Take Home TEST 2

Ahmad Adil

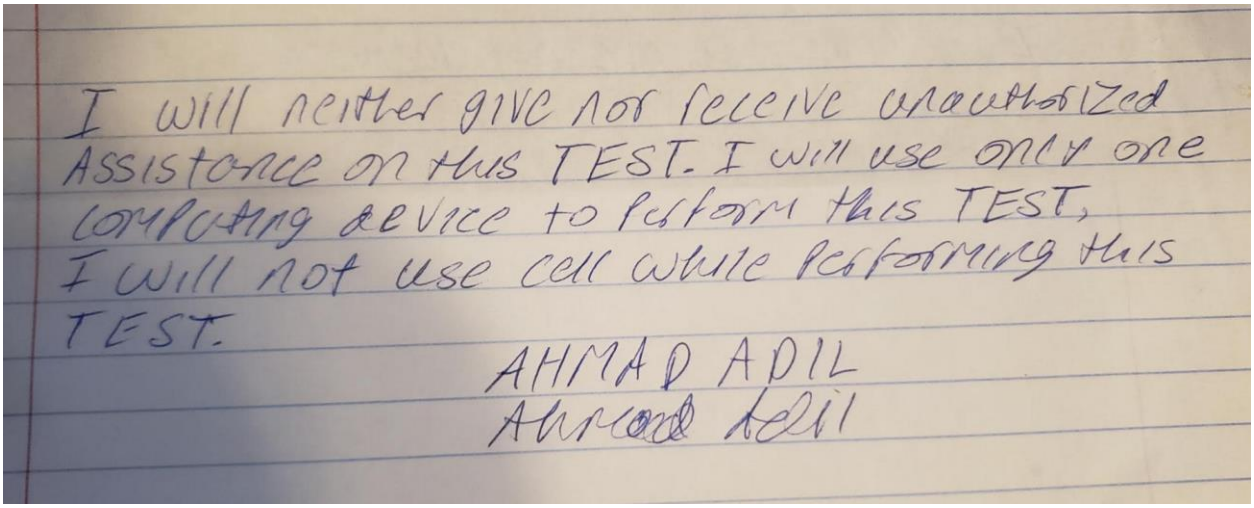
7/15/21

Professor Gertner

Computer Organization 342/343

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

Contents



Start Time and date: 4:00pm 7/13/21
End TIME and date: 4:00pm 7/14/21

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

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

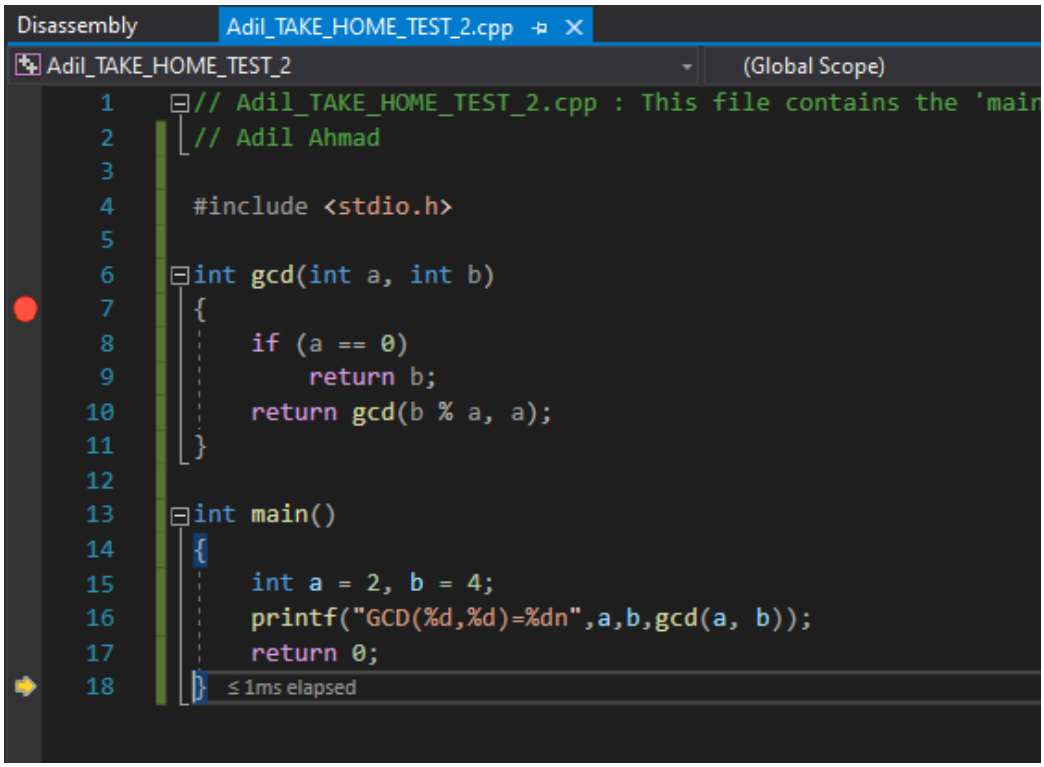


Figure 1: Adil_GCD

Here we this 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.

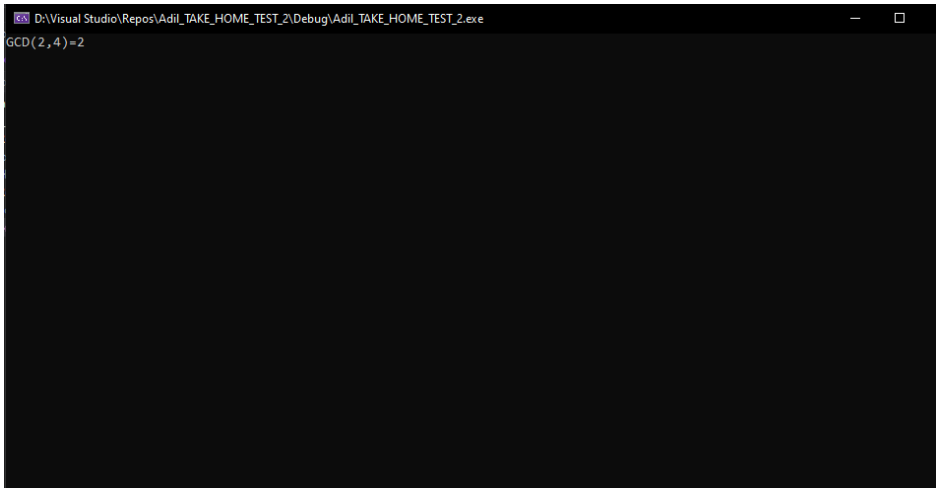


Figure 2:Program output

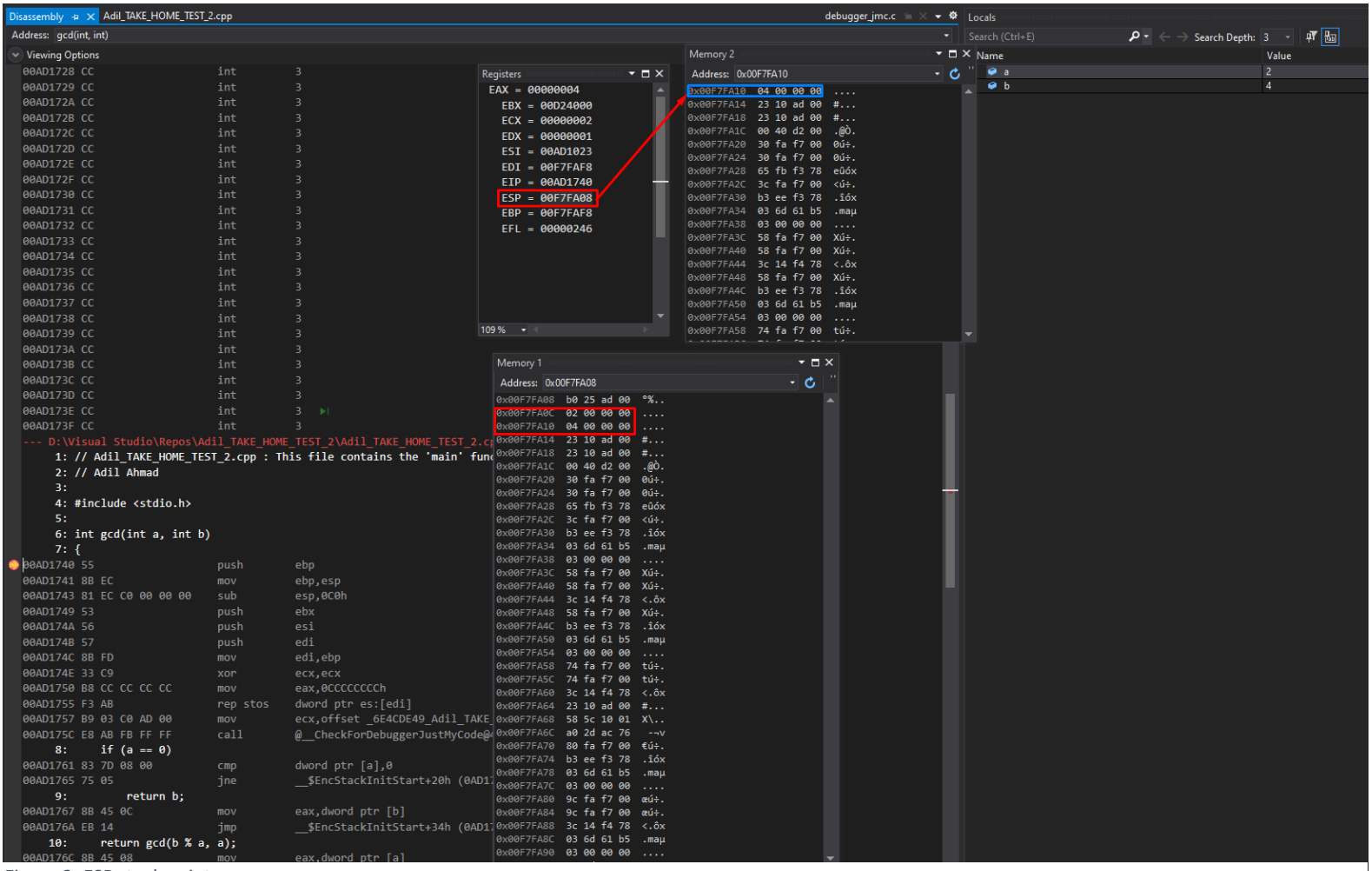


Figure 3: ESP stack pointer

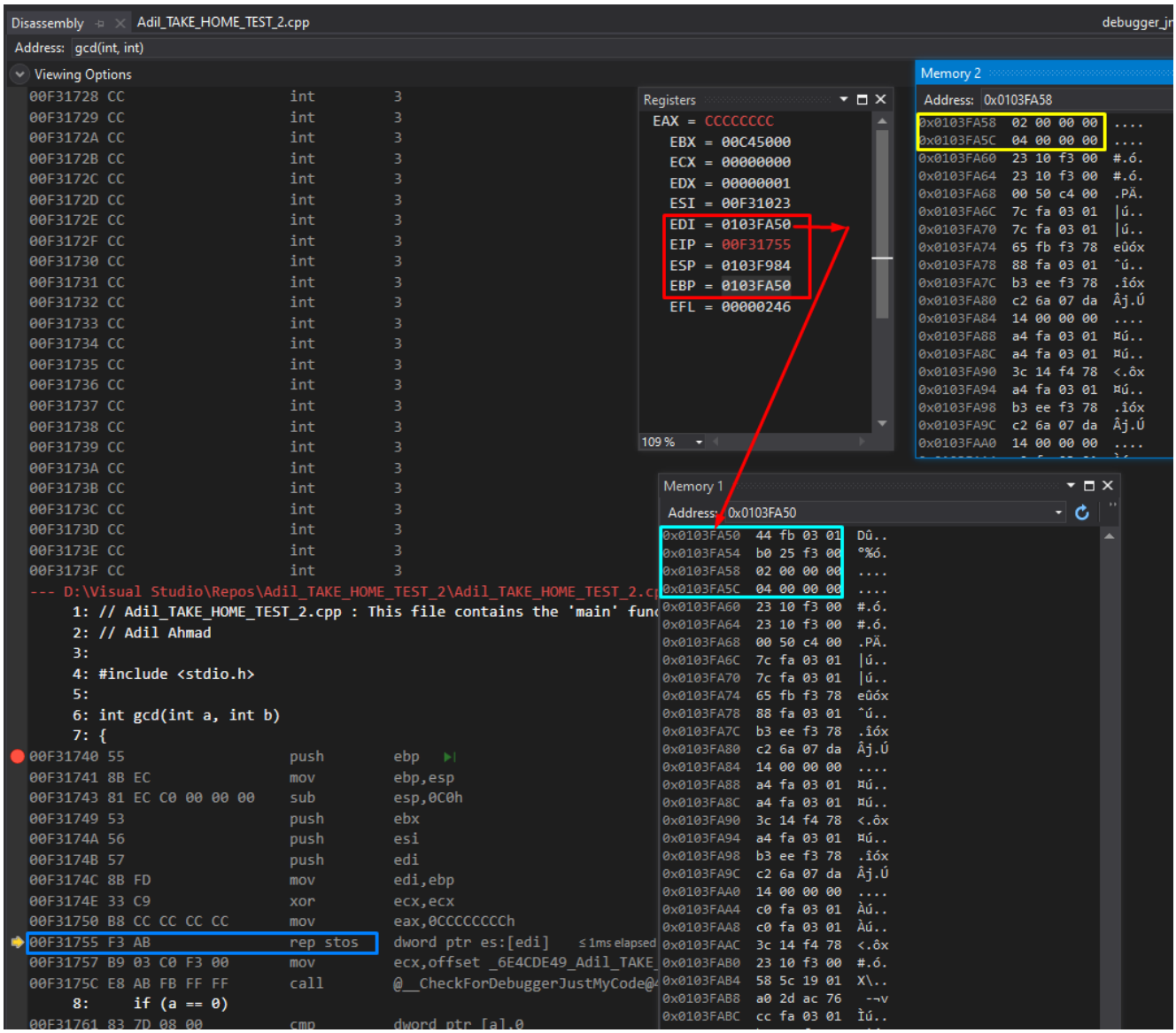


Figure 4: 2 and 4 being loaded into the stack

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 000000008 and that offset is where we can see the current stack. 0x0103FA50 + 000000008 = 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.

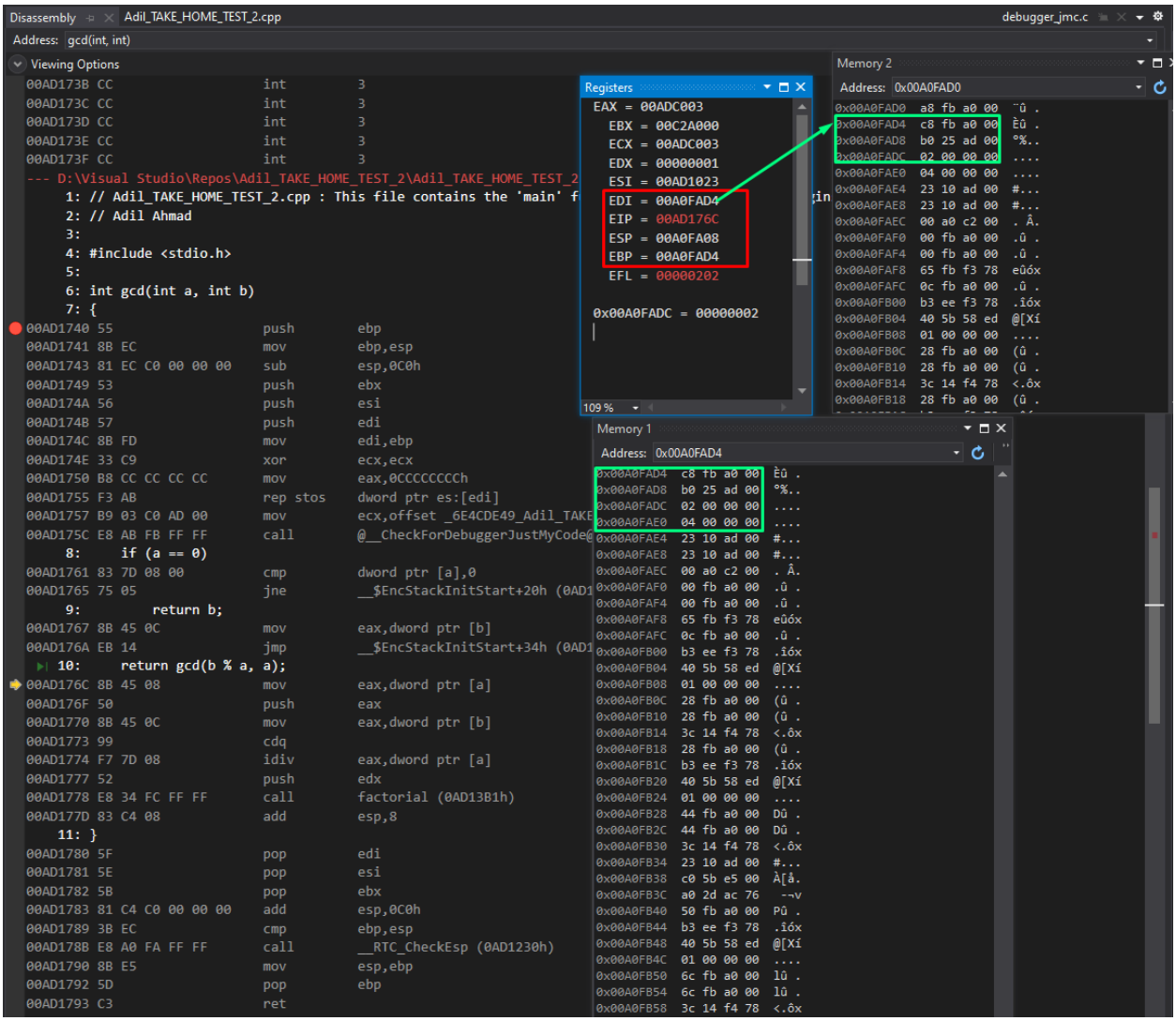


Figure 5: GCD function executing

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.

