

Take Home TEST 2

Ahmad Adil

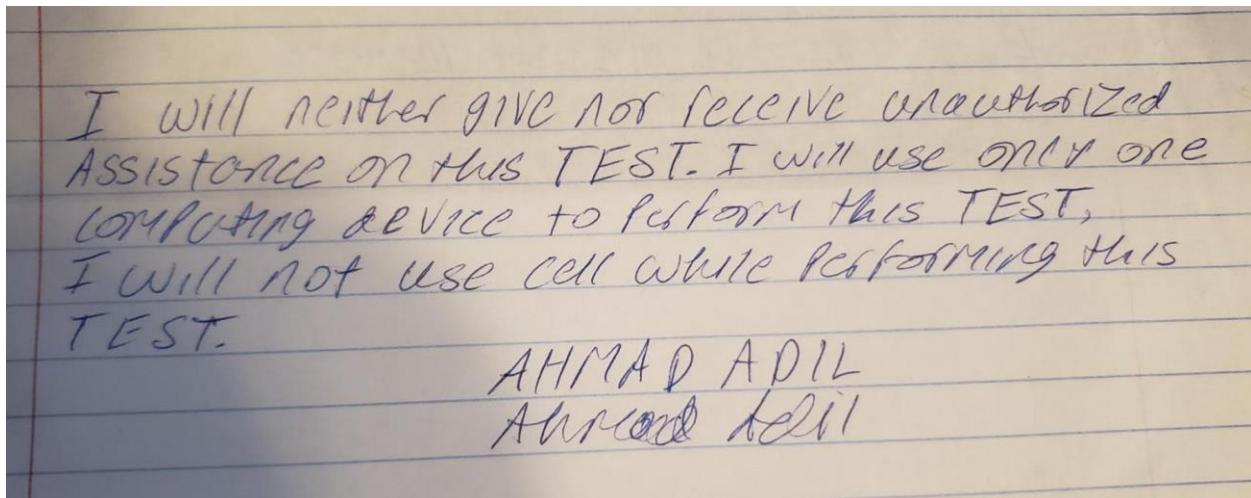
7/15/21

Professor Gertner

Computer Organization 342/343

## Contents

Objective: .....	3
X86 Intel on Windows 32-Bit .....	4
MIPS Within MARS SIMULATOR .....	16
Linux 64-Bit with GCC and GDB.....	27
Conclusion.....	38



Start Time and date: 4:00pm 7/13/21

End TIME and date: 4:00pm 7/14/21

## Objective:

The main objective of this take-home test is to understand how recursive programs and code works in various Instruction Set Architectures. The Processors we are using are 32-bit MIPS processor using MARS Simulator, Intel x86 with a 32-bit compiler in Microsoft Visual Studio environment, and finally Intel x86 processor with 64-bit compiler using Linux GCC and GDB. With these different environments we will compare the how the recursive program operates and be displaying the stack frame to explain the several differences and similarities between them.

## X86 Intel on Windows 32-Bit

```
Disassembly Adil_TAKE_HOME_TEST_2.cpp < X
Adil_TAKE_HOME_TEST_2 (Global Scope)
1 // Adil_TAKE_HOME_TEST_2.cpp : This file contains the 'main' function.  To change this file, edit the template.hxx
2 // Adil Ahmad
3
4 #include <stdio.h>
5
6 int gcd(int a, int b)
7 {
8     if (a == 0)
9         return b;
10    return gcd(b % a, a);
11 }
12
13 int main()
14 {
15     int a = 2, b = 4;
16     printf("GCD(%d,%d)=%dn", a, b, gcd(a, b));
17     return 0;
18 }
```

Figure 1: Adil\_GCD

Here we see the source code for the GCD program. The function takes in two integer values and performs a recursive GCD algorithm between them. The function also returns an integer value. Within main, I created two variables, a and b with them being 2 and 4 respectively. Main then computes the GCD and prints out the value.

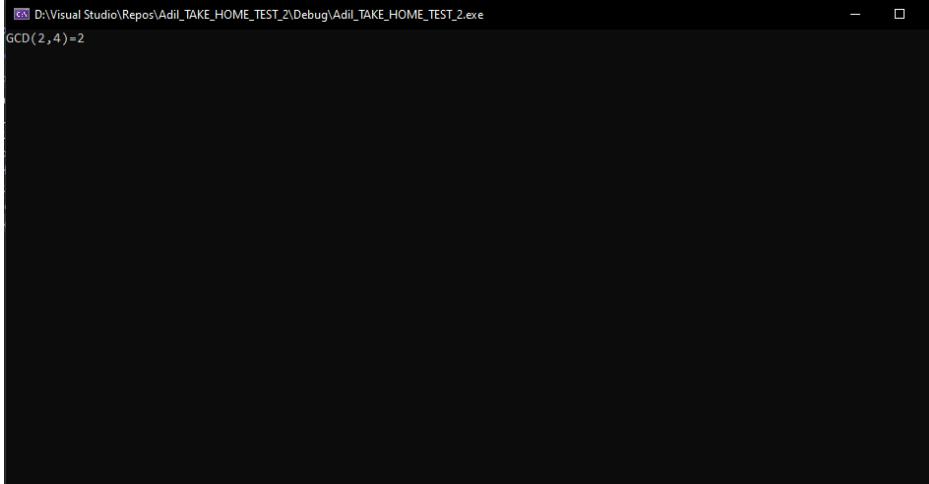


Figure 2: Program output

Disassembly Adil\_TAKE\_HOME\_TEST\_2.cpp

Address: gcd(int, int)

Registers

EAX = 00000004
EBX = 00024000
ECX = 00000002
EDX = 00000001
ESI = 00AD1023
EDI = 00F7FAF8
EIP = 00AD1740
<b>ESP = 00F7FA08</b>
EBP = 00F7FAF8
EFL = 00000246

Memory 2

Address: 0x00F7FA10
0x00F7FA10 04 00 00 00
0x00F7FA14 23 10 ad 00 #...
0x00F7FA18 23 10 ad 00 #...
0x00F7FA20 30 fa f7 00 .0f.
0x00F7FA24 30 fa f7 00 .0f.
0x00F7FA28 65 fb f3 78 .e0d.
0x00F7FA2C 3c fa f7 00 .c4.
0x00F7FA30 b3 ee f3 78 .i0x
0x00F7FA34 03 6d 61 b5 .m4u
0x00F7FA38 03 00 00 00 ....
0x00F7FA3C 58 fa f7 00 X0f.
0x00F7FA40 58 fa f7 00 X0f.
0x00F7FA44 3c 14 f4 78 <.0x
0x00F7FA48 58 fa f7 00 X0f.
0x00F7FA4C b3 ee f3 78 .16x
0x00F7FA50 03 6d 61 b5 .m4u
0x00F7FA54 03 00 00 00 ....
0x00F7FA58 74 fa f7 00 t0f.

Locals

Name	Value
a	2
b	4

Search (Ctrl+E) Search Depth: 3

Memory 1

Address: 0x00F7FA08
0x00F7FA08 b0 25 ad 00 %..
0x00F7FA0C 02 00 00 00 ....
0x00F7FA10 04 00 00 00 ....
0x00F7FA14 23 10 ad 00 #...
0x00F7FA18 23 10 ad 00 #...
0x00F7FA20 30 fa f7 00 .0f.
0x00F7FA24 30 fa f7 00 .0f.
0x00F7FA28 65 fb f3 78 .e0d.
0x00F7FA2C 3c fa f7 00 .c4.
0x00F7FA30 b3 ee f3 78 .i0x
0x00F7FA34 03 6d 61 b5 .m4u
0x00F7FA38 03 00 00 00 ....
0x00F7FA3C 58 fa f7 00 X0f.
0x00F7FA40 58 fa f7 00 X0f.
0x00F7FA44 3c 14 f4 78 <.0x
0x00F7FA48 58 fa f7 00 X0f.
0x00F7FA4C b3 ee f3 78 .16x
0x00F7FA50 03 6d 61 b5 .m4u
0x00F7FA54 03 00 00 00 ....
0x00F7FA58 74 fa f7 00 t0f.

--- D:\Visual Studio\Repos\Adil\_TAKE\_HOME\_TEST\_2\Adil\_TAKE\_HOME\_TEST\_2.cpp

```

1: // Adil_TAKE_HOME_TEST_2.cpp : This file contains the 'main' function.
2: // Adil Ahmad
3:
4: #include <stdio.h>
5:
6: int gcd(int a, int b)
7: {
    00AD1740 55 push    ebp
    00AD1741 8B EC mov     ebp,esp
    00AD1743 81 EC C0 00 00 00 sub    esp,0C0h
    00AD1749 53 push    ebx
    00AD174A 56 push    esi
    00AD174B 57 push    edi
    00AD174C 8B FD mov    edi,ebp
    00AD174E 33 C9 xor    ecx,ecx
    00AD1750 B8 CC CC CC CC mov    eax,0CCCCCCCCh
    00AD1755 F3 AB rep    stos dword ptr es:[edi]
    00AD1757 B9 03 C0 AD 00 mov    ecx,offset _6E4CDE49_Adil_TAKE
    00AD175C E8 AB FF FF call   @_CheckForDebuggerJustMyCode
    8: if (a == 0)
    00AD1761 83 7D 08 00 cmp    dword ptr [a],0
    00AD1765 75 05 jne    __$EncStackInitStart+20h (0AD1)
    9:     return b;
    00AD1767 B8 45 0C mov    eax,dword ptr [b]
    00AD176A EB 14 jmp    __$EncStackInitStart+34h (0AD1)
    10:    return gcd(b % a, a);
    00AD176C B8 45 08 mov    eax,dword ptr [a]

```

Figure 3: ESP stack pointer

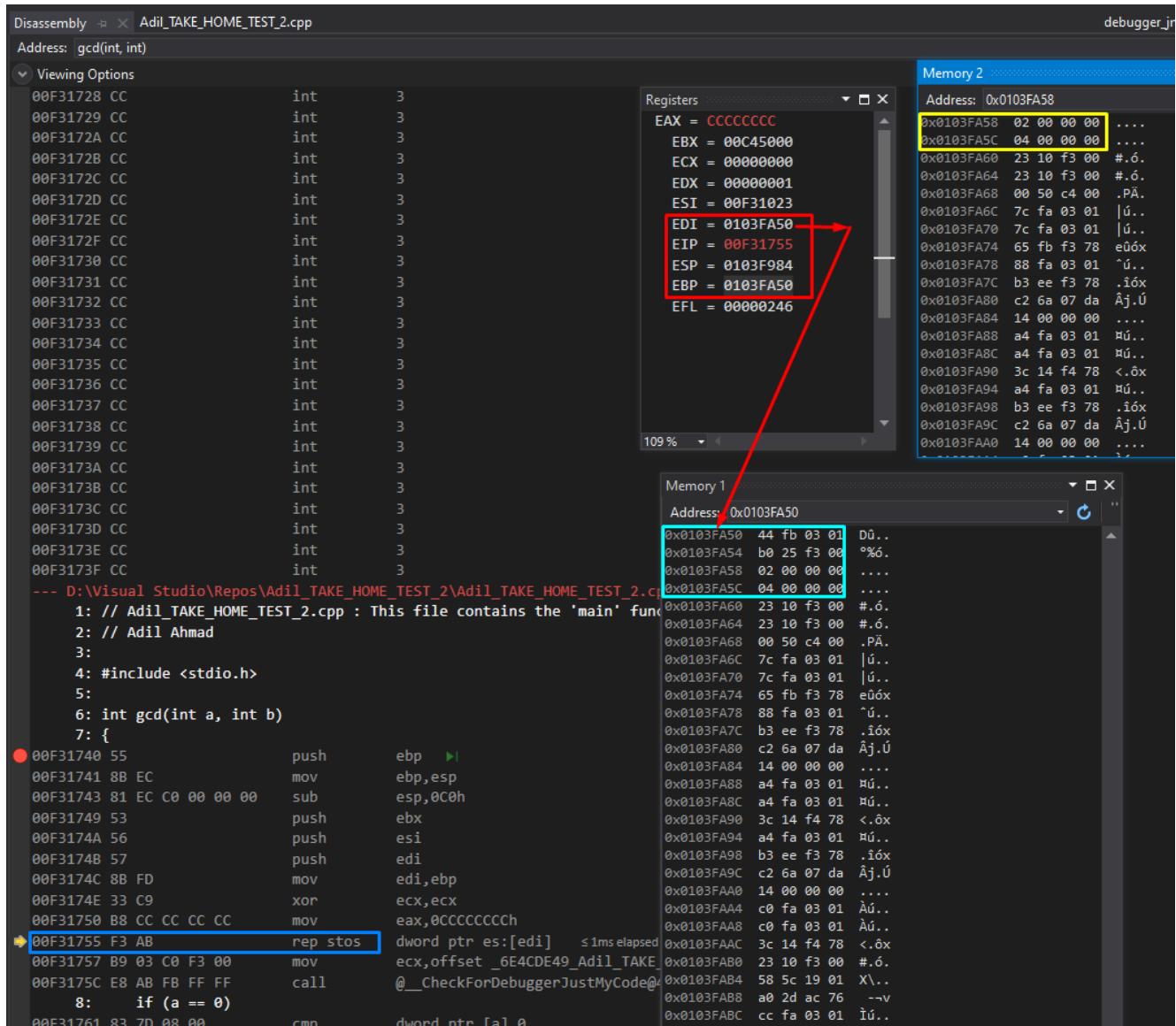


Figure 4: 2 and 4 being loaded into the stack

Within figure 4, we can see that both values, 2 and 4 are being loaded onto the stack. This is known because of two reasons. Within the assembly code we see the `edi` instruction is being called. Also using the base pointer, `0x0103FA50` is offset by `00000008` and that offset is where we can see the current stack.  $0x0103FA50 + 00000008 = 0x0103FA58$ , which is the location of

the value 2, which is variable a, loaded onto the stack. If we go up 4 bytes into the stack once more, we can then see the location of the value 4, which is variable b, loaded onto the stack.

