# StackFrame分析报告

## 夏雨轩-21009201006

**题目：用OllyDbg分析老师提供的StackFrame.exe中的main函数执行过程。**

用图示画出以下时刻的栈状态：

1. 刚进入 main

2. 执行完 long a=1, b=2;

3. add 所需参数已压栈，调用 add 之前

4. 进入 add 函数之后

5. 执行完 long x=a, y=b;之后

6. add 函数的 return 之前

7. add 返回到 main 之后

8. printf 所需参数已压栈，调用 printf 之前

9. printf 返回后 main 返回之前

用文字描述这 9 个栈状态，相邻的两者之间程序都执行了什么操作。

**栈帧是通过利用EBP寄存器作为栈帧指针，以及ESP寄存器作为栈顶指针来访问栈内的局部变量、参数和函数返回地址的一种机制。在调用函数时，首先需要将当前ESP的值保存到EBP中，从而在函数内部保持一个基准点。这样，无论ESP如何变化，通过EBP的值作为基准点，可以安全地访问到相关函数的局部变量、参数和返回地址。这就是EBP寄存器作为栈帧指针的作用。**

**以下是 StackFrame.Exe 的代码：**

#include "stdio.h"

long add(long a, long b)

{

long x = a, y = b;

return (x + y);

}

int main(int argc, char\* argv[])