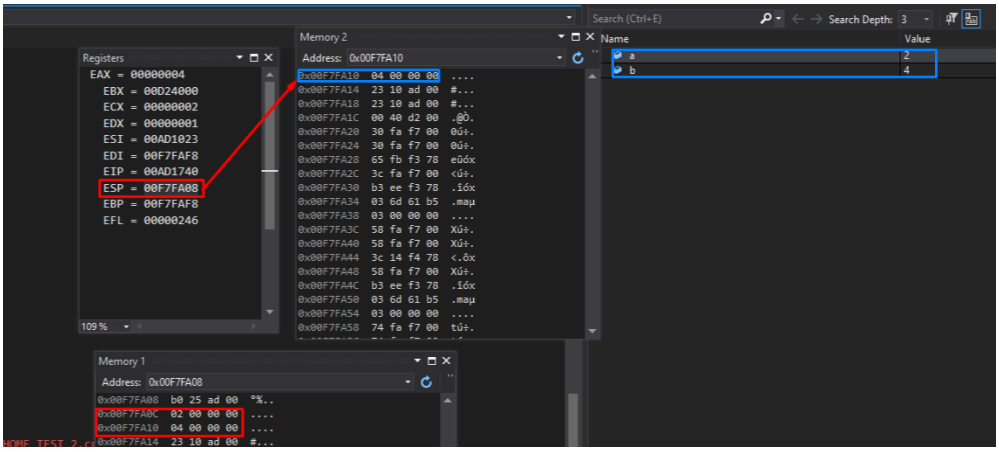


Figure 6: Original value of a and b

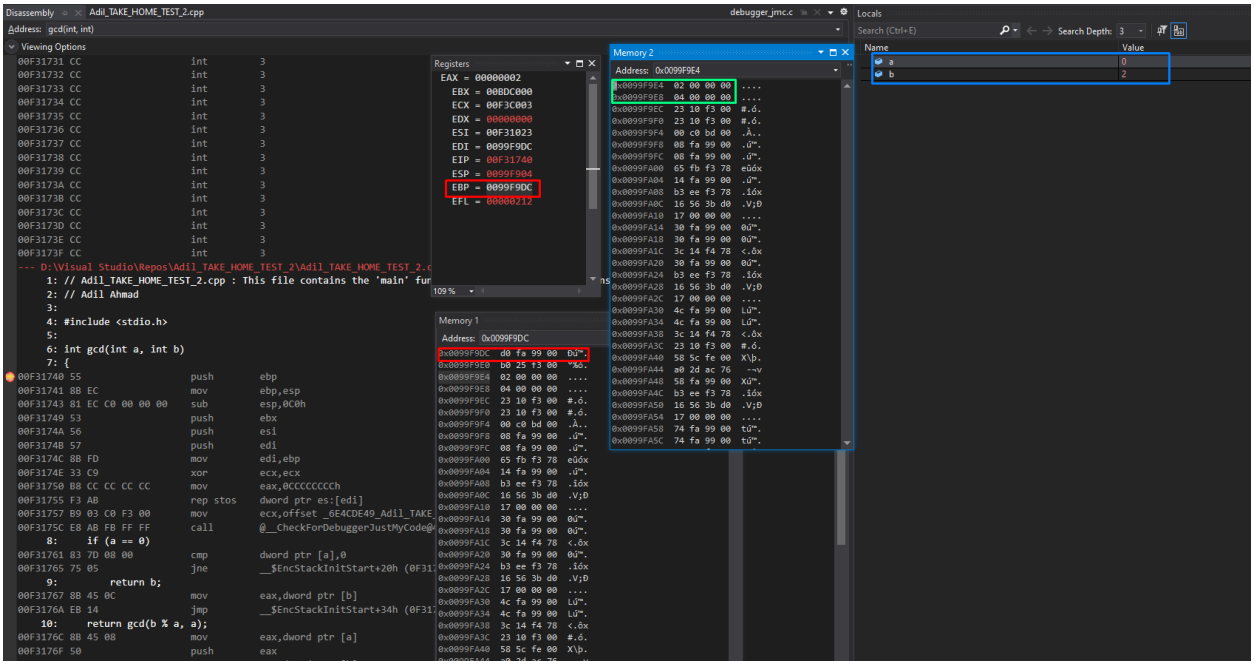


Figure 7: Value of a and b changed

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.

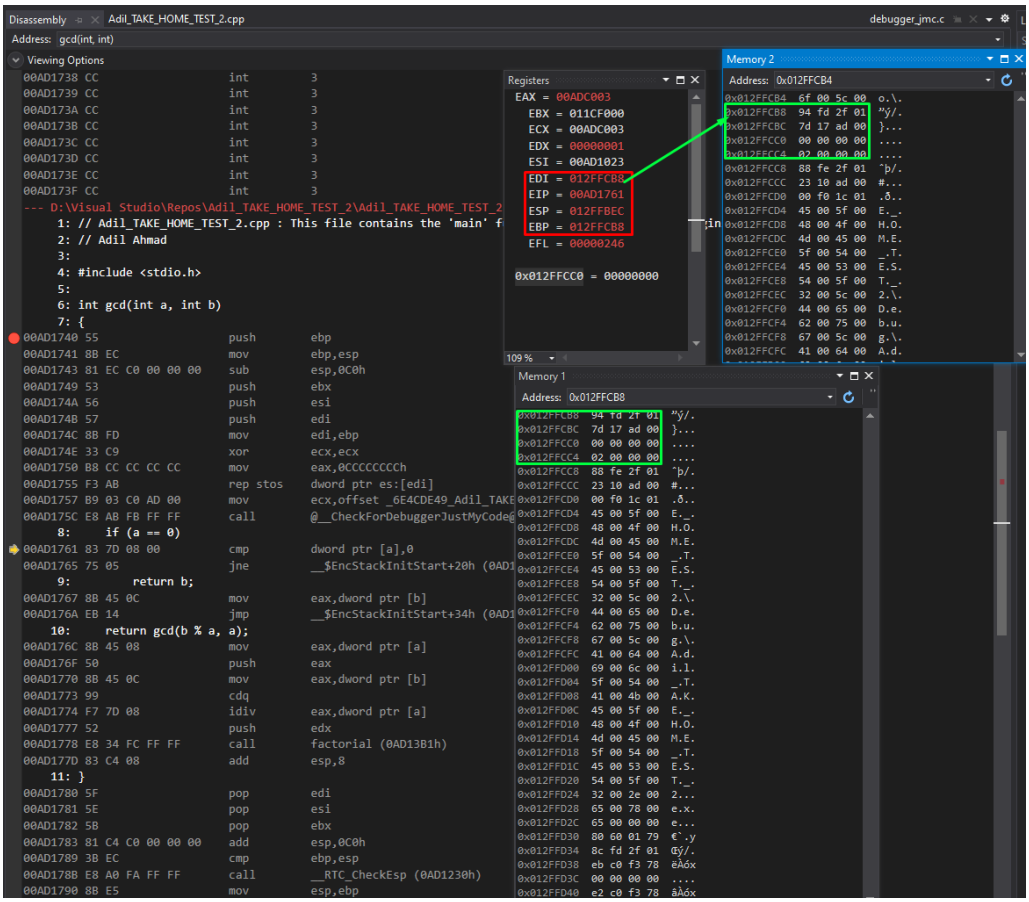


Figure 8: conditional statement on a

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 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.

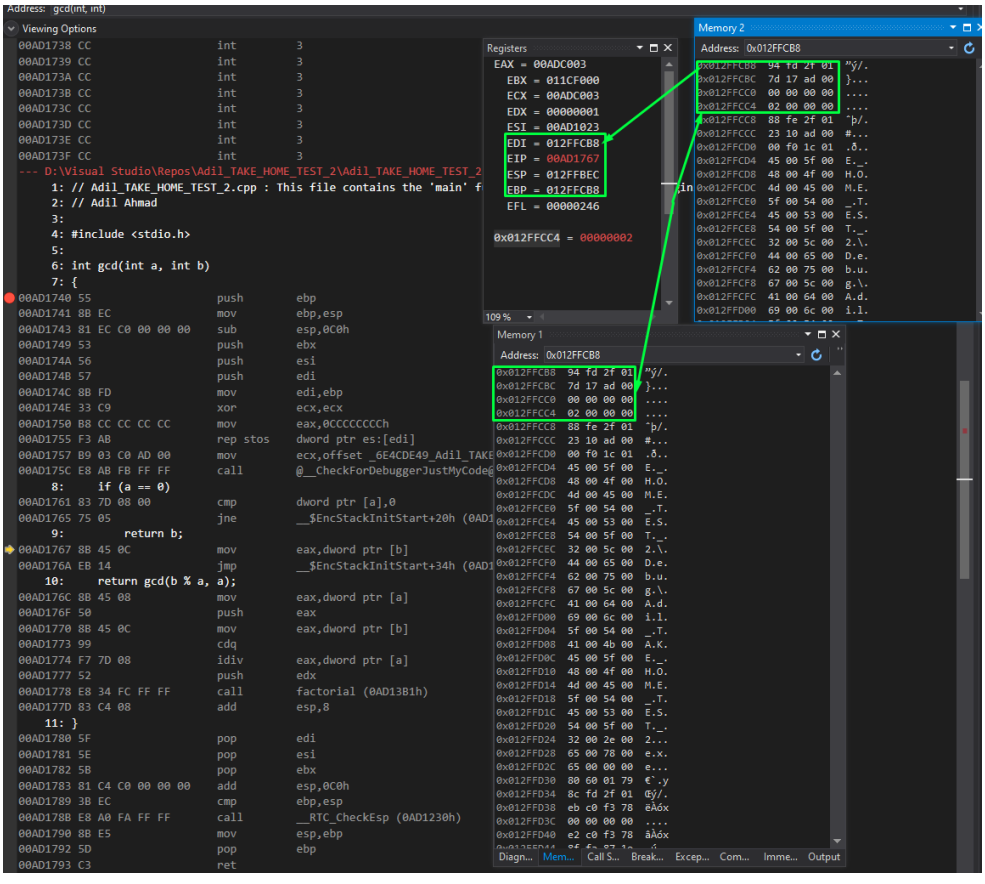


Figure 9: Return b next execution

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.

