

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 receive unauthorized assistance on this TEST. I will only use the computing device to perform this TEST, I will not use cell while performing this TEST.

Chris J

## Table of Contents

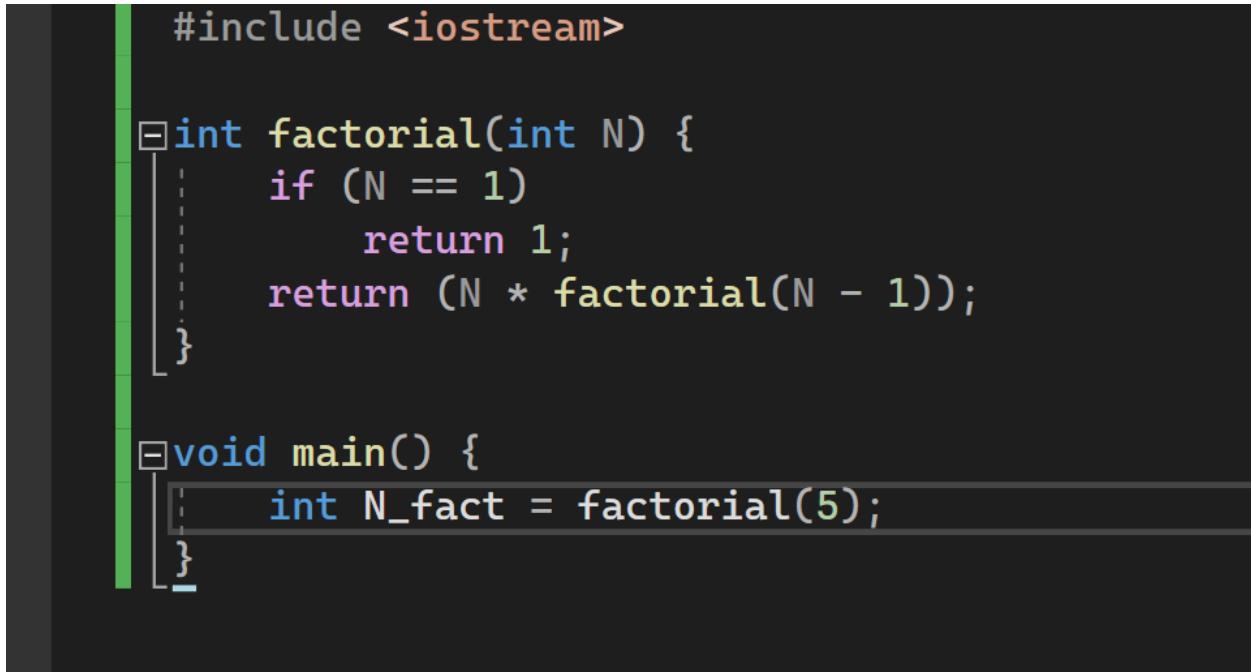
Objective: .....	3
Factorial: .....	4
Visual Studio: .....	4
Analysis: .....	5
MIPS: .....	17
Analysis: .....	18
LINUX GDB: .....	32
Analysis: .....	32
Time comparison: .....	38
GCD: .....	39
Visual Studio: .....	39
Analysis: .....	40
Mips: .....	42
Analysis: .....	44
Linux GDB: .....	52
Analysis: .....	53
Time Comparison: .....	54
Conclusion: .....	55

## 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);
}
```

A screenshot of a Visual Studio code editor window. The code is written in C++. It includes an include directive for iostream, a recursive function named factorial that takes an integer N and returns its factorial value, and a main function that calls factorial(5). The code is syntax-highlighted with colors for different language elements.

Figure 1. factorial code

Analysis:

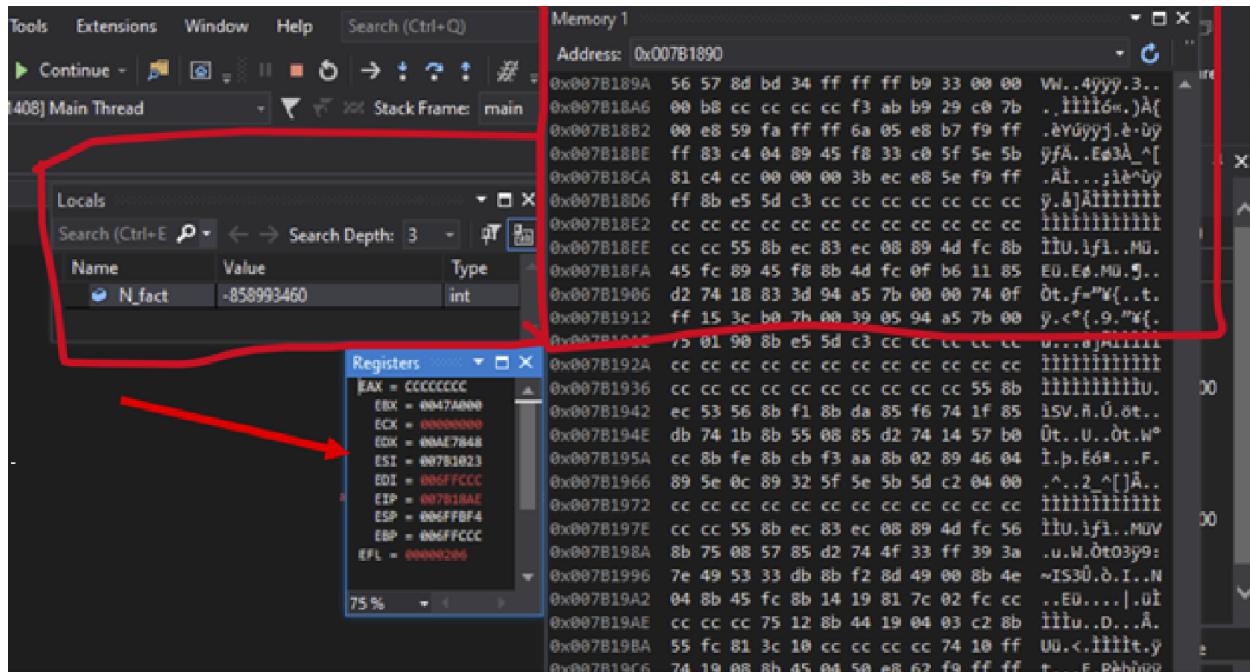
```
void main()
{
007B1890 push    ebp
007B1891 mov     ebp,esp
007B1893 sub    esp,0CCh
007B1899 push    ebx
007B189A push    esi
007B189B push    edi
007B189C lea     edi,[ebp-0CCh]
007B18A2 mov     ecx,33h
007B18A7 mov     eax,0CCCCCCCCh
007B18AC rep stos dword ptr es:[edi]
007B18AE mov     ecx,offset
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
```

```

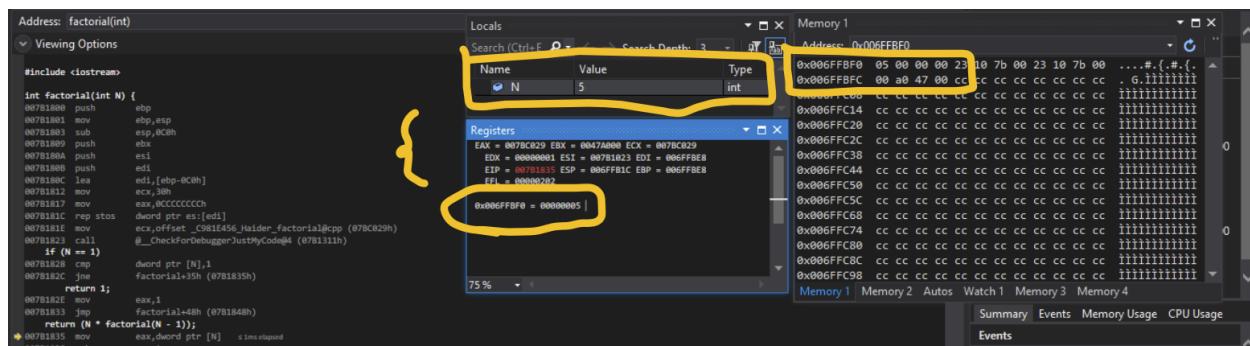
int factorial(int N) {
007B1800 push    ebp
007B1801 mov     ebp,esp
007B1803 sub    esp,0C0h
007B1809 push    ebx
007B180A push    esi
007B180B push    edi
007B180C lea     edi,[ebp-0C0h]
007B1812 mov     ecx,30h
007B1817 mov     eax,0CCCCCCCCCh
007B181C rep stos dword ptr es:[edi]
007B181E mov     ecx,offset _
007B1823 call    @_CheckForDebuggerJustMyCode@4 (07B1311h)
    if (N == 1)
007B1828 cmp     dword ptr [N],1
007B182C jne    factorial+35h (07B1835h)
        return 1;
007B182E mov     eax,1
007B1833 jmp    factorial+48h (07B1848h)
        return (N * factorial(N - 1));
007B1835 mov     eax,dword ptr [N]
007B1838 sub     eax,1
007B183B push    eax
007B183C call    factorial (07B1276h)
007B1841 add     esp,4
007B1844 imul   eax,dword ptr [N]
}
007B1848 pop    edi
007B1849 pop    esi
007B184A pop    ebx
007B184B add     esp,0C0h
007B1851 cmp     ebp,esp
007B1853 call    __RTC_CheckEsp (07B1235h)
007B1858 mov     esp,ebp
007B185A pop    ebp

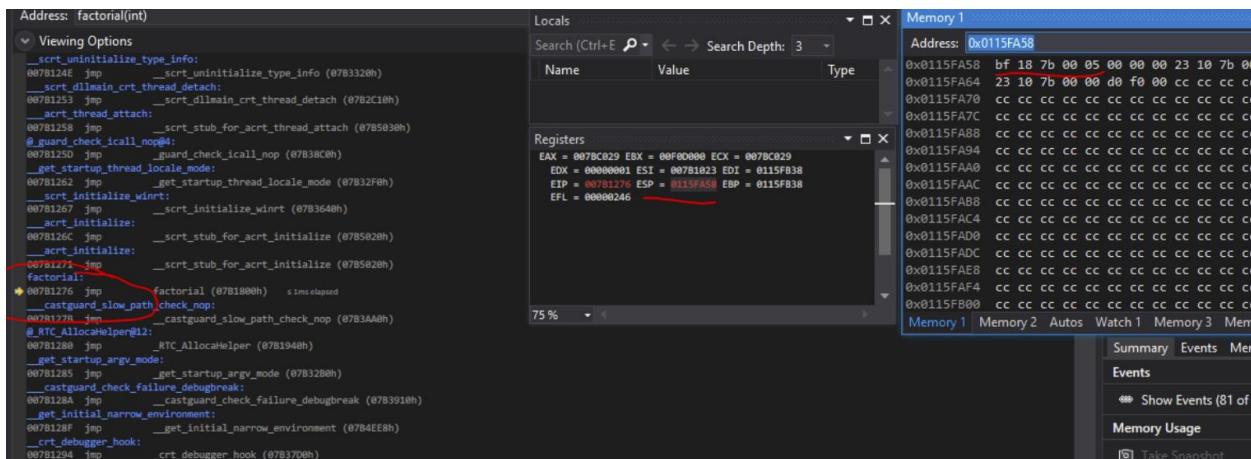
```

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 006FFC00. 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.

Address: factorial(int)

Viewing Options