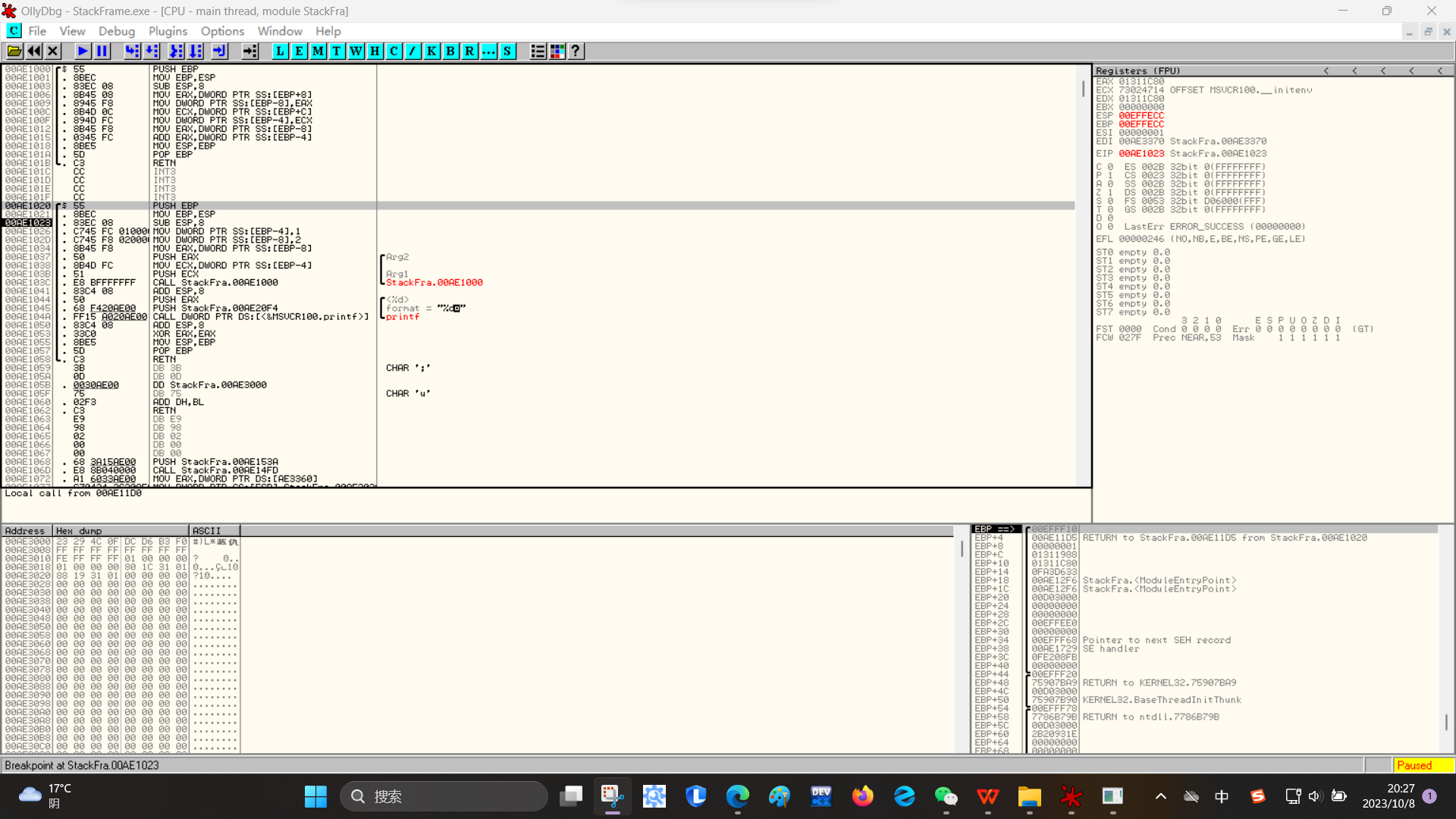
{

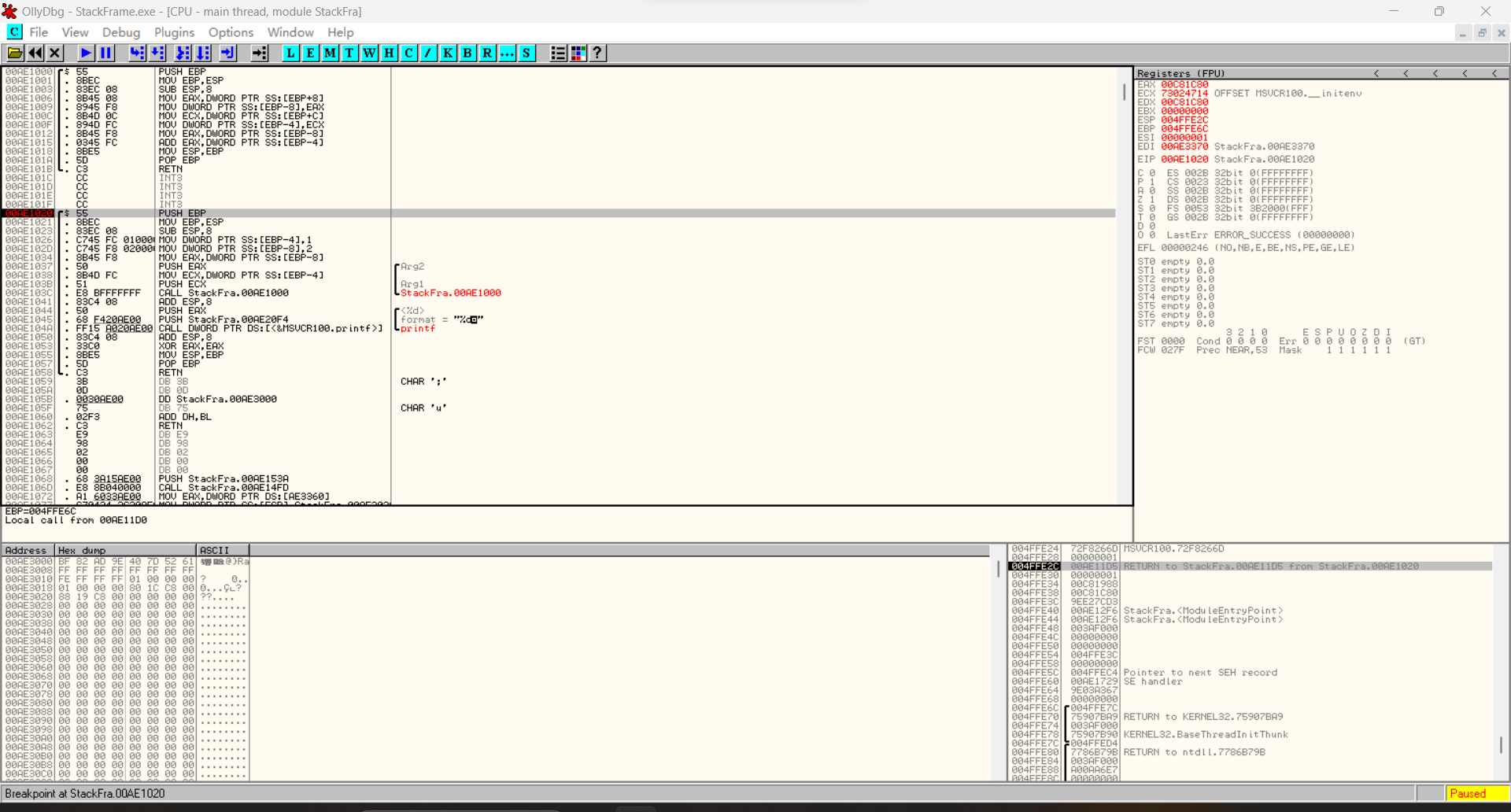
long a = 1, b = 2;

printf("%d\n", add(a, b));

return 0;

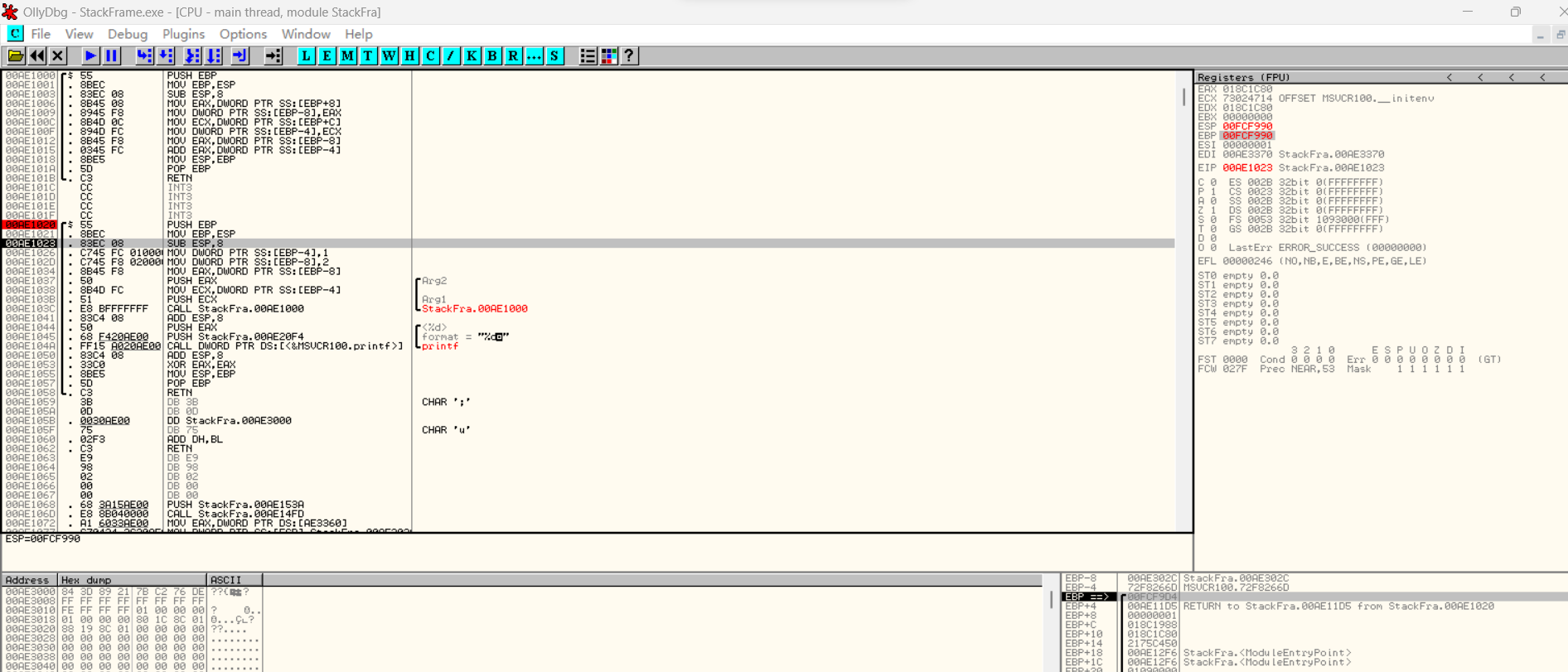
}

使用 OllyDbg 打开 StackFrame.Exe，此处使用的是版本 1.10。 当刚进入 `main()` 函数时，首先需要定位到该函数的起始位置。一种常用的方法是使用快捷键 Ctrl+G 进行快速定位。如果由于不同的运行环境导致位置不一样，也可以通过函数注释来进行定位。

例如，可以在代码中找到 `main()` 函数的起始位置，然后在其前面的位置（例如第 55 行）按下 F2 键来设置一个断点。接着按下 F9 键运行程序，此时可以在右下角的栈窗口中观察到运行 `main()` 函数后的栈状态。