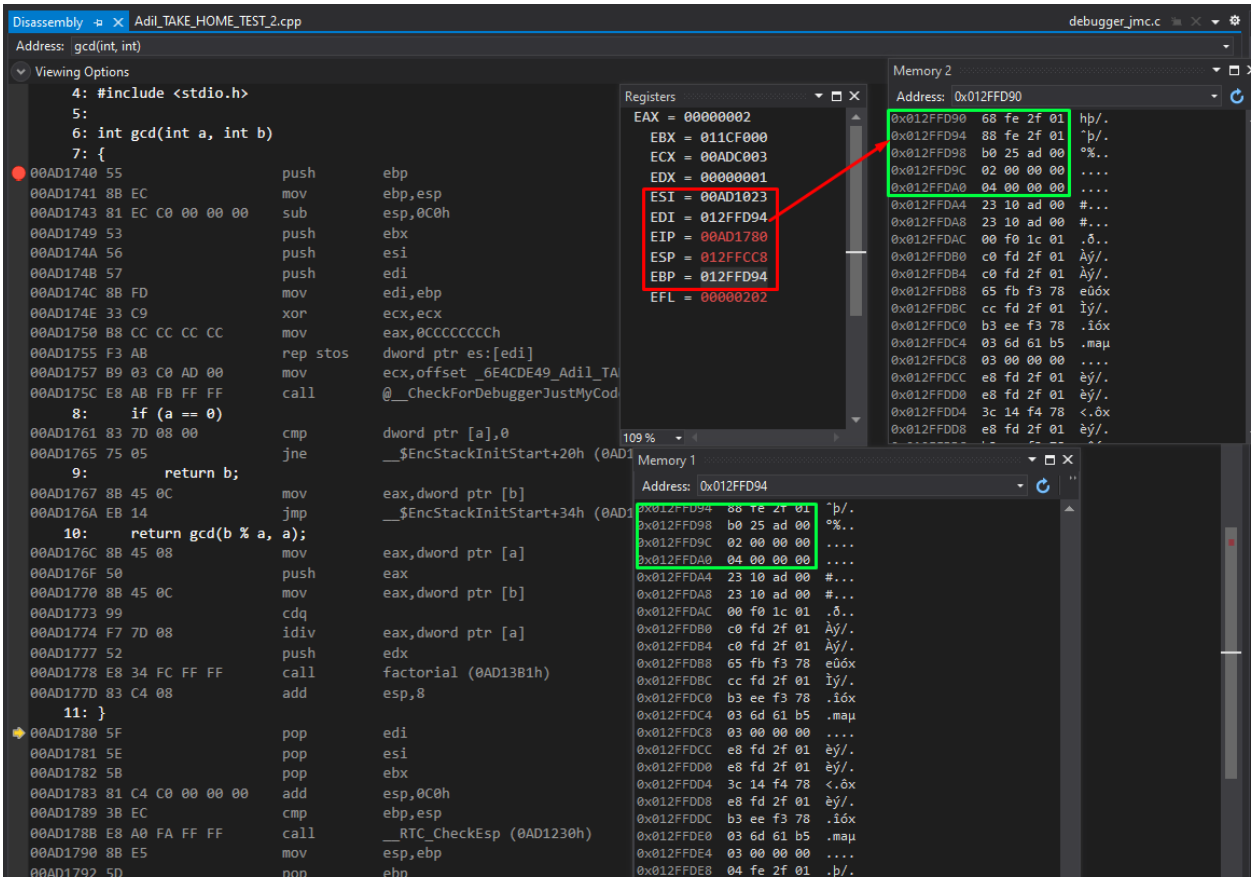


Figure 10: Program exiting GCD function

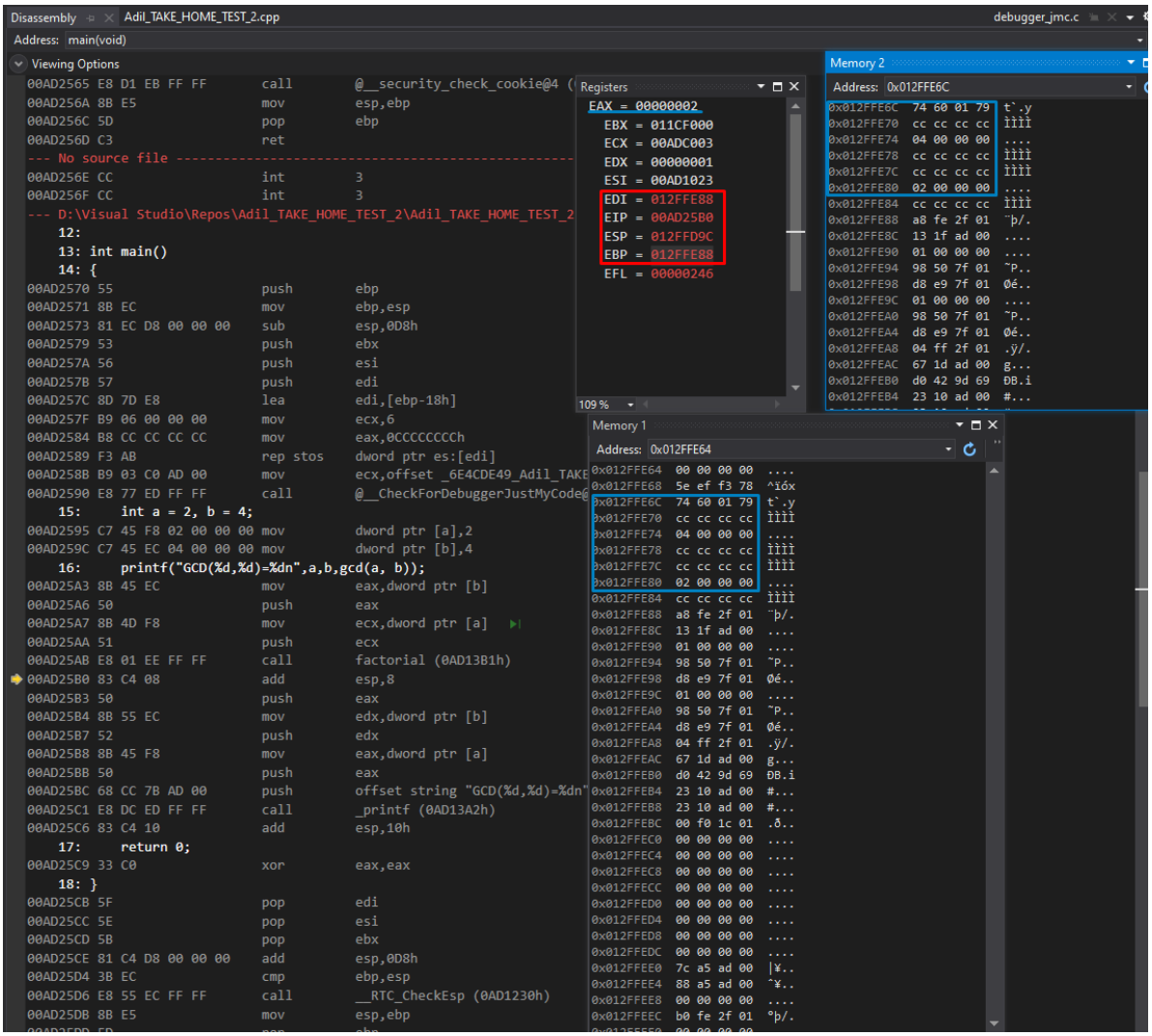


Figure 11: main function start

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

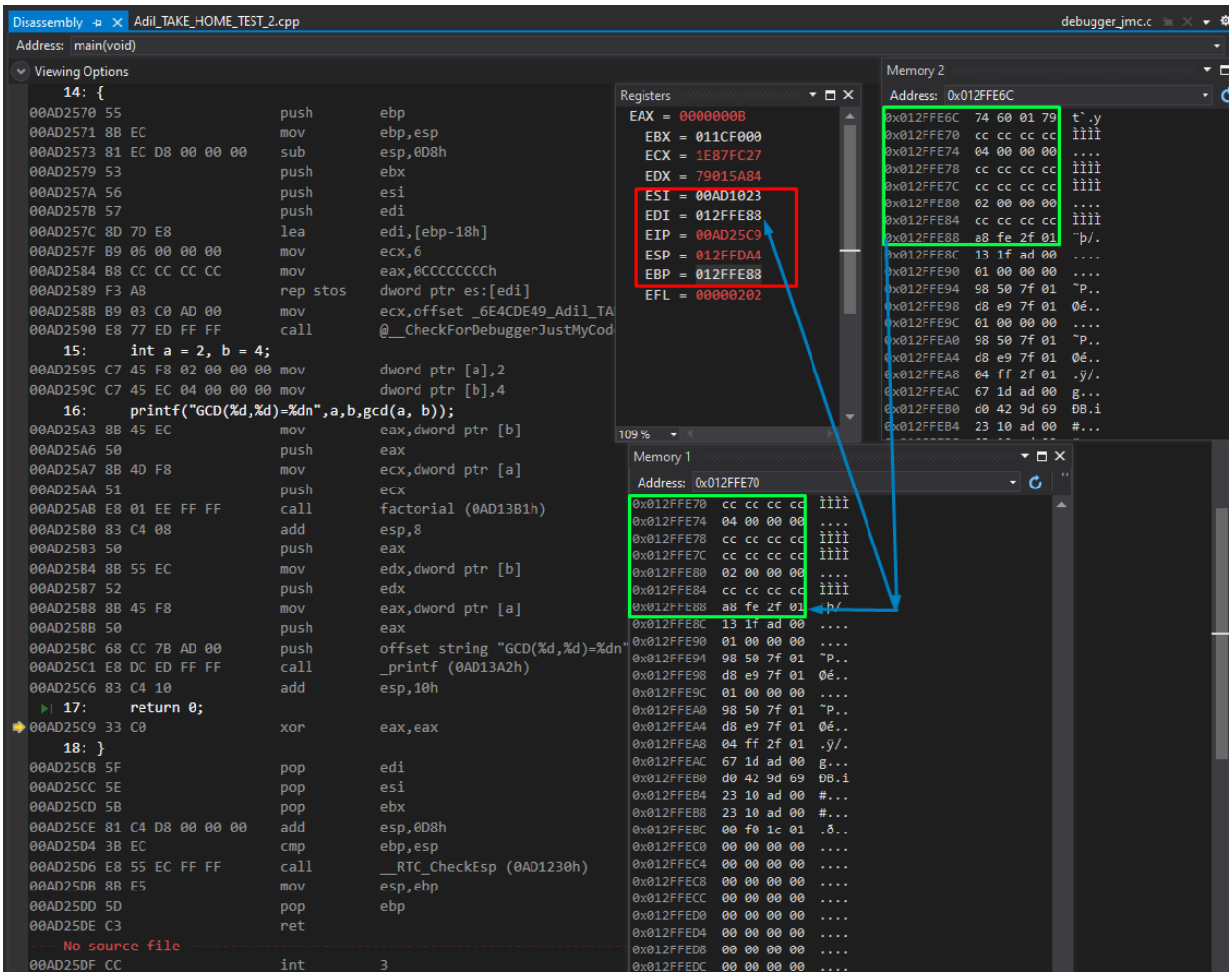


Figure 13: Main returning 0

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.

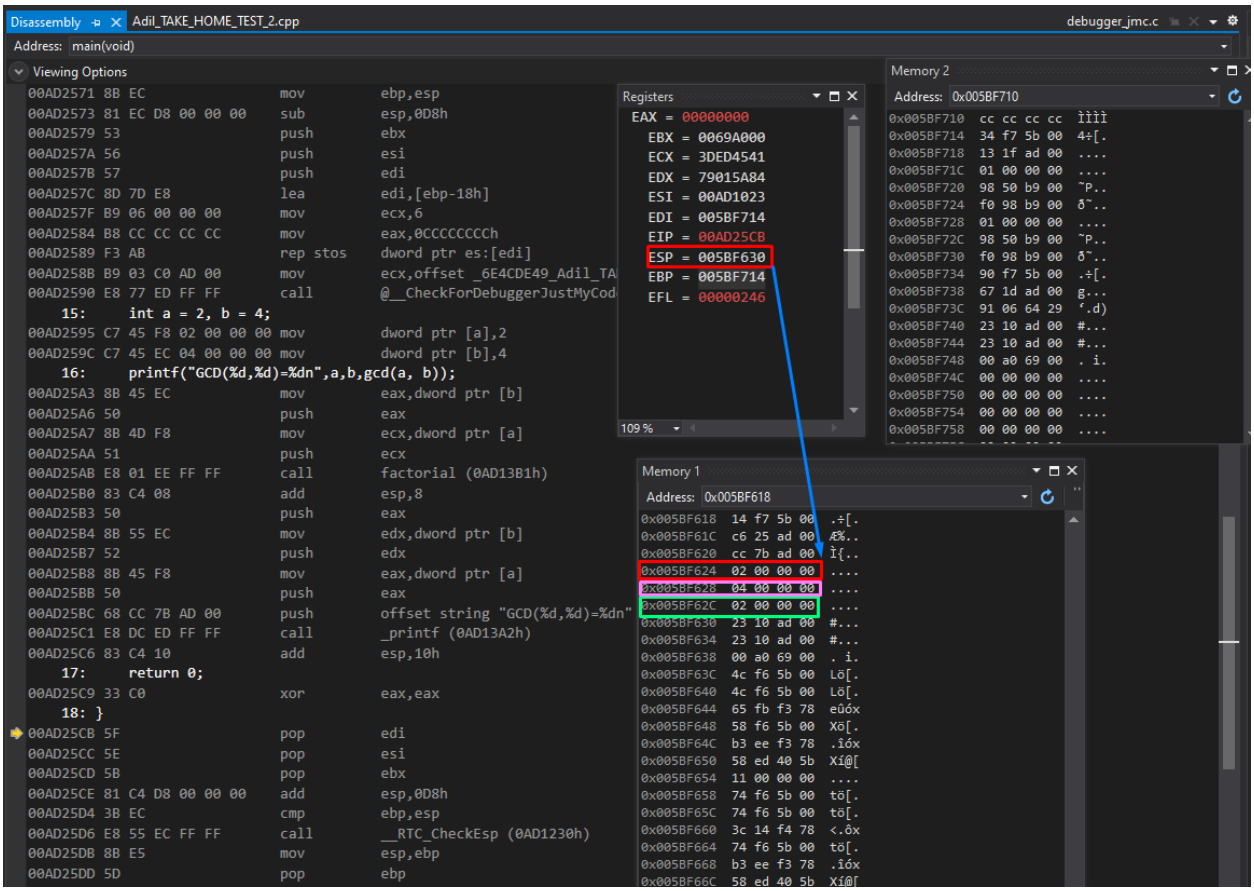


Figure 14 Stack Frame

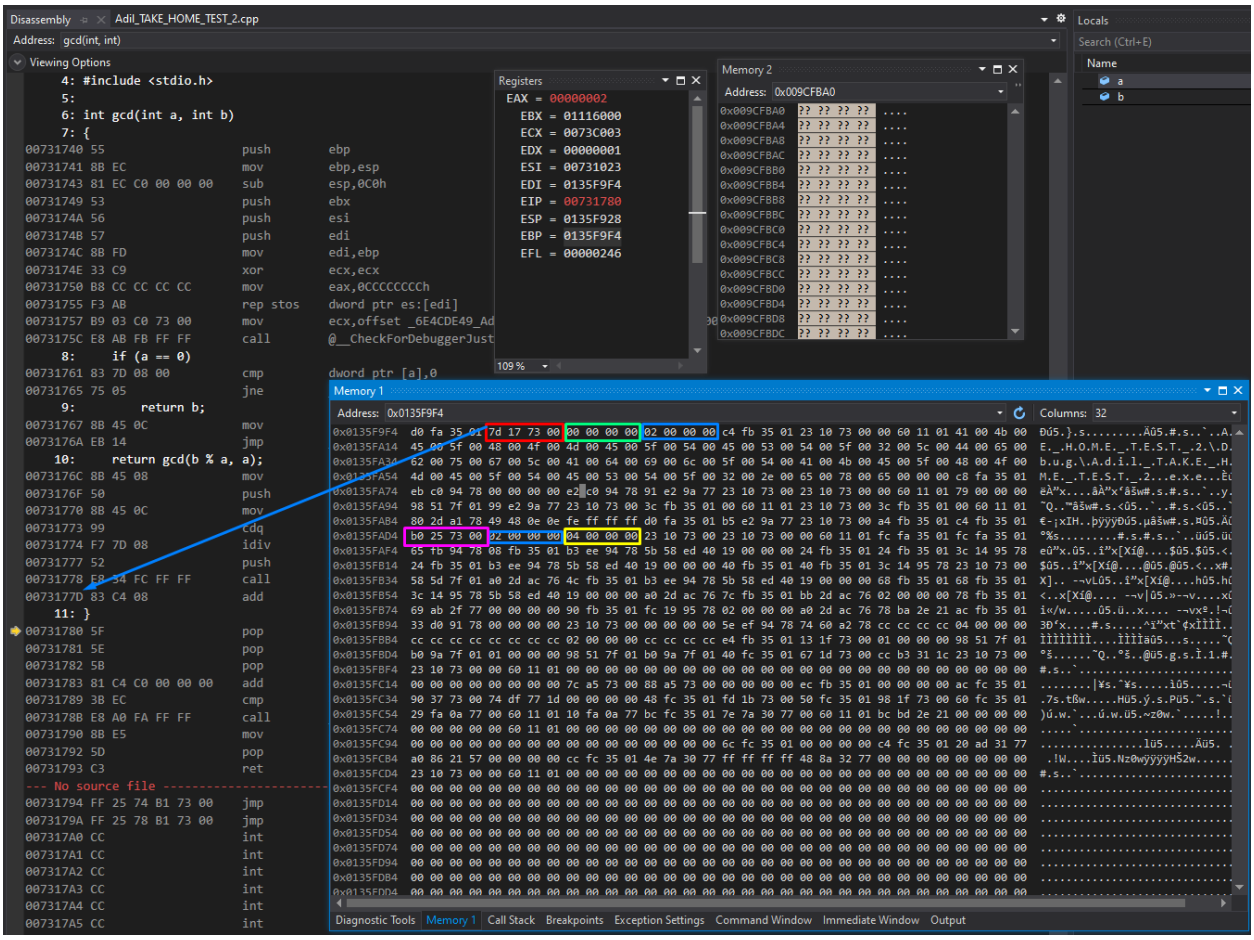


Figure 15: Stack frame

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_RECURSION.asm

```
.data
a: .word 2
b: .word 4
.text
main:
lw $a0,a  # load value a
lw $a1,b  #load value a
jal GCD # call funtion GCD

 add $a0,$v0,$zero
li $v0,1
syscall # print result
li $v0, 10 # exit program
syscall

GCD:
addi $sp, $sp, -12
sw $ra, 0($sp) # save function into stack
sw $s0, 4($sp) # save value $s0 into stack
sw $s1, 8($sp) # save value $s1 into stack

add $s0, $a0, $zero # s0 = a0 ( value a )
add $s1, $a1, $zero # s1 = a1 ( value b )

addi $t1, $zero, 0 # $t1 = 0
beq $s1, $t1, EXIT # if s1 == 0 EXIT

add $a0, $zero, $s1 # make a0 = $s1
div $s0, $s1 # a/b
mfhi $a1 # reminder of 2/4 which is equal to a%b
jal GCD

EXITGCD:
lw $ra, 0 ($sp)  # read registers from stack
lw $s0, 4 ($sp)
lw $s1, 8 ($sp)
addi $sp,$sp , 12 # bring back stack pointer
jr $ra
EXIT:
add $v0, $zero, $s0 # return $v0 = $s0 (2)
j EXITGCD
```

Figure 16: Adil_GCD_RECURSION.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 EUCLEDEAN 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.

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

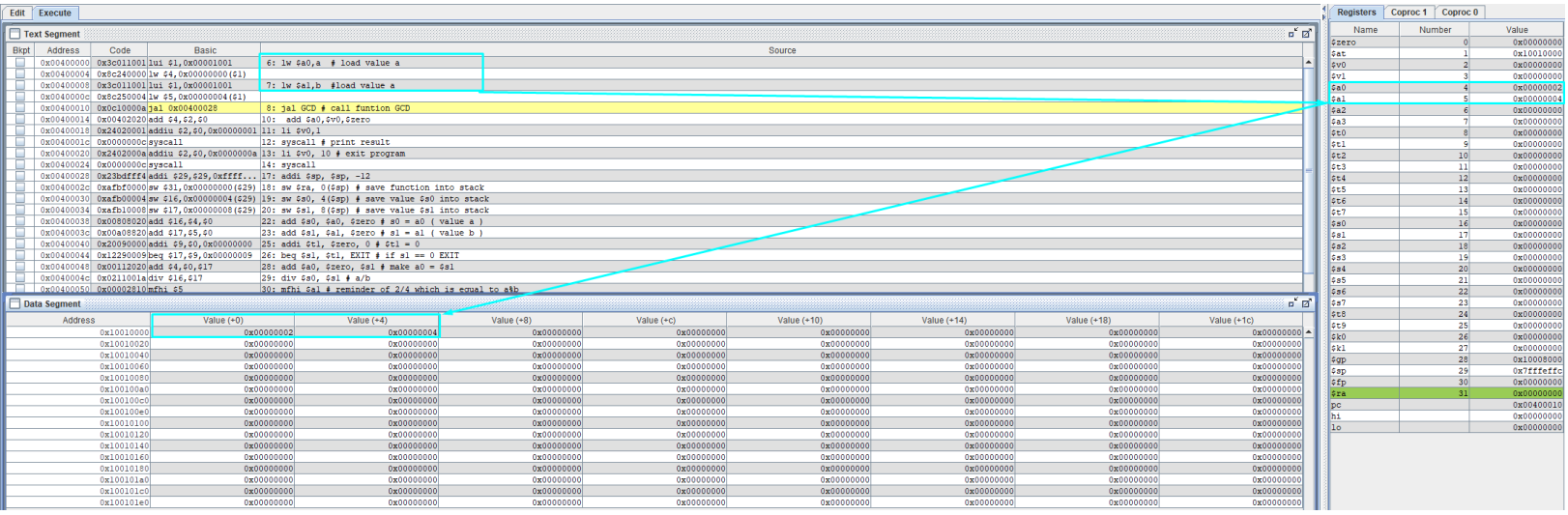


Figure 17: Variables a and b loaded into the registers

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.

