Take Home Test 1

MdShahid Bin Emdad

April 17th, 2022

CSC 34200

TABLE OF CONTENTS

Objective:

The objective of this test is to demonstrate, understand and ability to compare 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. Some examples were provided by the professor, and we were instructed to analyze all the examples in all three systems. We were all instructed to analyze programs described in Sections 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8 on all 3 platforms. Finally, we were instructed to explain little endian, and big endian where appropriate, demonstrate while loops, for loops, if-then-else statements on each platform, describe differences and similarities in each case.

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

2-2_1:

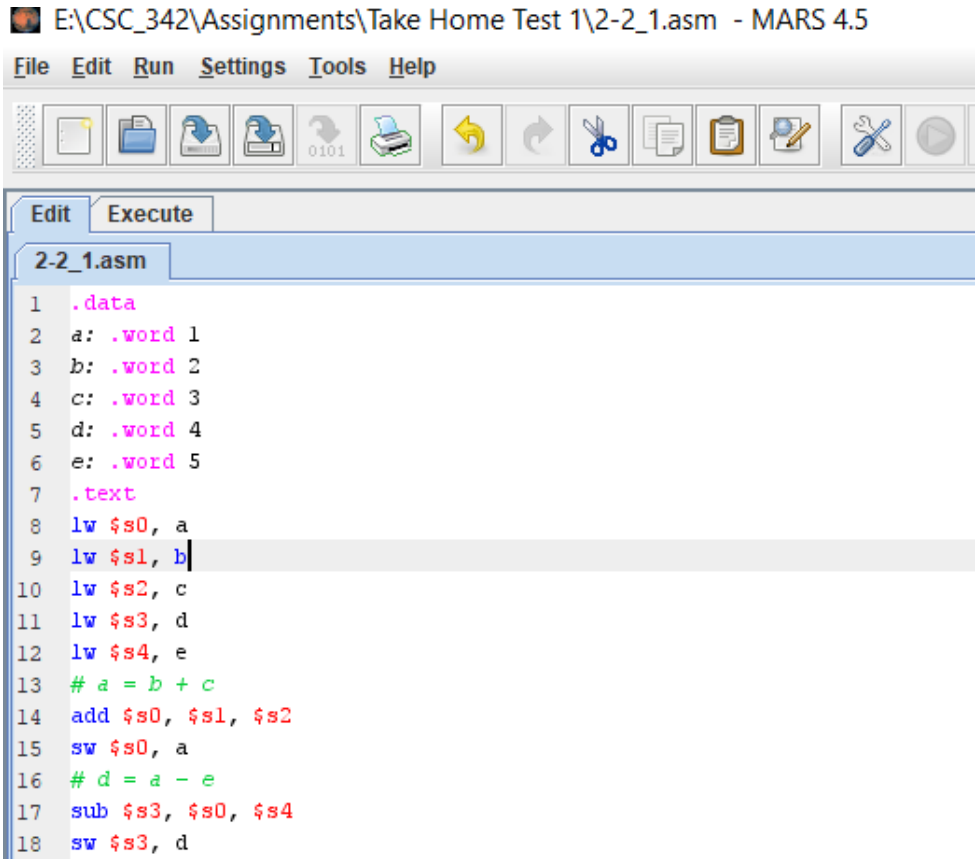

Figure 1: ASM code for 2-2_1

In figure 1, we can see the ASM code for the 2-2_1. It is doing two operations adding b and c, subtracting a and e. The variable a is the summation of b and c (b+c) and d is the difference of a and e(a-e).

In figure 2, we can see that it executed and operation completed succesfully.

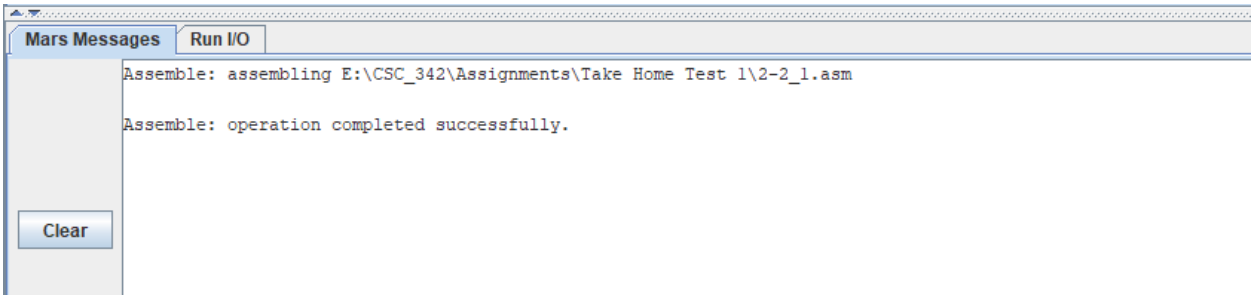

Figure 2: Operation Completion

In figure 3, we can see the text segment section for 2-2_1.asm. It is the disassembly of the MIPS file.

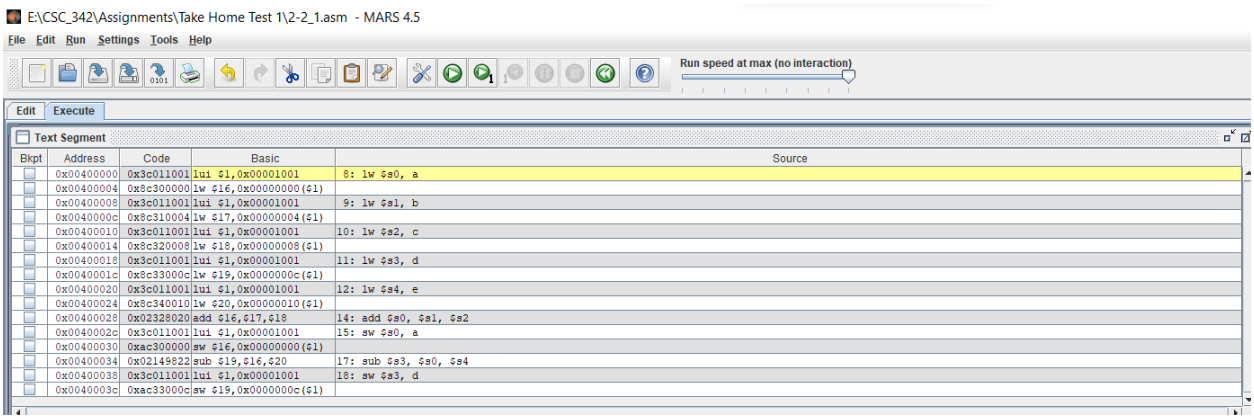

Figure 3: disassembly of 2-2_1.asm

The very first address is 0x00400000. It is generated by the execution of the stored instruction followed by lw $s0, a. The code column stores the address of every MIPS instruction, and it is 32bit long. The previous stored the first address and listed all the addresses. The next column is the machines turns the MIPS instructions to assembly code and execute it. The basic code lw $s0, a generated to assembly code, lui $1, 0x00001001. The assembly code lw mean the load of the word and sw is to store the word. The instructions add and sub are the necessary operations code. Source column shows the original code writes in the MIPS simulator.

