**第 5周作业 函数调用**

1. CPU 在执行CALL指令、RET指令时，会分别完成什么操作？

**答：CPU**在执行CALL指令会做两件事。

**① 将断点（返回）地址压栈，即将当前CALL指令的下一条指令地址压入堆栈。【在取出CALL指令的机器码，对指令进行译码后，EIP加上该指令的长度，从而指向CALL指令的下一条指令】【压栈时，先将栈顶指针ESP减4，，然后将被压的双字数据送到 ESP指向的单元】。**

**② 将子程序的入口地址送给 EIP。【根据 CALL 指令是直接跳转还是间接跳转，得到 EIP的方式有所不同。】**

1. 设执行如下语句前， ESP = 0x12345678，指令地址为 0x004A1090  
   004A1090 E8 32 00 00 00 call func ; 子程序直接调用  
   在执行CALL 后， EIP = ？ ESP =？ 栈顶的双字数据是多少？

**答：执行CALL 后，EIP=0x004A10C7**

**在取CALL指令、译码后，EIP = 0x004A1090 +5 = 0x004A1095  
 该CALL是直接调用，子程序的入口地址为：**

**EIP+0x00000032 = 0x4A1095+0x32=0x004A10C7 -> EIP**

**ESP=0x12345674**

**栈顶双字为：0x004A1095**

1. 设 EAX = 0x004A1090 EBX = 0x004A109A ESP=0x12345678

执行如下程序段 PUSH EAX

PUSH EBX

RET

执行 RET 指令后， EIP = ？ ESP = ？

**答：EIP=0x004A109A**

**ESP=0x12345674**

1. **对下面的程序进行调试 （x86, Debug版），观察反汇编窗口、寄存器窗口、内存窗口等，回答问题**

注意：为了看到更紧凑的变量空间分配方式，可将项目属性 C/C++ -> 代码生成-> “基本运行时检测”设置为“默认值”； “安全检查”设置为“禁用安全检查”。每个同学都根据自己的观察回答问题。

int fadd(int a, int b)

{ int temp;

a+=10;

b+=20;

temp = a + b;

return temp;

}

void main( )

{ int x = 0x1234;

int y = -32;

int result = 0;

result = fadd(x, y);

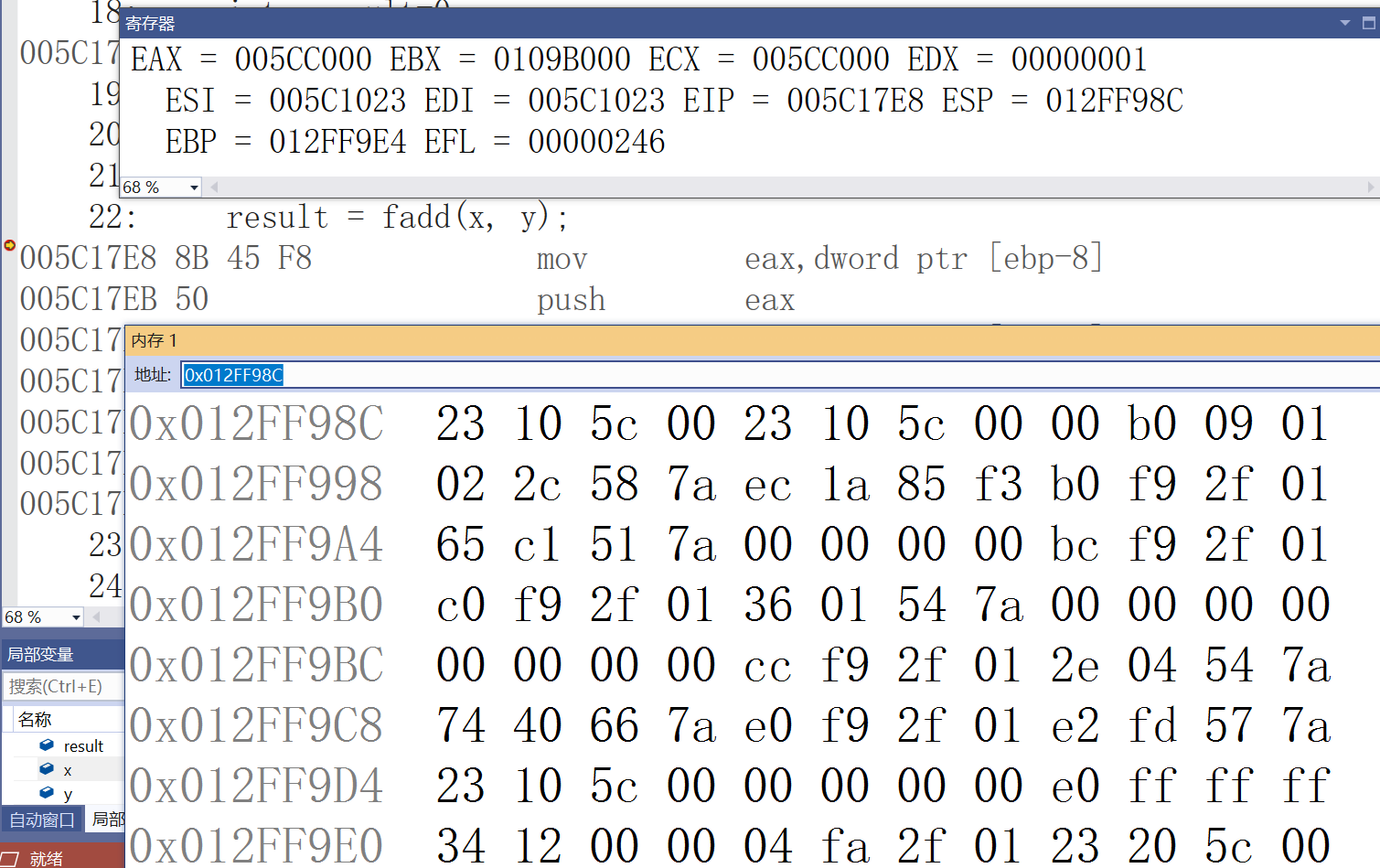
x=30;

