Christopher Lall

CSc 342/343 – Professor Gertner

Take Home Test 2

Due 4/25/22

Start Date & Time: 4/21/2022 10:00AM

End Date & Time: 4/21/2022 10:00PM

I will neither give nor recieve unauthorized assistance on this TEST. I will only use one computing device to perform this TEST, I will not use cell while performing this TEST.

Chris L

Table of Contents

Objective:

The goal of this test is to learn how recursive coding works in disassembled code written in GDB, MIPS, and GCC. In the Ubuntu Virtual Box, we'll use a 64-bit GDB disassembler. We'll use CPP or c code in Visual Studio for the ARM 32 bit. The MIPS simulator will be used for the MIPS analysis. We will be able to understand and explain different registers and how they change throughout recursive function calls using these disassembly windows.

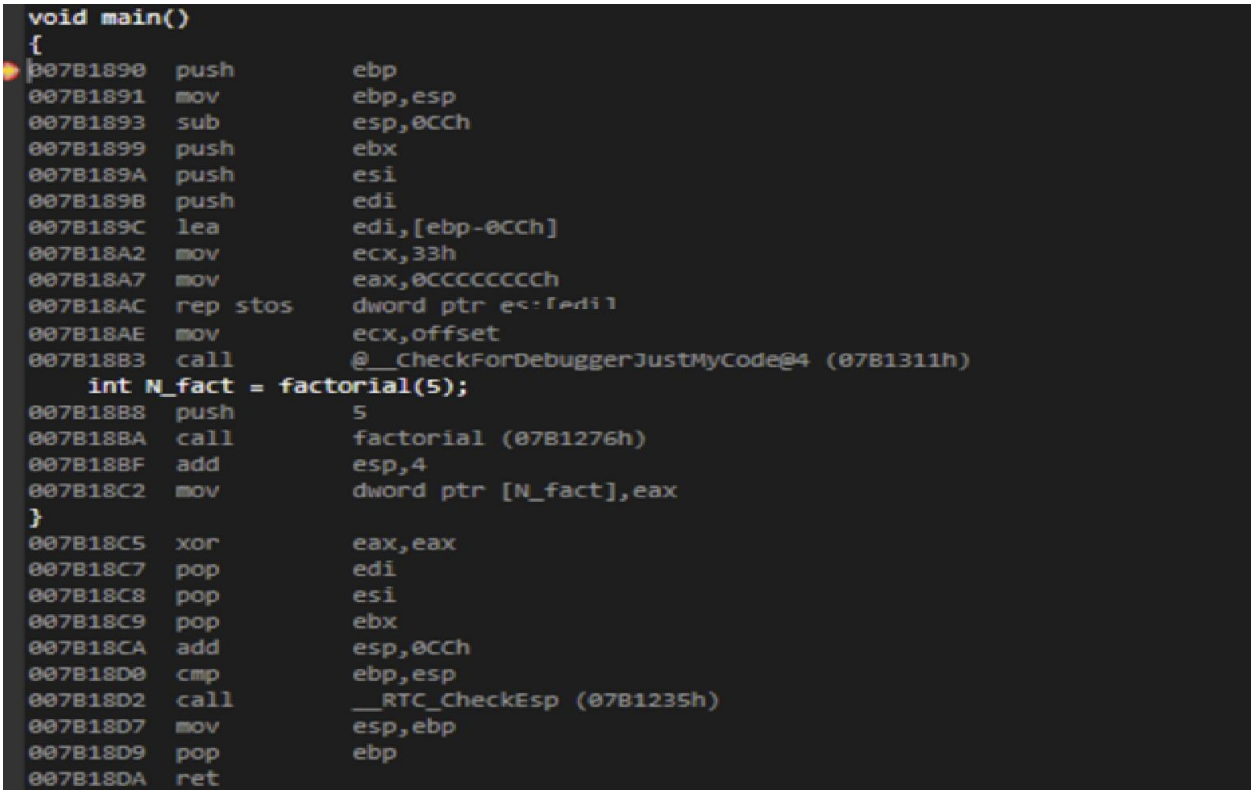
Factorial:

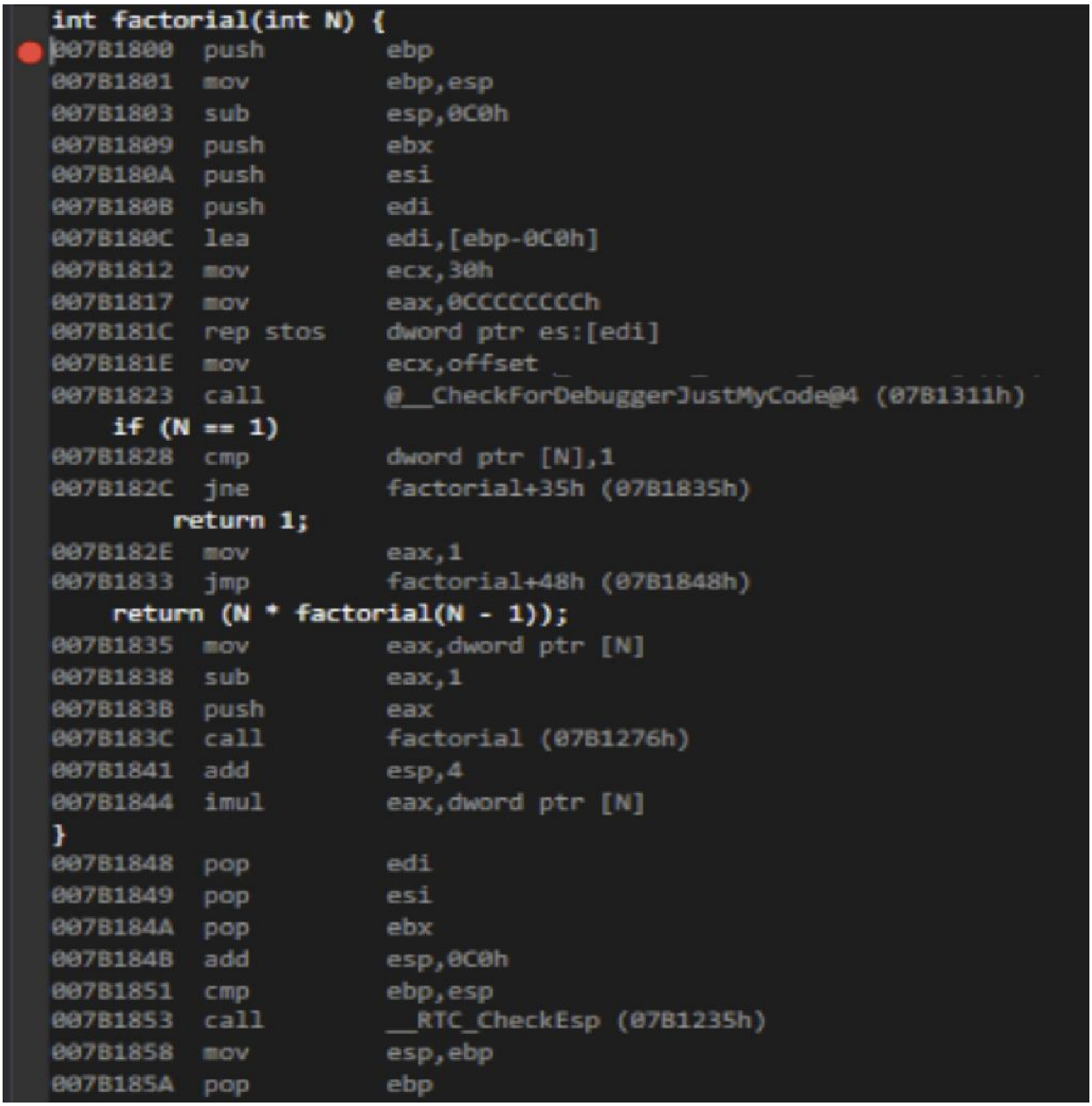
Visual Studio:

```
#include <iostream>

int factorial(int N) {
    if (N == 1)
        return 1;
    return (N * factorial(N - 1));
}

void main() {
    int N_fact = factorial(5);
}
```

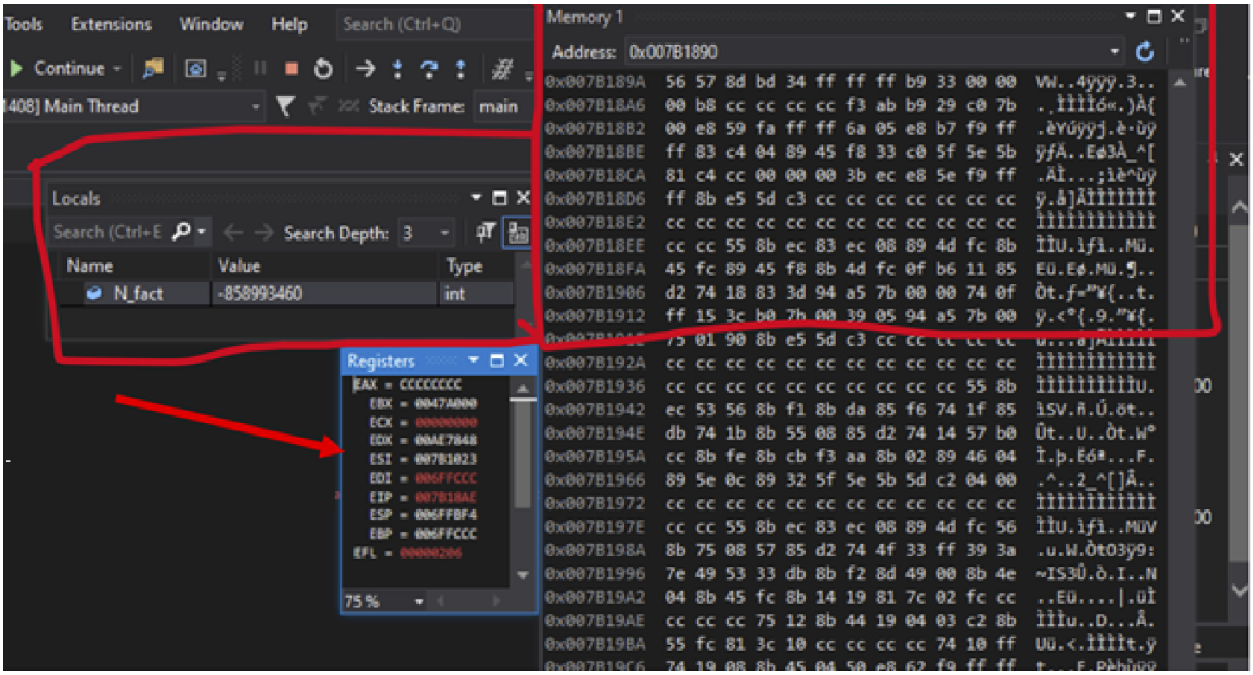
Figure 1. factorial code

Analysis:

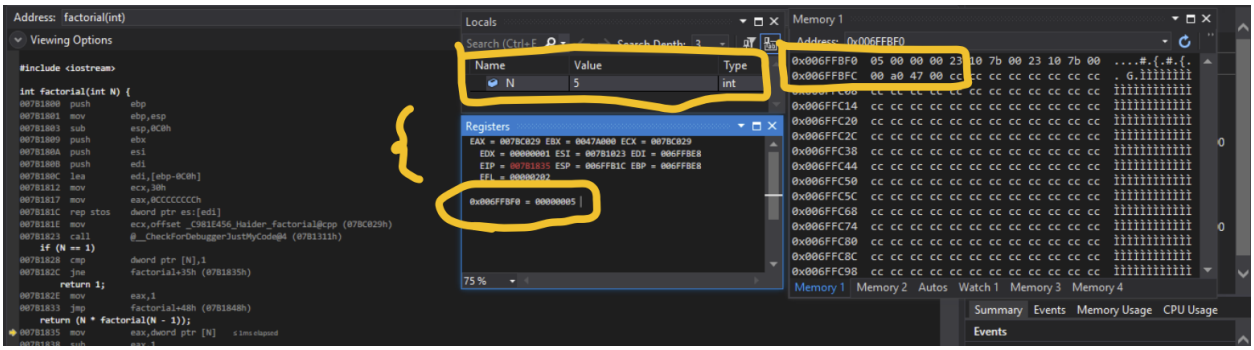

void main()
{
007B1890 push ebp
007B1891 mov ebp,esp
007B1893 sub esp,0CCh
007B1899 push ebx
007B189A push esi
007B189B push edi
007B189C lea edi,[ebp-0CCh]
007B18A2 mov ecx,33h
007B18A7 mov eax,0CCCCCCCCh
007B18AC rep stos dword ptr es:[edi]
007B18AE mov ecx,offset
007B18B3 call @__CheckForDebuggerJustMyCode@4 (07B1311h)
int N_fact = factorial(5);
007B18B8 push 5
007B18BA call factorial (07B1276h)
007B18BF add esp,4
007B18C2 mov dword ptr [N_fact],eax
}
007B18C5 xor eax,eax
007B18C7 pop edi
007B18C8 pop esi
007B18C9 pop ebx
007B18CA add esp,0CCh
007B18D0 cmp ebp,esp
007B18D2 call __RTC_CheckEsp (07B1235h)
007B18D7 mov esp,ebp
007B18D9 pop ebp
007B18DA ret

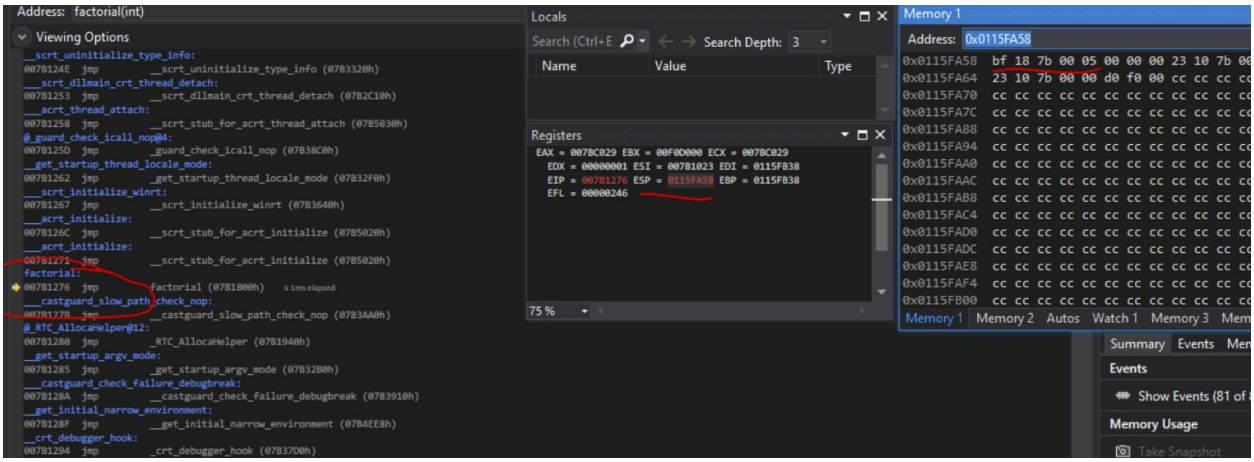


The above pictures is our disassembly window for the code. This shows the disassemble code for both the main function and the factorial function. Our value for the factorial function is 5.

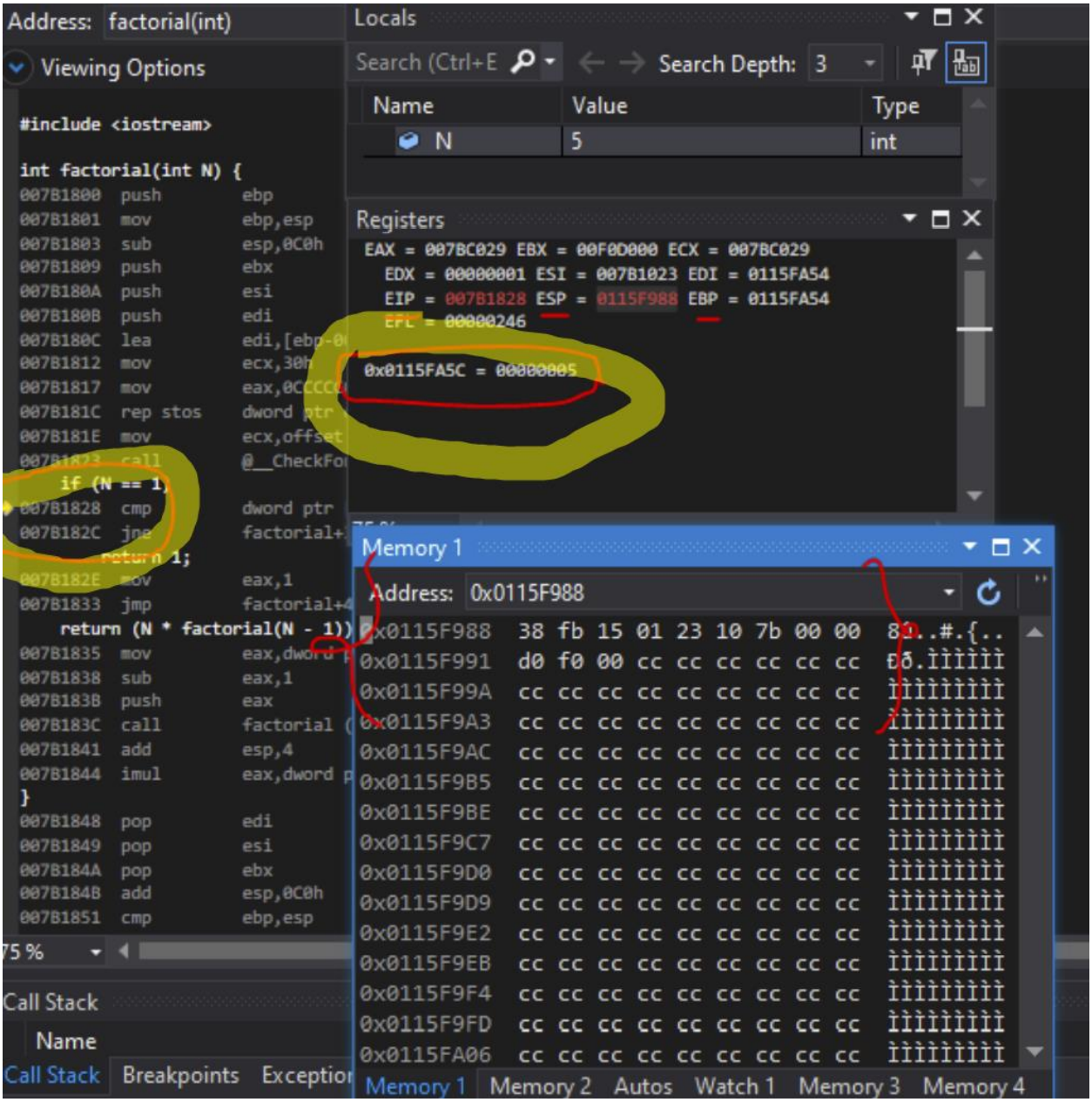


From this screen you can see that void main has been initialized for us. The ESP which is the stack pointer is 006FFbF4. The base pointer which is EBP is 006FFCCC. The next instruction is stored in EIP which is 007B18AE, this is our return address for now.