In figure 4, we can see the Data Segment (memory from the MIPS simulator.

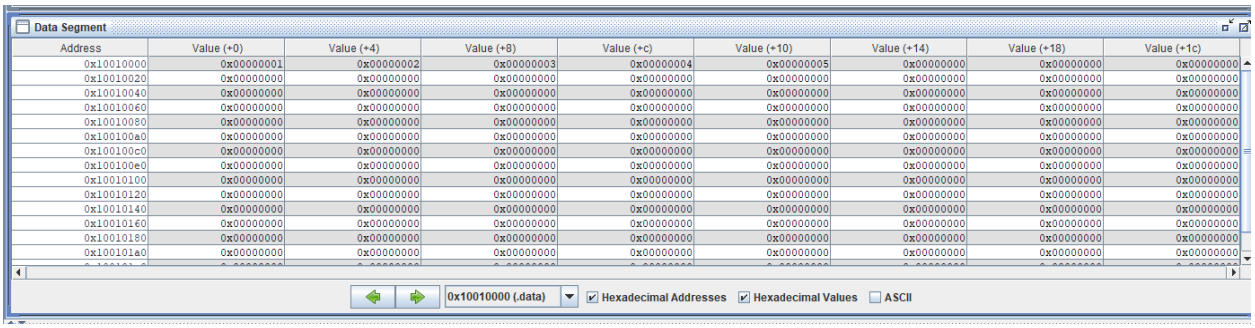

Figure 4: Memory from MIPS simulator

MIPS uses big endian address which means the most significant byte is placed on the lowest address in the hexadecimal address. For example, the significant value of +8 offset is 0 and significant bit is 0 and placed at the end of hexadecimal which is 0x00000003. The address, 0x10010000 where data being stored, and data storage starts.

In figure 5, we can see the register after the MIPS instructions executed. $t0: are fashionable registers and that they do now no longer want to be preserved. $s0: are stored registers and those want to be preserved for the duration of a call. $sp: that is our stack pointer. It factors to the pinnacle of the stack. $fp: This is our body pointer and unlike $sp it factors to the bottom of the stack. Pc: Program counter holds the deal with of the commands to be finished next.

Registers | Coproc 1 | Coproc 0

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

Figure 5: registers after execution of MIPS instructions

In figure 6, we can see the registers before arithmetic operations.

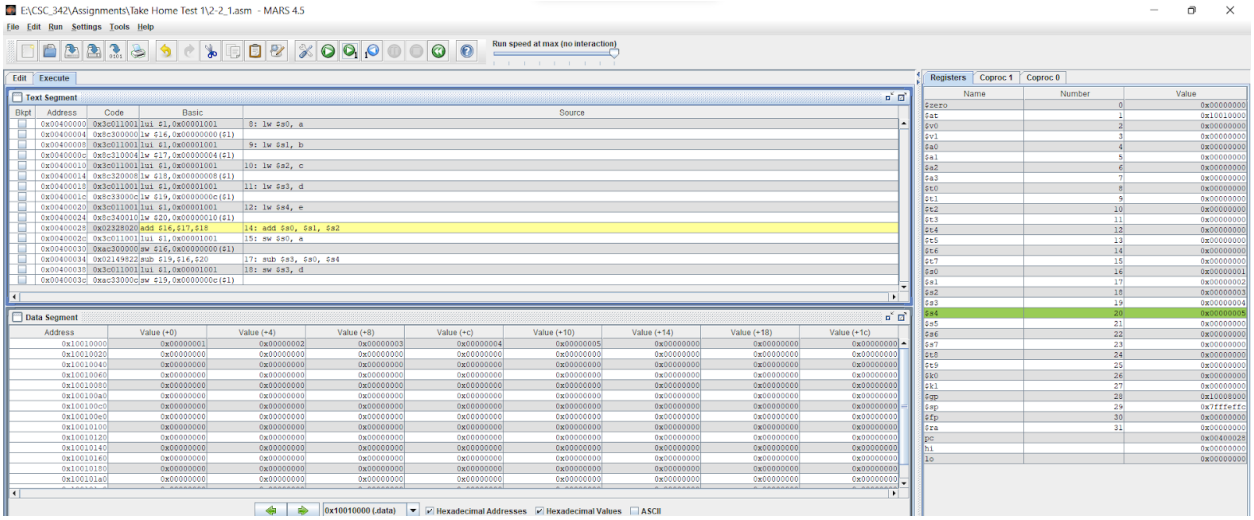

Figure 6: registers before arithmetic operations.

In figure 7, we can see the registers after arithmetic operations.

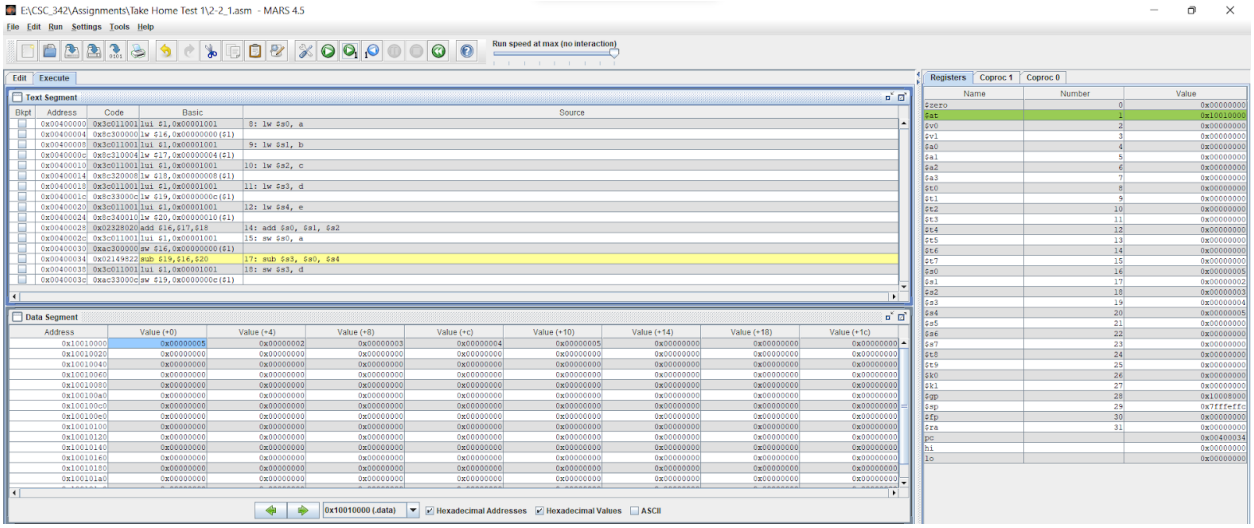

Figure 7: registers after arithmetic operations

We can see the register value stores under the name $at and the value of (+0) offset is 0x00000005.

2-2_2:

In figure 8, we can see the ASM code for the 2-2_2. It is an asm code arithmetic operations: addition and subsctractions.

