

# 同济大学计算机系

## 操作系统课程实验报告



学 号 2251557

姓 名 代文波

专 业 计算机科学与技术

授课老师 方钰

## 一、实验目的

结合课程所学知识，通过编写一个简单的 C++ 代码，并在 UNIX V6++ 中编译和运行调试，观察程序运行时栈帧的变化。通过实践，进一步掌握 UNIX V6++ 重新编译及运行调试的方法。

## 二、实验设备及工具

已配置好的 UNIX V6++ 运行和调试环境。

## 三、实验内容

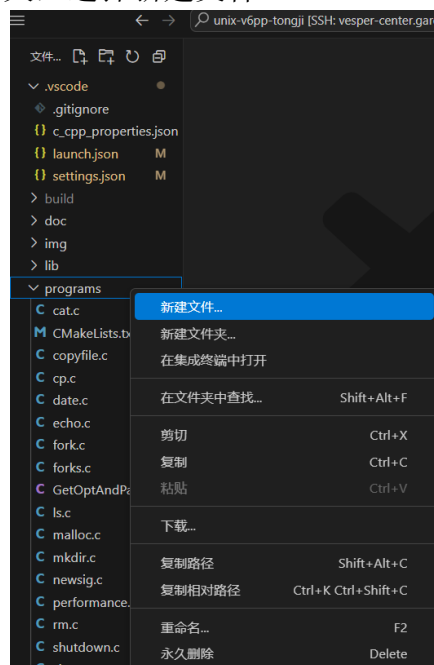
- 1、在 UNIX V6++ 中编译链接运行一个 C 语言程序
- 2、进行程序的调试与运行
- 3、观察 main1 函数的堆栈的变化
- 4、观察 sum 函数的堆栈的变化

## 四、实验操作

### 4.1 在 UNIX V6++ 中编译链接运行一个 C 语言程序

#### 4.1.1 在 program 文件夹中添加一个新的 C 语言文件

- (1) 右键 programs 文件夹，选择新建文件



- (2) 建立好了文件



```
QEMU - Press Ctrl+Alt+G to release grab
Machine View
welcome to Unix V6++ Tongji's Edition!
[/]#ls
Directory '/':
dev    bin    etc    Shell.exe
[/]#cd bin
[/bin]#ls
Directory '/bin':
test    fork    mkdir    stack    showStack    rm    sigTest    performance    trace    cp    date
sig    forks    malloc    cat    sig    shutdown    echo    testSTDOUT    copyfile    news
[/bin]#showStack
result=3
[/bin]#

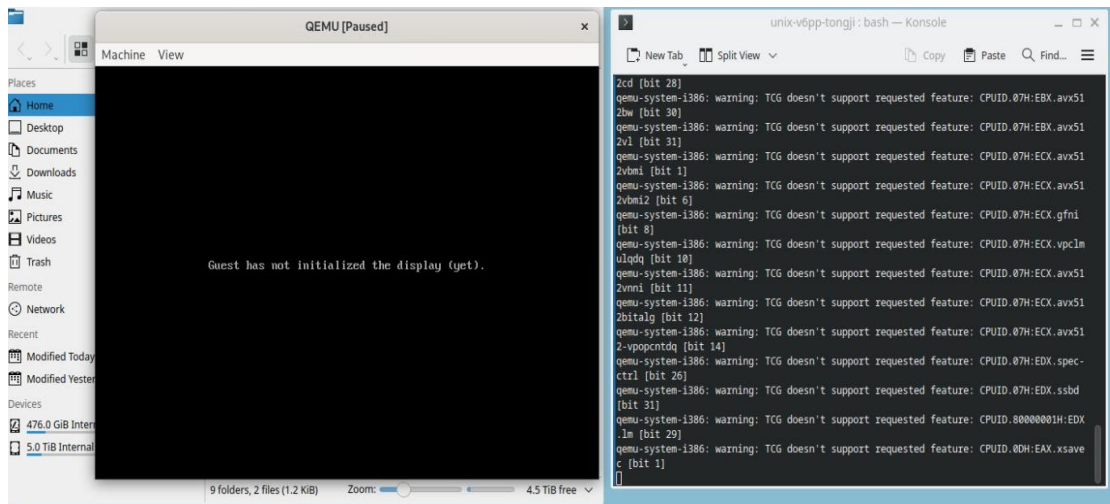
Process 1 finding dead son. They are Process 2 (Status:3) wait until child process Exit! Process 2
execing
regs->eax = -4294967294 , u.u_error = 2
Process 2 execing
Process 2 is exiting
end sleep
Process 2 (Status:5) end wait
Process 1 finding dead son. They are Process 3 (Status:3) wait until child process Exit! Process 3
execing
Process 3 is exiting
end sleep
Process 3 (Status:5) end wait
Process 1 finding dead son. They are Process 4 (Status:3) wait until child process Exit! Process 4
execing
Process 4 is exiting
end sleep
Process 4 (Status:5) end wait
```

(4) 修改调试目标:

```
文件(F) 编辑(E) 选择(S) 查看(V) ... unix-v6pp-tongji [SSH: vesper-center.gardlily.com]
. vscode > {} launch.json M x
. vscode > {} launch.json > Launch Targets > {} V6PP - build and debug kernel
3 "configurations": [
4 {
5     "name": "V6PP - build and debug kernel",
6     "type": "cppdbg",
7     "request": "launch",
8     "program": "${workspaceFolder}/target/objs/kernel.exe",
9     "args": [],
10    "cwd": "${workspaceFolder}/target",
11    "environment": {},
12    "console": "debugConsole",
13    //下面这个是调试内核
14    // "program": "${workspaceFolder}/target/objs/kernel.exe",
15    //下面这个是调试子程序,要调试哪个子程序就将其名称改在"apps-elf/"后面
16    // 例如, 要调试showStack程序
17    "program": "${workspaceFolder}/target/objs/apps-elf/showStack",
18    // "program": "${workspaceFolder}/target/objs/Shell.elf.exe",
19    // check this ^^^^^^^^^^^
20 }
```

## 4.2 开始程序的调试运行

### 4.2.1 启动 UNIX V6++的调试模式



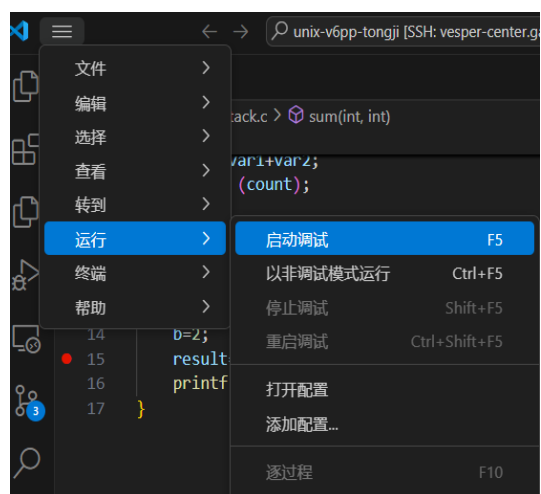
## 4.2.2 设置断点

```

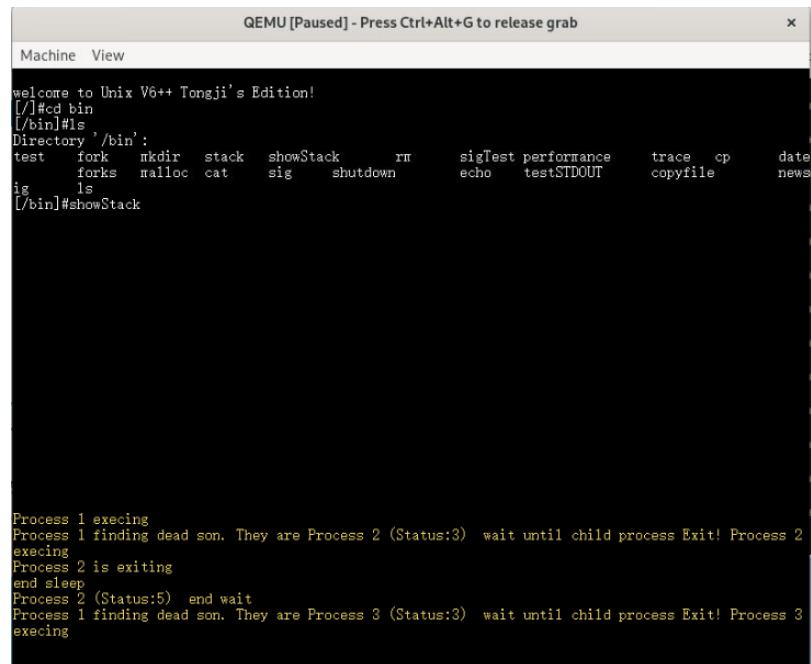
C showStack.c U X
programs > C showStack.c > sum(int, int)
1  #include <stdio.h>
2  int version = 1;
3  int sum(int var1,int var2)
4  {
5      int count;
6      version=2;
7      count=var1+var2;
8      return (count);
9  }
10 void main1()
11 {
12     int a,b,result;
13     a=1;
14     b=2;
15     result=sum(a,b);
16     printf("result=%d\n",result);
17 }

```

## 4.2.3 启动调试



#### 4.2.4 查看 UNIX V6++ 调试模式的运行状态

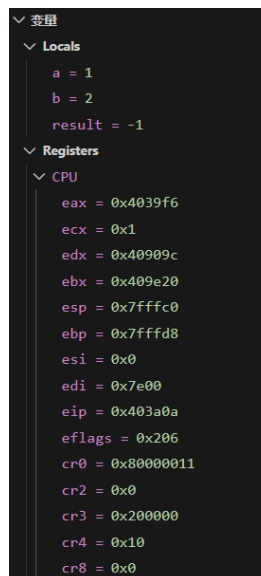


```
Machine View

welcome to Unix V6++ Tongji's Edition!
[/]#cd bin
[/bin]#ls
Directory '/bin':
test  fork  mkdir  stack  showStack  rm      sigTest  performance  trace  cp  date
ig    ls
[/bin]#showStack

Process 1 excec
Process 1 finding dead son. They are Process 2 (Status:3) wait until child process Exit! Process 2
excec
Process 2 is exiting
end sleep
Process 2 (Status:5) end wait
Process 1 finding dead son. They are Process 3 (Status:3) wait until child process Exit! Process 3
excec
```

#### 4.2.5 查看局部变量以及寄存器的值

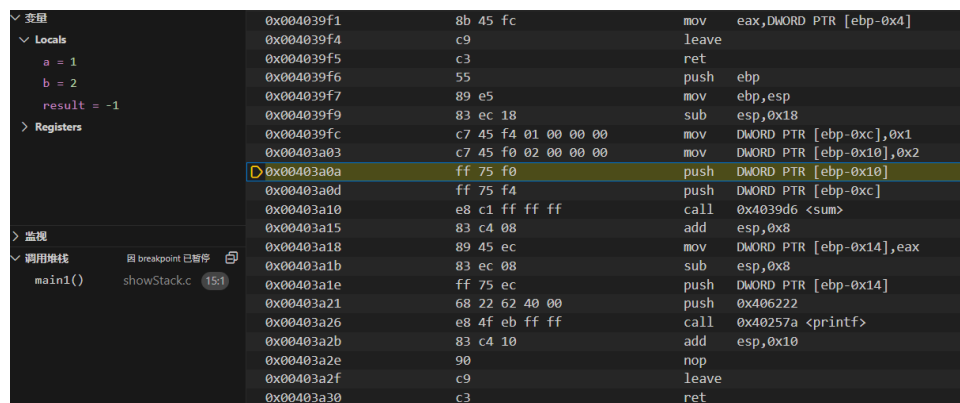


```

Locals
a = 1
b = 2
result = -1

Registers
CPU
eax = 0x4039f6
ecx = 0x1
edx = 0x40909c
ebx = 0x409e20
esp = 0x7fffc0
ebp = 0x7fffd8
esi = 0x0
edi = 0x7e00
eip = 0x403a0a
eflags = 0x206
cr0 = 0x80000011
cr2 = 0x0
cr3 = 0x200000
cr4 = 0x10
cr8 = 0x0
```