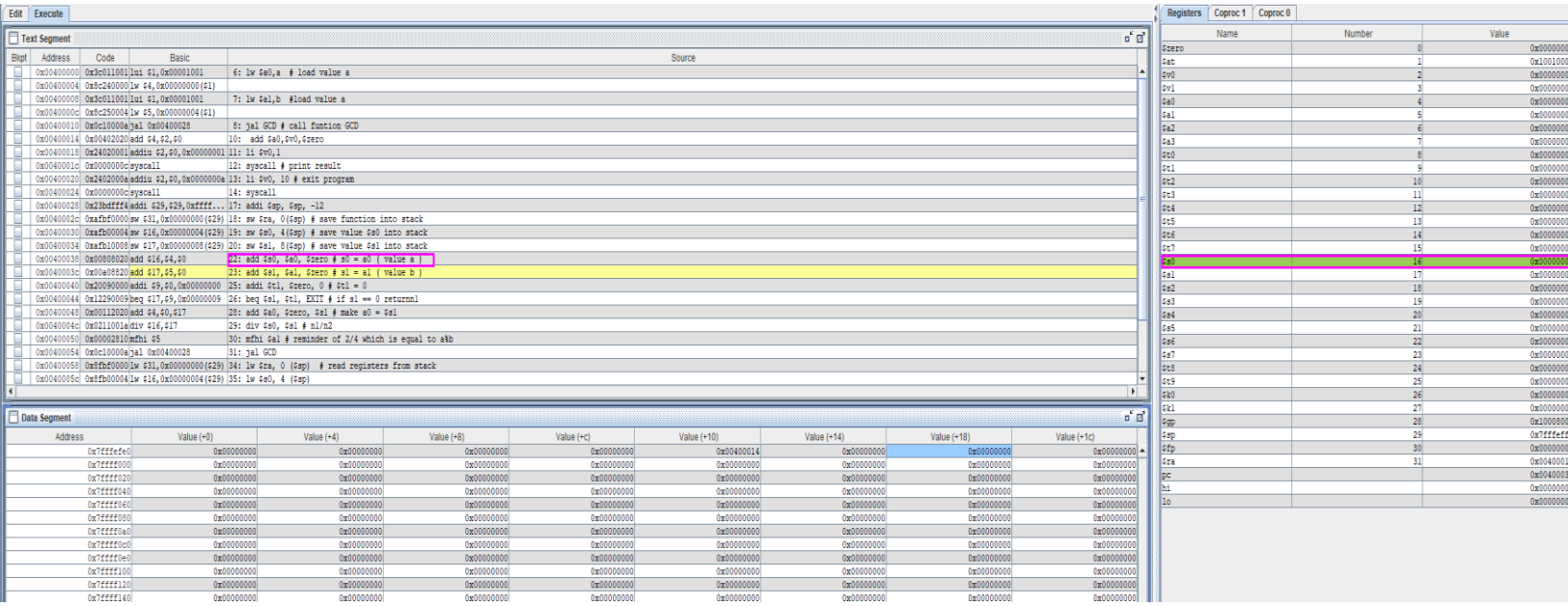


Figure 18: Variable loaded on stack

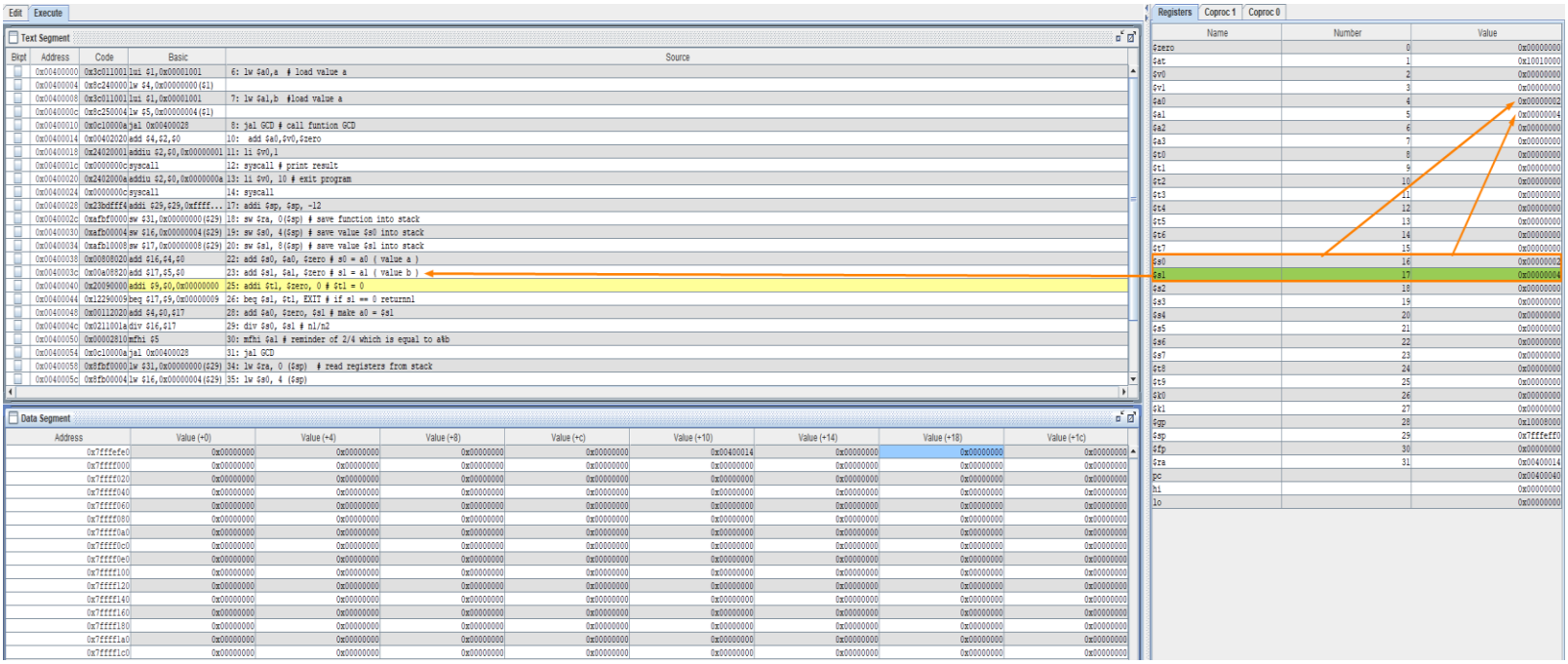


Figure 19: variable b on stack

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 0x00400040. Currently the program is within the GCD function computing the greatest common divider between the two variables.

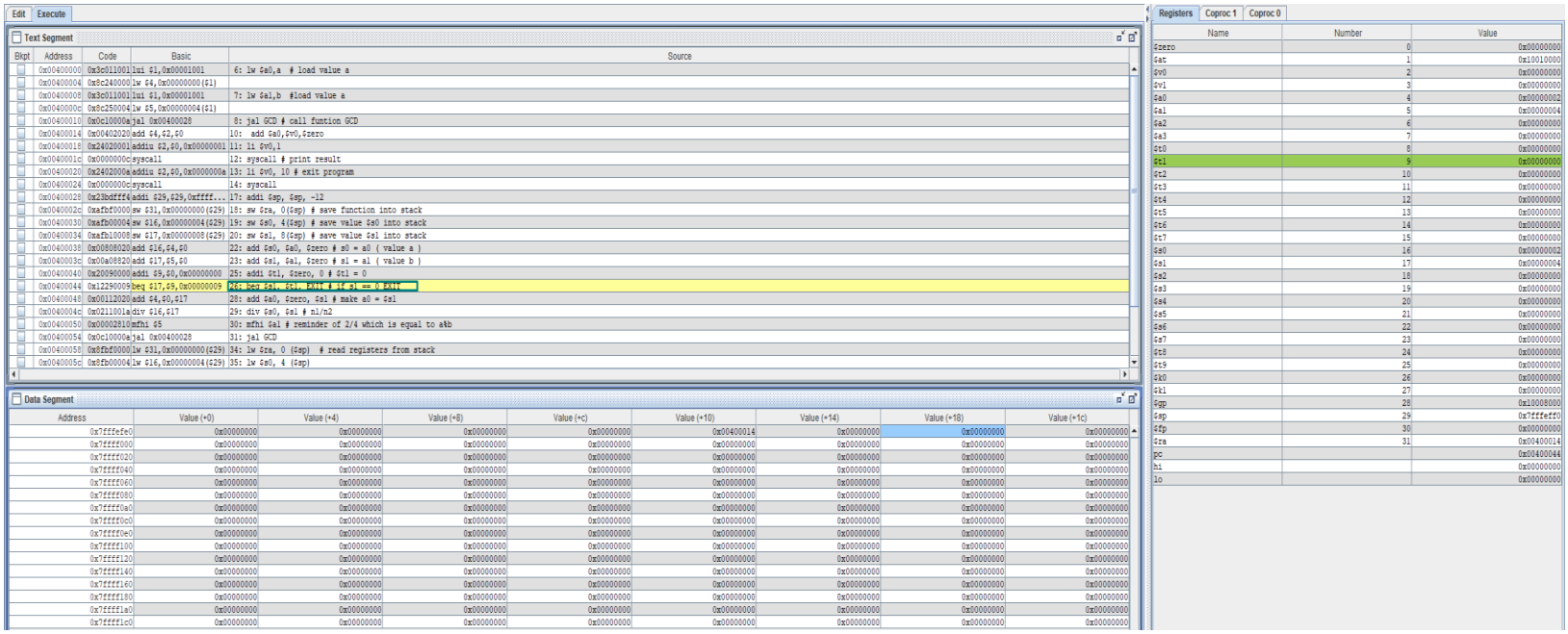


Figure 20: checking if a is currently equal to 0.

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.

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

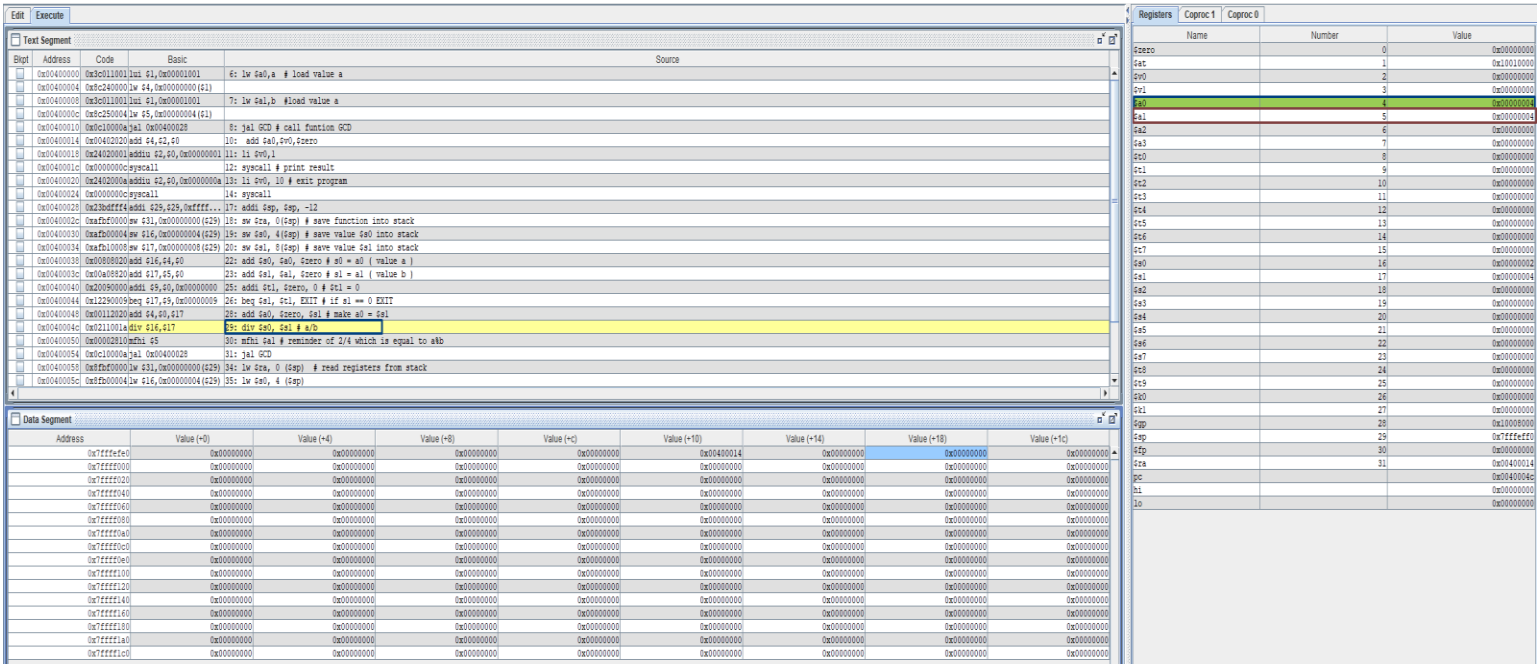


Figure 22: Division statement

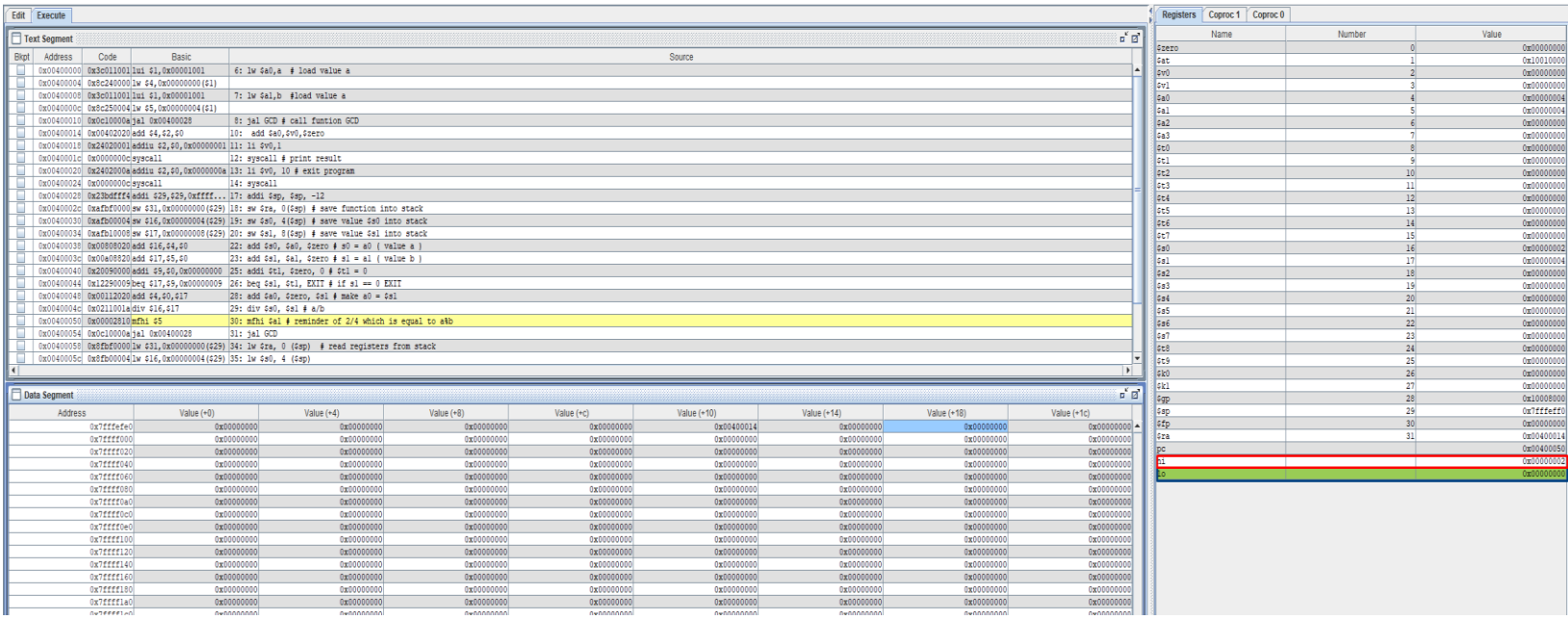


Figure 21:module function

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.

