# Take Home Test 2

**MdShahid Bin Emdad**

**April 25th, 2022**

**CSC 34200**

# TABLE OF CONTENTS

Objective: ..... 2
Description of Specifications, and Functionality: ..... 2
PART I: MARS Simulator: ..... 3
PART II: Visual Studio 2019 (Intel x86): ..... 19
PART III: x86_64 ISA, Linux 64-bit: ..... 23
Explanation: ..... 26
Time-Comparison: ..... 27
Conclusion: ..... 28

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

Shahid

start time and date: 04/23/22 10am
End time and date: 04/23/22 9:40pm

## Objective:

The objective of this test is to explore, understand and ability to demonstrate of recursive calls and stack frames in the three sets of architecture. MIPS instructions set architecture, Intel x86 ISA using Windows MS 32-bit compiler and debugger and a Intel X86_64 bit ISA processor running Linux, 64 bit gcc and gdb. We used recursive version of EUCLEDEAN algorithm for two integers a>0, b>0.

## Description of Specifications, and Functionality:

The digital system I used in this assignment is MARS to execute the assembly language and get the output, Microsoft Visual Studio 2019 to debug c programming languages program and get registers, disassemble and memory output. After that, I used Ubuntu Linux to run all the c files in that operating system and get output by using gdb. I used breakpoints while doing debugging.

***PART I: MARS Simulator:***

In figure 1, we can see the MIPS code on the MIPS simulator for recursive calls.

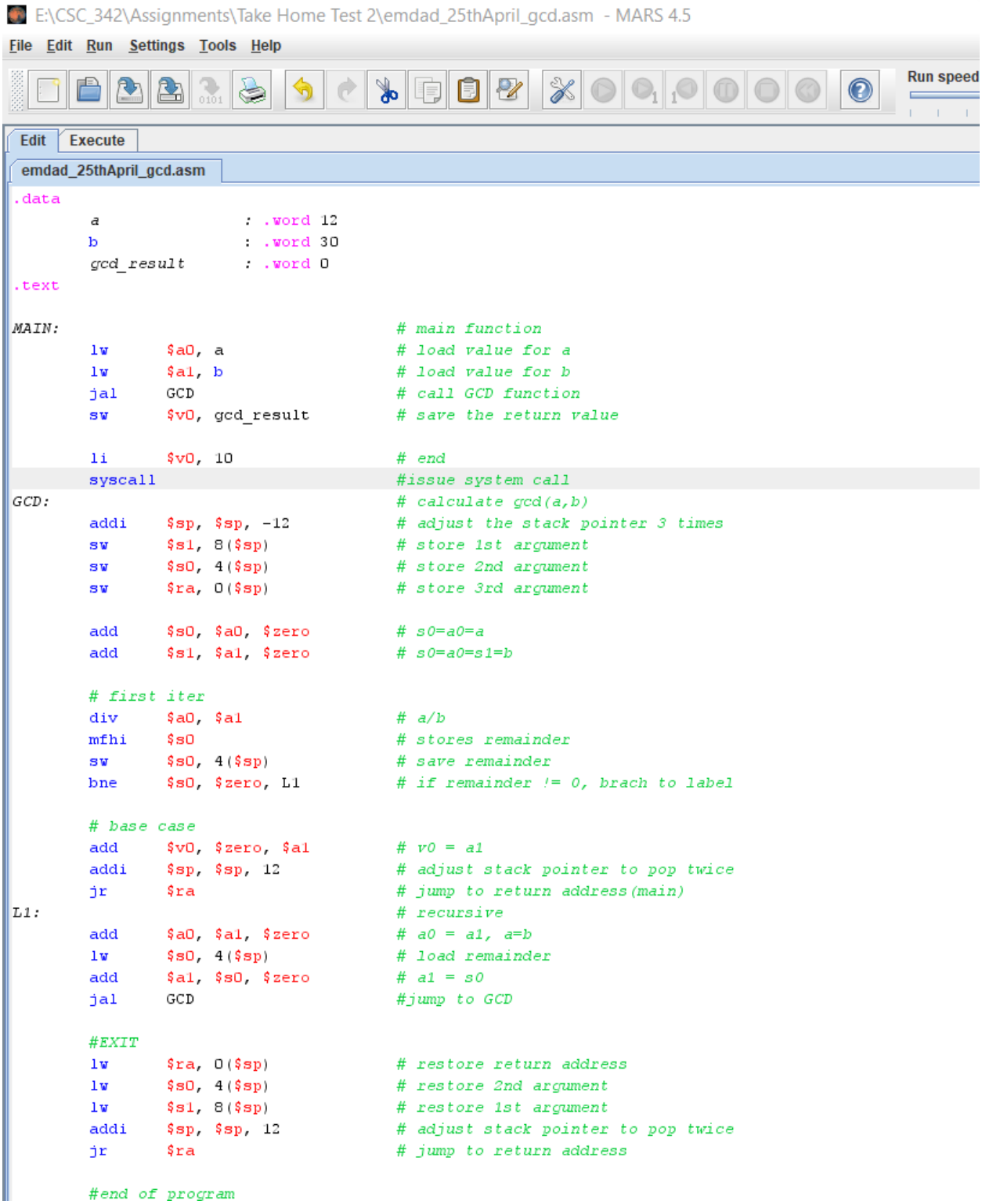


*Figure 1: MIPS code*

In figure 2, we can see that the operation completed successfully.

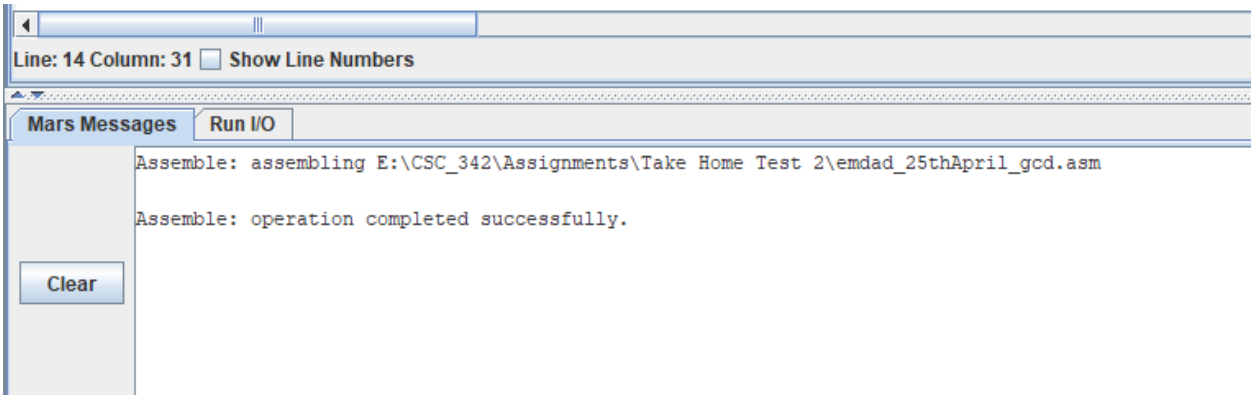


*Figure 2: operation completion report*

In figure 3, we can see the text segment, data segment and registers, labels for the MIPS code after execution.

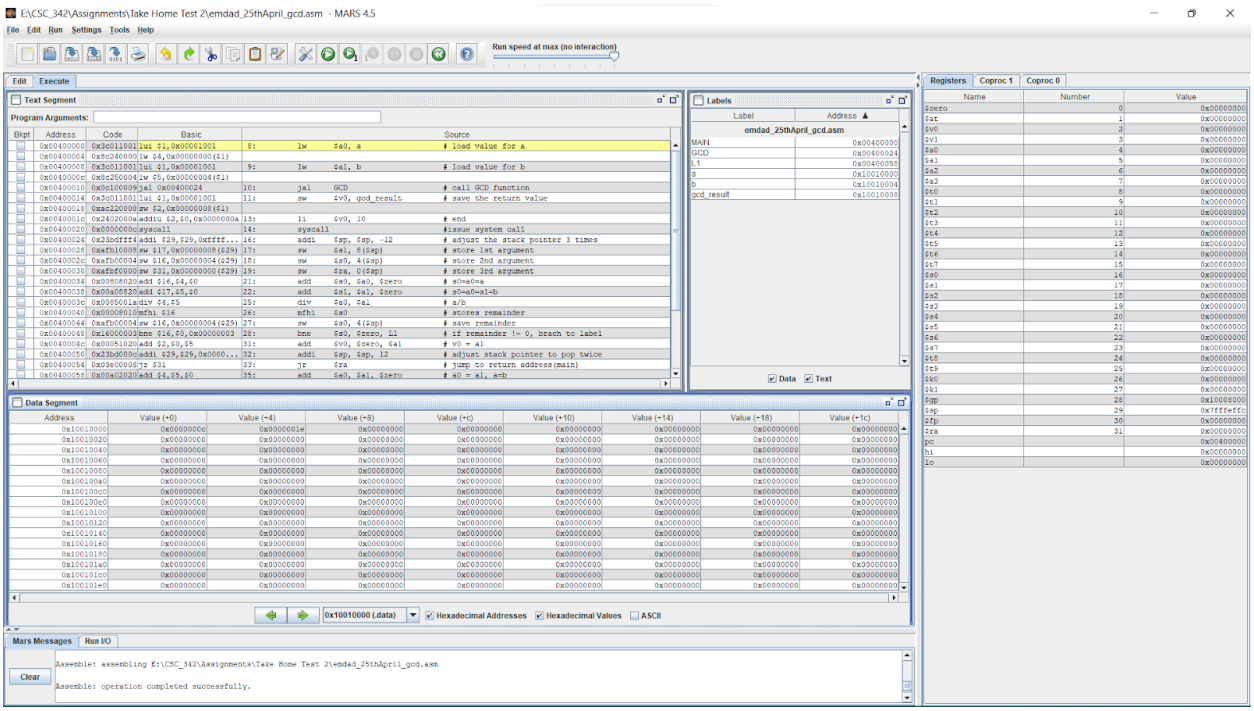


*Figure 3: text segment, data segment and registers window*

In figure 4, we can see the value for a stored in $a0.

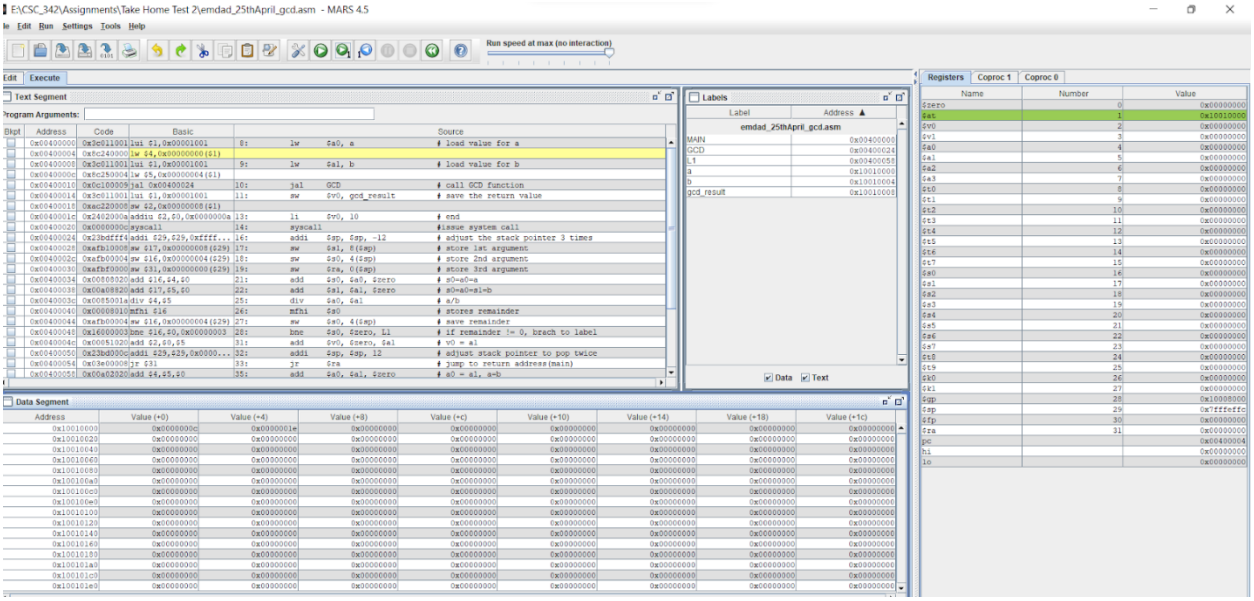

*Figure 4: value of a stored in $a0*

In figure 5, we can see the value for b stored in $a0.

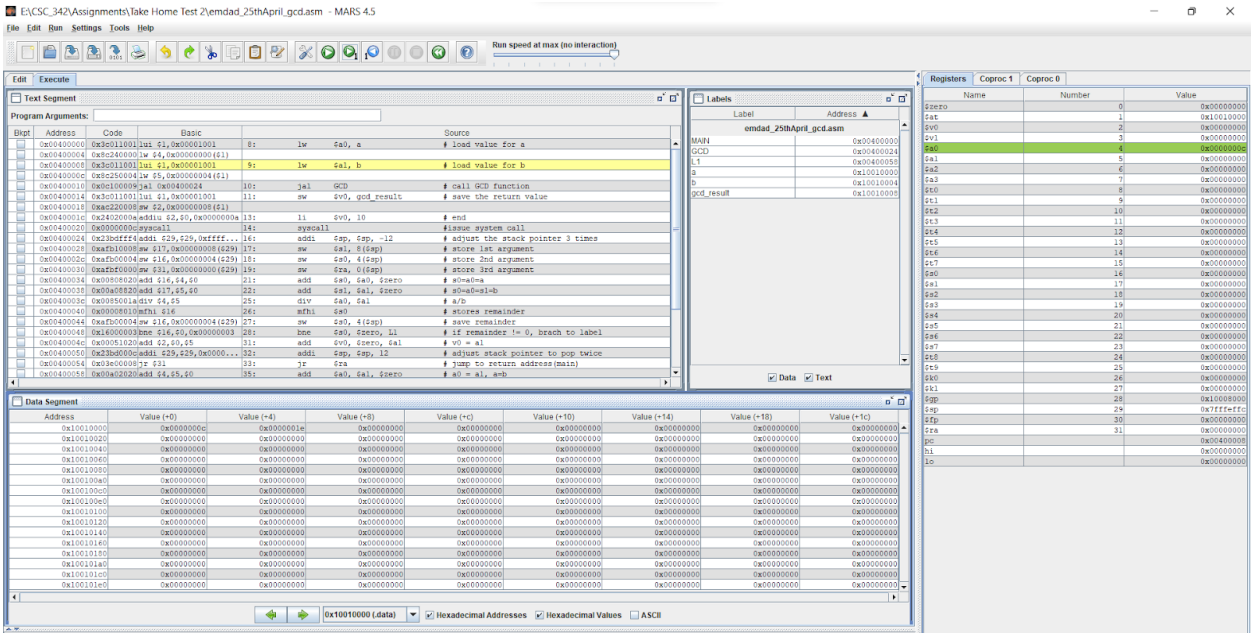

*Figure 5: value for b stored in $a0.*

In figure 6, we can see that it is back to main function after assigning the values into the variables.