#### 4.2.6 查看程序的汇编代码:

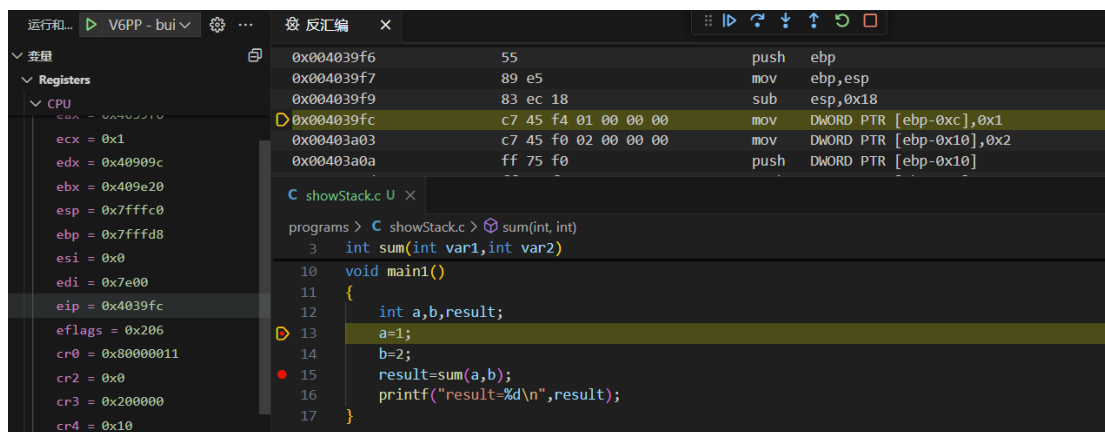


```

Locals
a = 1
b = 2
result = -1

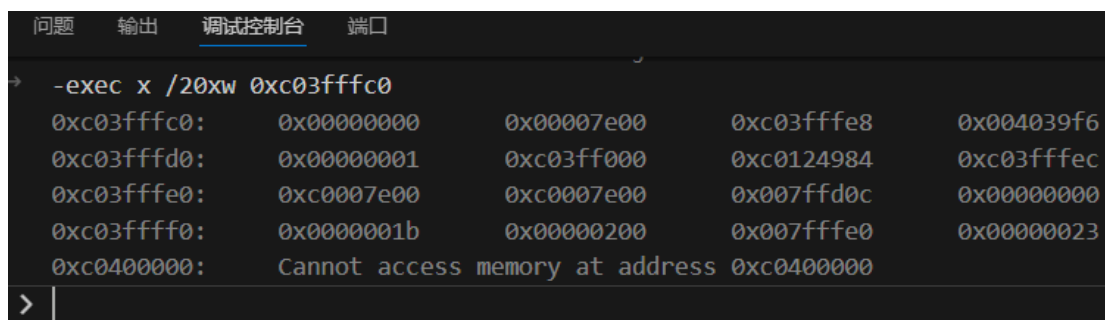
main1()
0x004039f1 8b 45 fc mov eax,DWORD PTR [ebp-0x4]
0x004039f4 c9 leave
0x004039f5 c3 ret
0x004039f6 55 push ebp
0x004039f7 89 e5 mov ebp,esp
0x004039f9 83 ec 18 sub esp,0x18
0x004039fc c7 45 f4 01 00 00 00 mov DWORD PTR [ebp-0xc],0x1
0x00403a03 c7 45 f0 02 00 00 00 mov DWORD PTR [ebp-0x10],0x2
0x00403a0a ff 75 f0 push DWORD PTR [ebp-0x10]
0x00403a0d ff 75 f4 push DWORD PTR [ebp-0xc]
0x00403a10 e8 c1 ff ff ff call 0x4039d6 <sum>
0x00403a15 83 c4 08 add esp,0x8
0x00403a18 89 45 ec mov DWORD PTR [ebp-0x14],eax
0x00403a1b 83 ec 08 sub esp,0x8
0x00403a1e ff 75 ec push DWORD PTR [ebp-0x14]
0x00403a21 68 22 62 40 00 push 0x406222
0x00403a26 e8 4f eb ff ff call 0x40257a <printf>
0x00403a2b 83 c4 10 add esp,0x10
0x00403a2e 90 nop
0x00403a2f c9 leave
0x00403a30 c3 ret
```

#### 4.2.7 根据 eip 查看对应的汇编代码：

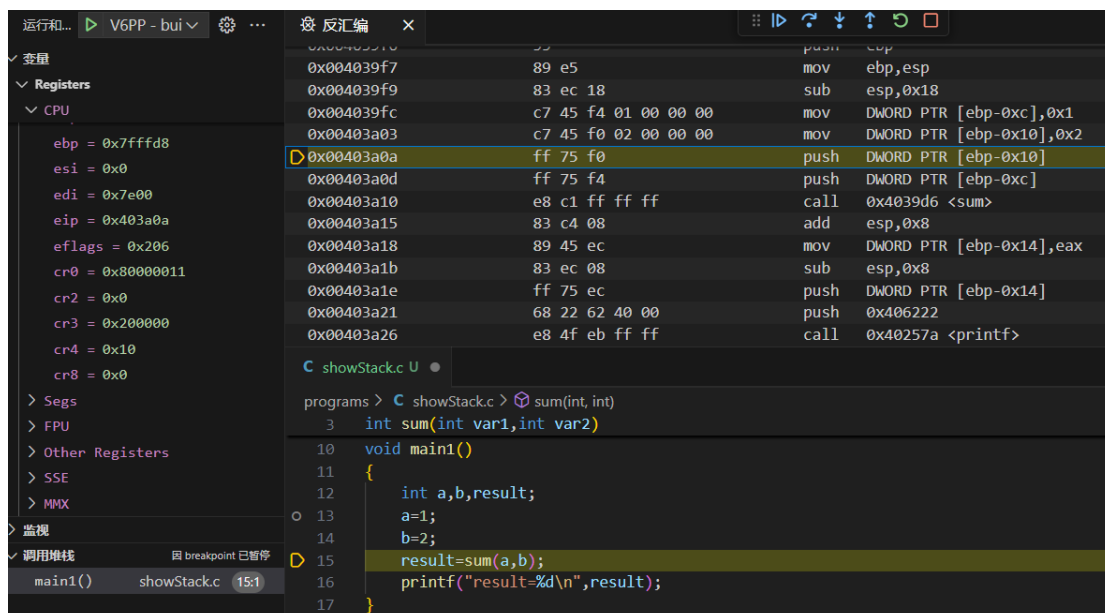


补充：这里需要注意的是：vscode 高亮显示的语句是下一步将要执行的语句！

#### 4.2.8 实现对内存单元的查看：



#### 4.2.9 观察 main1 堆栈变化（后面有详细的展示）



#### 4.2.10 相关的汇编代码：

0x004039f6	55	push	ebp
0x004039f7	89 e5	mov	ebp,esp
0x004039f9	83 ec 18	sub	esp,0x18
0x004039fc	c7 45 f4 01 00 00 00	mov	DWORD PTR [ebp-0xc],0x1
0x00403a03	c7 45 f0 02 00 00 00	mov	DWORD PTR [ebp-0x10],0x2
0x00403a0a	ff 75 f0	push	DWORD PTR [ebp-0x10]
0x00403a0d	ff 75 f4	push	DWORD PTR [ebp-0xc]
0x00403a10	e8 c1 ff ff ff	call	0x4039d6 <sum>
0x00403a15	83 c4 08	add	esp,0x8
0x00403a18	89 45 ec	mov	DWORD PTR [ebp-0x14],eax
0x00403a1b	83 ec 08	sub	esp,0x8
0x00403a1e	ff 75 ec	push	DWORD PTR [ebp-0x14]
0x00403a21	68 22 62 40 00	push	0x406222
0x00403a26	e8 4f eb ff ff	call	0x40257a <printf>
0x00403a2b	83 c4 10	add	esp,0x10
0x00403a2e	90	nop	
0x00403a2f	c9	leave	
0x00403a30	c3	ret	

### 4.3 复现 main1()用户栈的变化

#### 4.3.1 当前刚进入 main1()函数，刚刚将返回地址 0x00000008 填充到 esp 的位置

The screenshot shows a debugger window with the following components:

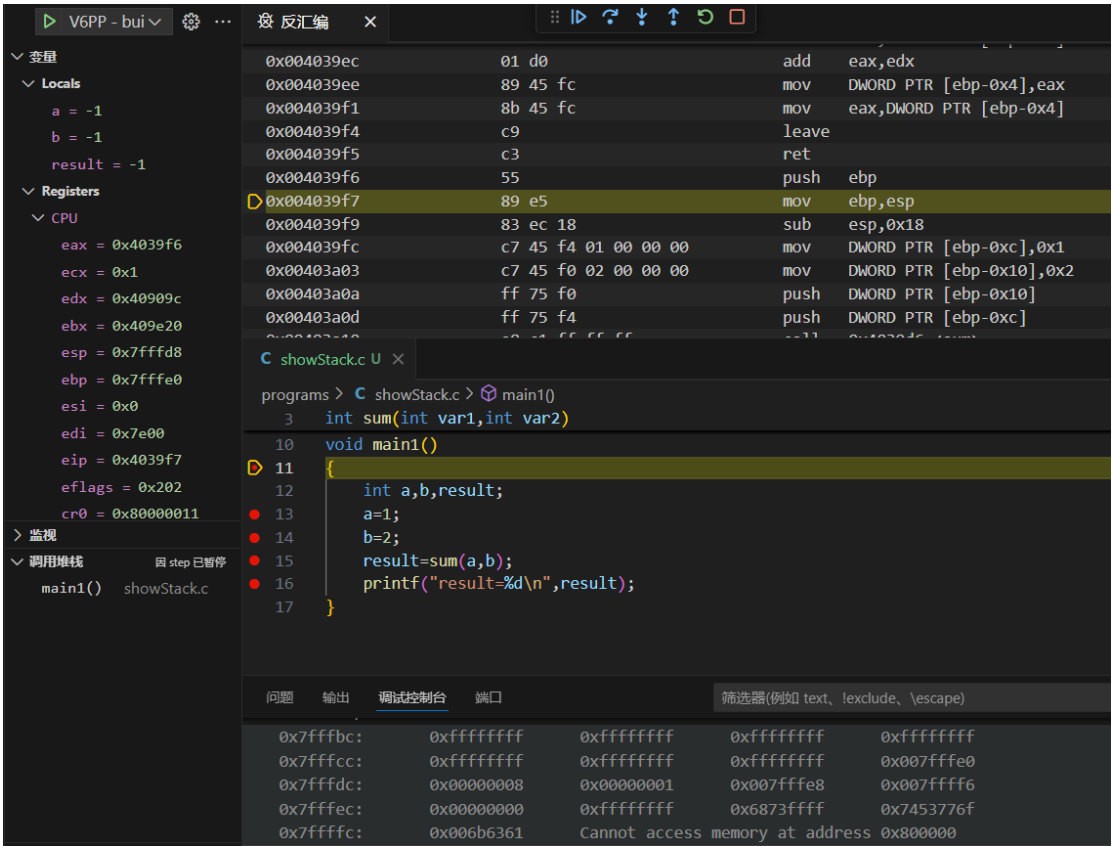
- Variables:** a = -1, b = -1, result = -1.
- Registers:** eax = 0x4039f6, ecx = 0x1, edx = 0x40909c, ebx = 0x409e20, esp = 0x7fffdc, ebp = 0x7fffe0, esi = 0x0, edi = 0x7e00, eip = 0x4039f6, eflags = 0x202, cr0 = 0x80000011.
- Stack:** main1() showStack.c
- Assembly:** The assembly code for main1() is shown, starting with push ebp, mov ebp, esp, sub esp, 0x18, mov DWORD PTR [ebp-0xc], 0x1, mov DWORD PTR [ebp-0x10], 0x2, push DWORD PTR [ebp-0x10], push DWORD PTR [ebp-0xc], call 0x4039d6 <sum>, add esp, 0x8, mov DWORD PTR [ebp-0x14], eax, sub esp, 0x8, push DWORD PTR [ebp-0x14], push 0x406222, call 0x40257a <printf>, add esp, 0x10, nop, leave, ret.
- Stack Frame:** The stack frame for main1() is shown, with the return address 0x00000008 at 0x7fffdc.