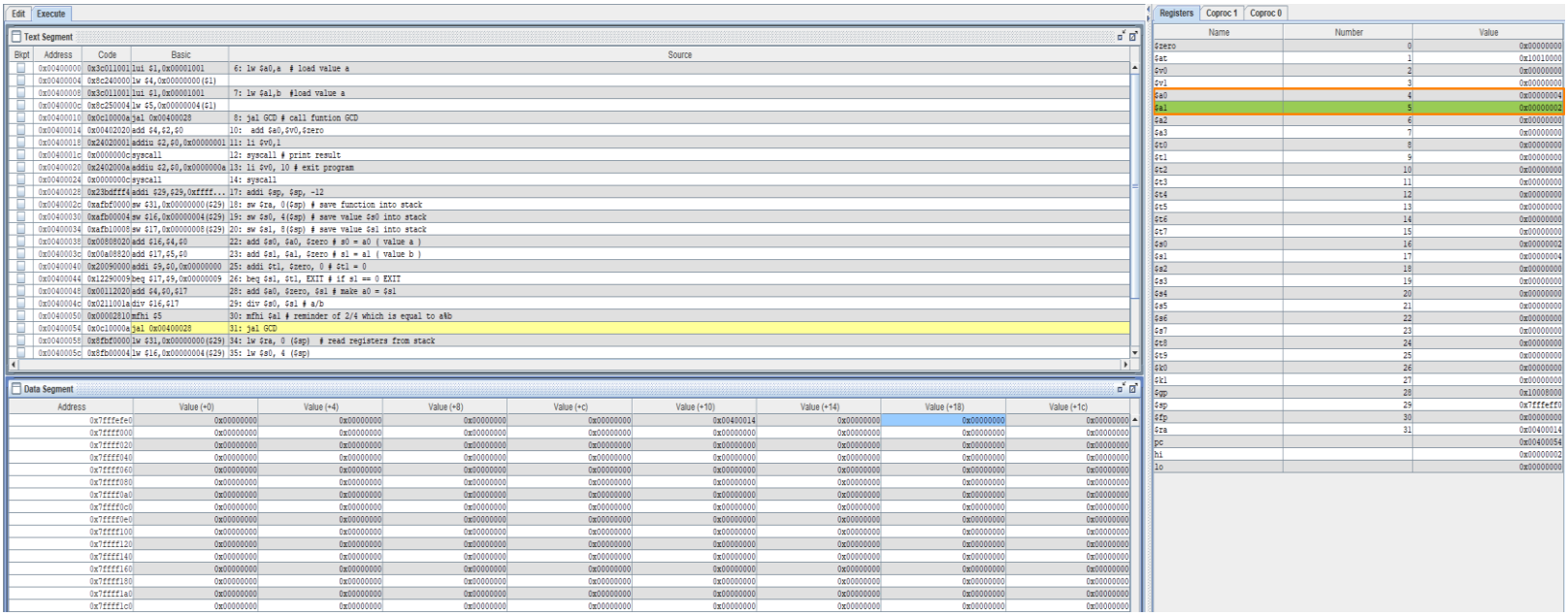


Figure 23: GCD executing again

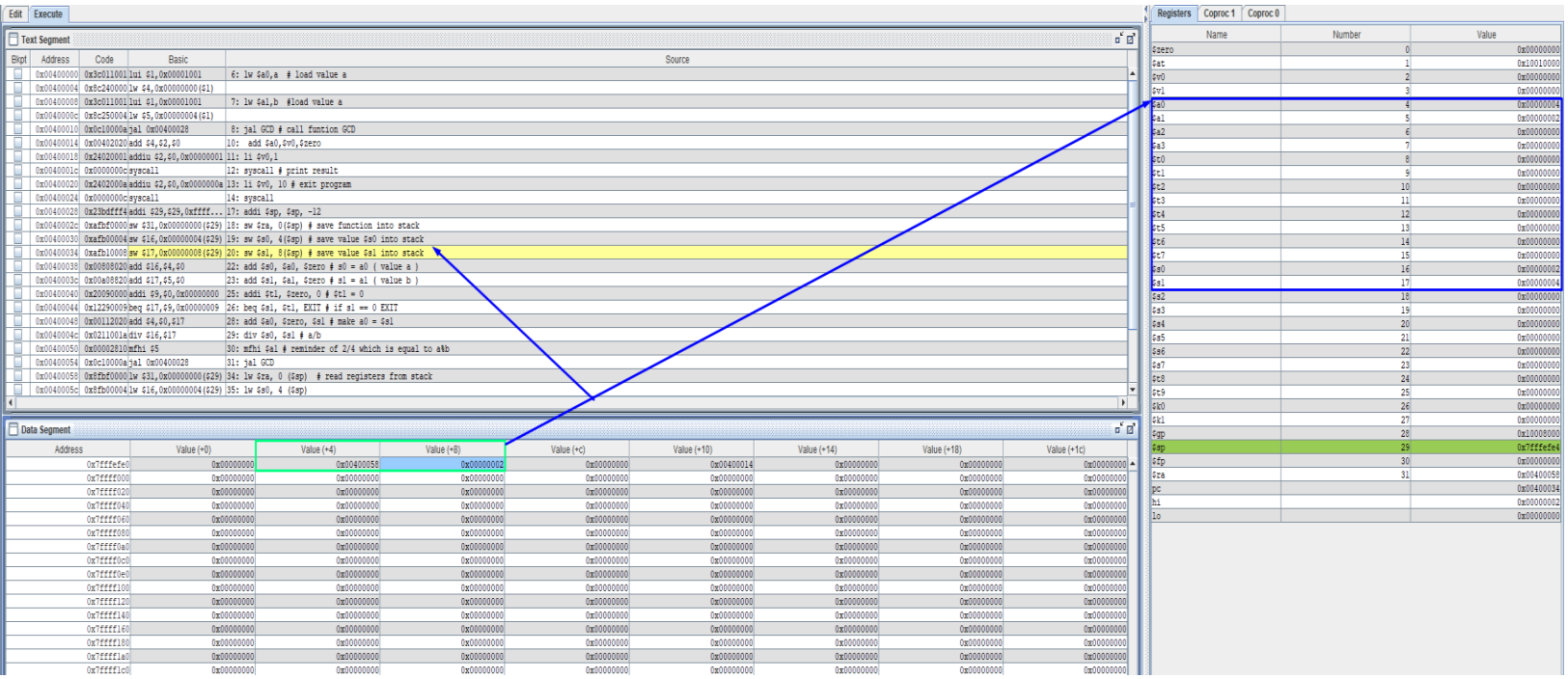


Figure 24: Placed onto the stack

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 0x7fffefe0 + 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 0x00400058.

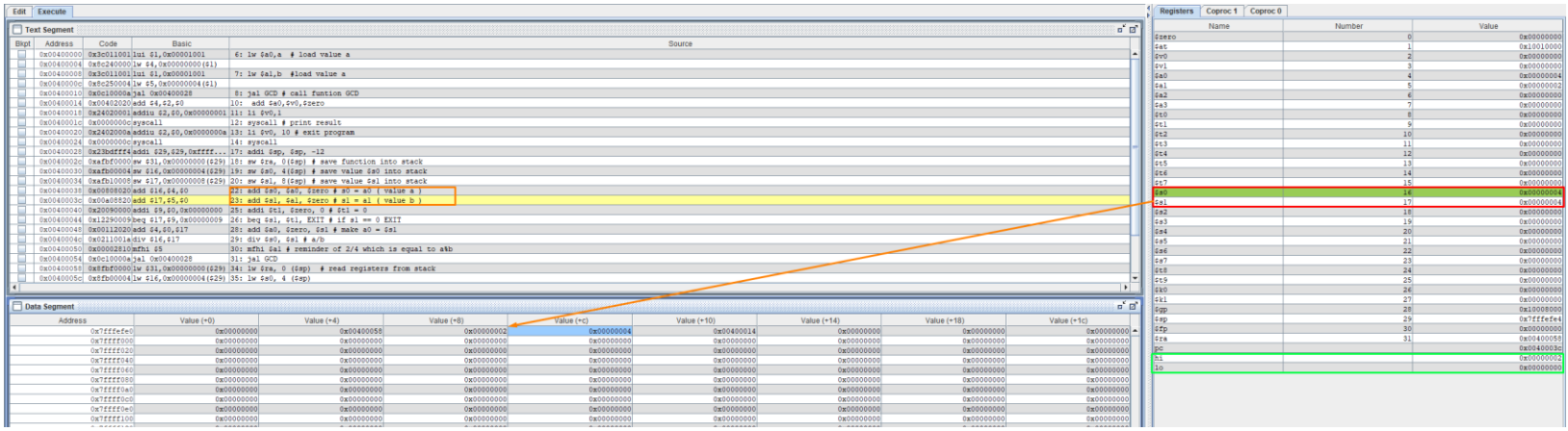


Figure 25: 4 being copied into $a0 and $a1

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

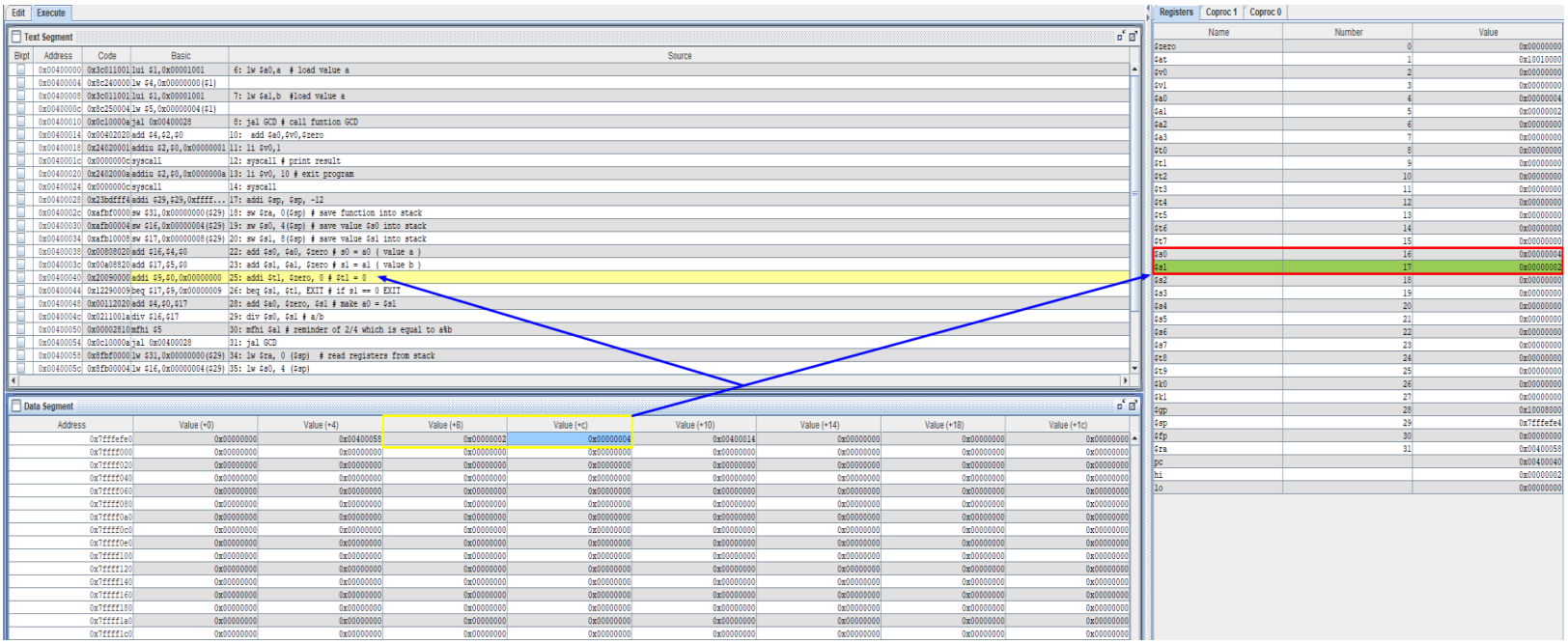


Figure 26: GCD between 0 and 2

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.

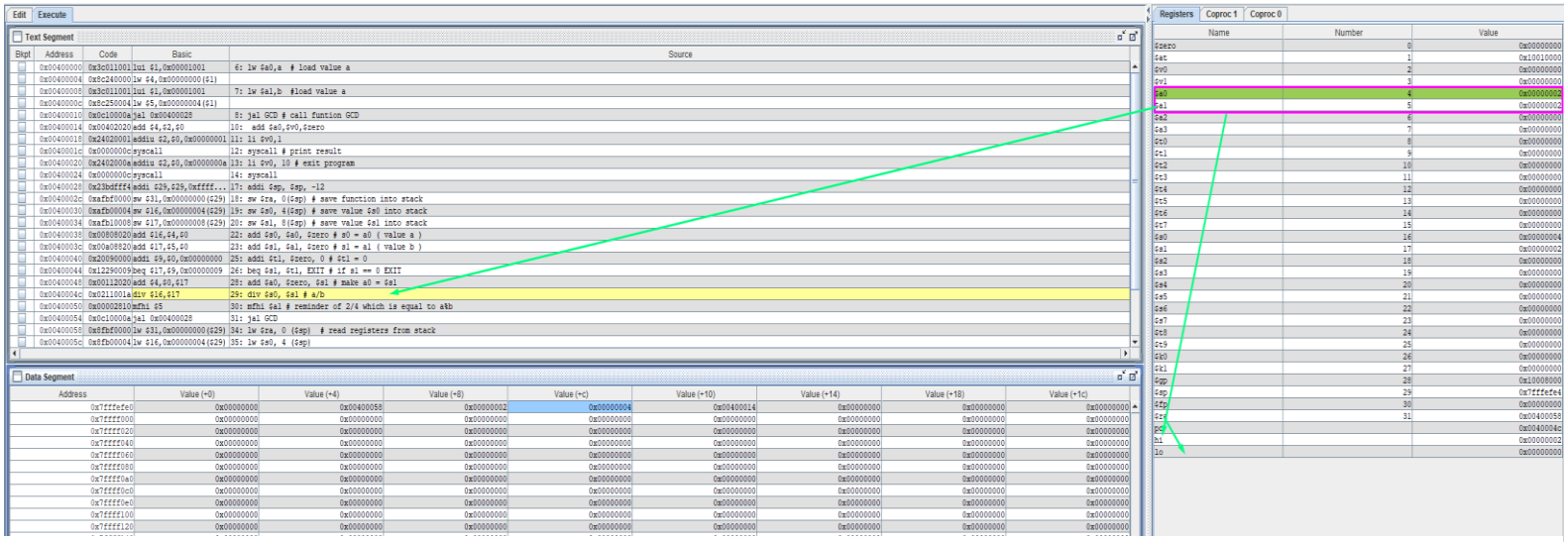


Figure 27: Division between 0 and 2

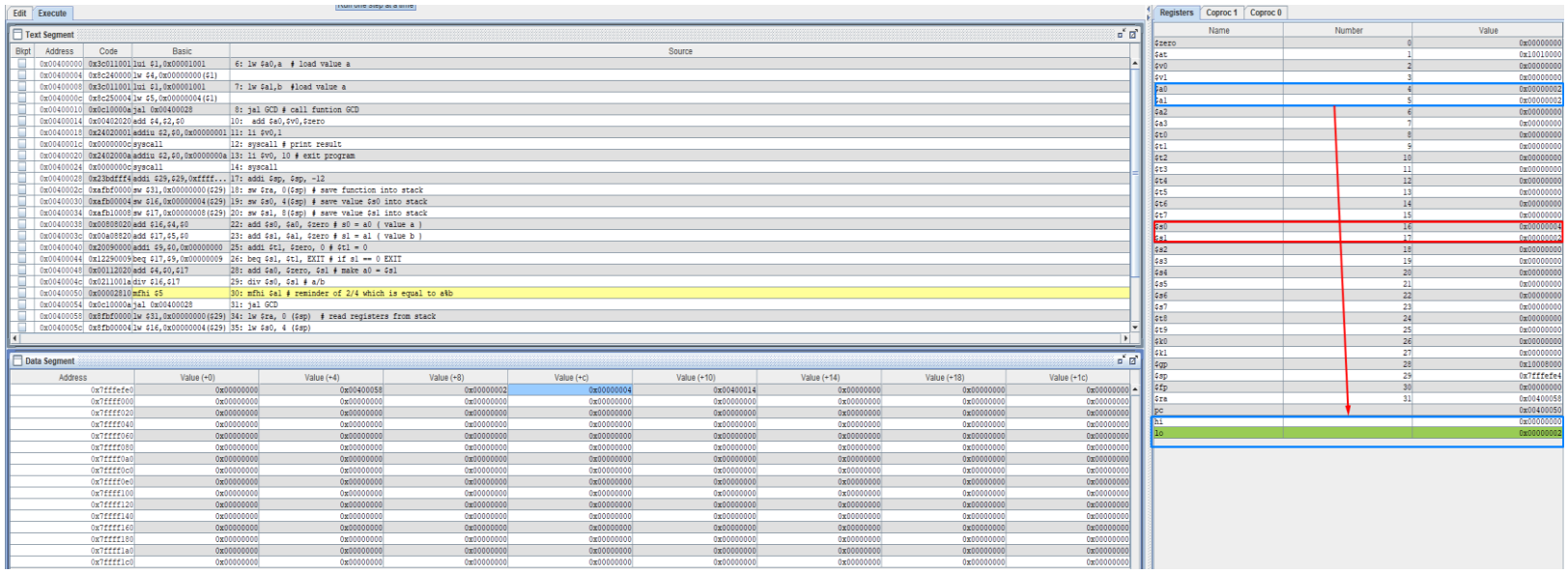


Figure 28: mod occurring with register LO

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.