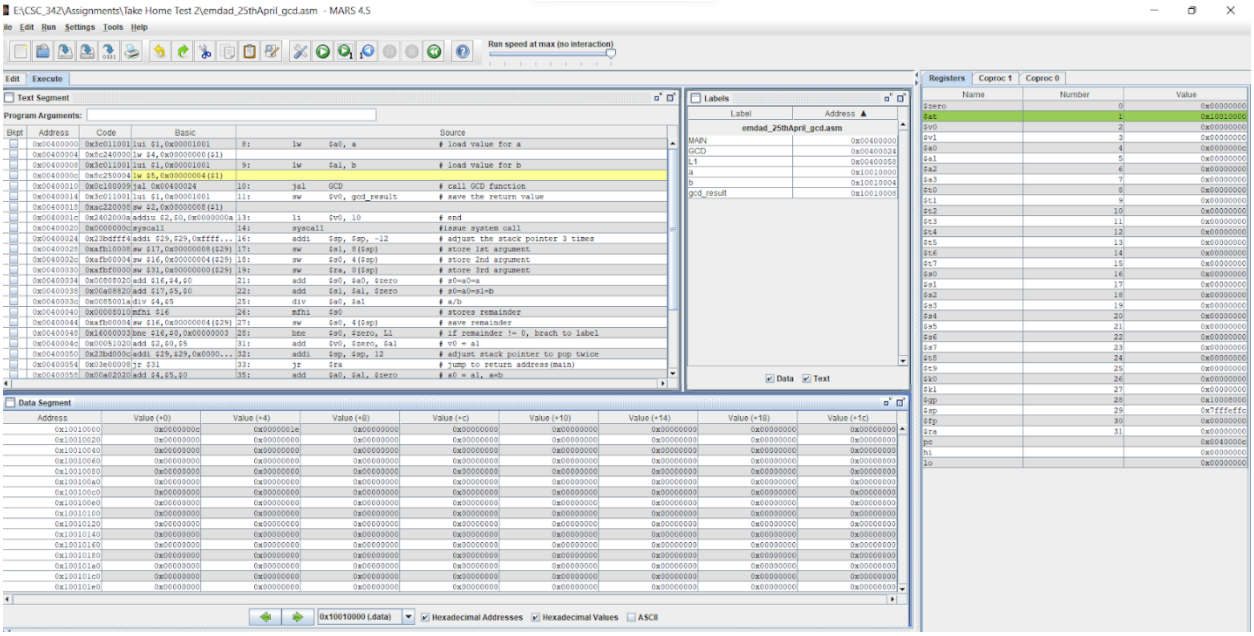

*Figure 6: back to main*

In figure 7, we can see that it is calling the GCD function to perform gcd(a,b).

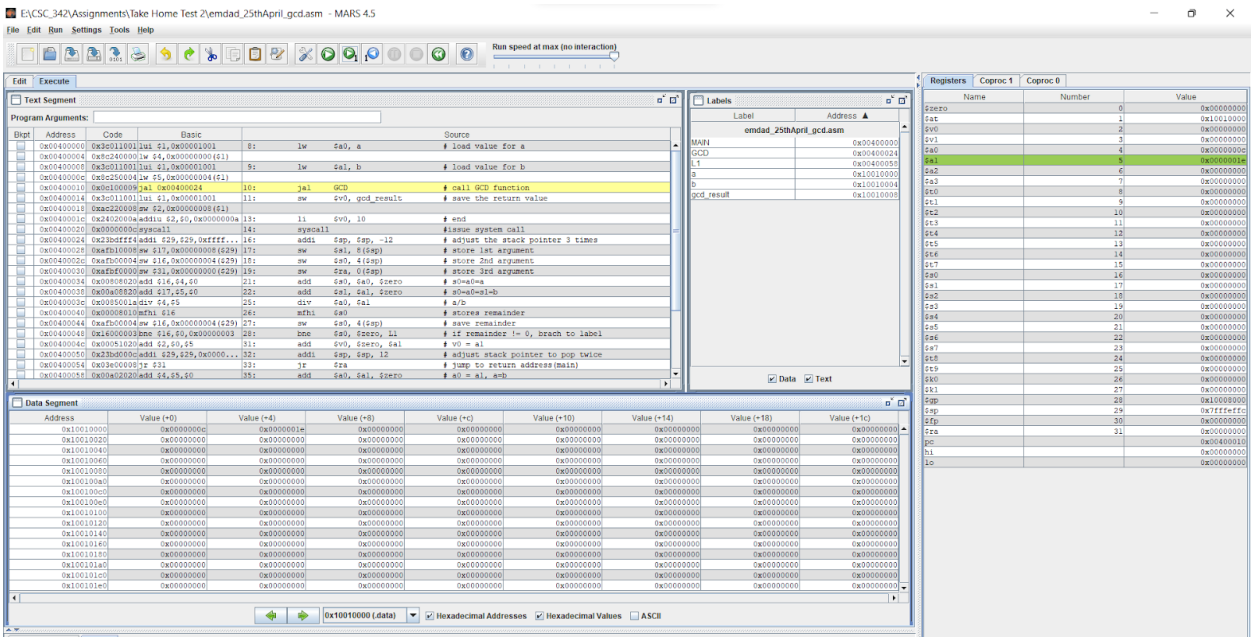

*Figure 7: calling the GCD function*

In figure 8, we can see that, the stack pointer moved and the address for $ra is 0x00400014.

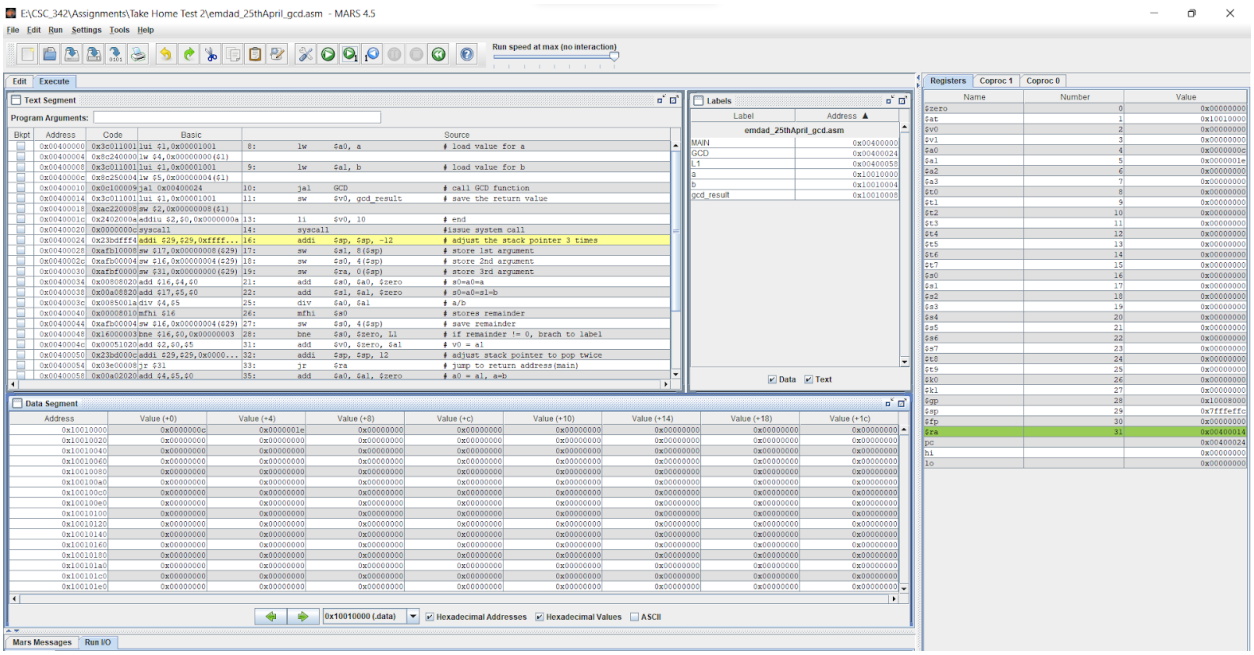

*Figure 8: stack pointer adjusted*

In figure 9, we can see that $s1 is storing the 1st argument. The address for $sp is 0x7fffeff0.

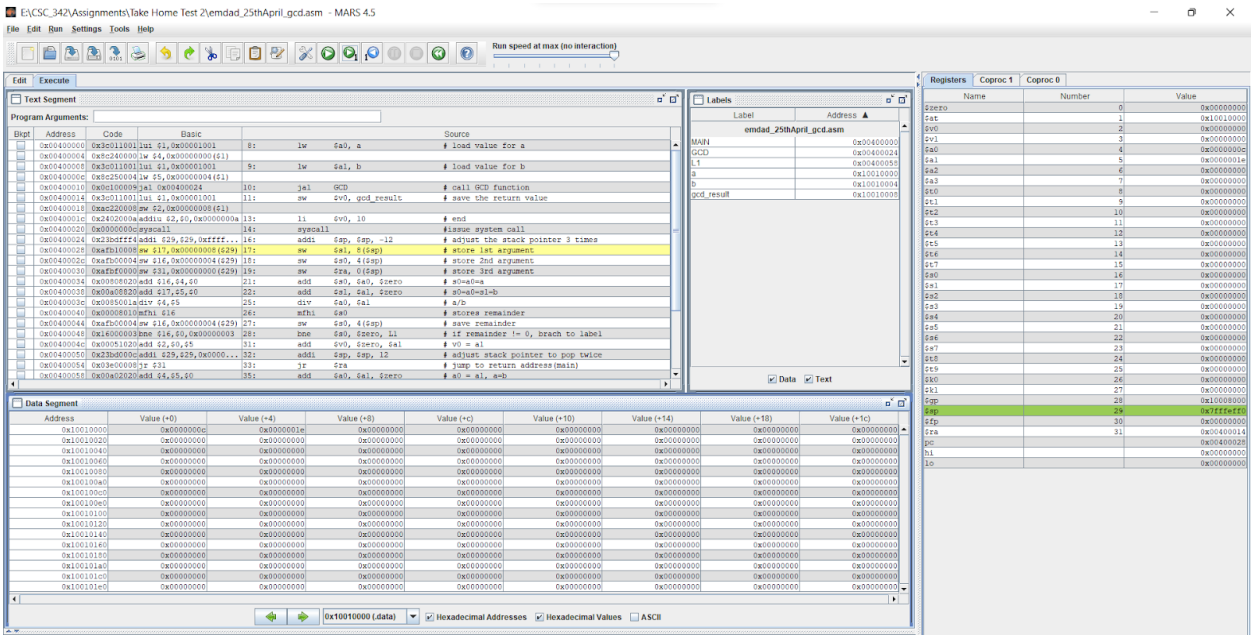

*Figure 9: stored 1st argument*

In figure 10, we can see that $s0 is storing the 2nd argument and $sp address is still same. The offset is +18 and the address value is 0.

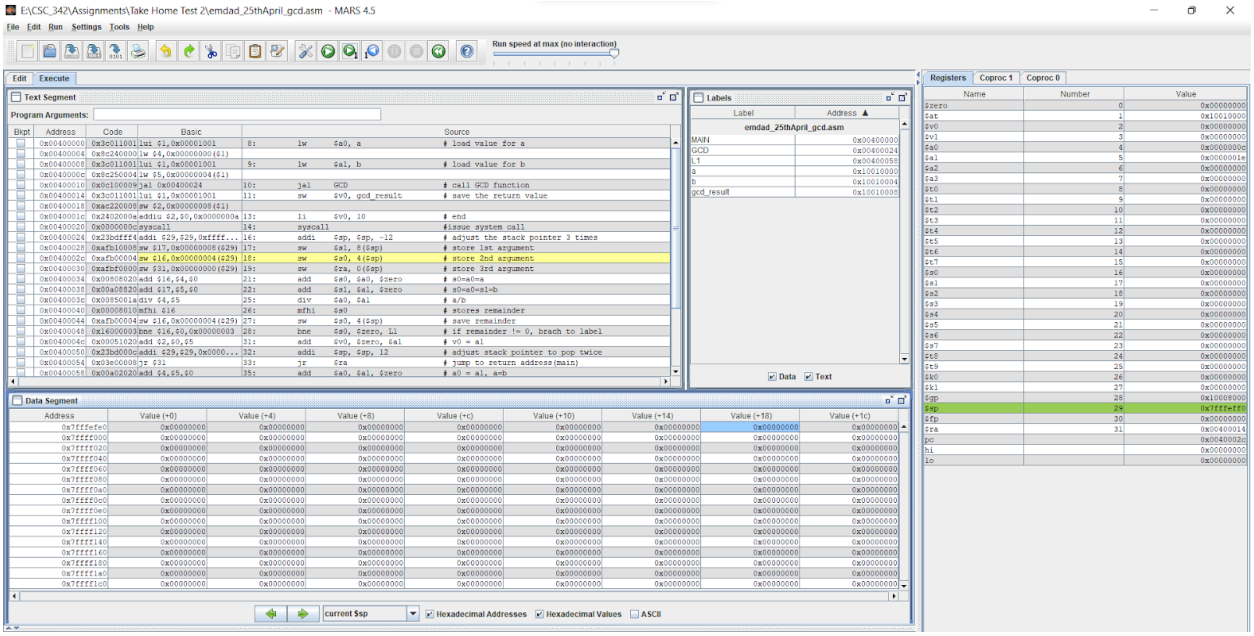

*Figure 10: 2nd argument storing*

In figure 11, we can see that $ra is storing the 3rd argument and $sp address is still same. The offset is +14 and the address, 0x7fffefe0 value is 0.

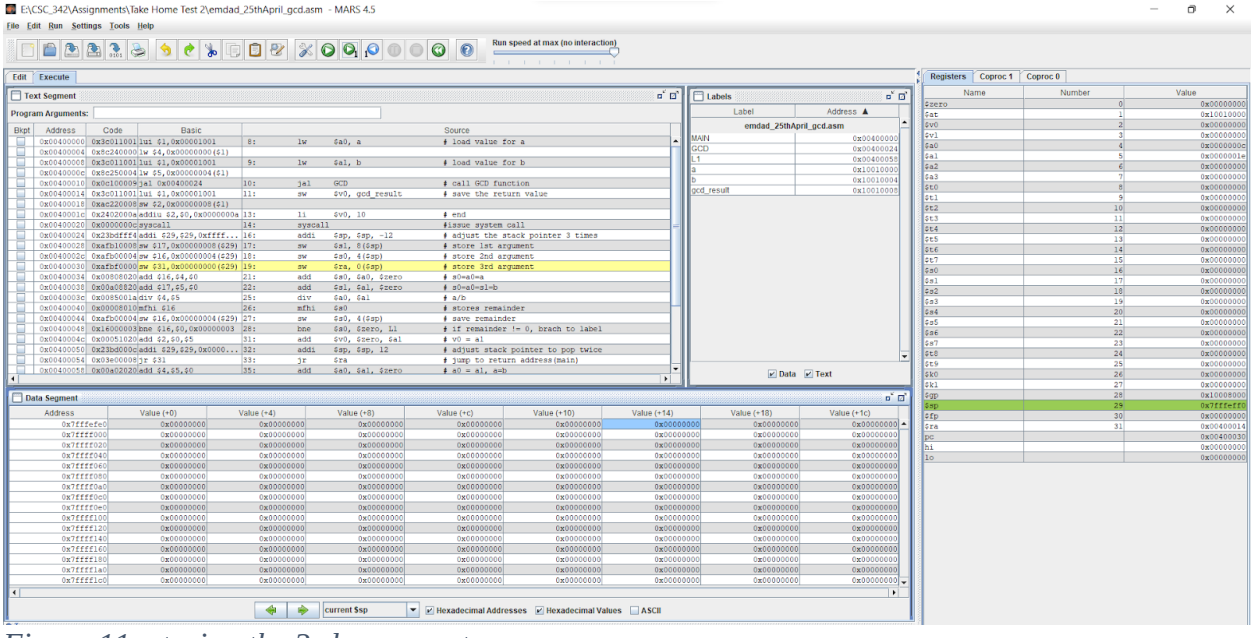

*Figure 11: storing the 3rd argument*

In figure 12, we can see that the replacing value happening on line 21, means s0=a0=a. The $sp address is still but the offset is +10 and value for the address, 0x7fffefe0 is 0x0040014.