在进入 `main()` 函数时，当前ESP的值为004FFE2C，而EBP的值为004FFE6C。这是因为在函数执行之前，先要将用作基准点的ESP值保存到EBP中，以便在函数内部可以安全地访问相关的局部变量、参数和返回地址。

在OllyDbg的栈窗口中，可以右键选择“address--relative to EBP”，确认EBP的位置，然后调试程序到EBP的状态。此时可以将地址转换为相对于EBP的偏移量，以便更好地理解栈的结构和数据存储方式。



在执行完 `long a=1, b=2;` 语句后，栈内情况如下：

| a = 1 |

| b = 2 |

|-------------------|

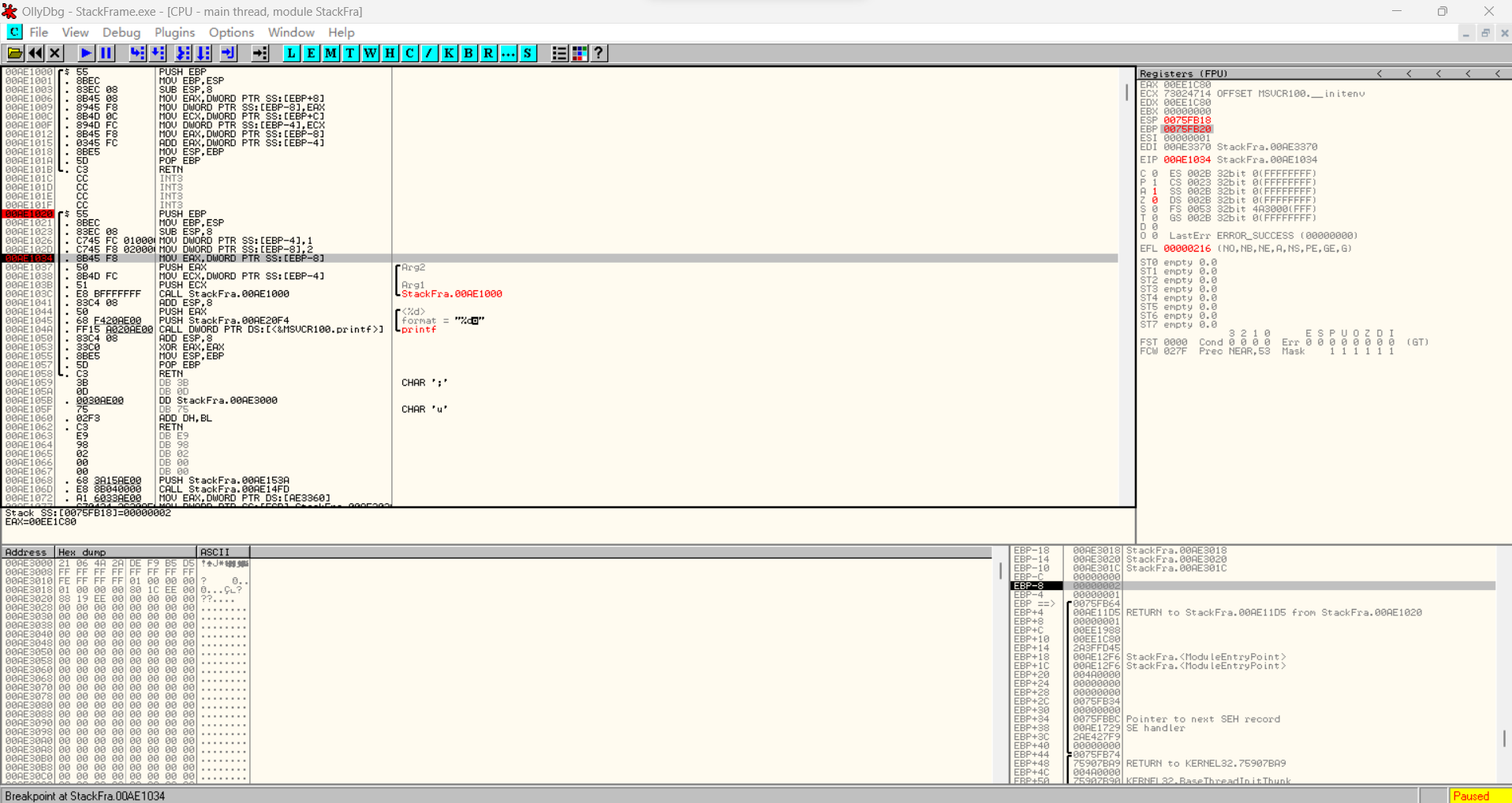
| |

| |

|-------------------| <--- ESP

这两条语句为变量 a 和 b 在栈中分配了空间，并将初始值分别设置为 1 和 2。在这个过程中，ESP的值减少了8个字节，这是为了为 a 和 b 两个局部变量在栈中开辟空间。