}

1. 画出 执行result=fadd(x,y) 之前的堆栈示意图。

要求在堆栈中标明 x、y、result位置，以及具体的地址值；指出 ESP的值、EBP的值；在堆栈中标明 ESP指向的位置。中间不相关的内容可以用省略号。

答：在本人调试中，在执行result=fadd(x,y) 之前，观察寄存器、内存，可看到如下信息：ESP = 0x012FF98C； EBP = 0x012FF9E4 ; x的地址为 0x012FF9E0； y的地址为 0x012FF9DC；result的地址为 0x012FF9D8。 x,y,z单元中的内容分别在 红、绿、蓝线条之上。

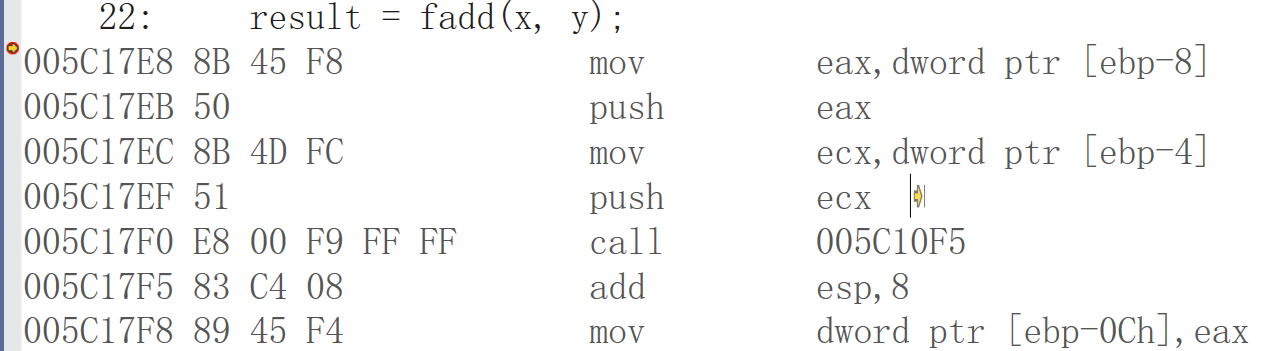


堆栈示意图

|  |  |  |
| --- | --- | --- |
| 符号地址 | 单元中的内容 | 地址(由小到大 |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
| ESP-> |  | 0x012FF98C |
|  | ……… |  |
| result | 0 | 0x012FF9D8 |
| y | -32 (0xffffffe0) | 0x012FF9DC |
| x | 0x1234 | 0x012FF9E0 |
| EBP -> |  | 0x012FF9E4 |

