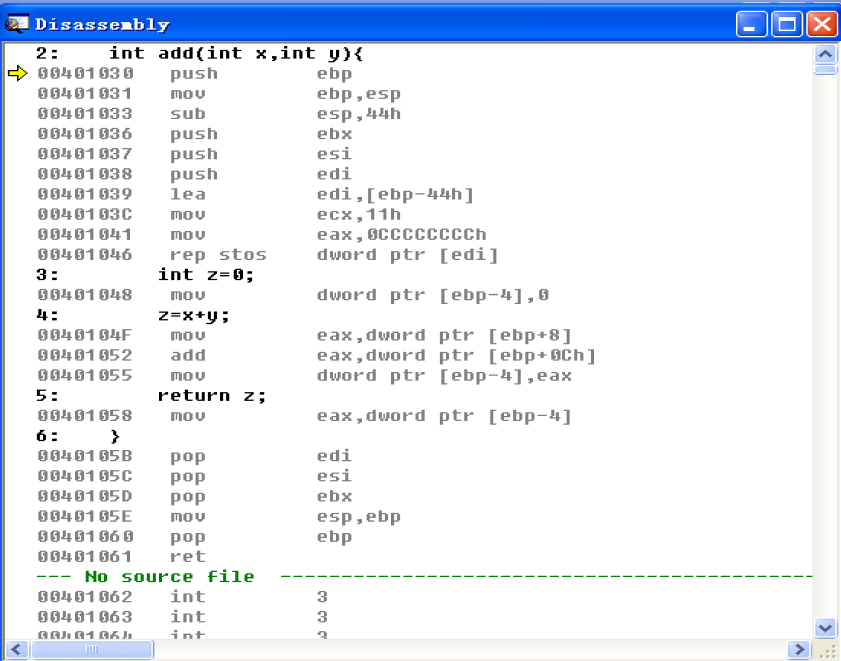
同时会把EIP寄存器里的值改写为call语句中所写的地址：



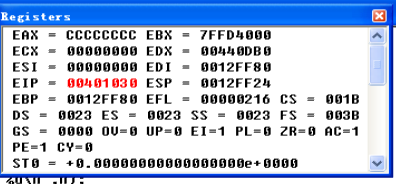
而进入到call语句所写的地址后，程序还会进一步跳转，才会进入add函数：



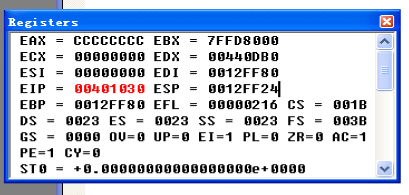
可以看见add函数的汇编语句如图：



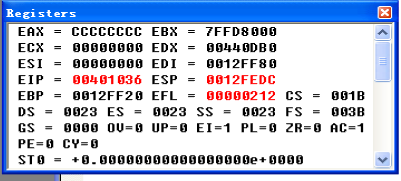
此时寄存器的值：



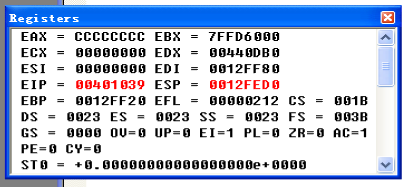
前三行的语句适用于调整栈帧，先记录原来的栈帧的基址，再将原来的栈顶更新为当前函数的调用栈的基址，并在栈中开辟出44字节的空间，在执行之前的寄存器值的情况如下：



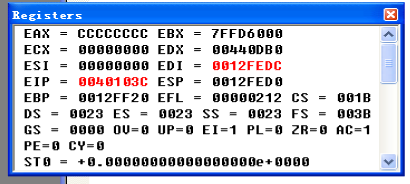
调整栈帧后，寄存器的值：



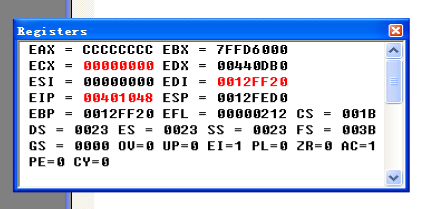
随后的三条push语句将主调函数保留的寄存器的值依次入栈，每次入栈一个寄存器的值，esp的值都会减4：



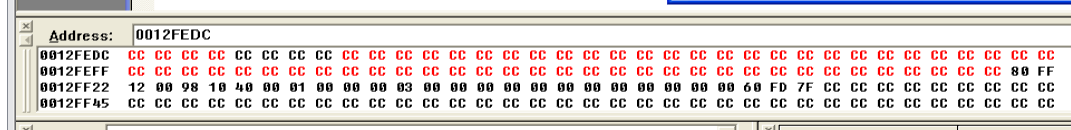
随后的lea语句计算刚才所开辟的44字节的栈顶部的地址，并赋值给edi：



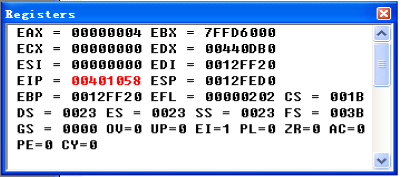
然后将ecx的值赋值为11，表示要进行一个次数为11的循环，然后将eax的值设置为0cccccccc,接下来的rep语句将重复11次，每次都将把eax中的数据放入edi所存的地址指向的内存中，并将edi中的地址值加4，将ecx的值减1，直到ecx的值减为0为止。



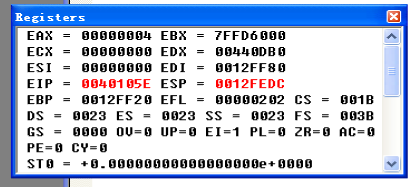
栈中的值：



随后的值是将栈中的前4个字节（变量z）赋值为0，然后将参数1移动到eax中，再将参数2加到eax中，最后再将eax的值赋给z，随后又把z的值赋给eax，作为返回值，

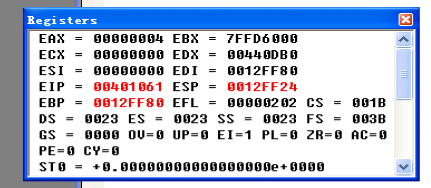


最后的return语句，首先的三条pop语句恢复主调函数保留的寄存器的值：

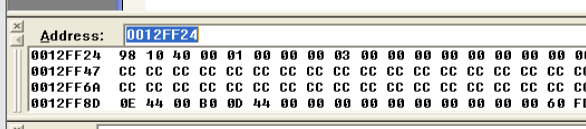


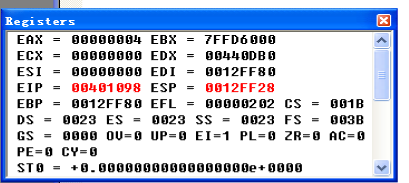
每次esp的值都会加4。

随后的两条语句是为了回复原来的栈帧基址和栈顶指针。



可以看到此时栈顶保存的值正是进入call语句之前，call语句下一条指令的地址，当执行到req语句时，程序将栈顶所存的地址赋值给EIP寄存器，并将栈顶的值弹出，esp的值加4，完成函数的返回。





**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