接下来的两条MOV指令分别将数据1和2保存在 [EBP-4] 和 [EBP-8] 中，即变量 a 和 b 的存储位置。

经过这两条指令的执行后，函数的栈状态如上所示。

1. **add 所需参数已压栈，调用 add 之前**



在调用 `add` 函数之前，函数的栈空间如下：

| a = 1 |

| b = 2 |

| x = |

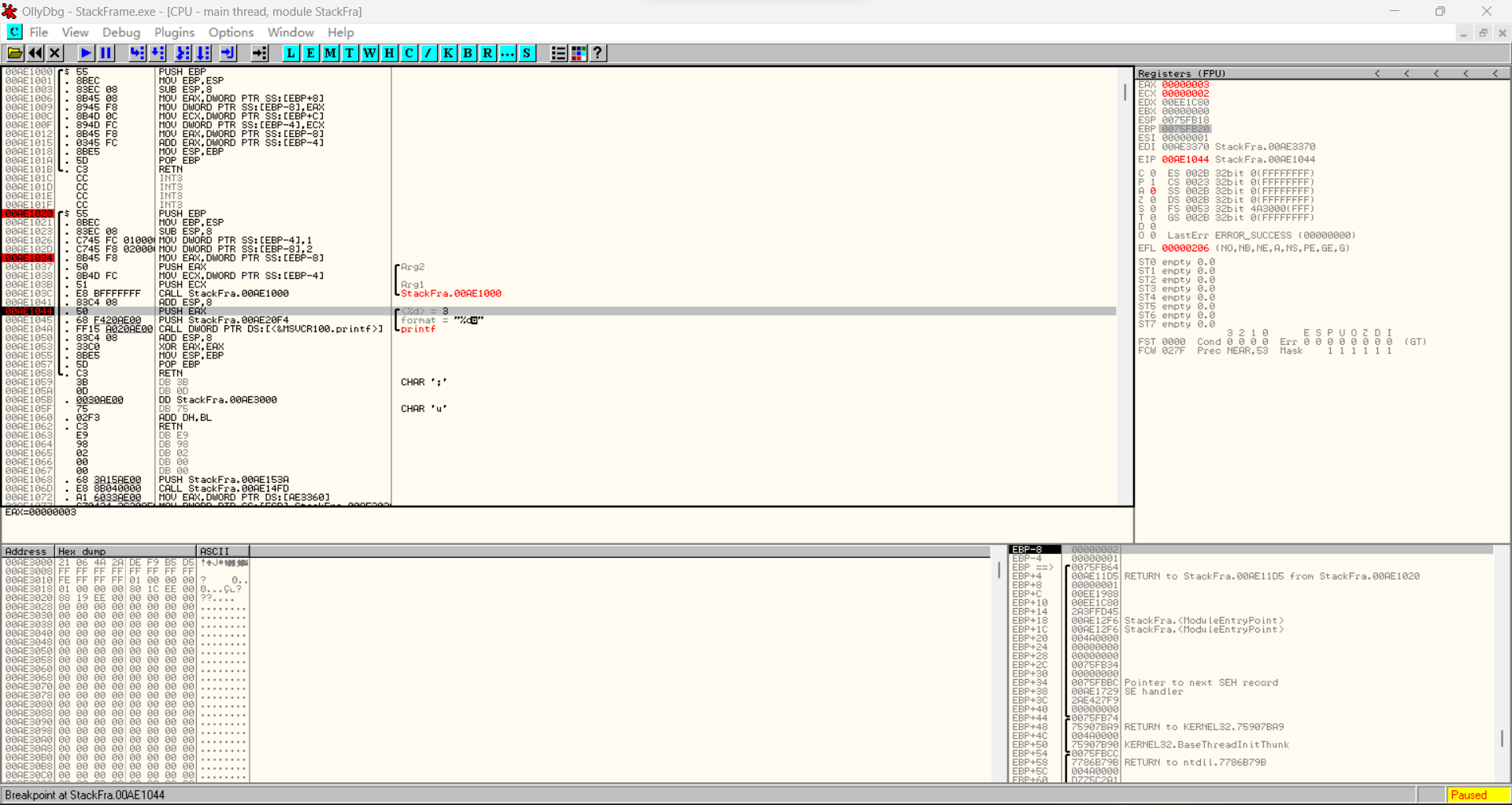
| y = |

|-------------------|

| |

| |

|-------------------| <--- ESP

在这之前，指令 `MOV EAX, [EBP-8]` 将从地址 [EBP-8] 处读取的32位数据移动到寄存器 EAX 中，然后将寄存器 EAX 中的值压入栈中，指令 `MOV ECX, [EBP-4]` 与此类似，它将从地址 [EBP-4] 处读取的32位数据移动到寄存器 ECX 中，然后将寄存器 ECX 中的值压入栈中。