```
#include <iostream>

int factorial(int N) {
    push    ebp
    mov     ebp,esp
    sub    esp,0C0h
    push    ebx
    push    esi
    push    edi
    lea     edi,[ebp-0C0h]
    mov     ecx,30h
    mov     eax,0CCCCCCC
    rep    stos
    mov     edx,offset factorial+4
    call   @__CheckForOverflowUnderflow
    if (N == 1)
        cmp    N,1
        jne    factorial+14h
        mov    eax,1
        jmp    factorial+14h
    return (N * factorial(N - 1))
    mov    eax,dword PTR [N]
    sub    eax,1
    push    eax
    call   factorial
    add    esp,4
    imul   eax,dword PTR [N]
    pop    edi
    pop    esi
    pop    ebx
    add    esp,0C0h
    cmp    ebp,esp
}
```

Locals

Name	Value	Type
N	5	int

Registers

Name	Value
EAX	007BC029
EBX	00F0D000
ECX	007BC029
EDX	00000001
ESI	007B1023
EDI	0115FA54
EIP	007B1828
ESP	0115F988
EBP	0115FA54
ECR	00000246

Memory 1

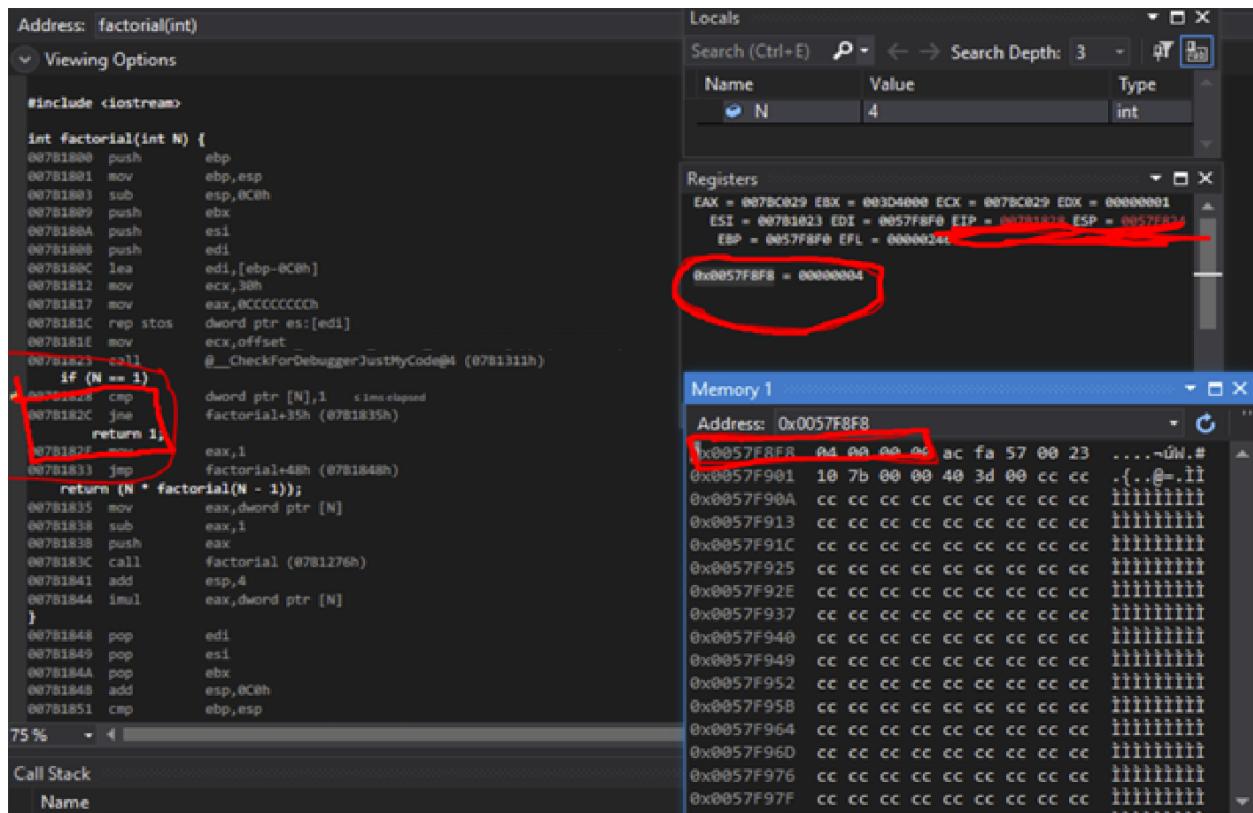
Address	Value
0x0115F988	38 fb 15 01 23 10 7b 00 00 80 ..{..
0x0115F991	d0 f0 00 cc cc cc cc cc cc cc ..
0x0115F99A	cc cc cc cc cc cc cc cc cc ..
0x0115F9A3	cc cc cc cc cc cc cc cc ..
0x0115F9AC	cc cc cc cc cc cc cc ..
0x0115F9B5	cc cc cc cc cc cc cc ..
0x0115F9BE	cc cc cc cc cc cc ..
0x0115F9C7	cc cc cc cc cc cc ..
0x0115F9D0	cc cc cc cc cc cc ..
0x0115F9D9	cc cc cc cc cc cc ..
0x0115F9E2	cc cc cc cc cc cc ..
0x0115F9EB	cc cc cc cc cc cc ..
0x0115F9F4	cc cc cc cc cc cc ..
0x0115F9FD	cc cc cc cc cc cc ..
0x0115FA06	cc cc cc cc cc cc ..

Call Stack

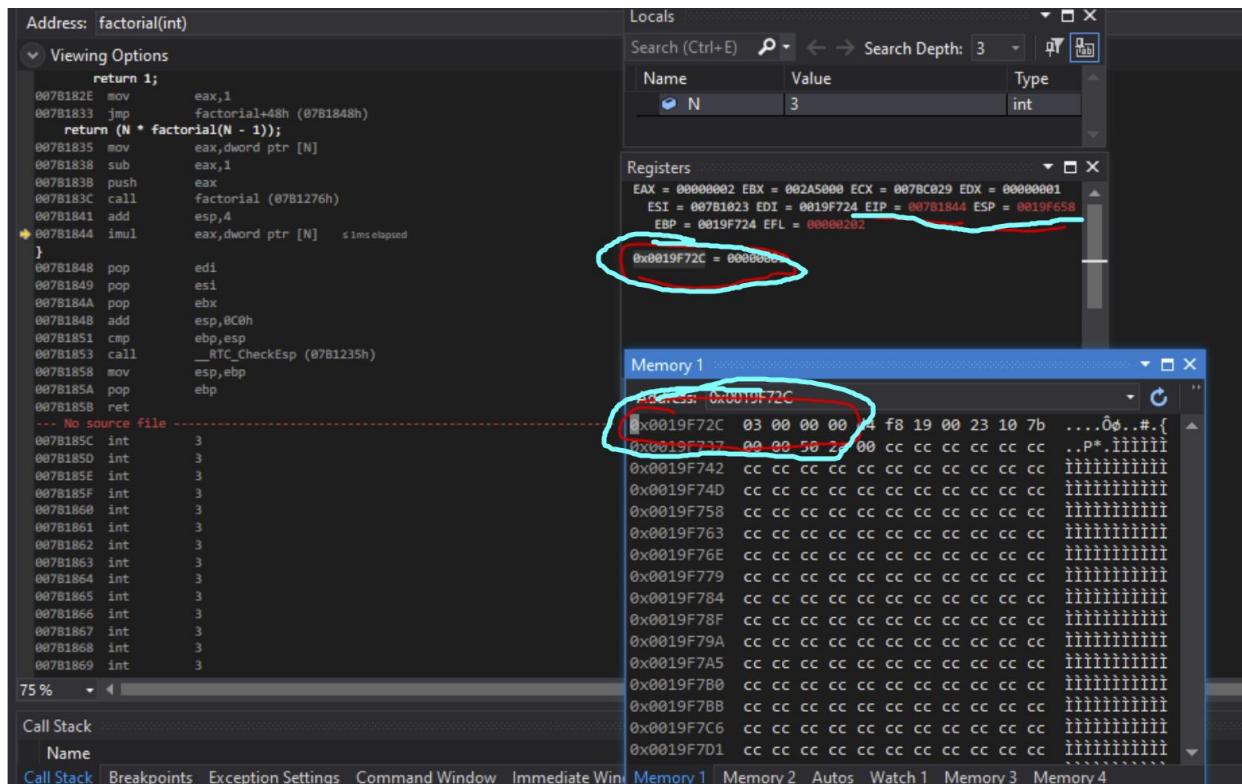
Name
Call Stack
Breakpoints
Exception

Memory 1 Memory 2 Autos Watch 1 Memory 3 Memory 4

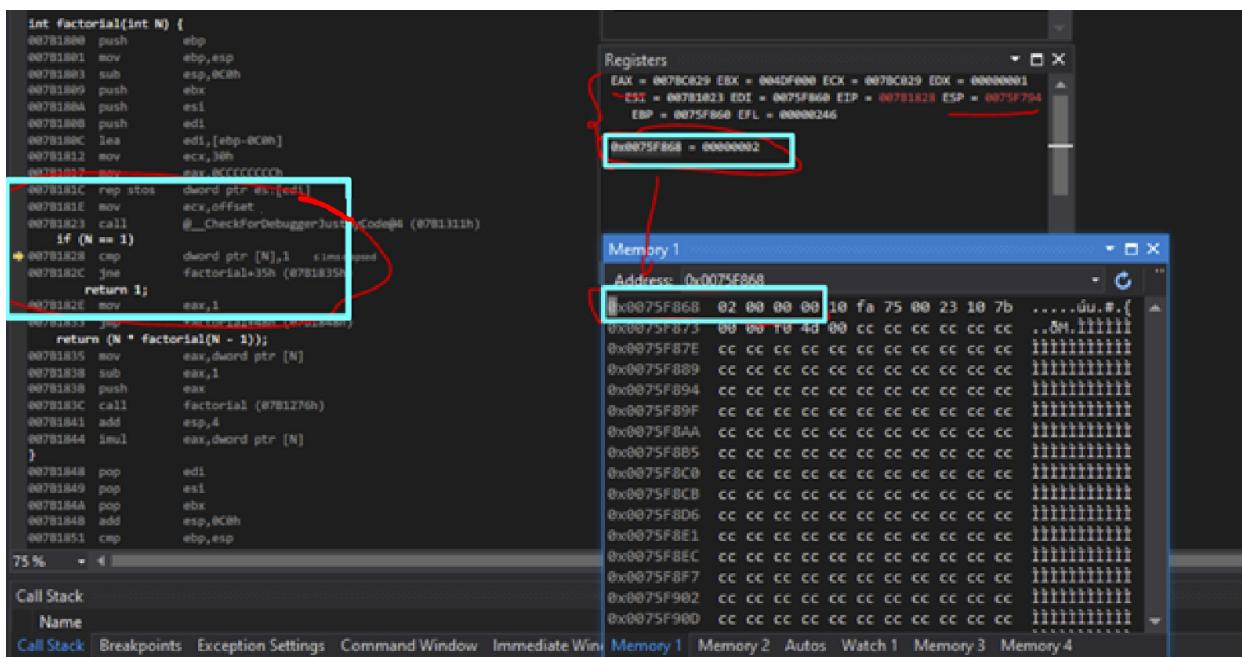
The base pointer is now pushed into the stack. The new stack frame is EBP value of 0x0115FA54. Now the code will check if our initial value is 1 and if it not then it will perform the jump instruction.



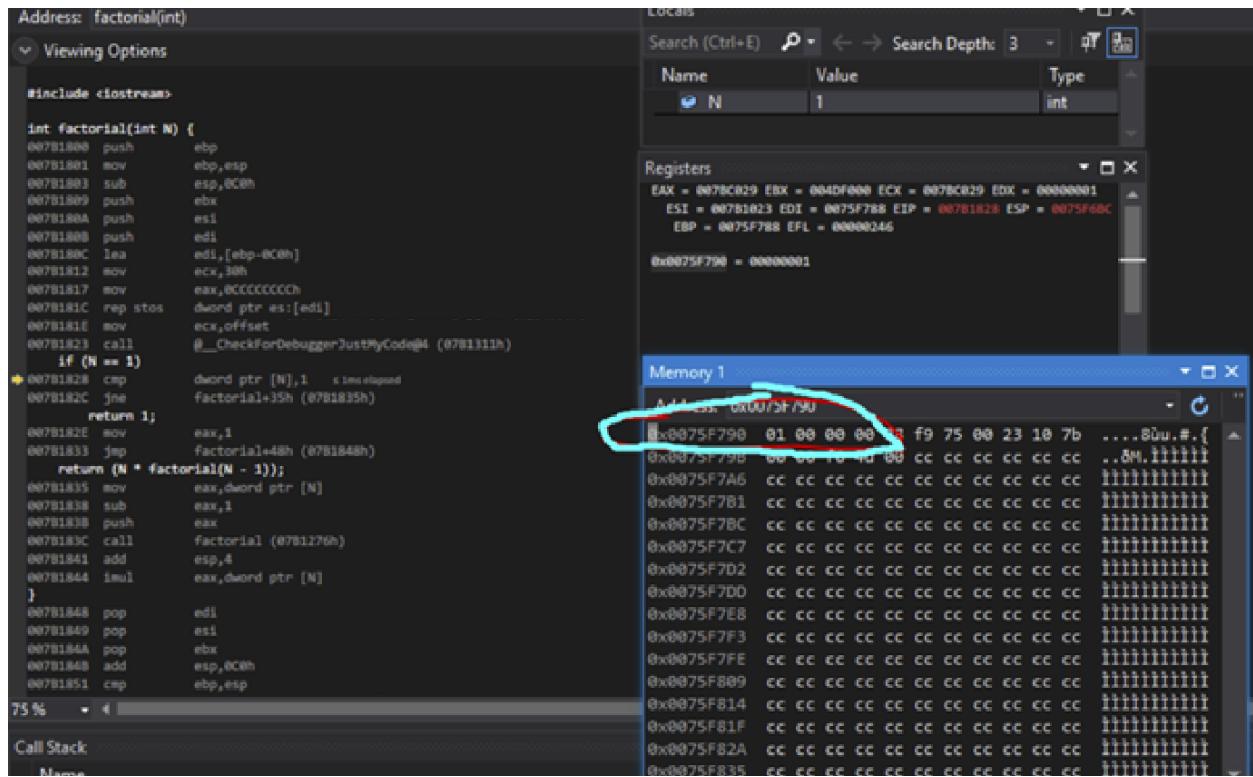
The value of our local variable `N` is reduced by one, to `4`; this value is saved in the `EAX` register as `0x00000004`. `0x0057F8F8` is the address where this is saved. Because this value was copied from the `EAX` registers, the `EAX` currently has a different value. The value `0x0057F9C8` is included 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.



Address: factorial(int)

Viewing Options

```
#include <iostream>

int factorial(int N) {
    007B1800 push    ebp
    007B1801 mov     ebp,esp
    007B1803 sub     esp,0C8h
    007B1809 push    ebx
    007B180A push    esi
    007B180B push    edi
    007B180C lea     edi,[ebp-0C8h]
    007B1812 mov     ecx,30h
    007B1817 mov     eax,0CCCCCCCCh
    007B181C rep stos    dword ptr es:[edi]
    007B181E mov     ecx,offset
    007B1823 call    @_CheckForDebuggerJustMyCode@4 (07B1311h)
    if (N == 1)
    * 007B1828 cmp     dword ptr [N],1    ; loc_1828
    007B182C jne     Factorial+35h (07B1835h)
        return 1;
    007B182E mov     eax,1
    007B1833 jmp     Factorial+48h (07B1848h)
        return (N * factorial(N - 1));
    007B1835 mov     eax,dword ptr [N]
    007B1838 sub     eax,1
    007B183B push    eax
    007B183C call    factorial (07B1276h)
    007B1841 add     esp,4
    007B1844 imul   eax,dword ptr [N]
}
007B1848 pop     edi
007B1849 pop     esi
007B184A pop     ebx
007B184B add     esp,0C8h
007B1851 cmp     ebp,esp
75% 4 Call Stack
    Names
```

Locals

Name	Value	Type
N	1	int

Registers

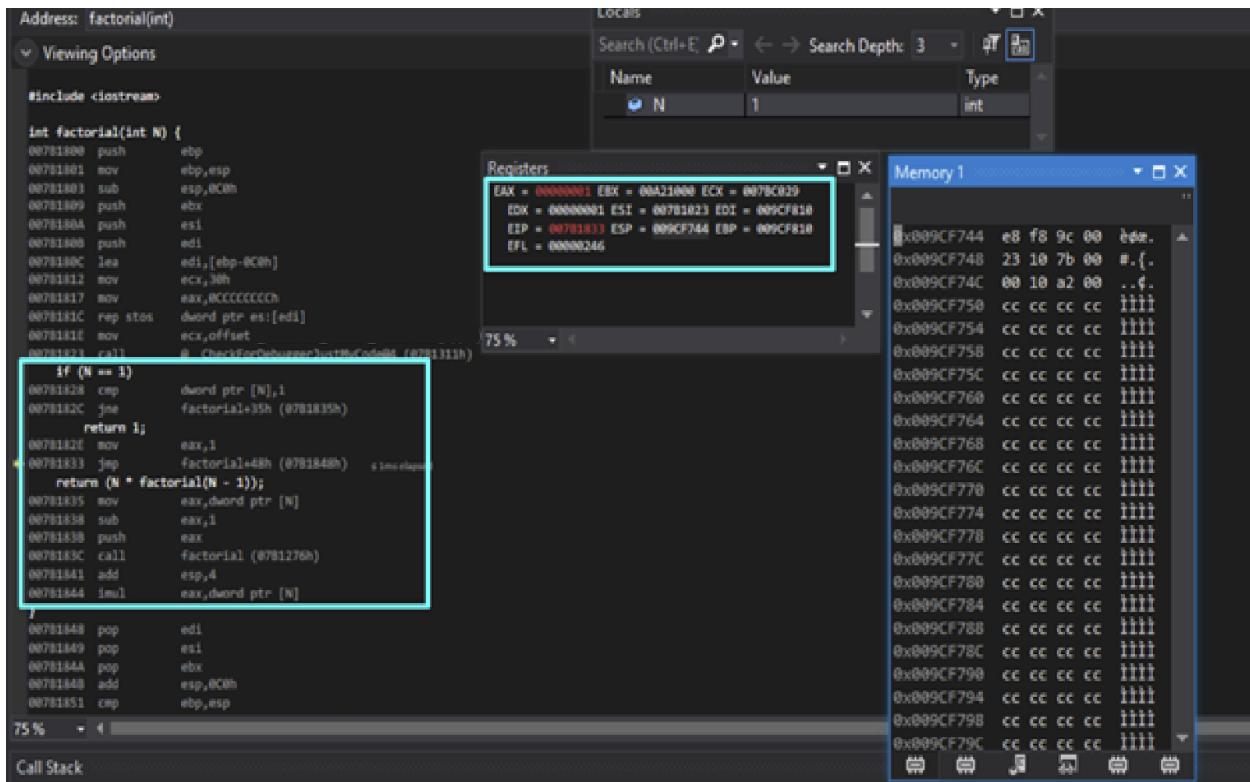
Name	Value
EAX	007BC029
EBX	004DF000
ECX	007BC029
EDX	00000001
ESI	007B1823
EDI	0075F788
EIP	007B1828
ESP	0075F60C
EBP	0075F788
EFL	000000246

Memory 1

Address	Value
0x0075F798	01 00 00 00 f9 75 00 23 10 7b ....80u.{
0x0075F79B	00 00 00 00 cc cc cc cc cc cc ..0M.iiiiii
0x0075F7A6	cc iiiiiiiiii
0x0075F7B1	cc iiiiiiiiii
0x0075F7B8	cc cc cc cc ee ee ee ee ee ee iiiiiiiiii
0x0075F7C7	cc iiiiiiiiii
0x0075F7D2	cc iiiiiiiiii
0x0075F7D9	cc iiiiiiiiii
0x0075F7E8	cc iiiiiiiiii
0x0075F7F3	cc iiiiiiiiii
0x0075F7FE	cc cc cc ee ee ee ee ee ee ee iiiiiiiiii
0x0075F809	cc cc cc ee ee ee ee ee ee ee iiiiiiiiii
0x0075F814	cc cc ee ee ee ee ee ee ee iiiiiiiiii
0x0075F81F	cc cc ee ee ee ee ee ee ee iiiiiiiiii
0x0075F82A	cc cc ee ee ee ee ee ee ee iiiiiiiiii
0x0075F835	cc cc ee ee ee ee ee ee ee iiiiiiiiii

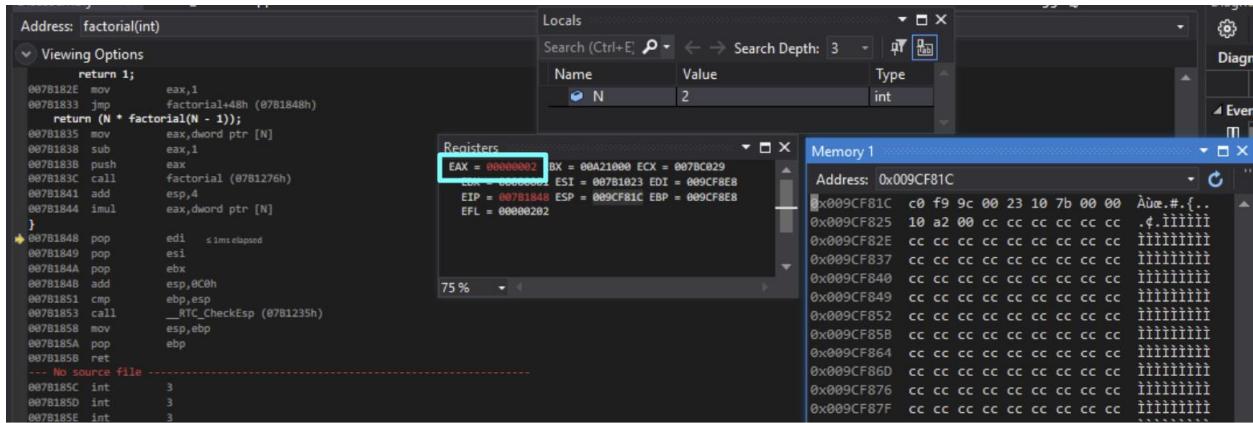
And now the value will get stored to the local variable again as well the EPI,ESP will change too. Once the value reaches 1 a jump will occur to instruction 0x00F8170E. This will end the function.

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.



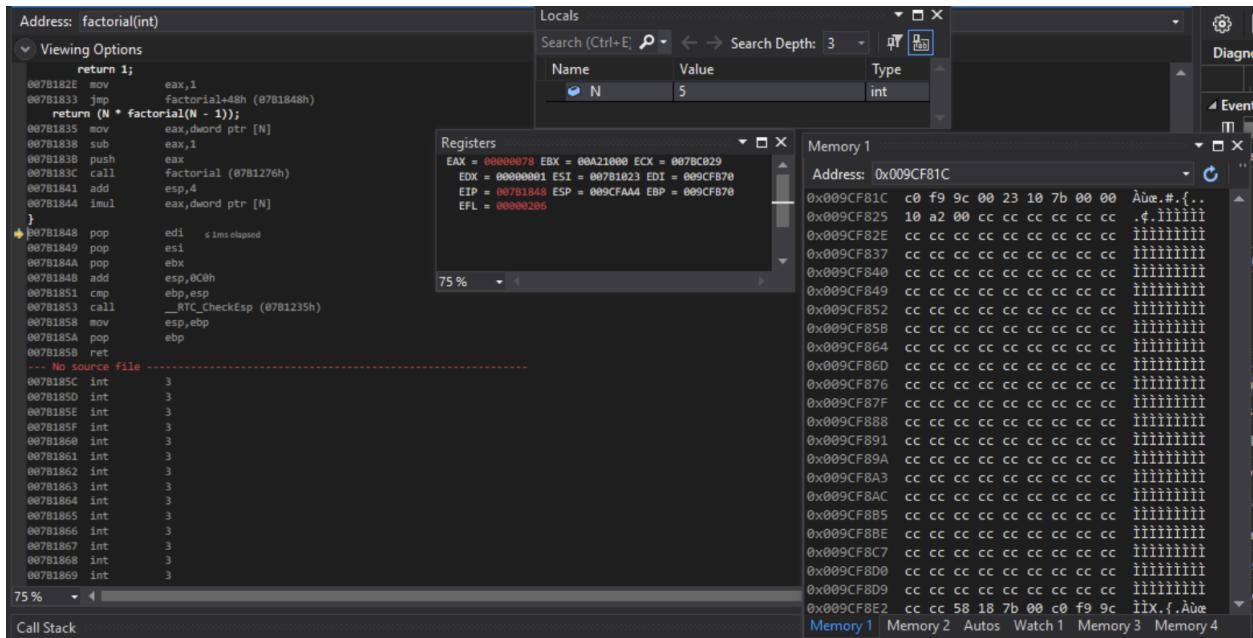
The stack frame is reset to the base pointer, which is the value of our EBP. The return address will now be 0x009cf8e8, which is the address at the top of the stack. The code returns to the base pointer after the return instruction is completed, and the return address is popped off the stack and deallocated.

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 0x00000078 which is 120 in

decimal value.



MIPS:

```
Lcall_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
```

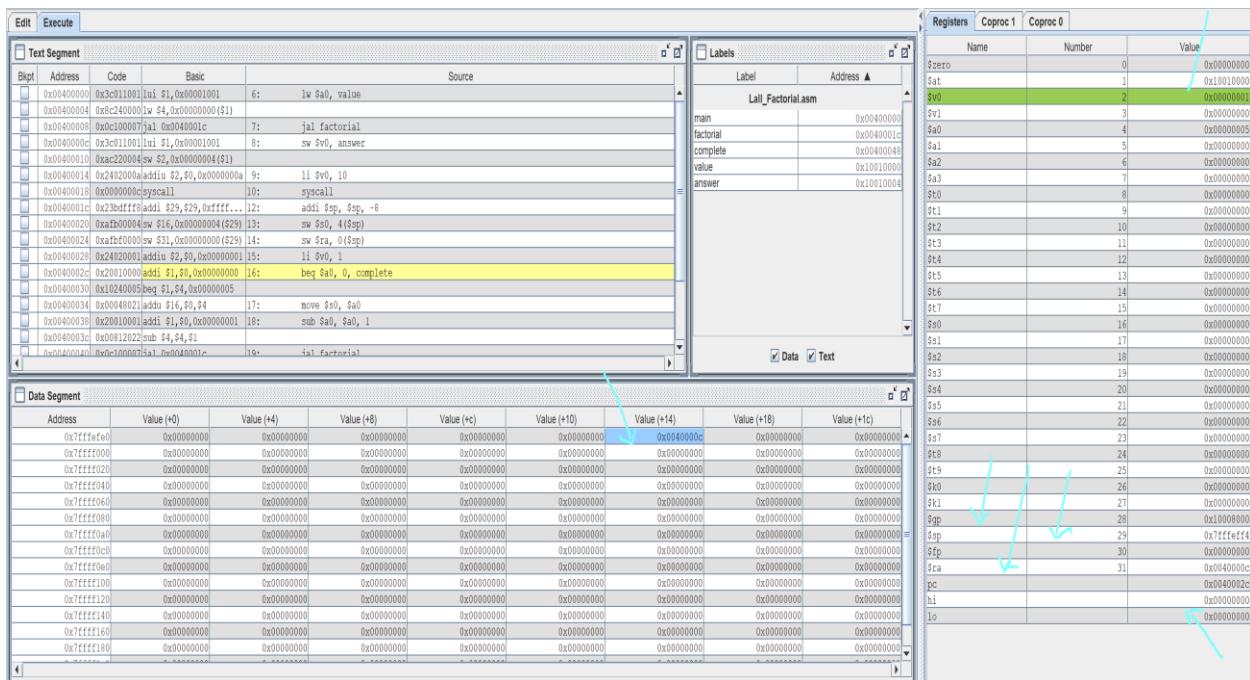
This is the code for factorial in MIPS assembly. The variable value is set to 5 for us in this case and this value is loaded on to the register \$a0. This is the value we will calculate the factorial for.

## Analysis:

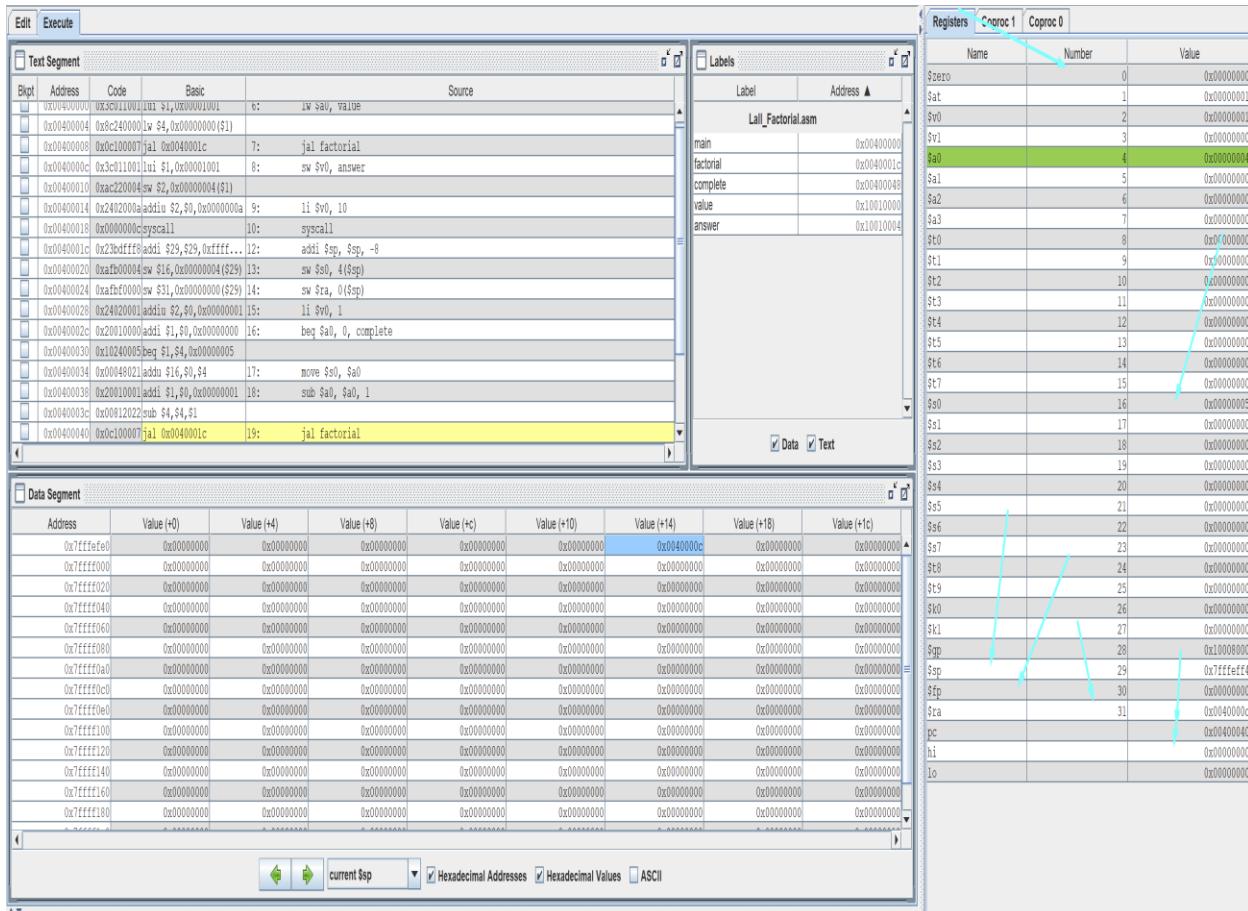
The screenshot shows the QEMU debugger interface with several windows open:

- Text Segment:** Shows assembly code with columns for Bpt, Address, Code, Basic, and Source. A yellow highlight is on the instruction at address 0x00400008, which is a `jal factorial` instruction.
- Labels:** Shows a list of labels and their addresses. The `factorial` label is highlighted.
- Registers:** Shows the state of various registers. The `$a0` register is highlighted and contains the value 5.
- Data Segment:** Shows memory starting at address 0x10010000. The `$a0` register is also highlighted here, containing the value 5.

The PC value is 0x00400008, which corresponds to our `jal factorial` 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 0x7ffffeffc.



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 0x7ffffeff8 and 0x7ffffeff4, respectively, and the value in 0x7ffffeff4 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.

The screenshot shows the assembly code for the factorial function. The PC is at address 0x0040001c, which corresponds to the 'jal factorial' instruction. The stack pointer \$ra is at address 0x00400044. The stack contains several local variables and temporary values.

Register	Name	Number	Value
\$zero	0		0x00000000
\$at	1		0x00000000
\$v0	2		0x00000078
\$v1	3		0x00000000
\$s0	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		0x00000005
\$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
\$t10	26		0x00000000
\$t11	27		0x00000000
\$t12	28		0x00000000
\$sp	29		0x7ffffeff4
\$fp	30		0x00000000
\$ra	31		0x00400044
pc			0x00400048
hi			0x00000000
lo			0x00000078

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.

The screenshot shows the assembly code for the factorial function. The PC is at address 0x0040001c, which corresponds to the 'jal factorial' instruction. The stack pointer \$ra is at address 0x00400044. The stack contains several local variables and temporary values. The stack pointer has moved to 0x7fffffec.

Register	Name	Number	Value
\$zero	0		0x00000000
\$at	1		0x00000000
\$v0	2		0x00000000
\$v1	3		0x00000000
\$s0	4		0x00000004
\$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		0x00000004
\$s1	17		0x00000000
\$s2	18		0x00000000
\$s3	19		0x00000000
\$s4	20		0x00000000
\$s5	21		0x00000000
\$s6	22		0x00000000
\$t7	23		0x00000000
\$s8	24		0x00000000
\$t9	25		0x00000000
\$t10	26		0x00000000
\$t11	27		0x00000000
\$sp	28		0x7ffffeff4
\$fp	29		0x7fffffec
\$ra	30		0x00400044
pc			0x00400048
hi			0x00000000
lo			0x00000078

This occurs just before our factorial function's third call. The stack pointer is now 0x7fffffec, and the value in \$s0 is now 0x00000004, which is saved in 0x7fffffec 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.

Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x00000001
\$v0	2	0x00000002
\$v1	3	0x00000003
\$a0	4	0x00000004
\$a1	5	0x00000005
\$a2	6	0x00000006
\$a3	7	0x00000007
\$t0	8	0x00000008
\$t1	9	0x00000009
\$t2	10	0x0000000A
\$t3	11	0x0000000B
\$t4	12	0x0000000C
\$t5	13	0x0000000D
\$t6	14	0x0000000E
\$t7	15	0x0000000F
\$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
\$t1	27	0x00000000
\$sp	28	0x10000000
\$fp	29	0x7fffffec
\$ra	30	0x00000000
pc	31	0x00400044
hi		0x00000000
lo		0x00000000

Right after the jump instruction is done performing now \$ra becomes 0x00400044 and pc now changes to 0x0040001c which is again our first instruction.

Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x00000001
\$v0	2	0x00000002
\$v1	3	0x00000003
\$a0	4	0x00000004
\$a1	5	0x00000005
\$a2	6	0x00000006
\$a3	7	0x00000007
\$t0	8	0x00000008
\$t1	9	0x00000009
\$t2	10	0x0000000A
\$t3	11	0x0000000B
\$t4	12	0x0000000C
\$t5	13	0x0000000D
\$t6	14	0x0000000E
\$t7	15	0x0000000F
\$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
\$t1	27	0x00000000
\$sp	28	0x10000000
\$fp	29	0x7fffffe4
\$ra	30	0x00000000
pc	31	0x00400044
hi		0x00000000
lo		0x00000000

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. 0x7fffffe4 is now the stack pointer. The value stored in \$s0

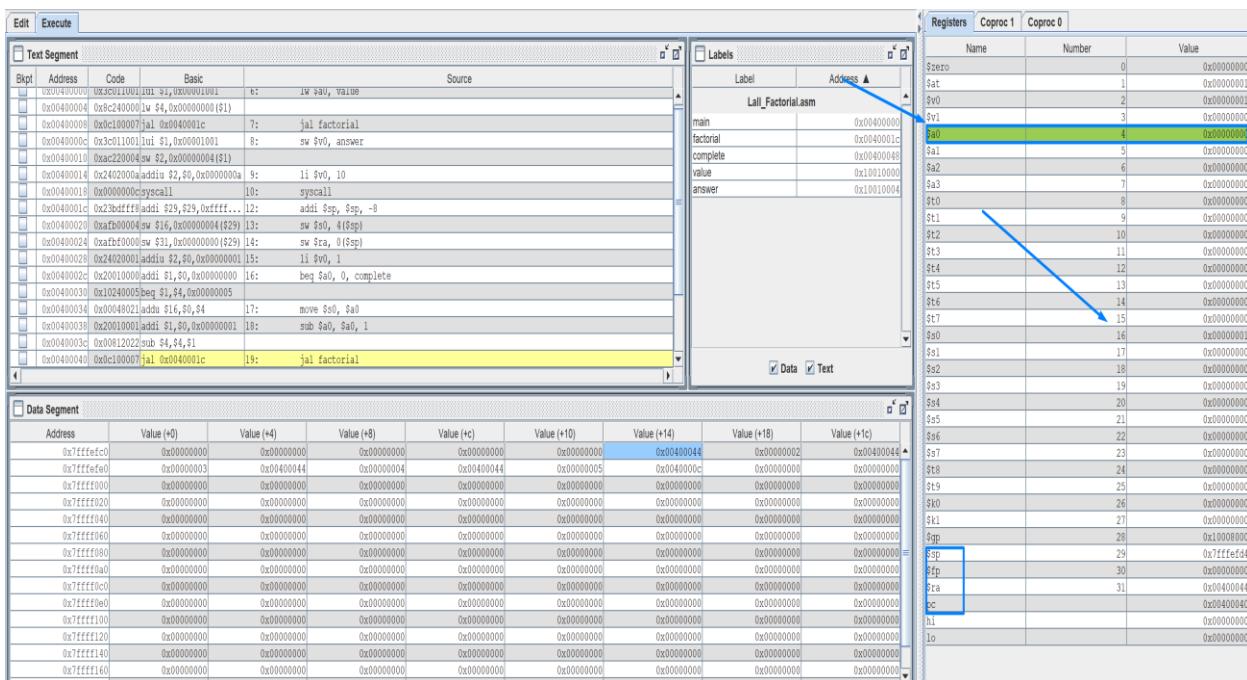
is 0x00000003, and \$a0 now equals 2 because 1 must be subtracted from 3. The value of \$ra is 0x00400044, which is not saved at 0x7fffe4, which is +4 offset from the stack pointer. The computer has now been configured to 0x00400040.

Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x00000001
\$v0	2	0x00000002
\$v1	3	0x00000003
\$a0	4	0x00000002
\$a1	5	0x00000001
\$a2	6	0x00000000
\$t0	7	0x00000000
\$t1	8	0x00000000
\$t2	9	0x00000000
\$t3	10	0x00000000
\$t4	11	0x00000000
\$t5	12	0x00000000
\$t6	13	0x00000000
\$t7	14	0x00000000
\$t8	15	0x00000000
\$t9	16	0x00000000
\$t10	17	0x00000000
\$t11	18	0x00000000
\$s0	19	0x00000000
\$s1	20	0x00000000
\$s2	21	0x00000000
\$s3	22	0x00000000
\$s4	23	0x00000000
\$t8	24	0x00000000
\$t9	25	0x00000000
\$t10	26	0x00000000
\$t11	27	0x00000000
\$t12	28	0x00000000
\$t13	29	0x7ffffe4
\$t14	30	0x00000000
\$t15	31	0x00400044
\$t16	32	0x00400044
\$t17	33	0x00000000
\$t18	34	0x00000000
\$t19	35	0x00000000
\$t20	36	0x00000000
\$t21	37	0x00000000
\$t22	38	0x00000000
\$t23	39	0x00000000
\$t24	40	0x00000000
\$t25	41	0x00000000
\$t26	42	0x00000000
\$t27	43	0x00000000
\$t28	44	0x00000000
\$t29	45	0x00000000
\$t30	46	0x00000000
\$t31	47	0x00000000
\$t32	48	0x00000000
\$t33	49	0x00000000
\$t34	50	0x00000000
\$t35	51	0x00000000
\$t36	52	0x00000000
\$t37	53	0x00000000
\$t38	54	0x00000000
\$t39	55	0x00000000
\$t40	56	0x00000000
\$t41	57	0x00000000
\$t42	58	0x00000000
\$t43	59	0x00000000
\$t44	60	0x00000000
\$t45	61	0x00000000
\$t46	62	0x00000000
\$t47	63	0x00000000
\$t48	64	0x00000000
\$t49	65	0x00000000
\$t50	66	0x00000000
\$t51	67	0x00000000
\$t52	68	0x00000000
\$t53	69	0x00000000
\$t54	70	0x00000000
\$t55	71	0x00000000
\$t56	72	0x00000000
\$t57	73	0x00000000
\$t58	74	0x00000000
\$t59	75	0x00000000
\$t60	76	0x00000000
\$t61	77	0x00000000
\$t62	78	0x00000000
\$t63	79	0x00000000
\$t64	80	0x00000000
\$t65	81	0x00000000
\$t66	82	0x00000000
\$t67	83	0x00000000
\$t68	84	0x00000000
\$t69	85	0x00000000
\$t70	86	0x00000000
\$t71	87	0x00000000
\$t72	88	0x00000000
\$t73	89	0x00000000
\$t74	90	0x00000000
\$t75	91	0x00000000
\$t76	92	0x00000000
\$t77	93	0x00000000
\$t78	94	0x00000000
\$t79	95	0x00000000
\$t80	96	0x00000000
\$t81	97	0x00000000
\$t82	98	0x00000000
\$t83	99	0x00000000
\$t84	100	0x00000000
\$t85	101	0x00000000
\$t86	102	0x00000000
\$t87	103	0x00000000
\$t88	104	0x00000000
\$t89	105	0x00000000
\$t90	106	0x00000000
\$t91	107	0x00000000
\$t92	108	0x00000000
\$t93	109	0x00000000
\$t94	110	0x00000000
\$t95	111	0x00000000
\$t96	112	0x00000000
\$t97	113	0x00000000
\$t98	114	0x00000000
\$t99	115	0x00000000
\$t100	116	0x00000000
\$t101	117	0x00000000
\$t102	118	0x00000000
\$t103	119	0x00000000
\$t104	120	0x00000000
\$t105	121	0x00000000
\$t106	122	0x00000000
\$t107	123	0x00000000
\$t108	124	0x00000000
\$t109	125	0x00000000
\$t110	126	0x00000000
\$t111	127	0x00000000
\$t112	128	0x00000000
\$t113	129	0x00000000
\$t114	130	0x00000000
\$t115	131	0x00000000
\$t116	132	0x00000000
\$t117	133	0x00000000
\$t118	134	0x00000000
\$t119	135	0x00000000
\$t120	136	0x00000000
\$t121	137	0x00000000
\$t122	138	0x00000000
\$t123	139	0x00000000
\$t124	140	0x00000000
\$t125	141	0x00000000
\$t126	142	0x00000000
\$t127	143	0x00000000
\$t128	144	0x00000000
\$t129	145	0x00000000
\$t130	146	0x00000000
\$t131	147	0x00000000
\$t132	148	0x00000000
\$t133	149	0x00000000
\$t134	150	0x00000000
\$t135	151	0x00000000
\$t136	152	0x00000000
\$t137	153	0x00000000
\$t138	154	0x00000000
\$t139	155	0x00000000
\$t140	156	0x00000000
\$t141	157	0x00000000
\$t142	158	0x00000000
\$t143	159	0x00000000
\$t144	160	0x00000000
\$t145	161	0x00000000
\$t146	162	0x00000000
\$t147	163	0x00000000
\$t148	164	0x00000000
\$t149	165	0x00000000
\$t150	166	0x00000000
\$t151	167	0x00000000
\$t152	168	0x00000000
\$t153	169	0x00000000
\$t154	170	0x00000000
\$t155	171	0x00000000
\$t156	172	0x00000000
\$t157	173	0x00000000
\$t158	174	0x00000000
\$t159	175	0x00000000
\$t160	176	0x00000000
\$t161	177	0x00000000
\$t162	178	0x00000000
\$t163	179	0x00000000
\$t164	180	0x00000000
\$t165	181	0x00000000
\$t166	182	0x00000000
\$t167	183	0x00000000
\$t168	184	0x00000000
\$t169	185	0x00000000
\$t170	186	0x00000000
\$t171	187	0x00000000
\$t172	188	0x00000000
\$t173	189	0x00000000
\$t174	190	0x00000000
\$t175	191	0x00000000
\$t176	192	0x00000000
\$t177	193	0x00000000
\$t178	194	0x00000000
\$t179	195	0x00000000
\$t180	196	0x00000000
\$t181	197	0x00000000
\$t182	198	0x00000000
\$t183	199	0x00000000
\$t184	200	0x00000000
\$t185	201	0x00000000
\$t186	202	0x00000000
\$t187	203	0x00000000
\$t188	204	0x00000000
\$t189	205	0x00000000
\$t190	206	0x00000000
\$t191	207	0x00000000
\$t192	208	0x00000000
\$t193	209	0x00000000
\$t194	210	0x00000000
\$t195	211	0x00000000
\$t196	212	0x00000000
\$t197	213	0x00000000
\$t198	214	0x00000000
\$t199	215	0x00000000
\$t200	216	0x00000000
\$t201	217	0x00000000
\$t202	218	0x00000000
\$t203	219	0x00000000
\$t204	220	0x00000000
\$t205	221	0x00000000
\$t206	222	0x00000000
\$t207	223	0x00000000
\$t208	224	0x00000000
\$t209	225	0x00000000
\$t210	226	0x00000000
\$t211	227	0x00000000
\$t212	228	0x00000000
\$t213	229	0x00000000
\$t214	230	0x00000000
\$t215	231	0x00000000
\$t216	232	0x00000000
\$t217	233	0x00000000
\$t218	234	0x00000000
\$t219	235	0x00000000
\$t220	236	0x00000000
\$t221	237	0x00000000
\$t222	238	0x00000000
\$t223	239	0x00000000
\$t224	240	0x00000000
\$t225	241	0x00000000
\$t226	242	0x00000000
\$t227	243	0x00000000
\$t228	244	0x00000000
\$t229	245	0x00000000
\$t230	246	0x00000000
\$t231	247	0x00000000
\$t232	248	0x00000000
\$t233	249	0x00000000
\$t234	250	0x00000000
\$t235	251	0x00000000
\$t236	252	0x00000000
\$t237	253	0x00000000
\$t238	254	0x00000000
\$t239	255	0x00000000
\$t240	256	0x00000000
\$t241	257	0x00000000
\$t242	258	0x00000000
\$t243	259	0x00000000
\$t244	260	0x00000000
\$t245	261	0x00000000
\$t246	262	0x00000000
\$t247	263	0x00000000
\$t248	264	0x00000000
\$t249	265	0x00000000
\$t250	266	0x00000000
\$t251	267	0x00000000
\$t252	268	0x00000000
\$t253	269	0x00000000
\$t254	270	0x00000000
\$t255	271	0x00000000
\$t256	272	0x00000000
\$t257	273	0x00000000
\$t258	274	0x00000000
\$t259	275	0x00000000
\$t260	276	0x00000000
\$t261	277	0x00000000
\$t262	278	0x00000000
\$t263	279	0x00000000
\$t264	280	0x00000000
\$t265	281	0x00000000
\$t266	282	0x00000000
\$t267	283	0x00000000
\$t268	284	0x00000000
\$t269	285	0x00000000
\$t270	286	0x00000000
\$t271	287	0x00000000
\$t272	288	0x00000000
\$t273	289	0x00000000
\$t274	290	0x00000000
\$t275	291	0x00000000
\$t276	292	0x00000000
\$t277	293	0x00000000
\$t278	294	0x00000000
\$t279	295	0x00000000
\$t280	296	0x00000000
\$t281	297	0x00000000
\$t282	298	0x00000000
\$t283	299	0x00000000
\$t284	300	0x00000000
\$t285	301	0x00000000
\$t286	302	0x00000000
\$t287	303	0x00000000
\$t288	304	0x00000000
\$t289	305	0x00000000
\$t290	306	0x00000000
\$t291	307	0x00000000
\$t292	308	0x00000000
\$t293	309	0x00000000
\$t294	310	0x00000000
\$t295	311	0x00000000
\$t296	312	0x00000000
\$t297	313	0x00000000
\$t298	314	0x00000000
\$t299	315	

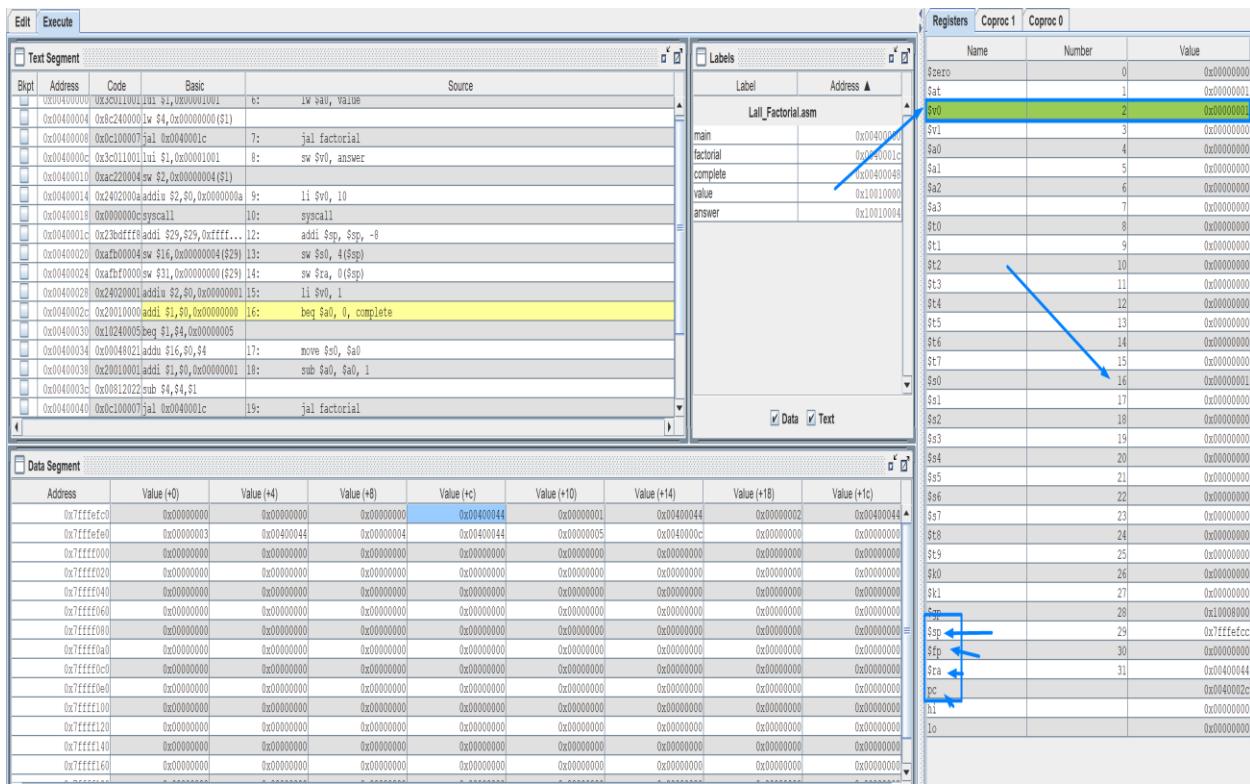
The stack pointer is 0x7fffffdc before the 5th call, and the data from \$s0 is placed at 0x7fffffe0 inside the stack. Inside the stack, the data in \$ra is 0x7fffffdc. The value of \$a0 is now decremented by one, resulting in 1 and pc becoming 0x00400040.

Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x00000001
\$v0	2	0x00000002
\$v1	3	0x00000003
\$a0	4	0x00000001
\$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	0x00400042
\$s1	17	0x00000000
\$s2	18	0x00000000
\$s3	19	0x00000000
\$s4	20	0x00000000
\$s5	21	0x00000000
\$s6	22	0x00000000
\$s7	23	0x00000000
\$s8	24	0x00000000
\$s9	25	0x00000000
\$s10	26	0x00000000
\$s11	27	0x00000000
\$s12	28	0x10000000
\$sp	29	0x7fffffdc
\$ra	30	0x00400044
pc	31	0x00400040
hi		0x00000000
lo		0x00000000

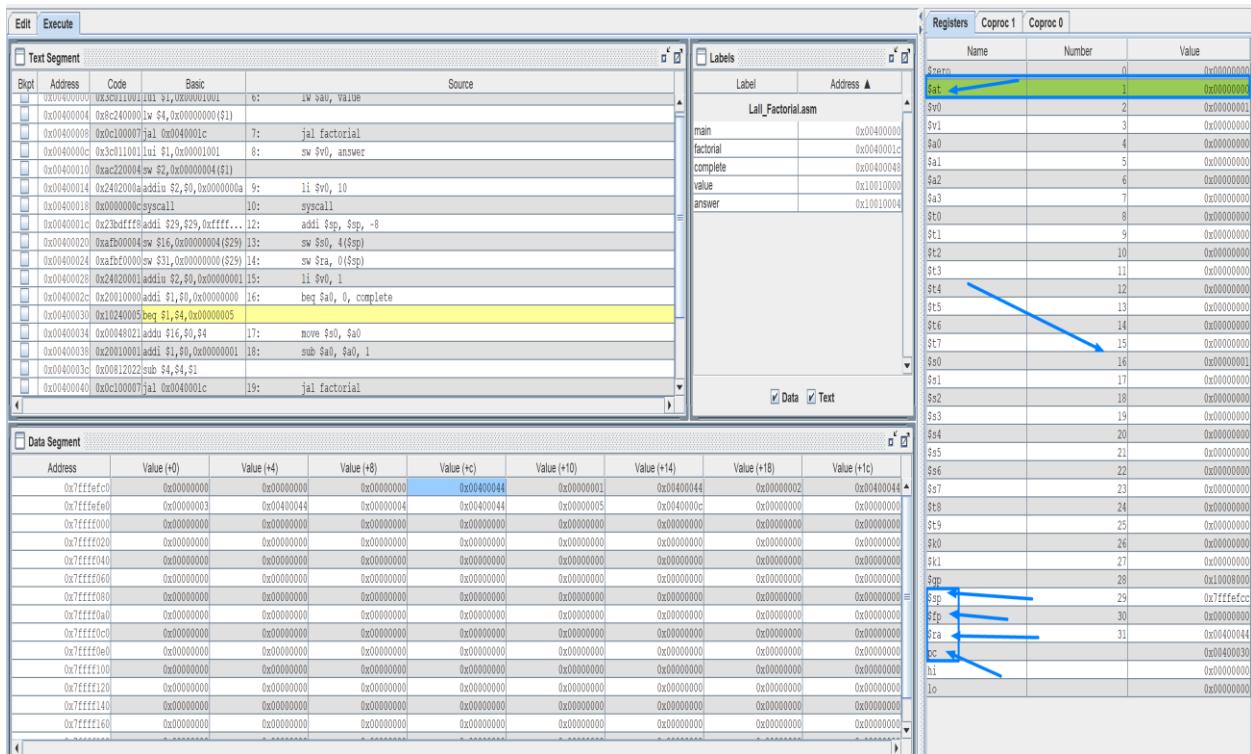
After the 5th call the pc is now set to 0x0040001c and the return address is 0x00400044.



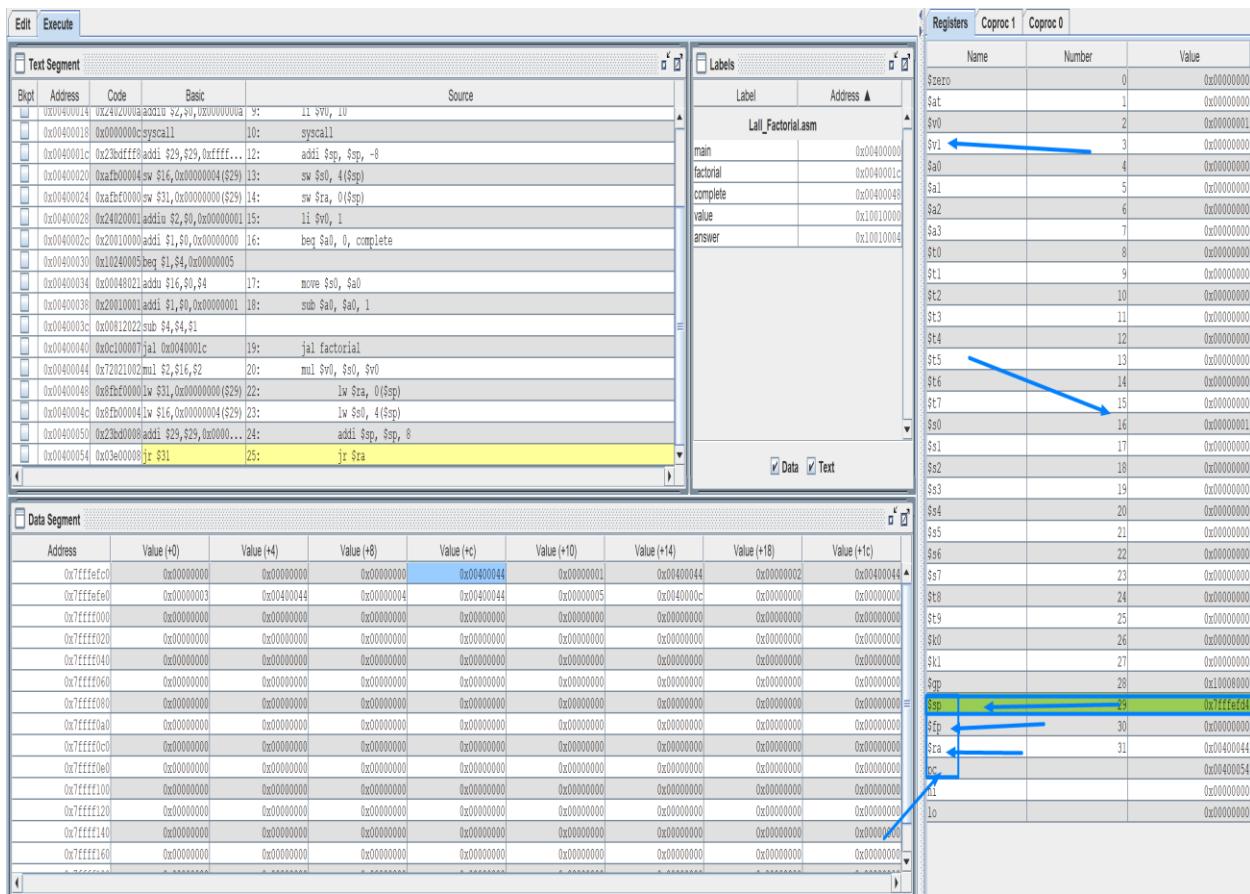
This is before we execute the final process, and we need to assign even more resources, which is why the data is in the second row. 0x7fffffd4 is now the stack pointer. Inside the stack, the value of register \$s0 is set to 0x7fffffd8, and it now has the value of 2. Inside the address 0x7fffffd4 is the 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.

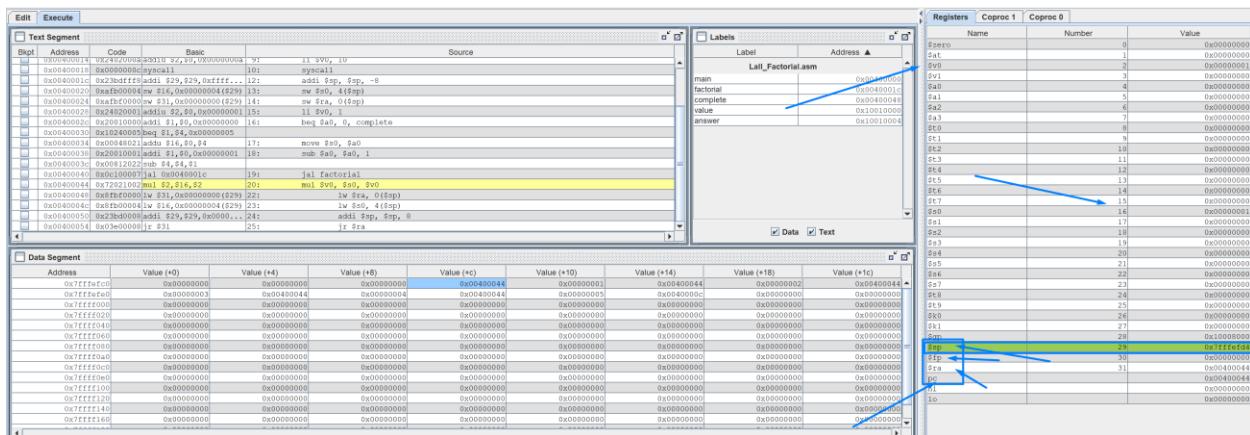


The beq branch now runs and the pc is set to 0x00400048, which is the first instruction under the Label we generated for the condition "complete".



Now we are about to 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.

Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x00000001
\$v0	2	0x00000002
\$v1	3	0x00000003
\$a0	4	0x00000004
\$a1	5	0x00000005
\$a2	6	0x00000006
\$a3	7	0x00000007
\$t0	8	0x00000008
\$t1	9	0x00000009
\$t2	10	0x0000000A
\$t3	11	0x0000000B
\$t4	12	0x0000000C
\$t5	13	0x0000000D
\$t6	14	0x0000000E
\$t7	15	0x0000000F
\$t8	16	0x00000000
\$t9	17	0x00000000
\$t10	18	0x00000000
\$t11	19	0x00000000
\$t12	20	0x00000000
\$t13	21	0x00000000
\$t14	22	0x00000000
\$t15	23	0x00000000
\$t16	24	0x00000000
\$t17	25	0x00000000
\$t18	26	0x00000000
\$t19	27	0x00000000
\$t20	28	0x00000000
\$t21	29	0x00000000
\$t22	30	0x00000000
\$t23	31	0x00000000
\$t24	32	0x00000000
\$t25	33	0x00000000
\$t26	34	0x00000000
\$t27	35	0x00000000
\$t28	36	0x00000000
\$t29	37	0x00000000
\$t30	38	0x00000000
\$t31	39	0x00000000
\$t32	40	0x00000000
\$t33	41	0x00000000
\$t34	42	0x00000000
\$t35	43	0x00000000
\$t36	44	0x00000000
\$t37	45	0x00000000
\$t38	46	0x00000000
\$t39	47	0x00000000
\$t40	48	0x00000000
\$t41	49	0x00000000
\$t42	50	0x00000000
\$t43	51	0x00000000
\$t44	52	0x00000000
\$t45	53	0x00000000
\$t46	54	0x00000000
\$t47	55	0x00000000
\$t48	56	0x00000000
\$t49	57	0x00000000
\$t50	58	0x00000000
\$t51	59	0x00000000
\$t52	60	0x00000000
\$t53	61	0x00000000
\$t54	62	0x00000000
\$t55	63	0x00000000
\$t56	64	0x00000000
\$t57	65	0x00000000
\$t58	66	0x00000000
\$t59	67	0x00000000
\$t60	68	0x00000000
\$t61	69	0x00000000
\$t62	70	0x00000000
\$t63	71	0x00000000
\$t64	72	0x00000000
\$t65	73	0x00000000
\$t66	74	0x00000000
\$t67	75	0x00000000
\$t68	76	0x00000000
\$t69	77	0x00000000
\$t70	78	0x00000000
\$t71	79	0x00000000
\$t72	80	0x00000000
\$t73	81	0x00000000
\$t74	82	0x00000000
\$t75	83	0x00000000
\$t76	84	0x00000000
\$t77	85	0x00000000
\$t78	86	0x00000000
\$t79	87	0x00000000
\$t80	88	0x00000000
\$t81	89	0x00000000
\$t82	90	0x00000000
\$t83	91	0x00000000
\$t84	92	0x00000000
\$t85	93	0x00000000
\$t86	94	0x00000000
\$t87	95	0x00000000
\$t88	96	0x00000000
\$t89	97	0x00000000
\$t90	98	0x00000000
\$t91	99	0x00000000
\$t92	100	0x00000000
\$t93	101	0x00000000
\$t94	102	0x00000000
\$t95	103	0x00000000
\$t96	104	0x00000000
\$t97	105	0x00000000
\$t98	106	0x00000000
\$t99	107	0x00000000
\$t100	108	0x00000000
\$t101	109	0x00000000
\$t102	110	0x00000000
\$t103	111	0x00000000
\$t104	112	0x00000000
\$t105	113	0x00000000
\$t106	114	0x00000000
\$t107	115	0x00000000
\$t108	116	0x00000000
\$t109	117	0x00000000
\$t110	118	0x00000000
\$t111	119	0x00000000
\$t112	120	0x00000000
\$t113	121	0x00000000
\$t114	122	0x00000000
\$t115	123	0x00000000
\$t116	124	0x00000000
\$t117	125	0x00000000
\$t118	126	0x00000000
\$t119	127	0x00000000
\$t120	128	0x00000000
\$t121	129	0x00000000
\$t122	130	0x00000000
\$t123	131	0x00000000
\$t124	132	0x00000000
\$t125	133	0x00000000
\$t126	134	0x00000000
\$t127	135	0x00000000
\$t128	136	0x00000000
\$t129	137	0x00000000
\$t130	138	0x00000000
\$t131	139	0x00000000
\$t132	140	0x00000000
\$t133	141	0x00000000
\$t134	142	0x00000000
\$t135	143	0x00000000
\$t136	144	0x00000000
\$t137	145	0x00000000
\$t138	146	0x00000000
\$t139	147	0x00000000
\$t140	148	0x00000000
\$t141	149	0x00000000
\$t142	150	0x00000000
\$t143	151	0x00000000
\$t144	152	0x00000000
\$t145	153	0x00000000
\$t146	154	0x00000000
\$t147	155	0x00000000
\$t148	156	0x00000000
\$t149	157	0x00000000
\$t150	158	0x00000000
\$t151	159	0x00000000
\$t152	160	0x00000000
\$t153	161	0x00000000
\$t154	162	0x00000000
\$t155	163	0x00000000
\$t156	164	0x00000000
\$t157	165	0x00000000
\$t158	166	0x00000000
\$t159	167	0x00000000
\$t160	168	0x00000000
\$t161	169	0x00000000
\$t162	170	0x00000000
\$t163	171	0x00000000
\$t164	172	0x00000000
\$t165	173	0x00000000
\$t166	174	0x00000000
\$t167	175	0x00000000
\$t168	176	0x00000000
\$t169	177	0x00000000
\$t170	178	0x00000000
\$t171	179	0x00000000
\$t172	180	0x00000000
\$t173	181	0x00000000
\$t174	182	0x00000000
\$t175	183	0x00000000
\$t176	184	0x00000000
\$t177	185	0x00000000
\$t178	186	0x00000000
\$t179	187	0x00000000
\$t180	188	0x00000000
\$t181	189	0x00000000
\$t182	190	0x00000000
\$t183	191	0x00000000
\$t184	192	0x00000000
\$t185	193	0x00000000
\$t186	194	0x00000000
\$t187	195	0x00000000
\$t188	196	0x00000000
\$t189	197	0x00000000
\$t190	198	0x00000000
\$t191	199	0x00000000
\$t192	200	0x00000000
\$t193	201	0x00000000
\$t194	202	0x00000000
\$t195	203	0x00000000
\$t196	204	0x00000000
\$t197	205	0x00000000
\$t198	206	0x00000000
\$t199	207	0x00000000
\$t200	208	0x00000000
\$t201	209	0x00000000
\$t202	210	0x00000000
\$t203	211	0x00000000
\$t204	212	0x00000000
\$t205	213	0x00000000
\$t206	214	0x00000000
\$t207	215	0x00000000
\$t208	216	0x00000000
\$t209	217	0x00000000
\$t210	218	0x00000000
\$t211	219	0x00000000
\$t212	220	0x00000000
\$t213	221	0x00000000
\$t214	222	0x00000000
\$t215	223	0x00000000
\$t216	224	0x00000000
\$t217	225	0x00000000
\$t218	226	0x00000000
\$t219	227	0x00000000
\$t220	228	0x00000000
\$t221	229	0x00000000
\$t222	230	0x00000000
\$t223	231	0x00000000
\$t224	232	0x00000000
\$t225	233	0x00000000
\$t226	234	0x00000000
\$t227	235	0x00000000
\$t228	236	0x00000000
\$t229	237	0x00000000
\$t230	238	0x00000000
\$t231	239	0x00000000
\$t232	240	0x00000000
\$t233	241	0x00000000
\$t234	242	0x00000000
\$t235	243	0x00000000
\$t236	244	0x00000000
\$t237	245	0x00000000
\$t238	246	0x00000000
\$t239	247	0x00000000
\$t240	248	0x00000000
\$t241	249	0x00000000
\$t242	250	0x00000000
\$t243	251	0x00000000
\$t244	252	0x00000000
\$t245	253	0x00000000
\$t246	254	0x00000000
\$t247	255	0x00000000
\$t248	256	0x00000000
\$t249	257	0x00000000
\$t250	258	0x00000000
\$t251	259	0x00000000
\$t252	260	0x00000000
\$t253	261	0x00000000
\$t254	262	0x00000000
\$t255	263	0x00000000
\$t256	264	0x00000000
\$t257	265	0x00000000
\$t258	266	0x00000000
\$t259	267	0x00000000
\$t260	268	0x00000000
\$t261	269	0x00000000
\$t262	270	0x00000000
\$t263	271	0x00000000
\$t264	272	0x00000000
\$t265	273	0x00000000
\$t266	274	0x00000000
\$t267	275	0x00000000
\$t268	276	0x00000000
\$t269	277	0x00000000
\$t270	278	0x00000000
\$t271	279	0x00000000
\$t272	280	0x00000000
\$t273	281	0x00000000
\$t274	282	0x00000000
\$t275	283	0x00000000
\$t276	284	0x00000000
\$t277	285	0x00000000
\$t278	286	0x00000000
\$t279	287	0x00000000
\$t280	288	0x00000000
\$t281	289	0x00000000
\$t282	290	0x00000000
\$t283	291	0x00000000
\$t284	292	0x00000000
\$t285	293	0x00000000
\$t286	294	0x00000000
\$t287	295	0x00000000
\$t288	296	0x00000000
\$t289	297	0x00000000
\$t290	298	0x00000000
\$t291	299	0x00000000
\$t292	300	0x00000000
\$t293	301	0x00000000
\$t294	302	0x00000000
\$t295	303	0x00000000
\$t296	304	0x00000000
\$t297	305	0x00000000
\$t298	306	0x00000000
\$t299	307	0x00000000
\$t300	308	0x00000000
\$t301	309	0x00000000
\$t302	310	0x00000000
\$t303	311	0x00000000
\$t304	312	0x00000000
\$t305	313	0x00000000
\$t306</		

The screenshot shows the Immunity Debugger interface with several windows open:

- Text Segment**: Displays assembly code for the `lali_factorial.asm` file. The code implements a factorial function using a recursive call. It includes labels like `main`, `factorial`, `complete`, `status`, and `answer`.
- Registers**: Shows the CPU registers for the current thread. Registers include `$rax` through `$r10`, `$r11` through `$r14`, `$r15`, `$r16`, `$r17`, `$r18`, `$r19`, `$r20`, `$r21`, `$r22`, `$r23`, `$r24`, `$r25`, `$r26`, `$r27`, `$r28`, `$r29`, `$r30`, and `$r31`. Values are mostly zero or reflect the current state of the program.
- Coproc 1** and **Coproc 0**: These windows show floating-point registers (FPU) and SIMD/YMM registers respectively, both of which are currently empty.
- Data Segment**: Displays memory dump information for the segment starting at `0xfffffe00`. It lists memory addresses from `0xfffffe00` to `0xfffffe10` and their corresponding values.

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

The screenshot shows the Immunity Debugger interface with several windows open:

- Registers**: Shows CPU registers (eax, ebx, ecx, edx, etc.) with their current values.
- Coproc 1**: Shows floating-point coprocessor registers.
- Coproc 0**: Shows floating-point coprocessor registers.
- Labels**: Shows the assembly labels and their addresses, including `main`, `factorial`, `complete`, `value`, and `answer`.
- Assembly**: The main assembly pane showing the assembly code for the factorial function. A blue arrow points from the `main` label in the Labels pane to the `main:` instruction in the assembly pane.
- Data Segment**: Shows the memory dump pane with data segments and their values.

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

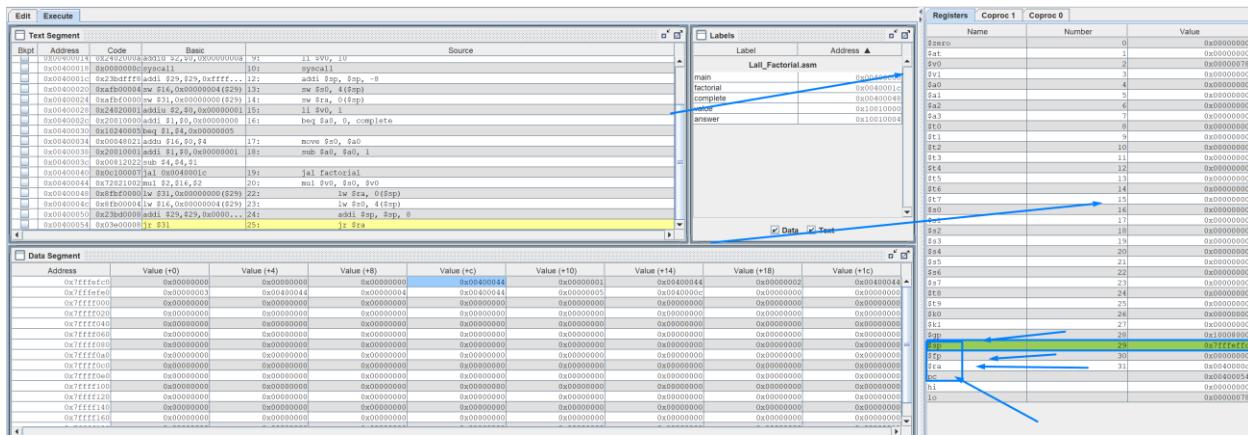
values are popped from stack again and the stack pointer again changed to 0x7ffffe4 like how it was

changing in previous calls.

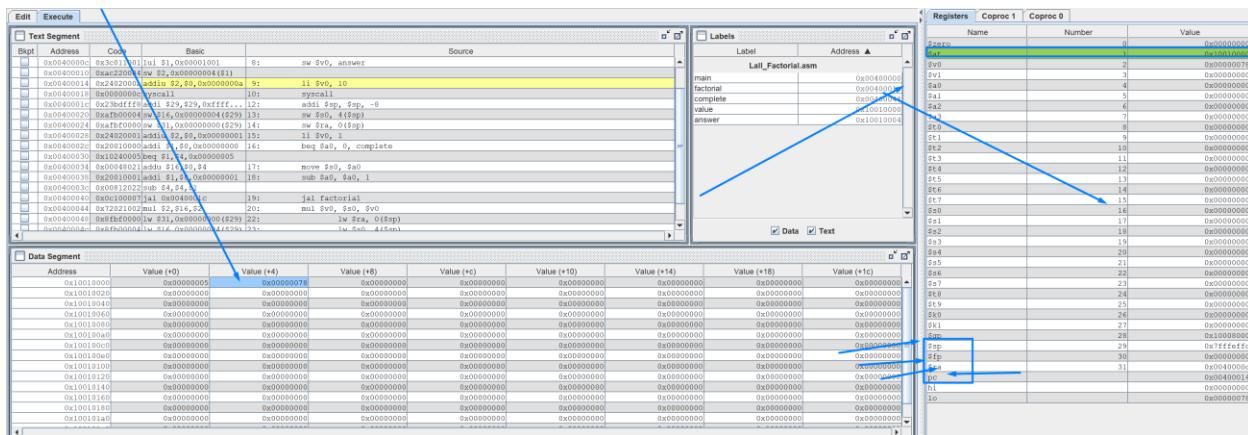
The screenshot shows the WinAPI debugger interface with several windows open:

- Text Segment**: Displays assembly code for `lall_factorial.asm`. The code implements a factorial function using recursion. It includes instructions like `push rbp`, `mov rbp, rsp`, `push r15`, `push r14`, `push r13`, `push r12`, `push r11`, `push r10`, `push r9`, `push r8`, `push r7`, `push r6`, `push r5`, `push r4`, `push r3`, `push r2`, `push r1`, `push rbp`, `sub rbp, r15`, `mov rbp, r15`, `push rbp`, `sub rbp, r14`, `push rbp`, `sub rbp, r13`, `push rbp`, `sub rbp, r12`, `push rbp`, `sub rbp, r11`, `push rbp`, `sub rbp, r10`, `push rbp`, `sub rbp, r9`, `push rbp`, `sub rbp, r8`, `push rbp`, `sub rbp, r7`, `push rbp`, `sub rbp, r6`, `push rbp`, `sub rbp, r5`, `push rbp`, `sub rbp, r4`, `push rbp`, `sub rbp, r3`, `push rbp`, `sub rbp, r2`, `push rbp`, `sub rbp, r1`, `push rbp`, `sub rbp, rbp`, `add rax, rbp`, `pop rbp`, `pop r15`, `pop r14`, `pop r13`, `pop r12`, `pop r11`, `pop r10`, `pop r9`, `pop r8`, `pop r7`, `pop r6`, `pop r5`, `pop r4`, `pop r3`, `pop r2`, `pop r1`, and `ret`.
- Registers**: Shows CPU registers with their current values.
- Coproc 1**: Shows floating-point coprocessor registers.
- Coproc 0**: Shows floating-point coprocessor registers.
- Labels**: Shows labels and addresses for the assembly code.
- Data Segment**: Shows memory dump of the data segment.

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 Lall_Factorial.c > main()
1 #include <stdio.h>
2
3 int factorial(int N)
4 {
5     if (N == 1)
6         return 1;
7     return(N * factorial(N - 1));
8
9 }
10
11 int main()
12 {
13     int N_fact = factorial(5);
14 }
```