地址	内容	功能
0x7fffdc	0x00000008	main1()的返回地址

main1()的用户栈帧

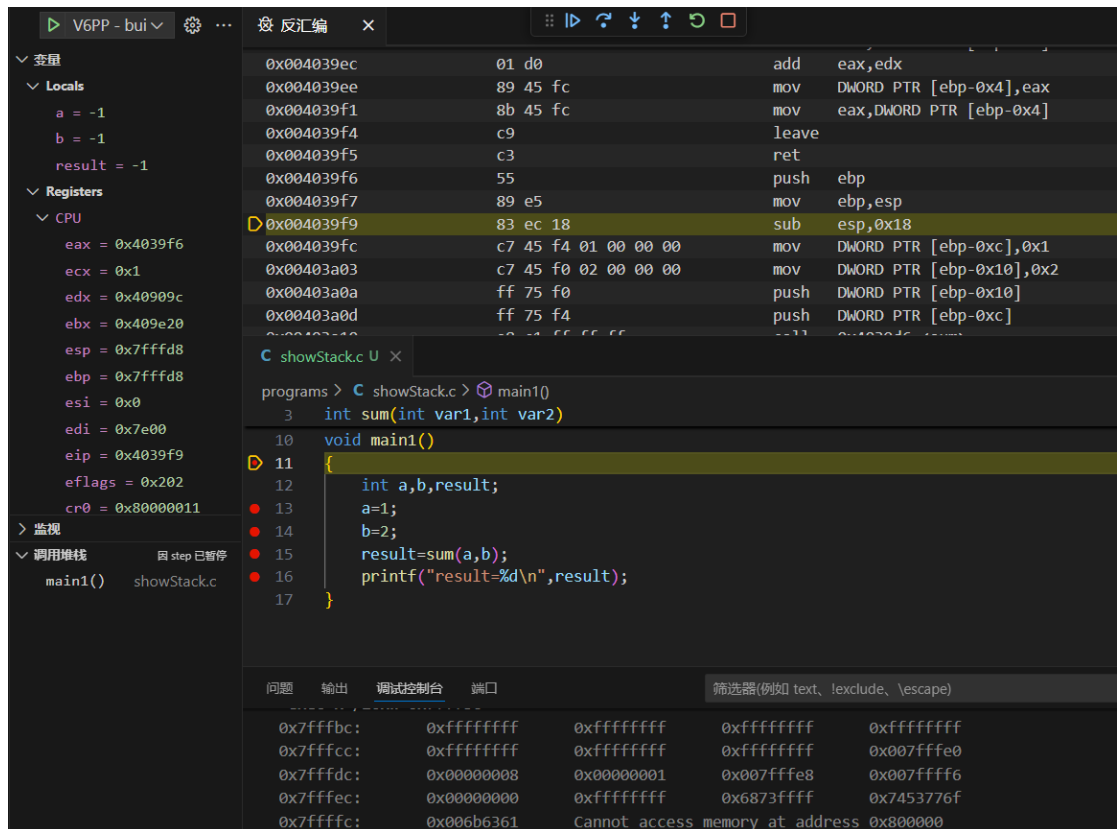
4.3.2 汇编指令：push ebp  
指令功能：将上一帧的ebp保存在这里



地址	内容	功能
0x7fffd8	0x007ffe0	记录前一帧的ebp
0x7fffdc	0x00000008	main1()的返回地址

main1()的用户栈帧

4.3.3 汇编指令：mov ebp,esp  
指令功能：这一步让ebp指向当前栈帧

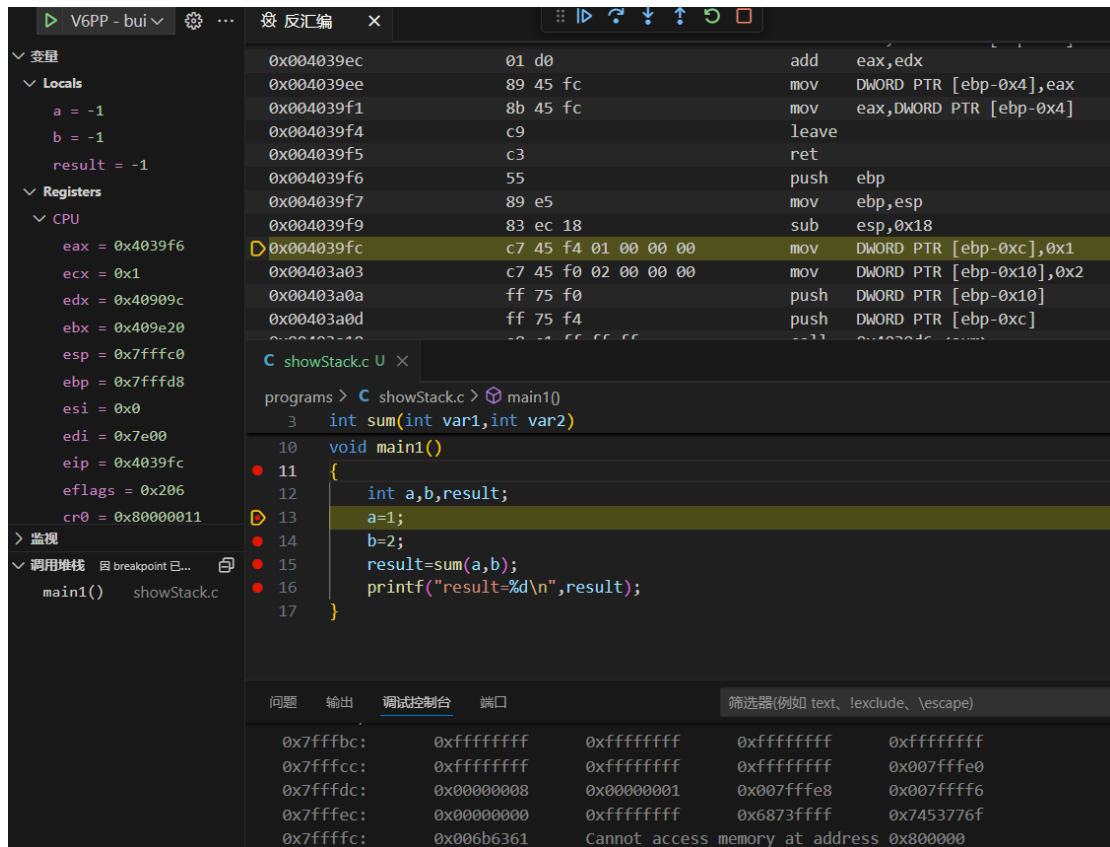


	地址	内容	功能
<u>ebp</u> →	0x7fffd8	0x007ffe0	记录前一帧的 <u>ebp</u>
	0x7fffdc	0x00000008	main1()的返回地址

main1()的用户栈帧

#### 4.3.4 汇编指令：sub esp,0x18

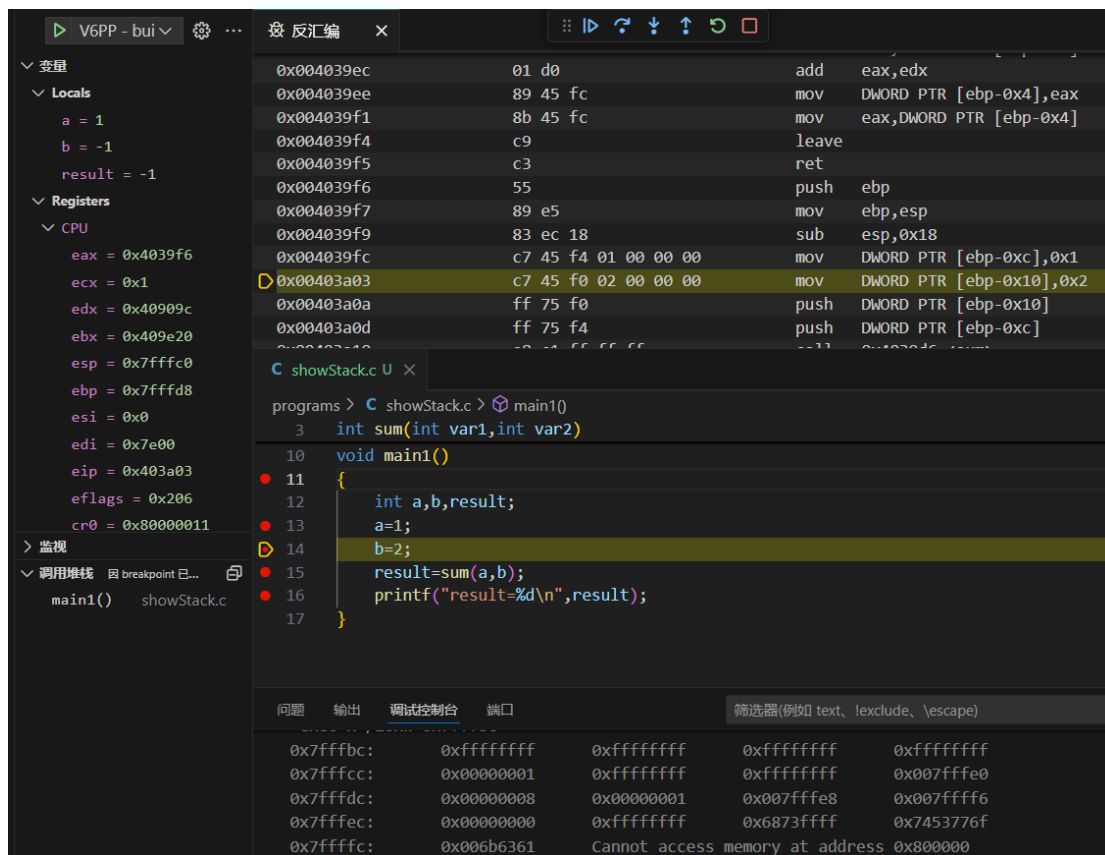
指令功能：这一步让 esp 上移 6 个字，为 main1()提供局部变量的位置