在压栈的过程中，[EBP-8]先入栈，[EBP-4]后入栈，之后的call语句进入被调用的函数之前。

1. **进入 add 函数之后**

在进入 `add` 函数之后，函数的栈内情况如下：

| a = 1 |

| b = 2 |

| x = |

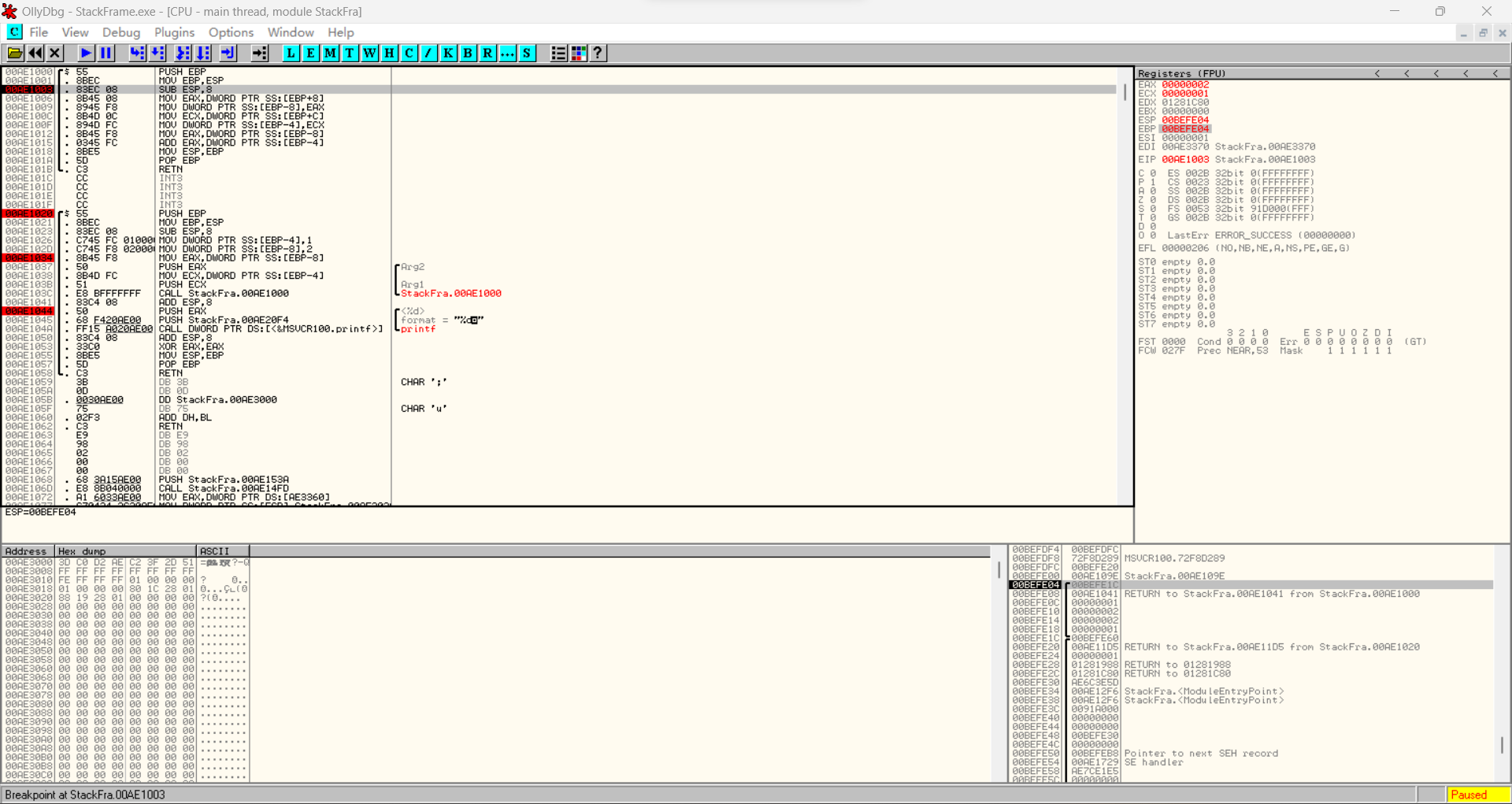
| y = |

|-------------------|

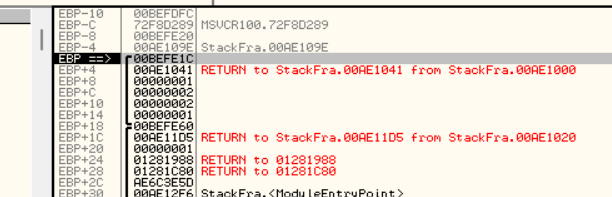
| |

| |

|-------------------| <--- ESP

这里的前两行汇编代码与开始执行 `main()` 函数时的代码完全相同。首先，将EBP值，也就是 `main()` 函数的基址指针，保存到栈中，然后将当前ESP存储到EBP中。这样，函数 `add` 的栈帧就生成了。在 `add` 函数内部，EBP值始终保持不变。

可以发现，main（）函数使用的EBP值被备份到了栈中，然后被赋予了一个新值，如图：



1. **执行完 long x=a, y=b;之后**

在执行完 `long x = a, y = b;` 语句后，在栈内存中为局部变量 x、y 分配了 8 个字节的空间。

add 函数的栈帧生成之后，EBP的值发生了变化，[EBP+8] 与 [EBP+C] 分别指向参数 a 与 b，而 [EBP-8] 与 [EBP-4] 则分别指向 add() 函数的两个局部变量 x、y。执行完 `long x = a, y = b;` 后，栈内情况如下：