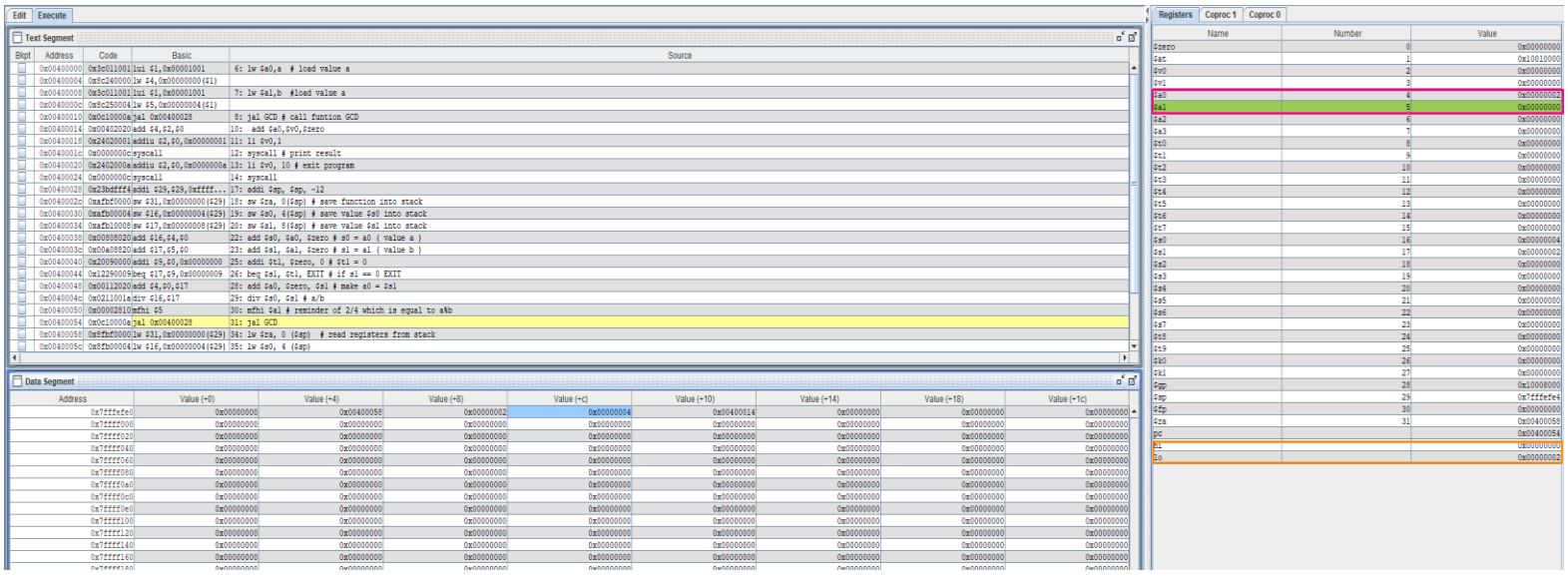


Figure 29: GCD function executed

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

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

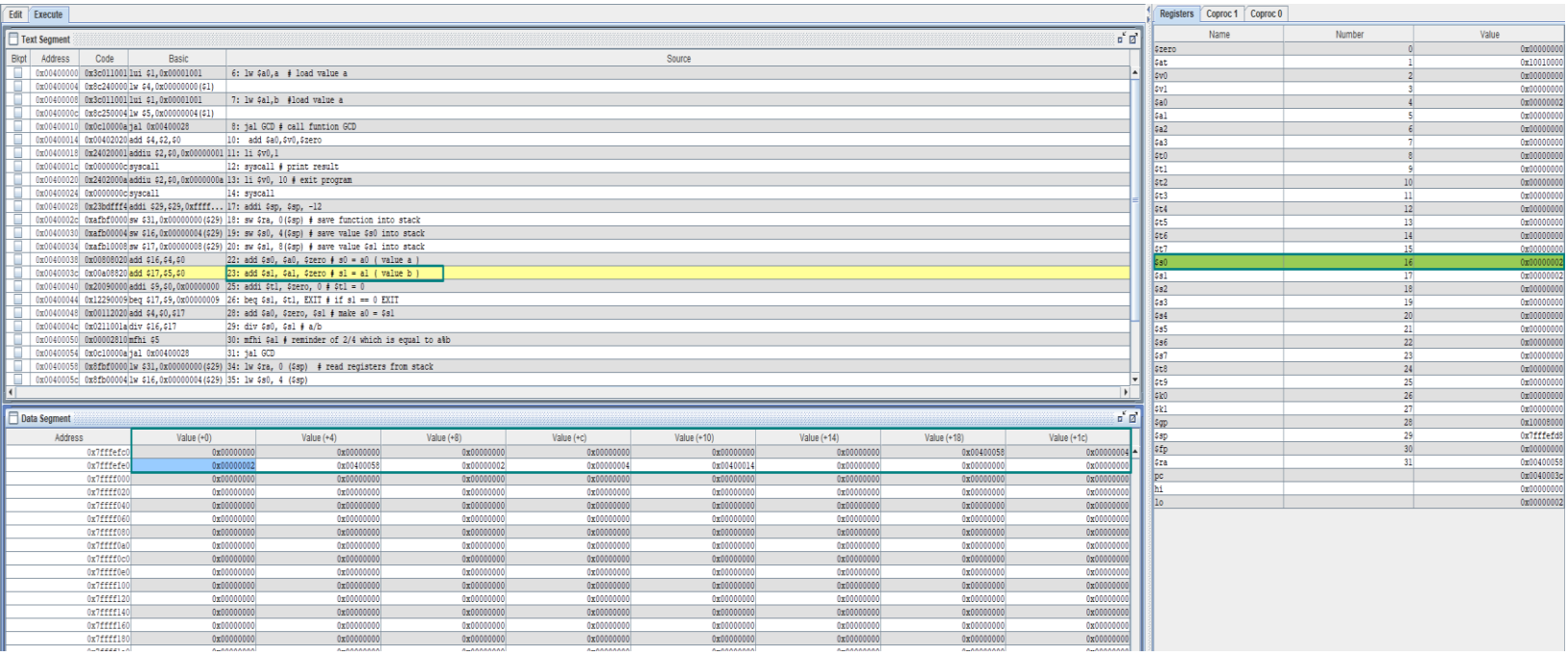


Figure 30: stack frame

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.

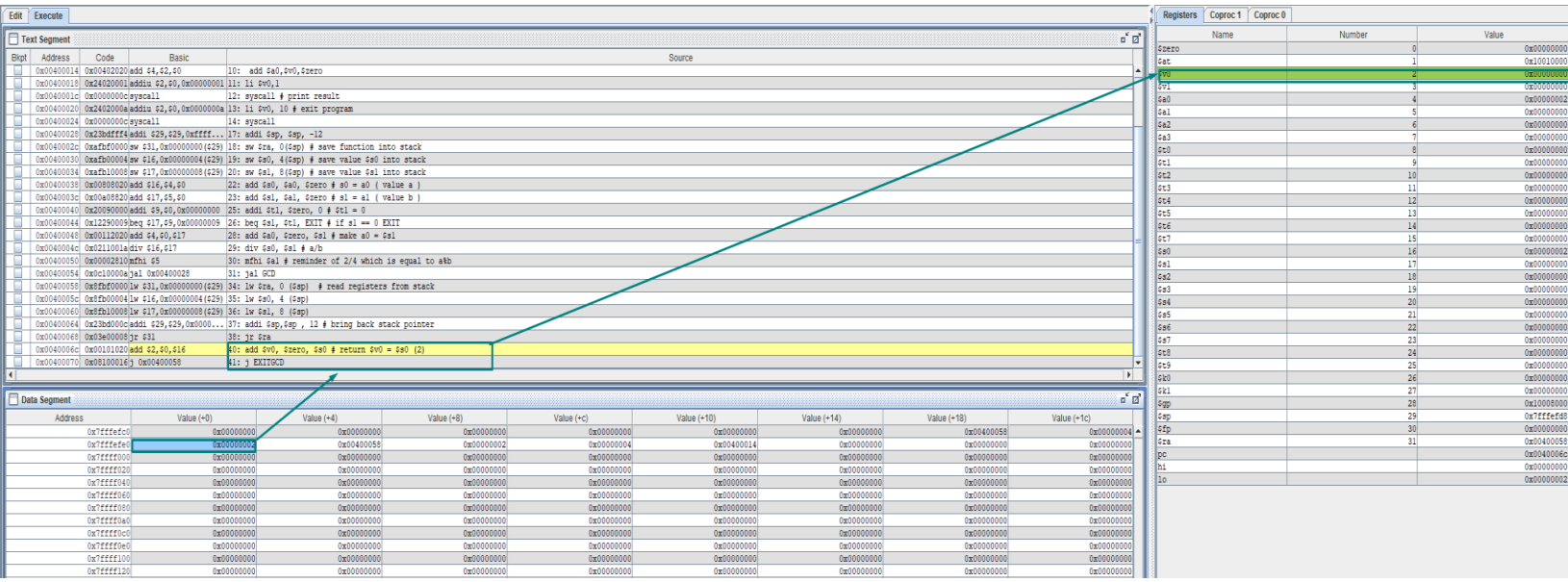


Figure 31: Clearing the registers

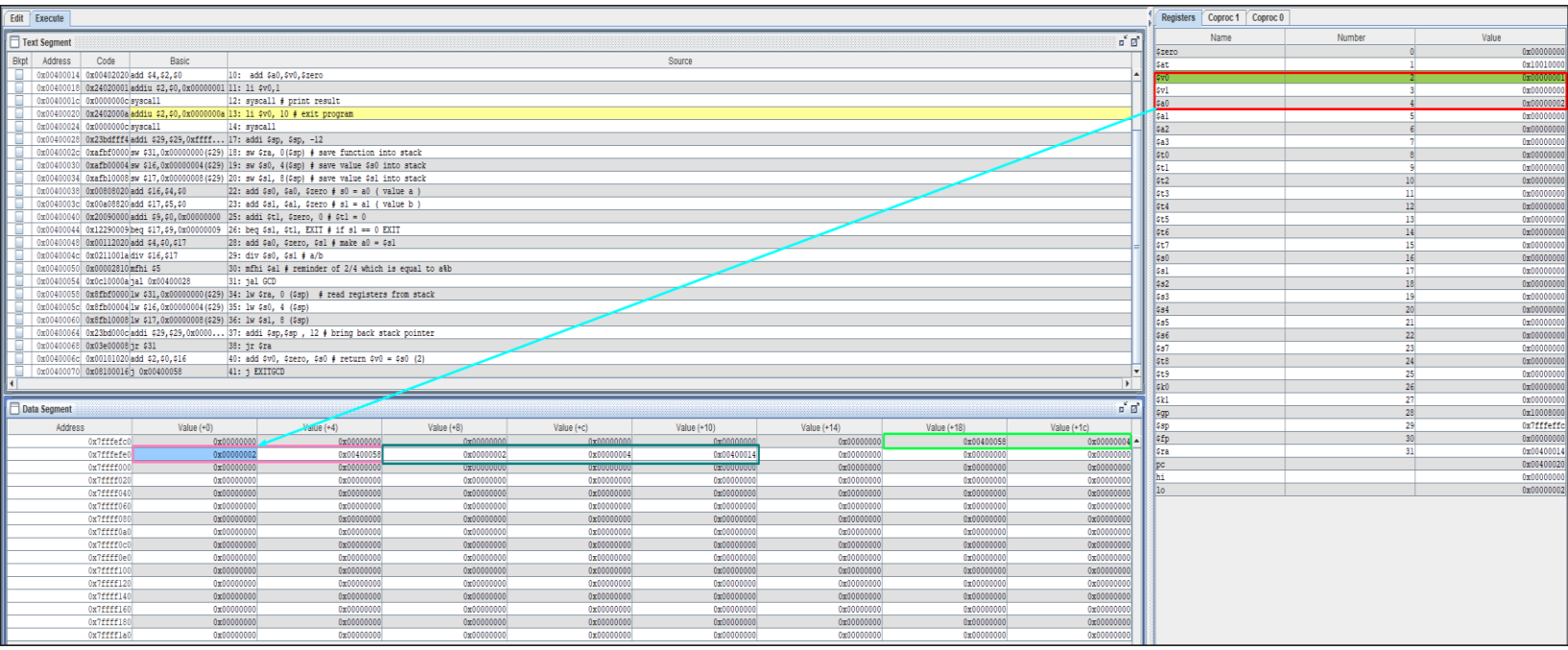


Figure 32: End of the program

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

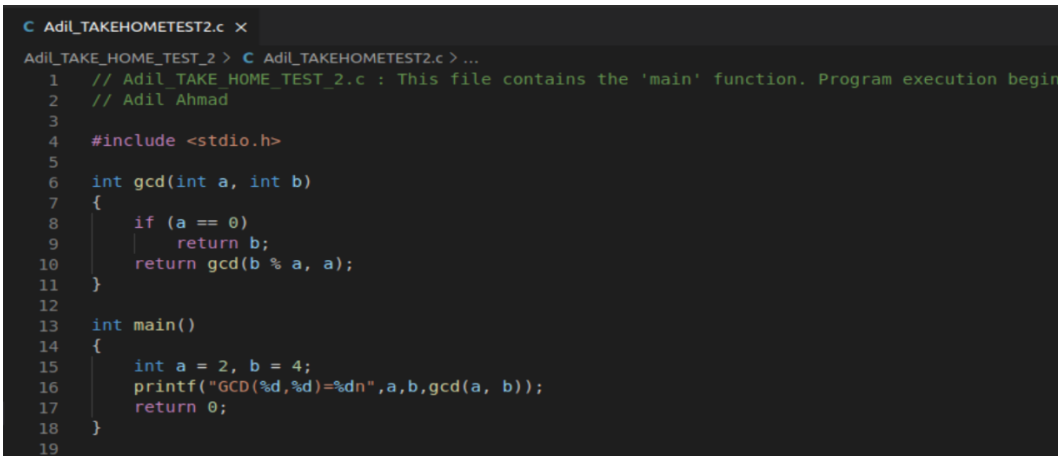


Figure 33: Adil_TAKEHOMETEST2.c

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.

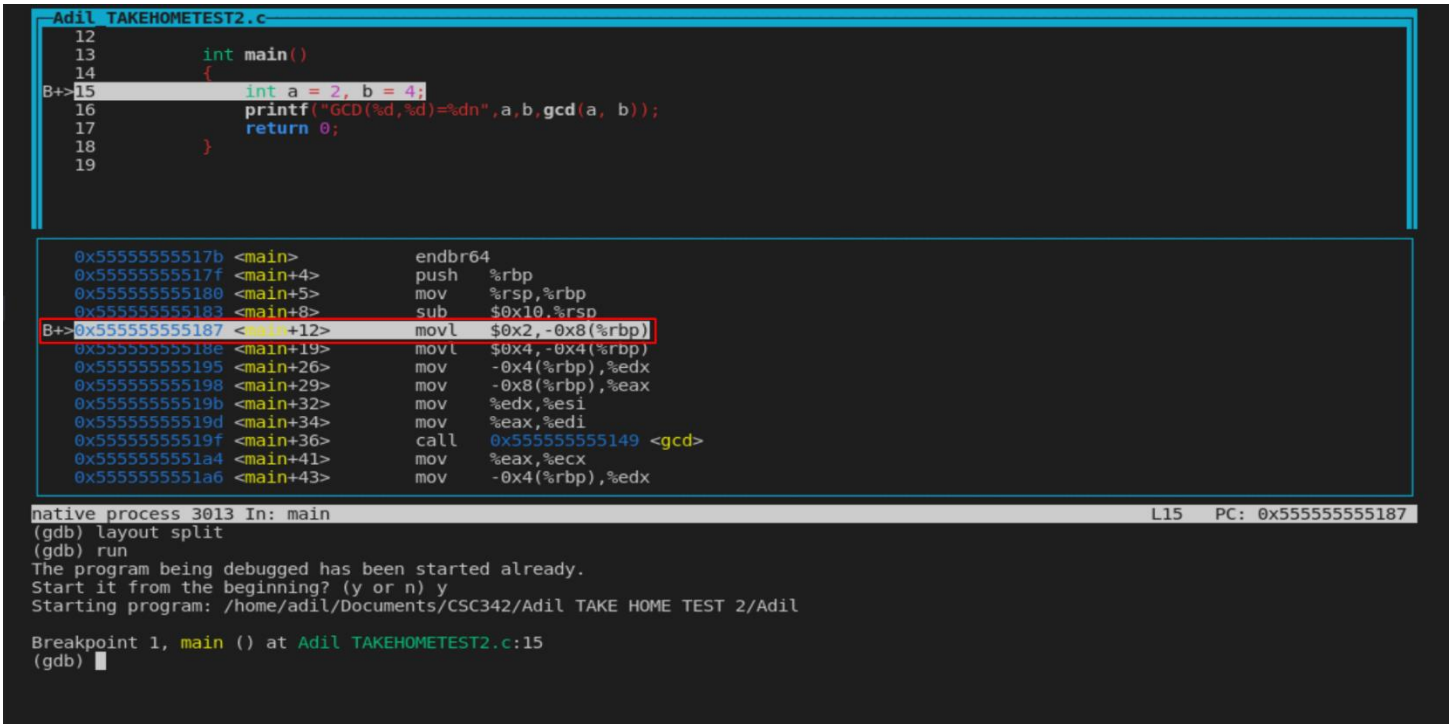


Figure 34: Breaking main

This here is the start of the program. First, we place a break point in the beginning of main. This 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 of offset. For the variable a, which holds 2, its is 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.