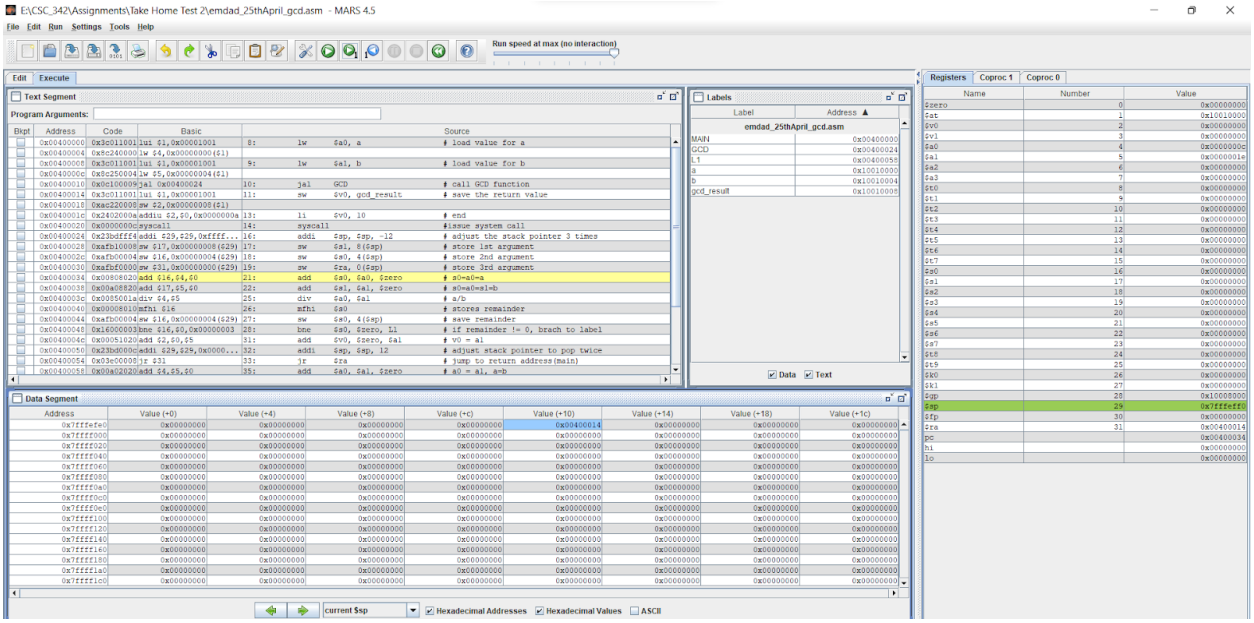

*Figure 12: add $s0, $a0*

In figure 13, we can see that the program moved to line 22 and now it is replacing value $s1 with $a1 which means s0=a0=s1=b. The register address for $s0 is 0x000000c.

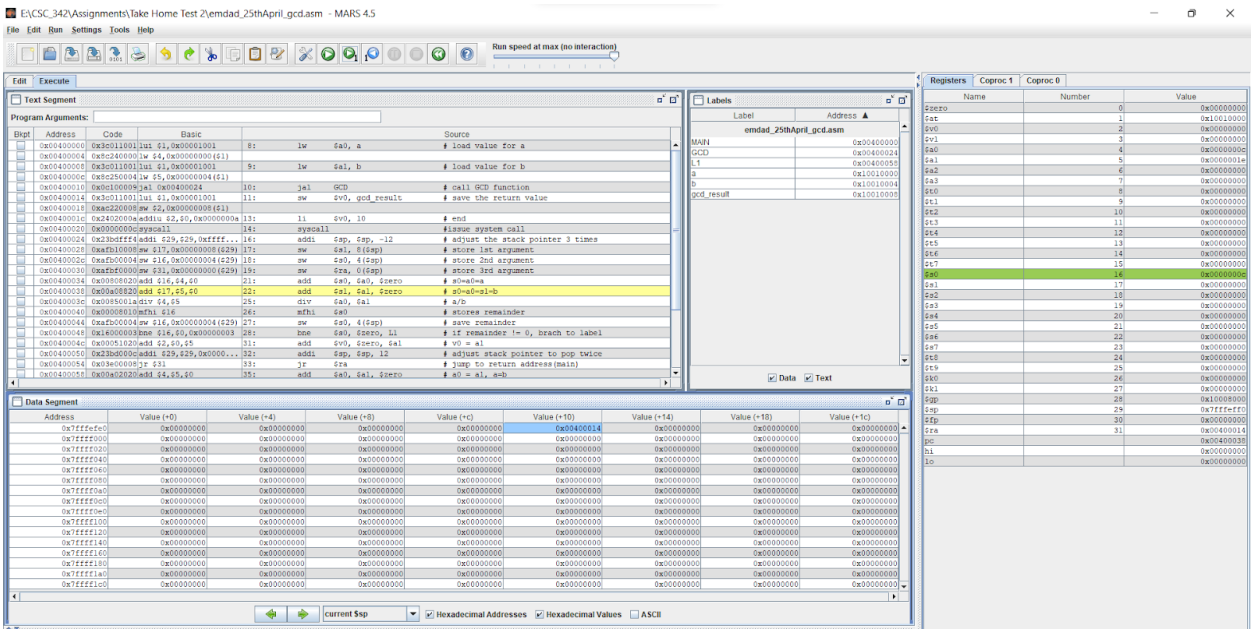

*Figure 13: add $s1, $a1*

In figure 14, we can see that it is doing division. We are diving a by b. and the stored address is $s1, 0x0000001e and the memory offset is +10 and value is 0x00400014.

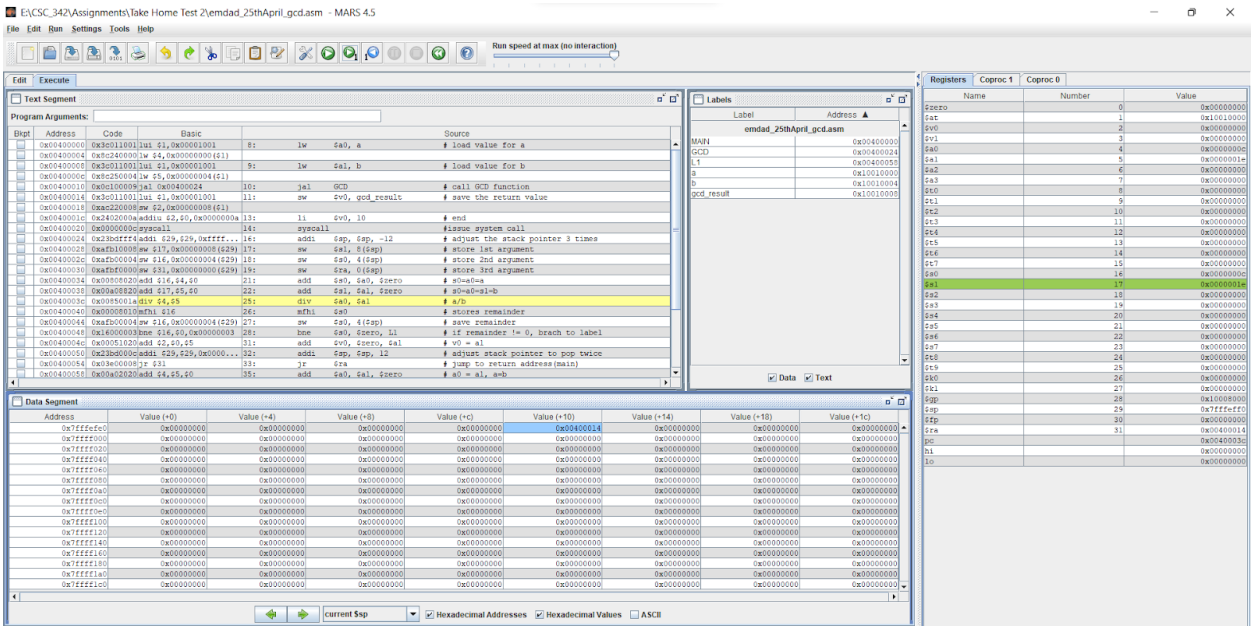

*Figure 14: division*

In figure 15, we can see that ***"mfhi"*** is storing the remainder. The register name is lo, and value is 0x00000000.

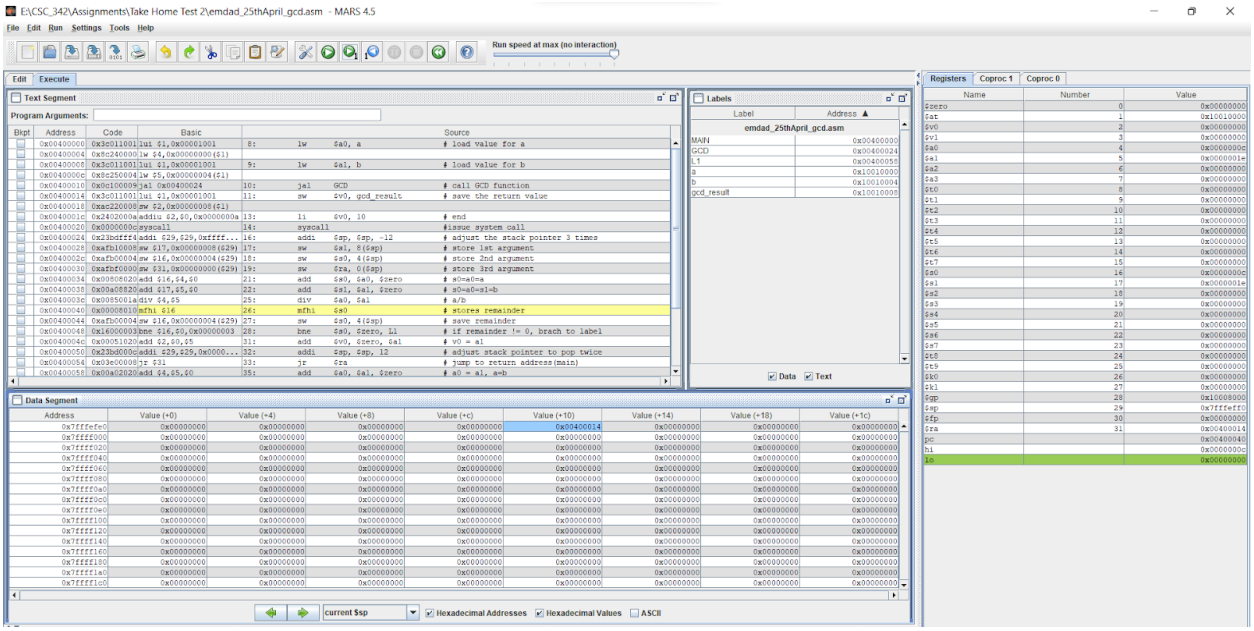

*Figure 15: storing remainder*

In figure 16, we can see that $s0 is saving the remainder and the register, $s0 and value is 0x000000c.

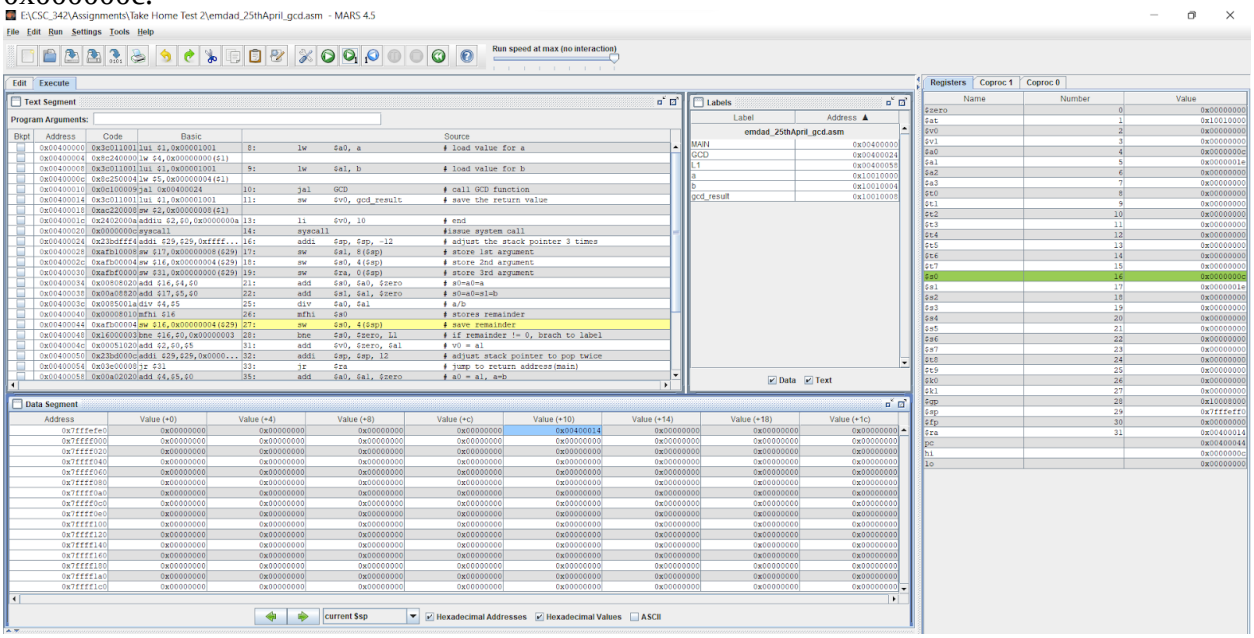

*Figure 16: save remainder into $s0*

In figure 17, we can see that the bne computation is happening and it is checking if the remainder != 0, if so, it will branch to label. The memory address offset is +14 and value stores is 0x00000c.

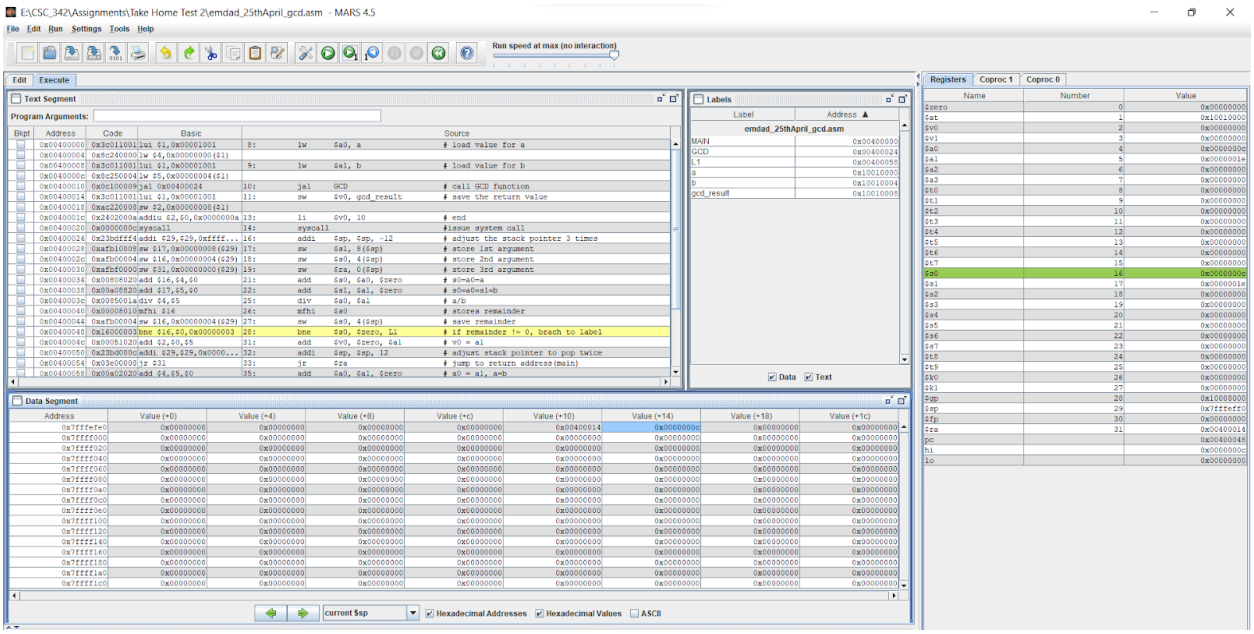

*Figure 17: BNE computation*

In figure 18, we can see that code went to line 35 to do a0 = a1 and a=b. The memory address value stays same.