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 0x000055555555129.

(gdb) info registers		
rax	0x5555555555158	93824992235864
rbx	0x0	0
rcx	0x0	0
rdx	0x7fffffffdec8	140737488346824
rsi	0x7fffffffdeb8	140737488346808
rdi	0x5	5
rbp	0x7fffffffdd70	0x7fffffffdd70
rsp	0x7fffffffdd70	0x7fffffffdd70
r8	0x7fffff7fa5bd0	140737353767888
r9	0x7fffff7fd9d00	140737353981184
r10	0xfffffffffffffff88	-1144
r11	0x202	514
r12	0x7fffffffdeb8	140737488346808
r13	0x5555555555158	93824992235864
r14	0x0	0
r15	0x7fffff7ffbc40	140737354120256
rip	0x5555555555131	0x5555555555131 <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: 0xfffffdd90          0x000007fff
(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 0xfffffdd90.

```
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 0x0000555555555145 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           0x7fffff7fa5bd0 140737353767888
r9           0x7fffff7fd9d00 140737353981184
r10          0xfffffffffffffff88 -1144
r11          0x202          514
r12          0x7fffffffdeb8 140737488346808
r13          0x5555555555158 93824992235864
r14          0x0            0
r15          0x7fffff7ffbc40 140737354120256
rip          0x5555555555131 0x5555555555131 <factorial+8>
eflags        0x206          [ PF IF ]
cs            0x33           51
ss            0x2b           43
ds            0x0            0
es            0x0            0
fs            0x0            0
qs            0x0            0
```

```
0x00005555555555131 <+8>:    sub   $0x10,%rsp
0x00005555555555135 <+12>:   mov    %edi,-0x4(%rbp)
0x00005555555555138 <+15>:   cmpl  $0x1,-0x4(%rbp)
0x0000555555555513c <+19>:   jne    0x5555555555145 <factorial+28>
0x0000555555555513e <+21>:   mov    $0x1,%eax
0x00005555555555143 <+26>:   jmp    0x5555555555156 <factorial+45>
0x00005555555555145 <+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
```

```
0x000055555555129 <+0>:    endbr64
0x00005555555512d <+4>:    push   %rbp
0x00005555555512e <+5>:    mov    %rsp,%rbp
0x000055555555131 <+8>:    sub    $0x10,%rsp
0x000055555555135 <+12>:   mov    %edi,-0x4(%rbp)
0x000055555555138 <+15>:   cmpl   $0x1,-0x4(%rbp)
0x00005555555513c <+19>:   jne    0x55555555145 <factorial+28>
0x00005555555513e <+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
```

```
(gdb) disassemble factorial
Dump of assembler code for function factorial:
0x000055555555129 <+0>:    endbr64
0x00005555555512d <+4>:    push   %rbp
0x00005555555512e <+5>:    mov    %rsp,%rbp
=> 0x000055555555131 <+8>:    sub    $0x10,%rsp
0x000055555555135 <+12>:   mov    %edi,-0x4(%rbp)
0x000055555555138 <+15>:   cmpl   $0x1,-0x4(%rbp)
0x00005555555513c <+19>:   jne    0x55555555145 <factorial+28>
0x00005555555513e <+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)
```