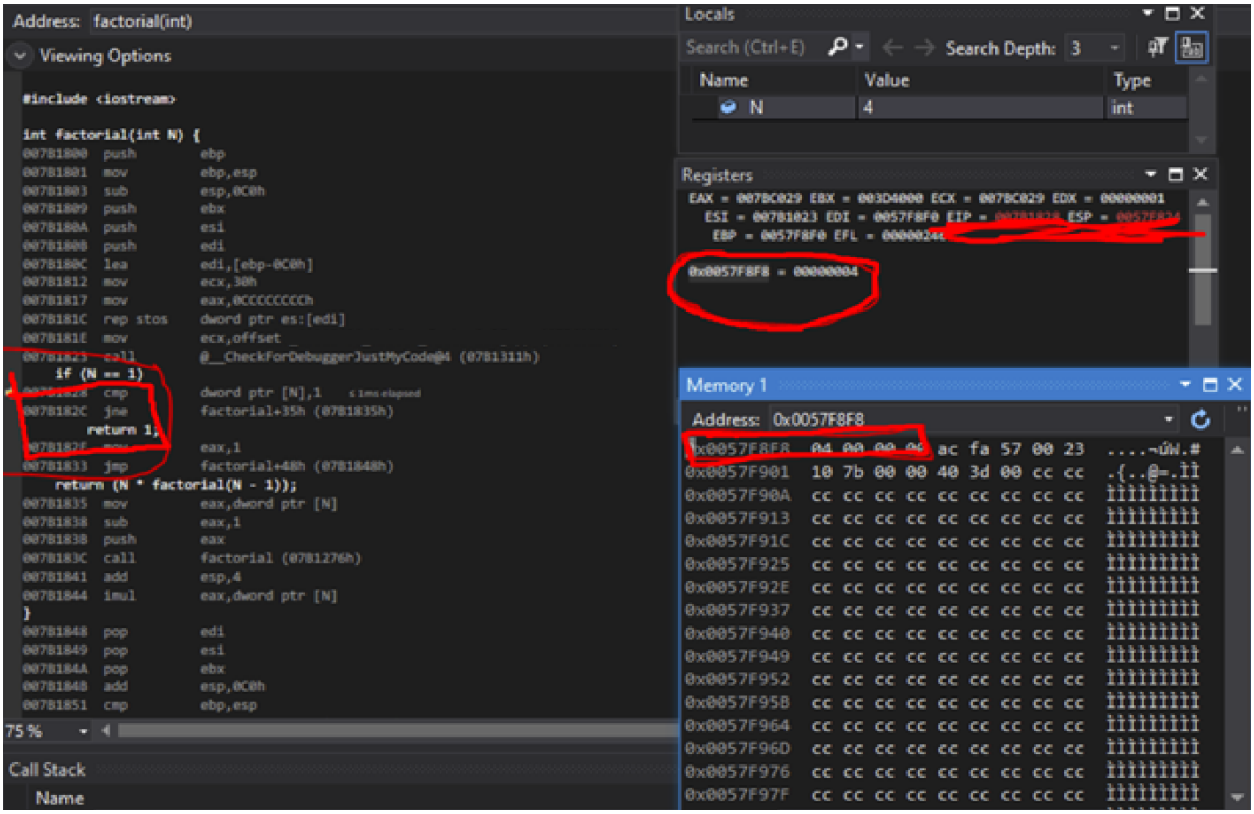




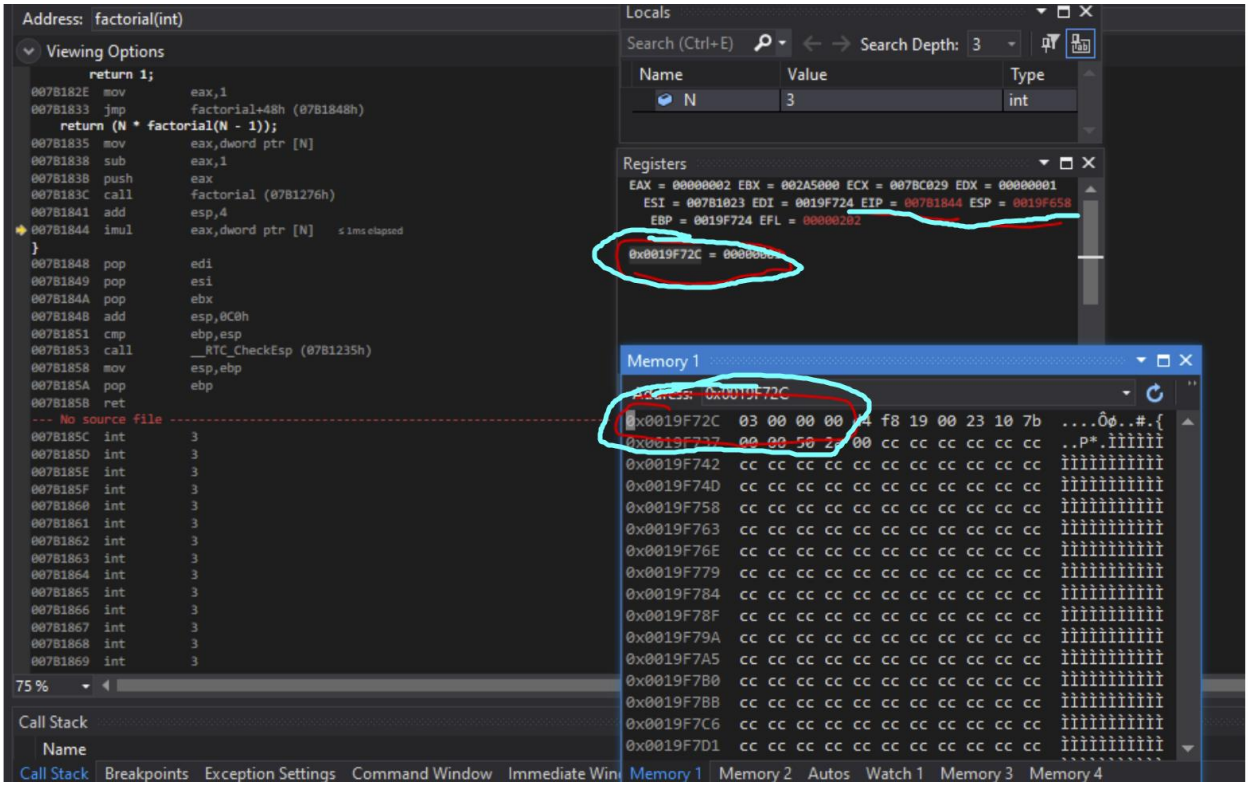
The local value of 5 is stored in 0x006FFFbF0 on the stack. After the call is done the instruction address is 0x007B1835. The code will jump to the factorial address which is 0x007B1276. The return address is now 0x05007b18bf which gets pushed into our stack.



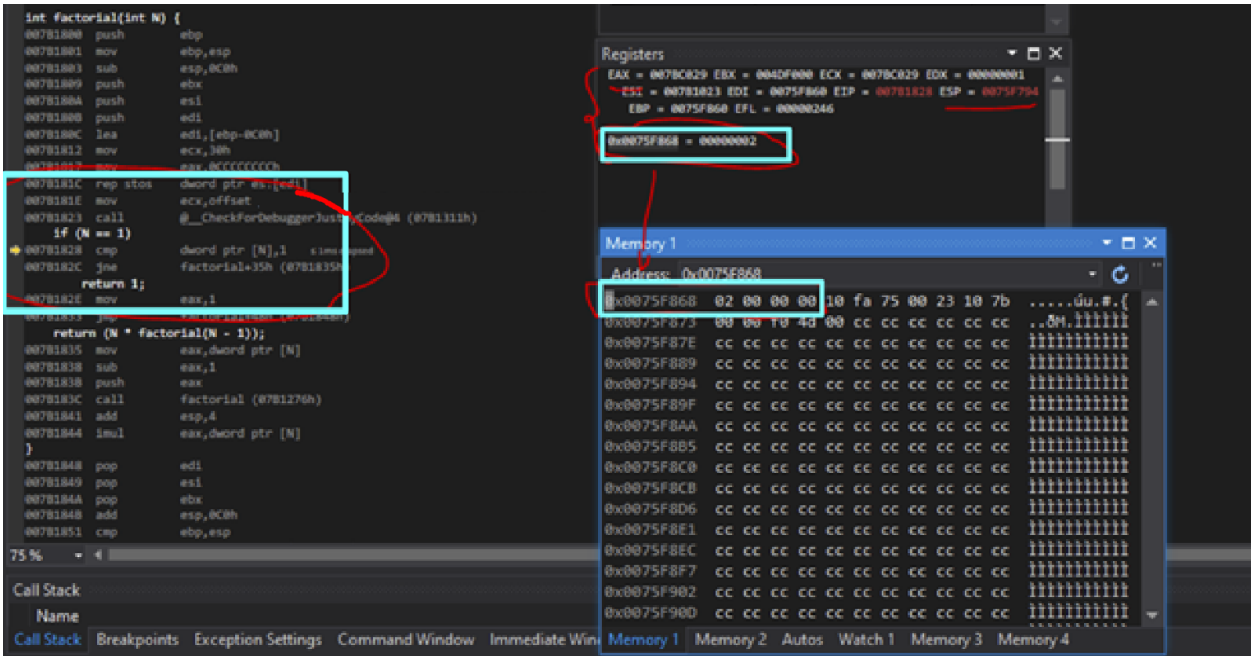
The base pointer is now pushed into the stack. The new stack frame is EBP value of 0x0115FA54. Now the code will check if our initial value is 1 and if it not then it will perform the jump instruction.



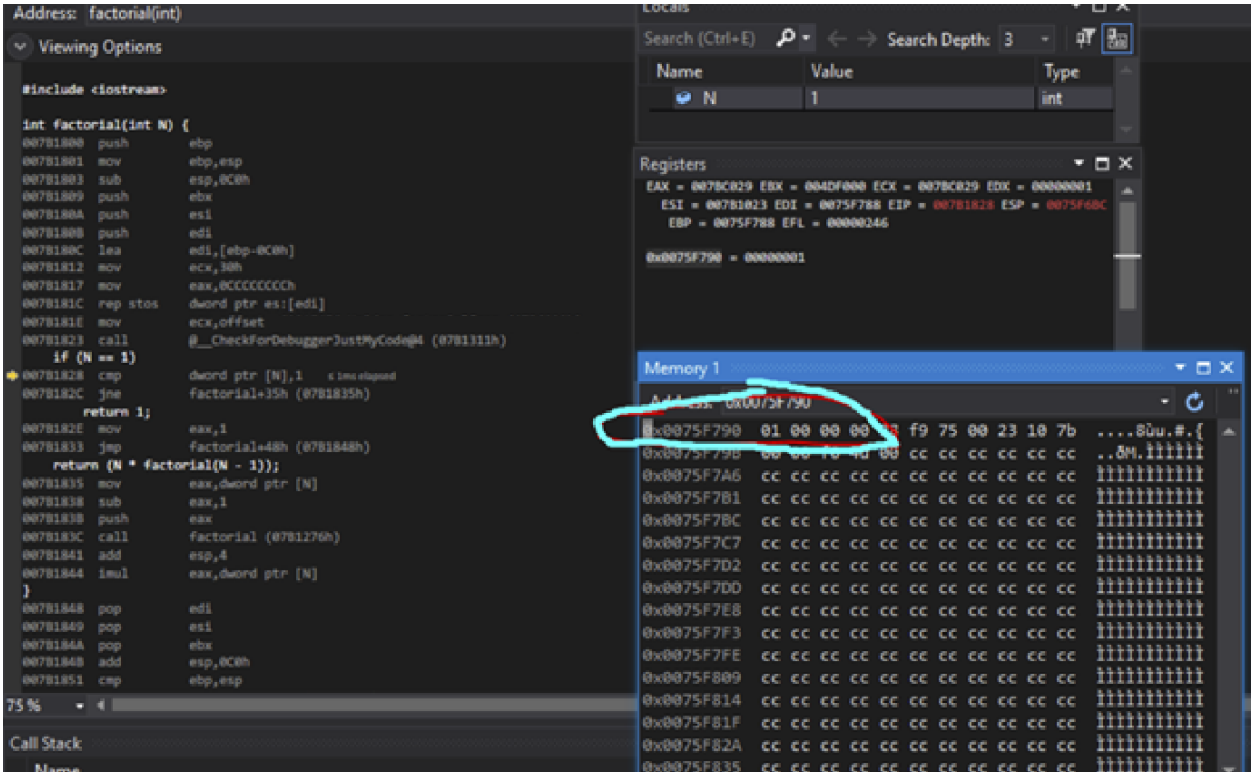
The value of our local variable 5 is reduced by one, to 4; this value is saved in the EAX register as 0x00000004. 0x0057F8F8 is the address where this is saved. Because this value was copied from the EAX registers, the EAX currently has a different value. The value 0x0057F9C8 is included within our stack pointer of 0x0057F824. After the call instruction is completed inside the factorial address, this is the address of the instruction.



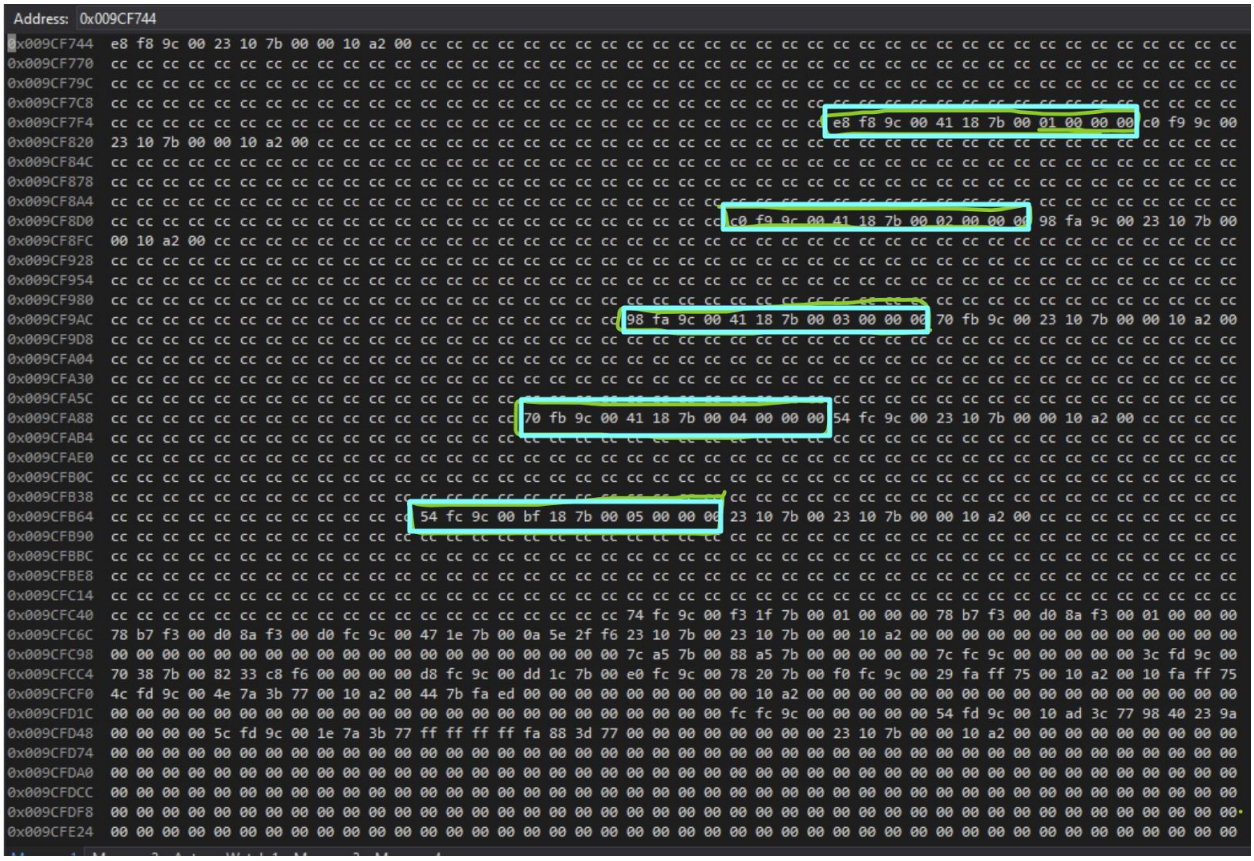
The value 4 is decreased again to 3 and EAX is then pushed on to the stack address of 0x0019F72C. The return address is 0x007B1844. The base pointer is at 0x0019F724.



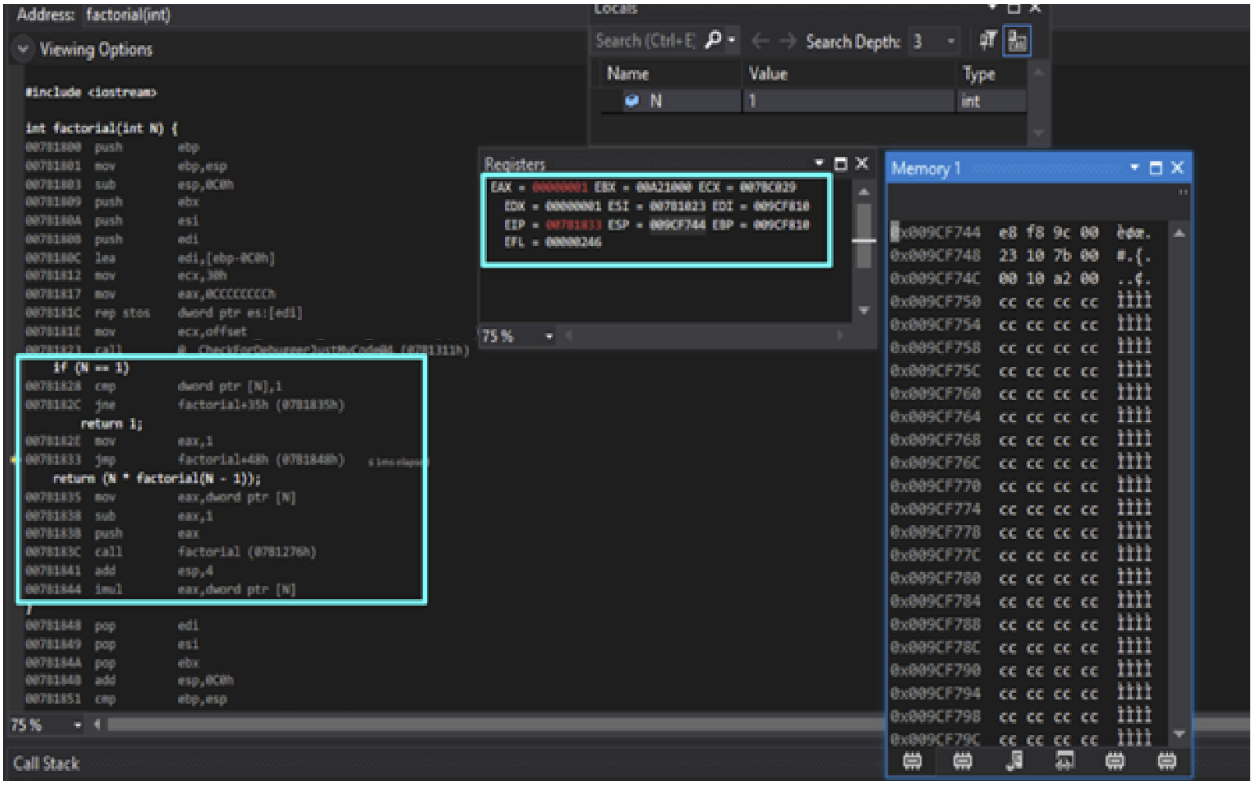
The value of N is decremented again by 1 and this time the value is 2 and the address it is stored is 0x0075F868. The EIP is 0x007B1728 and the ESP is 0x0075F794. Since the value still does not equal to 1 another call will occur.



And now the value will get stored to the local variable again as well the EPI,ESP will change too. Once the value reaches 1 a jump will occur to instruction 0x00F8170E. This will end the function.

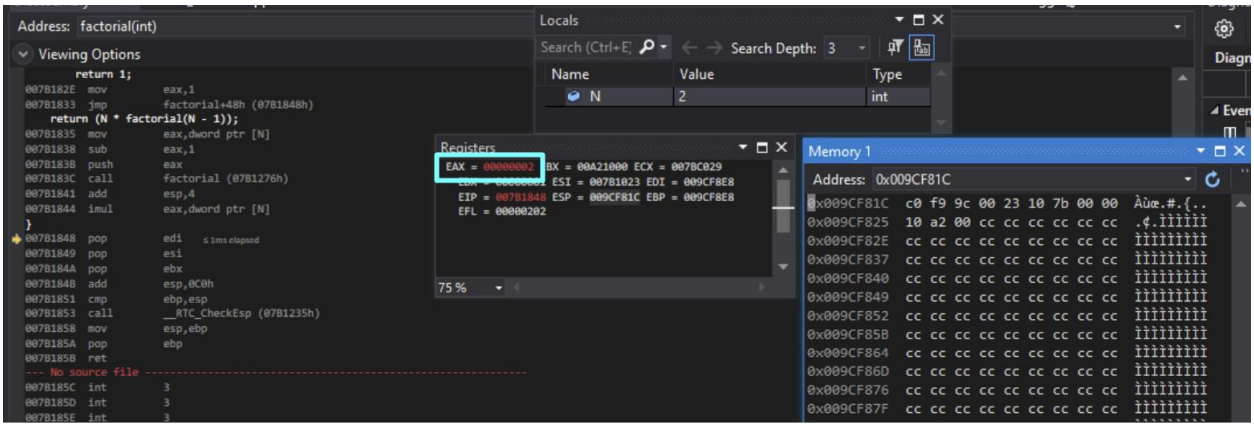


These are all values stored in the stack frame during the function call. The function has not started to return any values yet. The last 8 bits are pushed arguments. The middle 8 bits are return addresses for each pushed argument and then finally the first 8 bits at the beginning are base addresses.