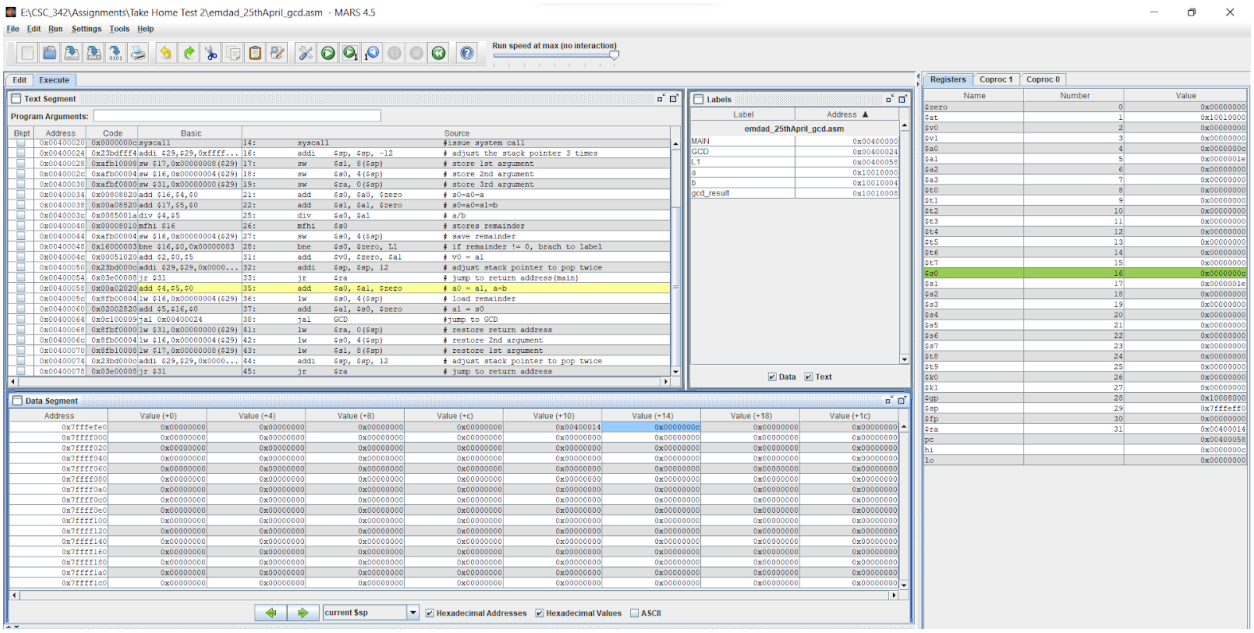

*Figure 18: add $a0, $a1*

In figure 19, we can see that line 36 is loading the remainder and the register moved to $a0 which value is 0x00001e. The offset value is 0 for +14.

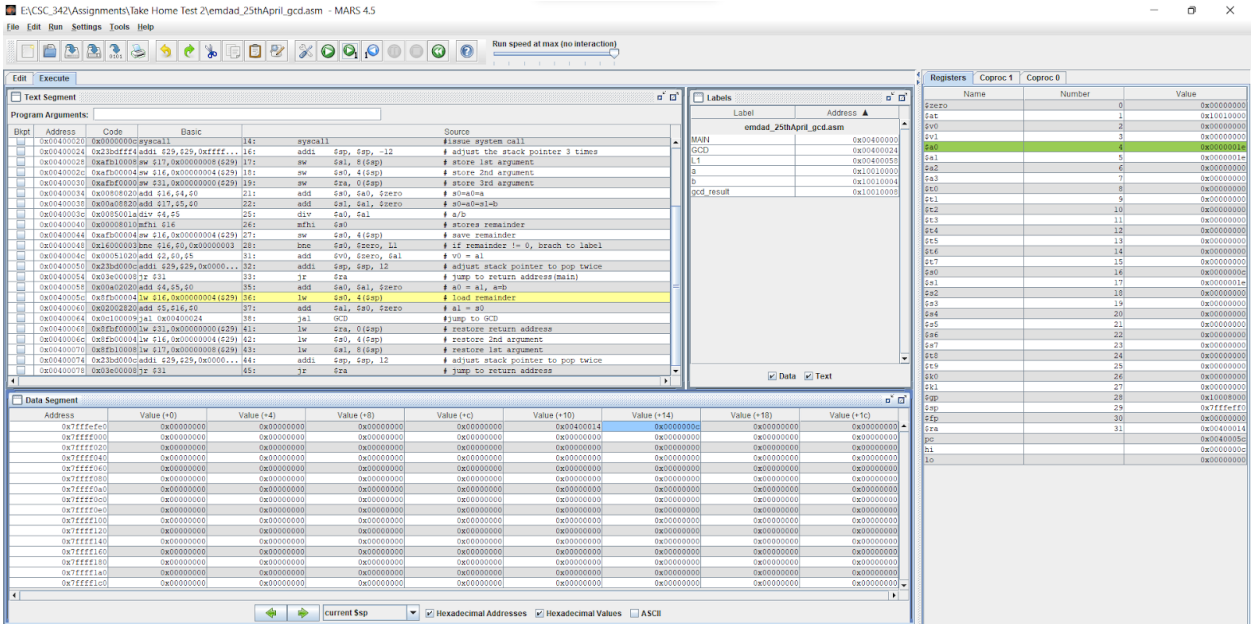

*Figure 19: loading remainder*

In figure 20, we can see that the values of $a0 is switching to $a1 and the value for register $s0 is 0x0000000c.

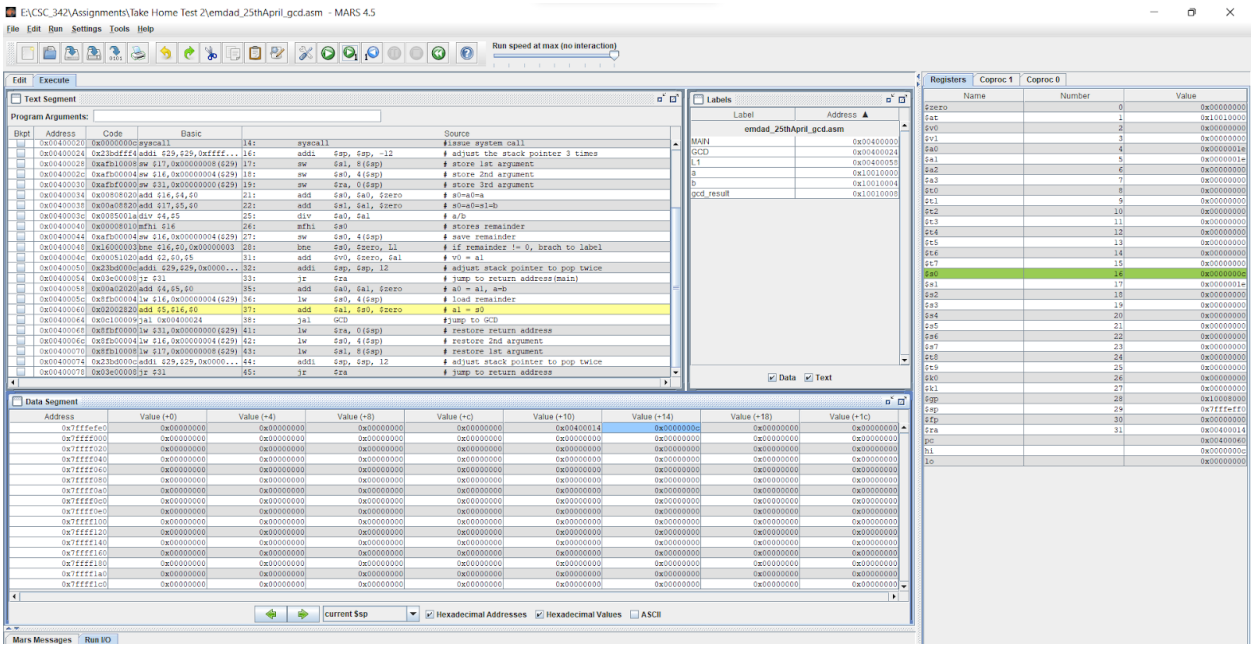

*Figure 20: add $a1, $a0*

In figure 21, we can see that the program jumps to GCD again.

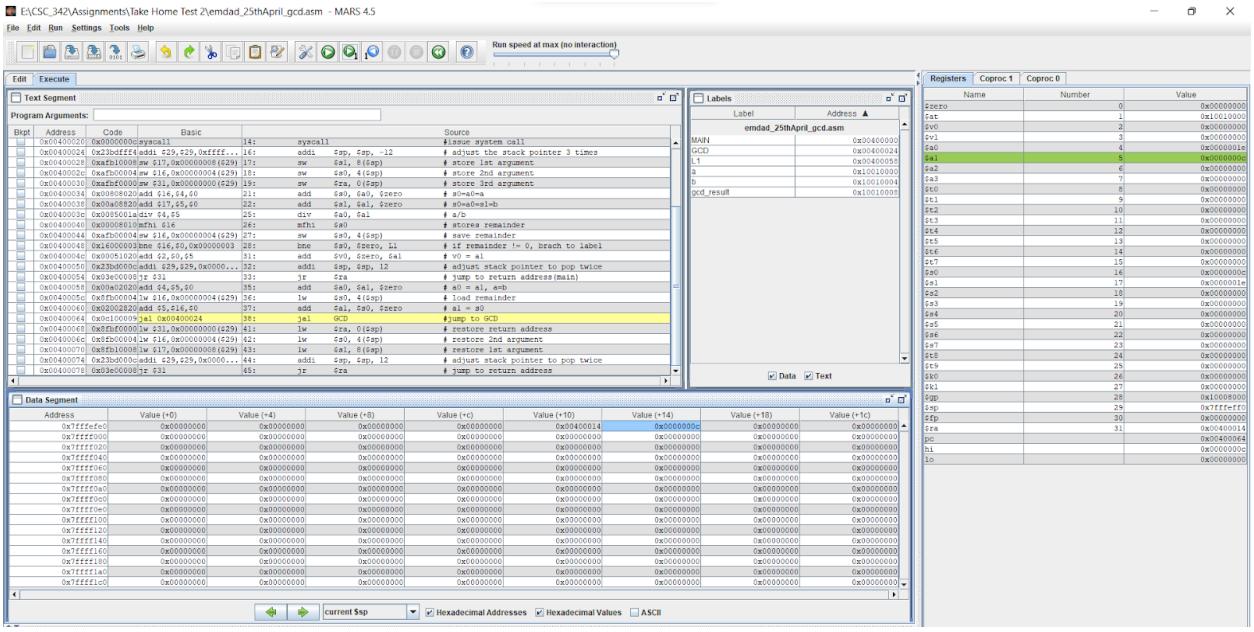

*Figure 21: jump to gcd*

In figure 22, we can see that we are adjusting the stack pointer now.

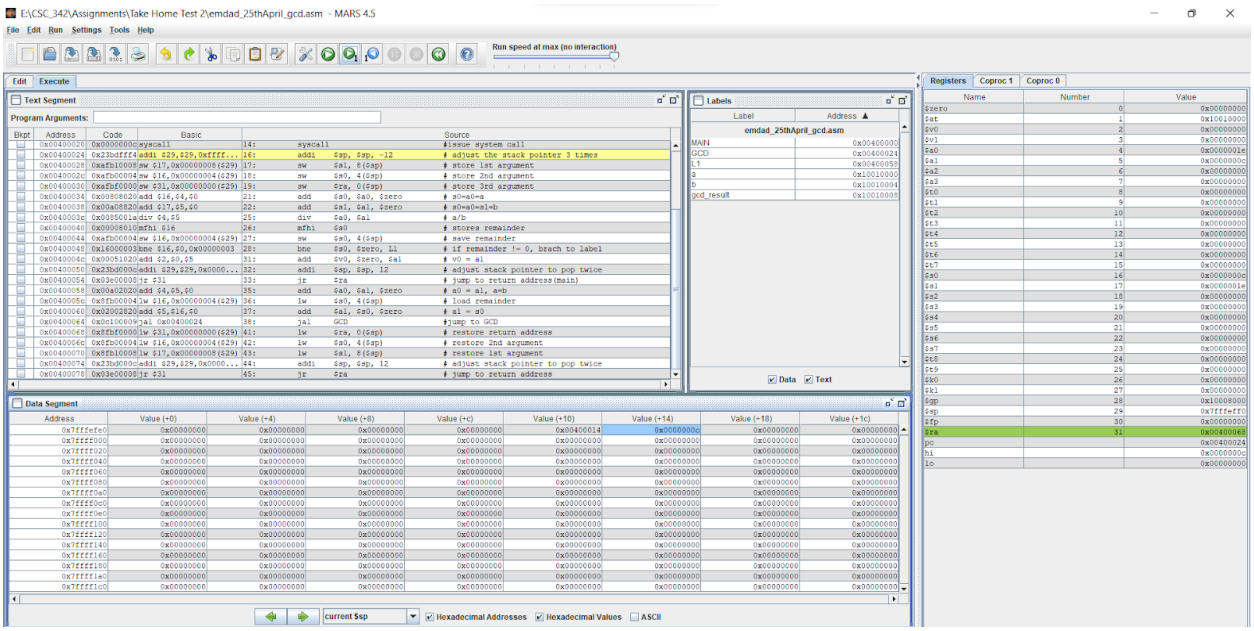

*Figure 22: adjusting stack pointer*

In figure 23, we can see that it is storing 1st argument and the register, $sp, value is 0x7fffefe4.