| a = 1 |

| b = 2 |

| x = 1 |

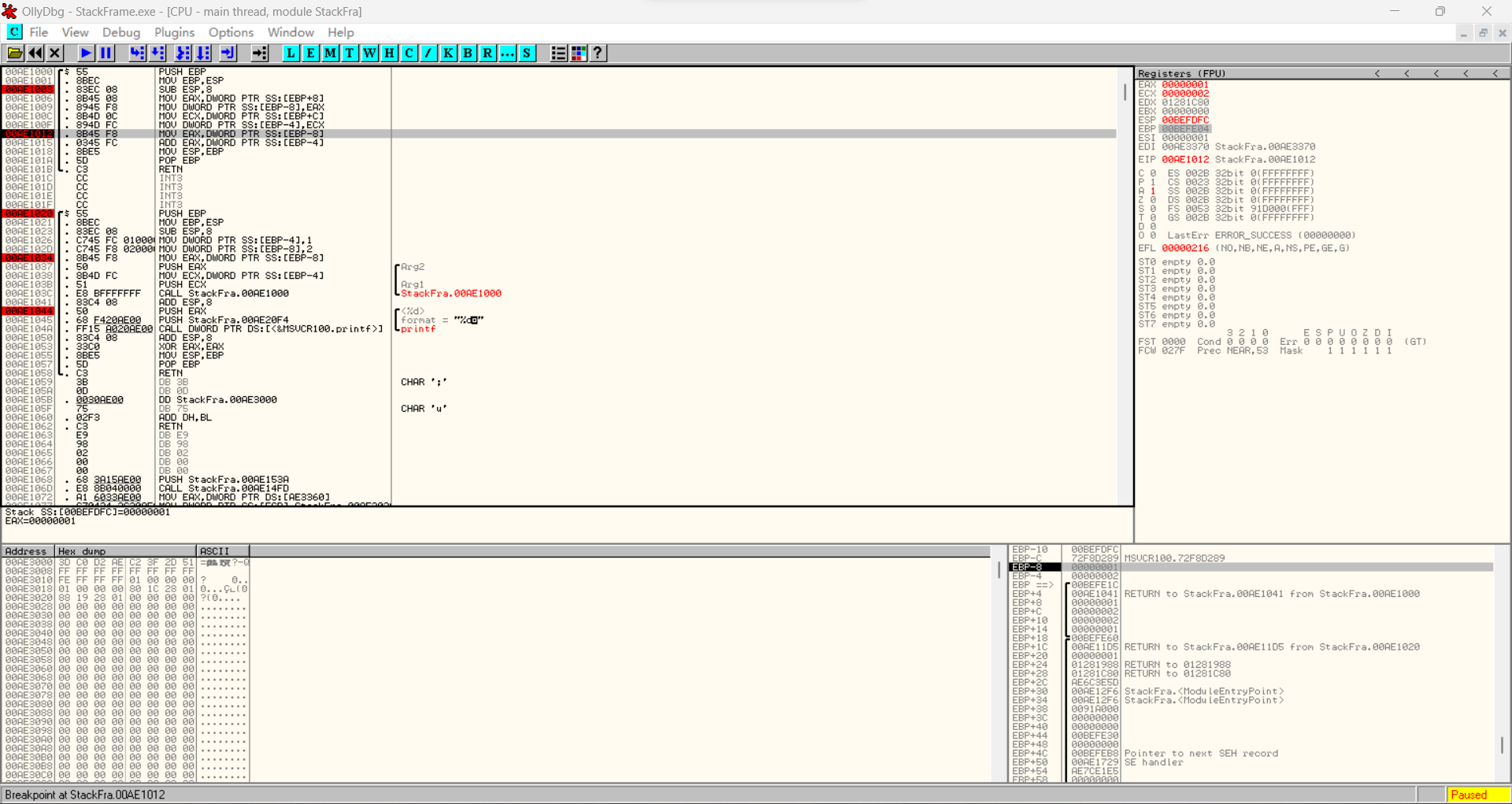
| y = 2 |

|-------------------|

| |

| |

|-------------------| <--- ESP

这里表示在栈中已经为变量 a、b、x、y 分配了相应的空间。

1. **add 函数的 return 之前**

在执行完 `return (x + y);` 语句前，栈内情况如下：

| a = 1 |

| b = 2 |

| x = 1 |

| y = 2 |

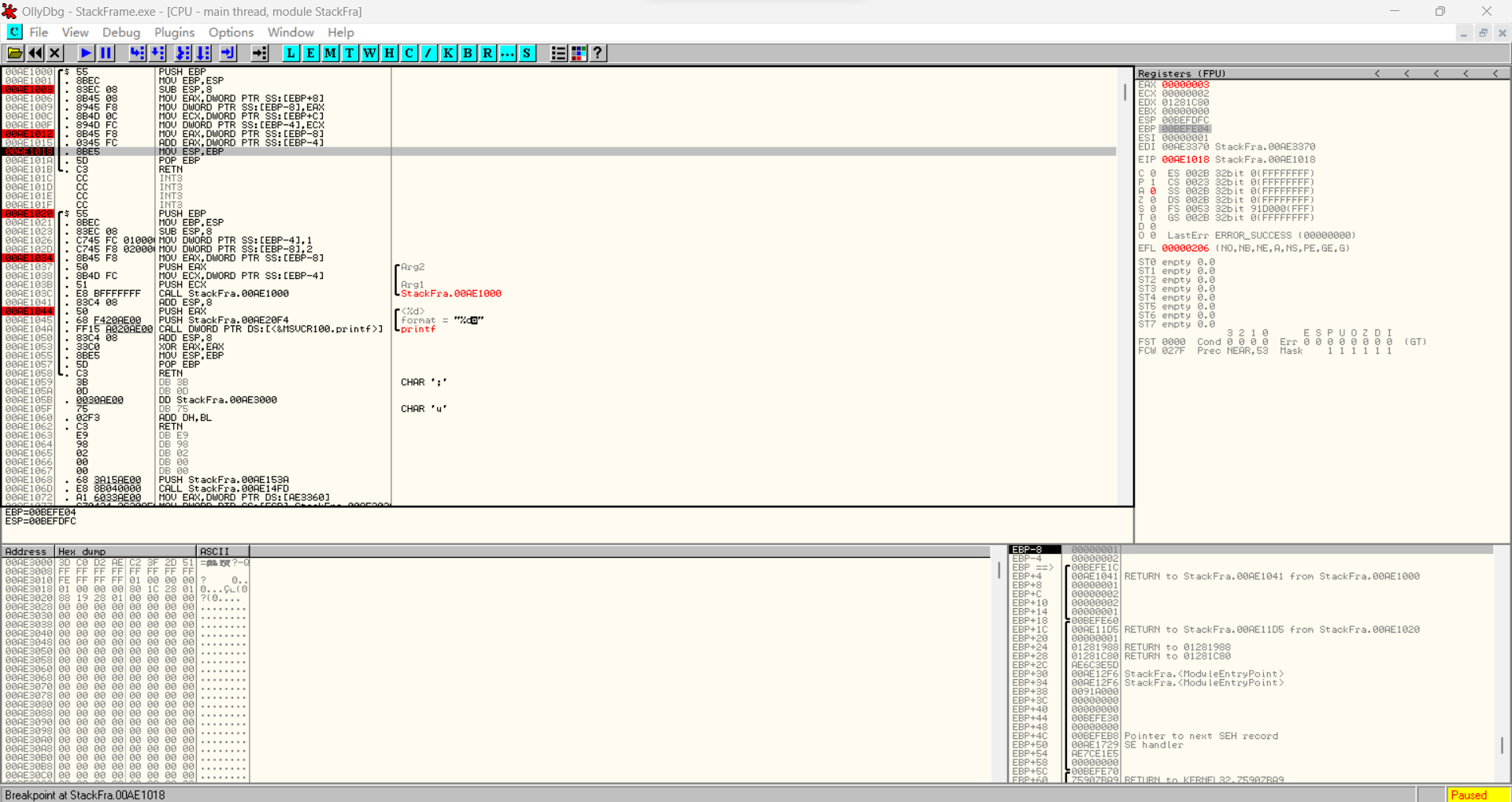
| return val |

|-------------------|

| |

| |

|-------------------| <--- ESP

在这里，`return (x + y);` 用于返回局部变量 x 与 y 的和，同时将变量 x 的值传送到 EAX 寄存器。然后，将变量 y 与 EAX 中的原值（即 x）相加，将运算结果存储在 EAX 中。运算后，EAX 中的值为 3。