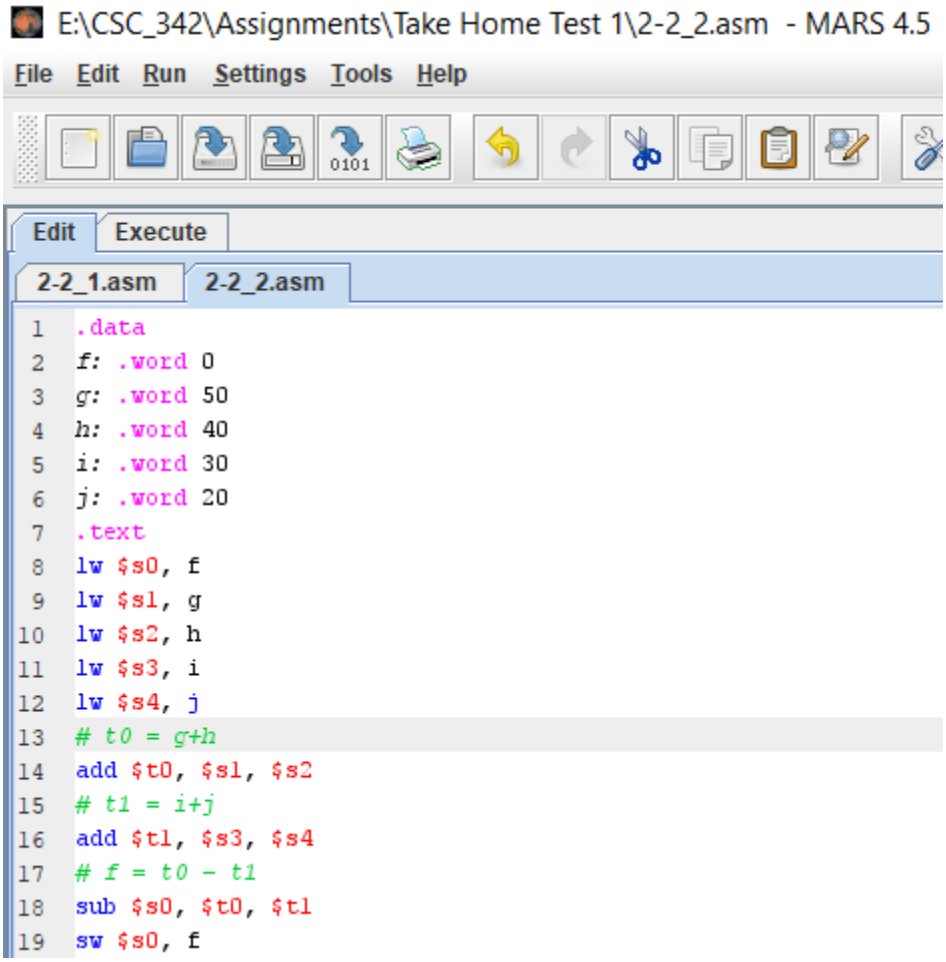

Figure 8: ASM code for 2-2_2

In figure 9, we can see that it executed and operation completed succesfully.

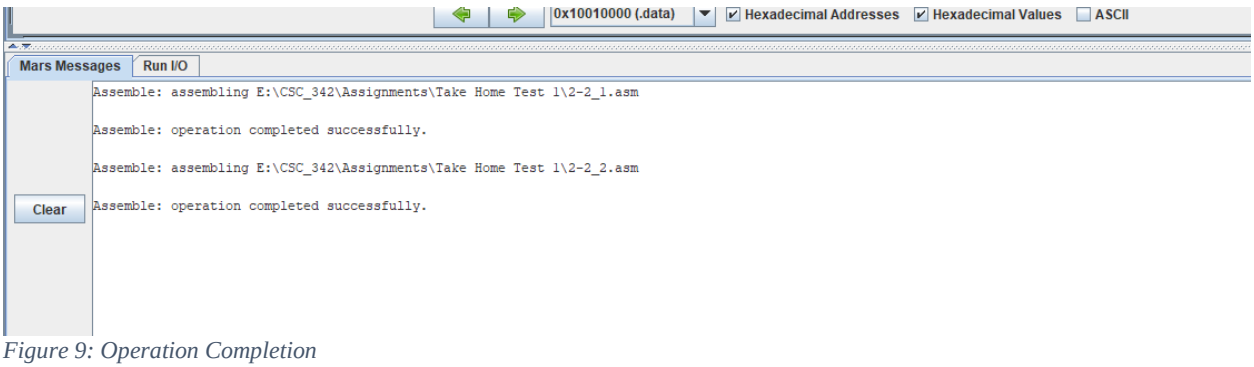

Figure 9: Operation Completion

In figure 10, we can see we can see the text segment section for 2-2_1.asm. It is the disassembly of the MIPS file. We can also see the memory and registers of 2-2_2.asm.

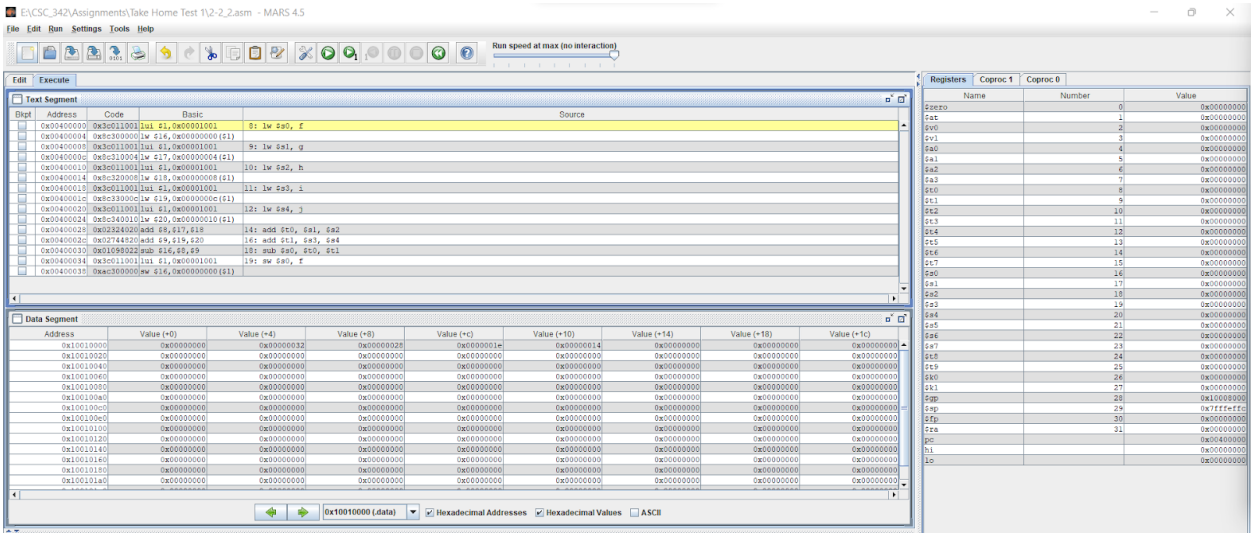

Figure 10: disassembly, memory, registers of 2-2_2.asm

The very first address is 0x00400000. It is generated by the execution of the stored instruction followed by lw $s0, f. The code column stores the address of every MIPS instruction, and it is 32bit long. The previous stored the first address and listed all the addresses. The next column is the machines turns the MIPS instructions to assembly code and execute it. The basic code lw $s0, f generated to assembly code, lui $1, 0x00001001. The assembly code lw mean the load of the word and sw is to store the word. The instructions add and sub are the necessary operations code. Source column shows the original code writes in the MIPS simulator.