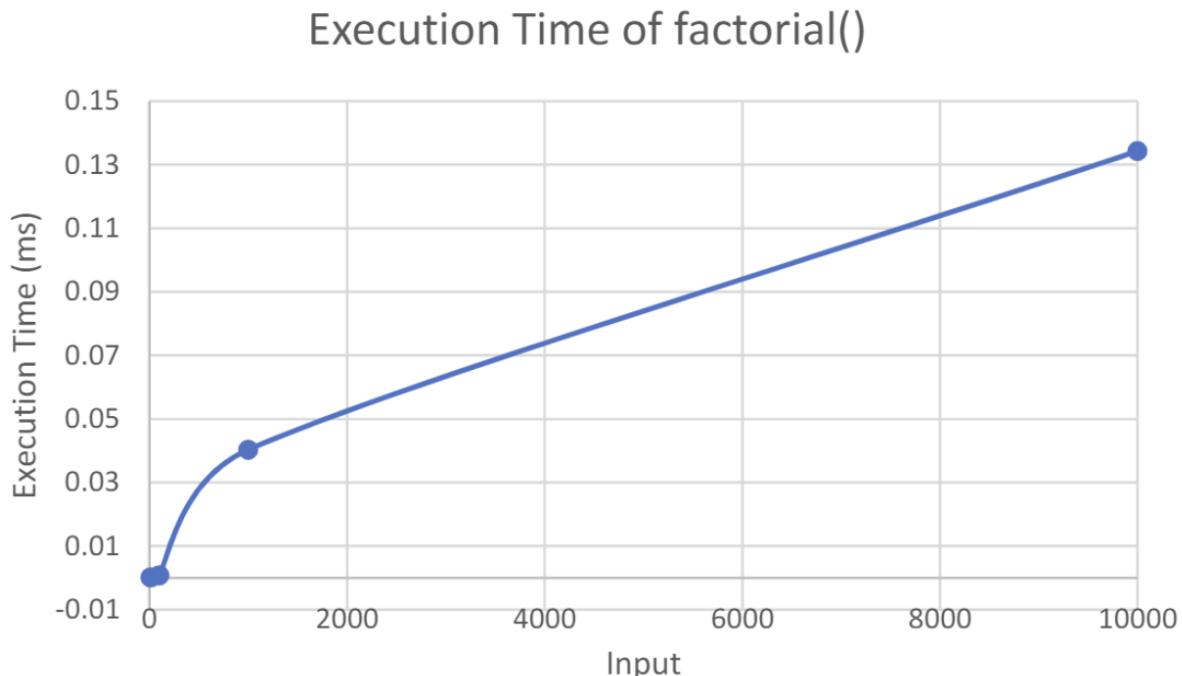
As you can see on the RDI, the value is now equal to 1, indicating that the jump will not occur again. The code will now begin to deallocate memory and return the items it previously stored. It will double the values as it returns them.

```
(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          0xfffffffffffffff88 -1144  
r11          0x202          514  
r12          0x7fffffffdeb8 140737488346808  
r13          0x5555555555158 93824992235864  
r14          0x0           0  
r15          0x7ffff7ffbc40 140737354120256  
rip          0x555555555516e 0x555555555516e <main+22>  
eflags        0x206          [ PF IF ]  
cs            0x33          51  
ss            0x2b          43  
ds            0x0           0  
es            0x0           0  
fs            0x0           0  
qs            0x0           0
```

```
0x00005555555555178 <+31>:    sub    $0x1,%eax  
0x0000555555555514b <+34>:    mov    %eax,%edi  
0x0000555555555514d <+36>:    call   0x5555555555129 <factorial>  
0x00005555555555152 <+41>:    imul   -0x4(%rbp),%eax  
0x00005555555555156 <+45>:    leave
```

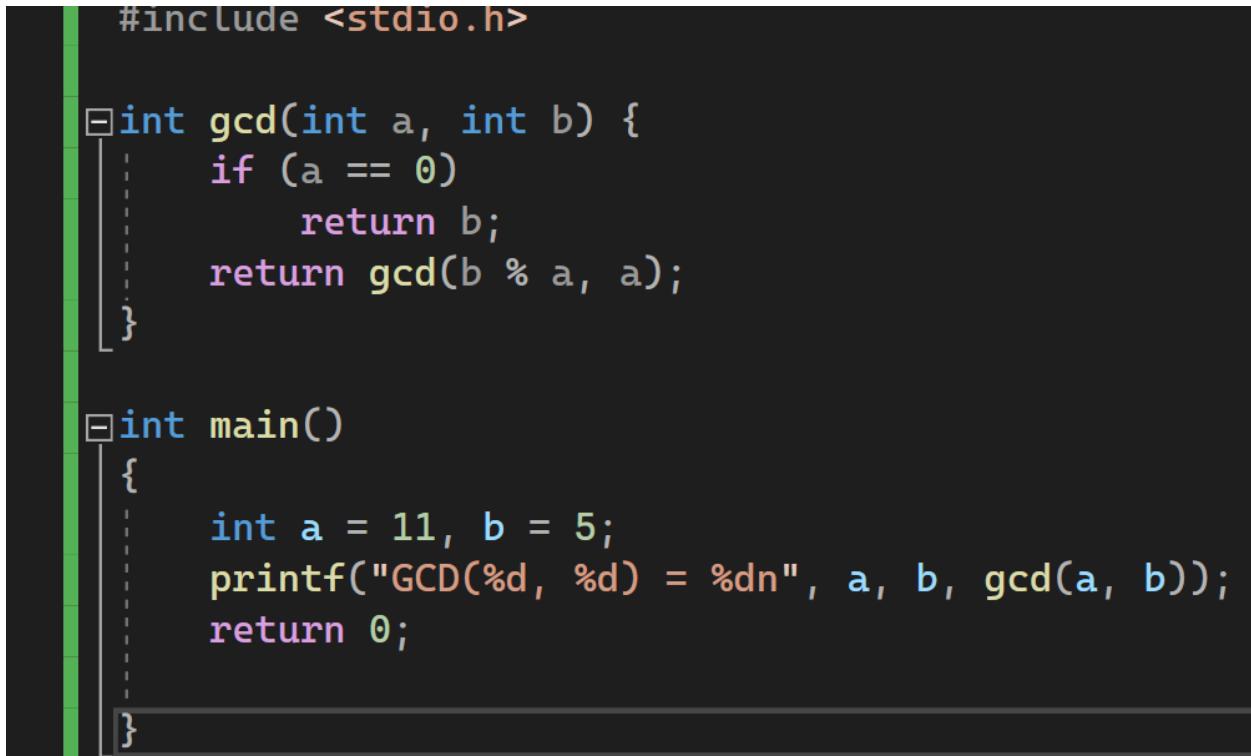
This is our final return so as you remember in our visual studio and MIPs the final answer was 120 which is 0x78 in hex and this is the same case for Linux. So RAX is 0x78 which is our final answer. The stack pointer now gets reset to the first stack pointer we had and now all the memory is deallocated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

Time comparison:



GCD:

Visual Studio:



```
#include <stdio.h>

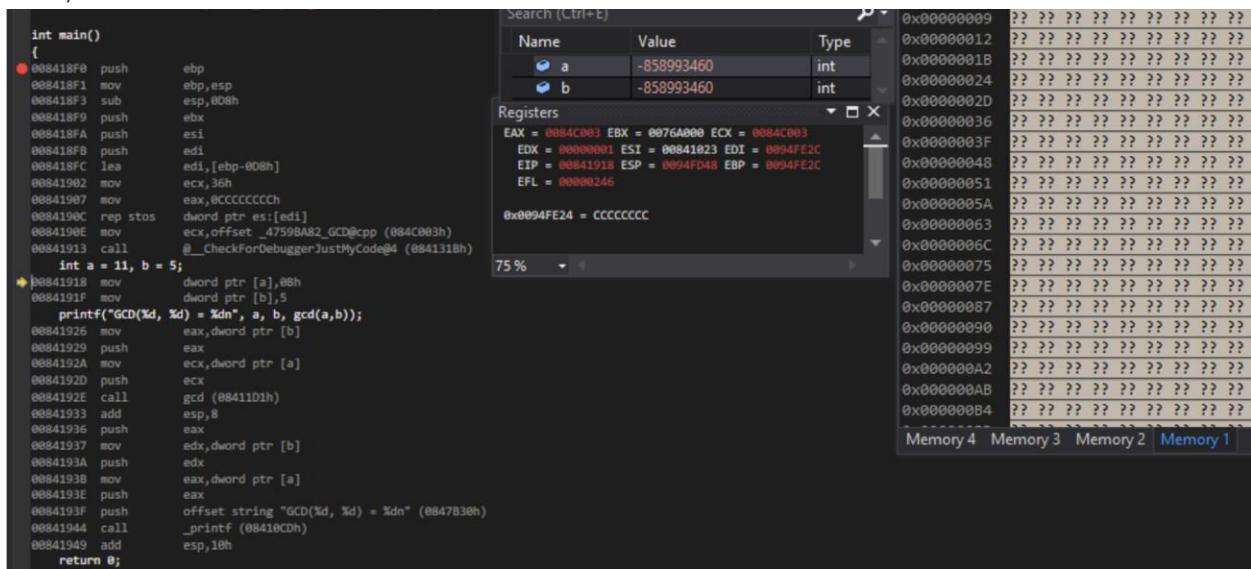
int gcd(int a, int b) {
    if (a == 0)
        return b;
    return gcd(b % a, a);
}

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

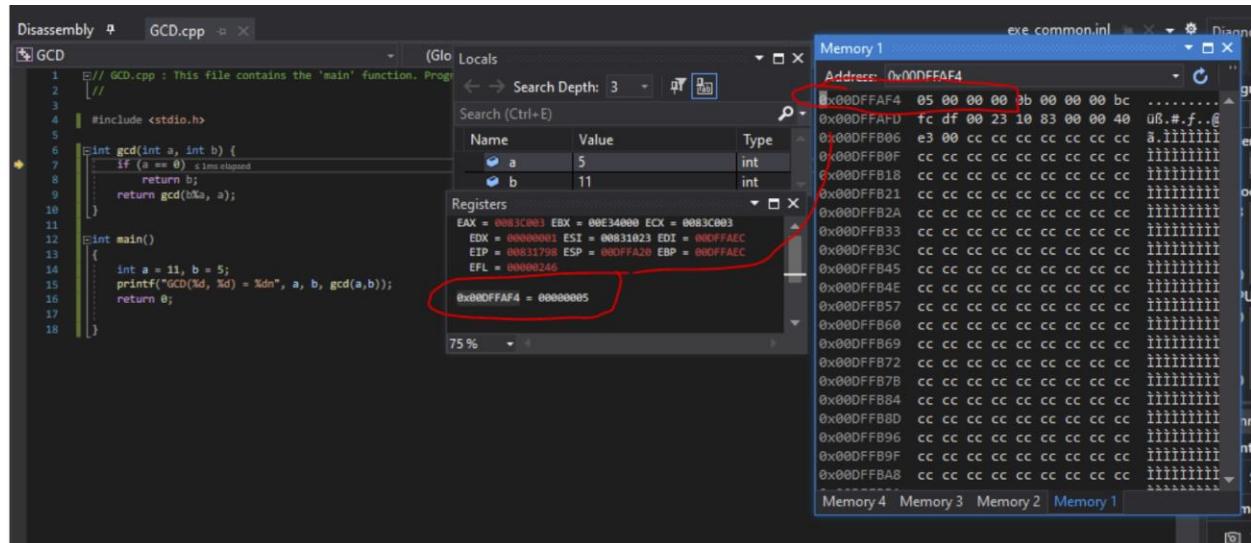
This is the visual studio code in CPP for our GCD function. The values we are hardcoding are 11 and 5

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:

```

1      .data
2 str1: .asciiz "Enter first integer n1: "
3 str2: .asciiz "Enter first integer n2: "
4 str3: .asciiz "The greatest common divisor of n1 and n2 is "
5 str4: .asciiz "\nThe least common multiple of n1 and n2 is "
6 newline: .asciiz "\n"
7 .align 2
8 .globl main
9 .text
10 main:
11     la $a0, str1
12     li $v0, 4
13     syscall
14     li $v0, 5
15     syscall
16     add $a1, $v0, $zero
17     la $a0, str2
18     li $v0, 4
19     syscall
20     li $v0, 5
21     syscall
22     add $a2, $v0, $zero
23     addi $sp, $sp, -8
24     sw $a1, 4($sp)
25     sw $a2, 0($sp)
26     jal gcd
27     lw $a2, 0($sp)
28     lw $a1, 4($sp)
29     addi $sp, $sp, 4
30     add $s0, $v0, $zero
31     sw $s0, 0($sp)
32     la $a0, str3
33     li $v0, 4
34     syscall
35     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
	0x00400000	0x3c011001	lui \$1,0x00001001
	0x00400004	0x34240000	ori \$4,\$1,0x00000000
	0x00400008	0x24020004	addiu \$2,\$0,0x00000004
	0x0040000c	0x0000000c	syscall
	0x00400010	0x24020005	addiu \$2,\$0,0x00000005
	0x00400014	0x0000000c	syscall
	0x00400018	0x00402820	add \$5,\$2,\$0
	0x0040001c	0x3c011001	lui \$1,0x00001001
	0x00400020	0x34240019	ori \$4,\$1,0x00000019
	0x00400024	0x24020004	addiu \$2,\$0,0x00000004
	0x00400028	0x0000000c	syscall
	0x0040002c	0x24020005	addiu \$2,\$0,0x00000005
	0x00400030	0x0000000c	syscall
	0x00400034	0x00403020	add \$6,\$2,\$0
	0x00400038	0x23bdffff8	addi \$29,\$29,0xffff...
	0x0040003c	0xaafa50004	sw \$5,0x00000004(\$29)
	0x00400040	0xaafa60000	sw \$6,0x00000000(\$29)
	0x00400044	0x0c10002b	jal 0x004000ac
	0x00400048	0x8fa60000	lw \$6,0x00000000(\$29)
	0x0040004c	0x8fa50004	lw \$5,0x00000004(\$29)
	0x00400050	0x23bd0004	addi \$29,\$29,0x0000...
	0x00400054	0x00408020	add \$16,\$2,\$0
	0x00400058	0xafbf00000	sw \$16,0x00000000(\$29)
	0x0040005c	0x3c011001	lui \$1,0x00001001
	0x00400060	0x34240032	ori \$4,\$1,0x00000032
	0x00400064	0x24020004	addiu \$2,\$0,0x00000004
	0x00400068	0x0000000c	syscall
	0x0040006c	0x24020001	addiu \$2,\$0,0x00000001
	0x00400070	0x02002020	add \$4,\$16,\$0
	0x00400074	0x0000000c	syscall
	0x00400078	0x3c011001	lui \$1,0x00001001
	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	0x23bdffff8	addi \$29,\$29,0xffff...	49:     addi \$sp, \$sp, -8
	0x004000b0	0xafbf0004	sw \$31,0x00000004(\$29)	50:     sw \$ra, 4(\$sp)
	0x004000b4	0x00a6001a	div \$5,\$6	51:     div \$a1, \$a2
	0x004000b8	0x000008010	mfhi \$16	52:     mfhi \$s0
	0x004000bc	0xafb00000	sw \$16,0x00000000(\$29)	53:     sw \$s0, 0(\$sp)
	0x004000c0	0x16000003	bne \$16,\$0,0x00000003	54:     bne \$s0, \$zero, L1
	0x004000c4	0x00c01020	add \$2,\$6,\$0	55:     add \$v0, \$a2, \$zero
	0x004000c8	0x23bd0008	addi \$29,\$29,0x0000...	56:     addi \$sp, \$sp, 8
	0x004000cc	0x03e00008	jr \$31	57:     addi \$sp, \$sp, 8
	0x004000d0	0x00c02820	add \$5,\$6,\$0	58:     jr \$ra
	0x004000d4	0x8fb00000	lw \$16,0x00000000(\$29)	59:     lw \$s0, 0(\$sp)
	0x004000d8	0x02003020	add \$6,\$16,\$0	60:     add \$a2, \$s0, \$zero
	0x004000dc	0x0c10002b	jal 0x004000ac	61:     lw \$s0, 0(\$sp)
	0x004000e0	0x8fbf0004	lw \$31,0x00000004(\$29)	62:     add \$a2, \$s0, \$zero
	0x004000e4	0x23bd0008	addi \$29,\$29,0x0000...	63:     jal gcd
	0x004000e8	0x03e00008	jr \$31	64:     lw \$ra, 4(\$sp)
				65:     addi \$sp, \$sp, 8
				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	0x0000000000
\$at	1	0x0000000000
\$v0	2	0x0000000000
\$v1	3	0x0000000000
\$a0	4	0x0000000000
\$a1	5	0x0000000000
\$a2	6	0x0000000000
\$a3	7	0x0000000000
\$t0	8	0x0000000000
\$t1	9	0x0000000000
\$t2	10	0x0000000000
\$t3	11	0x0000000000
\$t4	12	0x0000000000
\$t5	13	0x0000000000
\$t6	14	0x0000000000
\$t7	15	0x0000000000
\$s0	16	0x0000000000
\$s1	17	0x0000000000
\$s2	18	0x0000000000
\$s3	19	0x0000000000
\$s4	20	0x0000000000
\$s5	21	0x0000000000
\$s6	22	0x0000000000
\$s7	23	0x0000000000
\$t8	24	0x0000000000
\$t9	25	0x0000000000
\$k0	26	0x0000000000
\$k1	27	0x0000000000
\$gp	28	0x10008000
\$sp	29	0x7ffffeffc
\$fp	30	0x0000000000
\$ra	31	0x0000000000
pc		0x00400000
hi		0x0000000000
lo		0x0000000000

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.

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.

-- program is finished running --

Clear

-- program is finished running --

Enter first integer n1: 10  
Enter second integer n2: 5

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.

Bkpt	Address	Code	Basic	Source	Labels	Address	Registers		
	0x00400002	0x42400000+0x1	la \$v0, _zero		main	0x00400000	\$r0	0	0x00000000
	0x00400003	0x42400000+0x2	la \$v1, _zero				\$r1	1	0x10010000
	0x00400004	0x42400000+0x3	addi \$v0, \$v1, 0x00000004	12: li \$v0, 4			\$r2	2	0x00000005
	0x00400005	0x42400000+0x4	addi \$v0, \$v1, 0x00000005	13: syscall			\$r3	3	0x00000000
	0x00400006	0x42400000+0x5	addi \$v0, \$v1, 0x00000006	14: li \$v0, 5			\$r4	4	0x10010015
	0x00400007	0x42400000+0x6	addi \$v0, \$v1, 0x00000007	15: syscall			\$r5	5	0x00000004
	0x00400008	0x42400000+0x7	addi \$v0, \$v1, 0x00000008	16: li \$v0, 8			\$r6	6	0x00000005
	0x00400009	0x42400000+0x8	addi \$v0, \$v1, 0x00000009	17: la \$at, _str2			\$r7	7	0x00000000
	0x0040000a	0x42400000+0x9	addi \$v0, \$v1, 0x0000000a	18: addi \$sp, \$gp, -8			\$r8	8	0x00000000
	0x0040000b	0x42400000+0xa	addi \$v0, \$v1, 0x0000000b	19: la \$at, _str1			\$r9	9	0x00000000
	0x0040000c	0x42400000+0xb	addi \$v0, \$v1, 0x0000000c	20: addi \$sp, \$gp, -8			\$r10	10	0x00000000
	0x0040000d	0x42400000+0xc	addi \$v0, \$v1, 0x0000000d	21: syscall			\$r11	11	0x00000000
	0x0040000e	0x42400000+0xd	addi \$v0, \$v1, 0x0000000e	22: addi \$sp, \$gp, -8			\$r12	12	0x00000000
	0x0040000f	0x42400000+0xe	addi \$v0, \$v1, 0x0000000f	23: addi \$sp, \$gp, -8			\$r13	13	0x00000000
	0x00400010	0x42400000+0xf	addi \$v0, \$v1, 0x00000000	24: sw \$at, 4(\$sp)			\$r14	14	0x00000000
	0x00400011	0x42400000+0x10	addi \$v0, \$v1, 0x00000000	25: sw \$at, 8(\$sp)			\$r15	15	0x00000000
	0x00400012	0x42400000+0x11	addi \$v0, \$v1, 0x00000000	26: jal gcd	gcd	0x004000ac	\$r16	16	0x00000000
	0x00400013	0x42400000+0x12	addi \$v0, \$v1, 0x00000000				\$r17	17	0x00000000
	0x00400014	0x42400000+0x13	addi \$v0, \$v1, 0x00000000				\$r18	18	0x00000000
	0x00400015	0x42400000+0x14	addi \$v0, \$v1, 0x00000000				\$r19	19	0x00000000
	0x00400016	0x42400000+0x15	addi \$v0, \$v1, 0x00000000				\$r20	20	0x00000000
	0x00400017	0x42400000+0x16	addi \$v0, \$v1, 0x00000000				\$r21	21	0x00000000
	0x00400018	0x42400000+0x17	addi \$v0, \$v1, 0x00000000				\$r22	22	0x00000000
	0x00400019	0x42400000+0x18	addi \$v0, \$v1, 0x00000000				\$r23	23	0x00000000
	0x0040001a	0x42400000+0x19	addi \$v0, \$v1, 0x00000000				\$r24	24	0x00000000
	0x0040001b	0x42400000+0x1a	addi \$v0, \$v1, 0x00000000				\$r25	25	0x00000000
	0x0040001c	0x42400000+0x1b	addi \$v0, \$v1, 0x00000000				\$r26	26	0x00000000
	0x0040001d	0x42400000+0x1c	addi \$v0, \$v1, 0x00000000				\$r27	27	0x00000000
	0x0040001e	0x42400000+0x1d	addi \$v0, \$v1, 0x00000000				\$r28	28	0x00000000
	0x0040001f	0x42400000+0x1e	addi \$v0, \$v1, 0x00000000				\$r29	29	0x7fffffe4
	0x00400020	0x42400000+0x1f	addi \$v0, \$v1, 0x00000000				\$r30	30	0x00000000
	0x00400021	0x42400000+0x20	addi \$v0, \$v1, 0x00000000				\$r31	31	0x00000000
	0x00400022	0x42400000+0x21	addi \$v0, \$v1, 0x00000000						
	0x00400023	0x42400000+0x22	addi \$v0, \$v1, 0x00000000						
	0x00400024	0x42400000+0x23	addi \$v0, \$v1, 0x00000000						
	0x00400025	0x42400000+0x24	addi \$v0, \$v1, 0x00000000						
	0x00400026	0x42400000+0x25	addi \$v0, \$v1, 0x00000000						
	0x00400027	0x42400000+0x26	addi \$v0, \$v1, 0x00000000						
	0x00400028	0x42400000+0x27	addi \$v0, \$v1, 0x00000000						
	0x00400029	0x42400000+0x28	addi \$v0, \$v1, 0x00000000						
	0x0040002a	0x42400000+0x29	addi \$v0, \$v1, 0x00000000						
	0x0040002b	0x42400000+0x2b	addi \$v0, \$v1, 0x00000000						
	0x0040002c	0x42400000+0x2c	addi \$v0, \$v1, 0x00000000						
	0x0040002d	0x42400000+0x2d	addi \$v0, \$v1, 0x00000000						
	0x0040002e	0x42400000+0x2e	addi \$v0, \$v1, 0x00000000						
	0x0040002f	0x42400000+0x2f	addi \$v0, \$v1, 0x00000000						
	0x00400030	0x42400000+0x30	addi \$v0, \$v1, 0x00000000						
	0x00400031	0x42400000+0x31	addi \$v0, \$v1, 0x00000000						
	0x00400032	0x42400000+0x32	addi \$v0, \$v1, 0x00000000						
	0x00400033	0x42400000+0x33	addi \$v0, \$v1, 0x00000000						
	0x00400034	0x42400000+0x34	addi \$v0, \$v1, 0x00000000						
	0x00400035	0x42400000+0x35	addi \$v0, \$v1, 0x00000000						
	0x00400036	0x42400000+0x36	addi \$v0, \$v1, 0x00000000						
	0x00400037	0x42400000+0x37	addi \$v0, \$v1, 0x00000000						
	0x00400038	0x42400000+0x38	addi \$v0, \$v1, 0x00000000						
	0x00400039	0x42400000+0x39	addi \$v0, \$v1, 0x00000000						
	0x0040003a	0x42400000+0x3a	addi \$v0, \$v1, 0x00000000						
	0x0040003b	0x42400000+0x3b	addi \$v0, \$v1, 0x00000000						
	0x0040003c	0x42400000+0x3c	addi \$v0, \$v1, 0x00000000						
	0x0040003d	0x42400000+0x3d	addi \$v0, \$v1, 0x00000000						
	0x0040003e	0x42400000+0x3e	addi \$v0, \$v1, 0x00000000						
	0x0040003f	0x42400000+0x3f	addi \$v0, \$v1, 0x00000000						
	0x00400040	0x42400000+0x40	addi \$v0, \$v1, 0x00000000						
	0x00400041	0x42400000+0x41	addi \$v0, \$v1, 0x00000000						
	0x00400042	0x42400000+0x42	addi \$v0, \$v1, 0x00000000						
	0x00400043	0x42400000+0x43	addi \$v0, \$v1, 0x00000000						
	0x00400044	0x42400000+0x44	addi \$v0, \$v1, 0x00000000						
	0x00400045	0x42400000+0x45	addi \$v0, \$v1, 0x00000000						
	0x00400046	0x42400000+0x46	addi \$v0, \$v1, 0x00000000						
	0x00400047	0x42400000+0x47	addi \$v0, \$v1, 0x00000000						
	0x00400048	0x42400000+0x48	addi \$v0, \$v1, 0x00000000						
	0x00400049	0x42400000+0x49	addi \$v0, \$v1, 0x00000000						
	0x0040004a	0x42400000+0x4a	addi \$v0, \$v1, 0x00000000						
	0x0040004b	0x42400000+0x4b	addi \$v0, \$v1, 0x00000000						
	0x0040004c	0x42400000+0x4c	addi \$v0, \$v1, 0x00000000						
	0x0040004d	0x42400000+0x4d	addi \$v0, \$v1, 0x00000000						
	0x0040004e	0x42400000+0x4e	addi \$v0, \$v1, 0x00000000						
	0x0040004f	0x42400000+0x4f	addi \$v0, \$v1, 0x00000000						
	0x00400050	0x42400000+0x50	addi \$v0, \$v1, 0x00000000						
	0x00400051	0x42400000+0x51	addi \$v0, \$v1, 0x00000000						
	0x00400052	0x42400000+0x52	addi \$v0, \$v1, 0x00000000						
	0x00400053	0x42400000+0x53	addi \$v0, \$v1, 0x00000000						
	0x00400054	0x42400000+0x54	addi \$v0, \$v1, 0x00000000						
	0x00400055	0x42400000+0x55	addi \$v0, \$v1, 0x00000000						
	0x00400056	0x42400000+0x56	addi \$v0, \$v1, 0x00000000						
	0x00400057	0x42400000+0x57	addi \$v0, \$v1, 0x00000000						
	0x00400058	0x42400000+0x58	addi \$v0, \$v1, 0x00000000						
	0x00400059	0x42400000+0x59	addi \$v0, \$v1, 0x00000000						
	0x0040005a	0x42400000+0x5a	addi \$v0, \$v1, 0x00000000						
	0x0040005b	0x42400000+0x5b	addi \$v0, \$v1, 0x00000000						
	0x0040005c	0x42400000+0x5c	addi \$v0, \$v1, 0x00000000						
	0x0040005d	0x42400000+0x5d	addi \$v0, \$v1, 0x00000000						
	0x0040005e	0x42400000+0x5e	addi \$v0, \$v1, 0x00000000						
	0x0040005f	0x42400000+0x5f	addi \$v0, \$v1, 0x00000000						
	0x00400060	0x42400000+0x60	addi \$v0, \$v1, 0x00000000						
	0x00400061	0x42400000+0x61	addi \$v0, \$v1, 0x00000000						
	0x00400062	0x42400000+0x62	addi \$v0, \$v1, 0x00000000						
	0x00400063	0x42400000+0x63	addi \$v0, \$v1, 0x00000000						
	0x00400064	0x42400000+0x64	addi \$v0, \$v1, 0x00000000						
	0x00400065	0x42400000+0x65	addi \$v0, \$v1, 0x00000000						
	0x00400066	0x42400000+0x66	addi \$v0, \$v1, 0x00000000						
	0x00400067	0x42400000+0x67	addi \$v0, \$v1, 0x00000000						
	0x00400068	0x42400000+0x68	addi \$v0, \$v1, 0x00000000						
	0x00400069	0x42400000+0x69	addi \$v0, \$v1, 0x00000000						
	0x0040006a	0x42400000+0x6a	addi \$v0, \$v1, 0x00000000						
	0x0040006b	0x42400000+0x6b	addi \$v0, \$v1, 0x00000000						
	0x0040006c	0x42400000+0x6c	addi \$v0, \$v1, 0x00000000						
	0x0040006d	0x42400000+0x6d	addi \$v0, \$v1, 0x00000000						
	0x0040006e	0x42400000+0x6e	addi \$v0, \$v1, 0x00000000						
	0x0040006f	0x42400000+0x6f	addi \$v0, \$v1, 0x00000000						
	0x00400070	0x42400000+0x70	addi \$v0, \$v1, 0x00000000						
	0x00400071	0x42400000+0x71	addi \$v0, \$v1, 0x						