1. 给出result=fadd(x,y); 的反汇编截图；

答：

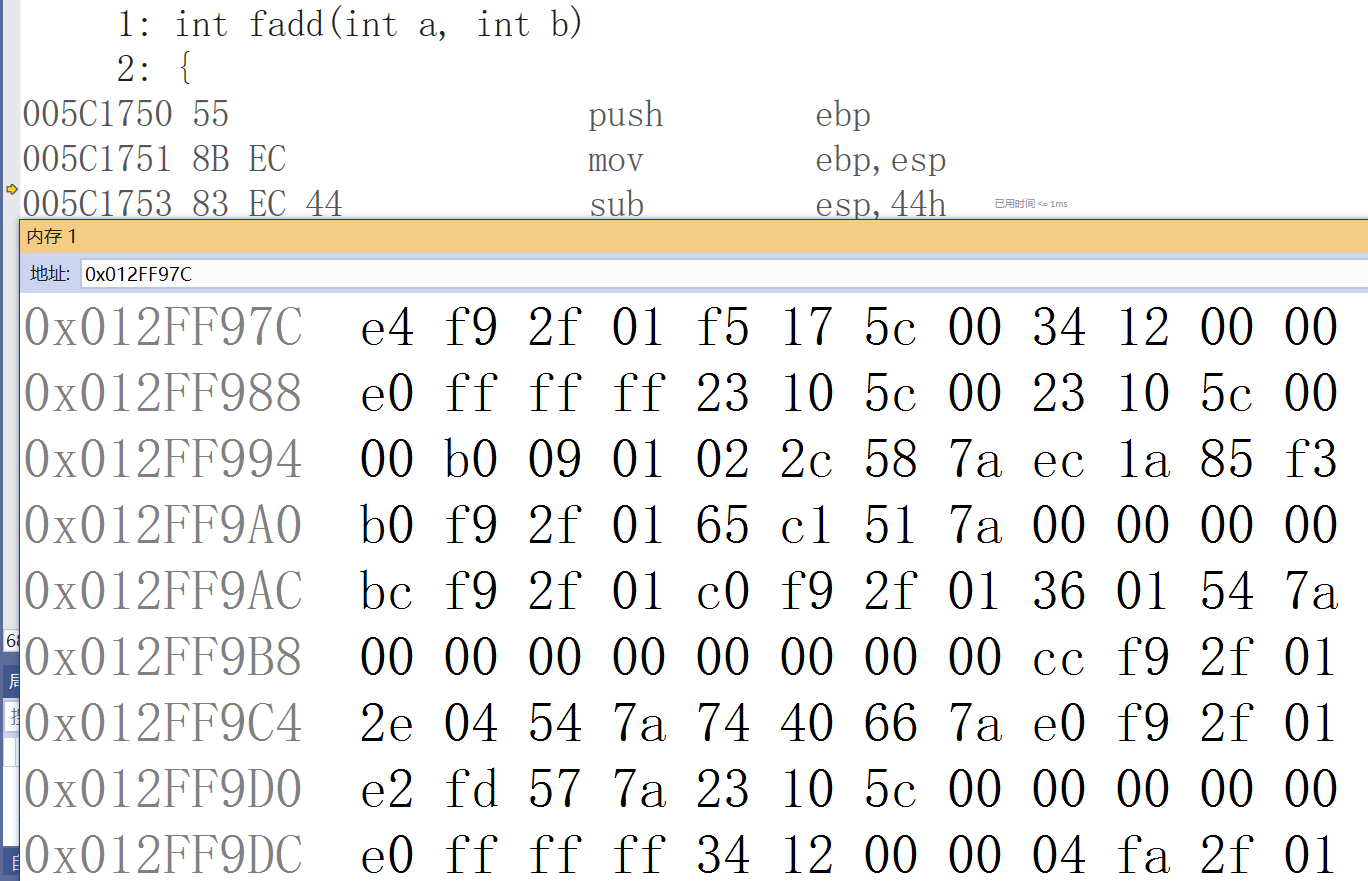


(3) 调用函数时是如何传递参数的，有何规律？参数所占的空间是如何“释放的”？（即参数在的位置）

**答：根据看到的反汇编代码，总结本程序的参数传递方式。**

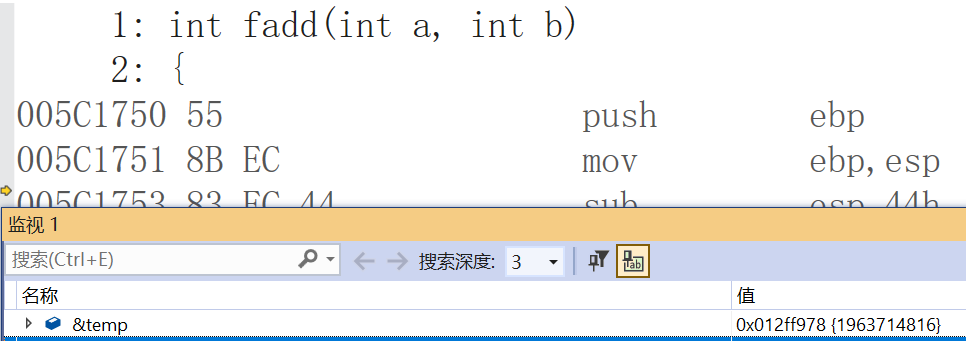
**按照参数列表从后向前用堆栈传递参数，即最右边的参数最先压栈，最左边的参数最后压栈。在函数返回后通过add esp,8，即两个int类型参数的长度来“释放”空间。**

（4）画出进入函数fadd，保存 EBP 后，堆栈的示意图（要包括main中局部变量x,y,result等相关内容，在栈单元外，标明地址；中间有些不相关的内存单元可用省略号代替）。写出参数 a、b的地址表达形式，以及具体的地址值。写出fadd中temp的地址表达形式，以及具体的地址值。



堆栈示意图

|  |  |  |
| --- | --- | --- |
| 符号地址 | 单元中的内容 | 地址(由小到大 |
|  |  |  |
| temp |  | 0x012FF978 |
| esp/ebp-> | 0x012ff9e4 原ebp | 0x012FF97C |
|  | 0x005c17f5 断点地址 | 0x012FF980 |
| a | 0x1234 | 0x012FF984 |
| b | -32 (0xffffffe0) | 0x012FF988 |
| 调用fadd前的栈顶 |  | 0x012FF98C |
|  | ……… |  |
| result | 0 | 0x012FF9D8 |
| y | -32 (0xffffffe0) | 0x012FF9DC |
| x | 0x1234 | 0x012FF9E0 |
| EBP -> |  | 0x012FF9E4 |



参数 a 的地址表达形式： [ebp+8], 具体地址：0x012FF984

参数 b 的地址表达形式： [ebp+0c], 具体地址：0x012FF988

temp 的地址表达形式： [ebp-04], 具体地址：0x012FF978

(5) 执行完 result=fadd(x,y)后，ESP、EBP的值各是多少？与执行result=fadd(x,y) 之前的 ESP、EBP是否相同？如何看待函数执行对栈空间的使用（参数以及局部变量的空间分配与释放如何完成的）？  