地址	内容	功能
0x7fffc0	0xffffffff	main1()的局部变量的预留空间
0x7fffc4	0xffffffff	main1()的局部变量的预留空间
0x7fffc8	0xffffffff	main1()的局部变量的预留空间
0x7ffcc	0xffffffff	main1()的局部变量的预留空间
0x7ffcd0	0xffffffff	main1()的局部变量的预留空间
0x7ffcd4	0xffffffff	main1()的局部变量的预留空间
ebp → 0x7ffcd8	0x007ffe0	记录前一帧的ebp
0x7ffcdc	0x00000008	main1()的返回地址

main1()的用户栈帧

4.3.5 汇编指令: `mov DWORD PTR [ebp-0xc], 0x1`  
 指令功能: 为 `main1()` 的局部变量 `a` 赋值

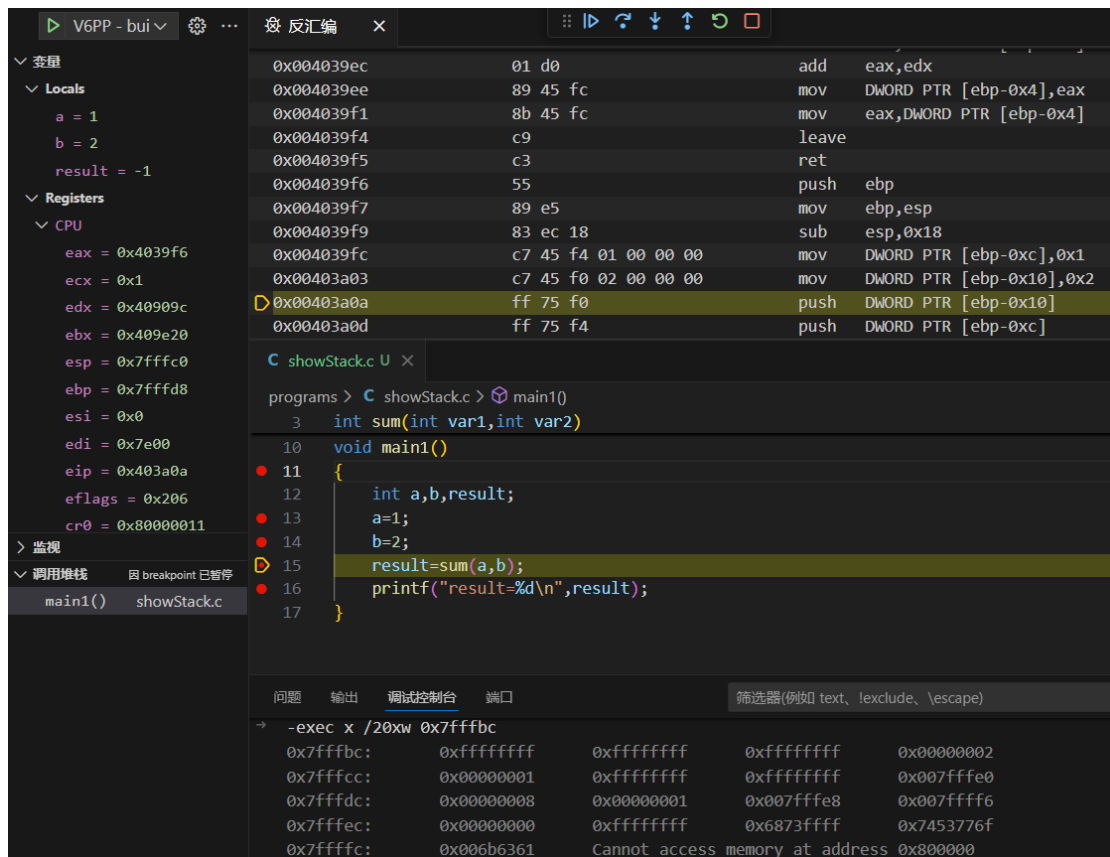


地址	内容	功能
0x7fffc0	0xffffffff	main1()的局部变量的预留空间
0x7fffc4	0xffffffff	main1()的局部变量的预留空间
0x7fffc8	0xffffffff	main1()的局部变量的预留空间
0x7ffcc	0x00000001	main1()的局部变量a
0x7fffd0	0xffffffff	main1()的局部变量的预留空间
0x7fffd4	0xffffffff	main1()的局部变量的预留空间
ebp → 0x7fffd8	0x007fffe0	记录前一帧的ebp
0x7fffdc	0x00000008	main1()的返回地址

main1()的用户栈帧

#### 4.3.6 汇编指令：mov DWORD PTR [ebp-0x10], 0x2

指令功能：为 main1()的局部变量 b 赋值



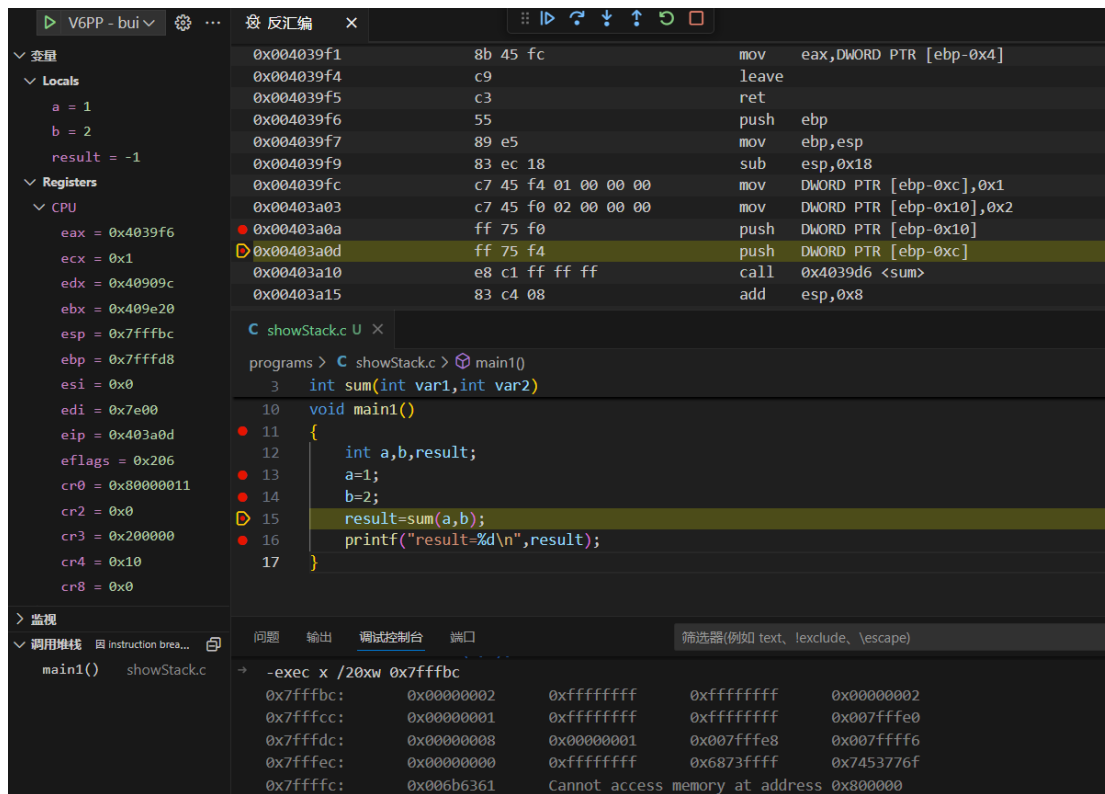
地址	内容	功能
0x7fffc0	0xffffffff	main1()的局部变量的预留空间
0x7fffc4	0xffffffff	main1()的局部变量的预留空间
0x7fffc8	0x00000002	main1()的局部变量b
0x7ffcc	0x00000001	main1()的局部变量a
0x7ffcd0	0xffffffff	main1()的局部变量的预留空间
0x7ffcd4	0xffffffff	main1()的局部变量的预留空间
<u>ebp</u> → 0x7ffcd8	0x007fffe0	记录前一帧的 <u>ebp</u>
0x7ffcdc	0x00000008	main1()的返回地址

main1()的用户栈帧

到这里，main1()的栈帧就创建完毕了，后面是对 sum 函数的栈帧准备参数

#### 4.3.7 指令：push DWORD PTR [ebp-0x10]

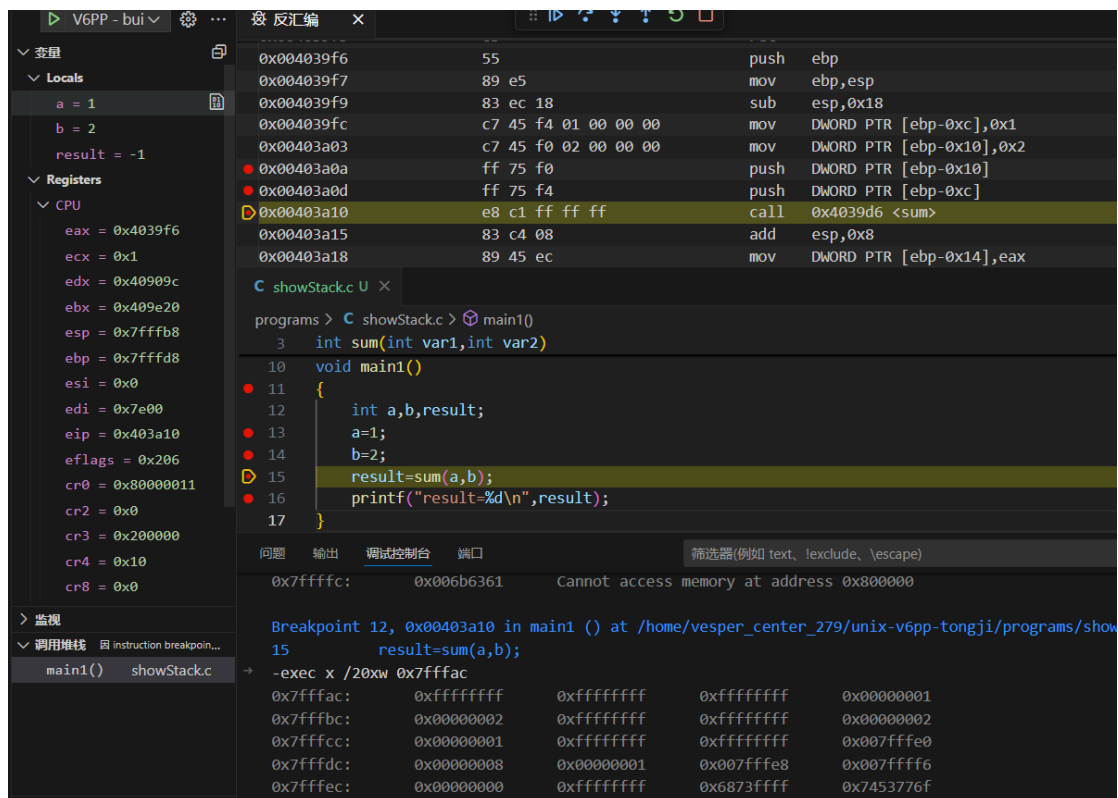
指令功能：把 sum 的参数 var2 的值（2）压栈，即将参数 b 压栈



	地址	内容	功能
sum()的用户栈帧	0x7fffb8	0x00000002	sum()函数的形参var2(=b)
	0x7fffc0	0xffffffff	main1()的局部变量的预留空间
	0x7fffc4	0xffffffff	main1()的局部变量的预留空间
main1()的用户栈帧	0x7fffc8	0x00000002	main1()的局部变量b(=2)
	0x7ffcc	0x00000001	main1()的局部变量a(=1)
	0x7ffcd0	0xffffffff	main1()的局部变量的预留空间
	0x7ffcd4	0xffffffff	main1()的局部变量的预留空间
ebp	0x7ffcd8	0x007fffe0	记录前一帧的ebp
	0x7ffcdc	0x00000008	main1()的返回地址