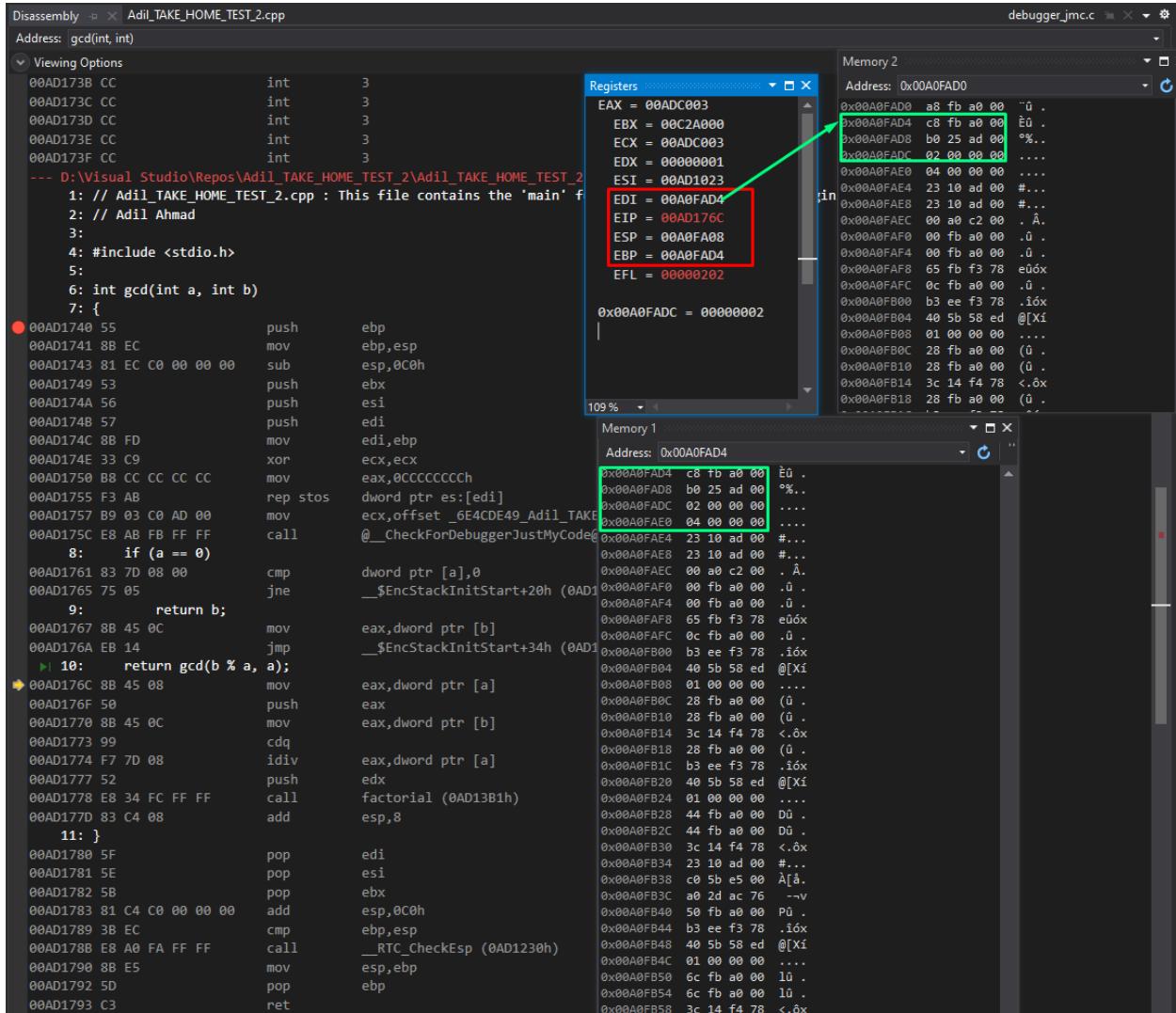


Figure 5: GCD function executing

Next within the stack, the GCD function starts to run. Here within this figure we see that the EIP is going to be executing the next instruction, 0x00F3176C. This instruction is what will run the GCD algorithm against variables a and b. Once the function computes the GCD, it will then store the remainder into a and the GCD value into b.

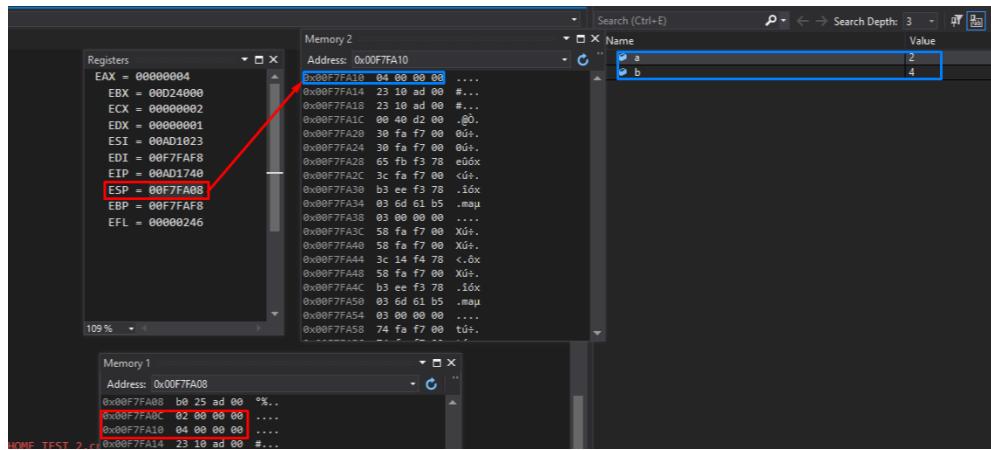


Figure 6: Original value of a and b

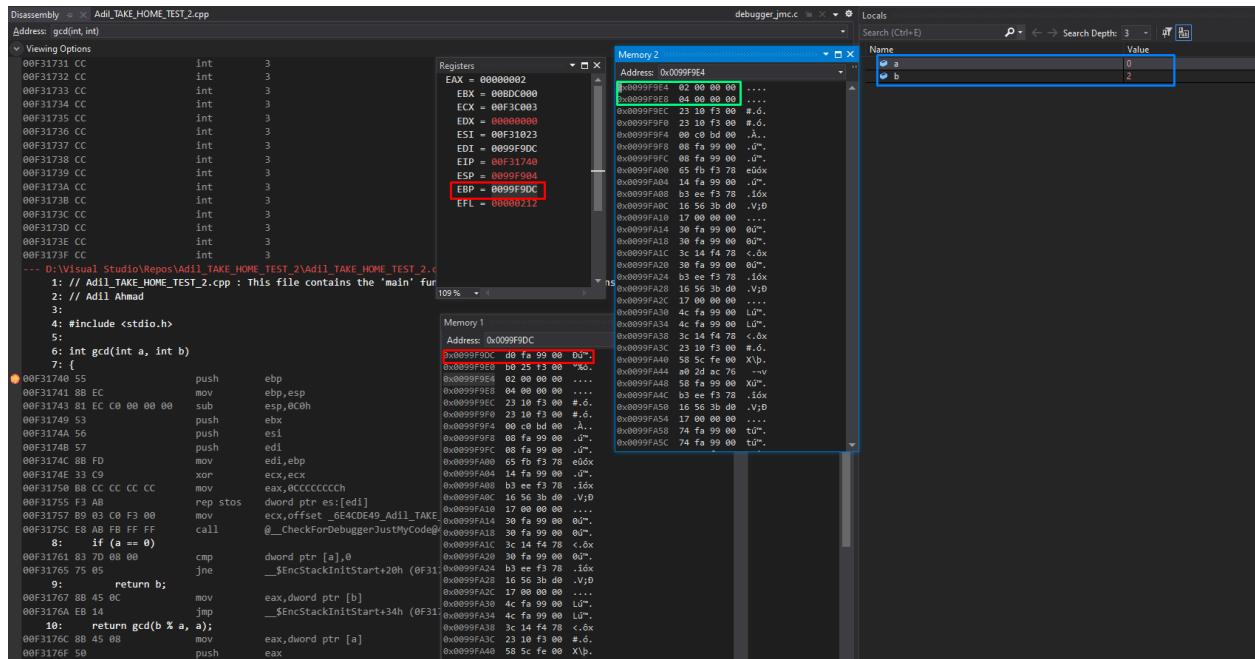


Figure 7: Value of a and b changed

The two figures shown above, indicate that the GCD algorithm has been completed. This also shown by the local variables window. This window contains the type and the value that each variable stores. We see that compared to figure 6, where before the GCD function takes place, that both a and b hold their initial values of 2 and 4. This is also shown and mention before on the stack frame. However, when we get to figure 7, we can see that the locals change and the value of a goes to 0 which is the remainder of the mod operation and that b becomes, which the GCD of 2 and 4. This is important later on for the recursion of the program.

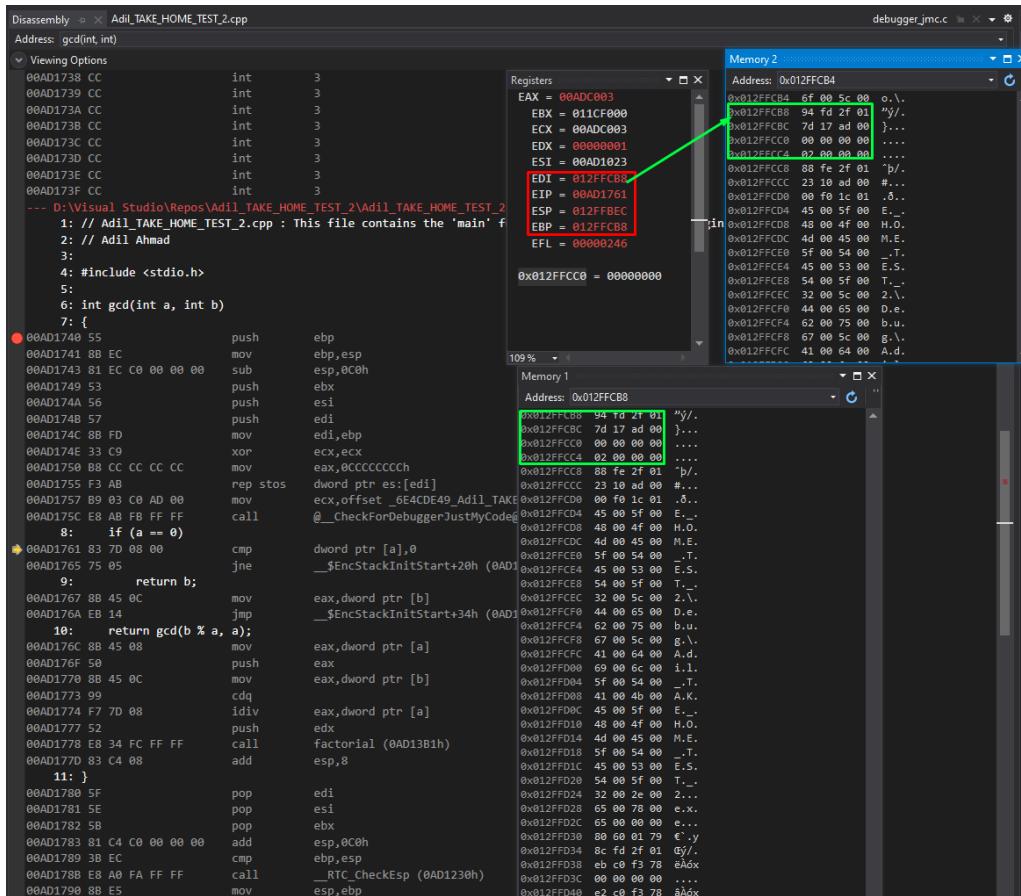


Figure 8: conditional statement on a

This figure above is the conditional check on the variable a to see if it is equal to 0. We know that a has become 0 from the GCD operation on both 2 and 4, so next it will return the value of b. This is shown within the memory window here as well. Within the memory window, we see that the base pointer is 0x012FFCB8 and when we offset that by 8 which what each instruction is offset by, we see the memory location of 0x12ffcc0. This location as shown within the memory window contains 0, as this is what is currently stored within a. Also, we can see the location of the current value of b 4 bytes offset from a, which is 0x12ffcc4.

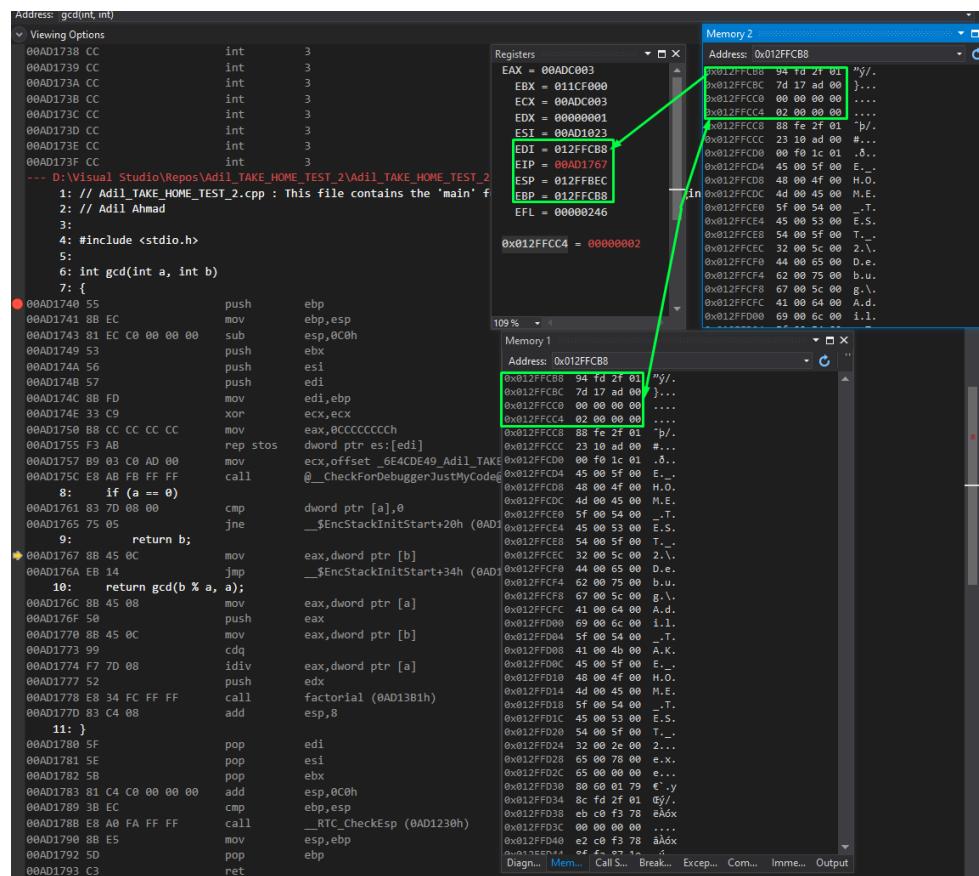


Figure 9: Return b next execution

Figure 9 here shows us what we previous in figure 8, mentioned. Now that the if statement condition was met, the next instruction that will execute is the return value of b. This current value of b is 2. We can confirm this within the memory window with the base pointer offset. Currently the base pointer is 0x012FFCB8, and a +12 offset from that will give us the return value of b. This is located at 0x12ffcc4.