 答：执行完该C语句后，ESP = 0x012FF98C； EBP = 0x012FF9E4。与执行语句前的ESP、EBP相同。  
 在编译时，函数参数、局部变量的地址表达形式就已经确定。一旦开始执行函数，当前的栈空间就为函数所用，根据ebp的当前值，以及各变量相对于ebp的位移量，就可以确定参数、局部变量的具体地址，相当于为它们分配了空间。函数运行结束，栈顶指针的移动到执行函数前的位置，栈空间给其他函数所使用，相当于“释放了”空间。

1. 设有如下程序

int fadd(int a, int b)

{ int temp;

temp = a + b;

return temp;

}

int\* fadd\_bug(int a, int b)

{ int temp; // 编译时会有警告

temp = a + b;

return &temp;

}

void main( )

{ int x = 1;

int y = 2;

int result;

int \*p;

p = fadd\_bug(x, y);

result = \*p; // ①

x = fadd(10,20);

result =\*p; // ②

}

同上一题：“基本运行时检测”设置为“默认值”； “安全检查”设置为“禁用安全检查”。

1. 在程序编译时，会给出什么警告信息？

**答： 返回局部变量的地址或临时 : temp**

1. 在 执行①处的 result =\*p后， result=? p=? (指p中存放的地址)  
    在 执行②处的 result =\*p后， result=? p=?

**答：** result=3 p= temp的地址 （可以填具体的值）

result=30 p= temp的地址 （p 中的内容没有变化，即p指向同一个单元； 但被p指向的单元中的内容发生了变化）

1. 为什么返回局部变量的地址是一种危险的做法？

**答：**函数运行结束后，原局部变量所占用的内存被“释放”，即对应的空间可被其他函数所使用。返回局部变量地址，而对应的空间又被其他函数所使用，其内容并非函数执行时的单元内容，处于不可控状态。返回局部变量的地址，然后通过该地址去进行操作，可能导致无法预料的问题。