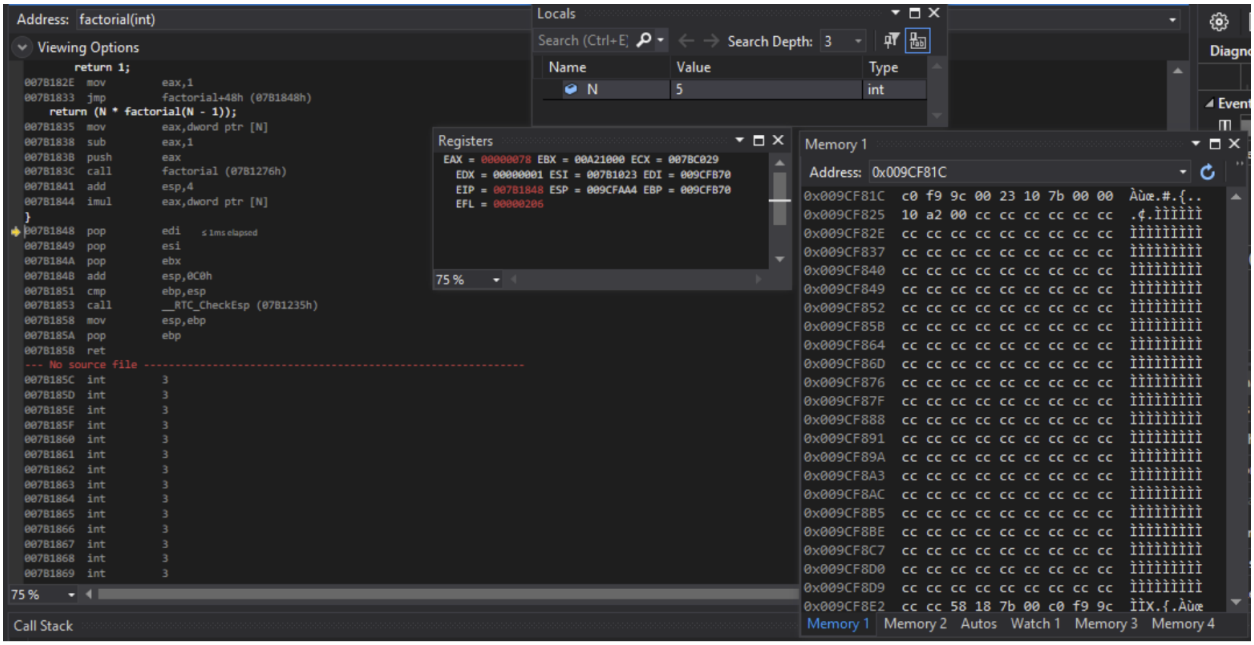


The stack frame is reset to the base pointer, which is the value of our EBP. The return address will now be 0x009cf8e8, which is the address at the top of the stack. The code returns to the base pointer after the return instruction is completed, and the return address is popped off the stack and deallocated.

As this continues the register EAX will store all the values of recursive call multiplication and the return address will keep getting deallocated from the stack until we get a final answer. I will show only one instance of this which is provided below. Below, 2 gets multiplied to 1 and the value is stored in the EAX register



This process will continue until we get all the factorial function call done and the final answer will be 120.



This is the picture of our final return call and as you can see the answer is 0x0000078 which is 120 in decimal value.

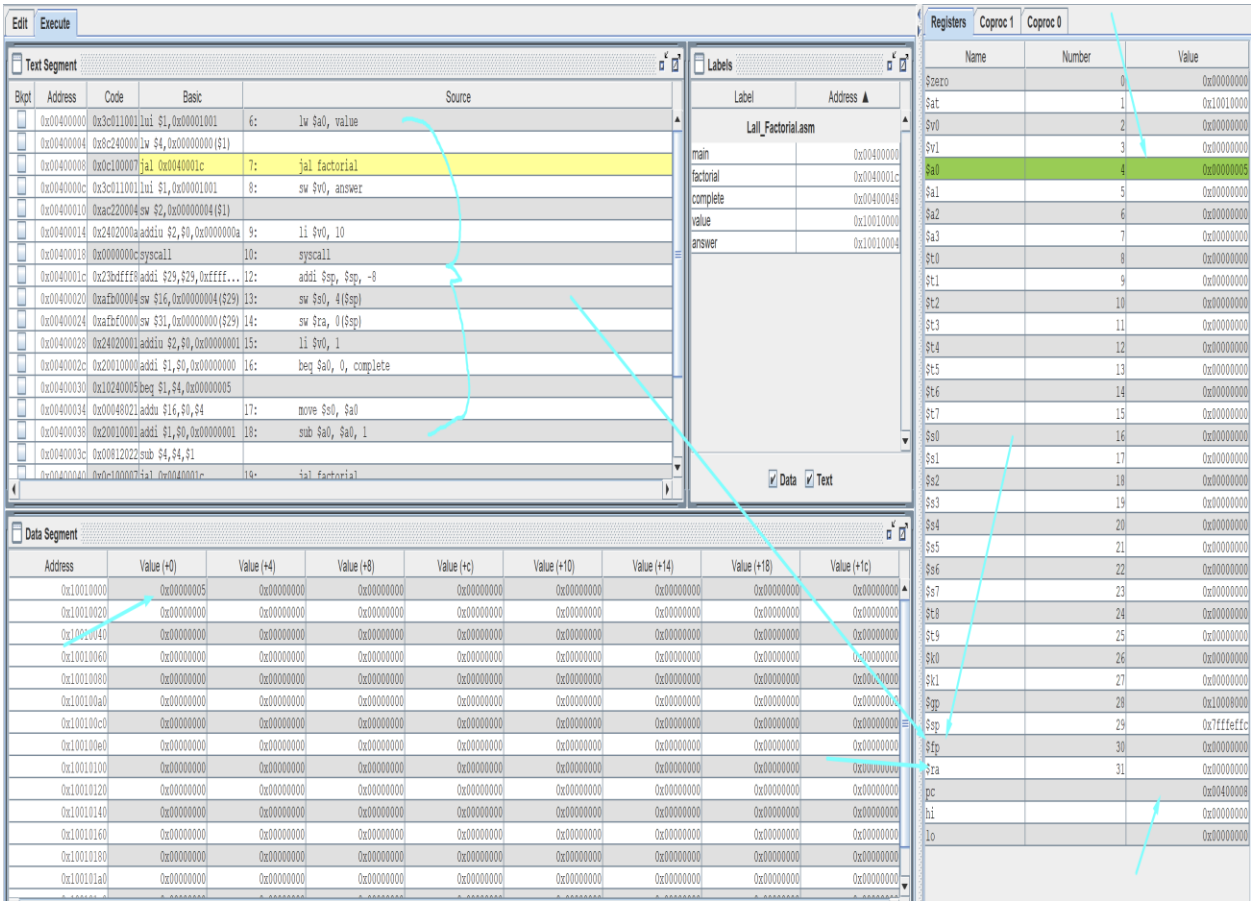
MIPS:

Lall_Factorial.asm

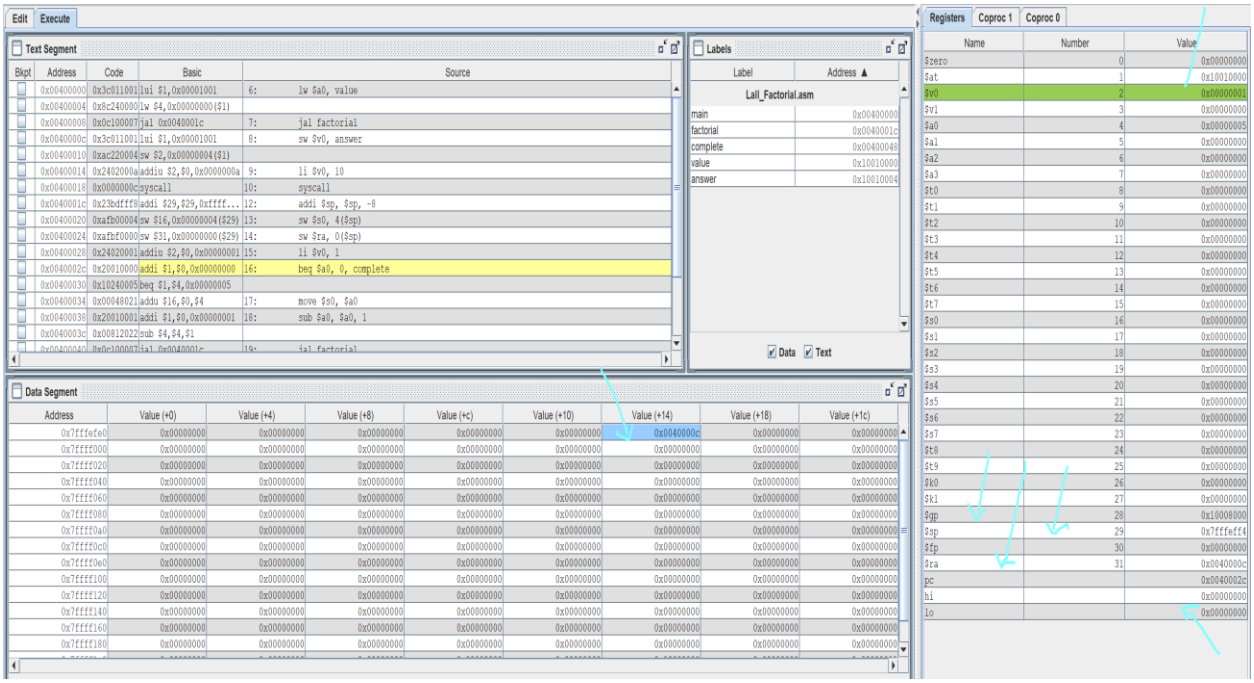
```
1   .data
2   value: .word 5
3   answer: .word
4  .text
5  main:
6           lw $a0, value
7           jal factorial
8           sw $v0, answer
9           li $v0, 10
10          syscall
11 factorial:
12          addi $sp, $sp, -8
13          sw $s0, 4($sp)
14          sw $ra, 0($sp)
15          li $v0, 1
16          beq $a0, 0, complete
17          move $s0, $a0
18          sub $a0, $a0, 1
19          jal factorial
20          mul $v0, $s0, $v0
21          complete:
22                  lw $ra, 0($sp)
23                  lw $s0, 4($sp)
24                  addi $sp, $sp, 8
25                  jr $ra
```

This is the code for factorial in MIPS assembly. The variable value is set to 5 for us in this case and this value is loaded on to the register $a0. This is the value we will calculate the factorial for.

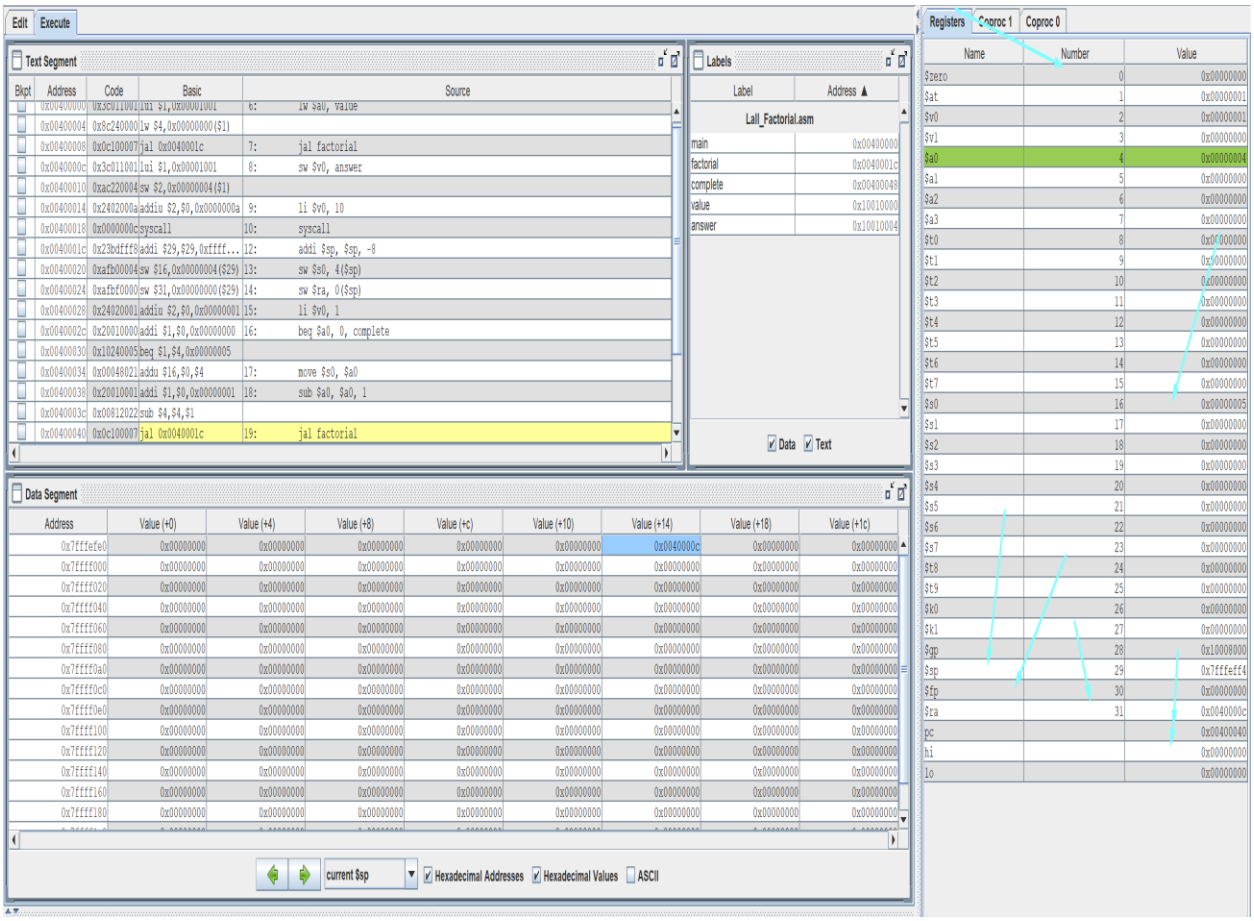
Analysis:



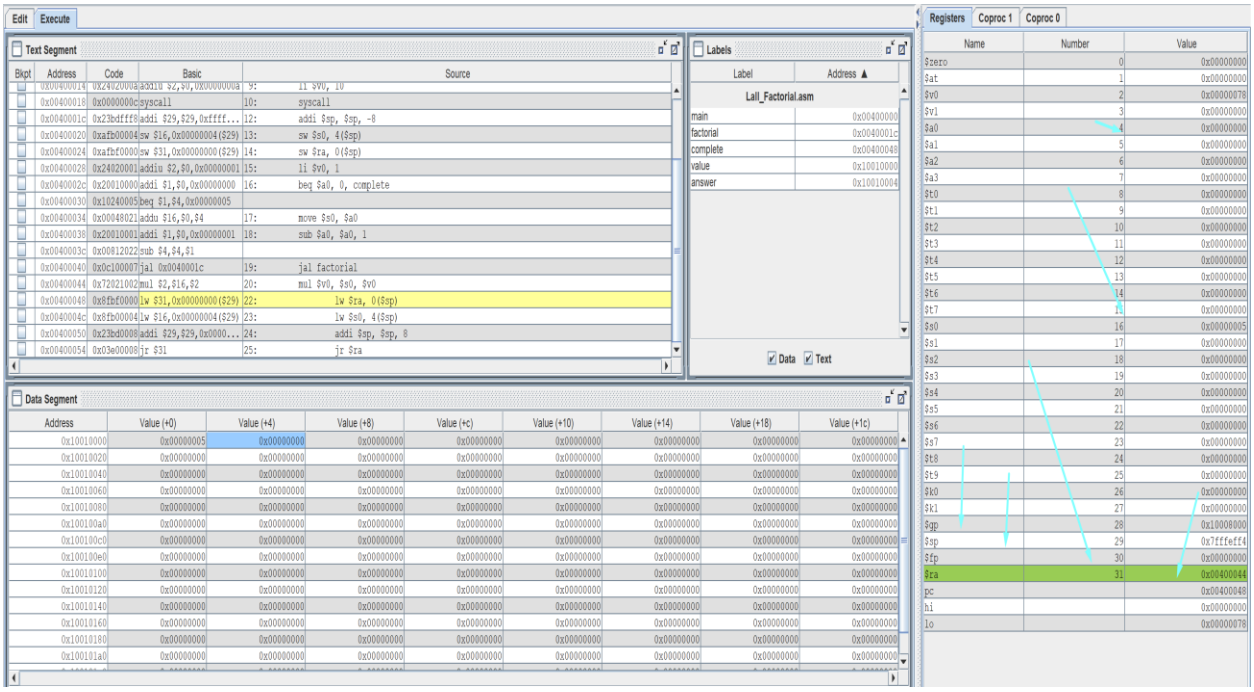
The PC value is 0x00400008, which corresponds to our jal instruction call on line 7 of our disassembled code. The register "ra" holds no instruction address because no instruction has been executed yet. This register will receive values after the jump and link instructions have been executed. Our input of 5 is stored at address 0x10010000. The variable value is 0x00000005, as you can see. The result will be saved at the address 0x10010004. When we run our program, we can see that the input value of 5 is placed in register $a0. Our stack pointer, or sp, is pointing to 0x7fffeffc.



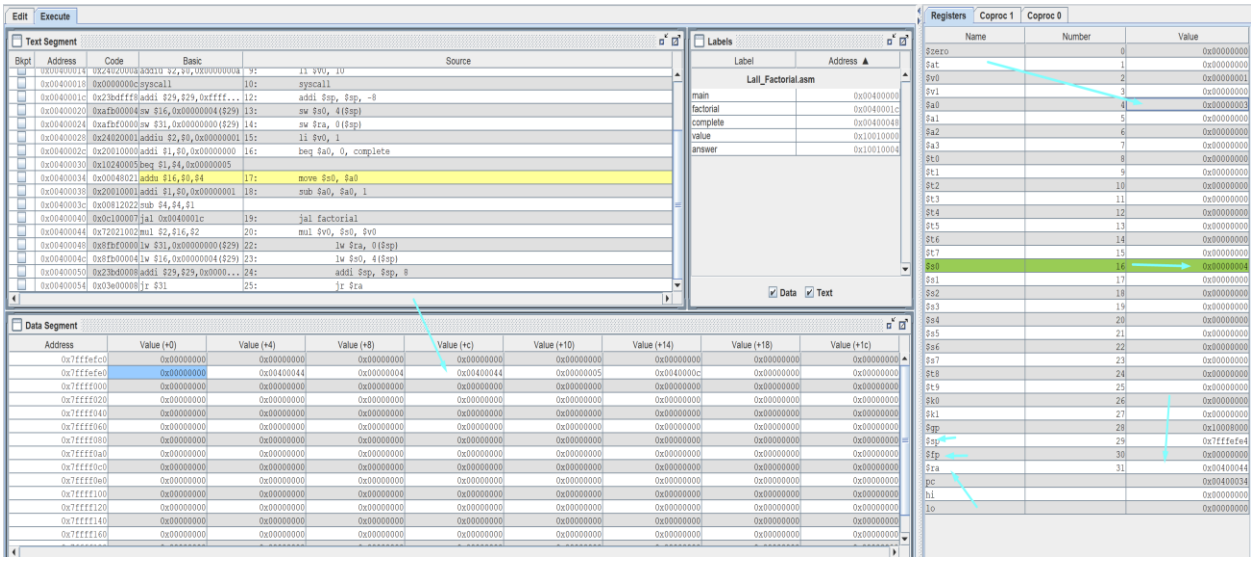
The initial procedure call has been finished, as seen in this image. As a result, the return address in the register $ra will be used in the jump and link instructions. The address of the next instruction following the jump and link instructions is 0x0040000c, which is stored in register $ra. Once all of the procedure calls for this function have been completed, this is our return address. The pc has been changed to 0x0040002c, and the sp has been changed to 0x7fff3ff4. The addresses of the $s0 and $ra registers are 0x7fffeff8 and 0x7fffeff4, respectively, and the value in 0x7fffeff4 is 0x0040000c, which is our return address. Also placed into register $v0 is the value 0x00000001, which is our default return value. The following instruction is 0x0040002c, which will verify if the argument is 0, and if it is, it will leap to sending a complete signal. Because the value of $a0 is 5 right now, the branch will bypass it.