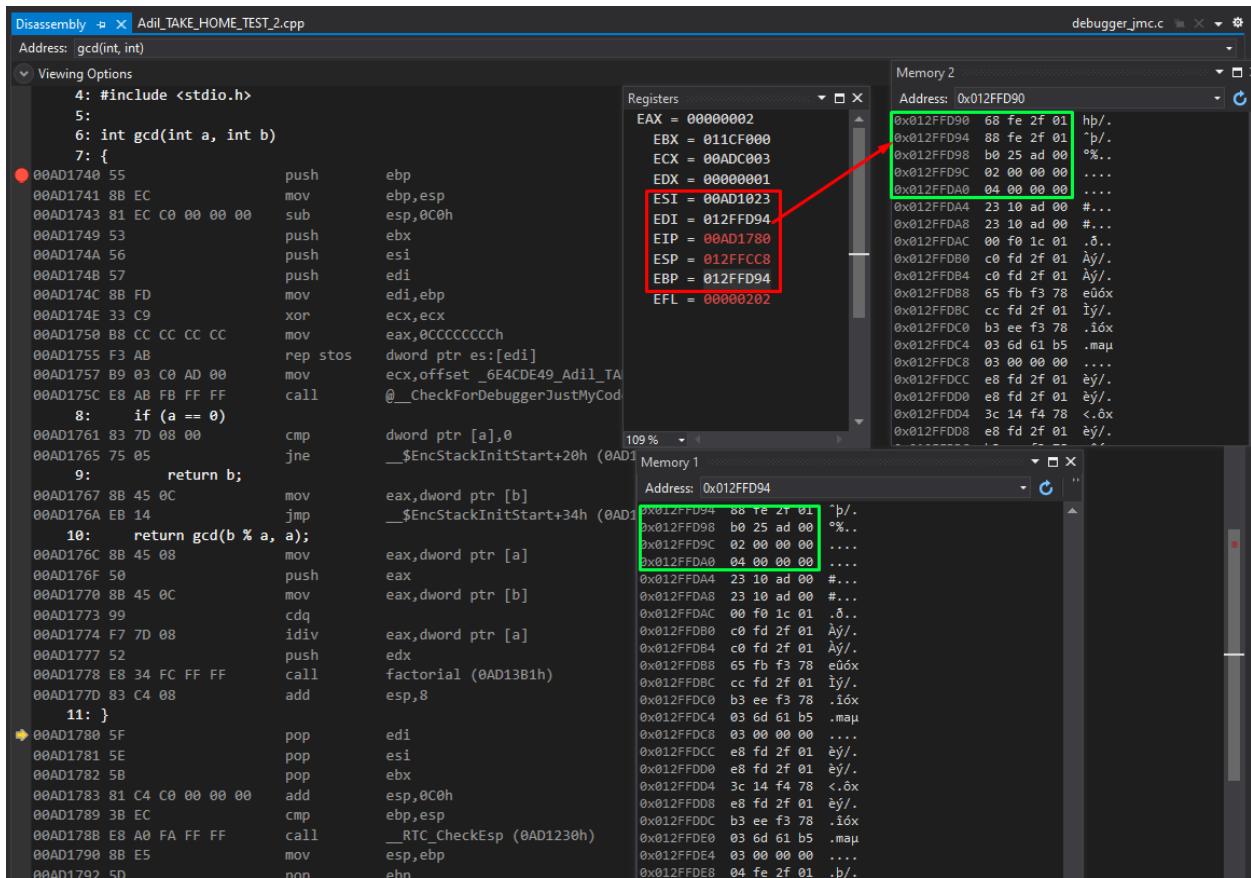


Figure 10: Program exiting GCD function

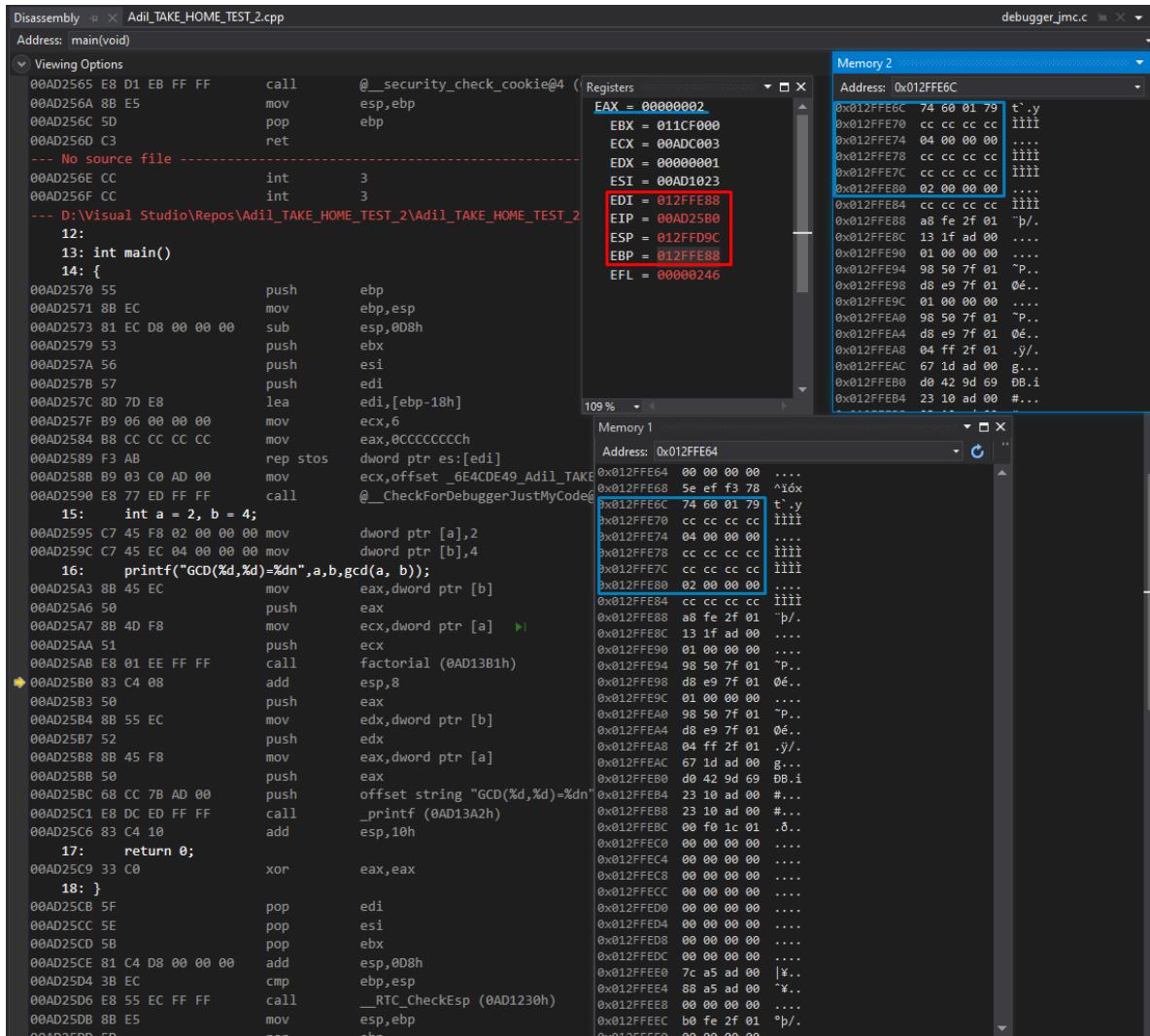


Figure 11: main function start

Figure 11 here shows the main function printing out the results for a and b. Within this we can see the stack frame for both a and b with 2 and 4 as their respective values. The EAX register is currently holding the value of 2, which going to be used for the GCD return value. Main here also calls the GCD function.

Name	Value	Type
gcd returned	2	int
a	2	int
b	4	int

Figure 12: Main calling GCD function

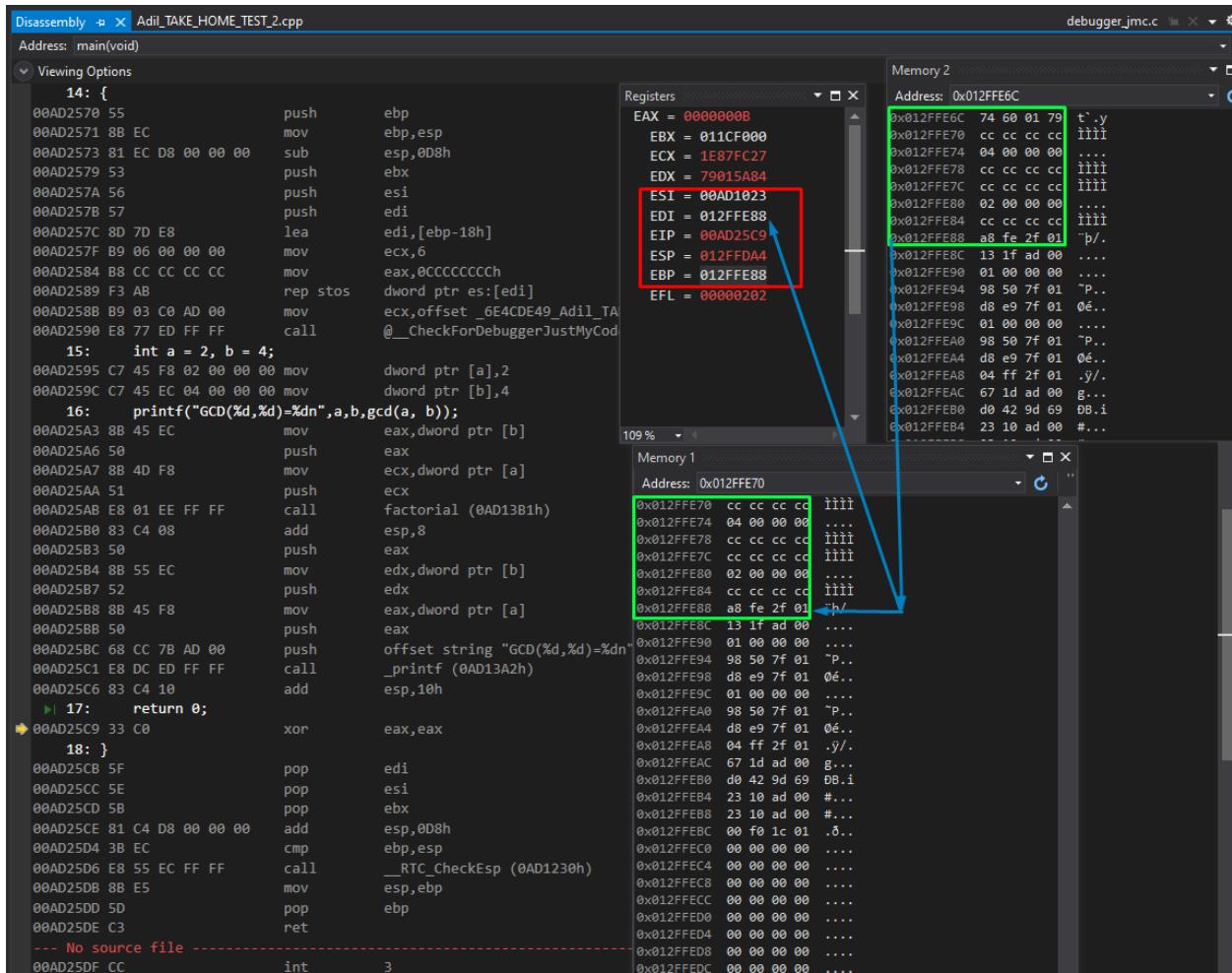


Figure 13: Main returning 0

This figure like figure 12 shows the values of the stack pane after GCD returned 2 as the GCD for both a and b. One thing here we can note is that the GCD value 2 is offset by the EDI register by 8 bytes. This is shown by EDI as 0x012FFE88, which holds the same register as the EBP, base pointer. The value of 2 is at 0x012FFE80 which makes the equation,  $0x012FFE88 - 00000008 = 0x012FFE80$ .