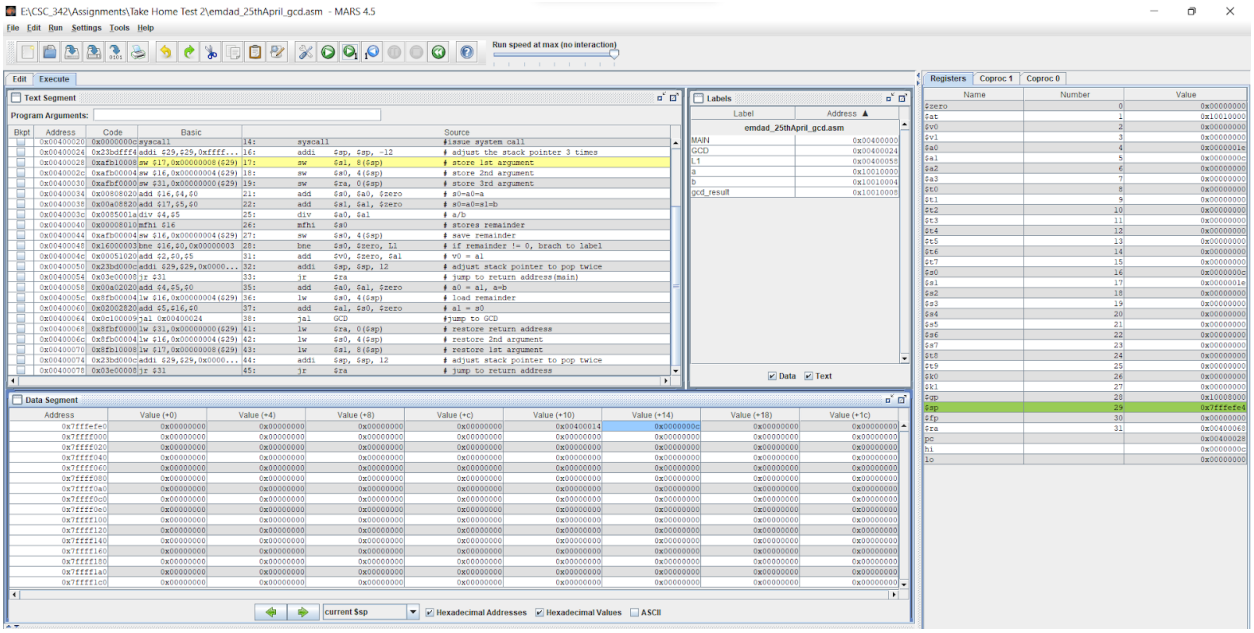

*Figure 23: storing 1st argument*

In figure 24, we can see that it is storing 2nd argument and the register, $sp, value is 0x7fffefd8. The offset is 0 and value is 0x0000000c.

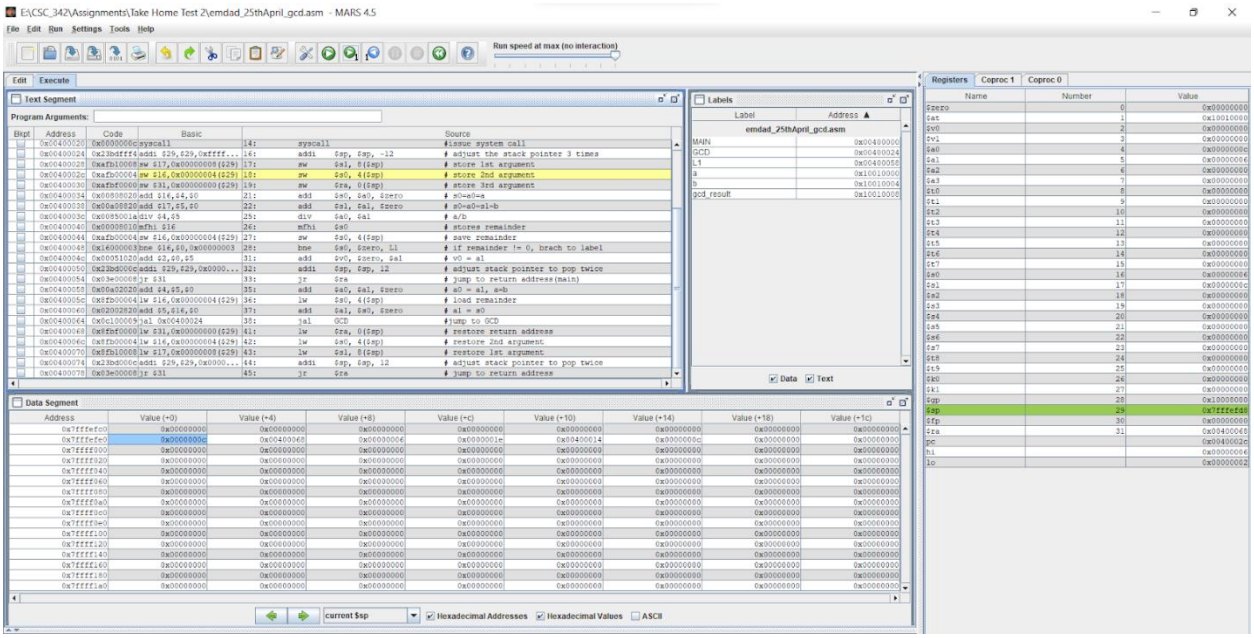

*Figure 24: storing 2nd argument*

In figure 25, we can see that it is storing 3rd argument and the register, $sp, value is 0x7fffefd8. The offset is 0 and value is 0x0000000c.

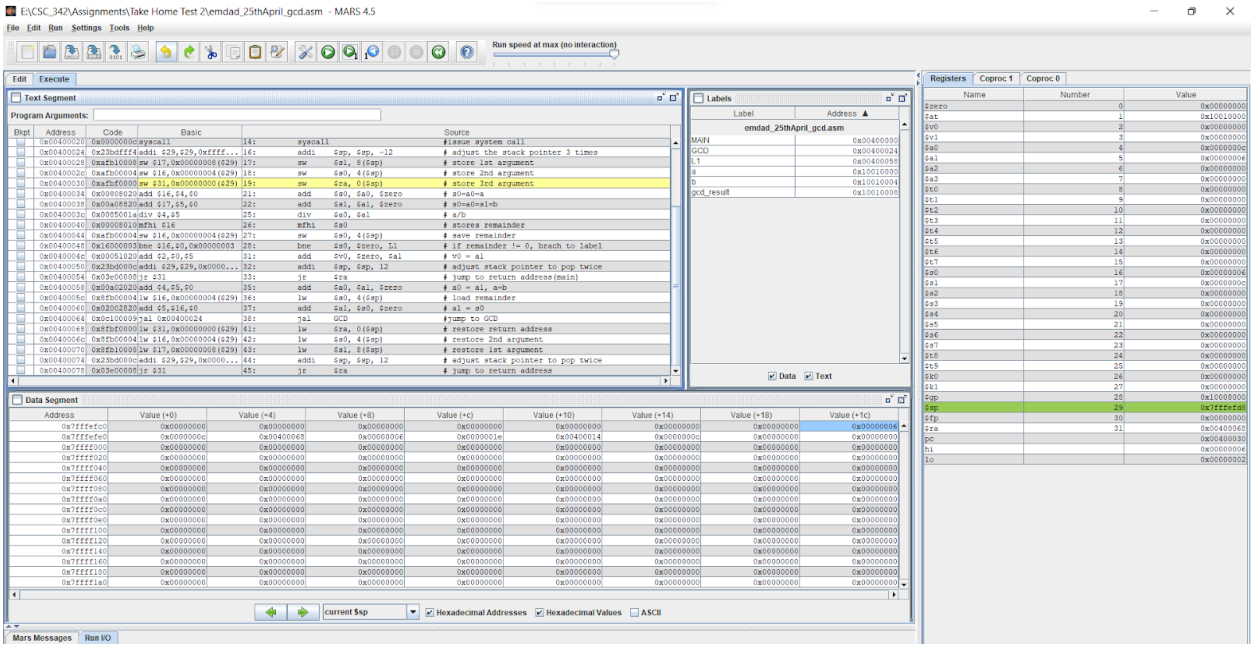

*Figure 25: storing 3rd argument*

In figure 26, we can see that line 41 is restoring the return address. The register moved to $ra and value is 0x00400068 and the memory offset is +1c and value is 0x00000000.

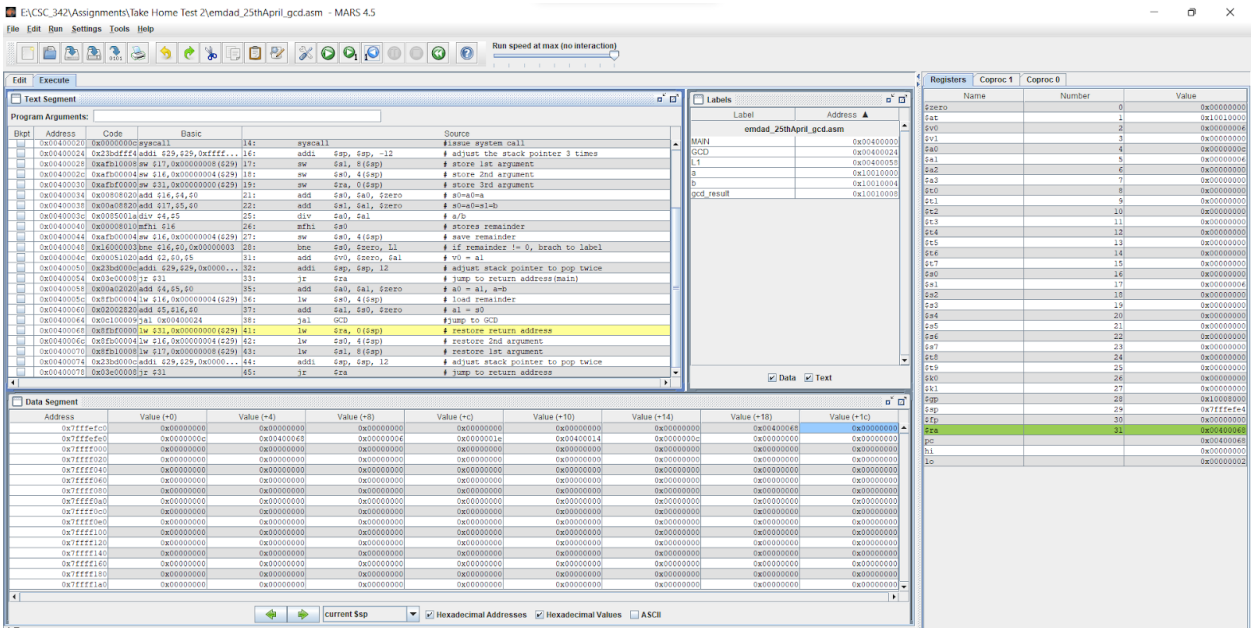

*Figure 26: restore return address*

In figure 27, we can see that line 42 is restoring 2nd argument.

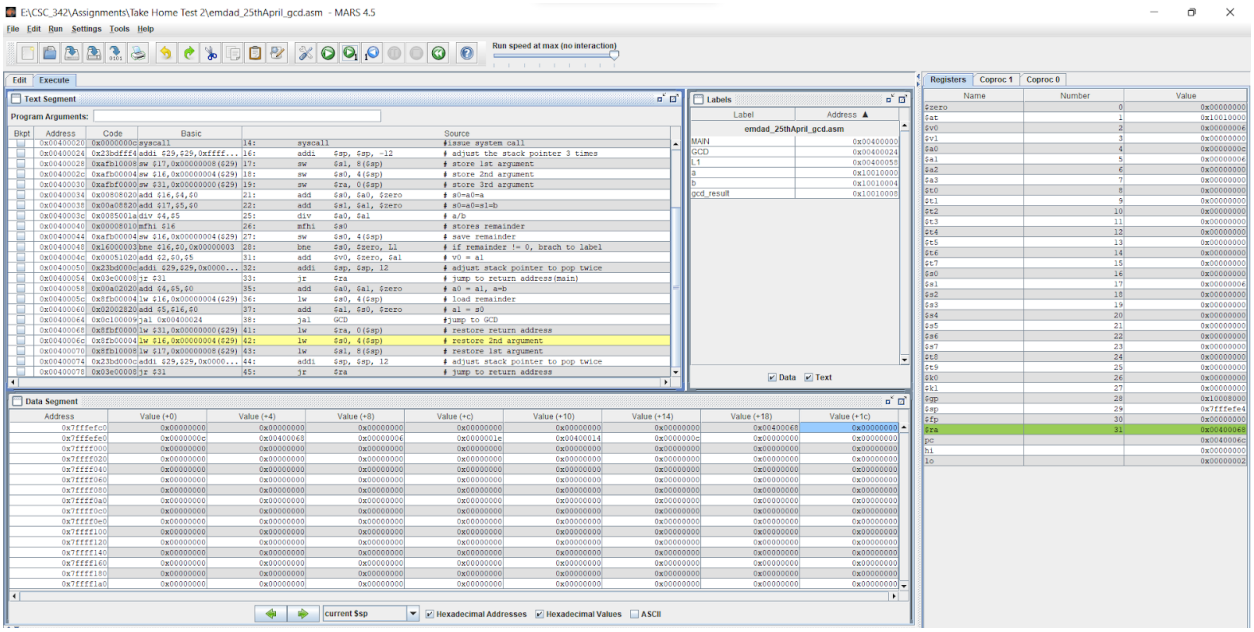

*Figure 27: restoring 2nd argument*

In figure 28, we can see that line 43 is restoring 1st argument.

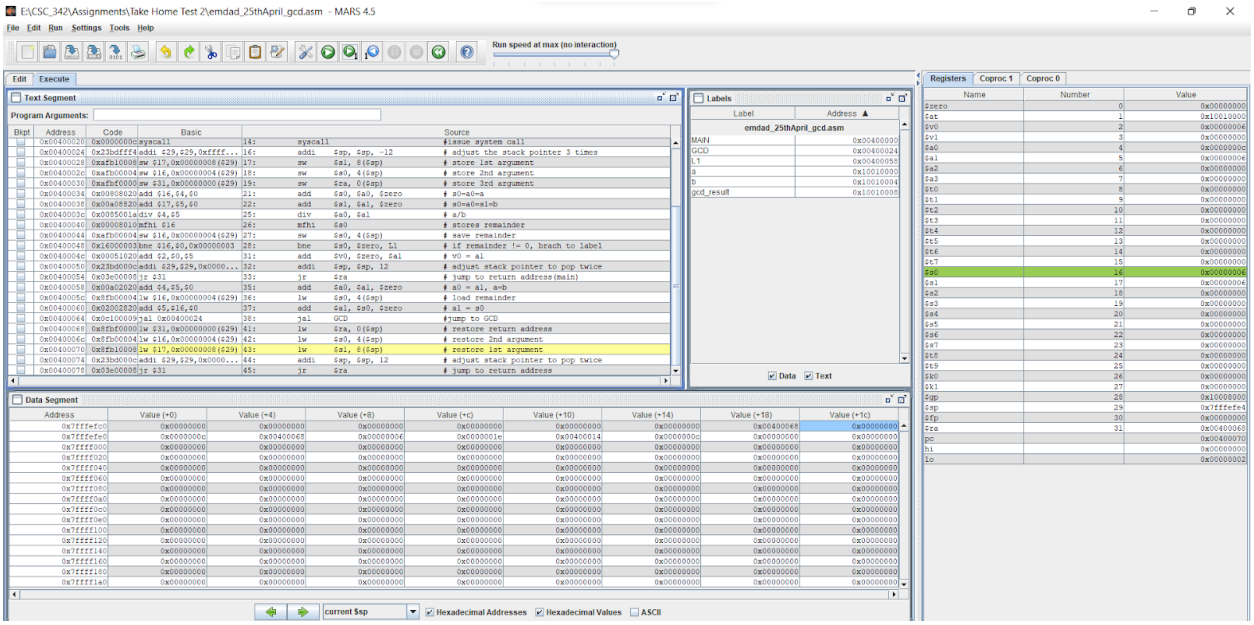

*Figure 28: restoring 1st argument*

In figure 29, we can see that line 43 is adjusting the stack pointer and pop registers.