This is before the factorial function is called again in the second procedure. The value of 0x00000005 is transferred into $s0, and the value of $a0 is decremented by one, resulting in a 4 instead of a 5. To conduct the factorial, it will now call the jump and link call one more.

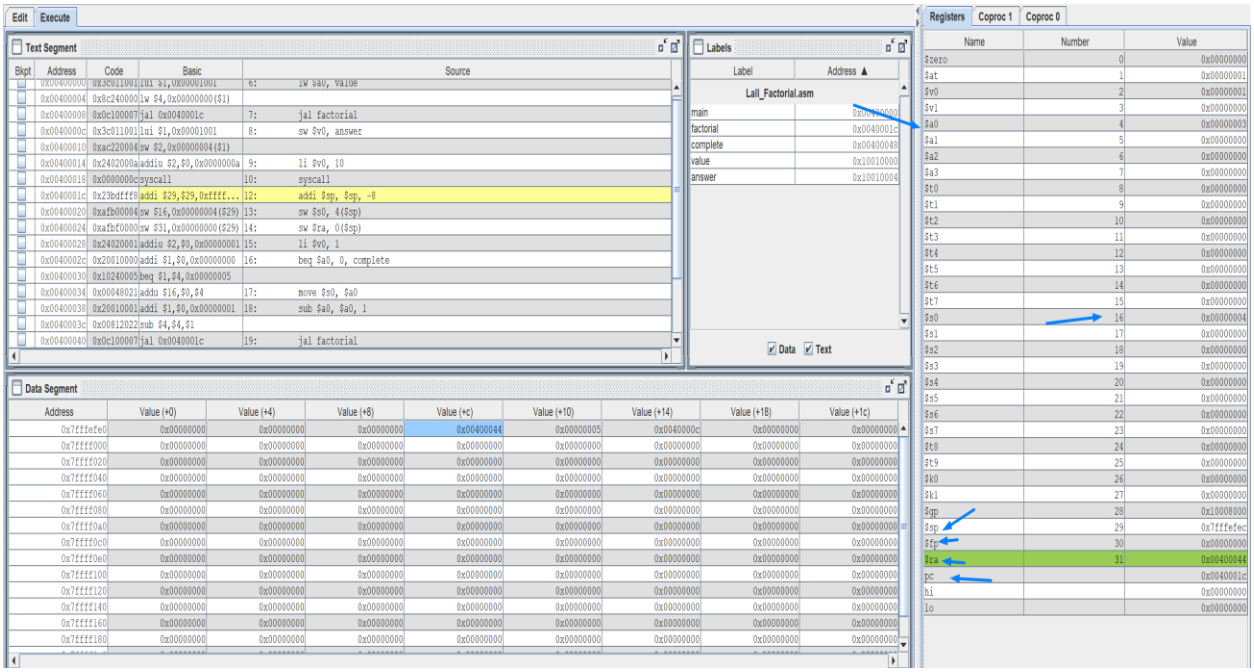


As you can see, the PC changes to 0x0040001c when the jump instruction is completed, which was our initial process from the factorial call. The value of the register $ra is now 0x00400044. This is the command that will be run immediately after the jump and link have been completed.

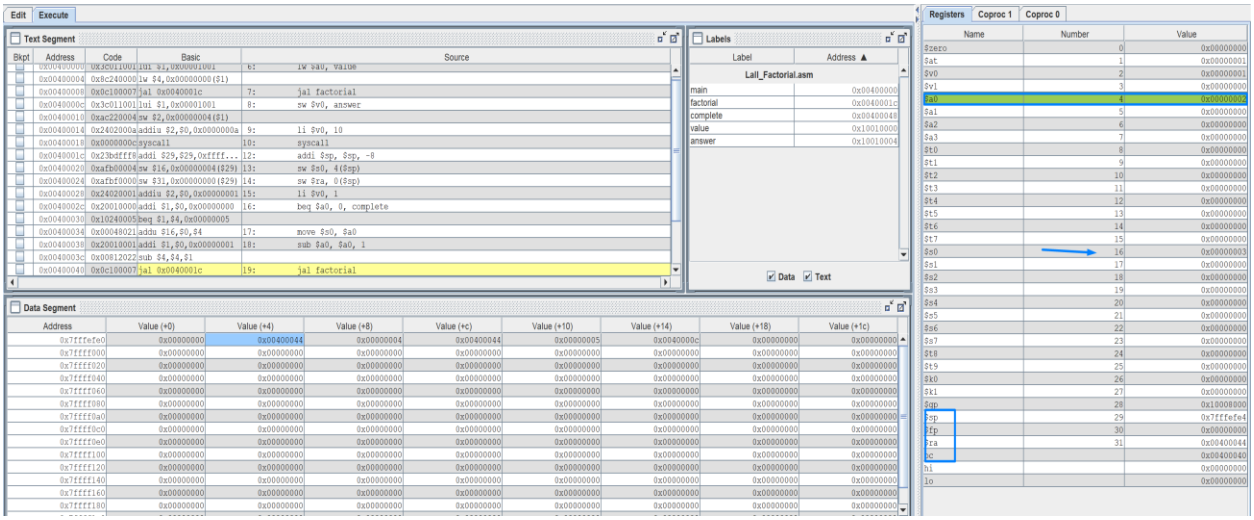


This occurs just before our factorial function's third call. The stack pointer is now 0x7fffeffc, and the value in $s0 is now 0x00000004, which is saved in 0x7fffefec inside the 18 stack. Because the value of 4

from the previous call is now decrementing by 1, $a0 is 3. 0x00400040 is the pc value for the next instruction.

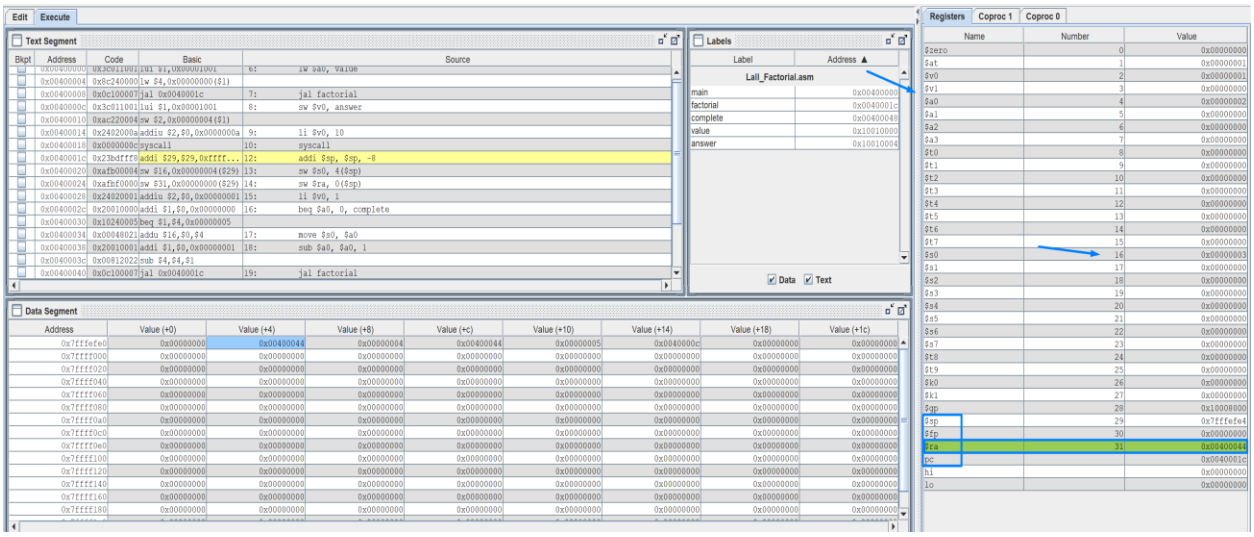


Right after the jump instruction is done performing now $ra becomes 0x0040044 and pc now changes to 0x0040001c which is again our first instruction.

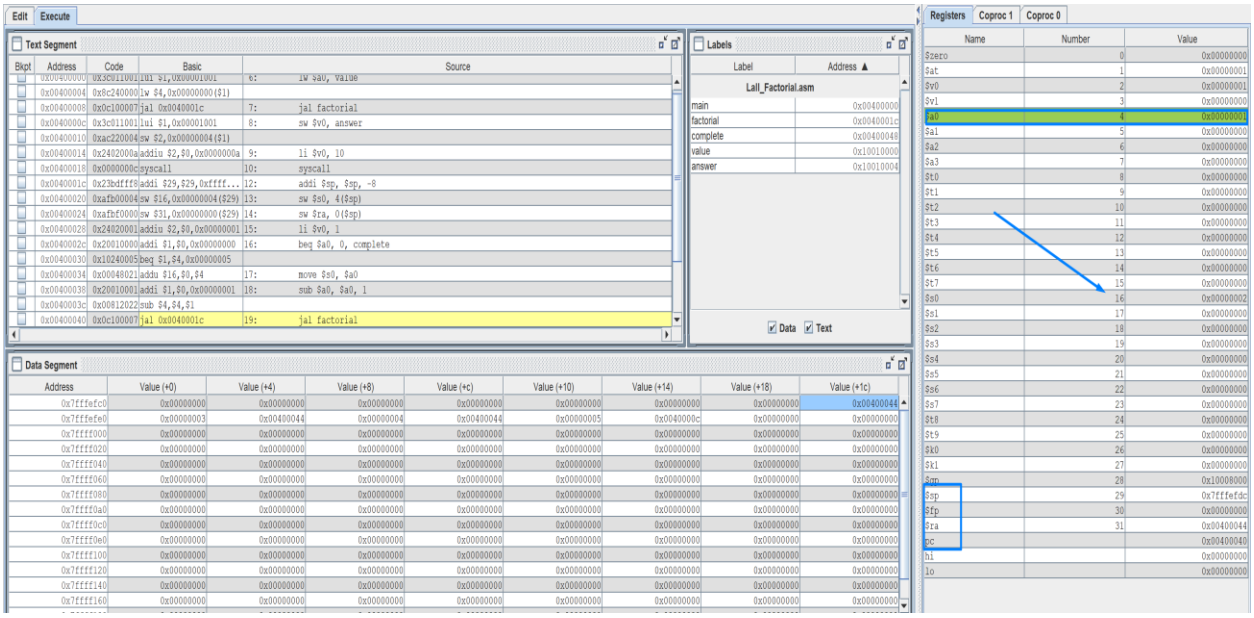


I'm not placing any pointers before the fourth call because all of the registers we're talking about have already been pointed to in the previous calls. 0x7fffefe4 is now the stack pointer. The value stored in $s0

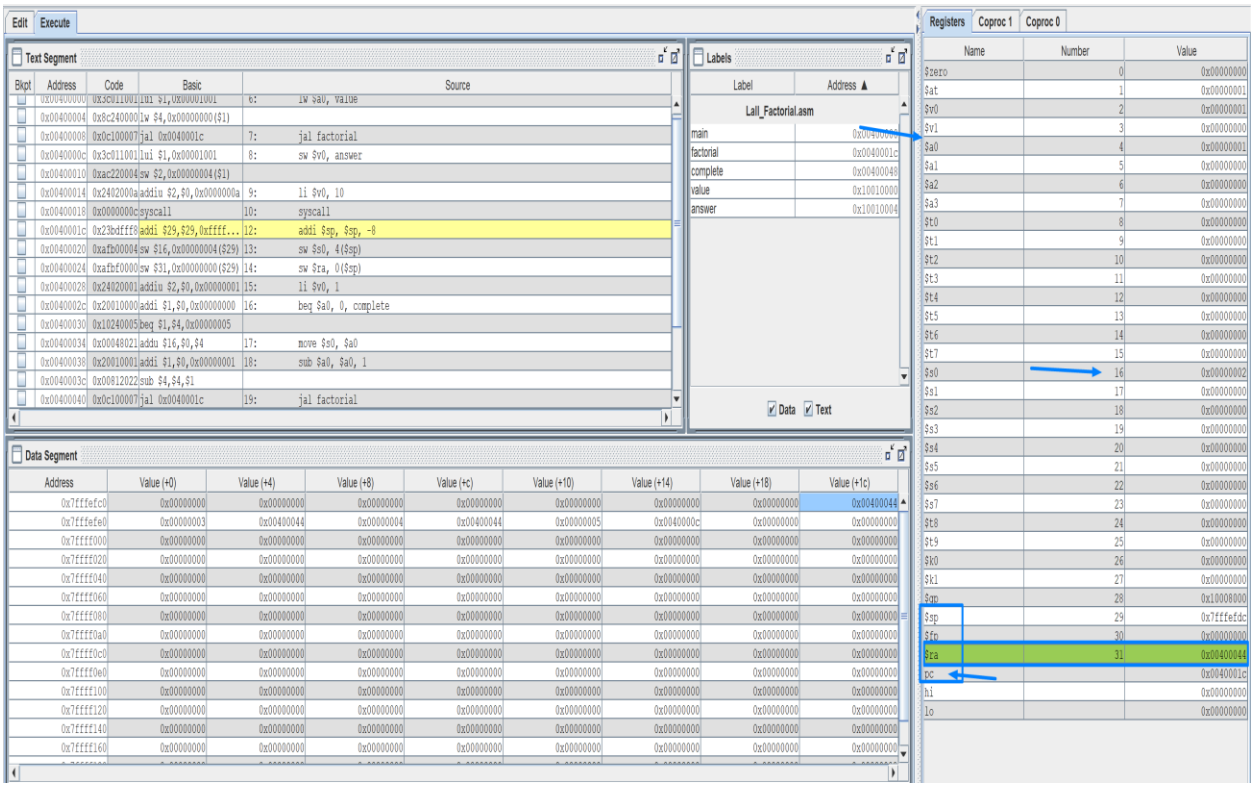
is 0x00000003, and $a0 now equals 2 because 1 must be subtracted from 3. The value of $ra is 0x00400044, which is not saved at 0x7fffefe4, which is +4 offset from the stack pointer. The computer has now been configured to 0x00400040.



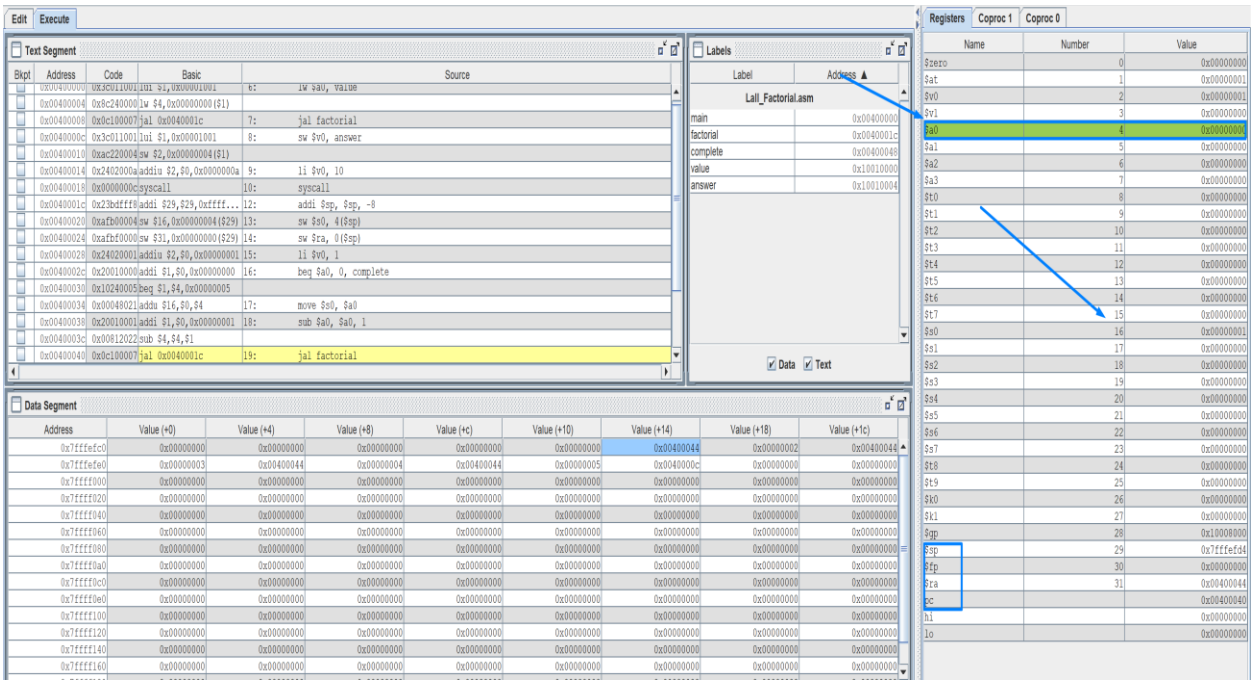
After the fourth call is done the pc will become 0x0040001c and the return address of 0x00400044 is stored in $ra.



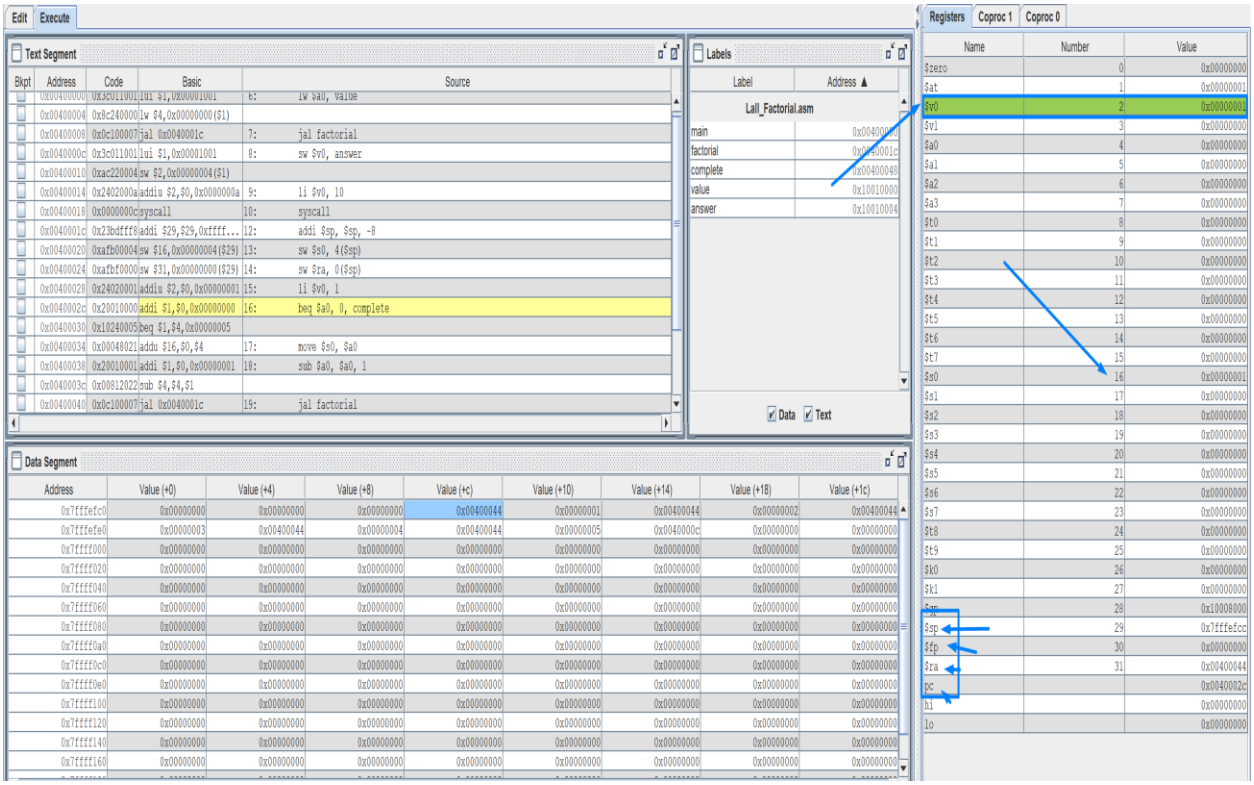
The stack pointer is 0x7fffefdc before the 5th call, and the data from $s0 is placed at 0x7fffefe0 inside the stack. Inside the stack, the data in $ra is 0x7fffefdc. The value of $a0 is now decremented by one, resulting in 1 and pc becoming 0x00400040.