#### 4.3.8 指令：push DWORD PTR [ebp-0xc]

指令功能：把 sum 的参数 var1 的值（1）压栈，即将参数 a 压栈

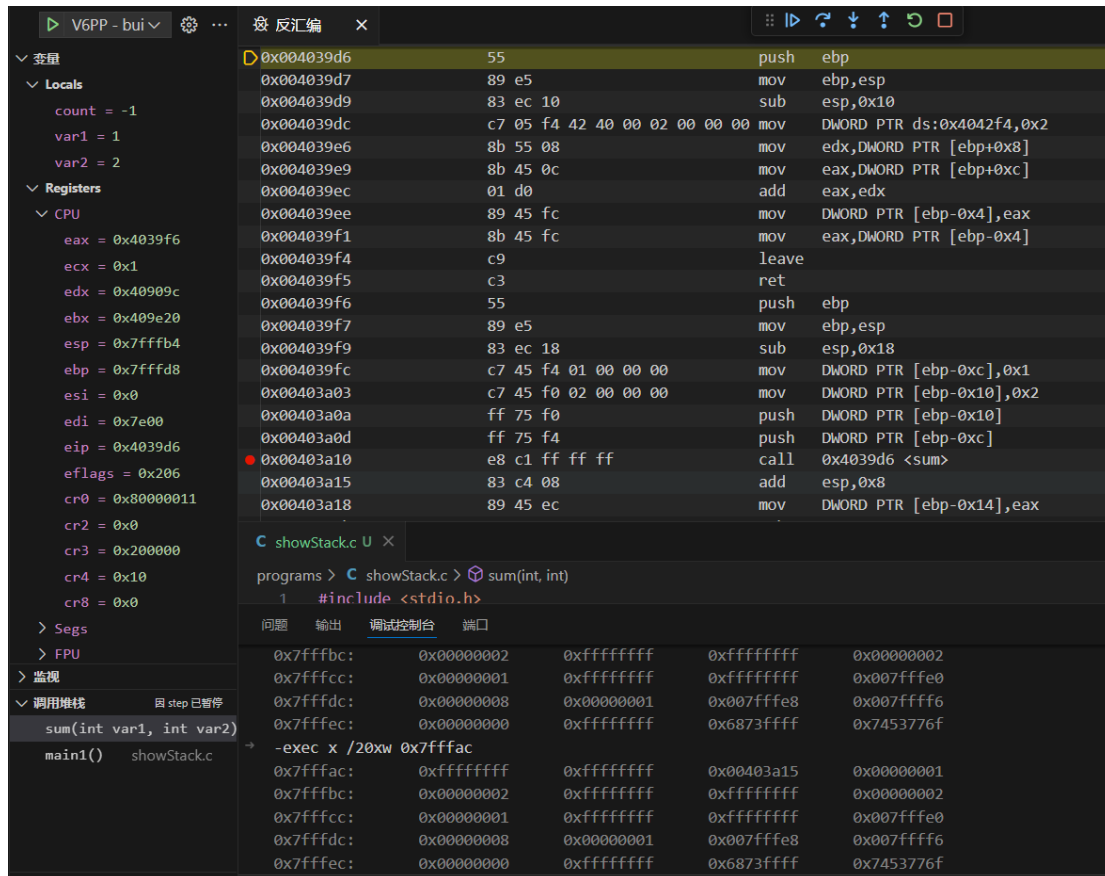


	地址	内容	功能
sum()的用户栈帧	0x7fffb8	0x00000001	sum()函数的形参var1(=a)
	0x7ffbbc	0x00000002	sum()函数的形参var2(=b)
main1()的用户栈帧	0x7fffc0	0xffffffff	main1()的局部变量的预留空间
	0x7fffc4	0xffffffff	main1()的局部变量的预留空间
	0x7fffc8	0x00000002	main1()的局部变量b(=2)
	0x7fffcc	0x00000001	main1()的局部变量a(=1)
	0x7fffd0	0xffffffff	main1()的局部变量的预留空间
	0x7fffd4	0xffffffff	main1()的局部变量的预留空间
	0x7fffd8	0x007fffe0	记录前一帧的ebp
	0x7fffdc	0x00000008	main1()的返回地址

这里补充一个调用 sum 函数的过程：

#### 4.3.9 指令：call 0x4039d6 <sum>

指令功能：调用 sum 函数，将函数的返回地址压入 sum()函数的用户栈



	地址	内容	功能
sum()的用户栈帧	0x7fffb4	0x00403a15	sum()函数的返回地址
	0x7fffb8	0x00000001	sum()函数的形参var1(=a)
	0x7ffbbc	0x00000002	sum()函数的形参var2(=b)
main1()的用户栈帧	0x7fffc0	0xffffffff	main1()的局部变量的预留空间
	0x7fffc4	0xffffffff	main1()的局部变量的预留空间
	0x7fffc8	0x00000002	main1()的局部变量b(=2)
	0x7fffcc	0x00000001	main1()的局部变量a(=1)
	0x7fffd0	0xffffffff	main1()的局部变量的预留空间
	0x7fffd4	0xffffffff	main1()的局部变量的预留空间
	0x7fffd8	0x007fffe0	记录前一帧的ebp
	0x7fffdc	0x00000008	main1()的返回地址

## 4.4 观察 sum 函数的变化

### 4.4.1 汇编代码的解释

```

=====
push ebp //前一栈帧的 ebp 存入当前栈
mov ebp, esp //修改 ebp 指向当前栈帧
sub esp, 0x10 //为局部变量预留一定的空间

```



```

mov DWORD PTR ds:0x4042f4, 0x2 //修改全局变量 version 的值为 2
mov edx, DWORD PTR [ebp+0x8] //这里让 edx 寄存器暂存参数 var1 的值(1)
mov eax, DWORD PTR [ebp+0xc] //这里让 eax 寄存器暂存参数 var2 的值(2)
add eax, edx //这里做了一次加法，并将结果保存到 eax 中
mov DWORD PTR [ebp-0x4], eax //将计算结果保存在局部变量 count 中
mov eax, DWORD PTR [ebp-0x4] //将 sum 函数的返回值保存在 eax 中
leave //删除 sum 用户栈的局部变量区域，并取出上一帧的 ebp
ret //把栈顶的返回地址取出赋值给 eip，实现函数调用的返回

```

#### 4.4.2 具体汇编代码执行的过程

(1) 指令：push ebp

指令功能：将前一栈帧的 ebp 存入当前栈

The screenshot shows a debugger interface with the following components:

- Registers Panel (Left):** Displays the state of various registers.
  - Locals:** count = -1, var1 = 1, var2 = 2.
  - Registers:**
    - eax = 0x4039f6
    - ecx = 0x1
    - edx = 0x40909c
    - ebx = 0x409e20
    - esp = 0x7ffffb0
    - ebp = 0x7ffffd8
    - esi = 0x0
    - edi = 0x7e00
    - eip = 0x4039d7
    - eflags = 0x206
    - cr0 = 0x80000011
    - cr2 = 0x0
    - cr3 = 0x200000
    - cr4 = 0x10
    - cr8 = 0x0
- Assembly Panel (Top Right):** Shows the assembly code being executed.
  - Address 0x004039d3: pop edi
  - Address 0x004039d4: pop ebp
  - Address 0x004039d5: ret
  - Address 0x004039d6: push ebp
  - Address 0x004039d7: mov ebp, esp (Current instruction)
  - Address 0x004039d9: sub esp, 0x10
  - Address 0x004039dc: mov DWORD PTR ds:0x4042f4, 0x2
  - Address 0x004039e6: mov edx, DWORD PTR [ebp+0x8]
  - Address 0x004039e9: mov eax, DWORD PTR [ebp+0xc]
  - Address 0x004039ec: add eax, edx
  - Address 0x004039ee: mov DWORD PTR [ebp-0x4], eax
  - Address 0x004039f1: mov eax, DWORD PTR [ebp-0x4]
  - Address 0x004039f4: leave
  - Address 0x004039f5: ret
- Source Code Panel (Bottom Right):** Shows the C code for the program.
  - File: showStack.c
  - Line 1: #include <stdio.h>
  - Line 2: int version = 1;
  - Line 3: int sum(int var1, int var2)
  - Line 4: {
  - Line 5: int count;
  - Line 6: version=2;
  - Line 7: count=var1+var2;
  - Line 8: return (count);
  - Line 9: }
  - Line 10: void main1()
  - Line 11: {
  - Line 12: int a.b.result;
- Stack Panel (Bottom Left):** Shows the stack memory layout.
  - Address 0x7ffff9c: -exec x /20xw 0x7ffff9c
  - Address 0x7ffff9c: 0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff, 0xffffffff
  - Address 0x7ffffac: 0xffffffff, 0x007ffffd8, 0x00403a15, 0x00000001, 0x00000002
  - Address 0x7ffffbc: 0x00000002, 0xffffffff, 0xffffffff, 0xffffffff, 0x00000002
  - Address 0x7ffffcc: 0x00000001, 0xffffffff, 0xffffffff, 0xffffffff, 0x007fffe0
  - Address 0x7ffffdc: 0x00000008, 0x00000001, 0x007fffe8, 0x007ffff6, 0x007ffff6

	地址	内容	功能
sum()的用户栈帧	0x7fffb0	0x007fffd8	记录前一帧的ebp
	0x7fffb4	0x00403a15	sum()函数的返回地址
	0x7fffb8	0x00000001	sum()函数的形参var1(=a)
	0x7fffb4	0x00000002	sum()函数的形参var2(=b)
main1()的用户栈帧	0x7ffc0	0xffffffff	main1()的局部变量的预留空间
	0x7ffc4	0xffffffff	main1()的局部变量的预留空间
	0x7ffc8	0x00000002	main1()的局部变量b(=2)
	0x7ffcc	0x00000001	main1()的局部变量a(=1)
	0x7ffd0	0xffffffff	main1()的局部变量的预留空间
	0x7ffd4	0xffffffff	main1()的局部变量的预留空间
	0x7ffdc	0x007fffe0	记录前一帧的ebp
	0x7ffdc	0x00000008	main1()的返回地址

(2) 指令: mov ebp, esp  
指令功能: 修改 ebp 指向当前栈帧

V6PP - bui

变量

Locals

Registers

CPU

Segs

FPU

监视

调用堆栈

反汇编

0x004039d3

5f

pop

edi

0x004039d4

5d

pop

ebp

0x004039d5

c3

ret

0x004039d6

55

push

ebp

0x004039d7

89 e5

mov

ebp,esp

0x004039d9

83 ec 10

sub

esp,0x10

0x004039dc

c7 05 f4 42 40 00 02 00 00 00

mov

DWORD PTR ds:0x4042f4,0x2

0x004039e6

8b 55 08

mov

edx,DWORD PTR [ebp+0x8]

0x004039e9

8b 45 0c

mov

eax,DWORD PTR [ebp+0xc]

0x004039ec

01 d0

add

eax,edx

0x004039ee

89 45 fc

mov

DWORD PTR [ebp-0x4],eax

0x004039f1

8b 45 fc

mov

eax,DWORD PTR [ebp-0x4]

0x004039f4

c9

leave

0x004039f5

c3

ret

C showStack.c U

programs > C showStack.c > sum(int, int)

1 #include <stdio.h>

2 int version = 1;

3 int sum(int var1,int var2)

4 {

5 int count;

6 version=2;

7 count=var1+var2;

8 return (count);

9 }

10 void main1()

11 {

12 int a.b.result;

问题

输出

调试控制台

端口

sum(int var1, int var2)

main1() showStack.c

-exec x /20xw 0x7fff9c

0x7fff9c: 0xffffffff 0xffffffff 0xffffffff 0xffffffff

0x7fffac: 0xffffffff 0x007fffd8 0x00403a15 0x00000001

0x7fffb4: 0x00000002 0xffffffff 0xffffffff 0x00000002

0x7fffb8: 0x00000001 0xffffffff 0xffffffff 0x007fffe0

0x7ffdc: 0x00000008 0x00000001 0x007fffe8 0x007ffff6

	地址	内容	功能
sum()的用户栈帧	0x7fffb0	0x007fffd8	记录前一帧的ebp
	0x7fffb4	0x00403a15	sum()函数的返回地址
	0x7fffb8	0x00000001	sum()函数的形参var1(=a)
	0x7ffbbc	0x00000002	sum()函数的形参var2(=b)
main1()的用户栈帧	0x7fffc0	0xffffffff	main1()的局部变量的预留空间
	0x7fffc4	0xffffffff	main1()的局部变量的预留空间
	0x7fffc8	0x00000002	main1()的局部变量b(=2)
	0x7fffcc	0x00000001	main1()的局部变量a(=1)
	0x7fffd0	0xffffffff	main1()的局部变量的预留空间
	0x7fffd4	0xffffffff	main1()的局部变量的预留空间
	0x7fffd8	0x007fffe0	记录前一帧的ebp
	0x7fffdc	0x00000008	main1()的返回地址

(3) 指令: sub esp, 0x10

指令功能: 为 sum 函数的局部变量预留一定的空间, 这里是预留了四个字的空间

The screenshot shows a debugger interface with the following components:

- Assembly Window:** Displays assembly instructions for the `sum` and `main1` functions. The `sum` function instructions include `pop edi`, `pop ebp`, `ret`, `push ebp`, `mov ebp, esp`, `sub esp, 0x10`, `mov DWORD PTR [ebp+0xc], 0x2`, `mov edx, DWORD PTR [ebp+0x8]`, `mov eax, DWORD PTR [ebp+0xc]`, `add eax, edx`, `mov DWORD PTR [ebp-0x4], eax`, `mov eax, DWORD PTR [ebp-0x4]`, `leave`, and `ret`. The `main1` function instructions include `ret`.
- Stack Window:** Shows the stack frame for `sum(int var1, int var2)`. The stack pointer `esp` is at `0x7fffa0`. The `ebp` register is at `0x7fffb0`. The `edi` register is at `0x7e00`. The `eip` register is at `0x4039dc`. The `eflags` register is at `0x206`. The `cr0` register is at `0x8000011`. The `cr2` register is at `0x0`. The `cr3` register is at `0x200000`. The `cr4` register is at `0x10`. The `cr8` register is at `0x0`.
- Command Window:** Shows the execution of the `sub esp, 0x10` instruction. The command is `-exec x /20xw 0x7fff9c`. The output shows the stack pointer `esp` being updated to `0x7fff9c`.