MIPS uses big endian address which means the most significant byte is placed on the lowest address in the hexadecimal address. For example, the significant value of +8 offset is 0 and significant bit is 0 and placed at the end of hexadecimal which is 0x00000028. The address, 0x10010000 where data being stored, and data storage starts. $t0: are fashionable registers and that they do now no longer want to be preserved. $s0: are stored registers and those want to be preserved for the duration of a call. $sp: that is our stack pointer. It factors to the pinnacle of the stack. $fp: This is our body pointer and unlike $sp it factors to the bottom of the stack. Pc: Program counter holds the deal with of the commands to be finished next.

In figure 11, we can see the registers before arithmetic operations.

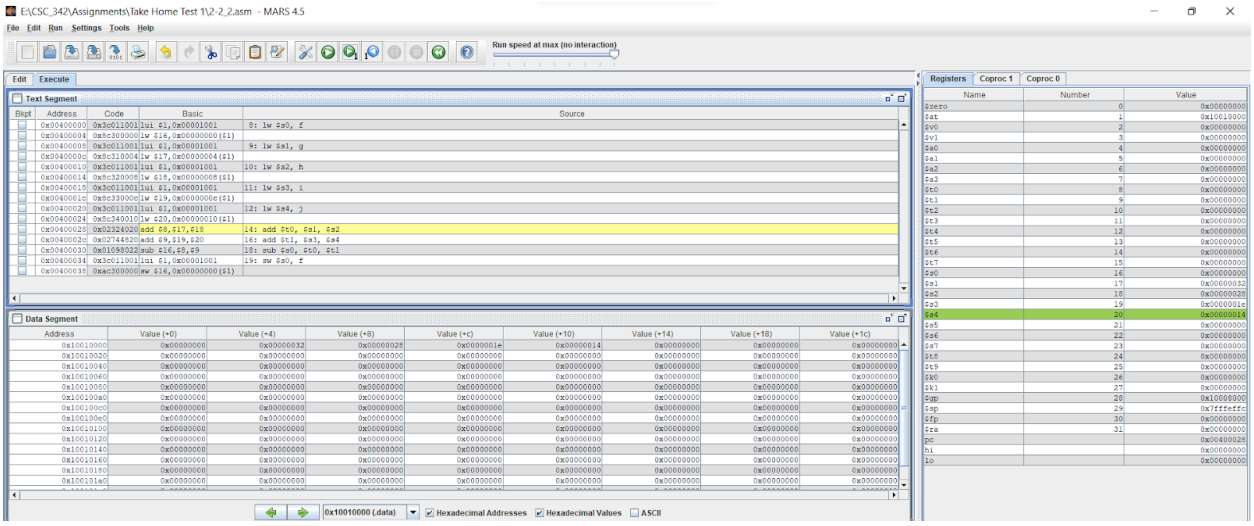

Figure 11: registers before arithmetic operations.

In figure 12, we can see the registers after arithmetic operations.

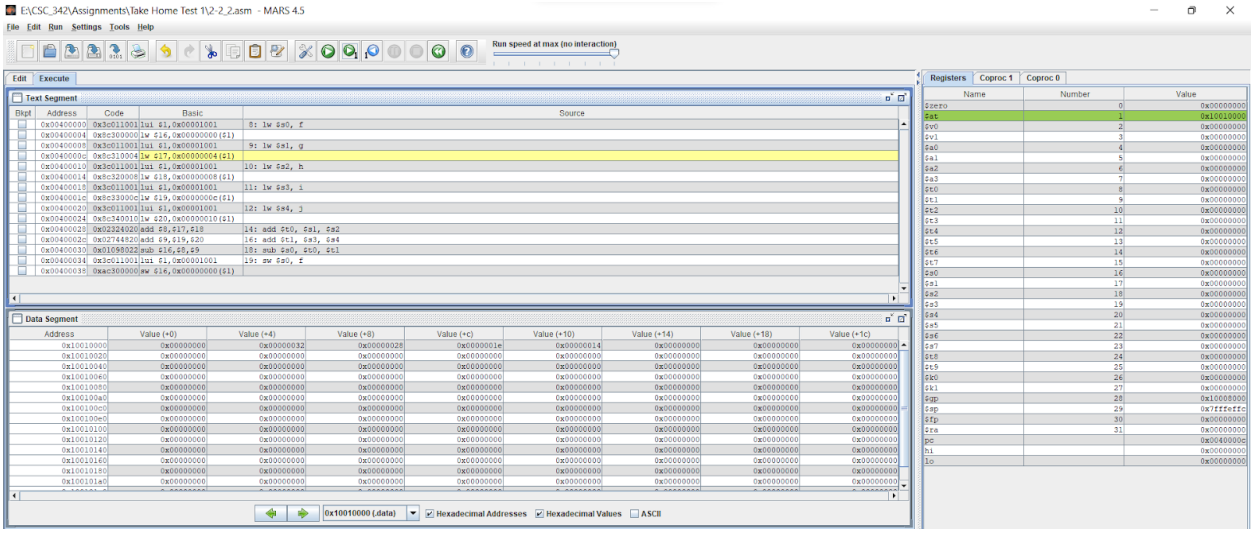

Figure 12: registers after arithmetic operations.

However, the difference between the code 2-2_1 and 2-2_2 is here we are using temporary registers, $s3 in the offset of 32 because we are storing to A [8] which holds4 bytes. If we multiple, we get 8*4=32.

2-3_1:

In figure 13, we can see the ASM code for 2-3_1. The arithmetic operations happening here are additions on MIPS registers and arrays contains the value of the variable.