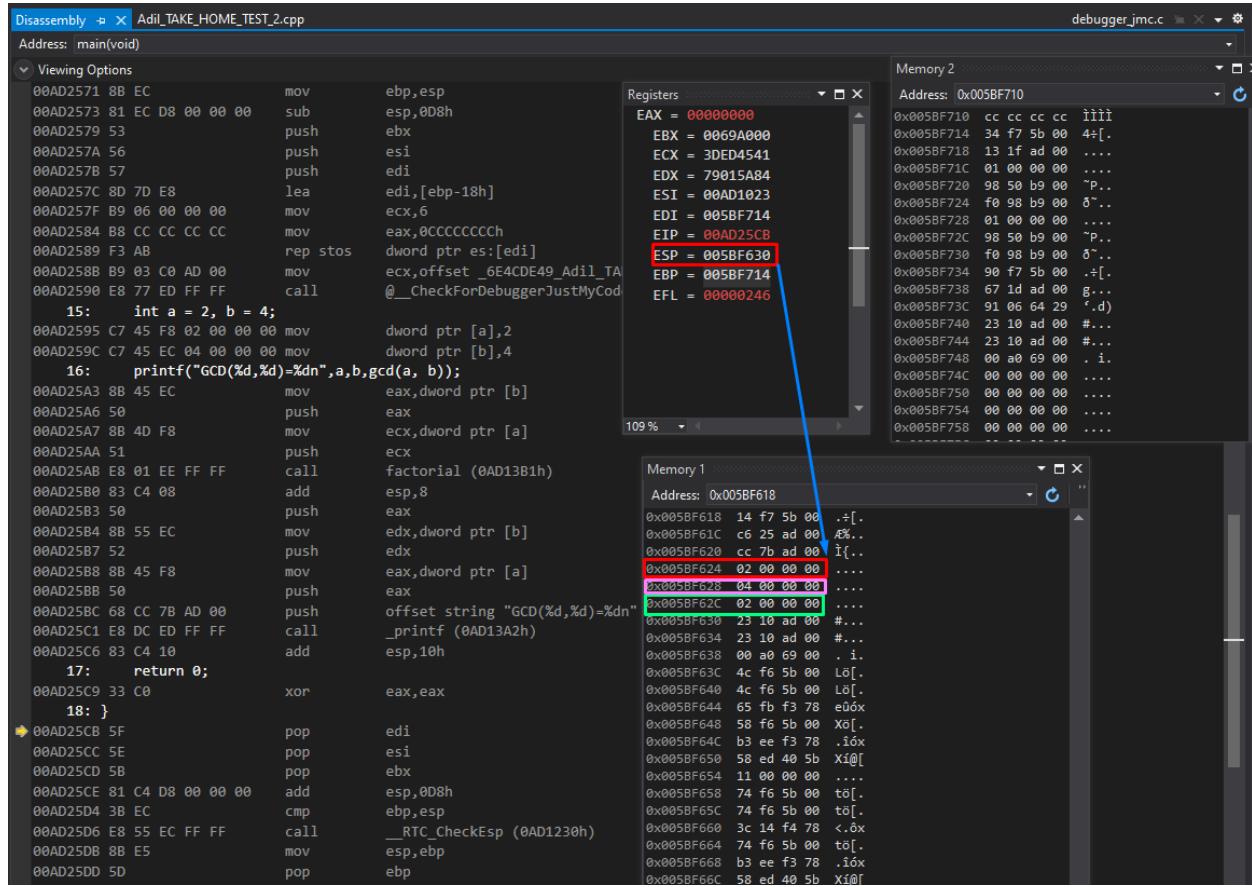


Figure 14 Stack Frame

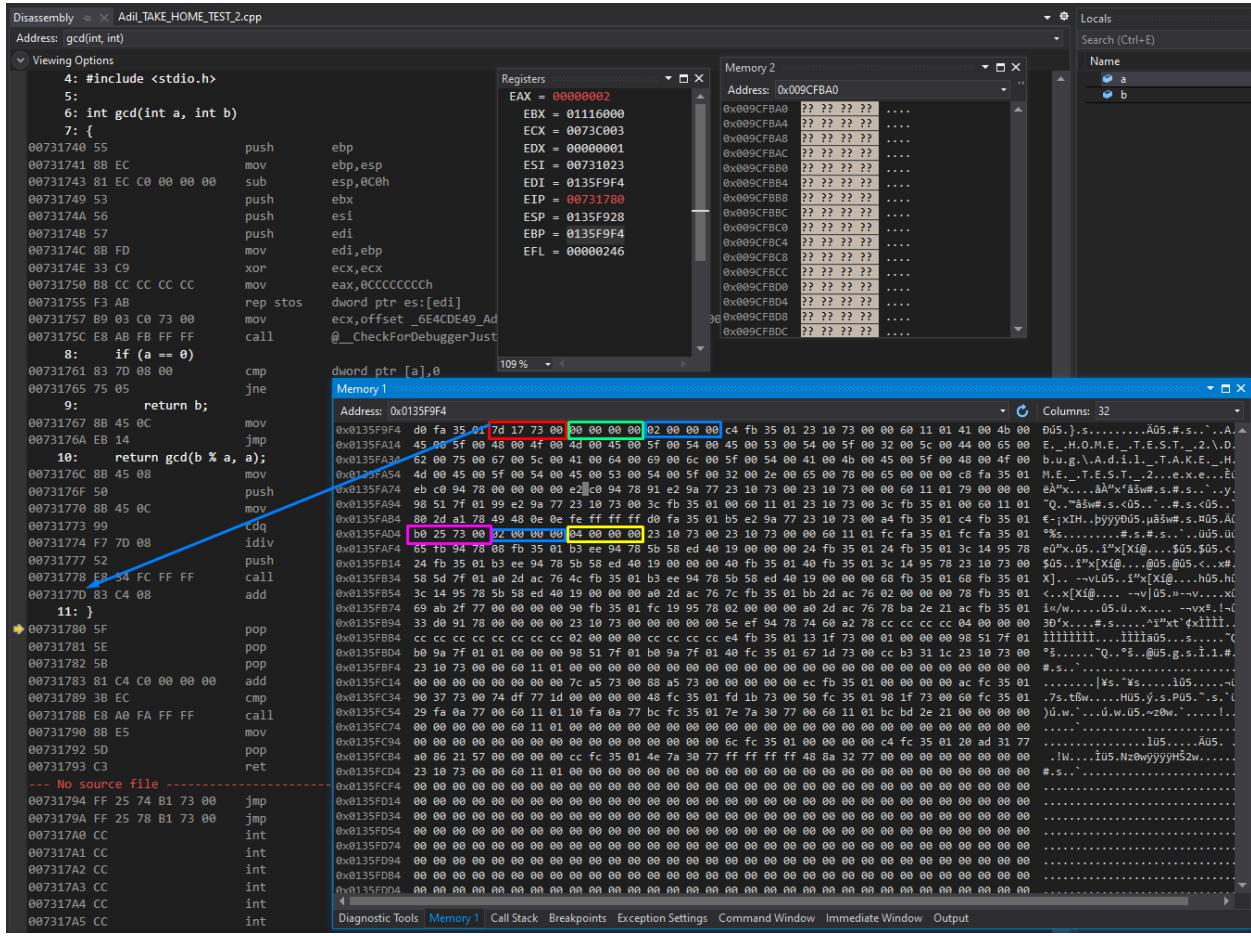


Figure 15: Stack frame

Here we finally see the program fully run and compiled. Within the memory 1 window we can see the stack frame present there. This is found using the ESP, or Stack pointer. The stack pointer currently is 0x005BF630. Before that we see the value of 2, at 0x005BF62C. This is most likely the GCD return value at the end of the main function. If subtract 4 bytes offset from that we can see the value of 4, at memory location, 0x005BF628. This indicates when 4 was loaded into the stack when the program first ran. And to conclude, 4 bytes before that register, we see the original value for a at 2 within memory location, 0x005BF624 which was the first value loaded into variable a.

## MIPS Within MARS SIMULATOR

```
Adil_GCD_RECUSION.asm
1 .data
2 a: .word 2
3 b: .word 4
4 .text
5 main:
6 lw $a0,a # load value a
7 lw $al,b #load value a
8 jal GCD # call function GCD
9
10 add $a0,$v0,$zero
11 li $v0,1
12 syscall # print result
13 li $v0, 10 # exit program
14 syscall
15
16 GCD:
17 addi $sp, $sp, -12
18 sw $ra, 0($sp) # save function into stack
19 sw $s0, 4($sp) # save value $s0 into stack
20 sw $sl, 8($sp) # save value $sl into stack
21
22 add $s0, $a0, $zero # s0 = a0 ( value a )
23 add $sl, $al, $zero # sl = al ( value b )
24
25 addi $tl, $zero, 0 # $tl = 0
26 beq $sl, $tl, EXIT # if sl == 0 EXIT
27
28 add $a0, $zero, $sl # make a0 = $sl
29 div $s0, $sl # a/b
30 mfhi $al # remainder of 2/4 which is equal to a%b
31 jal GCD
32
33 EXITGCD:
34 lw $ra, 0 ($sp) # read registers from stack
35 lw $s0, 4 ($sp)
36 lw $sl, 8 ($sp)
37 addi $sp,$sp , 12 # bring back stack pointer
38 jr $ra
39 EXIT:
40 add $v0, $zero, $s0 # return $v0 = $s0 (2)
41 j EXITGCD
```

Figure 16: Adil\_GCD\_RECUSION.asm

For this MIPS 32-bit MARS SIMULATION assembly code, we can see that the program is, computing the GCD of variables (a,b) using recursive version of EUCLIDEAN algorithm for two integers a>0, b>0. The values of a and b here are 2 and 4 respectively. This program utilizes function calls within MIPS to compute the algorithm.

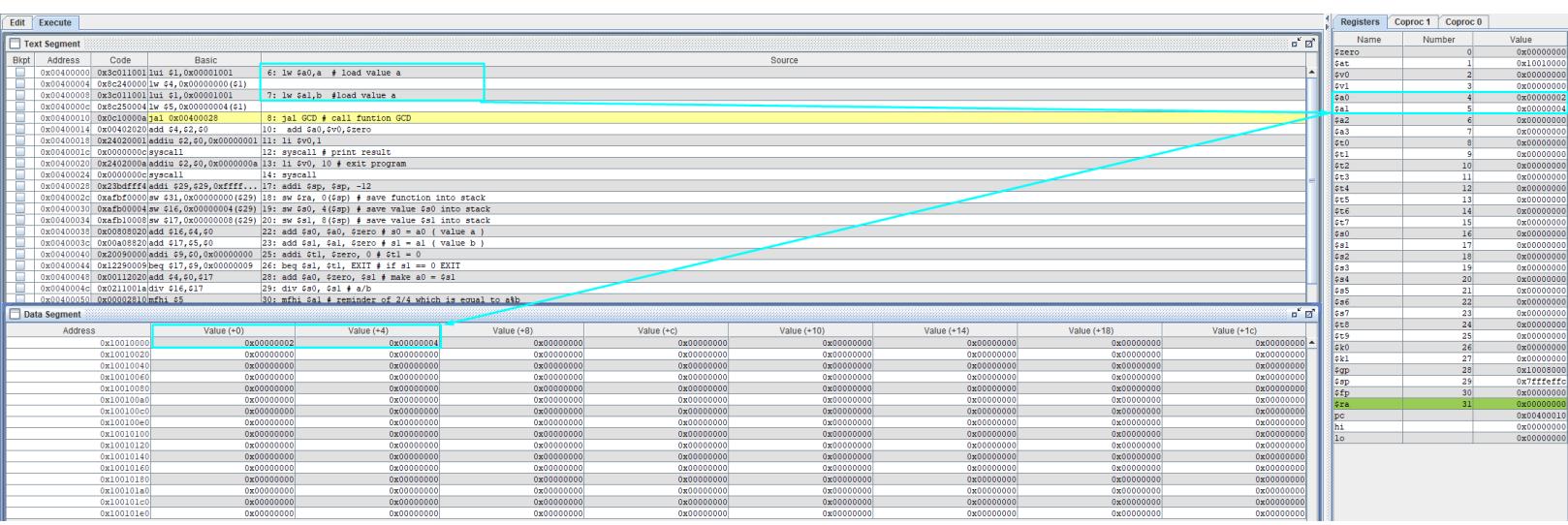


Figure 17: Variables a and b loaded into the registers

The first figure shown here show us that both variables a and b have been loaded onto the data segment and they have been loaded into their respective registers. The instruction that is highlighted is the next instruction to be executed next. This will call the GCD function to execute. They are loaded into register. Variable a is loaded into register \$a0 and variable b is loaded into register \$a1.

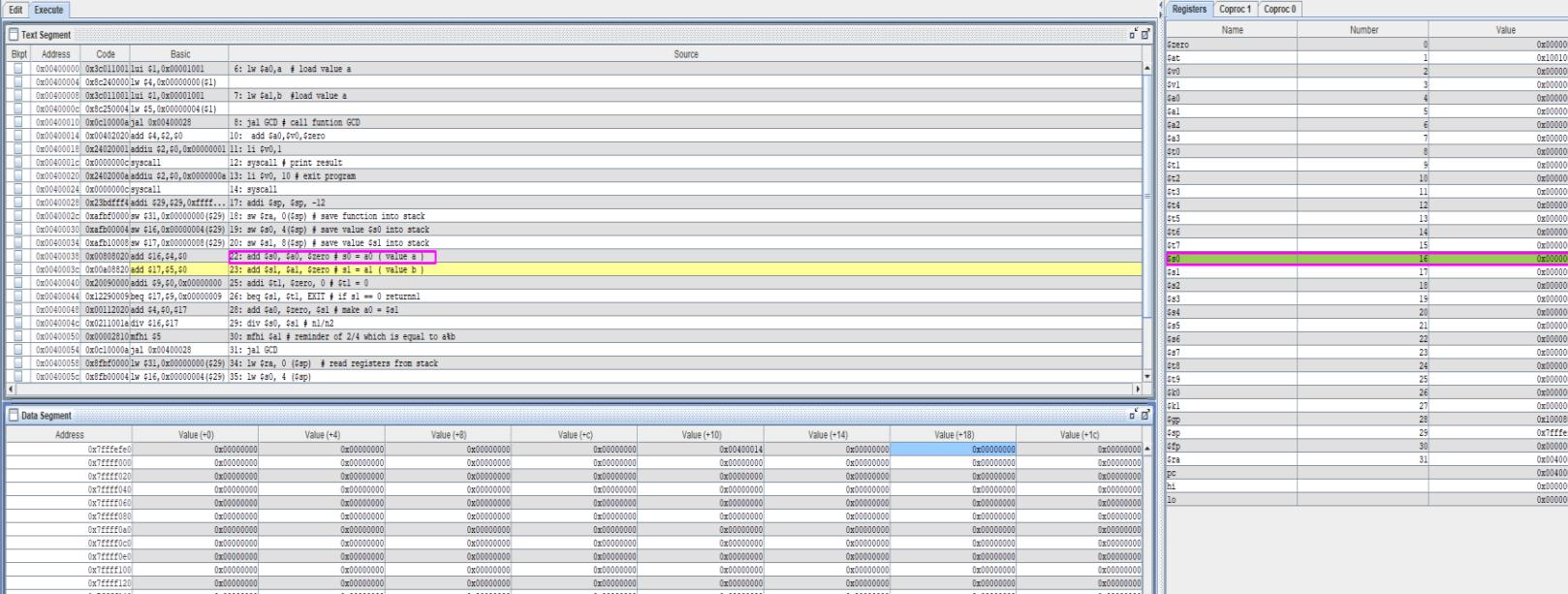


Figure 18: Variable loaded on stack

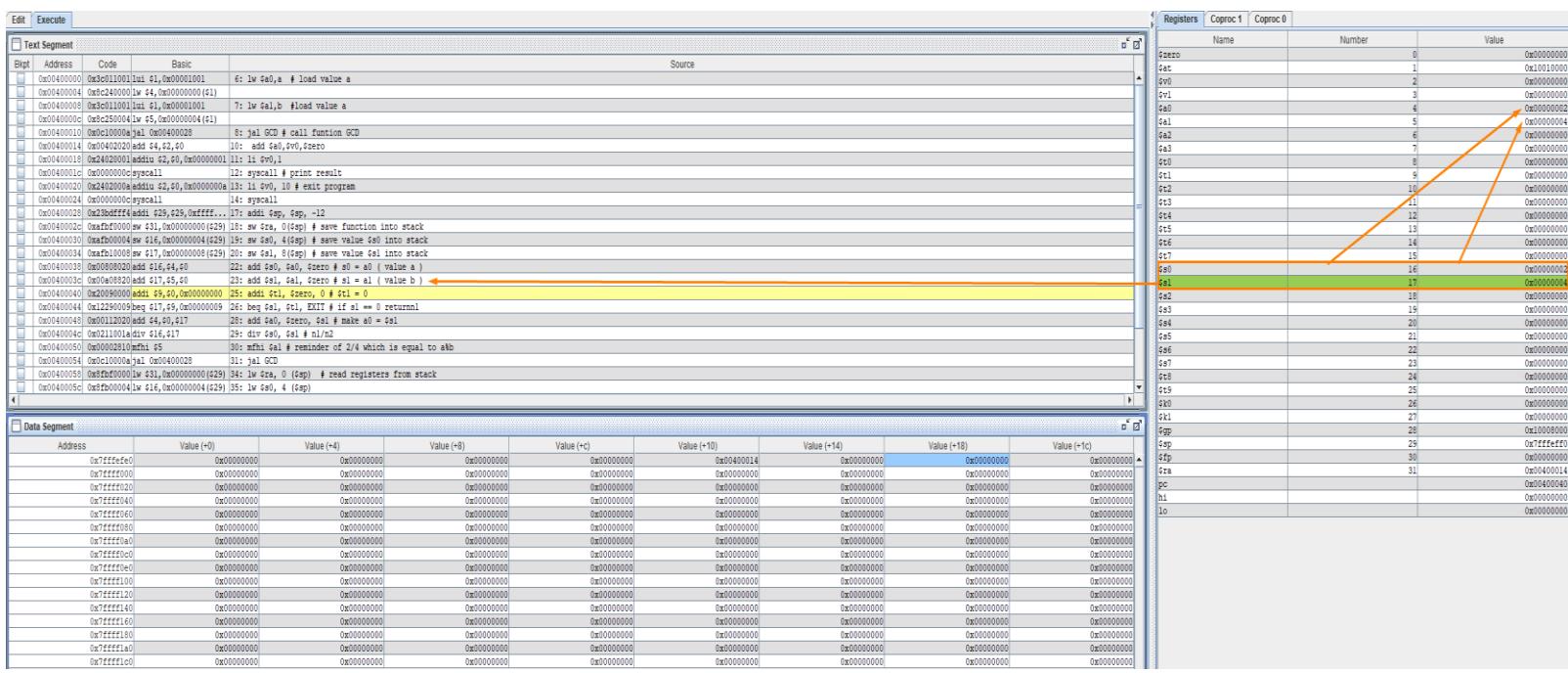


Figure 19: variable b on stack

These two figures that follow show that the variables, a and b have now been loaded onto the stack.

This is important as the GCD algorithm will be conducted on the stack. Here we can also see that the

program counter register, \$pc is pointing to the next instruction. This instruction is 0x004000040.

Currently the program is within the GCD function computing the greatest common divider between the two variables.