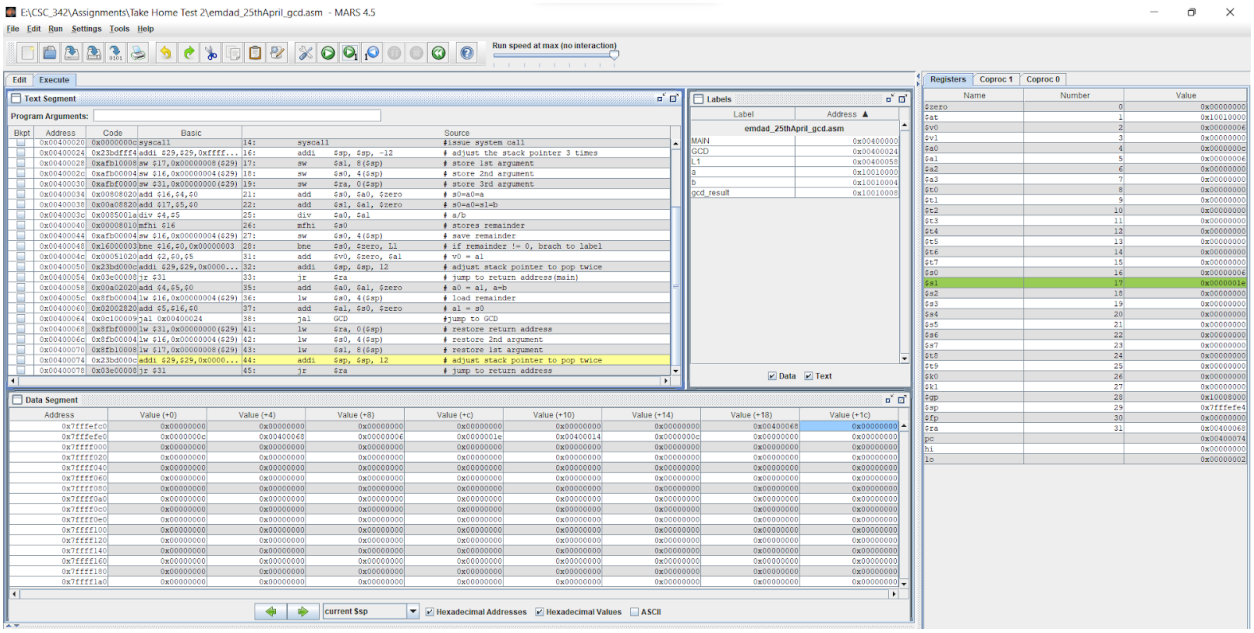

*Figure 29: adjusting the stack pointer*

In figure 30, we can see that line 45 jumps back to return address and register is $sp and value of its is 0x7fffeff.

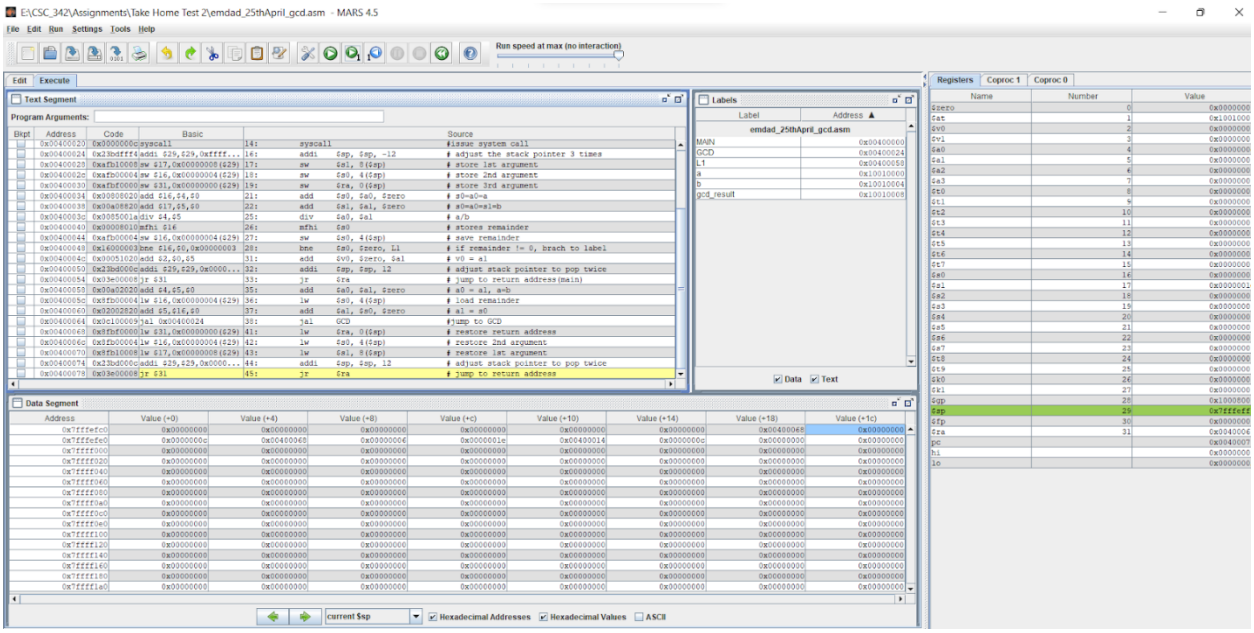

*Figure 30: jumps back to return address*

In figure 31, we can see that the syscall is closed and process ends. The value of $v0 is 0x000000a. The offset +8 has the value of 6 which is our gcd.

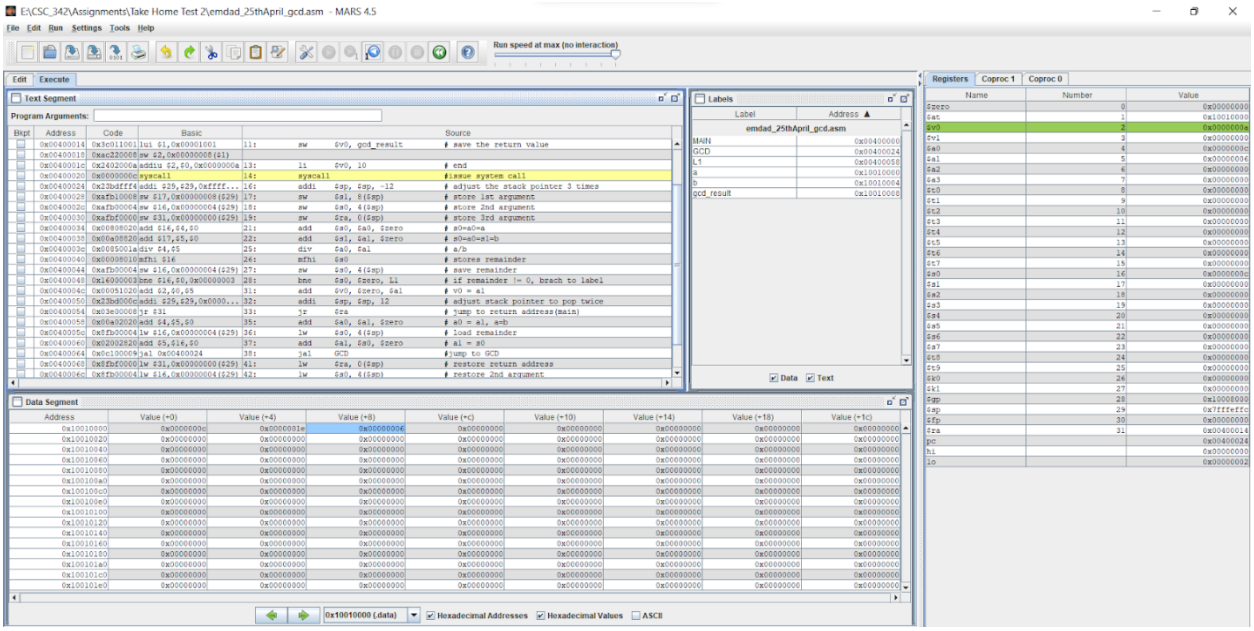

*Figure 31: returns gcd*

***PART II: Visual Studio 2019 (Intel x86):***

In figure 32, we can see the project directory for “emdad_25thApril_gcd”.

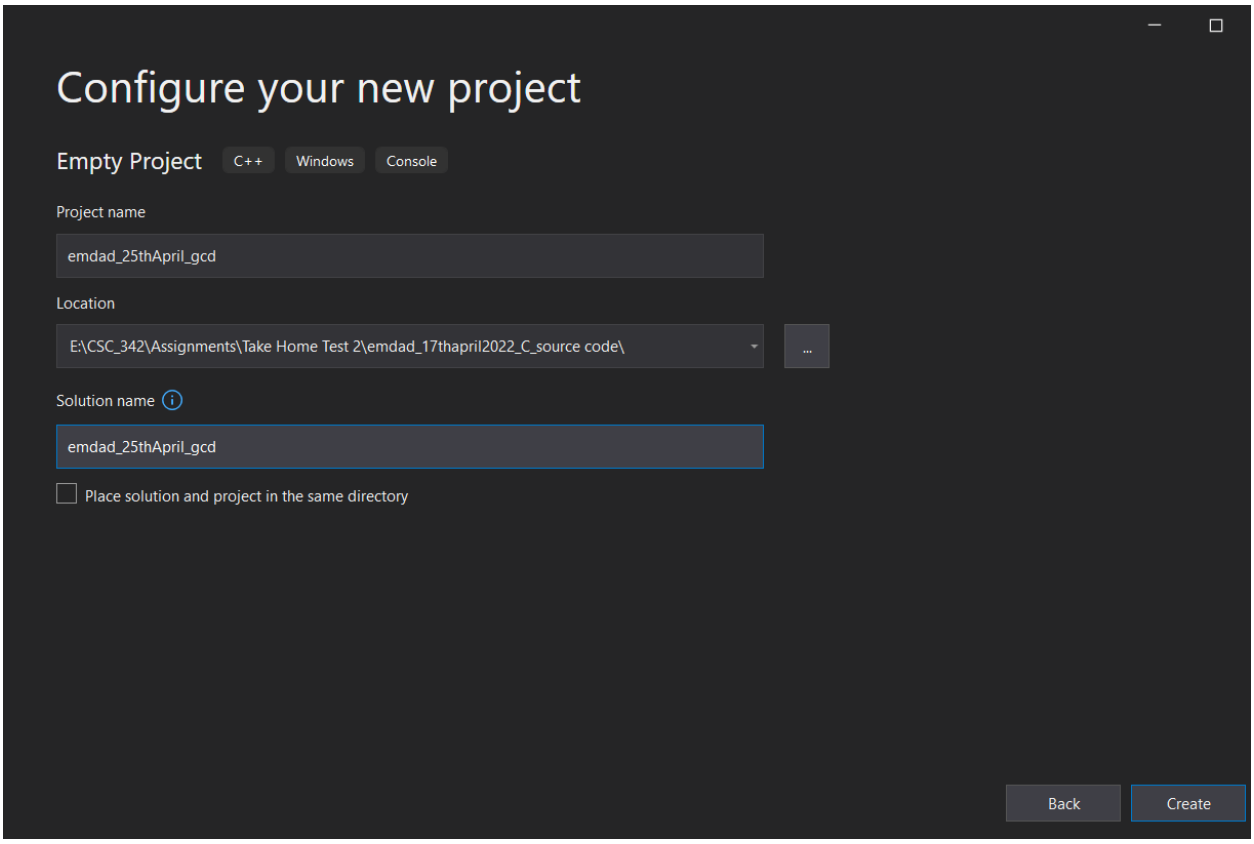


*Figure 32: project directory for “emdad_25thApril_gcd”*

In figure 33, we can see that I am adding .c file to the project which will perfom gcd(a,b).

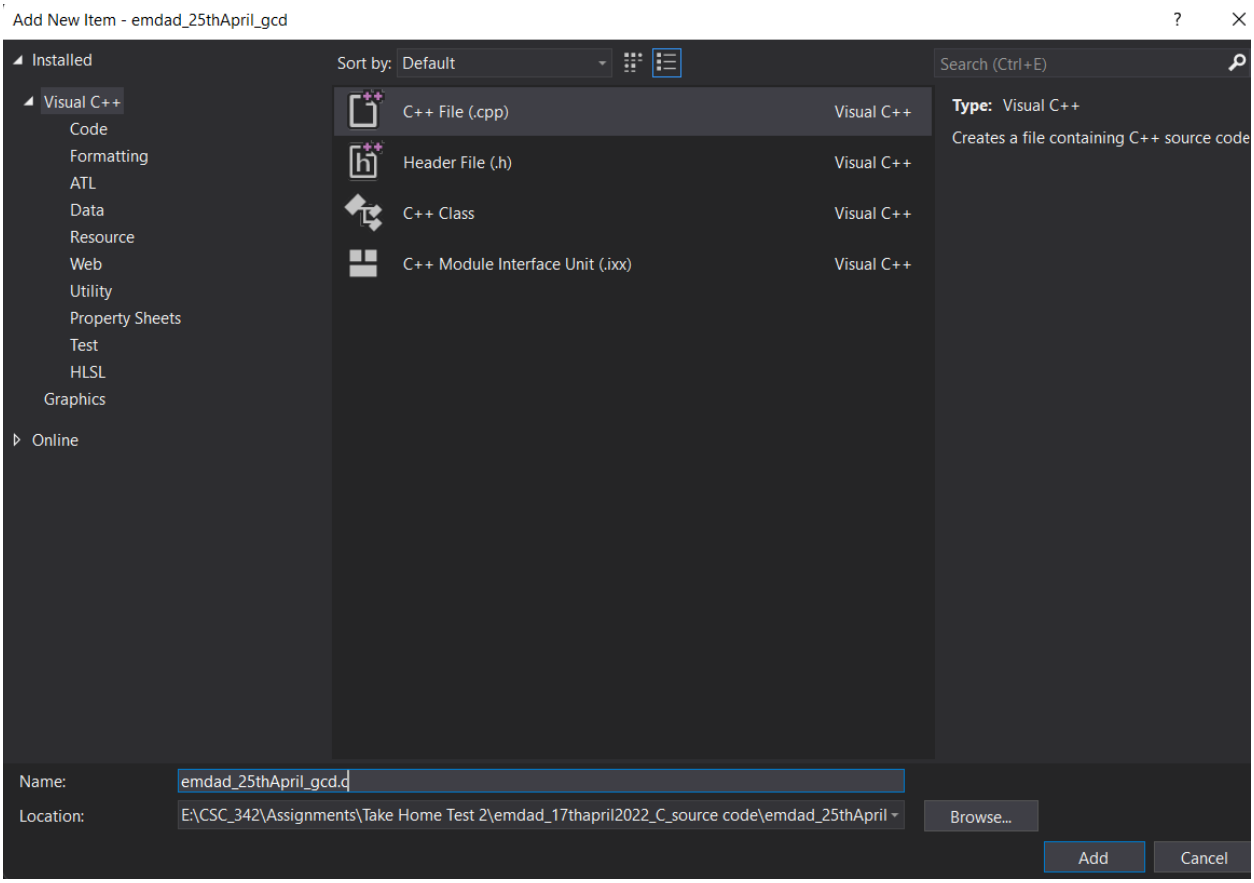


*Figure 33: creating .c file*

In figure 34, we can see that the code to perform gcd and it built successfully.