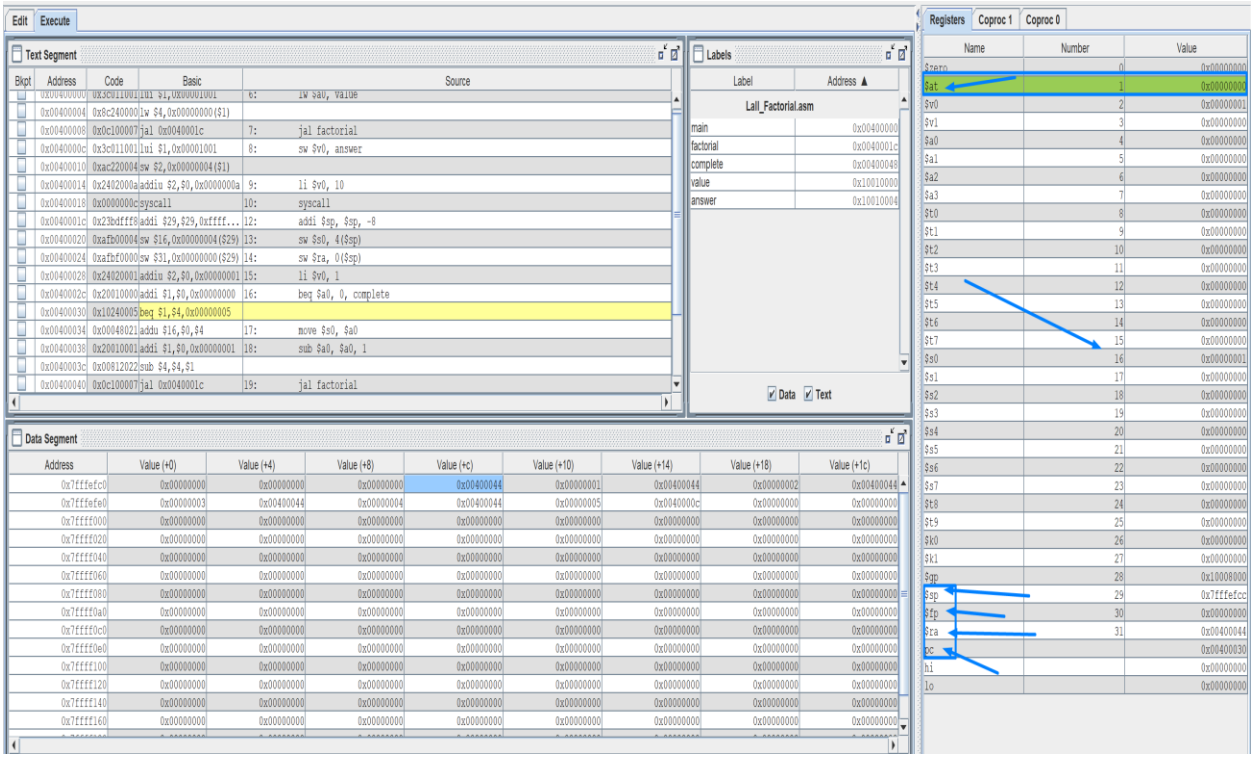
After the 5th call the pc is now set to 0x0040001c and the return address is 0x00400044.



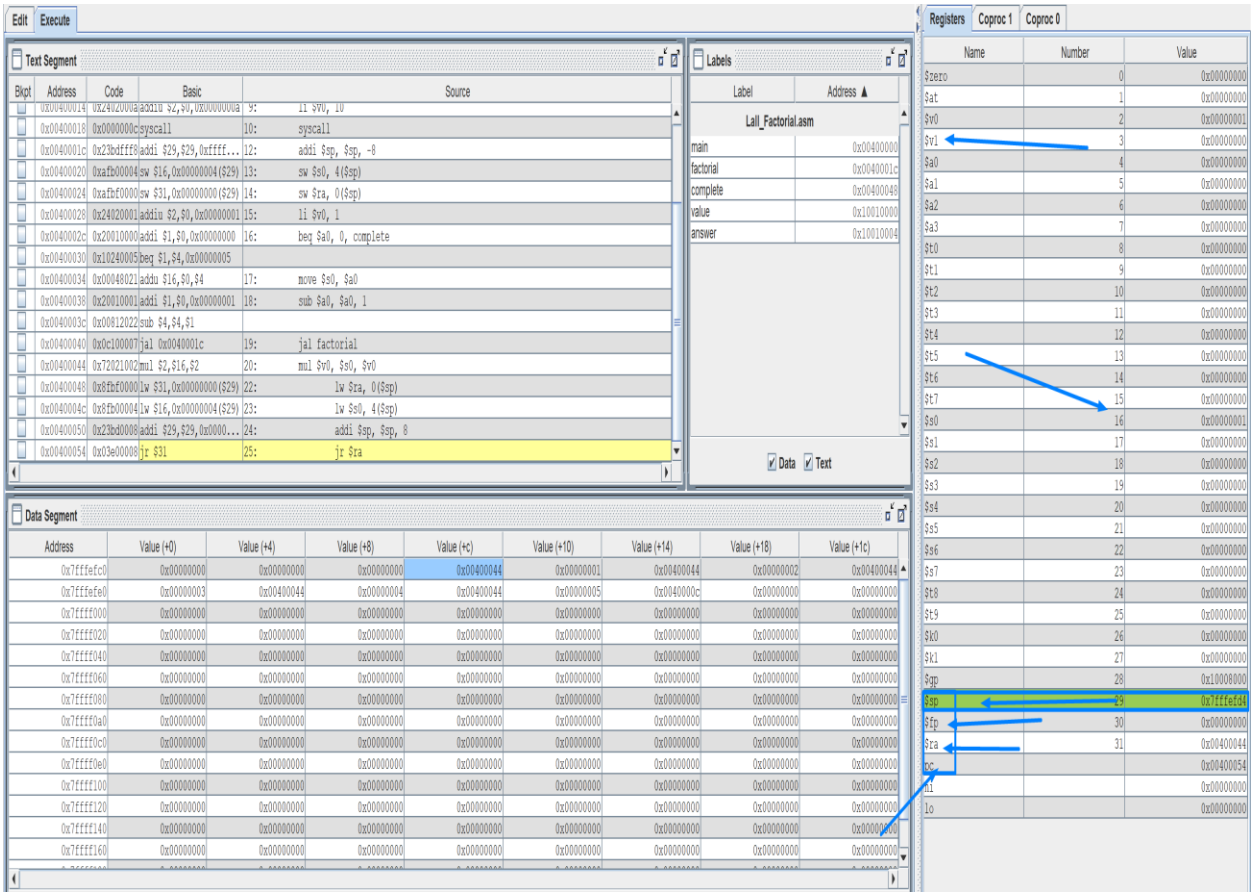
This is before we execute the final process, and we need to assign even more resources, which is why the data is in the second row. 0x7fffefd4 is now the stack pointer. Inside the stack, the value of register $s0 is set to 0x7fffefd8, and it now has the value of 2. Inside the address 0x7fffefd4 is the return address 0x00400044. Because 1-1 = 0, $a0 now equals 0.



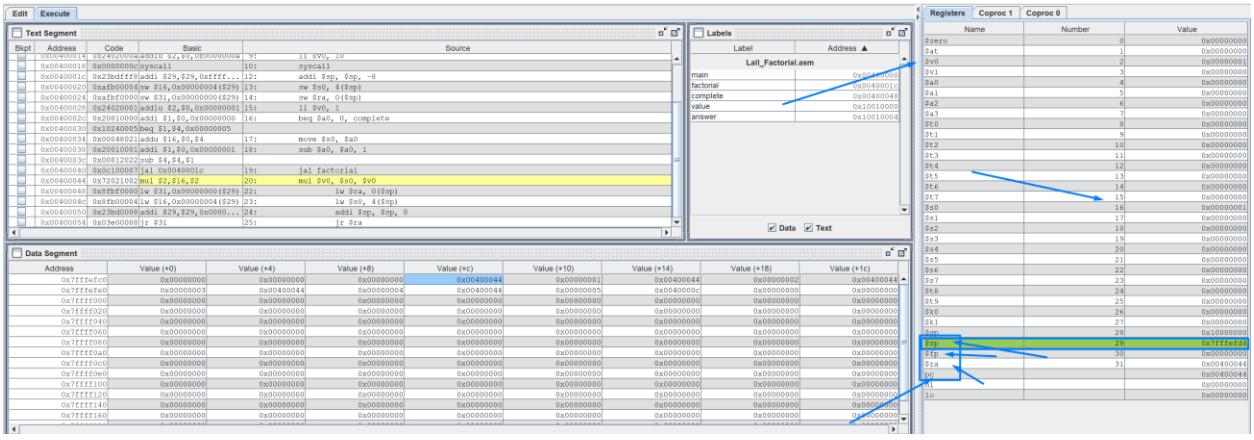
Now the branch call is executing because the value is now 0 so it is equal to what we coded for our condition. Now the values of $s0 and $ra gets saved to the stack. The value of $a0 is 0 and now the code will move toward “complete” instructions.



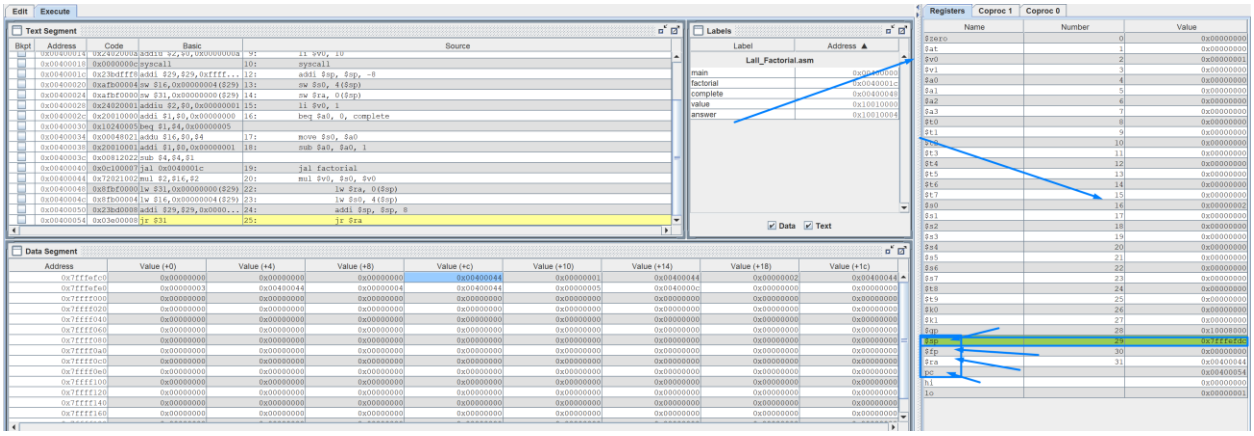
The beq branch now runs and the pc is set to 0x00400048, which is the first instruction under the Label we generated for the condition "complete".



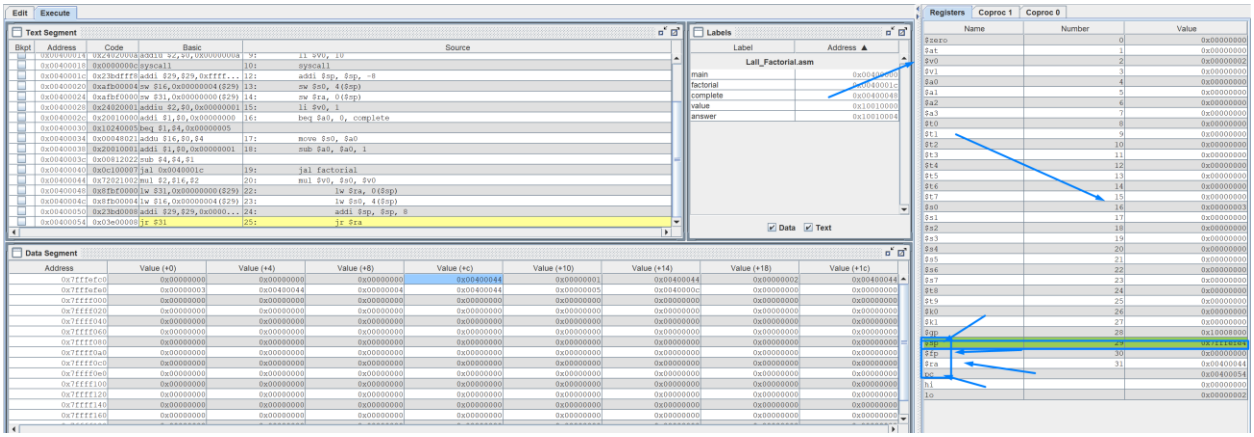
Now we are about the perform the jump inside this label and $ra will now contain 0x00400044 and $s0 will contain the value of 0x01. After this is done the memory this occupied will be deallocated. The stack pointer will become 0x7fffefd4.



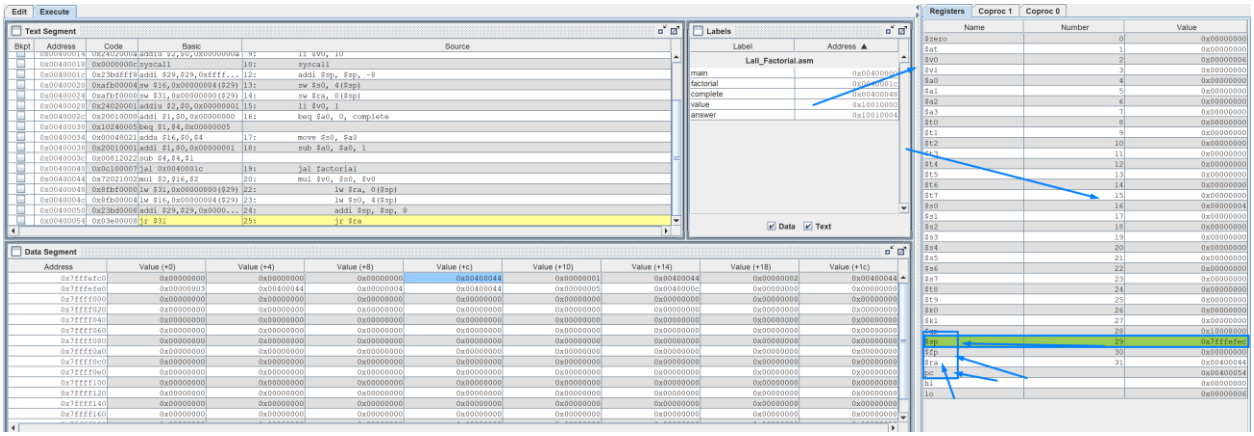
Now that the jump instruction has been performed, the values from the return registers and the program counter are copied. The pc counter is now set to 0x00400055, which is the address of our return value. The instruction now multiplies the value of $v0, which is 1, by the value of $s0, which is also 1, and stores the result in register $v0. We'll keep going inside this label until we've gotten all of the results into $v0.



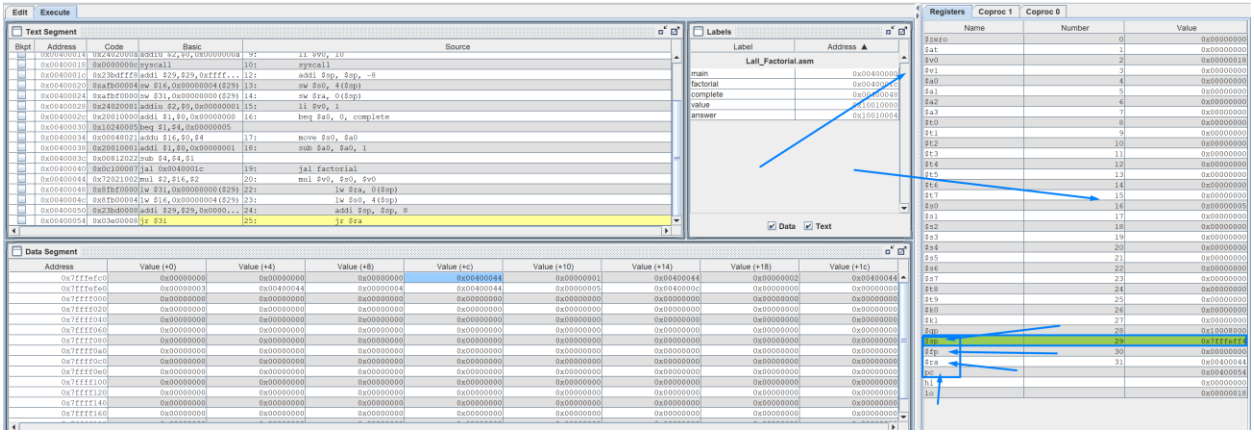
The result of the multiplication is now visible in $v0. The values of the $s0 register and $ra register are not popped from the stack and those are deallocated. Now it increments +1 so $s0 now becomes 2. This will keep happening until we reach back to 5 inside the $s0 register.



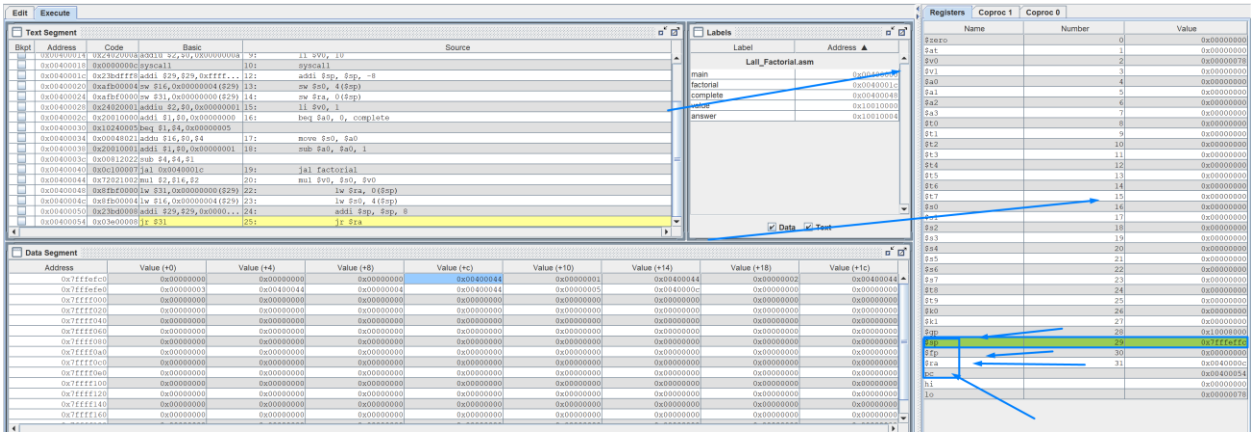
The value is again stored in $v0 and it is 1. This value is multiplied with $s0 value which is 2 so the result is 2.



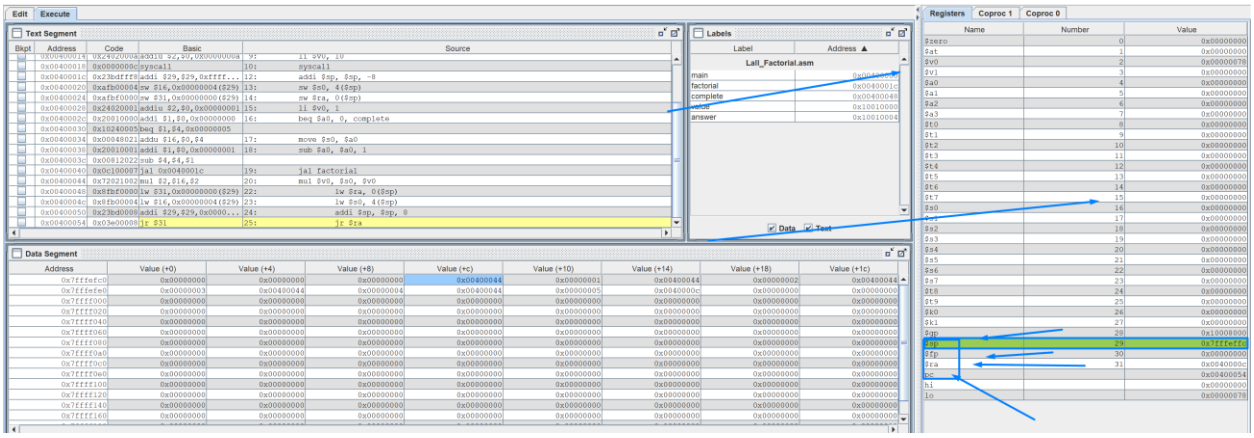
The value was 2 and now gets multiplied with 3 so it becomes 6 and is stored in $v0.