Now the code will divide the two numbers at register \$s0 it will hold the remainder of the division.

Since  $10 / 5 = 2$  and the remainder is 0 the value of 0x00000000 is shown in the green highlighted box in the registers section.

Name	Number	Value
\$zero	0	0x00000000
\$t0	1	0x10010000
\$t1	2	0x00000005
\$v1	3	0x00000000
\$a0	4	0x00000000
\$a1	5	0x00000004
\$a2	6	0x00000005
\$a3	7	0x00000000
\$t2	8	0x00000000
\$t3	9	0x00000000
\$t4	10	0x00000000
\$t5	11	0x00000000
\$t6	12	0x00000000
\$t7	13	0x00000000
\$s0	14	0x00000000
\$t8	15	0x00000000
\$t9	16	0x00000000
\$t10	17	0x00000000
\$t11	18	0x00000000
\$t12	19	0x00000000
\$t13	20	0x00000000
\$t14	21	0x00000000
\$t15	22	0x00000000
\$t16	23	0x00000000
\$t17	24	0x00000000
\$t18	25	0x00000000
\$t19	26	0x00000000
\$t20	27	0x00000000
\$sp	28	0x10000000
\$gp	29	0x00000000
\$fp	30	0x00000000

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.

Name	Number	Value
\$zero	0	0x00000000
\$t0	1	0x00000004
\$t1	2	0x00000005
\$v1	3	0x00000000
\$a0	4	0x00000000
\$a1	5	0x00000004
\$t2	6	0x00000005
\$t3	7	0x00000000
\$t4	8	0x00000000
\$t5	9	0x00000000
\$t6	10	0x00000000
\$t7	11	0x00000000
\$t8	12	0x00000000
\$t9	13	0x00000000
\$t10	14	0x00000000
\$t11	15	0x00000000
\$t12	16	0x00000000
\$t13	17	0x00000000
\$t14	18	0x00000000
\$t15	19	0x00000000
\$t16	20	0x00000000
\$t17	21	0x00000000
\$t18	22	0x00000000
\$t19	23	0x00000000
\$t20	24	0x00000000
\$sp	25	0x00000000
\$gp	26	0x00000000
\$fp	27	0x00000000
\$hi	28	0x7fffed44
\$lo	29	0x00000000

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

instruction.

Registers:

Name	Number	Value
\$zero	0	0x00000000
\$v0	1	0x15010000
\$v0	2	0x00000005
\$v0	3	0x00000000
\$v0	4	0x00000000
\$a0	5	0x00000005
\$a0	6	0x00000005
\$a0	7	0x00000000
\$a0	8	0x00000000
\$a0	9	0x00000000
\$a0	10	0x00000000
\$t1	11	0x00000000
\$t1	12	0x00000000
\$t1	13	0x00000000
\$t1	14	0x00000000
\$t1	15	0x00000000
\$t1	16	0x00000000
\$t1	17	0x00000000
\$t2	18	0x00000000
\$t2	19	0x00000000
\$t2	20	0x00000000
\$t2	21	0x00000000
\$t2	22	0x00000000
\$t3	23	0x00000000
\$t3	24	0x00000000
\$t3	25	0x00000000
\$t3	26	0x00000000
\$t3	27	0x00000000
\$t3	28	0x00000000
\$t3	29	0x1ffffef0
\$fp	30	0x7ffffef0
\$ra	31	0x00000040
hi		0x00000000
lo		0x00000002

Now the code is loading a1 and a2 and then popping one value from the stack and then at the register

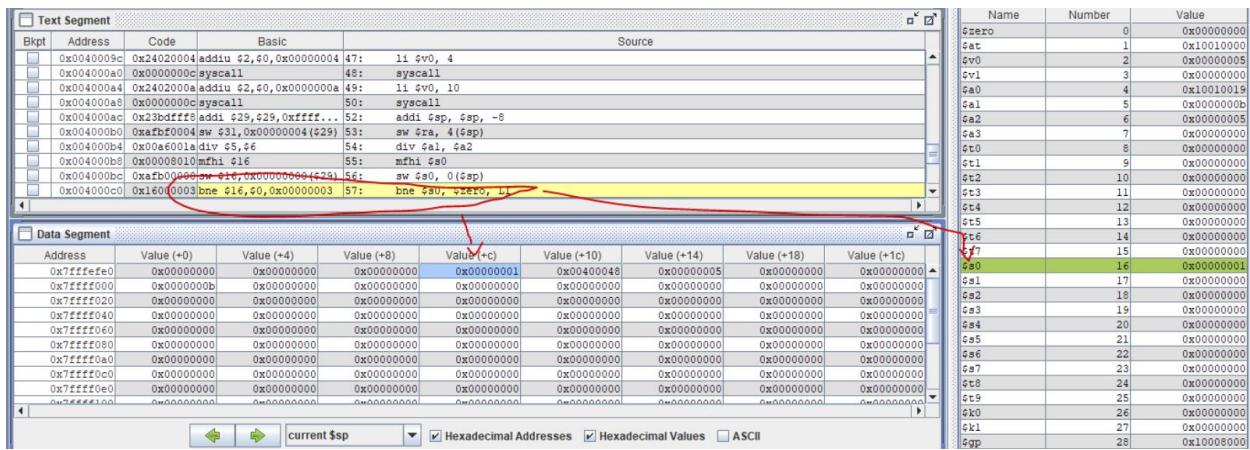
\$s0 it is storing the result of our GCD which in this case is 5.

Registers:

Name	Number	Value
\$zero	0	0x00000000
\$v0	1	0x15010000
\$v0	2	0x00000005
\$v0	3	0x00000000
\$v0	4	0x00000000
\$a0	5	0x00000005
\$a0	6	0x00000005
\$a0	7	0x00000000
\$a0	8	0x00000000
\$a0	9	0x00000000
\$a0	10	0x00000000
\$t1	11	0x00000000
\$t1	12	0x00000000
\$t1	13	0x00000000
\$t1	14	0x00000000
\$t1	15	0x00000000
\$t1	16	0x00000000
\$t1	17	0x00000000
\$t2	18	0x00000000
\$t2	19	0x00000000
\$t2	20	0x00000000
\$t2	21	0x00000000
\$t2	22	0x00000000
\$t3	23	0x00000000
\$t3	24	0x00000000
\$t3	25	0x00000000
\$t3	26	0x00000000
\$t3	27	0x00000000
\$t3	28	0x00000000
\$t3	29	0x1ffffef0
\$fp	30	0x7ffffef0
\$ra	31	0x00000040
hi		0x00000000
lo		0x00000002

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

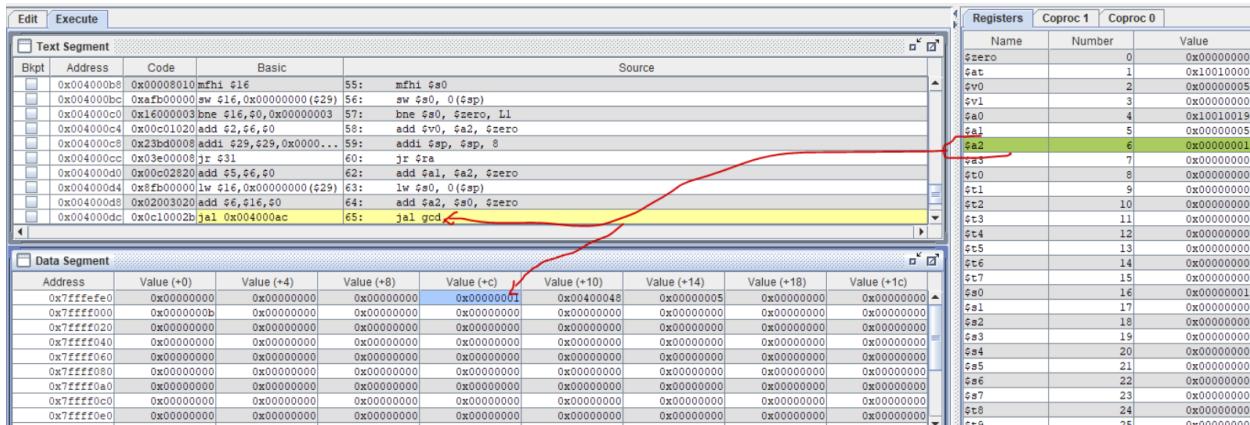
we would have to make is demonstrated below:



With inputs 11 and 5 there is a remainder of 1 which is saved into our stack as the value of 0x00000001.

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.

```
1 #include <stdio.h>
2
3 int gcd(int a, int b) {
4     if (a == 0)
5         return b;
6     return gcd(b % a, a);
7 }
8
9 int main()
10 {
11     int a = 11, b = 5;
12     printf("GCD(%d, %d) = %dn", a, b, gcd(a, b));
13     return 0;
14 }
15 }
```

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
```

No Notification

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

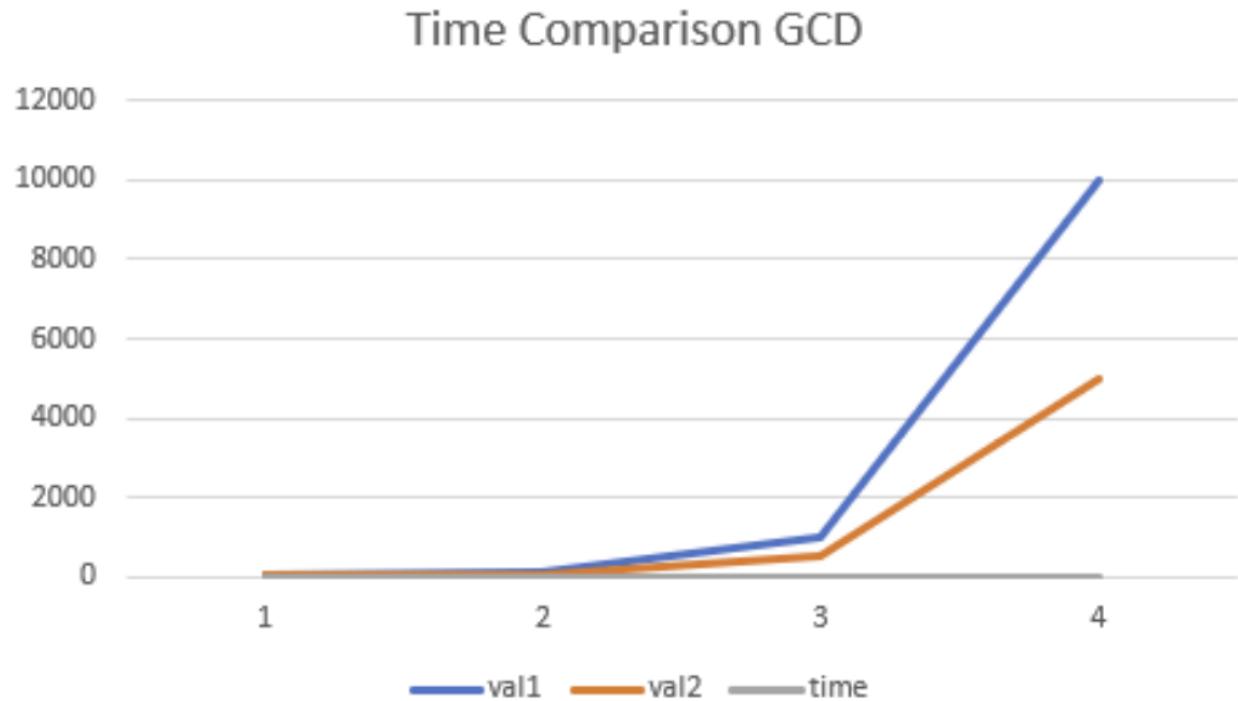
As you can see the registers RSI and RDI holds the values we hard coded in our code. The register RAX holds the dividend and RDX holds the number to divide by.

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:





### 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.