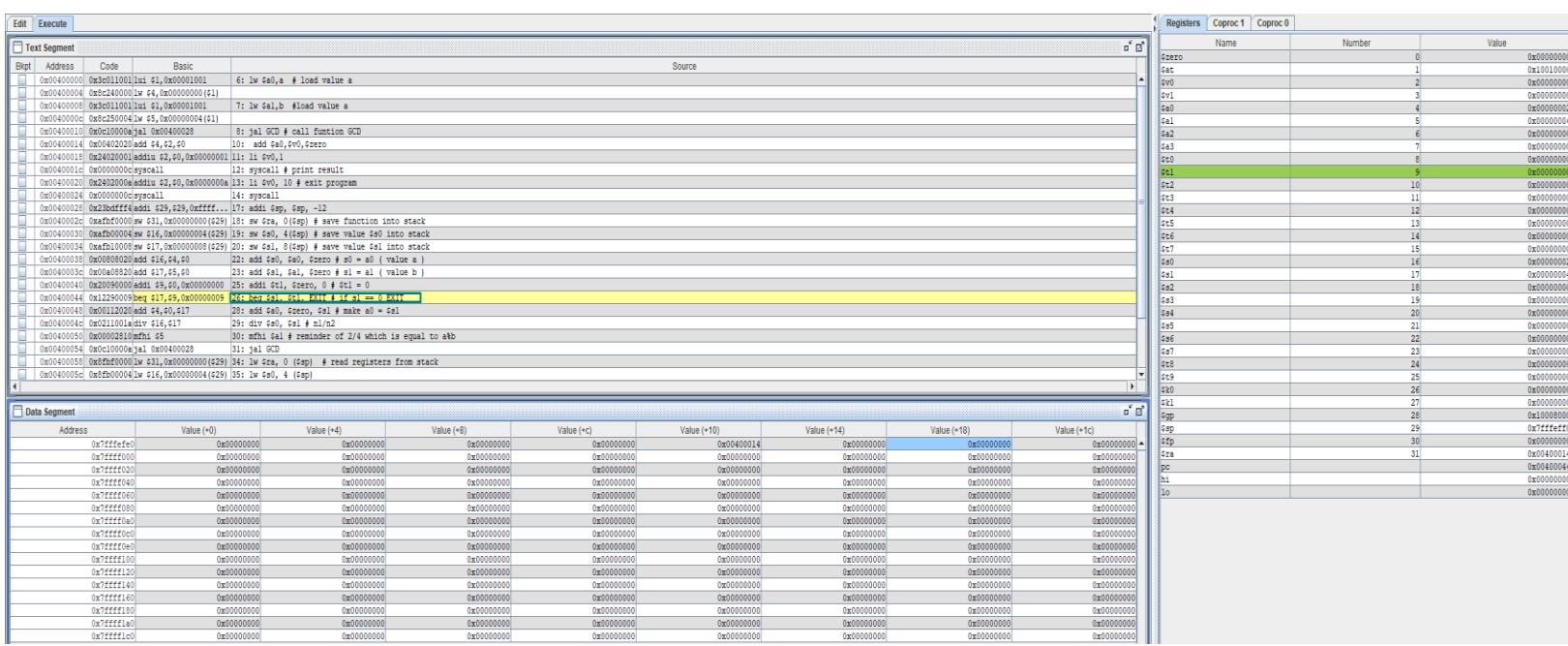


Figure 20: checking if a is currently equal to 0.

This figure above here shows us the condition check statement for is a equal to 0. This is the same to the if-else statement within C language. This is shown with the beq, instruction will check if the contents of registers \$t1 and \$s1, which contain the value of b and 0 are the same. This is the if( $b == 0$ ) then the program will jump to the EXIT statement. But we know they are not equal so, it will continue on to the module operation.

7/15/21

WIS, 21

CSC 342/343

The screenshot shows the Immunity Debugger interface with two main panes: the Assembly pane on the left and the Registers pane on the right.

**Assembly Pane:**

- Text Segment:**
  - Address: 0x00400000 - 0x0040005C
  - Code:

```

        .text
        .file "main.c"
        .section .text
        .globl _start
_start:
        .quad 0x400000
        .quad 0x400004
        .quad 0x400008
        .quad 0x40000C
        .quad 0x400010
        .quad 0x400014
        .quad 0x400018
        .quad 0x400020
        .quad 0x400024
        .quad 0x400028
        .quad 0x40002C
        .quad 0x400030
        .quad 0x400034
        .quad 0x400038
        .quad 0x400040
        .quad 0x400044
        .quad 0x400048
        .quad 0x40004C
        .quad 0x400050
        .quad 0x400054
        .quad 0x400058
        .quad 0x40005C
    
```
- Basic:**
  - Instruction details for each address, including opcodes, addresses, and comments like '# load value a' or '# save function into stack'.
- Source:**
  - Shows the C code corresponding to the assembly instructions.

**Registers Pane:**

Registers	Coproc 1	Coproc 0
Name	Number	Value
rzero	0	0xffffffff
sat	1	0x10010000
sp0	2	0xffffffff
cr1	3	0xffffffff
cr1	5	0x00000000
cr2	6	0xffffffff
cr3	7	0xffffffff
cr5	8	0xffffffff
cr7	9	0xffffffff
cr2	10	0xffffffff
cr3	11	0xffffffff
cr4	12	0xffffffff
cr5	13	0xffffffff
cr6	14	0xffffffff
cr7	15	0xffffffff
cr9	16	0xffffffff
cr1	17	0xffffffff
cr2	18	0xffffffff
cr3	19	0xffffffff
cr4	20	0xffffffff
cr5	21	0xffffffff
cr6	22	0xffffffff
cr7	23	0xffffffff
cr8	24	0xffffffff
cr9	25	0xffffffff
cr1	26	0xffffffff
cr2	27	0xffffffff
cr3	28	0xffffffff

*Figure 22: Division statement*

*Figure 21:module function*

These two figures show the module function working within the program. Here within the first figure we see the division portion take place between a and b. We know that b divided by a will give us 2 as the quotient. This is important as in a module operation there is the quotient and a remainder that determine if the module is 0 or 1. Within figure 21 we see that the \$HI register which is a special register used for store the result of multiplication and division. Their contents are accessed with special instructions mfhi and mflo. The \$HI register, is where high-order 32 bits are stored. For the \$LO, this is where low-order 32 bits are stored.

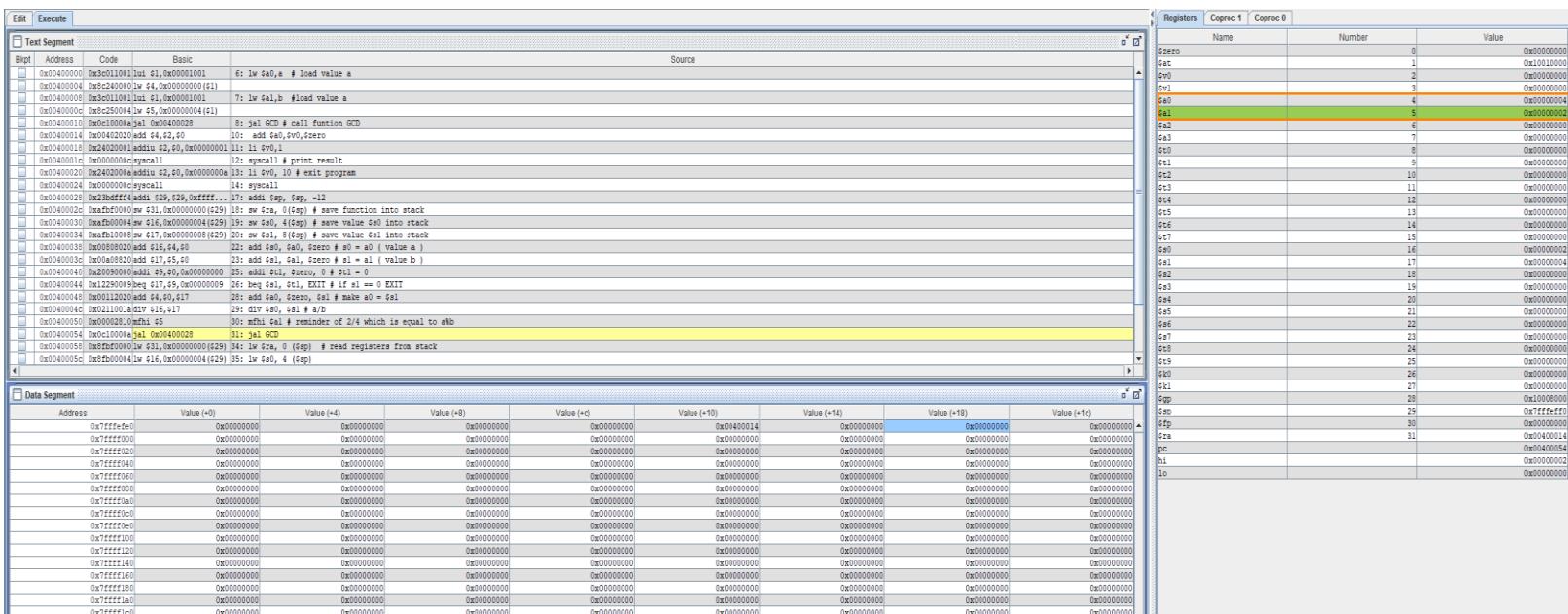


Figure 23: GCD executing again

Text Segment

Byte	Address	Code	Basic	Source
0x04000000	0x00110000	lui \$1,0x00001001	6: lw \$a0,a # load value a	
0x04000004	0x00240000	lw \$4,0x00000000(\$1)	7: lw \$a1,b #load value a	
0x04000008	0x00110000	lui \$1,0x00001001	8: jal \$0,GCD # call function GCD	
0x0400000C	0x00230000	lw \$5,0x00000004(\$1)	10: add \$a0,\$v0,\$zero	
0x04000010	0x00100000	jal 0x00400028	11: li \$v0,1	
0x04000014	0x00240200	add \$2,\$a2,\$0	12: syscall # print result	
0x04000018	0x00240000	addiu \$2,\$a2,\$0	13: li \$v0,10 # exit program	
0x04000022	0x00000000	syscall	14: sys	
0x04000026	0x0023ffff	add \$29,\$a2,\$0xffff...	15: addi \$sp,\$sp,-12	
0x04000028	0x00100000	sw \$1,0x00000000(\$29)	18: sw \$t1, 0(\$sp) # save function into stack	
0x04000030	0x00230000	lw \$16,0x00000004(\$29)	19: lw \$a0, 4(\$sp) # save value \$a0 into stack	
0x04000034	0x00100000	sw \$17,0x00000008(\$29)	20: sw \$t1, 8(\$sp) # save value \$a1 into stack	
0x04000038	0x00100000	add \$16,\$a0	22: add \$a0,\$a0,\$zero # \$0 = a0 ( value a )	
0x0400003C	0x00240200	add \$17,\$a1,\$0	23: add \$a1,\$a1,\$zero # \$1 = a1 ( value b )	
0x04000040	0x00230000	addiu \$17,\$a1,\$0	25: addi \$t1,\$zero, 0 # \$t1 = 0	
0x04000044	0x00230000	lw \$15,0x00000008	26: beg \$t1, t1, EXIT if t1 == 0 EXIT	
0x04000048	0x00110200	add \$14,\$a1,\$17	28: add \$a1,\$zero,\$a1 # make a0 = \$a1	
0x0400004C	0x00211010	div \$16,\$a1,\$17	29: div \$a1,\$a1,\$a1 # a/b	
0x04000050	0x00100000	mfhi \$t1	30: mfhi \$t1 # remainder of 2/a which is equal to ab	
0x04000054	0x00100000	jal 0x00400028	31: jal GCD	
0x04000058	0x00230000	lw \$31,0x00000008(\$29)	32: lw \$t1, 0 (\$sp) # read registers from stack	
0x0400005C	0x00230000	lw \$31,0x00000004(\$29)	33: lw \$s0, 4 (\$sp)	

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x7ffffe0	0x00000000	0x00400058	0x00000002	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7fffff000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7fffff100	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7fffff140	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7fffff160	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7fffff180	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7fffff1a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7fffff1c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

Registers

Name	Number	Value
\$zero	0	0x00000000
\$v0	1	0x10010000
\$v1	2	0x00000000
\$t1	3	0x00000000
\$a0	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	0x00000000
\$s1	17	0x00000000
\$s2	18	0x00000000
\$s3	19	0x00000000
\$a4	20	0x00000000
\$s5	21	0x00000000
\$s6	22	0x00000000
\$t8	23	0x00000000
\$t9	24	0x00000000
\$t10	25	0x00000000
\$t11	26	0x00000000
\$t12	27	0x00000000
\$t13	28	0x00000000
\$sp	29	0x7fffffe1
\$fp	30	0x00000000
\$ra	31	0x00400034
pc		0x00000000
hi		0x00000002
lo		0x00000000

Figure 24: Placed onto the stack

Figure 23 here shows us that the quotient has been stored within the stack. This is shown within the data segment. Within the data segment, we see that the value of 2 is stored, 8 bytes from the base stack address at  $0x7fffffe0 + 00000008$ . Within the register window we can also see the previous values of a and b stored within the \$a0 and \$a1 along with \$s0 and \$s1 registers. The \$pc register is also pointing to the next instruction at  $0x004000058$ .

Text Segment

Byte	Address	Code	Basic	Source
0x04000000	0x00110000	lui \$1,0x00001001	6: lw \$a0,a # load value a	
0x04000004	0x00240000	lw \$4,0x00000000(\$1)	7: lw \$a1,b #load value a	
0x04000008	0x00110000	lui \$1,0x00001001	8: jal \$0,GCD # call function GCD	
0x0400000C	0x00230000	lw \$5,0x00000004(\$1)	10: add \$a0,\$v0,\$zero	
0x04000010	0x00100000	jal 0x00400028	11: li \$v0,1	
0x04000014	0x00240200	add \$2,\$a2,\$0	12: syscall # print result	
0x04000018	0x00240000	addiu \$2,\$a2,\$0	13: li \$v0,10 # exit program	
0x04000022	0x00000000	syscall	14: sys	
0x04000026	0x0023ffff	add \$29,\$a2,\$0xffff...	15: addi \$sp,\$sp,-12	
0x04000028	0x00100000	sw \$1,0x00000000(\$29)	18: sw \$t1, 0(\$sp) # save function into stack	
0x04000030	0x00230000	lw \$16,0x00000004(\$29)	19: lw \$a0, 4(\$sp) # save value \$a0 into stack	
0x04000034	0x00100000	sw \$17,0x00000008(\$29)	20: sw \$t1, 8(\$sp) # save value \$a1 into stack	
0x04000038	0x00100000	add \$16,\$a0	22: add \$a0,\$a0,\$zero # \$0 = a0 ( value a )	
0x0400003C	0x00240200	add \$17,\$a1,\$0	23: add \$a1,\$a1,\$zero # \$1 = a1 ( value b )	
0x04000040	0x00230000	addiu \$17,\$a1,\$0	25: addi \$t1,\$zero, 0 # \$t1 = 0	
0x04000044	0x00230000	lw \$15,0x00000008	26: beg \$t1, t1, EXIT if t1 == 0 EXIT	
0x04000048	0x00110200	add \$14,\$a1,\$17	28: add \$a1,\$zero,\$a1 # make a0 = \$a1	
0x0400004C	0x00211010	div \$16,\$a1,\$17	29: div \$a1,\$a1,\$a1 # a/b	
0x04000050	0x00100000	mfhi \$t1	30: mfhi \$t1 # remainder of 2/a which is equal to ab	
0x04000054	0x00100000	jal 0x00400028	31: jal GCD	
0x04000058	0x00230000	lw \$31,0x00000008(\$29)	32: lw \$t1, 0 (\$sp) # read registers from stack	
0x0400005C	0x00230000	lw \$31,0x00000004(\$29)	33: lw \$s0, 4 (\$sp)	