	地址	内容	功能
sum()的用户栈帧	0x7ffa0	0xffffffff	sum()的局部变量的预留空间
	0x7ffa4	0xffffffff	sum()的局部变量的预留空间
	0x7ffa8	0xffffffff	sum()的局部变量的预留空间
	0x7ffac	0xffffffff	sum()的局部变量的预留空间
	0x7ffb0	0x007ffd8	记录前一帧的ebp
	0x7ffb4	0x00403a15	sum()函数的返回地址
	0x7ffb8	0x00000001	sum()函数的形参var1(=a)
	0x7ffbc	0x00000002	sum()函数的形参var2(=b)
main1()的用户栈帧	0x7ffc0	0xffffffff	main1()的局部变量的预留空间
	0x7ffc4	0xffffffff	main1()的局部变量的预留空间
	0x7ffc8	0x00000002	main1()的局部变量b(=2)
	0x7ffcc	0x00000001	main1()的局部变量a(=1)
	0x7ffd0	0xffffffff	main1()的局部变量的预留空间
	0x7ffd4	0xffffffff	main1()的局部变量的预留空间
	0x7ffd8	0x007ffe0	记录前一帧的ebp
	0x7ffdc	0x00000008	main1()的返回地址

(4) 指令: mov DWORD PTR ds:0x4042f4, 0x2

指令功能: 这里是将全局变量(version)赋值为 2, 从这里我们也可以知道全局变量的地址: 0x4042f4

The screenshot shows a debugger interface with the following components:

- Assembly Window:** Displays assembly instructions with their addresses and hex values. The instruction `mov edx, DWORD PTR [ebp+0x8]` is highlighted.
- Registers Window:** Shows the current state of CPU registers. `eax = 0x4039f6`, `ecx = 0x1`, `edx = 0x40909c`, `ebx = 0x409e20`, `esp = 0x7ffa0`, `ebp = 0x7ffb0`, `esi = 0x0`, `edi = 0x7e00`, `eip = 0x4039e6`, `eflags = 0x206`, `cr0 = 0x80000011`, `cr2 = 0x0`, `cr3 = 0x200000`.
- Memory Window:** Shows the memory address `0x4042f4` containing the value `0x00000002`.
- Source Code Window:** Shows the C code for the `sum` and `main1` functions. The line `count=var1+var2;` is highlighted.
- Command Window:** Shows the command `-exec x /20xw 0x4042f4` and the resulting memory dump.

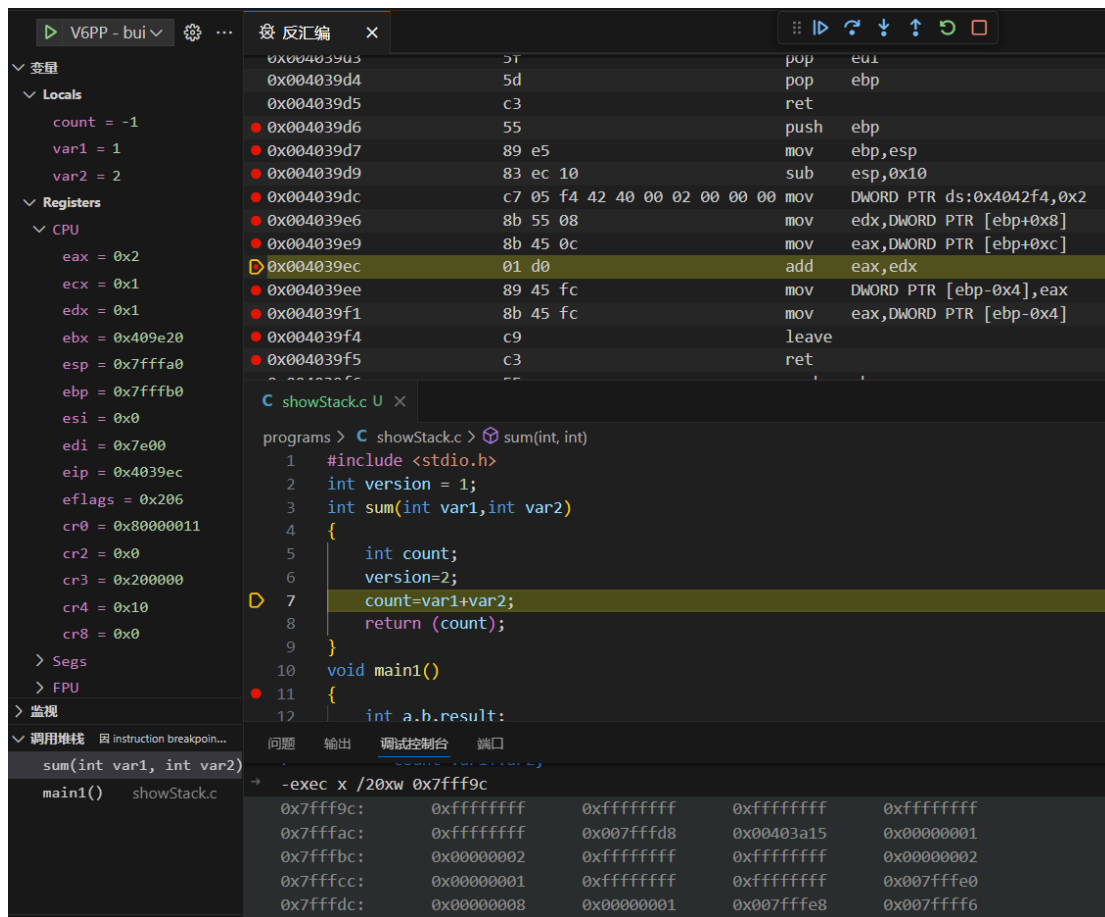
地址	内容	功能
0x4042f4	0x00000002	全局变量version的值

(5) 指令: mov edx, DWORD PTR [ebp+0x8]  
指令功能: 这里让 edx 寄存器暂存参数 var1 的值(1)

Debugger screenshot showing assembly code and stack memory. The assembly window displays instructions from 0x004039d3 to 0x004039f5. The instruction at 0x004039e9 is highlighted: `mov eax, DWORD PTR [ebp+0xc]`. The stack window shows memory addresses from 0x7fff9c to 0x7fffdc. The instruction at 0x004039e9 is highlighted in the assembly window.

寄存器	内容	功能
edx	0x00000001	暂存参数var1的值(1)

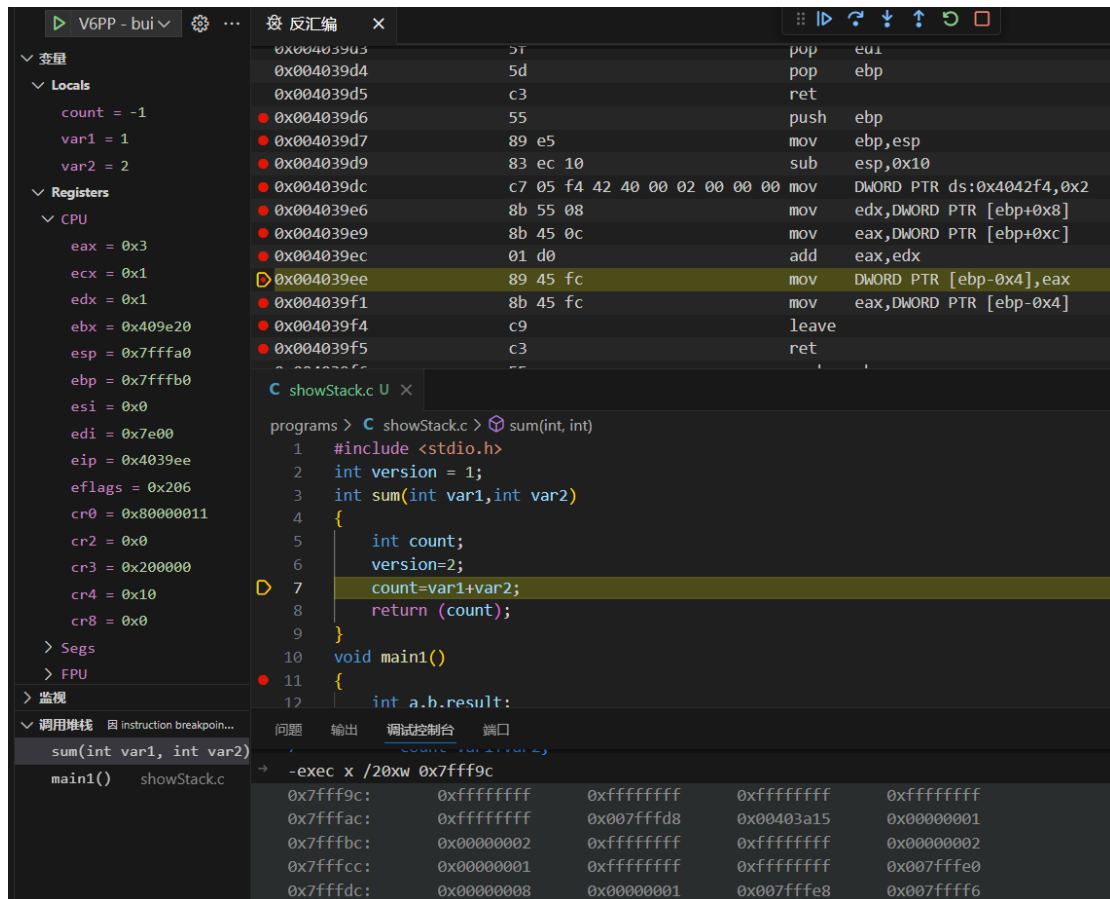
(6) 指令: mov eax, DWORD PTR [ebp+0xc]  
指令功能: 这里让 eax 寄存器暂存参数 var2 的值(2)



寄存器	内容	功能
<u>eax</u>	0x00000002	暂存参数var2的值(2)