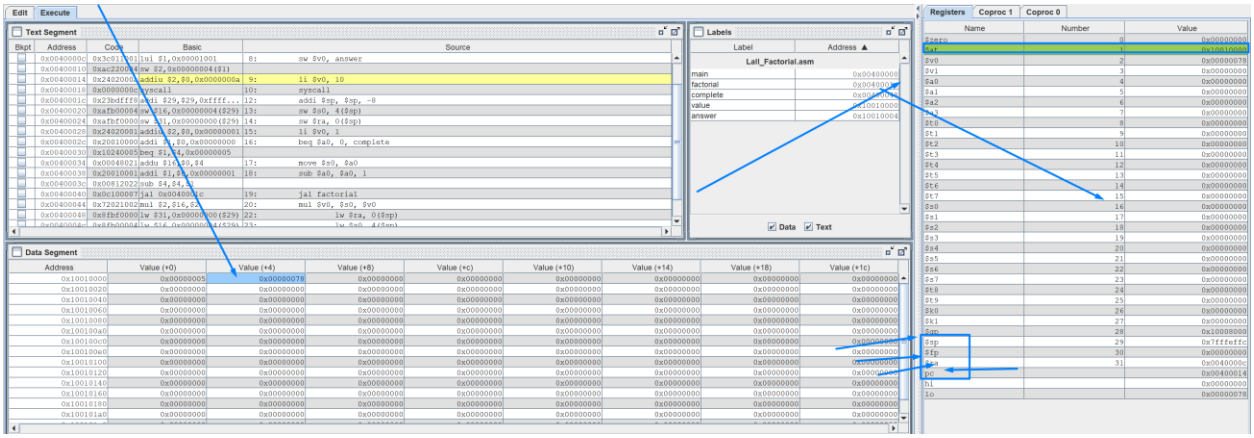
Now the value is 18 because 6 * 4 is equal to 24 and hex of that is 0x18. As you can see the $s0 and $ra values are popped from stack again and the stack pointer again changed to 0x7fffefe4 like how it was changing in previous calls.



Now we multiply the value of 5 by our previous answer to get the value of 120, which is 0x78 in hex and visible in our $v0 register. $s0 now equals 0 and $ra equals 0x0040000c. Now we'll go on to the main function of this program.



The final jump is done and pc goes back to the address of 0x0040000c which is right after our factorial function call in main. Now this code will store the value of $v0 which is 0x78 into the memory.



As you can see, the 0x78 value has been loaded to the address 0x10010004, which is the static variable's address location. Our code will now quit, and the mips procedure for a recursive factorial call will be complete.

LINUX GDB:

```c
C Lall_Factorial.c > main()
#include <stdio.h>

int factorial(int N)
{
    if (N == 1)
        return 1;
    return(N * factorial(N - 1));

}

int main()
{
    int N_fact = factorial(5);
}
```

Analysis:

```
   0x0000555555555129 <+0>:     endbr64
   0x000055555555512d <+4>:     push   %rbp
   0x000055555555512e <+5>:     mov    %rsp,%rbp
=> 0x0000555555555131 <+8>:     sub    $0x10,%rsp
   0x0000555555555135 <+12>:    mov    %edi,-0x4(%rbp)
   0x0000555555555138 <+15>:    cmpl   $0x1,-0x4(%rbp)
   0x000055555555513c <+19>:    jne    0x555555555145 <factorial+28>
   0x000055555555513e <+21>:    mov    $0x1,%eax
   0x0000555555555143 <+26>:    jmp    0x555555555156 <factorial+45>
   0x0000555555555145 <+28>:    mov    -0x4(%rbp),%eax
   0x0000555555555148 <+31>:    sub    $0x1,%eax
   0x000055555555514b <+34>:    mov    %eax,%edi
   0x000055555555514d <+36>:    call   0x555555555129 <factorial>
   0x0000555555555152 <+41>:    imul   -0x4(%rbp),%eax
   0x0000555555555156 <+45>:    leave
   0x0000555555555157 <+46>:    ret
```

This is the disassemble window for the factorial function before the first call is made. The first instruction to be executed is stored in 0x0000555555555129.

```
(gdb) info registers
rax            0x555555555158      93824992235864
rbx            0x0                 0
rcx            0x0                 0
rdx            0x7fffffffdec8      140737488346824
rsi            0x7fffffffdeb8      140737488346808
rdi            0x5                 5
rbp            0x7fffffffdd70      0x7fffffffdd70
rsp            0x7fffffffdd70      0x7fffffffdd70
r8             0x7ffff7fa5bd0      140737353767888
r9             0x7ffff7fd9d00      140737353981184
r10            0xfffffffffffffb88  -1144
r11            0x202               514
r12            0x7fffffffdeb8      140737488346808
r13            0x555555555158      93824992235864
r14            0x0                 0
r15            0x7ffff7ffbc40      140737354120256
rip            0x555555555131      0x555555555131 <factorial+8>
eflags         0x202               [ IF ]
cs             0x33                51
ss             0x2b                43
ds             0x0                 0
es             0x0                 0
fs             0x0                 0
gs             0x0                 0
```

This is all the registers and how they are changing during the first call. As you can see above. I will be showing the registers together next time along with the step call, so this is the only time I will display all the registers. As you can see from this diagram the base pointer RBP is 0x7fffffffdd70 and the RSP the stack pointer is pointing to 0x7fffffffdd70. The argument RDI will get pushed into the stack and if the RDI does not equal to 1 then a jump will be performed.

```
(gdb) x/2xw ($rsp)
0x7fffffffdd70: 0xffffdd90      0x00007fff
(gdb) print /x $rbp
$1 = 0x7fffffffdd70
(gdb) print /x $rsp
$2 = 0x7fffffffdd70
```

The RSP holds the next instructions to be executed so the next one is at address 0xffffdd90.

```
Dump of assembler code for function factorial:
   0x0000555555555129 <+0>:     endbr64
   0x000055555555512d <+4>:     push   %rbp
   0x000055555555512e <+5>:     mov    %rsp,%rbp
=> 0x0000555555555131 <+8>:     sub    $0x10,%rsp
   0x0000555555555135 <+12>:    mov    %edi,-0x4(%rbp)
   0x0000555555555138 <+15>:    cmpl   $0x1,-0x4(%rbp)
   0x000055555555513c <+19>:    jne    0x555555555145 <factorial+28>
   0x000055555555513e <+21>:    mov    $0x1,%eax
   0x0000555555555143 <+26>:    jmp    0x555555555156 <factorial+45>
   0x0000555555555145 <+28>:    mov    -0x4(%rbp),%eax
   0x0000555555555148 <+31>:    sub    $0x1,%eax
   0x000055555555514b <+34>:    mov    %eax,%edi
   0x000055555555514d <+36>:    call   0x555555555129 <factorial>
   0x0000555555555152 <+41>:    imul   -0x4(%rbp),%eax
   0x0000555555555156 <+45>:    leave
   0x0000555555555157 <+46>:    ret
```

The jump will occur at 0x00005555555555145 since the value was not equal to 1. The argument will be moved into the register RDX using this instruction. This procedure will repeat until the value is not equal to one, just like the previous two platforms, so I'll skip that part and go straight to when the values are returned once the call is completed.

```
(gdb) info registers
rax            0x3                 3
rbx            0x0                 0
rcx            0x0                 0
rdx            0x7fffffffdec8      140737488346824
rsi            0x7fffffffdeb8      140737488346808
rdi            0x3                 3
rbp            0x7fffffffdd30      0x7fffffffdd30
rsp            0x7fffffffdd30      0x7fffffffdd30
r8             0x7ffff7fa5bd0      140737353767888
r9             0x7ffff7fd9d00      140737353981184
r10            0xfffffffffffffb88  -1144
r11            0x202               514
r12            0x7fffffffdeb8      140737488346808
r13            0x555555555158      93824992235864
r14            0x0                 0
r15            0x7ffff7ffbc40      140737354120256
rip            0x555555555131      0x555555555131 <factorial+8>
eflags         0x206               [ PF IF ]
cs             0x33                51
ss             0x2b                43
ds             0x0                 0
es             0x0                 0
fs             0x0                 0
gs             0x0                 0
```

```
0x0000555555555131 <+8>:     sub    $0x10,%rsp
0x0000555555555135 <+12>:    mov    %edi,-0x4(%rbp)
0x0000555555555138 <+15>:    cmpl   $0x1,-0x4(%rbp)
0x000055555555513c <+19>:    jne    0x555555555145 <factorial+28>
0x000055555555513e <+21>:    mov    $0x1,%eax
0x0000555555555143 <+26>:    jmp    0x555555555156 <factorial+45>
0x0000555555555145 <+28>:    mov    -0x4(%rbp),%eax
```

This is the third call so after the value of 4 got stored in the stack now the code moves to the value of 3 which as you can see on the diagram is being stored at RDI. The RSP is pointing to 0x7fffffffdd30.

```
(gdb) info registers
rax            0x2                 2
rbx            0x0                 0
rcx            0x0                 0
rdx            0x7fffffffdec8      140737488346824
rsi            0x7fffffffdeb8      140737488346808
rdi            0x2                 2
rbp            0x7fffffffdd10      0x7fffffffdd10
rsp            0x7fffffffdd10      0x7fffffffdd10
```

```
0x0000555555555129 <+0>:     endbr64
0x000055555555512d <+4>:     push   %rbp
0x000055555555512e <+5>:     mov    %rsp,%rbp
0x0000555555555131 <+8>:     sub    $0x10,%rsp
0x0000555555555135 <+12>:    mov    %edi,-0x4(%rbp)
0x0000555555555138 <+15>:    cmpl   $0x1,-0x4(%rbp)
0x000055555555513c <+19>:    jne    0x555555555145 <factorial+28>
0x000055555555513e <+21>:    mov    $0x1,%eax
```

Same steps and explanation as before but this time the value is decremented to 2 because we are subtracting n-1.

```
rax            0x1                 1
rbx            0x0                 0
rcx            0x0                 0
rdx            0x7fffffffdec8      140737488346824
rsi            0x7fffffffdeb8      140737488346808
rdi            0x1                 1
rbp            0x7fffffffdcf0      0x7fffffffdcf0
rsp            0x7fffffffdcf0      0x7fffffffdcf0
```

```
Dump of assembler code for function factorial:
   0x0000555555555129 <+0>:     endbr64
   0x000055555555512d <+4>:     push   %rbp
   0x000055555555512e <+5>:     mov    %rsp,%rbp
=> 0x0000555555555131 <+8>:     sub    $0x10,%rsp
   0x0000555555555135 <+12>:    mov    %edi,-0x4(%rbp)
   0x0000555555555138 <+15>:    cmpl   $0x1,-0x4(%rbp)
   0x000055555555513c <+19>:    jne    0x555555555145 <factorial+28>
   0x000055555555513e <+21>:    mov    $0x1,%eax
```

```
(gdb) step
(gdb) step
(gdb) step
factorial (N=0) at factorial.c:4
(gdb) step
(gdb) step
(gdb) step
factorial (N=0) at factorial.c:4
(gdb)
```

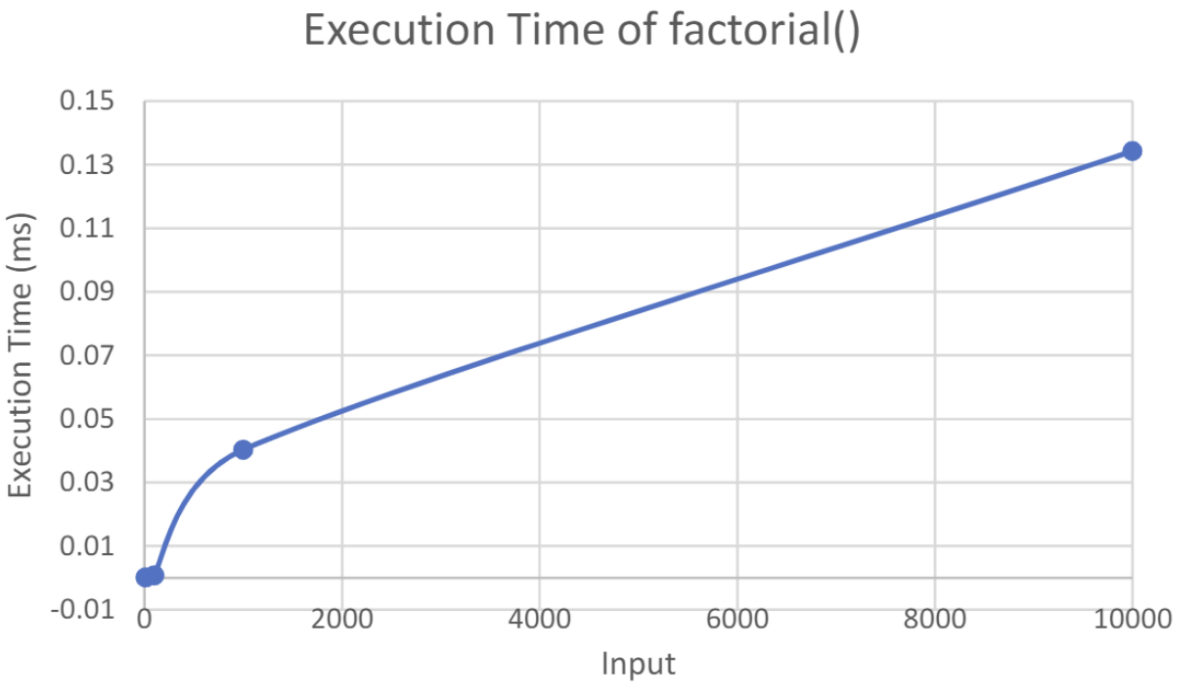
As you can see on the RDI, the value is now equal to 1, indicating that the jump will not occur again. The code will now begin to deallocate memory and return the items it previously stored. It will double the values as it returns them.

```
(gdb) info registers
rax            0x78               120
rbx            0x0                0
rcx            0x0                0
rdx            0x7fffffffdec8     140737488346824
rsi            0x7fffffffdeb8     140737488346808
rdi            0x1                1
rbp            0x7fffffffdd90     0x7fffffffdd90
rsp            0x7fffffffdd80     0x7fffffffdd80
r8             0x7ffff7fa5bd0     140737353767888
r9             0x7ffff7fd9d00     140737353981184
r10            0xfffffffffffffb88 -1144
r11            0x202              514
r12            0x7fffffffdeb8     140737488346808
r13            0x555555555158     93824992235864
r14            0x0                0
r15            0x7ffff7ffbc40     140737354120256
rip            0x55555555516e     0x55555555516e <main+22>
eflags         0x206              [ PF IF ]
cs             0x33               51
ss             0x2b               43
ds             0x0                0
es             0x0                0
fs             0x0                0
gs             0x0                0
```

```
0x000055555555514b <+34>:    mov    %eax,%edi
0x000055555555514d <+36>:    call   0x5555555555129 <factorial>
0x0000555555555152 <+41>:    imul   -0x4(%rbp),%eax
0x0000555555555156 <+45>:    leave
```

This is our final return so as you remember in our visual studio and MIPs the final answer was 120 which is 0x78 in hex and this is the same case for Linux. So RAX is 0x78 which is our final answer. The stack pointer now gets reset to the first stack pointer we had and now all the memory is deallocated so if the user wants the function can restart again. The RBP has been reset as well.

Time comparison:



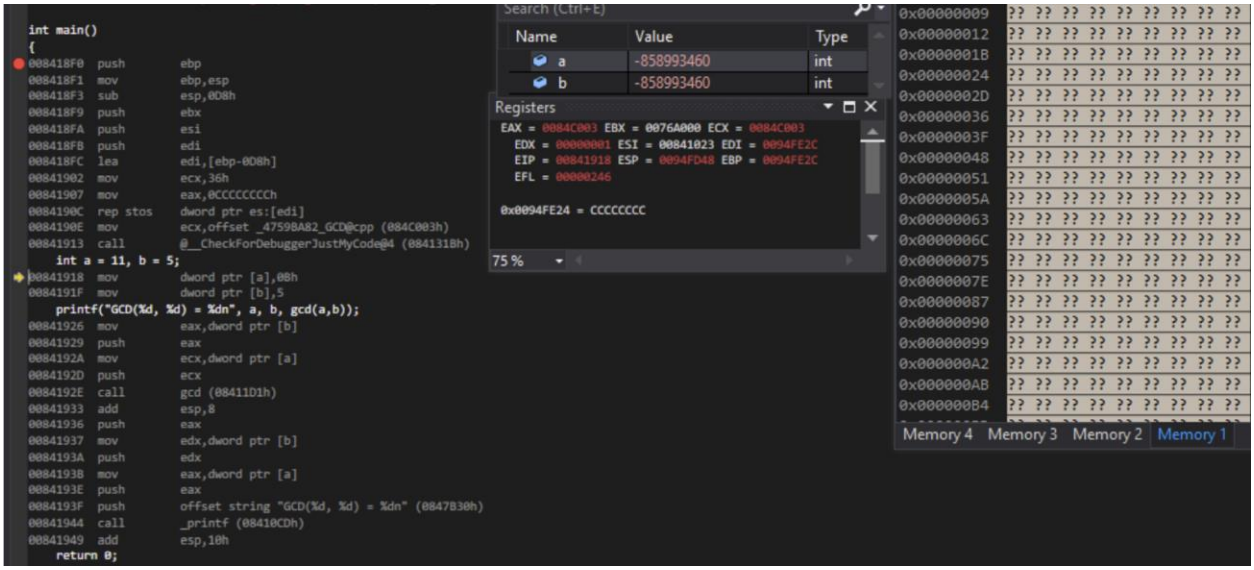
GCD:

Visual Studio:

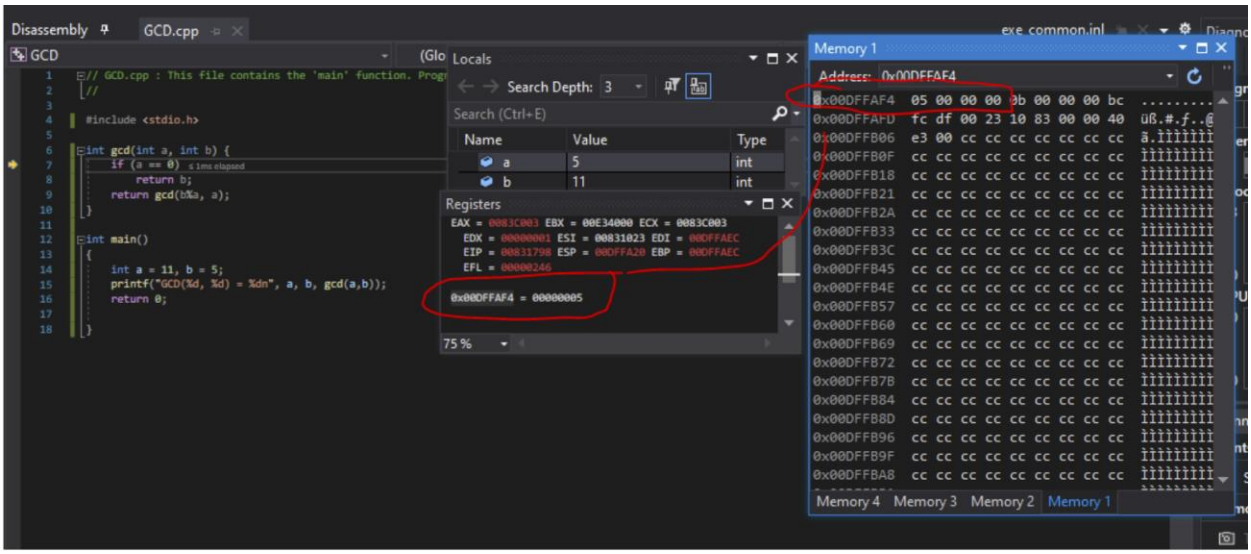
```c
#include <stdio.h>

int gcd(int a, int b) {
    if (a == 0)
        return b;
    return gcd(b % a, a);
}

int main()
{
    int a = 11, b = 5;
    printf("GCD(%d, %d) = %dn", a, b, gcd(a, b));
    return 0;

}
```

This is the visual studio code in CPP for our GCD function. The values we are hardcoding are 11 and 5 and trying to find the GCD of that.

Analysis:



This is before our gcd function starts calling the function and the values are not set yet either.



As you can see, the values 5 and 11 have been placed into the stack using the address 0x00DFFAF4, and our function will then begin dividing. 0x0113FA10 has the value 11 loaded in it. The GCD recursion call will now begin.

The EAX will be the number we are dividing by 5 so then that value gets stored in the register. Also the ECX will count how many times we are running the loop. Our return address is 0x00831770 which is what our code will return after the call is done.

The base case is being checked here and since it is not 0, it is 11 that means it has to go to the GCD call again.