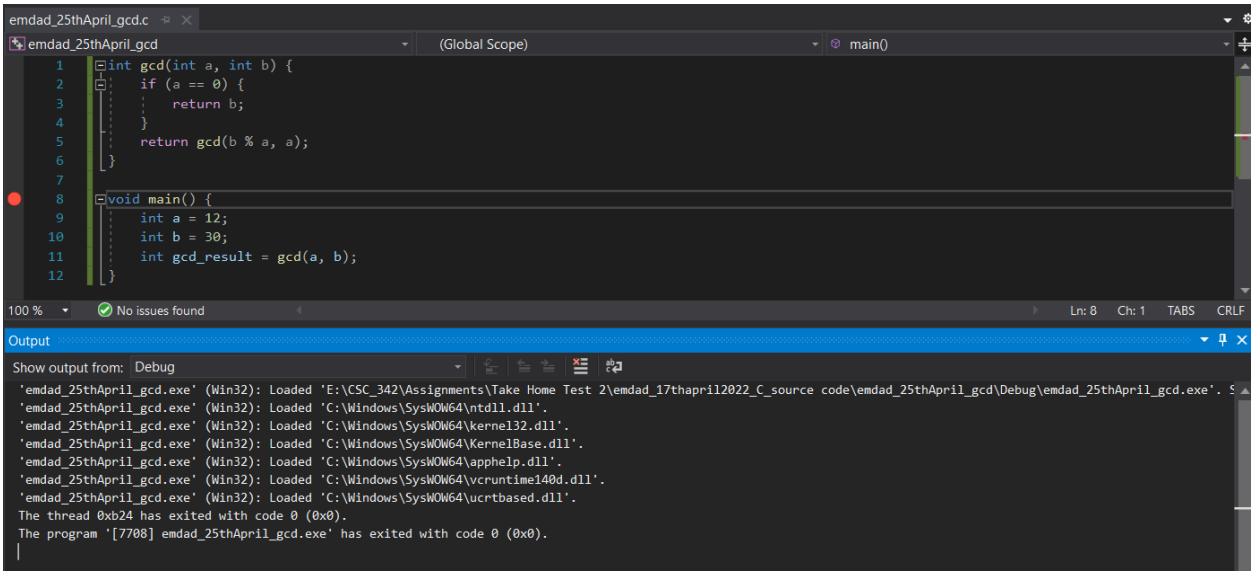


*Figure 34: gcd code and built successfully.*

In figure 35, we can see the registers window at the top and disassembly at the left side and memory address on the right side.

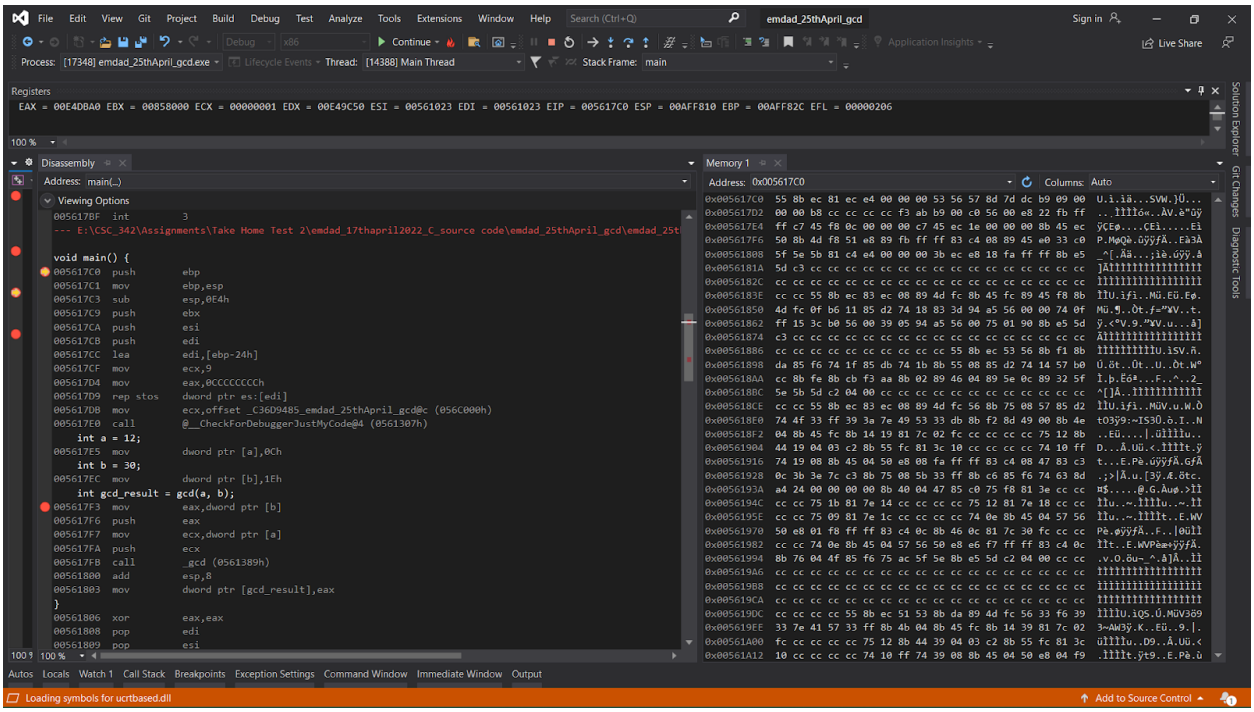

*Figure 35: registers, disassembly, and memory address window*

In figure 36, we can see the registers, EAX = 0000001E, ECX = 0000000C EIP = 00C31750 ESP = 0077F928. The memory address 0x00561840 and 0x005618D0 has same EIP and EAX.

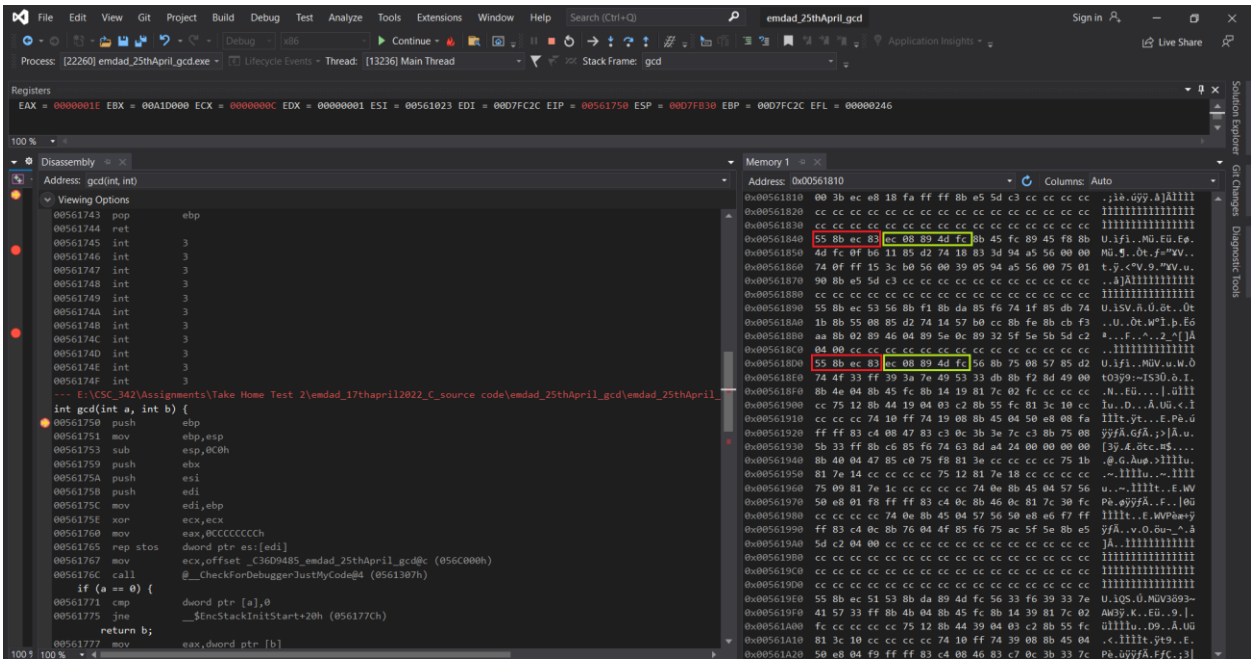

*Figure 36: breakpoints (Part 1)*

In figure 37, we can see that EAX = 00000002 EBX = 00408000 ECX = 00C3C000 EDX = 00000006 ESI = 00C31023 EDI = 006FFD84 EIP = 00C31750 ESP = 006FFCAC EBP = 006FFD84 EFL = 00000206.

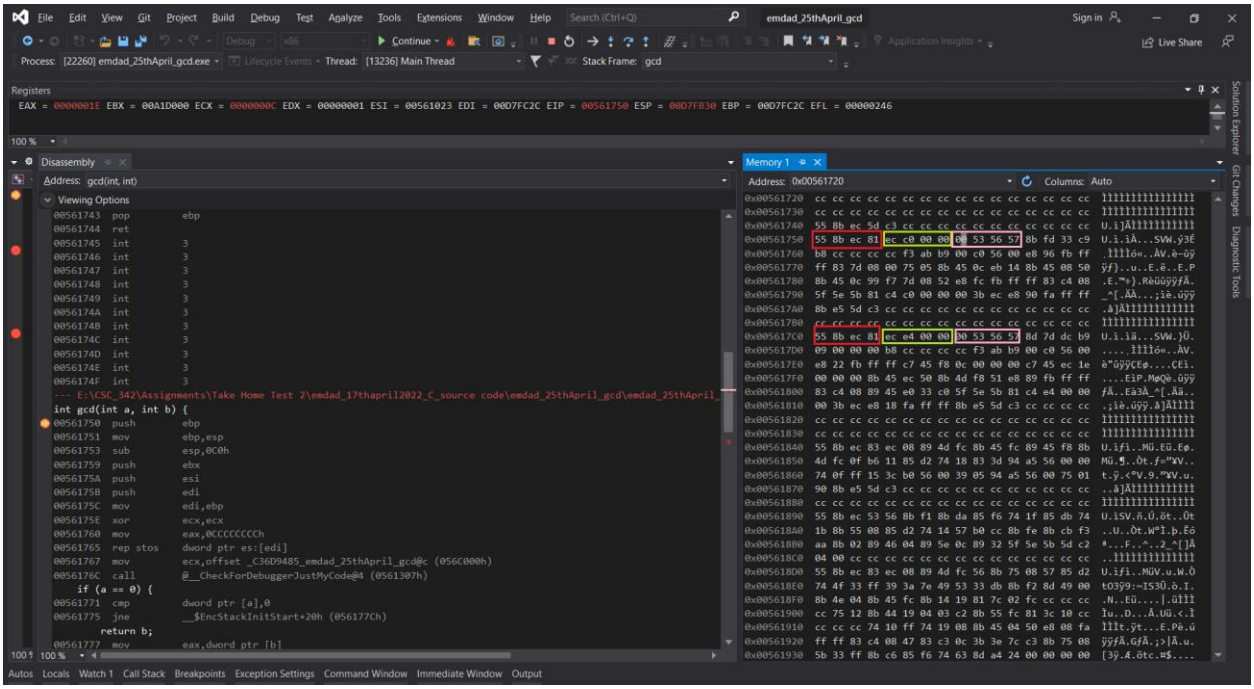


*Figure 37: breakpoints (Part 2)*

In figure 38, we can see that EAX = 00C3C000 EBX = 00408000 ECX = 00C3C000 EDX = 00000001 ESI = 00C31023 EDI = 006FFCA8 EIP = 00C3177C ESP = 006FFBDC EBP = 006FFCA8 EFL = 00000206.

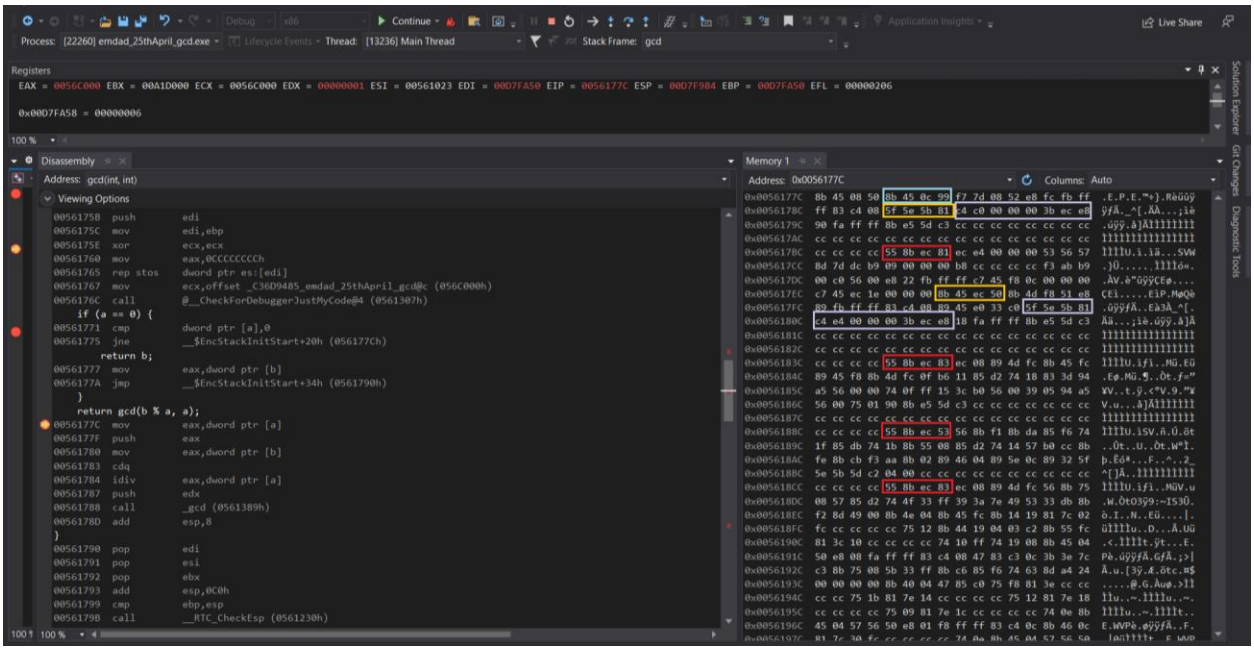

*Figure 38: breakpoints (Part 3)*

## ***PART III: x86_64 ISA, Linux 64-bit:***

In figure 39, we can see the source code for emdad_25thApril_gcd.c in linux editor.

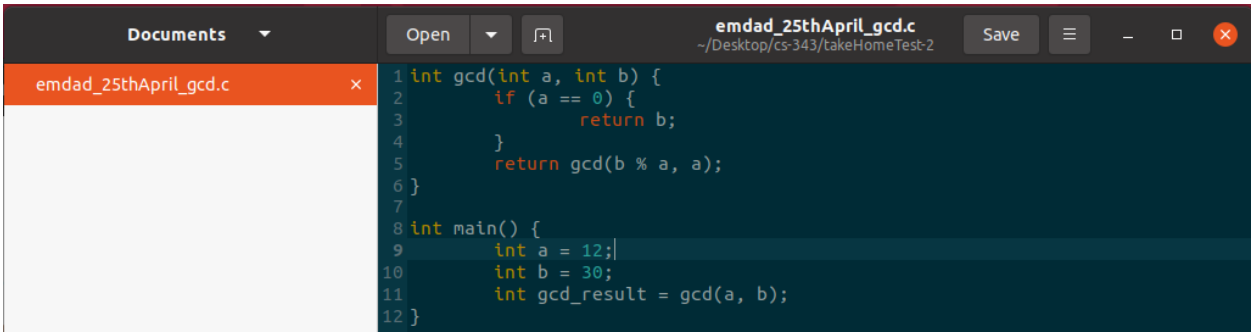


*Figure 39: source code for emdad_25thApril_gcd.c*

The main function started at line 10 and then there are some variables initialized. We called the gcd function in the main function and printed the value of it. The gcd function is a recursion function which call itself and keep running until a == 0. The gcd for 12 and 30 is 6.