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x7ffffe0	0x00000000	0x00400058	0x00000002	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7fffff000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7fffff100	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7fffff140	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7fffff160	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7fffff180	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7fffff1a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x7fffff1c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

Registers

Name	Number	Value
\$zero	0	0x00000000
\$v0	1	0x10010000
\$v1	2	0x00000000
\$a0	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	0x00000000
\$s1	17	0x00000004
\$s2	18	0x00000000
\$s3	19	0x00000000
\$a4	20	0x00000000
\$s5	21	0x00000000
\$s6	22	0x00000000
\$t8	23	0x00000000
\$t9	24	0x00000000
\$t10	25	0x00000000
\$t11	26	0x00000000
\$t12	27	0x00000000
\$t13	28	0x00000000
\$sp	29	0x00000000
\$fp	30	0x00000000
\$ra	31	0x00400034
pc		0x00000000
hi		0x00000002
lo		0x00000000

Figure 25: 4 being copied into \$a0 and \$a1

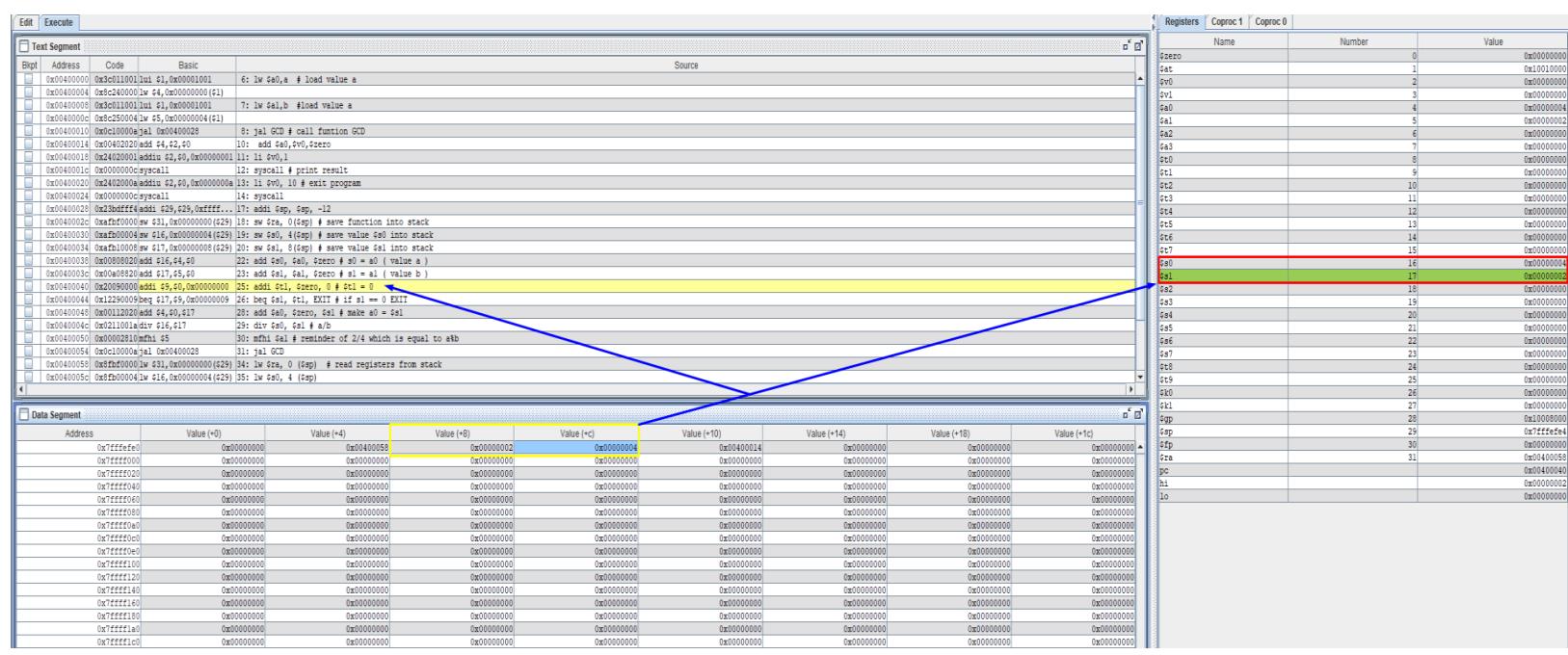


Figure 26: GCD between 0 and 2

This figure here shows us the stack and register before the second time GCD within the program will

execute. We know that after its execution it will exit the function and write 2 into the stack only. One

thing to note is the registers of \$s0 and \$s1 now contain the values of each other. This means that \$s0

contains, 4 and \$s1 contains 2. This is important for when the division is going to occur.

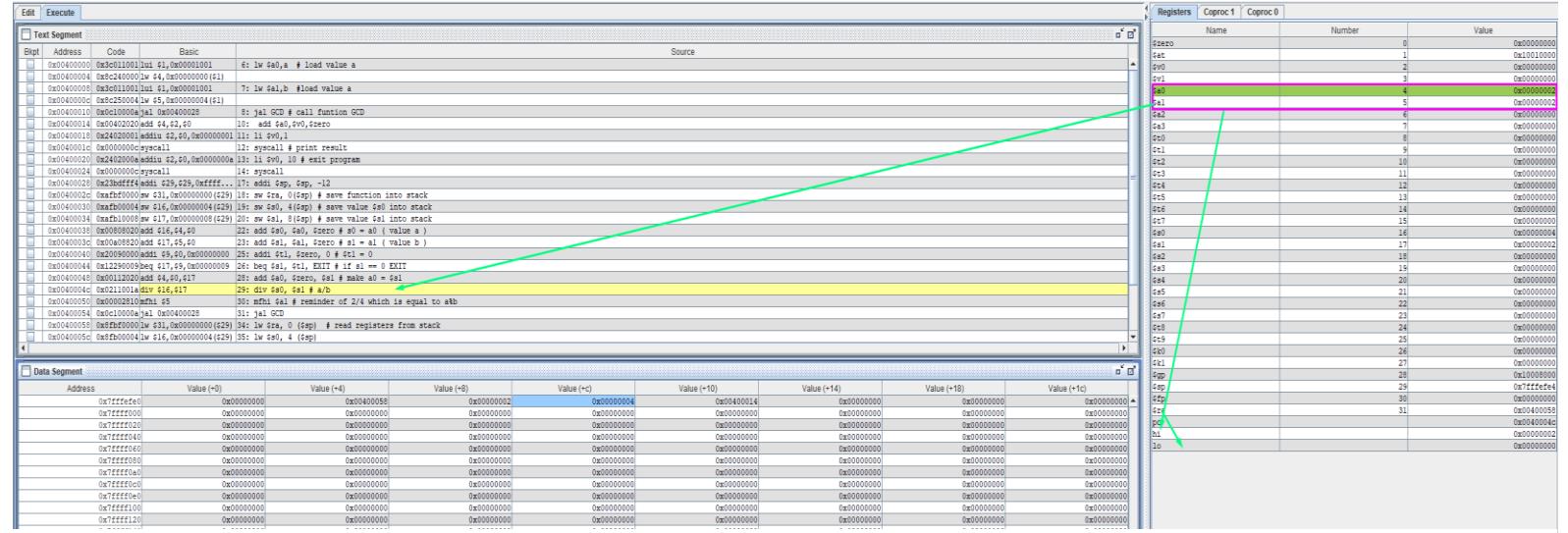


Figure 27: Division between 0 and 2

Registers Coproc 1 Coproc 0

Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x10010000
\$v0	2	0x00000000
\$v1	3	0x00000000
\$a0	4	0x00000002
\$a1	5	0x00000002
\$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
\$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	0xffffffff
\$t22	30	0x00000000
\$t23	31	0x00000000
pc	32	0x00400050
hi	33	0x00000000
lo	34	0x00000002

Figure 28: mod occurring with register LO

This figure above here shows us the mod operation occurring between 0 and 2. The register LO which is used for low-order 32 bits are stored is storing the quotient while, HI for high order 32 bits is where the remainder is being stored. Also, the registers \$a0 and \$a1 both contain the value of 2. Also \$t1 also contains the value of 0.

Registers Coproc 1 Coproc 0

Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x10010000
\$v0	2	0x00000000
\$v1	3	0x00000000
\$a0	4	0x00000002
\$a1	5	0x00000002
\$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
\$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	0xffffffff
\$t22	30	0x00000000
\$t23	31	0x00000000
pc	32	0x00400054
hi	33	0x00000000
lo	34	0x00000002

Figure 29: GCD function executed

This figure shows that the GCD function executed and that 2 and 0 have been stored onto the stack.

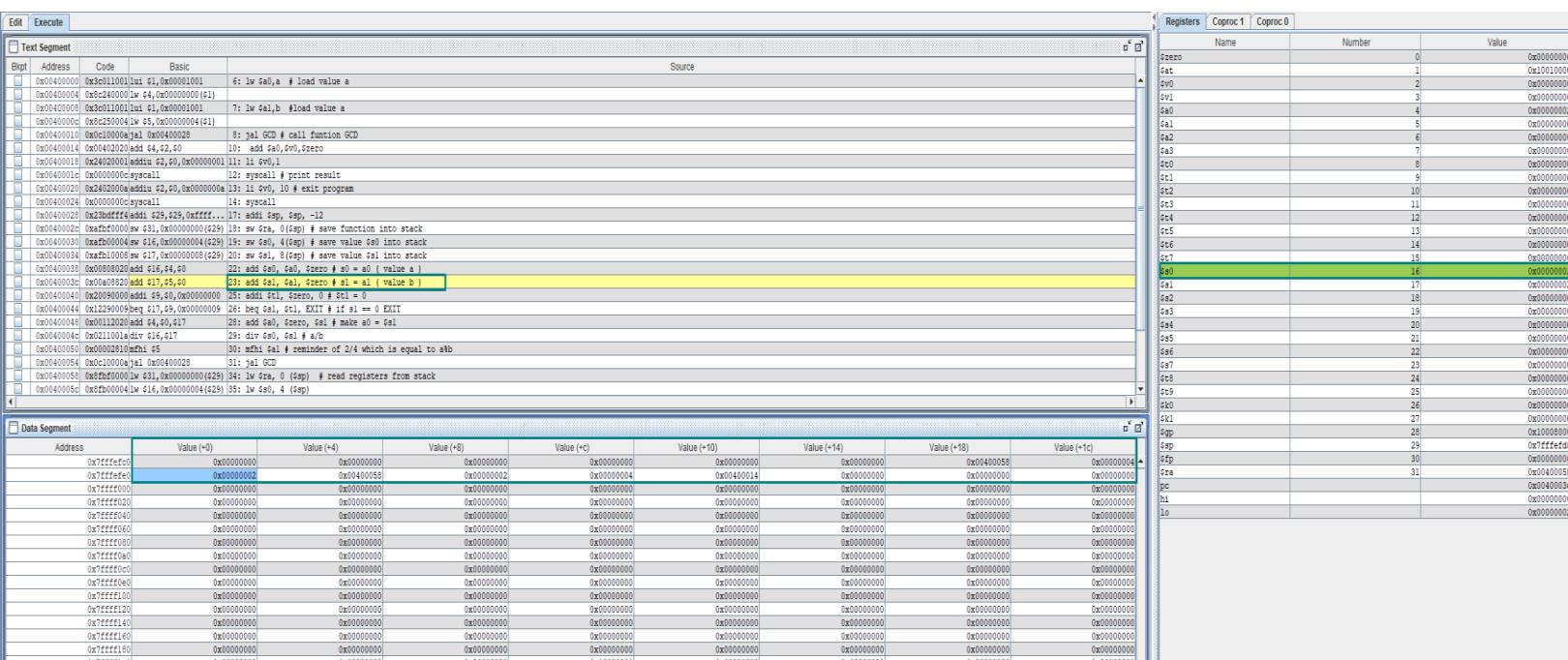


Figure 30: stack frame

This figure above here shows us the complete stack frame and all of the values that have been placed on the stack with their offsets.