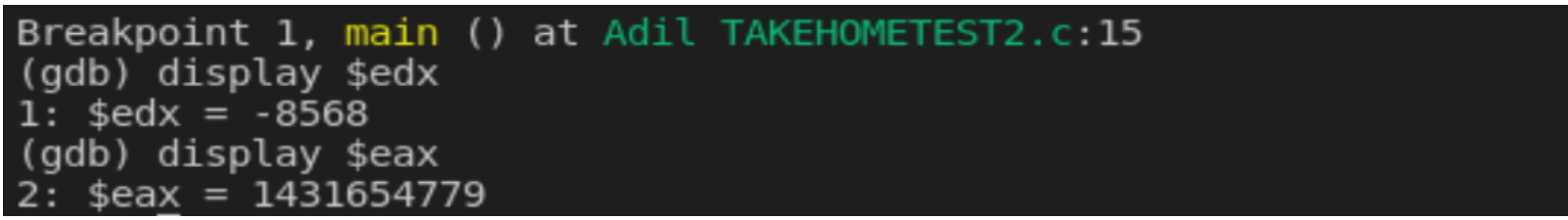


Figure 35: This shows the initial value of both edx and eax

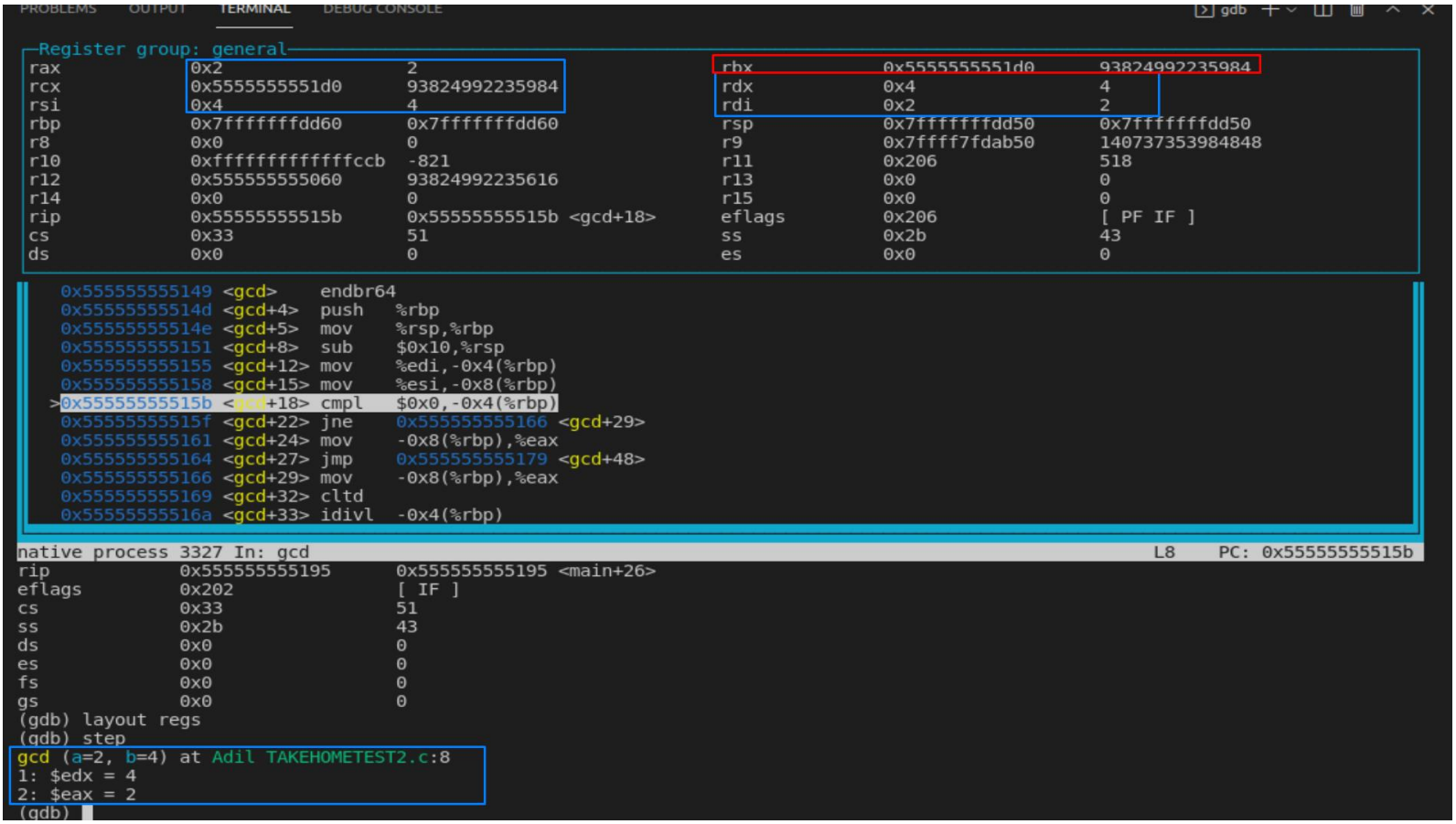


Figure 36: Windows showing 2 and 4

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.

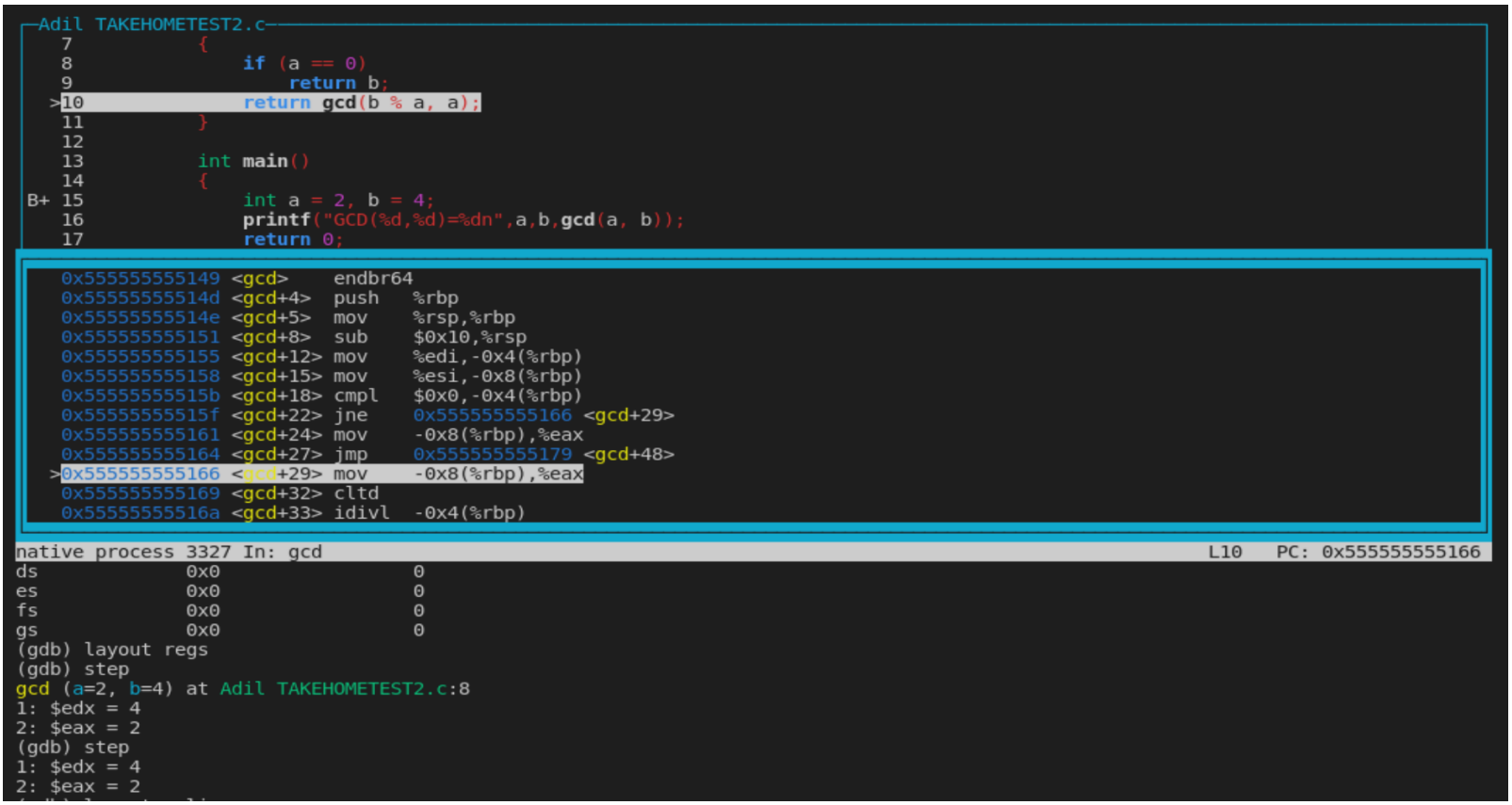


Figure 38: Returning value

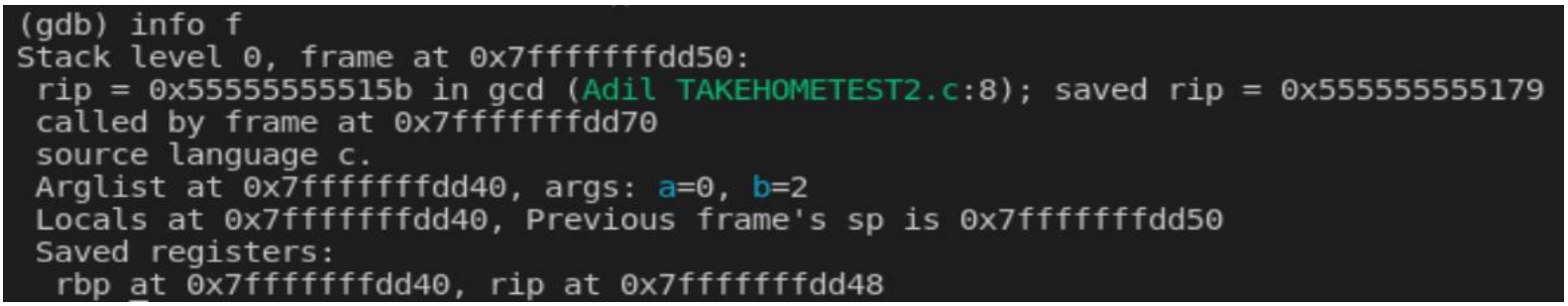


Figure 37: Stack frame

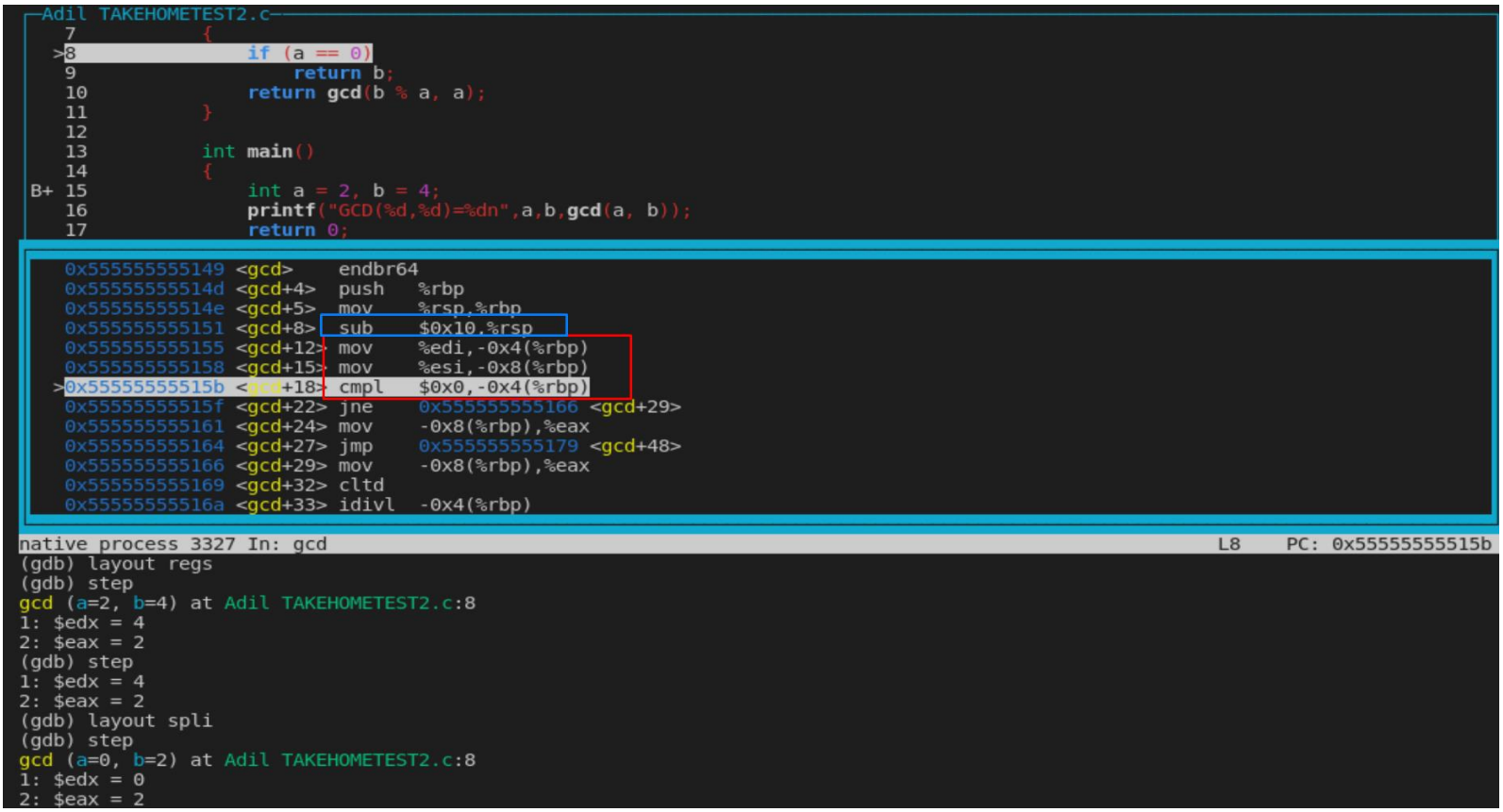


Figure 39: conditional statement

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           {
B+ 15               int a = 2, b = 4;
   16               printf("GCD(%d,%d)=%dn",a,b,gcd(a, b));
   17               return 0;

   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)
   0x55555555516d <gcd+36> mov    -0x4(%rbp),%eax
   0x555555555170 <gcd+39> mov    %eax,%esi
native process 3327 In: gcd                                   L9    PC: 0x555555555161
(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
(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.

```
  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)
  0x55555555516d <gcd+36> mov    -0x4(%rbp),%eax
  0x555555555170 <gcd+39> mov    %eax,%esi
```

Figure 40: jmp function to be executed

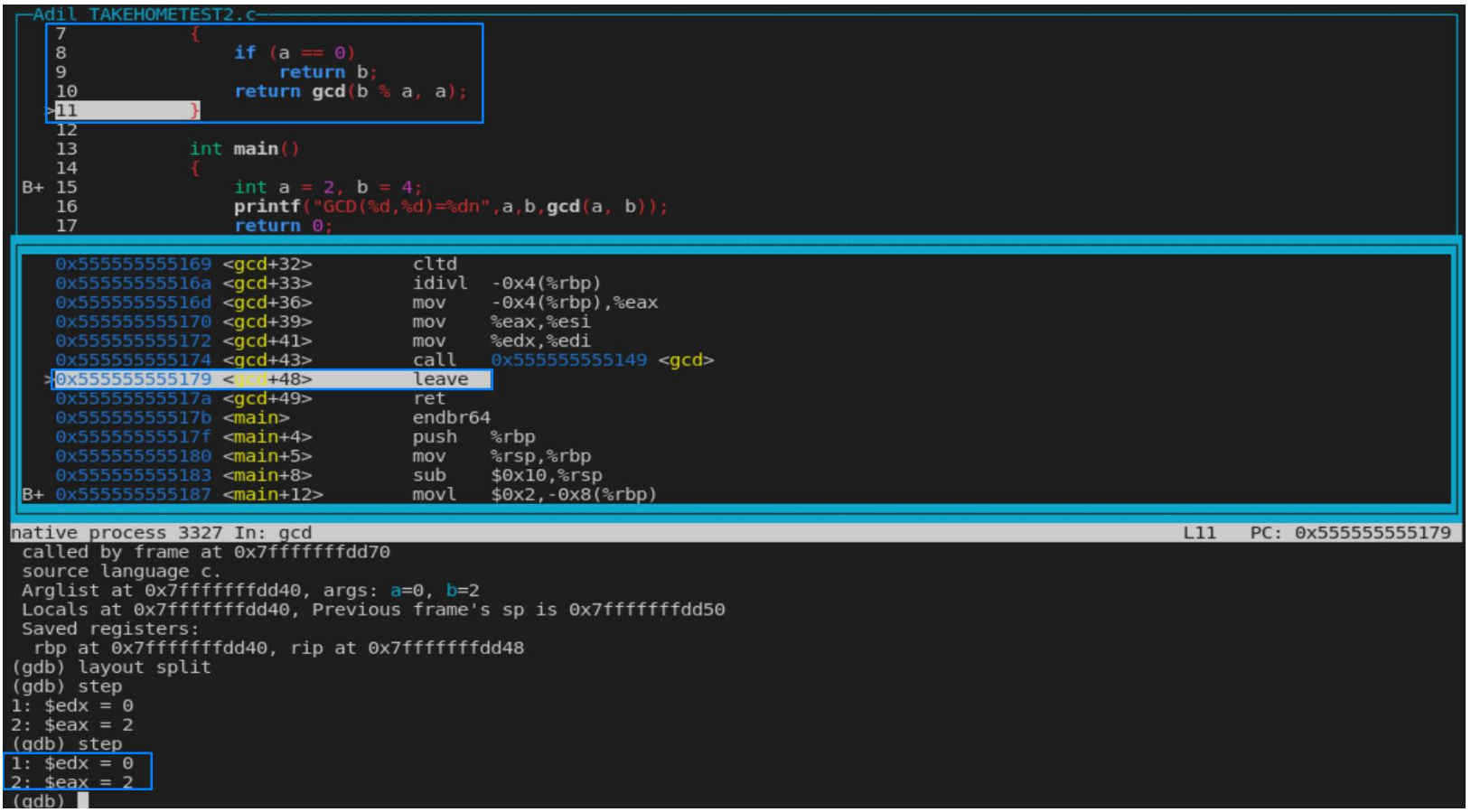
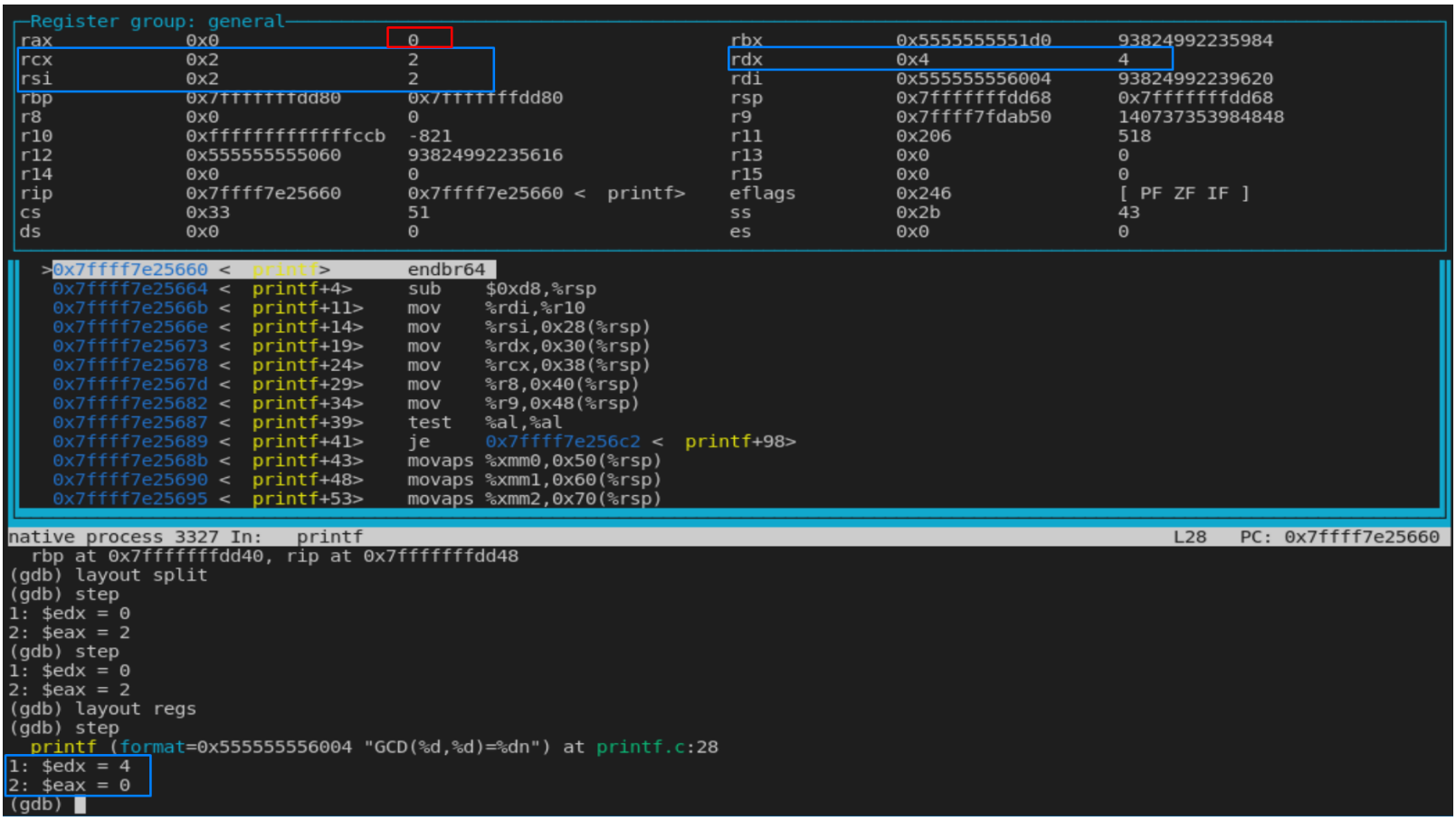