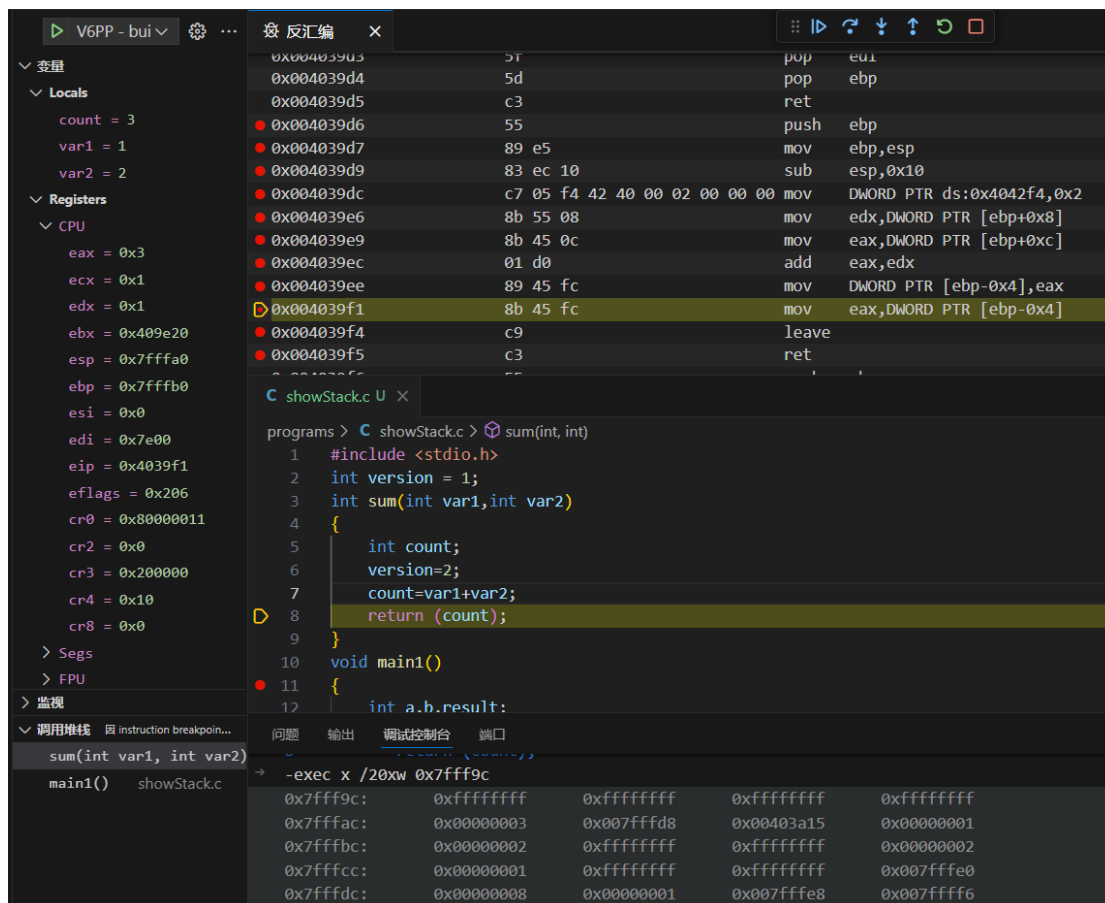
(7) 指令: add eax, edx

指令功能: 这一步实施加法, 让寄存器 edx 和 eax 的值相加, 并保存结果到 eax 中。



寄存器	内容	功能
<u>eax</u>	0x00000003	暂存运算结果的值(3)

(8) 指令: `mov DWORD PTR [ebp-0x4], eax`  
 指令功能: 将计算结果保存在局部变量 `count` 中

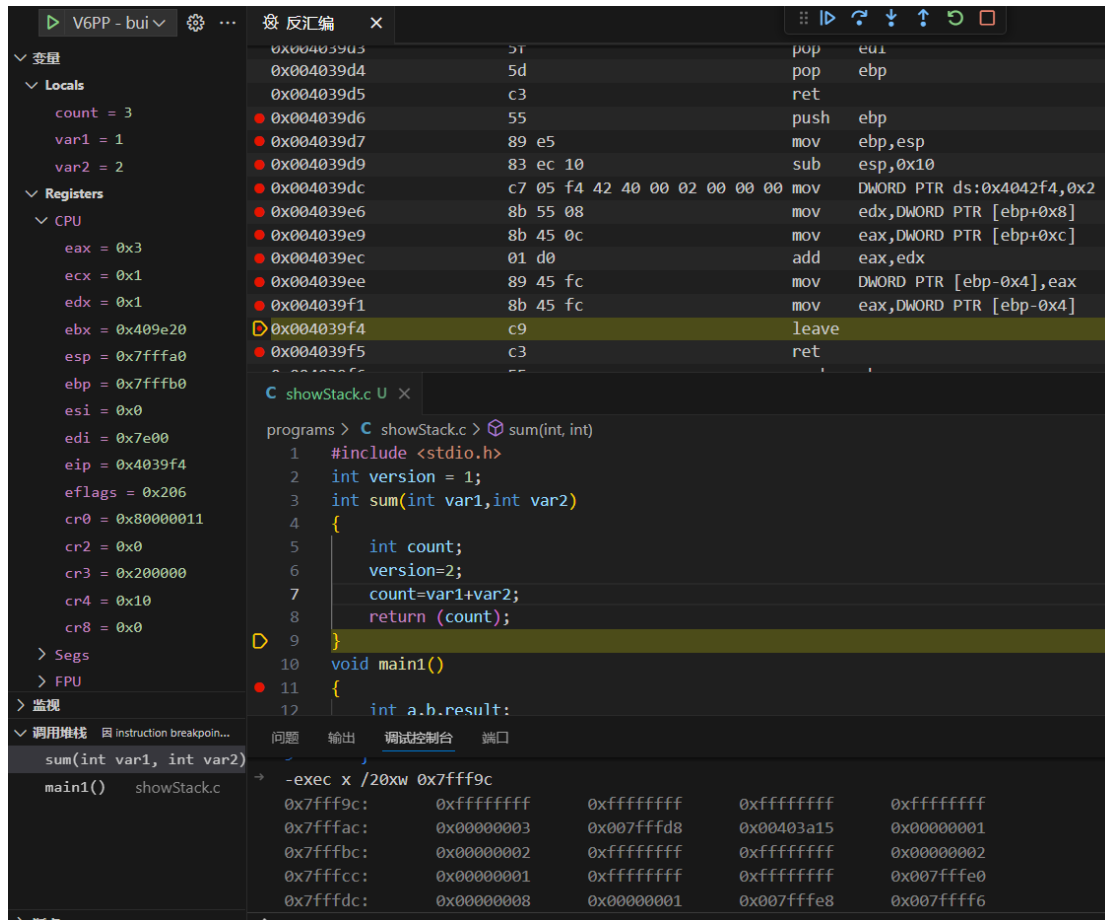


这里是将寄存器 `eax` 的值（3）保存到目前 `ebp` 所在位置的上一个字空间

	地址	内容	功能
sum()的用户栈帧 ebp	0x7ffa0	0xffffffff	sum()的局部变量的预留空间
	0x7ffa4	0xffffffff	sum()的局部变量的预留空间
	0x7ffa8	0xffffffff	sum()的局部变量的预留空间
	0x7ffac	0x00000003	sum()的局部变量count(=3)
	0x7ffb0	0x007fffd8	记录前一帧的ebp
	0x7ffb4	0x00403a15	sum()函数的返回地址
	0x7ffb8	0x00000001	sum()函数的形参var1(=a)
	0x7ffbc	0x00000002	sum()函数的形参var2(=b)
main1()的用户栈帧	0x7ffc0	0xffffffff	main1()的局部变量的预留空间
	0x7ffc4	0xffffffff	main1()的局部变量的预留空间
	0x7ffc8	0x00000002	main1()的局部变量b(=2)
	0x7ffcc	0x00000001	main1()的局部变量a(=1)
	0x7ffd0	0xffffffff	main1()的局部变量的预留空间
	0x7ffd4	0xffffffff	main1()的局部变量的预留空间
	0x7ffd8	0x007fffe0	记录前一帧的ebp
	0x7ffdc	0x00000008	main1()的返回地址

（10）指令：`mov eax, DWORD PTR [ebp-0x4]`  
指令功能：将 `sum` 函数的返回值保存在 `eax` 中





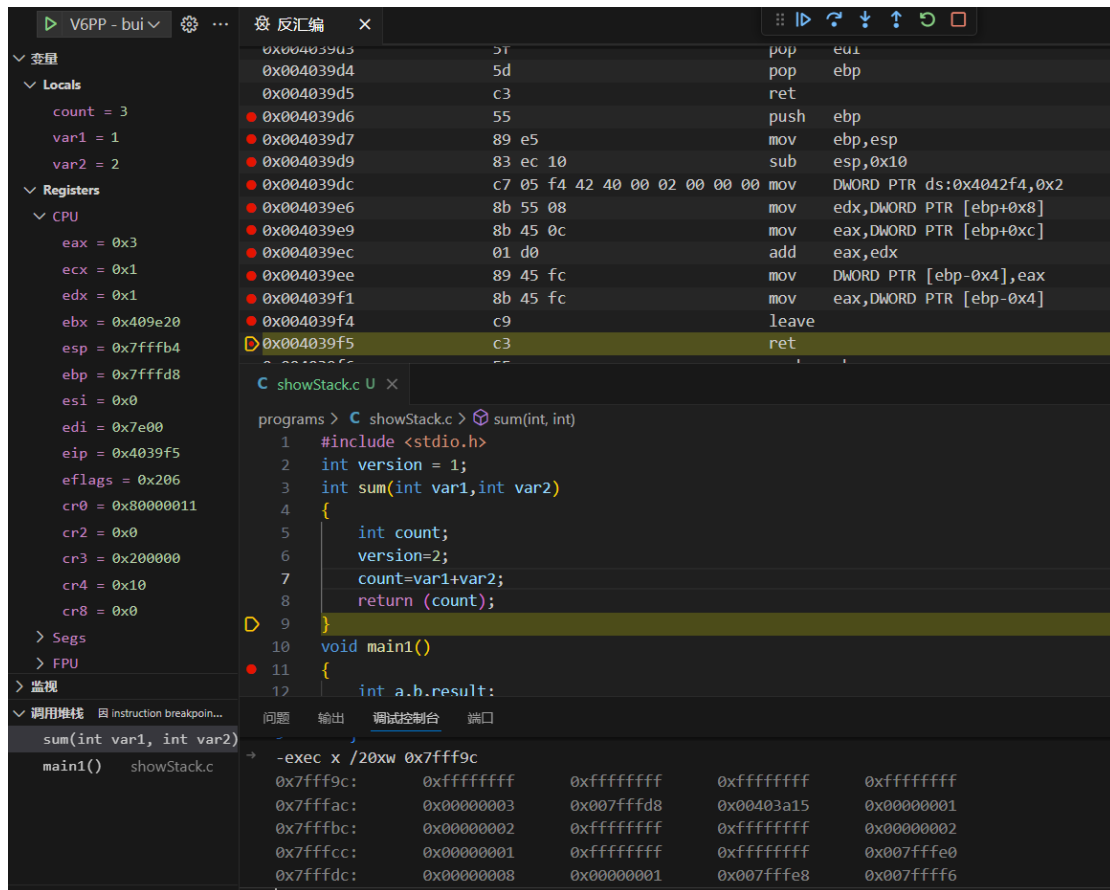
这一步是把函数最终的返回值 3 赋值给寄存器 `eax`

	地址	内容	功能
sum()的用户栈帧	0x7fffb0	0x007fffd8	记录前一帧的 <code>ebp</code>
	0x7fffb4	0x00403a15	sum()函数的返回地址
	0x7fffb8	0x00000001	sum()函数的形参var1(=a)
	0x7ffbbc	0x00000002	sum()函数的形参var2(=b)
main1()的用户栈帧	0x7fffc0	0xffffffff	main1()的局部变量的预留空间
	0x7fffc4	0xffffffff	main1()的局部变量的预留空间
	0x7fffc8	0x00000002	main1()的局部变量b(=2)
	0x7fffcc	0x00000001	main1()的局部变量a(=1)
	0x7fffd0	0xffffffff	main1()的局部变量的预留空间
	0x7fffd4	0xffffffff	main1()的局部变量的预留空间
	0x7fffd8	0x007fffe0	记录前一帧的 <code>ebp</code>
	0x7fffdc	0x00000008	main1()的返回地址