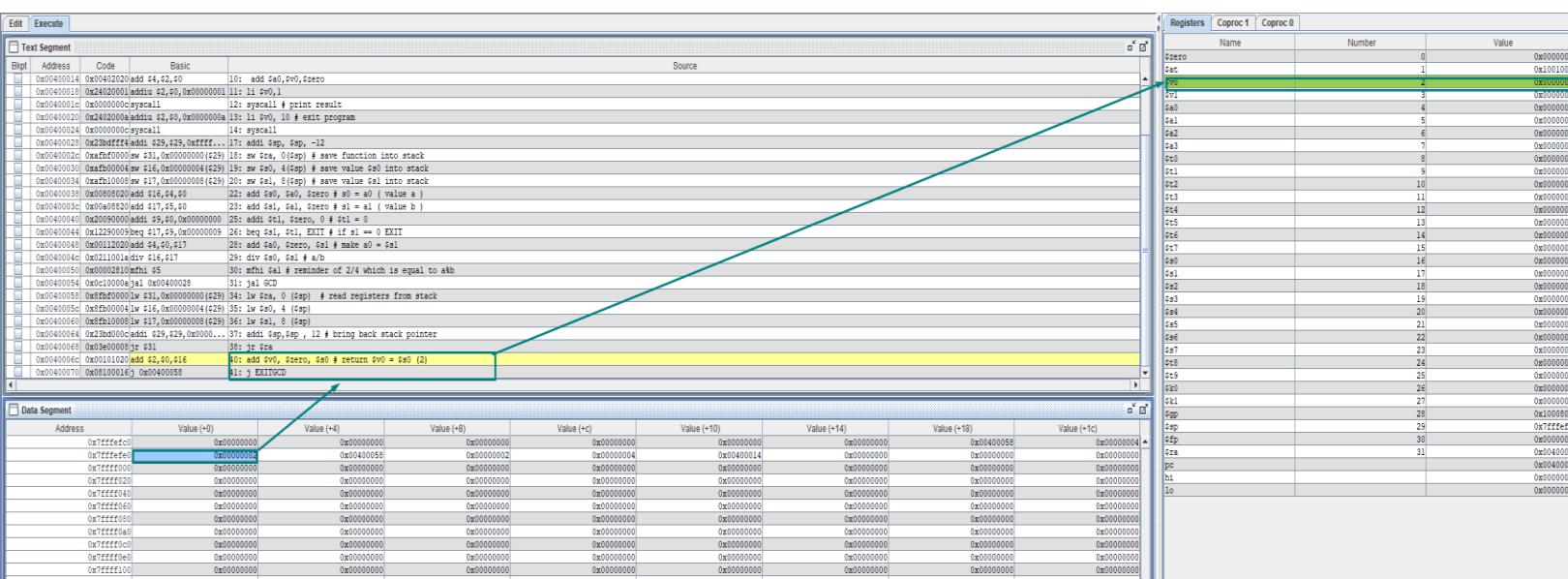


Figure 31: Clearing the registers

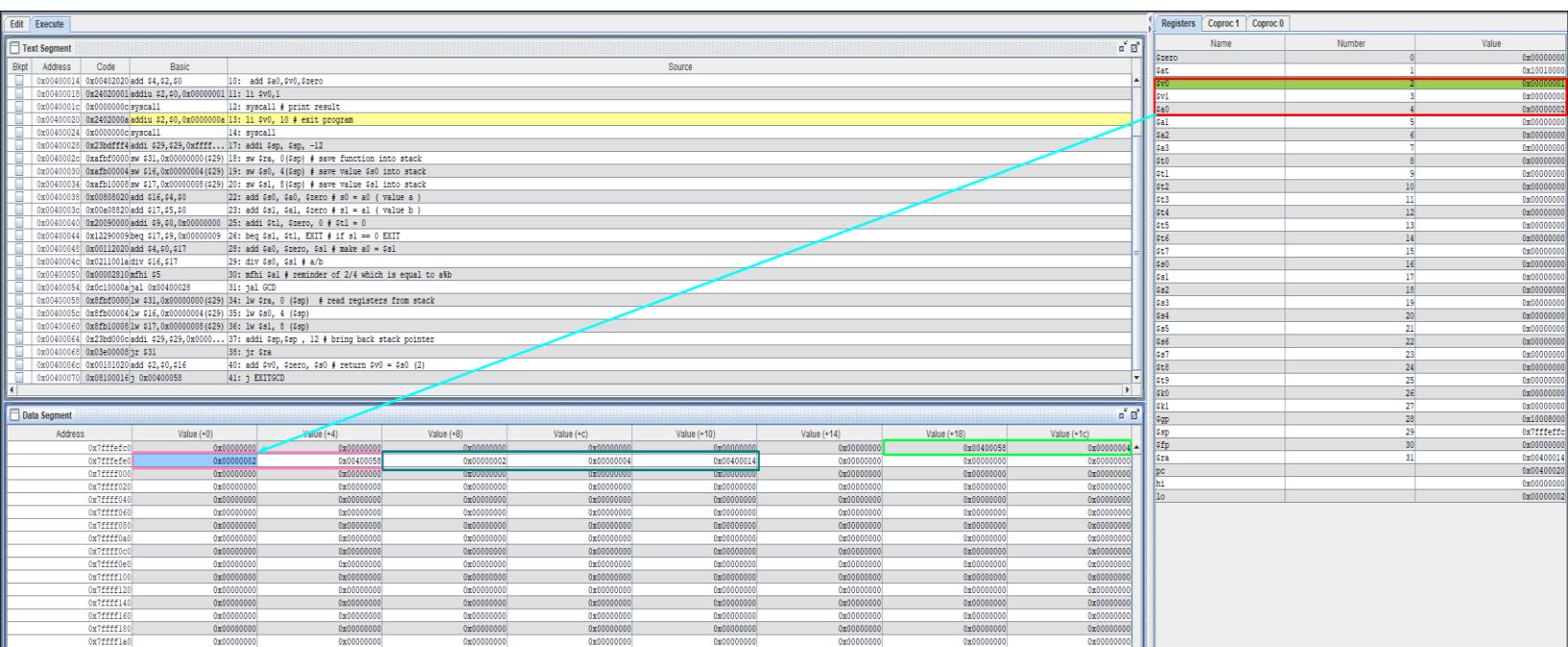


Figure 32: End of the program

The program has fully executed and now has returned the GCD of original values of 2 and 4. Also along with this is that the stack now contains all the values that have been pushed to it. We see that the program has also cleared all the registers to 0 and only contains \$a0 with 2. The program now will exit with a syscall to finish.

## Linux 64-Bit with GCC and GDB

```
c Adil_TAKEHOMETEST2.c
Adil_TAKE_HOME_TEST_2 > C Adil_TAKEHOMETEST2.c > ...
1 // Adil_TAKE_HOME_TEST_2.c : This file contains the 'main' function. Program execution begins at address 0x401010, which is located at the top of the main routine
2 // Adil Ahmad
3
4 #include <stdio.h>
5
6 int gcd(int a, int b)
7 {
8     if (a == 0)
9         return b;
10    return gcd(b % a, a);
11 }
12
13 int main()
14 {
15     int a = 2, b = 4;
16     printf("GCD(%d,%d)=%dn",a,b,gcd(a, b));
17     return 0;
18 }
19
```

Figure 33: Adil\_TAKEHOMETEST2.c

This here is the source code for the GCD program. The function takes in two integer values and performs a recursive GCD algorithm between them. The function also returns an integer value. Within main, I created two variables, a and b with them being 2 and 4 respectively. Main them computes the GCD and prints out the value.

```
Adil_TAKEHOMETEST2.c
12
13     int main()
14 {
15         int a = 2, b = 4;
16         printf("GCD(%d,%d)=%dn",a,b,gcd(a, b));
17         return 0;
18     }
19
```

0x55555555517b <main> endbr64
0x55555555517f <main+4> push %rbp
0x555555555180 <main+5> mov %rsp,%rbp
0x555555555183 <main+8> sub \$0x10,%rsp
B+>0x555555555187 <main+12> movl \$0x2,-0x8(%rbp)
0x55555555518e <main+19> movl \$0x4,-0x4(%rbp)
0x555555555195 <main+26> mov -0x4(%rbp),%edx
0x555555555198 <main+29> mov -0x8(%rbp),%eax
0x55555555519b <main+32> mov %edx,%esi
0x55555555519d <main+34> mov %eax,%edi
0x55555555519f <main+36> call 0x555555555149 <gcd>
0x5555555551a4 <main+41> mov %eax,%ecx
0x5555555551a6 <main+43> mov -0x4(%rbp),%edx

native process 3013 In: main
(gdb) layout split
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/adil/Documents/CSC342/Adil TAKE HOME TEST 2/Adil

Breakpoint 1, main () at Adil\_TAKEHOMETEST2.c:15
(gdb) ■

Figure 34: Breaking main

This here is the start of the program. First, we place a break point in the beginning of main. This is shown with the address of the movl instruction. This instruction will move both the initial variables, a and b into the stack frame. This is done by using an offset. For the variable a, which holds 2, its offset from the \$rbp, which is the register base pointer, by 8 bytes. This is the same as the visual studio example, showing that the stack and memory offsets correlate to each other. For the variable of b, which holds the value of 4 it is offset into memory by 4 bytes. This is known with the hexadecimal value of -0x04 and -0x08 respectively for the variables. The negative sign means that it is offset by a negative.

```
Breakpoint 1, main () at Adil TAKEHOMETEST2.c:15
(gdb) display $edx
1: $edx = -8568
(gdb) display $eax
2: $eax = 1431654779
```

Figure 35: This shows the initial value of both edx and eax

The screenshot shows a debugger window with two main panes. The top pane displays assembly code for the GCD function, with addresses ranging from 0x55555555149 to 0x5555555516a. The bottom pane shows the state of various registers. A red box highlights the \$rbx register, which contains the value 0x555555551d0. Another red box highlights the \$rdx register, which contains the value 0x4. The assembly code includes instructions like push %rbp, mov %rsp,%rbp, sub \$0x10,%rsp, mov %edi,-0x4(%rbp), and cmpb \$0x0,-0x4(%rbp). The PC register is at 0x5555555515b.

Register	Value	Description
rax	0x2	
rcx	0x555555551d0	93824992235984
rsi	0x4	
rbp	0x7fffffffdd60	0x7fffffffdd60
r8	0x0	0
r10	0xfffffffffffffccb	-821
r12	0x55555555060	93824992235616
r14	0x0	0
rip	0x5555555515b	0x5555555515b <gcd+18>
cs	0x33	51
ds	0x0	0
rbx	0x555555551d0	93824992235984
rdx	0x4	4
rdi	0x2	2
rsp	0x7fffffffdd50	0x7fffffffdd50
r9	0x7ffff7fdab50	140737353984848
r11	0x206	518
r13	0x0	0
r15	0x0	0
eflags	0x206	[ PF IF ]
ss	0x2b	43
es	0x0	0

```

0x55555555149 <gcd>    endbr64
0x5555555514d <gcd+4>  push   %rbp
0x5555555514e <gcd+5>  mov    %rsp,%rbp
0x55555555151 <gcd+8>  sub    $0x10,%rsp
0x55555555155 <gcd+12> mov    %edi,-0x4(%rbp)
0x55555555158 <gcd+15> mov    %esi,-0x8(%rbp)
>0x5555555515b <gcd+18> cmpl   $0x0,-0x4(%rbp)
0x5555555515f <gcd+22> jne    0x55555555166 <gcd+29>
0x55555555161 <gcd+24>  mov    -0x8(%rbp),%eax
0x55555555164 <gcd+27>  jmp    0x55555555179 <gcd+48>
0x55555555166 <gcd+29>  mov    -0x8(%rbp),%eax
0x55555555169 <gcd+32>  cltd
0x5555555516a <gcd+33>  idivl  -0x4(%rbp)

native process 3327 In: gcd
rip      0x55555555195  0x55555555195 <main+26>
eflags   0x202          [ IF ]
cs       0x33           51
ss       0x2b           43
ds       0x0            0
es       0x0            0
fs       0x0            0
gs       0x0            0
(gdb) layout regs
(gdb) step
gcd (a=2, b=4) at Adil TAKEHOMETEST2.c:8
1: $edx = 4
2: $eax = 2
(gdb) 
```

Figure 36: Windows showing 2 and 4

This window here shows the main initial values of both 2 and 4 loaded into the stack and loaded into the registers \$edx and \$eax. In the top window we can see all of the registers of the program. For the register \$rax, which is used to store a function's return value. For this we know that since the GCD function will return 2 as the value from it because of how the GCD between 2 and 4 is 2. Finally, we see that the \$rdx register, which holds the value of 4 and \$rdi which holds the value of 2.

```

Adil TAKEHOMETEST2.c-----
7             {
8                 if (a == 0)
9                     return b;
10                return gcd(b % a, a);
11            }
12
13            int main()
14            {
15                 int a = 2, b = 4;
16                 printf("GCD(%d,%d)=%dn",a,b,gcd(a, b));
17                 return 0;
}

0x555555555149 <gcd>    endbr64
0x55555555514d <gcd+4>  push   %rbp
0x55555555514e <gcd+5>  mov    %rsp,%rbp
0x555555555151 <gcd+8>  sub    $0x10,%rsp
0x555555555155 <gcd+12> mov    %edi,-0x4(%rbp)
0x555555555158 <gcd+15> mov    %esi,-0x8(%rbp)
0x55555555515b <gcd+18> cmpl   $0x0,-0x4(%rbp)
0x55555555515f <gcd+22> jne    0x555555555166 <gcd+29>
0x555555555161 <gcd+24> mov    -0x8(%rbp),%eax
0x555555555164 <gcd+27> jmp    0x555555555179 <gcd+48>
>0x555555555166 <gcd+29> mov    -0x8(%rbp),%eax
0x555555555169 <gcd+32> cltd
0x55555555516a <gcd+33> idivl  -0x4(%rbp)

native process 3327 In: gcd
ds          0x0          0
es          0x0          0
fs          0x0          0
gs          0x0          0
(gdb) layout regs
(gdb) step
gcd (a=2, b=4) at Adil TAKEHOMETEST2.c:8
1: $edx = 4
2: $eax = 2
(gdb) step
1: $edx = 4
2: $eax = 2

```

Figure 38: Returning value

```

(gdb) info f
Stack level 0, frame at 0x7fffffffdd50:
rip = 0x55555555515b in gcd (Adil TAKEHOMETEST2.c:8); saved rip = 0x555555555179
called by frame at 0x7fffffffdd70
source language c.
Arglist at 0x7fffffffdd40, args: a=0, b=2
Locals at 0x7fffffffdd40, Previous frame's sp is 0x7fffffffdd50
Saved registers:
    rbp at 0x7fffffffdd40, rip at 0x7fffffffdd48

```

Figure 37: Stack frame

```

Adil TAKEHOMETEST2.c
7
8     {
9         if (a == 0)
10            return b;
11        return gcd(b % a, a);
12    }
13
14    int main()
15    {
16        int a = 2, b = 4;
17        printf("GCD(%d,%d)=%dn", a, b, gcd(a, b));
18        return 0;
19    }

0x5555555555149 <gcd>    endbr64
0x555555555514d <gcd+4>  push    %rbp
0x555555555514e <gcd+5>  mov     %rsp,%rbp
0x5555555555151 <gcd+8>  sub    $0x10,%rsp
0x5555555555155 <gcd+12> mov     %edi,-0x4(%rbp)
0x5555555555158 <gcd+15> mov     %esi,-0x8(%rbp)
>0x555555555515b <gcd+18> cmpl   $0x0,-0x4(%rbp)
0x555555555515f <gcd+22> jne    0x5555555555166 <gcd+29>
0x5555555555161 <gcd+24>  mov     -0x8(%rbp),%eax
0x5555555555164 <gcd+27>  jmp    0x5555555555179 <gcd+48>
0x5555555555166 <gcd+29>  mov     -0x8(%rbp),%eax
0x5555555555169 <gcd+32>  cltd
0x555555555516a <gcd+33>  idivl  -0x4(%rbp)

native process 3327 In: gcd
(gdb) layout regs
(gdb) step
gcd (a=2, b=4) at Adil TAKEHOMETEST2.c:8
1: $edx = 4
2: $eax = 2
(gdb) step
1: $edx = 4
2: $eax = 2
(gdb) layout spli
(gdb) step
gcd (a=0, b=2) at Adil TAKEHOMETEST2.c:8
1: $edx = 0
2: $eax = 2

```

Figure 39: conditional statement

In this figure, we see that the values of the \$edx and \$eax registers have changed. This is due to GCD function being executed. What happened was that the mod of 4 and 2 results in 0. This is because doing mod only calculates the remainder of the operation. Since, 4 mod 2 results in 0, with 2 being the remainder, this means that 2 becomes the quotient, which is variable b and 0 becomes variable a. This is shown with the resulting \$edx register now containing the value of 0. While the register \$eax, now contains the value of 2. In the assembly break down we can see that these values are being offset into the stack again. This is a repeat of the function call before, since gcd will execute one more time in order to break out the function and enter the print statement.

```

Adil TAKEHOMETEST2.c
7          {
8              if (a == 0)
9                  return b;
10             return gcd(b % a, a);
11         }
12
13     int main()
14     {
15         int a = 2, b = 4;
16         printf("GCD(%d,%d)=%dn",a,b,gcd(a, b));
17         return 0;

```

```

0x555555555514e <gcd+5>    mov    %rsp,%rbp
0x5555555555151 <gcd+8>    sub    $0x10,%rsp
0x5555555555155 <gcd+12>   mov    %edi,-0x4(%rbp)
0x5555555555158 <gcd+15>   mov    %esi,-0x8(%rbp)
0x555555555515b <gcd+18>   cmpl   $0x0,-0x4(%rbp)
0x555555555515f <gcd+22>   jne    0x5555555555166 <gcd+29>
>0x5555555555161 <gcd+24>   mov    -0x8(%rbp),%eax
0x5555555555164 <gcd+27>   jmp    0x5555555555179 <gcd+48>
0x5555555555166 <gcd+29>   mov    -0x8(%rbp),%eax
0x5555555555169 <gcd+32>   cltd
0x555555555516a <gcd+33>   idivl  -0x4(%rbp)
0x555555555516d <gcd+36>   mov    -0x4(%rbp),%eax
0x5555555555170 <gcd+39>   mov    %eax,%esi