Figure 42: exiting the GCD function

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 0x555555555174, is the recursion within this entire program. We see that the function calls the memory location of 0x555555555149, which is the beginning of the GCD function. The leave function at memory location 0x555555555179 will make the program enter the main function.



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.

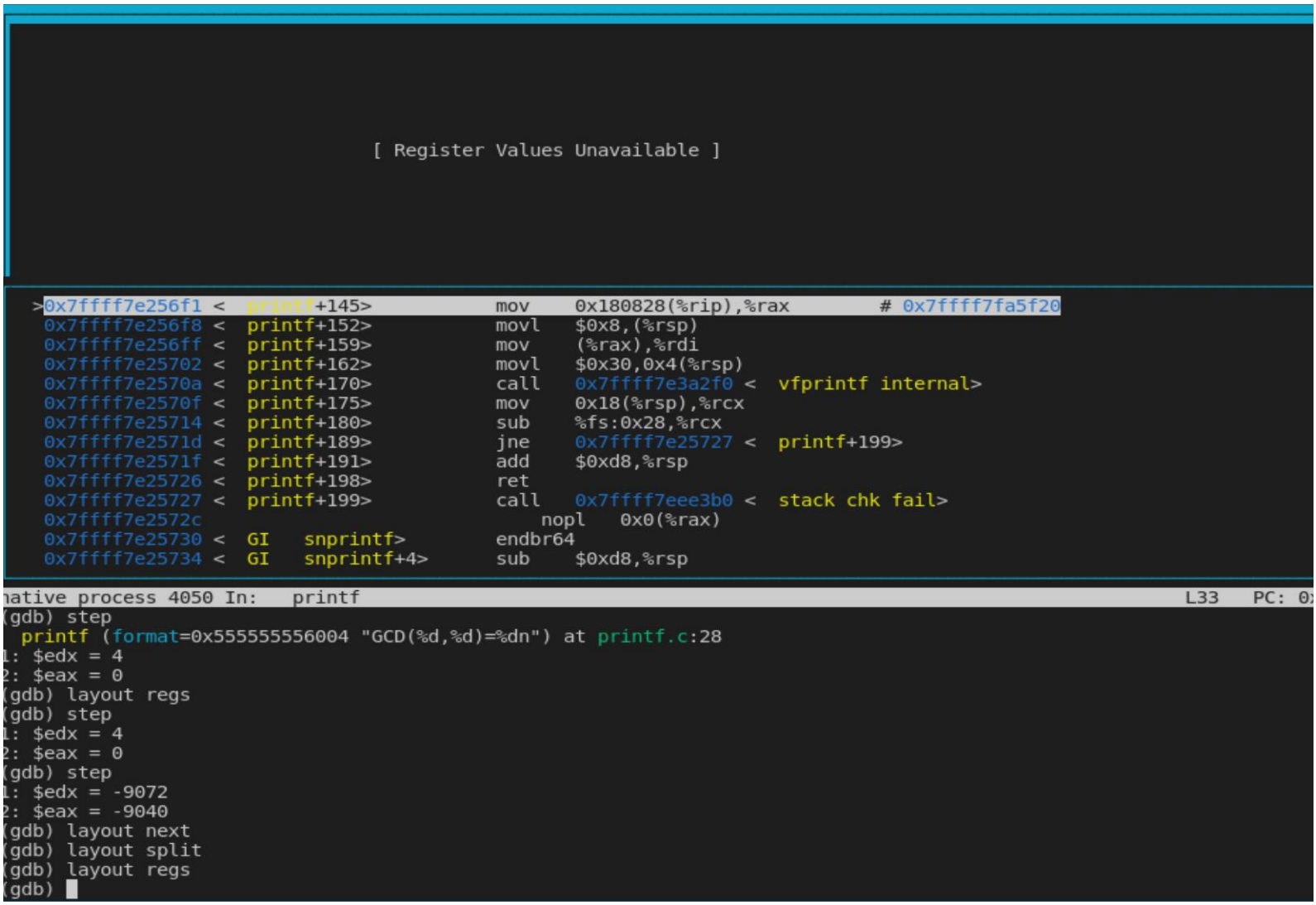


Figure 43: Print function

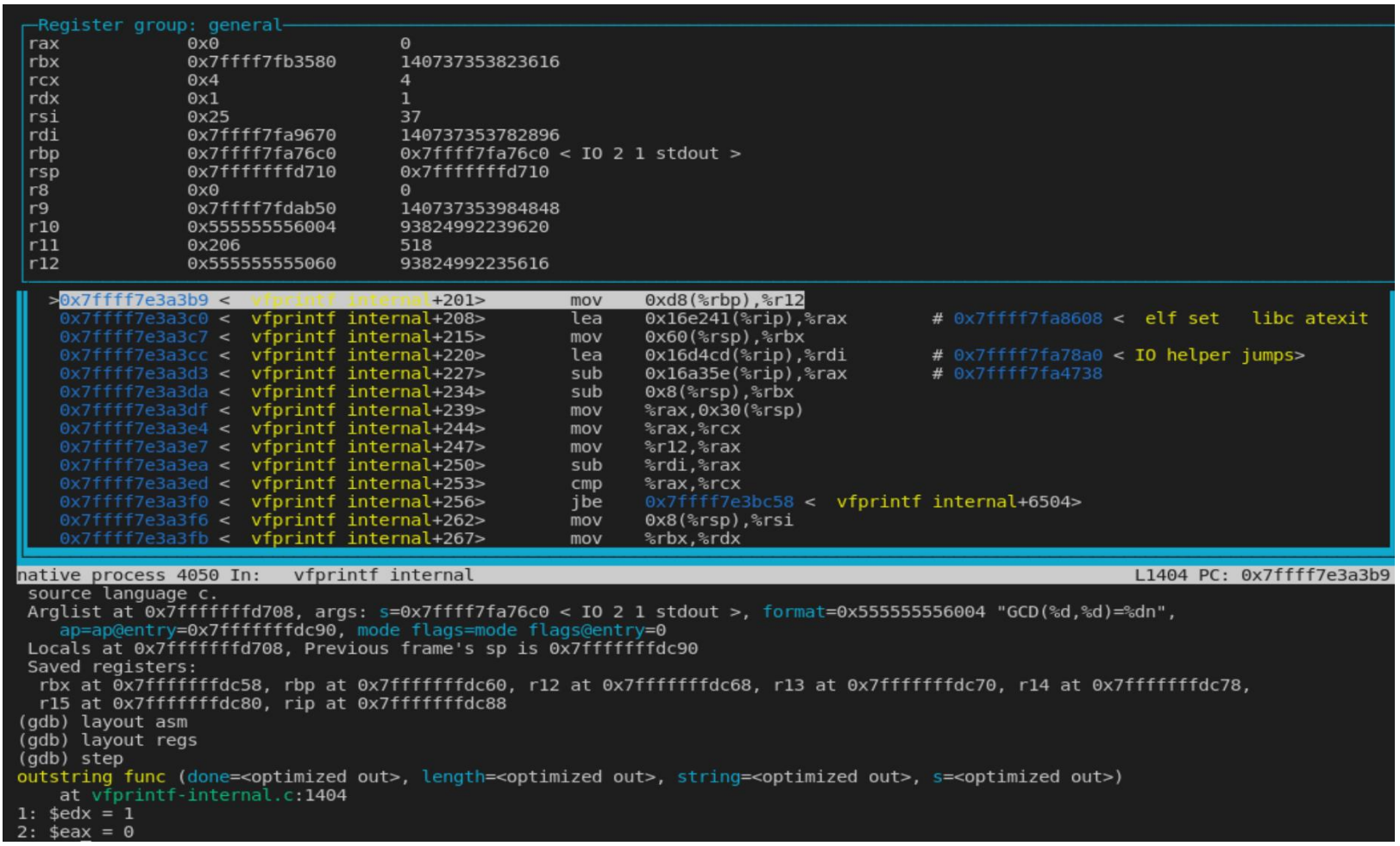


Figure 45: Values being reset to 0

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

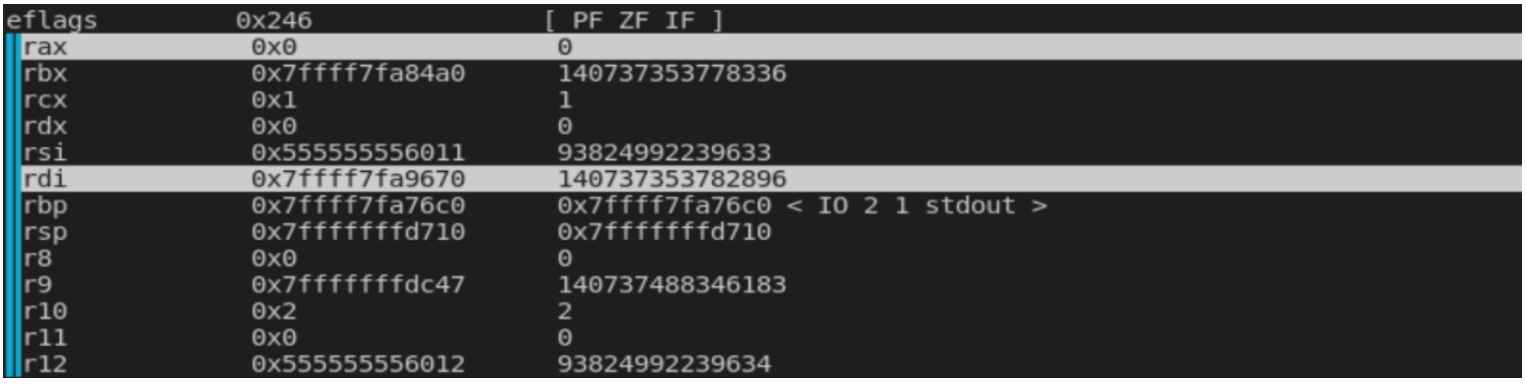


Figure 44: Registers holding value

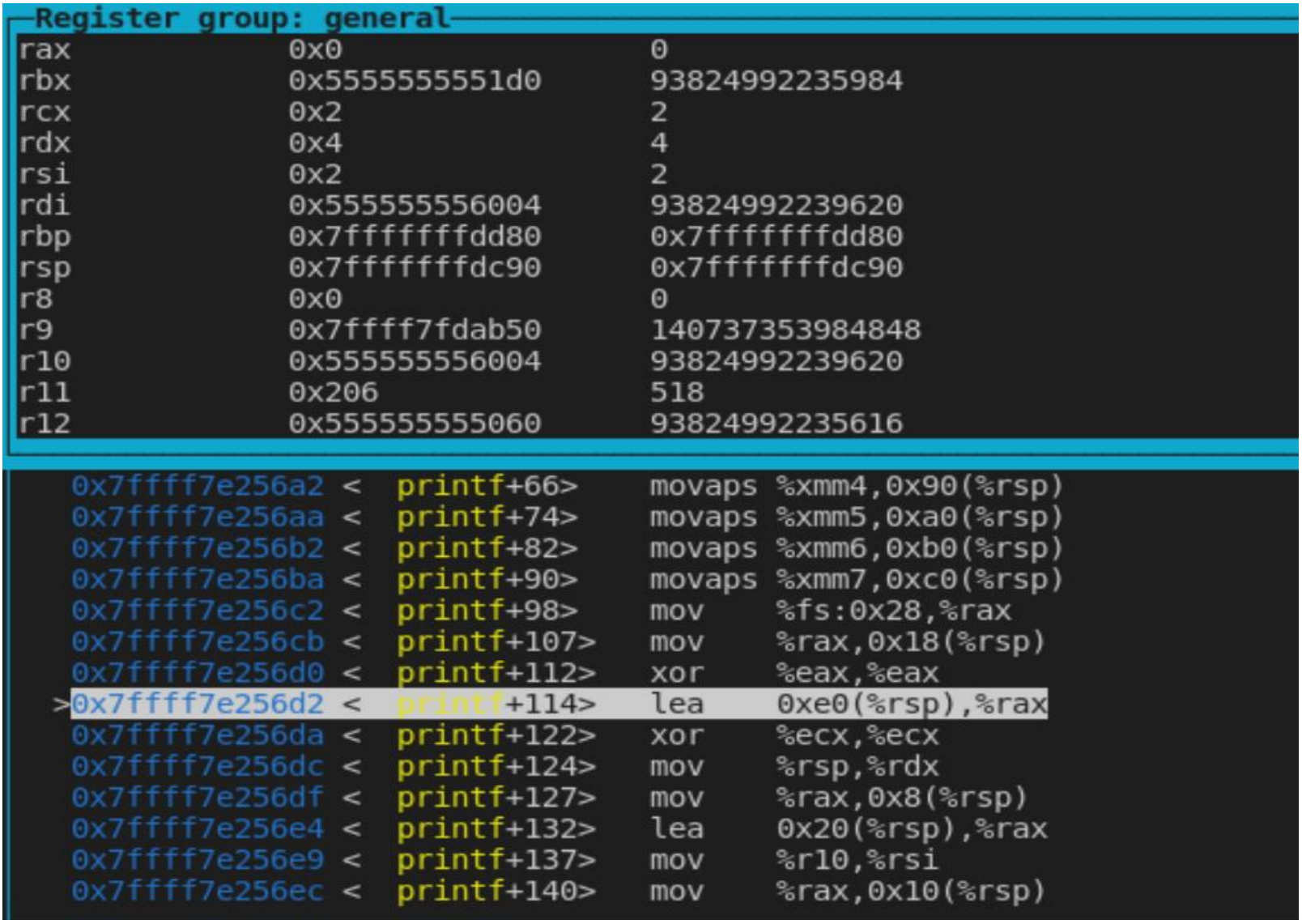


Figure 46: Complete stack frame

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

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.