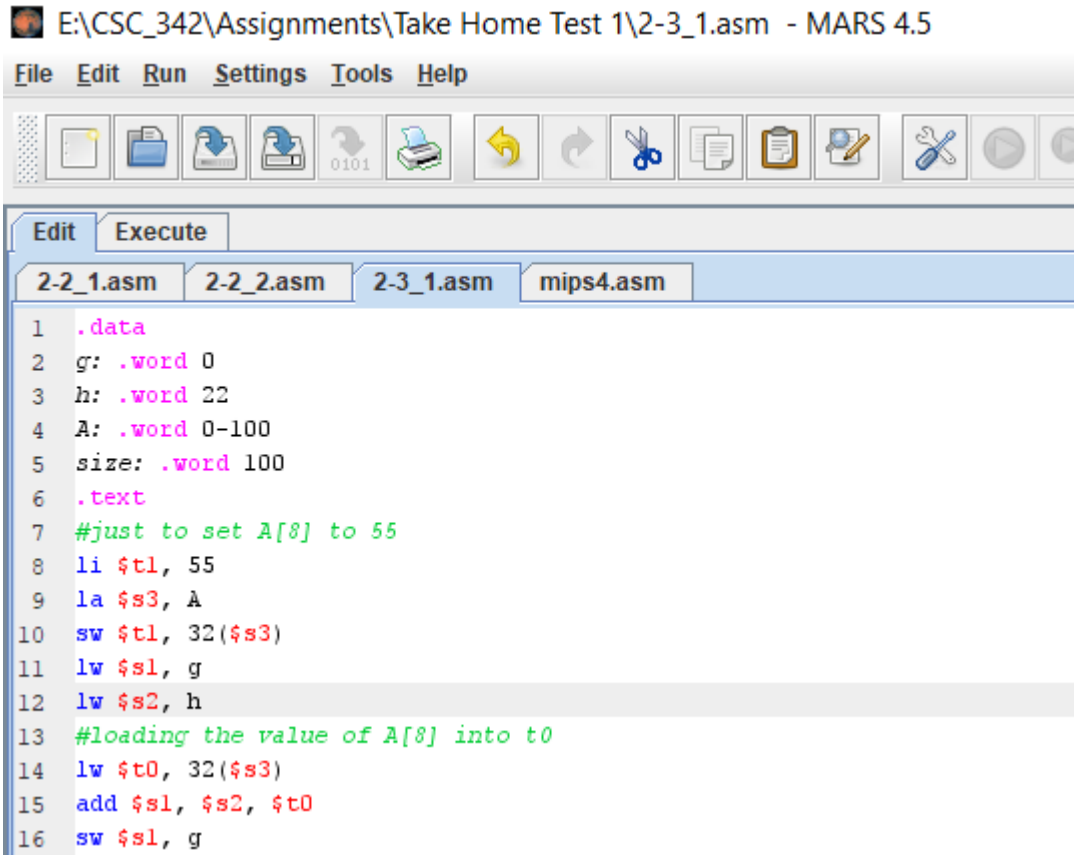

Figure 13: ASM code for 2-3_1

In figure 14, we can see that it executed, and operation completed successfully.

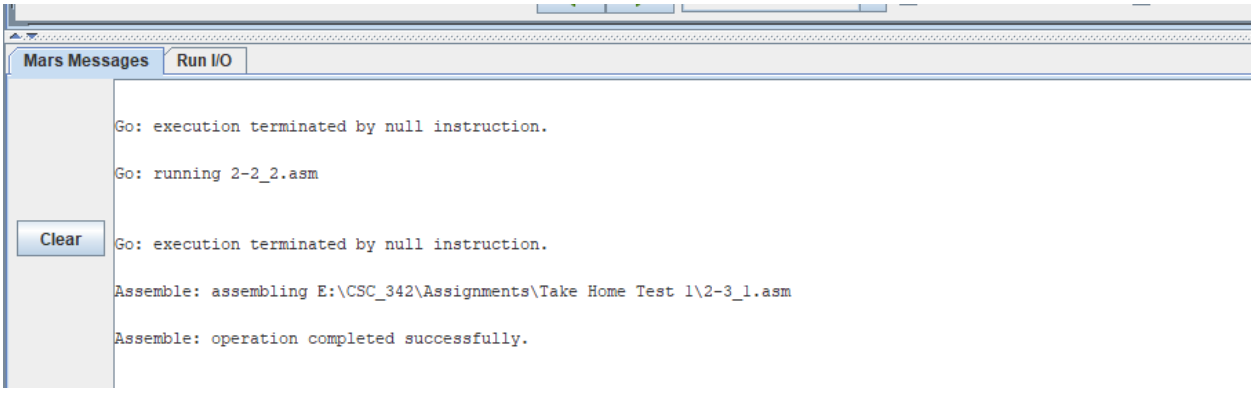

Figure 14: Operation Completion

In figure 15, we can see the disassembly, memory, registers of MIPS file, 2-3_1.asm.

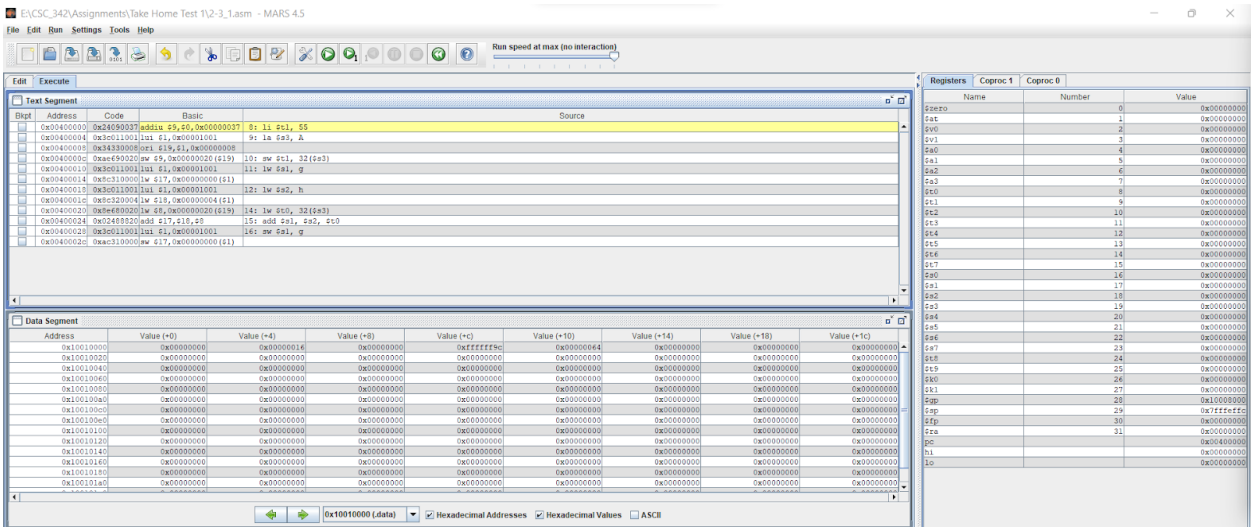

Figure 15: disassembly, memory, registers of MIPS file, 2-3_1.asm

The second address is 0x00400000. It is generated by the execution of the stored instruction followed by la $s3, A. The code column stores the address of every MIPS instruction, and it is 32bit long. The array address being assigned to register $s3. The previous stored the first address and listed all the addresses. The next column is the machines turns the MIPS instructions to assembly code and execute it. The basic code la $s3, A, generated to assembly code, lui $1, 0x00001001. The assembly code lw mean the load of the word and sw is to store the word. The instructions add and sub are the necessary operations code. Source column shows the original code writes in the MIPS simulator.

MIPS uses big endian address which means the most significant byte is placed on the lowest address in the hexadecimal address. For example, the significant value of +0 offset is 0 and placed at the end of hexadecimal which is 0x0000004d. The address, 0x10010000 where data being stored, and data storage starts. $t0: are fashionable registers and that they do now no longer want to be preserved. $s0: are stored registers and those want to be preserved for the duration of a call. $sp: that is our stack pointer. It factors to the pinnacle of the stack. $fp: This is our body pointer and unlike $sp it factors to the bottom of the stack. Pc: Program counter holds the deal with of the commands to be finished next.

In figure 16, we can see the registers after arithmetic operations.