(11) 指令: `leave`

指令功能: 这一步主要分为两个步骤: `mov esp, ebp` 以及 `pop ebp`

首先, `esp=ebp=0x7fffb0`, 然后上一帧的 `ebp` (`0x007fffd8`) 出栈, 这里使得 `ebp=0x007fffd8`、`esp=0x7fffb0+0x4=0x7fffb4`



	地址	内容	功能
sum()的用户栈帧	0x7fffb4	0x00403a15	sum()函数的返回地址
	0x7fffb8	0x00000001	sum()函数的形参var1(=a)
	0x7fffb4	0x00000002	sum()函数的形参var2(=b)
main1()的用户栈帧	0x7fffc0	0xffffffff	main1()的局部变量的预留空间
	0x7fffc4	0xffffffff	main1()的局部变量的预留空间
	0x7fffc8	0x00000002	main1()的局部变量b(=2)
	0x7fffcc	0x00000001	main1()的局部变量a(=1)
	0x7fffd0	0xffffffff	main1()的局部变量的预留空间
	0x7fffd4	0xffffffff	main1()的局部变量的预留空间
	0x7fffd8	0x007fffe0	记录前一帧的ebp
	0x7fffdc	0x00000008	main1()的返回地址

(12) 指令: ret

指令功能: 这一步是把栈顶的返回地址取出赋值给 eip, 使得 eip=0x00403a15

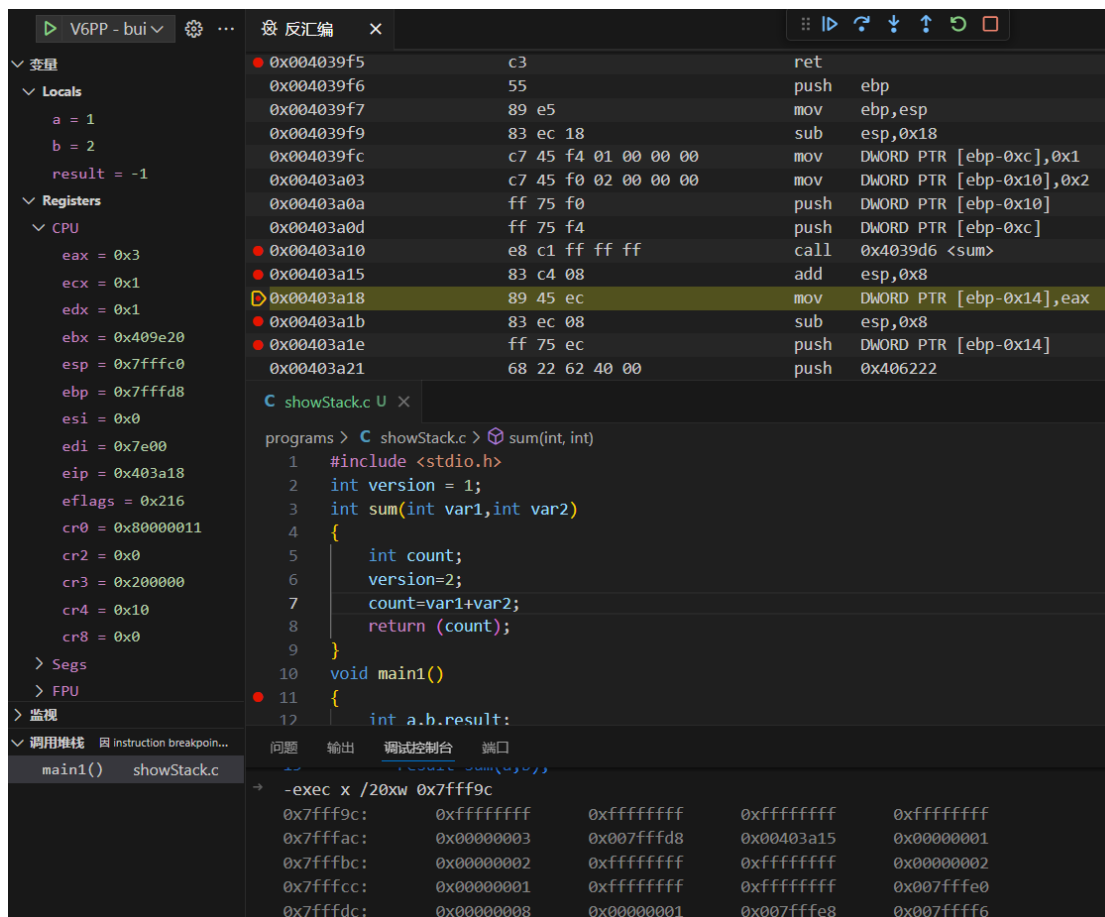
The screenshot shows a debugger interface with the following components:

- Registers:**
  - `eax = 0x3`
  - `ecx = 0x1`
  - `edx = 0x1`
  - `ebx = 0x409e20`
  - `esp = 0x7fffb8`
  - `ebp = 0x7fffd8`
  - `esi = 0x0`
  - `edi = 0x7e00`
  - `eip = 0x403a15`
  - `eflags = 0x206`
  - `cr0 = 0x80000011`
  - `cr2 = 0x0`
  - `cr3 = 0x200000`
  - `cr4 = 0x10`
  - `cr8 = 0x0`
- Assembly Window:**
  - Address `0x004039ec`: `add eax,edx`
  - Address `0x004039ee`: `mov DWORD PTR [ebp-0x4],eax`
  - Address `0x004039f1`: `mov eax,DWORD PTR [ebp-0x4]`
  - Address `0x004039f4`: `leave`
  - Address `0x004039f5`: `ret`
  - Address `0x004039f6`: `push ebp`
  - Address `0x004039f7`: `mov ebp,esp`
  - Address `0x004039f9`: `sub esp,0x18`
  - Address `0x004039fc`: `mov DWORD PTR [ebp-0xc],0x1`
  - Address `0x00403a03`: `mov DWORD PTR [ebp-0x10],0x2`
  - Address `0x00403a0a`: `push DWORD PTR [ebp-0x10]`
  - Address `0x00403a0d`: `push DWORD PTR [ebp-0xc]`
  - Address `0x00403a10`: `call 0x4039d6 <sum>`
  - Address `0x00403a15`: `add esp,0x8`
- Stack Window:**
  - Address `0x7fff9c`: `0xffffffff`
  - Address `0x7fffac`: `0x00000003`
  - Address `0x7fffbc`: `0x00000002`
  - Address `0x7fffcc`: `0x00000001`
  - Address `0x7fffdc`: `0x00000008`

	地址	内容	功能
sum()的用户栈帧	0x7fffb8	0x00000001	sum()函数的形参var1(=a)
	0x7fffb8	0x00000002	sum()函数的形参var2(=b)
main1()的用户栈帧	0x7fffc0	0xffffffff	main1()的局部变量的预留空间
	0x7fffc4	0xffffffff	main1()的局部变量的预留空间
	0x7fffc8	0x00000002	main1()的局部变量b(=2)
	0x7fffcc	0x00000001	main1()的局部变量a(=1)
	0x7fffd0	0xffffffff	main1()的局部变量的预留空间
	0x7fffd4	0xffffffff	main1()的局部变量的预留空间
	0x7fffd8	0x007fffe0	记录前一帧的ebp
	0x7fffdc	0x00000008	main1()的返回地址

下面补充递归返回时的一条指令

(13) 指令: `add esp,0x8`  
 指令功能: 这里是删除 `sum` 函数的实参部分



	地址	内容	功能
main1()的用户栈帧	0x7fffc0	0xffffffff	main1()的局部变量的预留空间
	0x7fffc4	0xffffffff	main1()的局部变量的预留空间
	0x7fffc8	0x00000002	main1()的局部变量b(=2)
	0x7fffcc	0x00000001	main1()的局部变量a(=1)
	0x7fffd0	0xffffffff	main1()的局部变量的预留空间
	0x7fffd4	0xffffffff	main1()的局部变量的预留空间
	0x7fffd8	0x007ffe0	记录前一帧的ebp
ebp	0x7fffdc	0x00000008	main1()的返回地址

4.4.3 在 main1 的汇编代码中，从 sum 返回后执行的指令“add esp, 0x8”的目的是什么？

答：这句指令的功能是删除 sum 函数的实参部分（共有两个字单元）

4.5

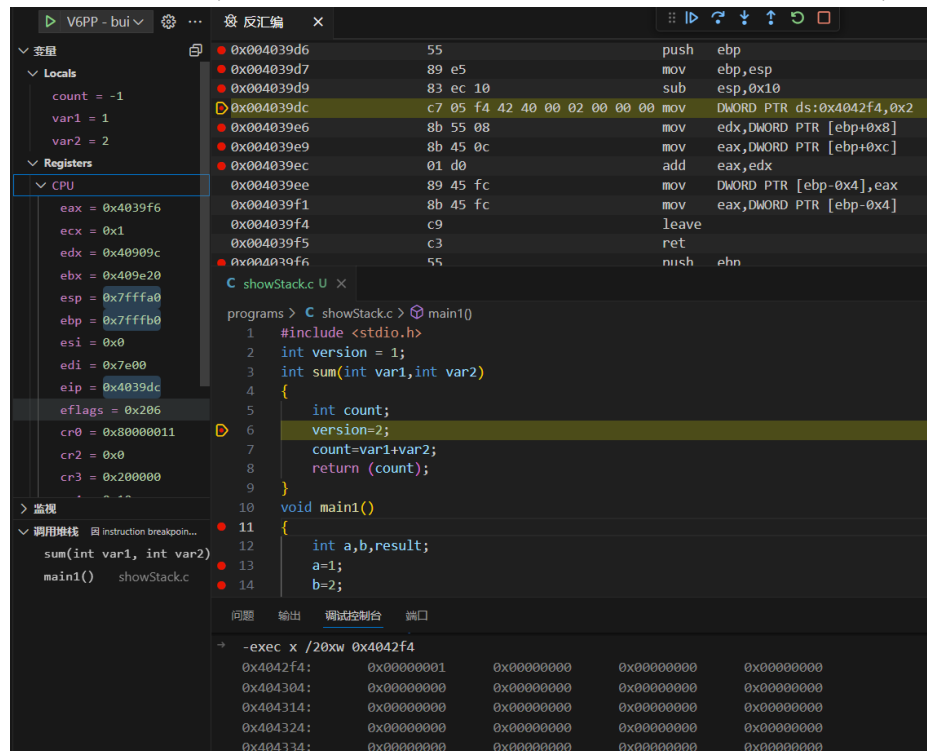
4.5.1 在 sum 的汇编代码中，“mov DWORD PTR ds:0x4042f4, 0x2”的作用是什么？

答：这句代码的作用是将全局变量（version）赋值为 2

4.5.2 结合课堂 学习的知识，尝试解释 ds:0x4042f4 这个地址对应的是什么？为什么？

答：ds:0x4042f4 这个地址对应的是全局变量 version，原因如下：

首先，下图是将要执行 `mov DWORD PTR ds:0x4042f4, 0x2` 的断点调试截图，可以看到 `mov DWORD PTR ds:0x4042f4, 0x2` 所对应的源代码是 `version=2`，而且当前 `0x4042f4` 位置的值为 1（这里的 1 应该是全局变量创建时所赋的值）



其次，下图是执行完 `mov DWORD PTR ds:0x4042f4, 0x2` 的断点调试截图，可以看到这时 `0x4042f4` 的值变成了 2，所以我们可以推断 `0x4042f4` 是全局变量 `version` 的地址，即 `ds:0x4042f4` 这个地址对应的是全局变量 `version`。