```

```

native process 3327 In: gcd
(gdb) info f
Stack level 0, frame at 0x7fffffffdd50:
rip = 0x555555555515b in gcd (Adil TAKEHOMETEST2.c:8); saved rip = 0x5555555555179
called by frame at 0x7fffffffdd70
source language c.
Arglist at 0x7fffffffdd40, args: a=0, b=2
Locals at 0x7fffffffdd40, Previous frame's sp is 0x7fffffffdd50
Saved registers:
    rbp at 0x7fffffffdd40, rip at 0x7fffffffdd48
(gdb) layout split
(gdb) step
1: $edx = 0
2: $eax = 2
(gdb) 

```

Figure 41: Return b going to be executed

Return b here will be executed, since we know a = 0 and the condition is now met.

```

0x555555555514e <gcd+5>    mov    %rsp,%rbp
0x5555555555151 <gcd+8>    sub    $0x10,%rsp
0x5555555555155 <gcd+12>   mov    %edi,-0x4(%rbp)
0x5555555555158 <gcd+15>   mov    %esi,-0x8(%rbp)
0x555555555515b <gcd+18>   cmpl   $0x0,-0x4(%rbp)
0x555555555515f <gcd+22>   jne    0x5555555555166 <gcd+29>
>0x5555555555161 <gcd+24>   mov    -0x8(%rbp),%eax
0x5555555555164 <gcd+27>   jmp    0x5555555555179 <gcd+48>
0x5555555555166 <gcd+29>   mov    -0x8(%rbp),%eax
0x5555555555169 <gcd+32>   cltd
0x555555555516a <gcd+33>   idivl  -0x4(%rbp)
0x555555555516d <gcd+36>   mov    -0x4(%rbp),%eax
0x5555555555170 <gcd+39>   mov    %eax,%esi

```

Figure 40: jmp function to be executed

```

Adil TAKEHOMETEST2.c
7         {
8             if (a == 0)
9                 return b;
10            return gcd(b % a, a);
11        }
12
13    int main()
14    {
15        int a = 2, b = 4;
16        printf("GCD(%d,%d)=%dn",a,b,gcd(a, b));
17        return 0;
}

0x55555555169 <gcd+32>      cltd
0x5555555516a <gcd+33>      idivl -0x4(%rbp),%eax
0x5555555516d <gcd+36>      mov  -0x4(%rbp),%eax
0x55555555170 <gcd+39>      mov  %eax,%esi
0x55555555172 <gcd+41>      mov  %edx,%edi
0x55555555174 <gcd+43>      call 0x55555555149 <gcd>
>0x55555555179 <gcd+48>      leave
0x5555555517a <gcd+49>      ret
0x5555555517b <main>      endbr64
0x5555555517f <main+4>      push %rbp
0x55555555180 <main+5>      mov  %rsp,%rbp
0x55555555183 <main+8>      sub  $0x10,%rsp
B+ 0x55555555187 <main+12>     movl $0x2,-0x8(%rbp)

native process 3327 In: gcd
called by frame at 0x7fffffffdd70
source language c.
Arlist at 0x7fffffffdd40, args: a=0, b=2
Locals at 0x7fffffffdd40, Previous frame's sp is 0x7fffffffdd50
Saved registers:
  rbp at 0x7fffffffdd40, rip at 0x7fffffffdd48
(gdb) layout split
(gdb) step
1: $edx = 0
2: $eax = 2
(gdb) step
1: $edx = 0
2: $eax = 2
(gdb) 
```

Figure 42: exiting the GCD function

This figure above shows that the condition has been met and that the return value of b has been given.

This is shown with the Register listing on the bottom. This is shown with the resulting \$edx register now containing the value of 0. While the register \$eax, now contains the value of 2. In the assembly window we can see that the next function to be called is the leave function. But, before that function we can mention the call function that was skipped. The call function at memory location 0x55555555174, is the recursion within this entire program. We see that the function calls the memory location of 0x55555555149, which is the beginning of the GCD function. The leave function at memory location 0x55555555179 will make the program enter the main function.

```

Register group: general
rax      0x0          0
rcx      0x2          2
rsi      0x2          2
rbp      0x7fffff7e25680 0x7fffff7e25680
r8       0x0          0
r10     0xffffffffffffccb -821
r12     0x555555555060 93824992235616
r14     0x0          0
rip      0x7ffff7e25660 0x7ffff7e25660 < printf>
cs       0x33         51
ds       0x0          0
rbx      0x5555555551d0 93824992235984
rdx      0x4          4
rdi      0x555555556004 93824992239620
rsp      0x7fffff7fd68 0x7fffff7fd68
r9       0x7ffff7fdab50 140737353984848
r11     0x206        518
r13     0x0          0
r15     0x0          0
eflags   0x246        [ PF ZF IF ]
ss       0x2b         43
es       0x0          0

>0x7ffff7e25660 < printf>    endbr64
0x7ffff7e25664 < printf+4>    sub    $0xd8,%rsp
0x7ffff7e2566b < printf+11>   mov    %rdi,%r10
0x7ffff7e2566e < printf+14>   mov    %rsi,0x28(%rsp)
0x7ffff7e25673 < printf+19>   mov    %rdx,0x30(%rsp)
0x7ffff7e25678 < printf+24>   mov    %rcx,0x38(%rsp)
0x7ffff7e2567d < printf+29>   mov    %r8,0x40(%rsp)
0x7ffff7e25682 < printf+34>   mov    %r9,0x48(%rsp)
0x7ffff7e25687 < printf+39>   test   %al,%al
0x7ffff7e25689 < printf+41>   je     0x7ffff7e256c2 < printf+98>
0x7ffff7e2568b < printf+43>   movaps %xmm0,0x50(%rsp)
0x7ffff7e25690 < printf+48>   movaps %xmm1,0x60(%rsp)
0x7ffff7e25695 < printf+53>   movaps %xmm2,0x70(%rsp)

native process 3327 In: _printf
rbp at 0x7fffffffdd40, rip at 0x7fffffffdd48
(gdb) layout split
(gdb) step
1: $edx = 0
2: $eax = 2
(gdb) step
1: $edx = 0
2: $eax = 2
(gdb) layout regs
(gdb) step
printf (format=0x555555556004 "GCD(%d,%d)=%dn") at printf.c:28
1: $edx = 4
2: $eax = 0
(gdb) 
```

Here we see that the print statement has been executed and that the statement 2 is now being printed to the user. This is shown with the \$rcx and \$rsi registers. These registers are both holding the value of 2 which shows that 2 is now at the top of the stack and that the next value 0 stored in register \$rax will be used to reset the stack frame after the program completely ends.

[ Register Values Unavailable ]

```
>0x7ffff7e256f1 < printf+145>      mov    0x180828(%rip),%rax      # 0x7ffff7fa5f20
0x7ffff7e256f8 < printf+152>      movl   $0x8,(%rsp)
0x7ffff7e256ff < printf+159>      mov    (%rax),%rdi
0x7ffff7e25702 < printf+162>      movl   $0x30,0x4(%rsp)
0x7ffff7e2570a < printf+170>      call   0x7ffff7e3a2f0 < vfprintf internal>
0x7ffff7e2570f < printf+175>      mov    0x18(%rsp),%rcx
0x7ffff7e25714 < printf+180>      sub    %fs:0x28,%rcx
0x7ffff7e2571d < printf+189>      jne    0x7ffff7e25727 < printf+199>
0x7ffff7e2571f < printf+191>      add    $0xd8,%rsp
0x7ffff7e25726 < printf+198>      ret
0x7ffff7e25727 < printf+199>      call   0x7ffff7eee3b0 < stack chk fail>
0x7ffff7e2572c
0x7ffff7e25730 < GI  snprintf>    nopl   0x0(%rax)
0x7ffff7e25734 < GI  snprintf+4>   endbr64
                                         sub    $0xd8,%rsp
```

L33 PC: 0:

```
native process 4050 In:  printf
(gdb) step
printf (format=0x555555556004 "GCD(%d,%d)=%dn") at printf.c:28
1: $edx = 4
2: $eax = 0
(gdb) layout regs
(gdb) step
1: $edx = 4
2: $eax = 0
(gdb) step
1: $edx = -9072
2: $eax = -9040
(gdb) layout next
(gdb) layout split
(gdb) layout regs
(gdb) █
```

Figure 43: Print function

```

Register group: general
rax      0x0          0
rbx      0x7fffff7fb3580 140737353823616
rcx      0x4          4
rdx      0x1          1
rsi      0x25         37
rdi      0x7fffff7fa9670 140737353782896
rbp      0x7fffff7fa76c0 0x7fffff7fa76c0 < IO 2 1 stdout >
rsp      0x7fffffff7d710 0x7fffffff7d710
r8       0x0          0
r9       0x7fffff7fdab50 140737353984848
r10     0x555555556004 93824992239620
r11     0x206         518
r12     0x5555555555060 93824992235616

>0x7fffff7e3a3b9 < vfprintf_internal+201>    mov  0xd8(%rbp),%r12
0x7fffff7e3a3c0 < vfprintf internal+208>    lea   0x16e241(%rip),%rax      # 0x7fffff7fa8608 < elf set libc atexit
0x7fffff7e3a3c7 < vfprintf internal+215>    mov  0x60(%rsp),%rbx
0x7fffff7e3a3cc < vfprintf internal+220>    lea   0x16d4cd(%rip),%rdi      # 0x7fffff7fa78a0 < IO helper jumps>
0x7fffff7e3a3d3 < vfprintf internal+227>    sub  0x16a35e(%rip),%rax      # 0x7fffff7fa4738
0x7fffff7e3a3da < vfprintf internal+234>    sub  0x8(%rsp),%rbx
0x7fffff7e3a3df < vfprintf internal+239>    mov  %rax,0x30(%rsp)
0x7fffff7e3a3e4 < vfprintf internal+244>    mov  %rax,%rcx
0x7fffff7e3a3e7 < vfprintf internal+247>    mov  %r12,%rax
0x7fffff7e3a3ea < vfprintf internal+250>    sub  %rdi,%rax
0x7fffff7e3a3ed < vfprintf internal+253>    cmp  %rax,%rcx
0x7fffff7e3a3f0 < vfprintf internal+256>    jbe  0x7fffff7e3bc58 < vfprintf internal+6504>
0x7fffff7e3a3f6 < vfprintf internal+262>    mov  0x8(%rsp),%rsi
0x7fffff7e3a3fb < vfprintf internal+267>    mov  %rbx,%rdx

native process 4050 In: vfprintf internal
source language c.
Arglist at 0x7fffffff7d708, args: s=0x7fffff7fa76c0 < IO 2 1 stdout >, format=0x555555556004 "GCD(%d,%d)=%dn",
ap=ap@entry=0x7fffffff7dc90, mode flags=mode_flags@entry=0
Locals at 0x7fffffff7d708, Previous frame's sp is 0x7fffffff7dc90
Saved registers:
rbx at 0x7fffffff7dc58, rbp at 0x7fffffff7dc60, r12 at 0x7fffffff7dc68, r13 at 0x7fffffff7dc70, r14 at 0x7fffffff7dc78,
r15 at 0x7fffffff7dc80, rip at 0x7fffffff7dc88
(gdb) layout asm
(gdb) layout regs
(gdb) step
outstring func (done=<optimized out>, length=<optimized out>, string=<optimized out>, s=<optimized out>)
at vfprintf-internal.c:1404
1: $edx = 1
2: $eax = 0
L1404 PC: 0x7fffff7e3a3b9

```

Figure 45: Values being reset to 0

Here the program is resetting all of the values of the registers to 0.

eflags	0x246	[ PF ZF IF ]
rax	0x0	0
rbx	0x7fffff7fa84a0	140737353778336
rcx	0x1	1
rdx	0x0	0
rsi	0x555555556011	93824992239633
rdi	0x7fffff7fa9670	140737353782896
rbp	0x7fffff7fa76c0	0x7fffff7fa76c0 < IO 2 1 stdout >
rsp	0x7fffffff7d710	0x7fffffff7d710
r8	0x0	0
r9	0x7fffffff7dc47	140737488346183
r10	0x2	2
r11	0x0	0
r12	0x555555556012	93824992239634

Figure 44: Registers holding value

Register group: general		
rax	0x0	0
rbx	0x5555555551d0	93824992235984
rcx	0x2	2
rdx	0x4	4
rsi	0x2	2
rdi	0x555555556004	93824992239620
rbp	0x7fffffffdd80	0x7fffffffdd80
rsp	0x7fffffffddc90	0x7fffffffddc90
r8	0x0	0
r9	0x7ffff7fdab50	140737353984848
r10	0x555555556004	93824992239620
r11	0x206	518
r12	0x555555555060	93824992235616
0x7fff7e256a2 < printf+66>	movaps	%xmm4, 0x90(%rsp)
0x7fff7e256aa < printf+74>	movaps	%xmm5, 0xa0(%rsp)
0x7fff7e256b2 < printf+82>	movaps	%xmm6, 0xb0(%rsp)
0x7fff7e256ba < printf+90>	movaps	%xmm7, 0xc0(%rsp)
0x7fff7e256c2 < printf+98>	mov	%fs:0x28,%rax
0x7fff7e256cb < printf+107>	mov	%rax, 0x18(%rsp)
0x7fff7e256d0 < printf+112>	xor	%eax,%eax
>0x7fff7e256d2 < printf+114>	lea	0xe0(%rsp),%rax
0x7fff7e256da < printf+122>	xor	%ecx,%ecx
0x7fff7e256dc < printf+124>	mov	%rsp,%rdx
0x7fff7e256df < printf+127>	mov	%rax, 0x8(%rsp)
0x7fff7e256e4 < printf+132>	lea	0x20(%rsp),%rax
0x7fff7e256e9 < printf+137>	mov	%r10,%rsi
0x7fff7e256ec < printf+140>	mov	%rax, 0x10(%rsp)

Figure 46: Complete stack frame

This figure above shows us the complete stack frame after the program ends. Here this shows us the initial values of the variables. The first value of 2 is the value of a, along with variable b holding 4. This is the initial values for a and b. Then after the end we see 2 pushed onto the top of the stack and returns that value of 2.

## Conclusion

In conclusion, I learned a lot within this lab about the various environments, which are The Processors we are using are 32-bit MIPS processor using MARS Simulator, Intel x86 with a 32-bit compiler in Microsoft Visual Studio environment, and finally Intel x86 processor with 64-bit compiler using Linux GCC and GDB.

For the MIPS 32-bit MIPS Within MARS SIMULATOR the procedure calls are done using the jmp and jal instructions. When this instruction is performed, it will jump to the designated instruction address while storing the return address in a register. In MIPS there is two registers that are important for multiplication and division operations. The *HI* and *LO* registers are 32-bit registers which hold or accumulate the results of a multiplication or addition. You cannot operate on them directly. These registers are called automatically and are special instructions which are separate from the other ones.

For X86 Intel on Windows 32-Bit using Microsoft Visual Studio, function calls are done by pushing the argument on the stack. After the call, a new stack frame is initialized, and the return address is pushed onto the stack instead of into a register. For this program the offset from the base pointer which held the values of the variables was 8 bytes. At the end of each function call, the stack frame will be reverted to its previous state before the call and then it will return to the return address on top of the stack.

Within Linux 64-Bit with GCC and GDB, the Function calls are done using the jmp and callq instruction. This stores the return address in stack and then performs a jump to the called

Adil, Ahmad  
7/15/21  
Professor Gertner  
CSC 342/343

function address. For the program, stack frames are initialized during each function call. At the end of each function call, the stack frame is reverted to its previous state and the registers are stored to their original states.