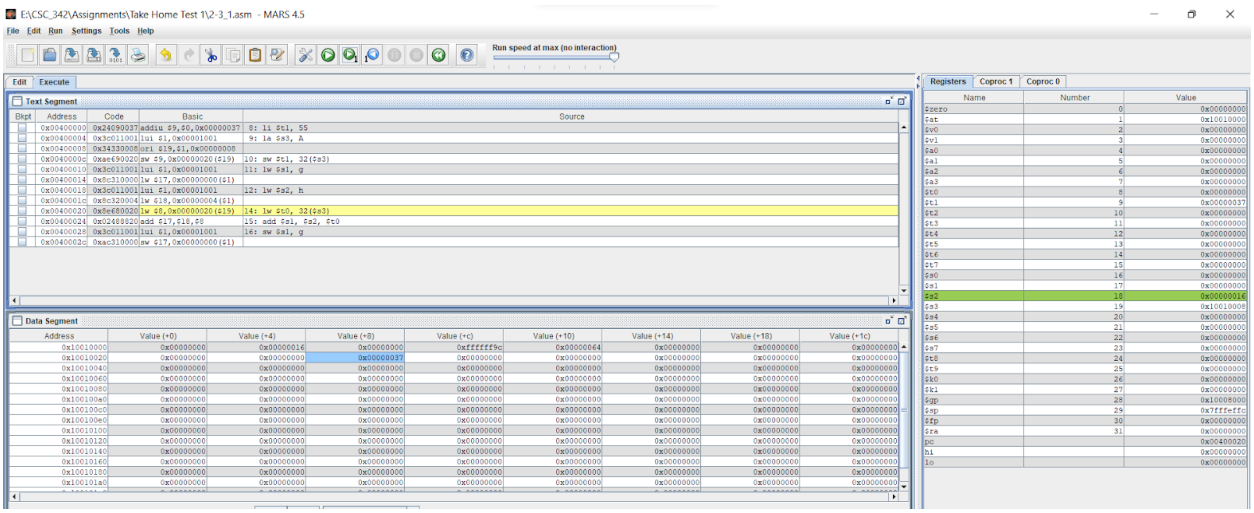

Figure 16: registers after arithmetic operations.

2-3_2:

In figure 17, we can see the ASM code for 2-3_2. The arithmetic operations happening here are additions on MIPS registers and arrays contains the value of the variable.

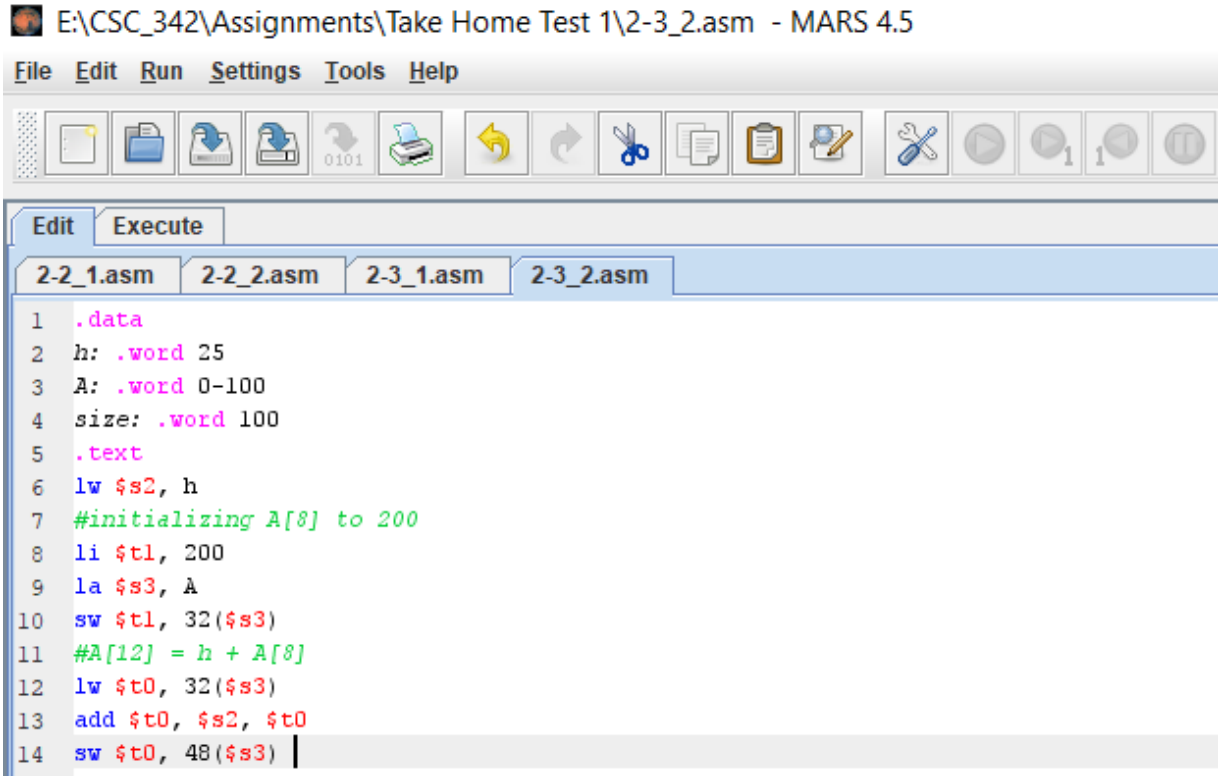

Figure 17: ASM code for 2-3_2

In figure 18, we can see that it executed, and operation completed successfully.

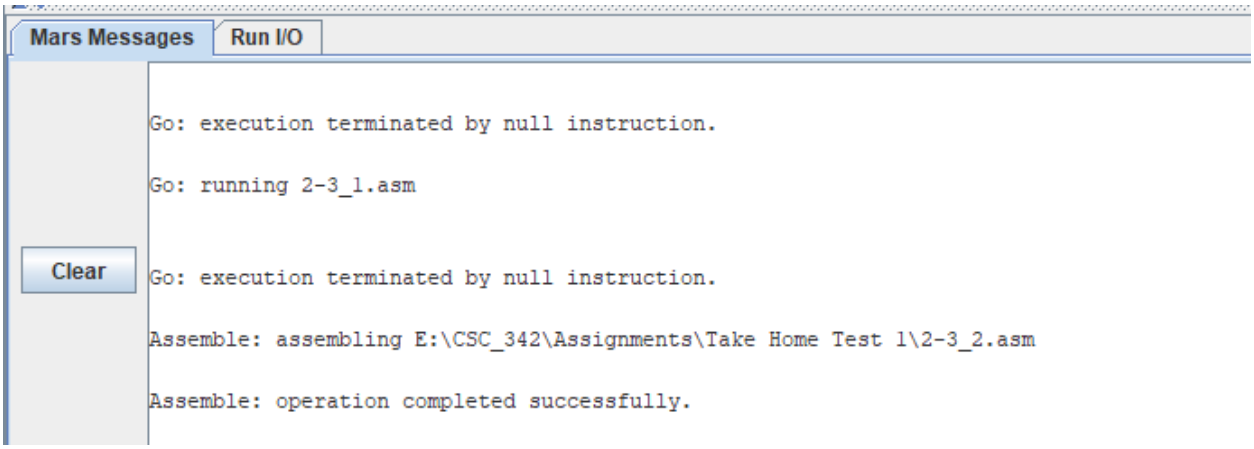

Figure 18: Operation Completion

In figure 19, we can see the disassembly, memory, registers of MIPS file, 2-3_2.asm.