In figure 40, we can see that we compiled the code and used gdb a.out to run gdb. Then we did break main to put a breakpoint on the main function. Then, I did run to run the gbd and select-frame is 0.

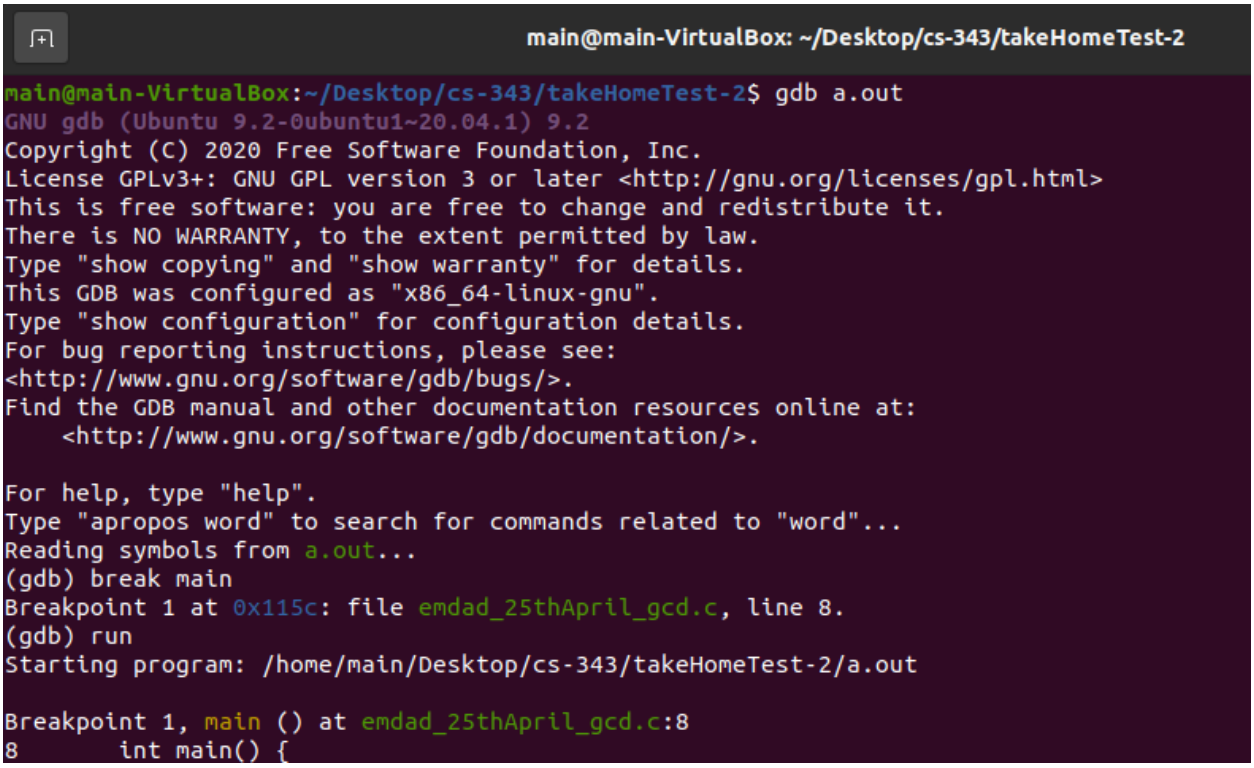


*Figure 40: code compilation and run gdb*

In figure 41, we can see the disassembly code.

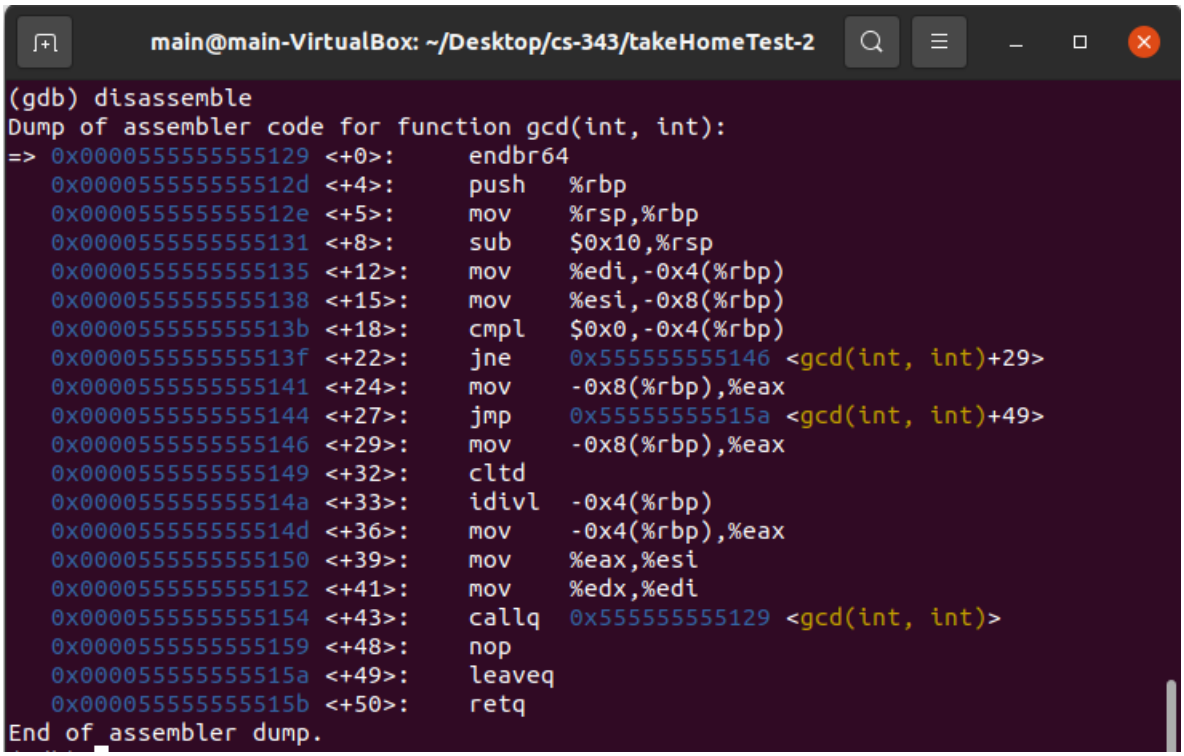


*Figure 41: disassembly code.*

After running the disassemble code, we get the assembler code of function main. The rsp is the stack pointer and the rbp stores that pointer. Callq calling the function gcd and then storing the address of the new generated value. The mov instruction stores all  local values in registers.

```
(gdb) nexti 3
0x0000555555555131      1       int gcd(int a, int b) {
```

*Figure 42: nexti command*

In figure 42, we can see the nexti 3 online and the address is 0x0000555555555131.

```
(gdb) printf "rbp:%x\nrsp:%x\n",$rbp,$rsp
rbp:ffffdf40
rsp:ffffdf40
```

*Figure 43: print rbp and rsp*

In figure 43, we can see the print value of rbp and rsp by using the command:
printf "rbp:%x\nrsp:%x\n",$rbp,$rsp

```
(gdb) x/12xw $rsp
0x7fffffffdf40: 0xffffdf60      0x00007fff      0x55555185      0x00005555
0x7fffffffdf50: 0xffffe050      0x0000000c      0x0000001e      0x00000000
0x7fffffffdf60: 0x00000000      0x00000000      0xf7de70b3      0x00007fff
```

*Figure 44: print RBP, return address, argument*

In figure 44, we can see that the saved RBP which is 0xfffe050 and 0x0000000c and return address is 0x0000001e and 0x00000000.

In figure 42, we can see the info registers of the code.

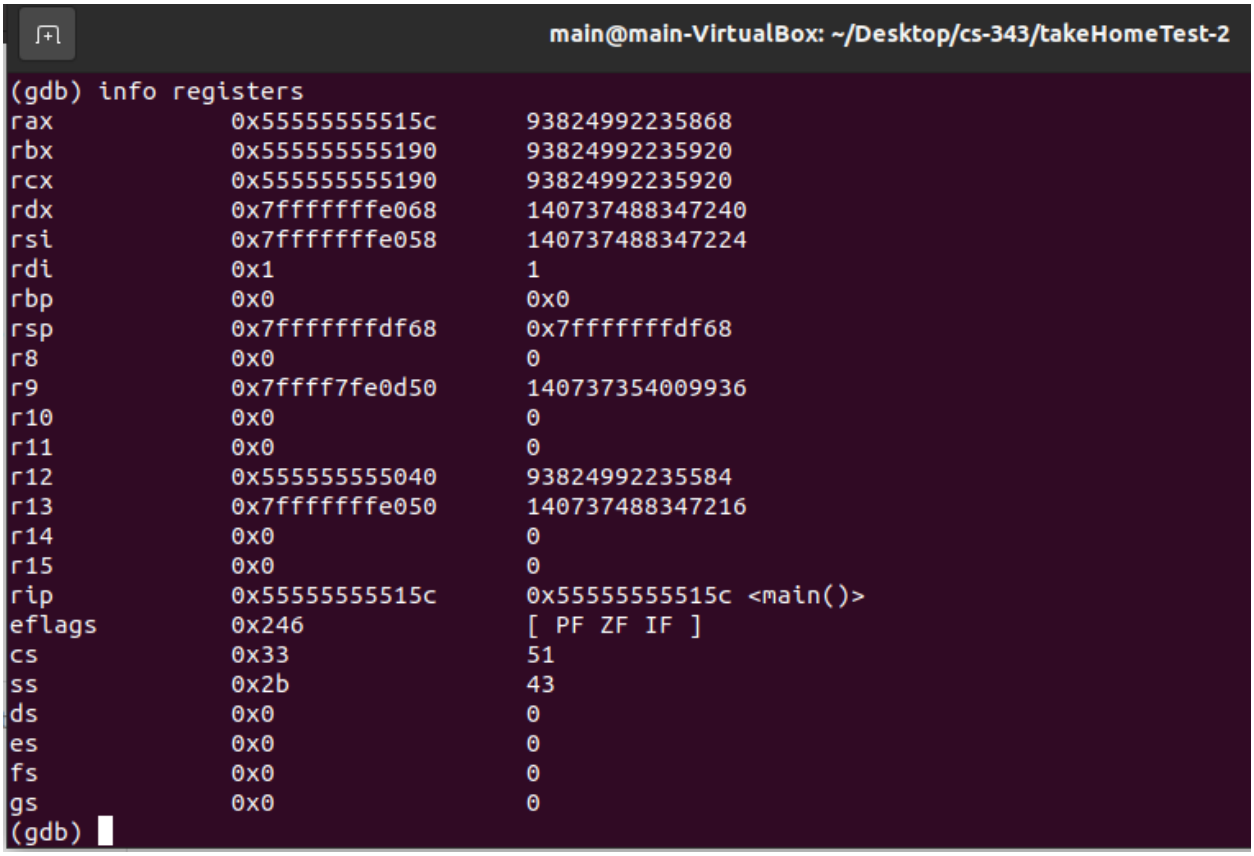


*Figure 45: info registers of the code.*

The RBP register is the base pointer. The add instruction calculates the addition of registers edx and eax. These two registers contain local data. Subcommands are subtracted. Memory address of a local static variable.

## Explanation:

The MIPS MARS simulator is a 32-bit processor, and each instruction it executes is also 32-bit long. MIPS processors have registers such as zeros, stack pointers, frame pointers, and program counters. A stack frame is a data frame (that is, an area of memory) that is created (that is, an area of memory). In the (allocated) stack. A new stack frame is created each time the function is called. The new frame is initialized by pushing the old base pointer onto the stack. The new base pointer. It is initialized and allocates space (by subtraction) for the required data from the stack pointer. Next, all arguments are pushed onto the stack (using a negative offset using the local base pointer) variable). Finally, the return address (that is, the address after the function call) is shifted up. This process in the stack repeats for each function call before proceeding to the next function call. For  new stack frames. The behavior of stack frames may vary from program to program and best ways to understand stack frames is to use recursion.

In contrast to MIPS, Intel has one operand in a register. The second operand can be placed in memory. Transfer the data using the "mov" instruction. Intel implements a loop with jne and jmp. Intel function calls are made using the call instruction, which sets an instruction pointer to an instruction that requests a jump to the function. The return address is stored on the stack for later use when the function call completes. Static variables are stored in memory away from the stack. Local variables are assigned positions in the stack, and the values are initially loaded directly into memory rather than into registers.

I ran a GDB disassembler using Oracle VirtualMachine. Ubuntu Linux instructions. The data is stored in memory in little endian notation. The result of the instruction is stored in the register on the right, unlike Intel, which was stored in the register on the left. Moving data between registers and memory is done using mov. This is the same as Intel, copying the data from the left operand to the right operand.

## Time-Comparison:

In figure 46, we can see the time comparison table for input vs execution time.

| A | B | Time(s) |
|---|---|---|
| 12 | 30 | 0.000152 |
| 100 | 50 | 0.000153 |
| 1000 | 500 | 0.000154 |
| 10000 | 50000 | 0.000155 |

*Figure 46: time comparison table*

In figure 47, we can see the graph execution for the table in chart.

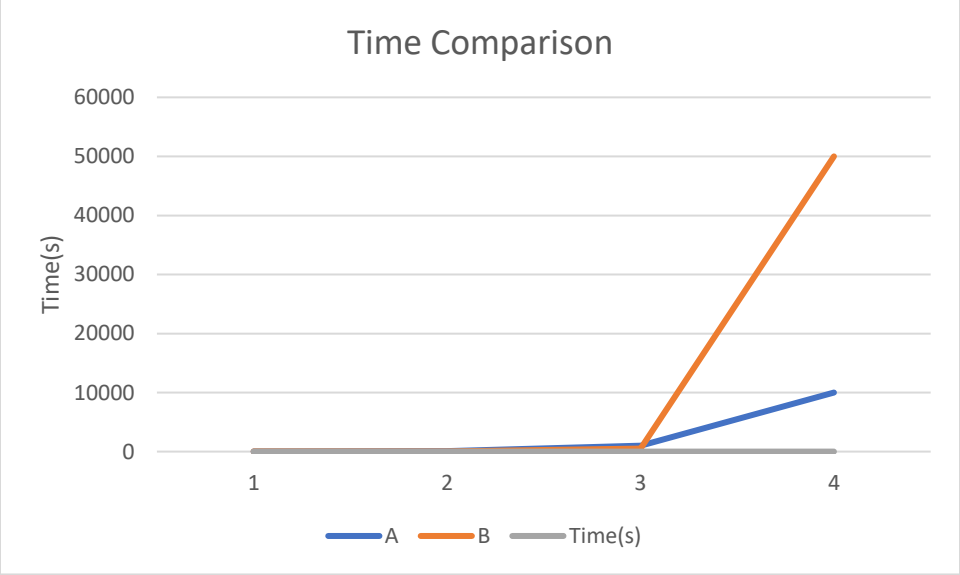


*Figure 47: graph for GCD execution time*

## Conclusion:

This take-home test taught me how to demonstrate recursive calls and stack frames on MIPS instructions set architecture, Intel x86 ISA using Windows MS 32-bit compiler and debugger and a Intel X86_64 bit ISA processor running Linux, 64 bit gcc and gdb. Further, I learned and understood about registers, memory, and disassembly and how to debug code overall.