Now the EAX becomes 11 because that is the number, we have to divide with 5 to get a remainder. The stack pointer is pointing to 11 as well.

Now we divide 11 by 5, and the remainder of 1 is saved in argument 1, which is saved in our stack and can be accessed using the stack pointer. This argument will now be divided once more.

Now the first argument does turn to zero and the divisor is 1 so our gcd base condition should work. Our Stack pointer is pointing to 0x0000000. My EBP gets pushed into the stack to return the value.

Now the base pointer returns the calls and their value so we can figure out what is our greatest common divisor.

We get the value of 1 returning and that is our final answer.

Mips:

```
        .data
str1: .asciiz "Enter first integer n1: "
str2: .asciiz "Enter first integer n2: "
str3: .asciiz "The greatest common divisor of n1 and n2 is "
str4: .asciiz "\nThe least common multiple of n1 and n2 is "
newline: .asciiz "\n"
.align 2
.globl main
.text
main:
    la $a0, str1
    li $v0, 4
    syscall
    li $v0, 5
    syscall
    add $a1, $v0, $zero
    la $a0, str2
    li $v0, 4
    syscall
    li $v0, 5
    syscall
    add $a2, $v0, $zero
    addi $sp, $sp, -8
    sw $a1, 4($sp)
    sw $a2, 0($sp)
    jal gcd
    lw $a2, 0($sp)
    lw $a1, 4($sp)
    addi $sp, $sp, 4
    add $s0, $v0, $zero
    sw $s0, 0($sp)
    la $a0, str3
    li $v0, 4
    syscall
    li $v0, 1
```

```
36      add $a0, $s0, $zero
37      syscall
38      la $a0, str4
39      li $v0, 4
40      syscall
41      li $v0, 1
42      add $a0, $s1, $zero
43      syscall
44      la $a0, newline
45      li $v0, 4
46      syscall
47      li $v0, 10
48      syscall
49  gcd:
50      addi $sp, $sp, -8
51      sw $ra, 4($sp)
52      div $a1, $a2
53      mfhi $s0
54      sw $s0, 0($sp)
55      bne $s0, $zero, L1
56      add $v0, $a2, $zero
57      addi $sp, $sp, 8
58      jr $ra
59  L1:
60      add $a1, $a2, $zero
61      lw $s0, 0($sp)
62      add $a2, $s0, $zero
63      jal gcd
64      lw $ra, 4($sp)
65      addi $sp, $sp, 8
66      jr $ra
```

Analysis:

Text Segment

Bkpt	Address	Code	Basic	Sou
	0x00400000	0x3c011001	lui $1,0x00001001	11: la $a0, str1
	0x00400004	0x34240000	ori $4,$1,0x00000000	
	0x00400008	0x24020004	addiu $2,$0,0x00000004	12: li $v0, 4
	0x0040000c	0x0000000c	syscall	13: syscall
	0x00400010	0x24020005	addiu $2,$0,0x00000005	14: li $v0, 5
	0x00400014	0x0000000c	syscall	15: syscall
	0x00400018	0x00402820	add $5,$2,$0	16: add $a1, $v0, $zero
	0x0040001c	0x3c011001	lui $1,0x00001001	17: la $a0, str2
	0x00400020	0x34240019	ori $4,$1,0x00000019	
	0x00400024	0x24020004	addiu $2,$0,0x00000004	18: li $v0, 4
	0x00400028	0x0000000c	syscall	19: syscall
	0x0040002c	0x24020005	addiu $2,$0,0x00000005	20: li $v0, 5
	0x00400030	0x0000000c	syscall	21: syscall
	0x00400034	0x00403020	add $6,$2,$0	22: add $a2, $v0, $zero
	0x00400038	0x23bdfff8	addi $29,$29,0xffff...	23: addi $sp, $sp, -8
	0x0040003c	0xafa50004	sw $5,0x00000004($29)	24: sw $a1, 4($sp)
	0x00400040	0xafa60000	sw $6,0x00000000($29)	25: sw $a2, 0($sp)
	0x00400044	0x0c10002b	jal 0x004000ac	26: jal gcd
	0x00400048	0x8fa60000	lw $6,0x00000000($29)	27: lw $a2, 0($sp)
	0x0040004c	0x8fa50004	lw $5,0x00000004($29)	28: lw $a1, 4($sp)
	0x00400050	0x23bd0004	addi $29,$29,0x0000...	29: addi $sp, $sp, 4
	0x00400054	0x00408020	add $16,$2,$0	30: add $s0, $v0, $zero
	0x00400058	0xafb00000	sw $16,0x00000000($29)	31: sw $s0, 0($sp)
	0x0040005c	0x3c011001	lui $1,0x00001001	32: la $a0, str3
	0x00400060	0x34240032	ori $4,$1,0x00000032	
	0x00400064	0x24020004	addiu $2,$0,0x00000004	33: li $v0, 4
	0x00400068	0x0000000c	syscall	34: syscall
	0x0040006c	0x24020001	addiu $2,$0,0x00000001	35: li $v0, 1
	0x00400070	0x02002020	add $4,$16,$0	36: add $a0, $s0, $zero
	0x00400074	0x0000000c	syscall	37: syscall
	0x00400078	0x3c011001	lui $1,0x00001001	38: la $a0, str4
	0x0040007c	0x3424005f	ori $4,$1,0x0000005f	
	0x00400080	0x24020004	addiu $2,$0,0x00000004	39: li $v0, 4

	0x00400084	0x0000000c	syscall	40:	syscall
	0x00400088	0x24020001	addiu $2,$0,0x00000001	41:	li $v0, 1
	0x0040008c	0x02202020	add $4,$17,$0	42:	add $a0, $s1, $zero
	0x00400090	0x0000000c	syscall	43:	syscall
	0x00400094	0x3c011001	lui $1,0x00001001	44:	la $a0, newline
	0x00400098	0x3424008b	ori $4,$1,0x0000008b		
	0x0040009c	0x24020004	addiu $2,$0,0x00000004	45:	li $v0, 4
	0x004000a0	0x0000000c	syscall	46:	syscall
	0x004000a4	0x2402000a	addiu $2,$0,0x0000000a	47:	li $v0, 10
	0x004000a8	0x0000000c	syscall	48:	syscall
	0x004000ac	0x23bdfff8	addi $29,$29,0xffff...	50:	addi $sp, $sp, -8
	0x004000b0	0xafbf0004	sw $31,0x00000004($29)	51:	sw $ra, 4($sp)
	0x004000b4	0x00a6001a	div $5,$6	52:	div $a1, $a2
	0x004000b8	0x00008010	mfhi $16	53:	mfhi $s0
	0x004000bc	0xafb00000	sw $16,0x00000000($29)	54:	sw $s0, 0($sp)
	0x004000c0	0x16000003	bne $16,$0,0x00000003	55:	bne $s0, $zero, L1
	0x004000c4	0x00c01020	add $2,$6,$0	56:	add $v0, $a2, $zero
	0x004000c8	0x23bd0008	addi $29,$29,0x0000...	57:	addi $sp, $sp, 8
	0x004000cc	0x03e00008	jr $31	58:	jr $ra
	0x004000d0	0x00c02820	add $5,$6,$0	60:	add $a1, $a2, $zero
	0x004000d4	0x8fb00000	lw $16,0x00000000($29)	61:	lw $s0, 0($sp)
	0x004000d8	0x02003020	add $6,$16,$0	62:	add $a2, $s0, $zero
	0x004000dc	0x0c10002b	jal 0x004000ac	63:	jal gcd
	0x004000e0	0x8fbf0004	lw $31,0x00000004($29)	64:	lw $ra, 4($sp)
	0x004000e4	0x23bd0008	addi $29,$29,0x0000...	65:	addi $sp, $sp, 8
	0x004000e8	0x03e00008	jr $31	66:	jr $ra

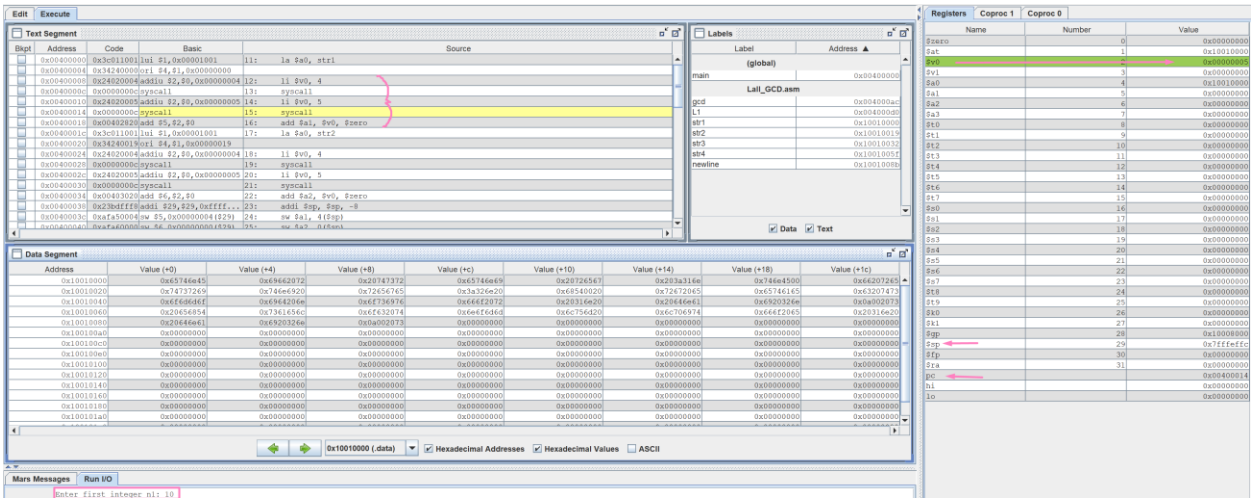
This is the window where all the instructions we provided above gets turned into machine language.

Now let us look at how the registers are changing as we run this code.

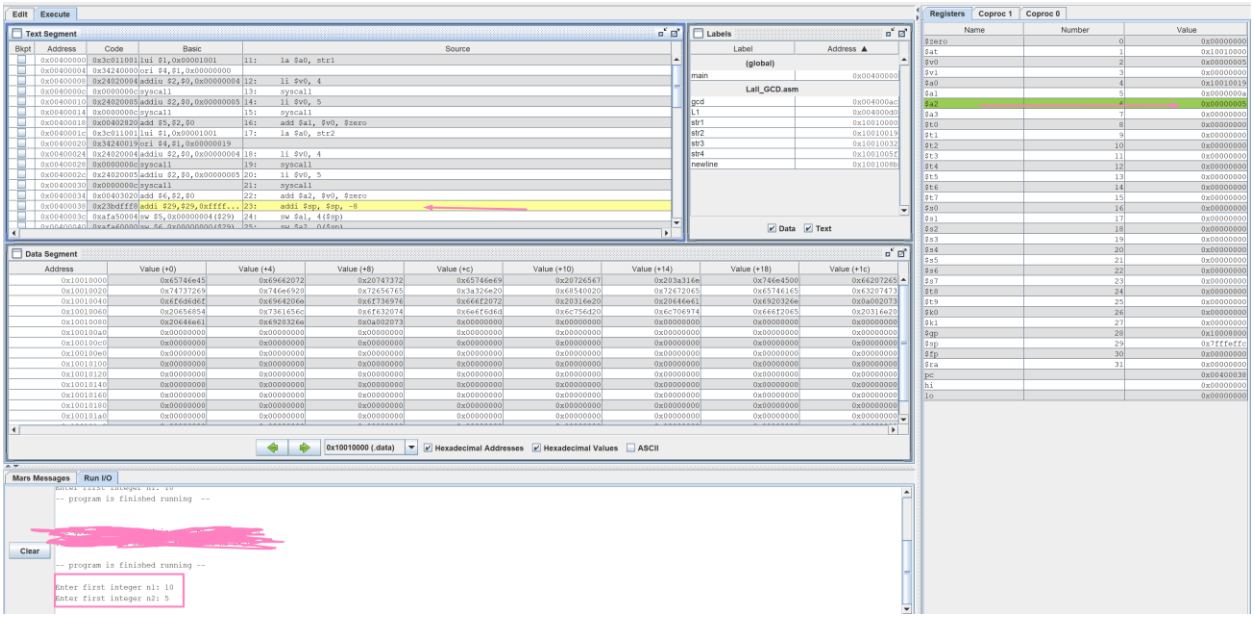
Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000
$t1	9	0x00000000
$t2	10	0x00000000
$t3	11	0x00000000
$t4	12	0x00000000
$t5	13	0x00000000
$t6	14	0x00000000
$t7	15	0x00000000
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00400000
hi		0x00000000
lo		0x00000000

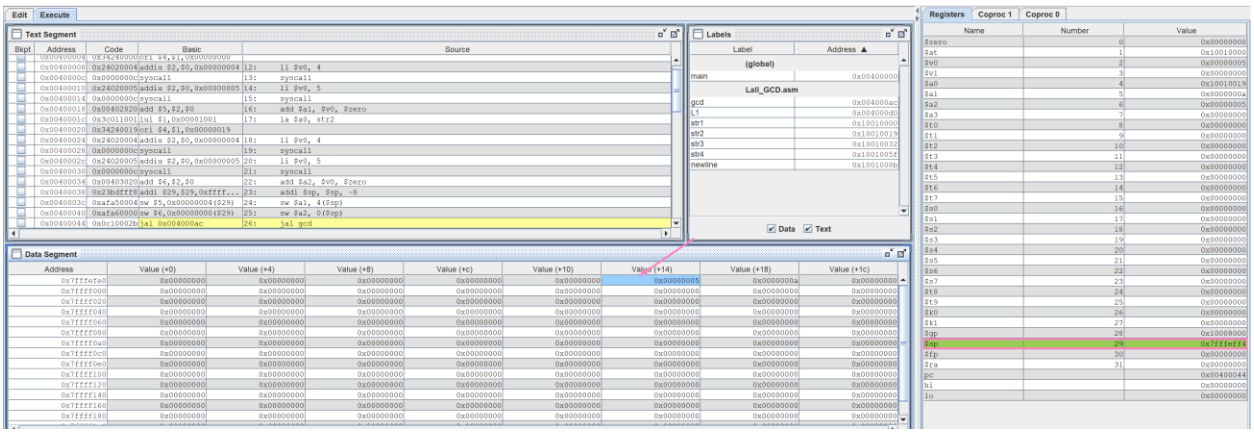
Registers are zero since we have not started yet.



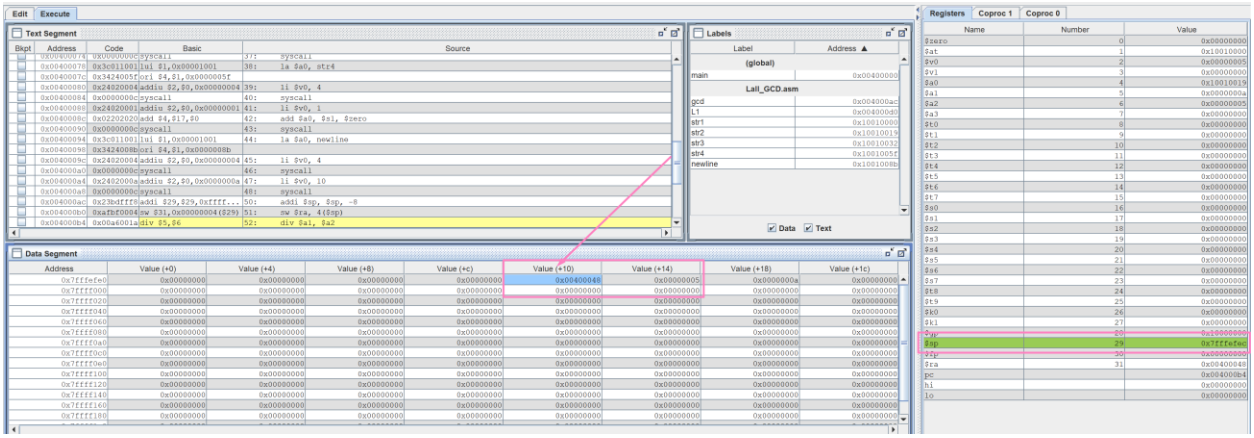
The code is asking us for first input which will be 10. As you can see it is loading 5 on to the register v0 but what this 5 is just the input/output value for a string output which we will be showing the result with at the very end of this code.



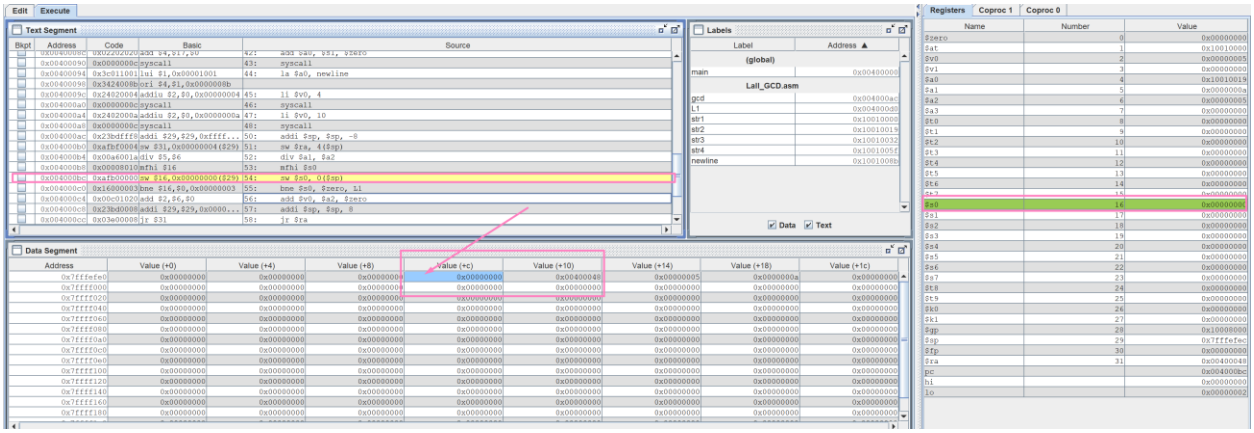
As you can see, our two values of 10 and 5 were saved to $a1 and $a0, respectively, from which we can calculate the GCD. Now it will make room in the stack for these values, but it will be saved with a value of -8 from the stack pointer.



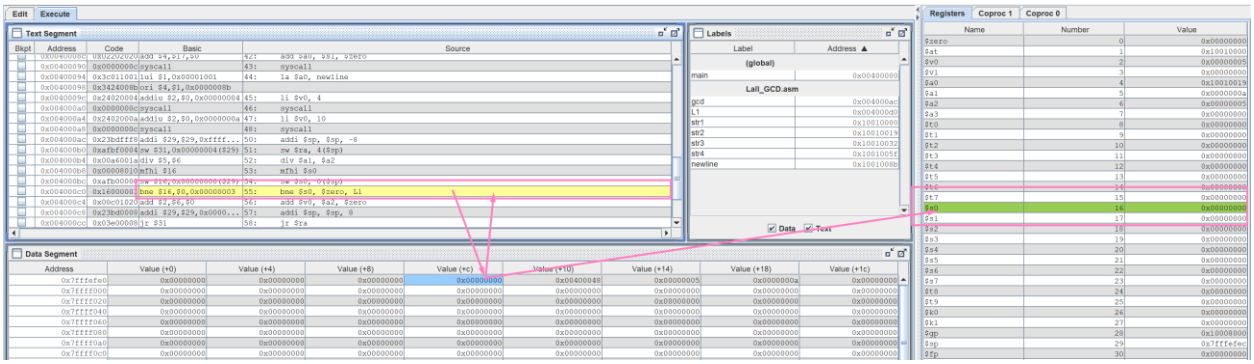
As you can see the two values are not stored in the stack with -8 offset from the stack pointer.