1. **add 返回到 main 之后**

在返回到 `main()` 函数之后，栈的状态如下：

| a = 1 |

| b = 2 |

| |

|-------------------|

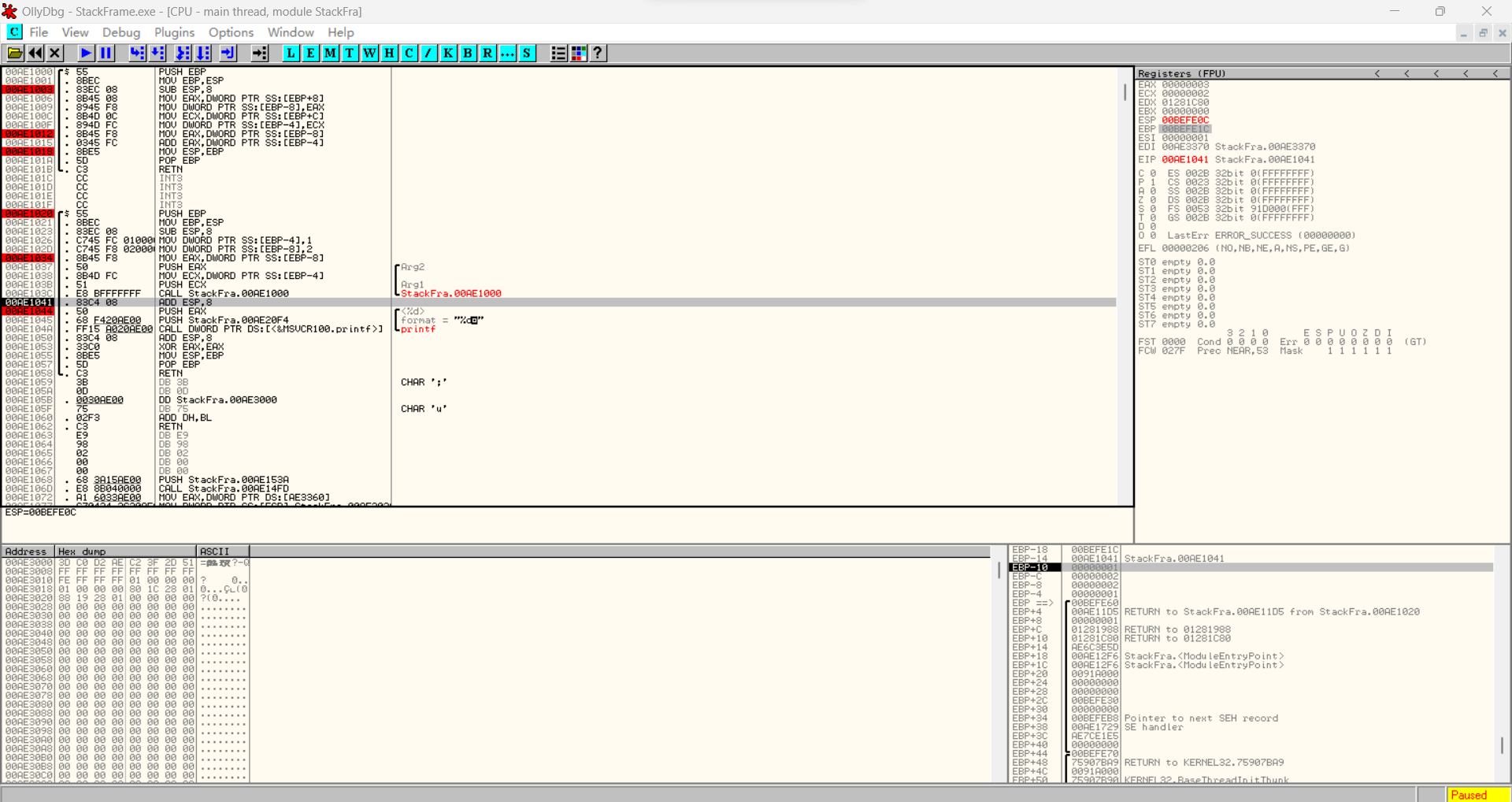
| |

| |

|-------------------| <--- ESP

第一行汇编指令将当前的EBP值赋给ESP，这与之前的 `MOV EBP, ESP` 命令相对应。`MOV EBP, ESP` 命令将函数 `add()` 开始执行时的ESP值放入EBP，而在函数执行完毕时，使用地址 `00AE1018` 处的 `MOV ESP, EBP` 命令将存储在EBP中的值恢复到ESP中。

接着，第二行命令恢复了函数 `add()` 开始执行时备份到栈中的EBP值，它与之前的 `PUSH EBP` 命令对应。

最后一行命令使得存储在栈中的返回地址被返回，从而返回到了 `main()` 函数。

1. **printf 所需参数已压栈，调用 printf 之前**

在调用执行 `printf` 前，程序运行到了地址 `00AE1041` 处。在这里，使用了 `ADD` 命令将ESP加上了8个字节，因为在压栈之后，已经不需要参数 a 和 b 了。这两个长整型参数一共占了8个字节，所以将ESP加上8，将它们从栈中清理掉。

参数压栈之后，栈的状态如下：

| format str |

| add result |

|-------------------|

| a = 1 |

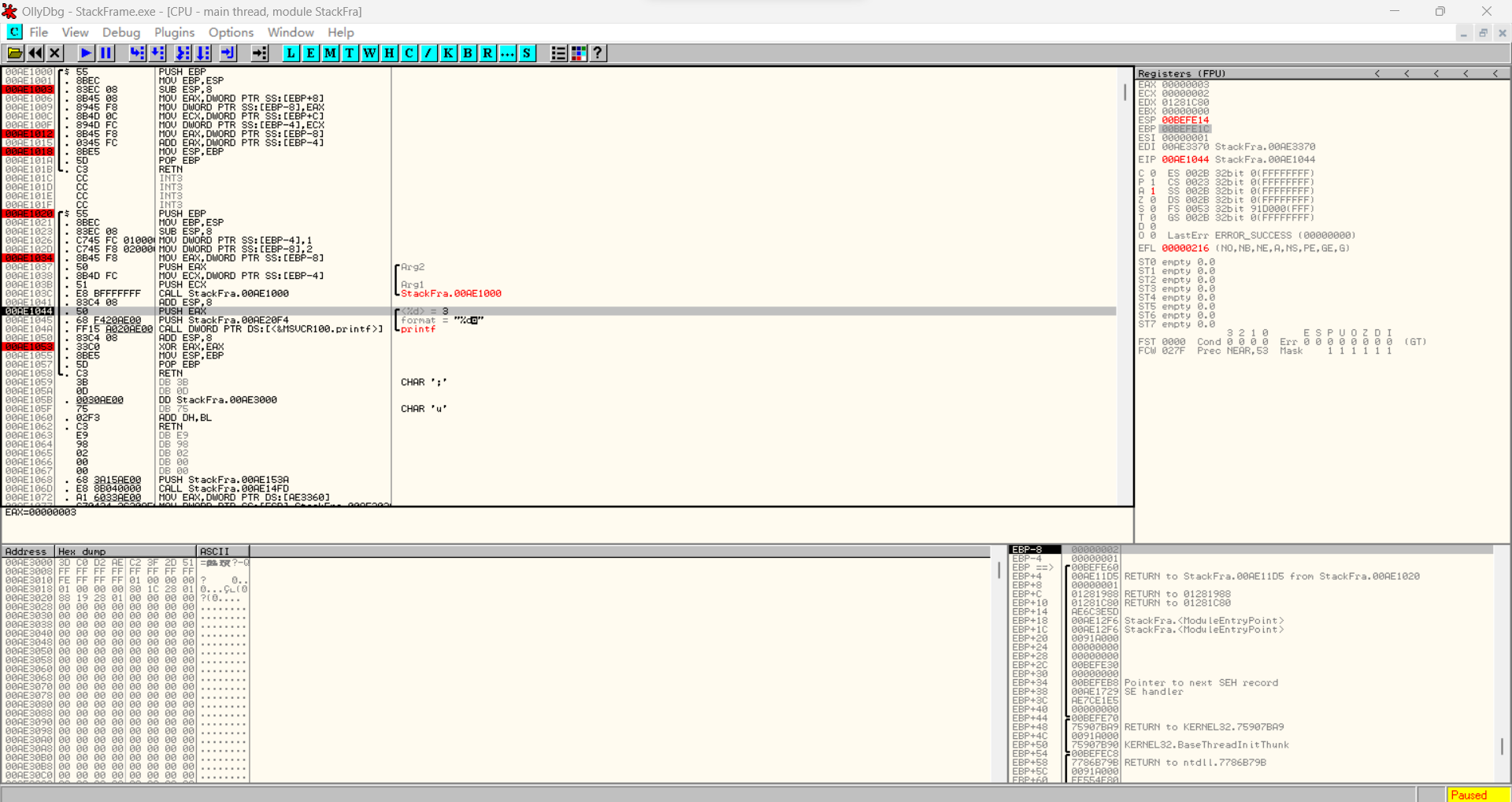
| b = 2 |

|-------------------|

| |

| |

|-------------------| <--- ESP

这里表示参数已经被压栈，准备进行 `printf` 函数的调用。

1. **printf 返回后 main 返回之前**

在调用 `printf` 后，EAX寄存器中存储着函数 `add()` 的返回值，即执行加法运算后的结果值3。

在第三行的命令中调用了 `printf()` 函数，`printf()` 函数执行完毕后通过 `ADD` 命令删除了参数。这发生在返回到 `main()` 函数之前。