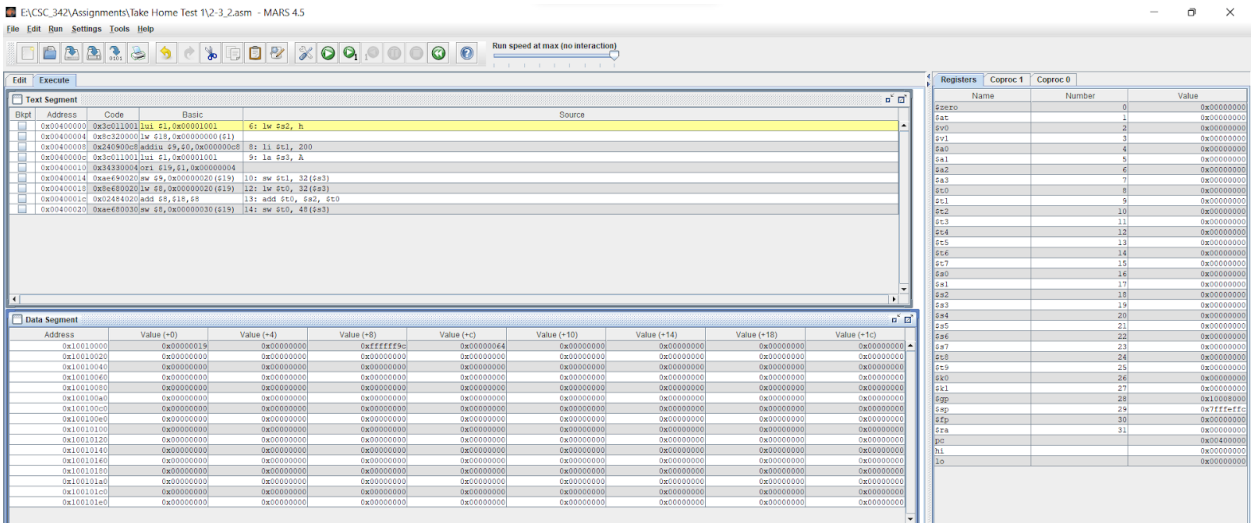

Figure 19: disassembly, memory, registers of MIPS file, 2-3_2.asm.

The very first address is 0x00400000. It is generated by the execution of the stored instruction followed by la $s2, h. The code column stores the address of every MIPS instruction, and it is 32bit long. The array address being assigned to register $s3. The previous stored the first address and listed all the addresses. The next column is the machines turns the MIPS instructions to assembly code and execute it. The basic code la $s2, h, generated to assembly code, lui $1, 0x00001001. The assembly code lw mean the load of the word and sw is to store the word. The instructions add and sub are the necessary operations code. Source column shows the original code writes in the MIPS simulator.

The stack pointer is at value (+14), addrres is 0x000000e1 and in decimal its 255. The value is stored in $t0. $t0: are fashionable registers and that they do now no longer want to be preserved. $s0: are stored registers and those want to be preserved for the duration of a call. $sp: that is our stack pointer. It factors to the pinnacle of the stack. $fp: This is our body pointer and unlike $sp it factors to the bottom of the stack. Pc: Program counter holds the deal with of the commands to be finished next.

In figure 20, we can see the registers before arithmetic operations.

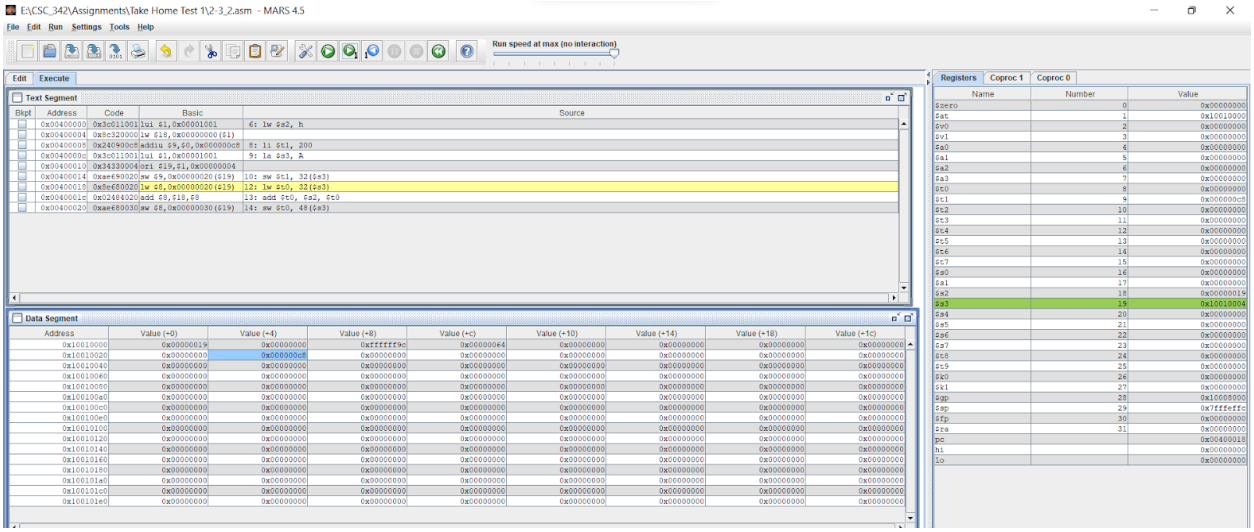

Figure 20: registers before arithmetic operations.

In figure 7, we can see the registers after arithmetic operations.

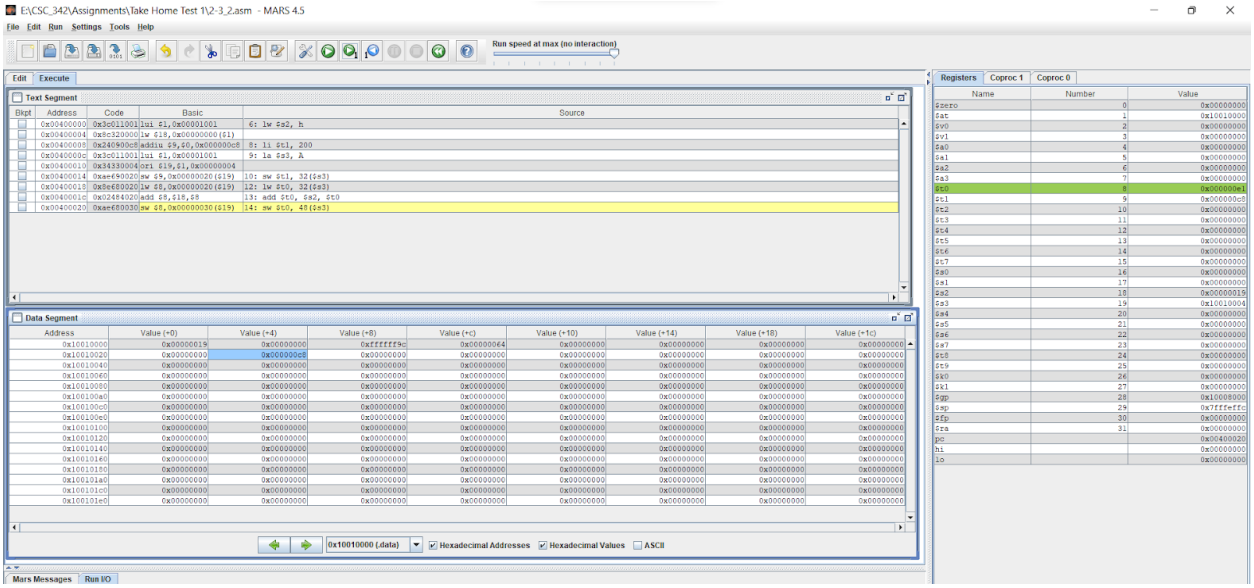

Figure 21: registers after arithmetic operations.