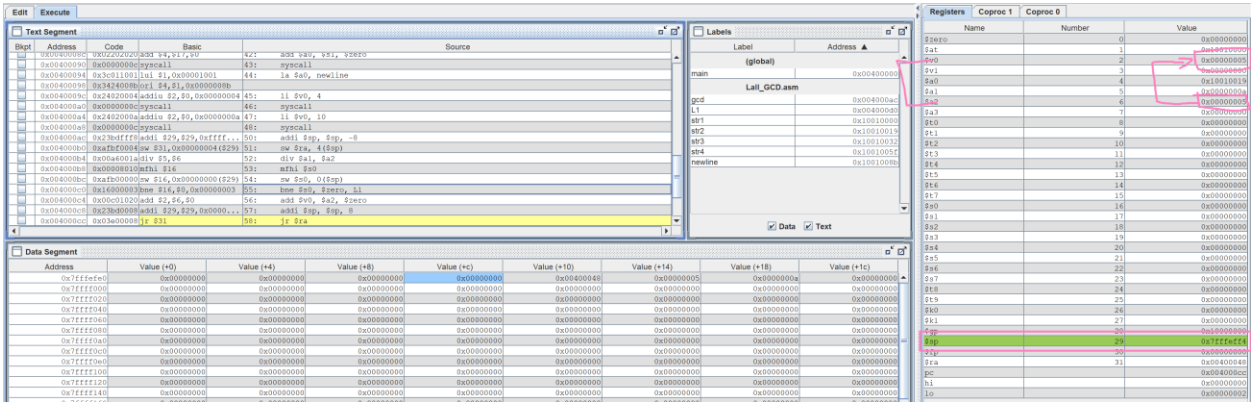
The blue highlighted box is the return address after the GCD function is done calling. Our return address is 0x00400048.



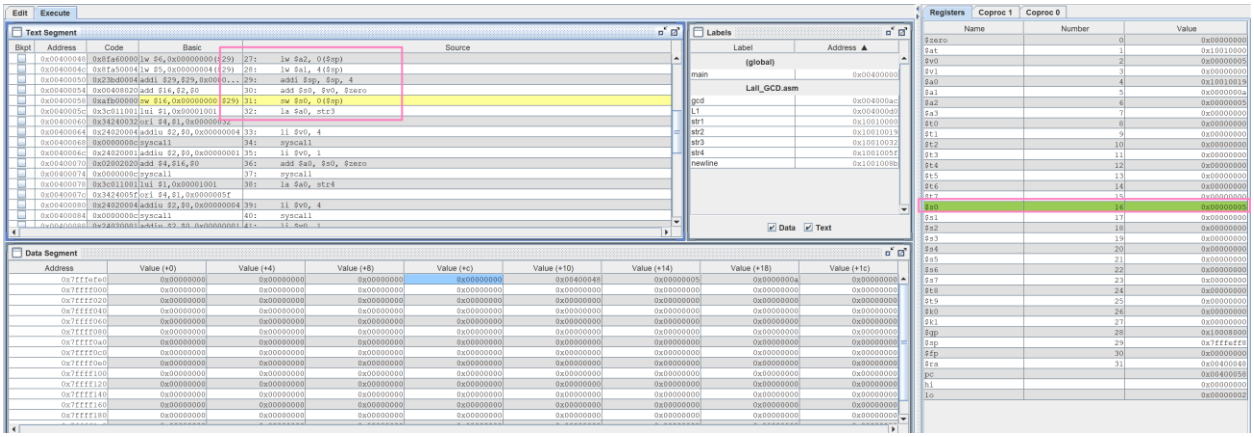
Now the code will divide the two number is at register $s0 it will hold the remainder of the division. Since 10 / 5 = 2 and the remainder is 0 the value of 0x0000000 is shown in the green highlighted box in the registers section.



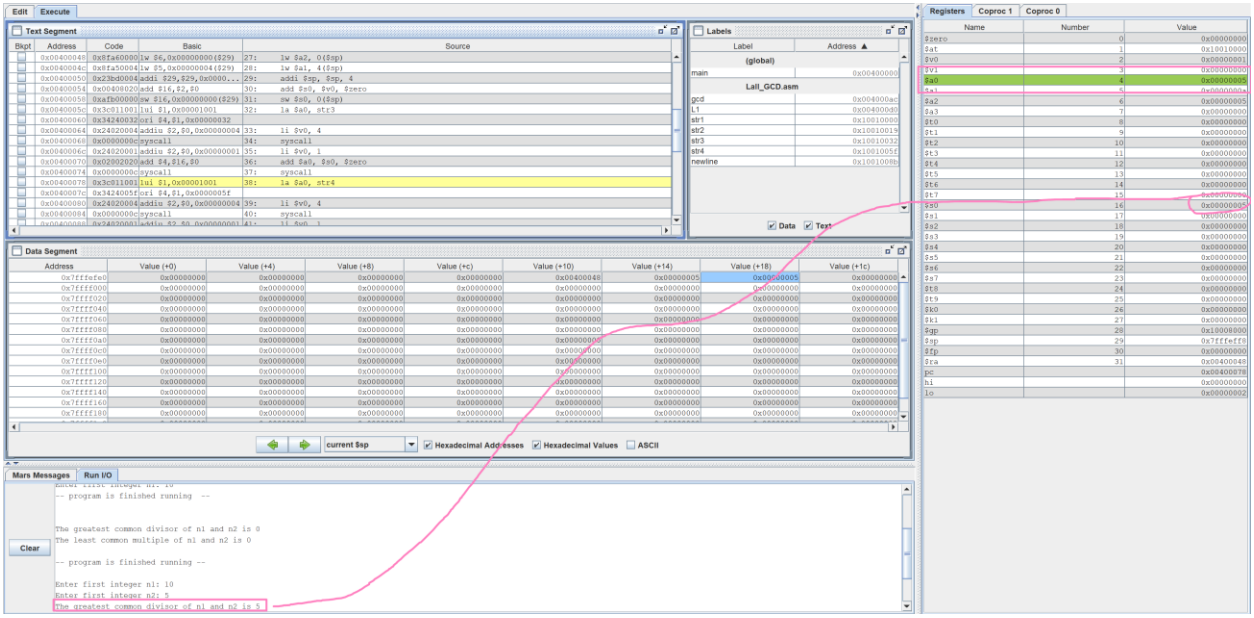
Now our code will verify the recursive call condition, which is if $s0 is equal to 0, then continue; otherwise, continuing invoking the gcd function and dividing the two arguments recursively until you have the remainder of 0. Later on, I'll provide another recursive call example. However, because it is 0, the code will continue following the BNE line for the time being.



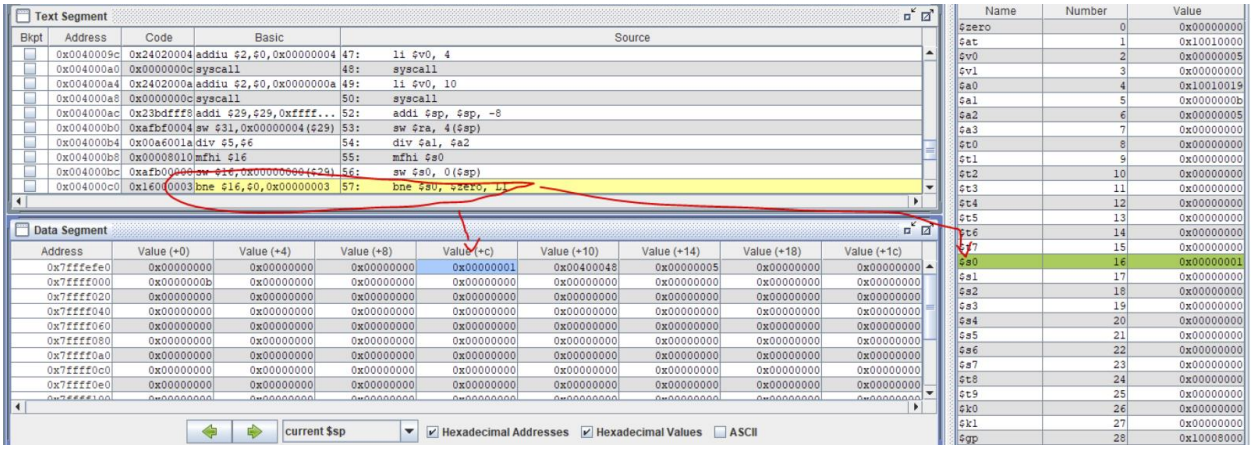
Now value of register a1 goes to v0 register and then we call our return address through the jump instruction.



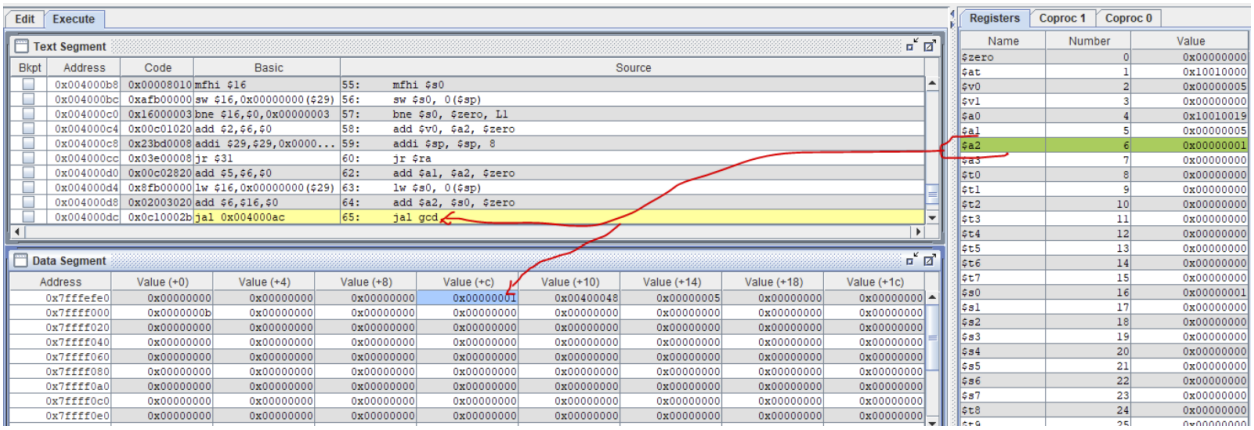
Now the code is loading a1 and a2 and then popping one value from the stack and then at the register $s0 it is storing the result of our GCD which in this case is 5.



Now as you can see the output of our result is 5. If the BNE did not equal to 0 then the extra calls that we would have to make is demonstrated below:



With inputs 11 and 5 there is a remainder of 1 which is saved into our stack as the value of 0x0000001. Now instead of returning since the bne condition does not meet it will go to the second step in the GCD recursive call.



Now register $a2 changes its value to the remainder and the whole process of GCD will start again to return the answer. This is how the recursive call is done in MIPS FOR GCD.

Linux GDB:

The code for this section is the same as our CPP code except in Linux I made it a c file.

```c
#include <stdio.h>

int gcd(int a, int b) {
    if (a == 0)
        return b;
    return gcd(b % a, a);
}

int main()
{
    int a = 11, b = 5;
    printf("GCD(%d, %d) = %dn", a, b, gcd(a, b));
    return 0;

}
```

Analysis:

```
Dump of assembler code for function gcd:
   0x0000555555555149 <+0>:     endbr64
   0x000055555555514d <+4>:     push   %rbp
   0x000055555555514e <+5>:     mov    %rsp,%rbp
=> 0x0000555555555151 <+8>:     sub    $0x10,%rsp
   0x0000555555555155 <+12>:    mov    %edi,-0x4(%rbp)
   0x0000555555555158 <+15>:    mov    %esi,-0x8(%rbp)
   0x000055555555515b <+18>:    cmpl   $0x0,-0x4(%rbp)
   0x000055555555515f <+22>:    jne    0x555555555166 <gcd+29>
   0x0000555555555161 <+24>:    mov    -0x8(%rbp),%eax
   0x0000555555555164 <+27>:    jmp    0x555555555179 <gcd+48>
   0x0000555555555166 <+29>:    mov    -0x8(%rbp),%eax
   0x0000555555555169 <+32>:    cltd
   0x000055555555516a <+33>:    idivl  -0x4(%rbp)
   0x000055555555516d <+36>:    mov    -0x4(%rbp),%eax
   0x0000555555555170 <+39>:    mov    %eax,%esi
   0x0000555555555172 <+41>:    mov    %edx,%edi
   0x0000555555555174 <+43>:    call   0x555555555149 <gcd>
   0x0000555555555179 <+48>:    leave
   0x000055555555517a <+49>:    ret
End of assembler dump.
```

```
(gdb) disassemble
Dump of assembler code for function main:
   0x000055555555517b <+0>:     endbr64
   0x000055555555517f <+4>:     push   %rbp
   0x0000555555555180 <+5>:     mov    %rsp,%rbp
=> 0x0000555555555183 <+8>:     sub    $0x10,%rsp
   0x0000555555555187 <+12>:    movl   $0xb,-0x8(%rbp)
   0x000055555555518e <+19>:    movl   $0x5,-0x4(%rbp)
   0x0000555555555195 <+26>:    mov    -0x4(%rbp),%edx
   0x0000555555555198 <+29>:    mov    -0x8(%rbp),%eax
   0x000055555555519b <+32>:    mov    %edx,%esi
   0x000055555555519d <+34>:    mov    %eax,%edi
   0x000055555555519f <+36>:    call   0x555555555149 <gcd>
   0x00005555555551a4 <+41>:    mov    %eax,%ecx
   0x00005555555551a6 <+43>:    mov    -0x4(%rbp),%edx
   0x00005555555551a9 <+46>:    mov    -0x8(%rbp),%eax
   0x00005555555551ac <+49>:    mov    %eax,%esi
   0x00005555555551ae <+51>:    lea    0xe4f(%rip),%rax        # 0x555555556004
   0x00005555555551b5 <+58>:    mov    %rax,%rdi
   0x00005555555551b8 <+61>:    mov    $0x0,%eax
   0x00005555555551bd <+66>:    call   0x555555555050 <printf@plt>
   0x00005555555551c2 <+71>:    mov    $0x0,%eax
   0x00005555555551c7 <+76>:    leave
   0x00005555555551c8 <+77>:    ret
```

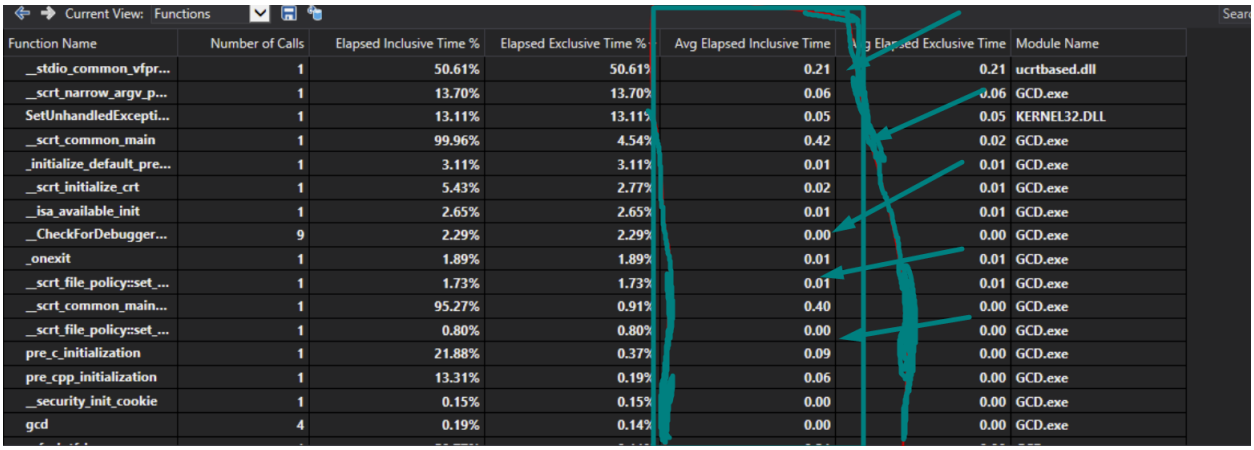
These are the disassemble code on GDB. The base pointer will get our two values of 5 and 11.

As you can see the registers RSI and RDI holds the values we hard coded in our code. The register RAX holds the dividend and RDX holds the number to divide by.

Now the function compares the two values to see if we actually get the base case which is to test to see if argument 1 is equal to zero and since it does not equal to zero it will recursively call the GCD function

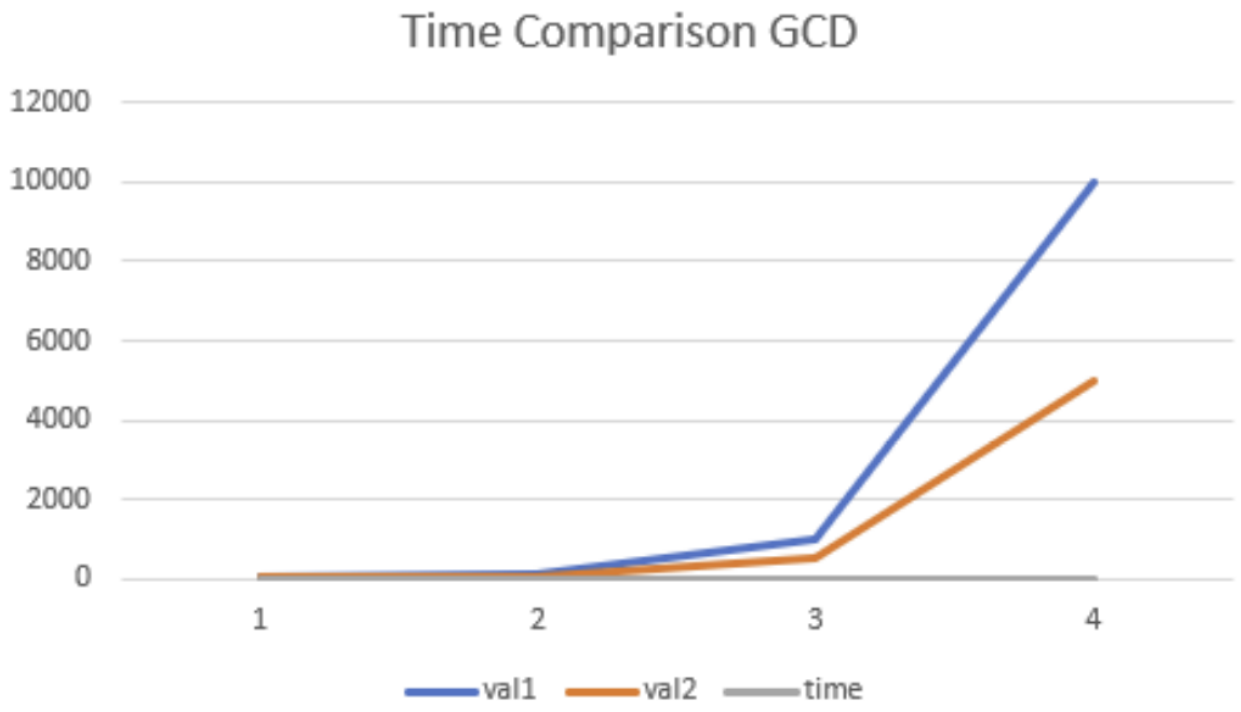
After running the second comparison as you can see one of the registers does become 0 so that means our base case is now satisfied, thus our code can start returning the answer.

Time Comparison:



Current View: Functions

Function Name	Number of Calls	Elapsed Inclusive Time %	Elapsed Exclusive Time %	Avg Elapsed Inclusive Time	g Elapsed Exclusive Time	Module Name
__stdio_common_vfpr...	1	50.61%	50.61%	0.21	0.21	ucrtbased.dll
__scrt_narrow_argv_p...	1	13.70%	13.70%	0.06	0.06	GCD.exe
SetUnhandledExcepti...	1	13.11%	13.11%	0.05	0.05	KERNEL32.DLL
__scrt_common_main	1	99.96%	4.54%	0.42	0.02	GCD.exe
_initialize_default_pre...	1	3.11%	3.11%	0.01	0.01	GCD.exe
__scrt_initialize_crt	1	5.43%	2.77%	0.02	0.01	GCD.exe
__isa_available_init	1	2.65%	2.65%	0.01	0.01	GCD.exe
__CheckForDebugger...	9	2.29%	2.29%	0.00	0.00	GCD.exe
_onexit	1	1.89%	1.89%	0.01	0.01	GCD.exe
__scrt_file_policy::set_...	1	1.73%	1.73%	0.01	0.01	GCD.exe
__scrt_common_main...	1	95.27%	0.91%	0.40	0.00	GCD.exe
__scrt_file_policy::set_...	1	0.80%	0.80%	0.00	0.00	GCD.exe
pre_c_initialization	1	21.88%	0.37%	0.09	0.00	GCD.exe
pre_cpp_initialization	1	13.31%	0.19%	0.06	0.00	GCD.exe
__security_init_cookie	1	0.15%	0.15%	0.00	0.00	GCD.exe
gcd	4	0.19%	0.14%	0.00	0.00	GCD.exe



Conclusion:

This lab helped me grasp how three different systems handle recursive functions. In MARS, VS, and Linux GDB, I was able to simulate the test and create time comparisons based on performance and functionality. I was able to see how the registers save and return values in which they store data in addresses for the code.