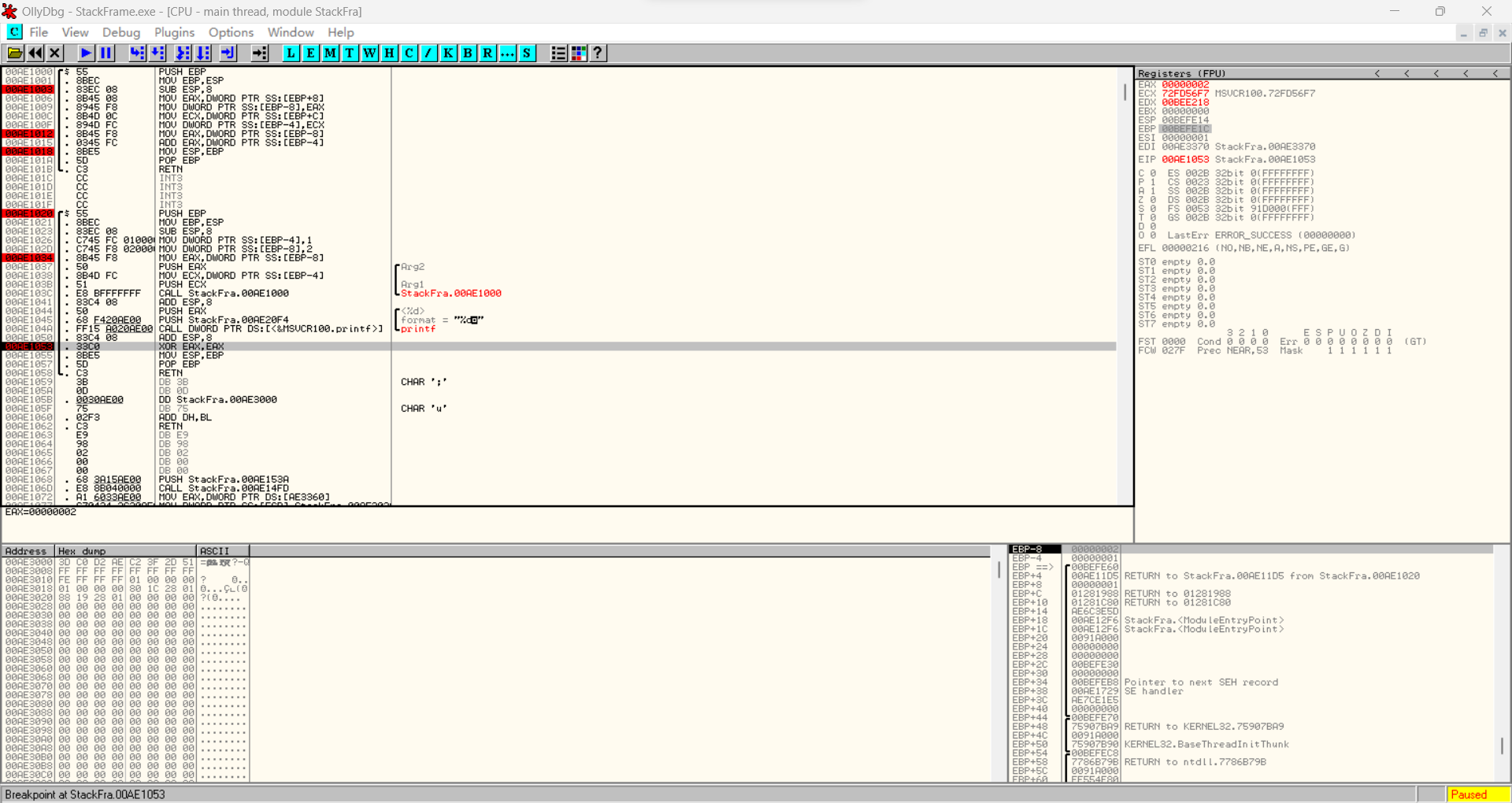
此时，栈的状态如下：

|-------------------|

| |

| |

|-------------------| <--- ESP

表示在调用 `printf` 后，参数已经被清理，准备返回到 `main()` 函数。

**以下是相关函数执行过程中的栈状态：**

**1. 刚进入 `main` 函数：**

**```**

**| |**

**| |**

**| |**

**| main() |**

**| |**

**| |**

**|------------------| <--- ESP**

**```**

**2. 执行完 `long a = 1, b = 2;`：**

**```**

**| |**

**| |**

**| |**

**| main() |**

**| a = 1 |**

**| b = 2 |**

**|------------------|<--- ESP**

**```**

**此时，变量 `b` 先入栈，接着是变量 `a`。**

**3. 在调用 `add` 函数之前：**

**```**

**| |**

**| |**

**| a = 1 |**

**| b = 2 |**

**| add() |**

**|------------------|<--- ESP**

**```**

**4. 进入 `add` 函数之后：**

**```**

**| |**

**| |**

**| a = 1 |**

**| b = 2 |**

**| x = |**

**| y = |**

**|------------------|<--- ESP**

**```**

**5. 执行完 `long x = a, y = b;` 之后：**

**```**

**| |**

**| |**

**| a = 1 |**

**| b = 2 |**

**| x = 1 |**

**| y = 2 |**

**|------------------|<--- ESP**

**```**

**6. 在 `add` 函数的返回之前：**

**```**

**| |**

**| |**

**| a = 1 |**

**| b = 2 |**

**| x = 1 |**

**| y = 2 |**

**| return val |<--- ESP**

**|------------------|**

**```**

**7. 返回到 `main` 函数之后：**

**```**

**| |**

**| |**

**| a = 1 |**

**| b = 2 |**

**| |**

**| |**

**|------------------|<--- ESP**

**```**

**8. 在调用 `printf` 之前：**

**```**

**| |**

**| |**

**| format str |**

**| add result |**

**| |**

**| |**

**|------------------|<--- ESP**

**```**

**9. 在 `printf` 返回后，返回到 `main` 之前：**

**```**

**| |**

**| |**

**| |**

**|------------------|<--- ESP**

**```**

**这些图示模拟了栈状态，实际的栈布局可能会受到硬件平台的影响。**