We can see the value of (+4) offset is 0x000000c8 and in the register value of $t0 is 0x000000e1.

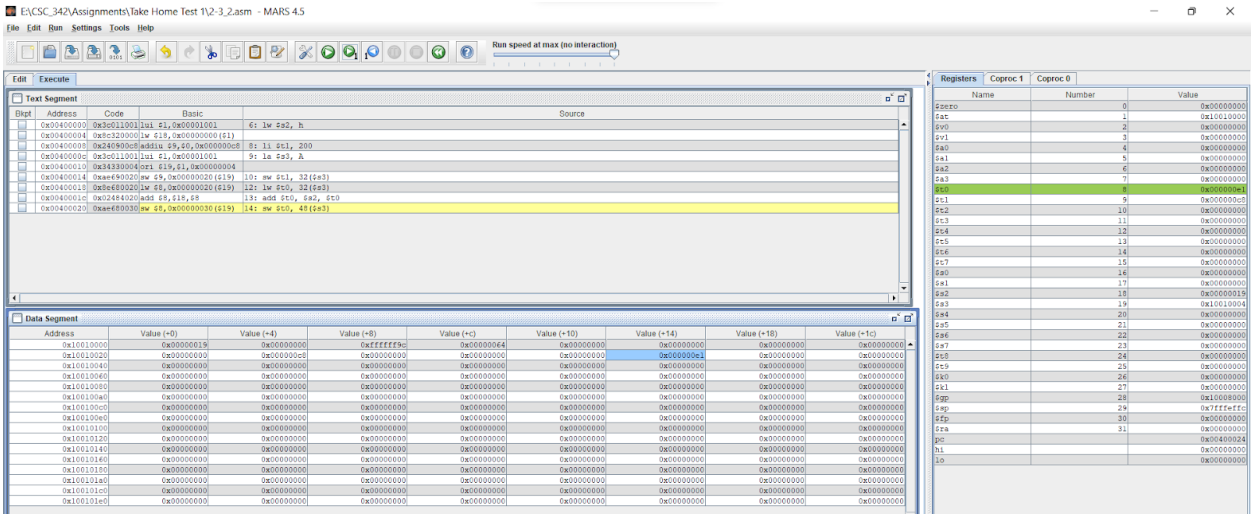

Figure 22: memory value changed

In figure 22, we can see the value of (+14) offset is 0x000000e1 and in the register value of $t0 is 0x000000e1 and address is 0x10010020.

2-5_2:

In figure 23, we can see the ASM code for 2-5_2. The arithmetic operations happening here are additions on MIPS registers and arrays contains the value of the variable.

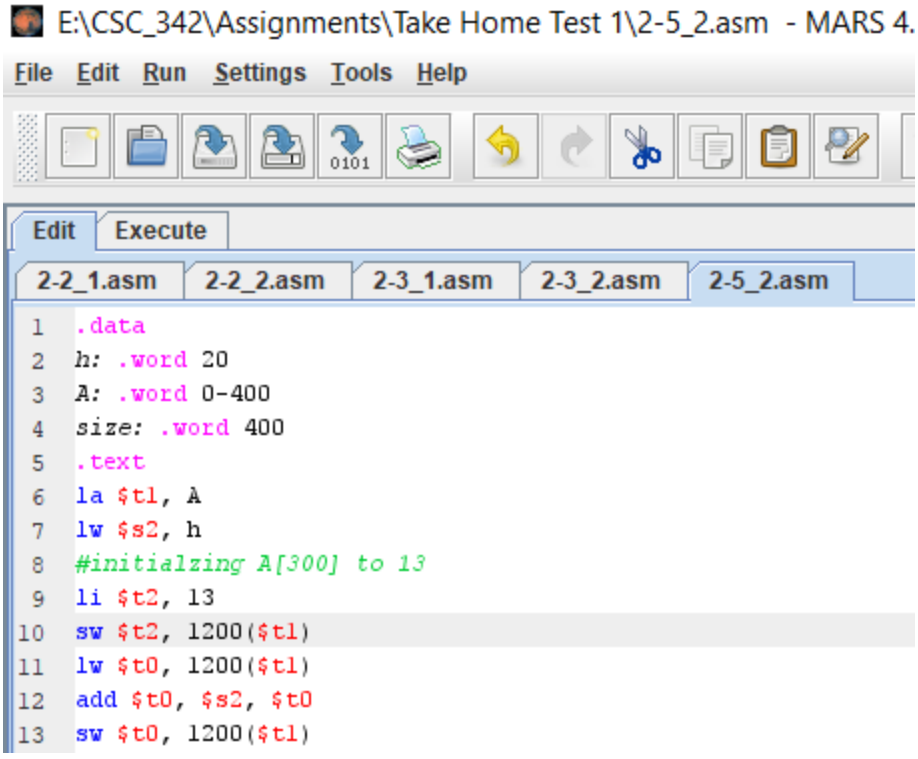

Figure 23: ASM code for 2-5_2

In figure 24, we can see that it executed and operation completed successfully.

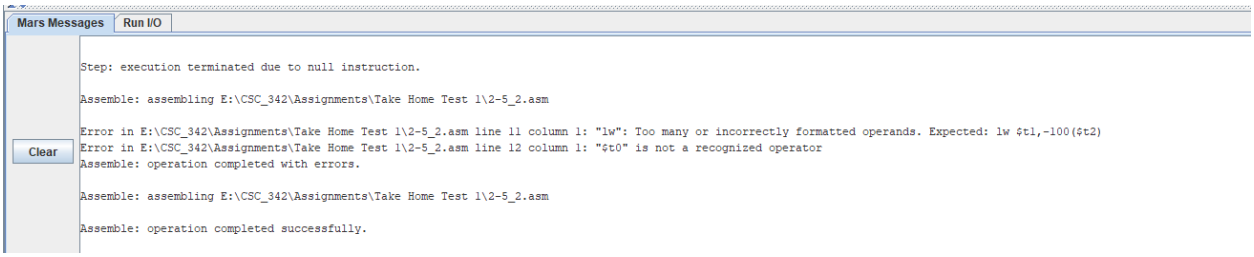

Figure 24: Operation Completion

In figure 25, we can see the disassembly, memory, registers of MIPS file, 2-5_2.asm.

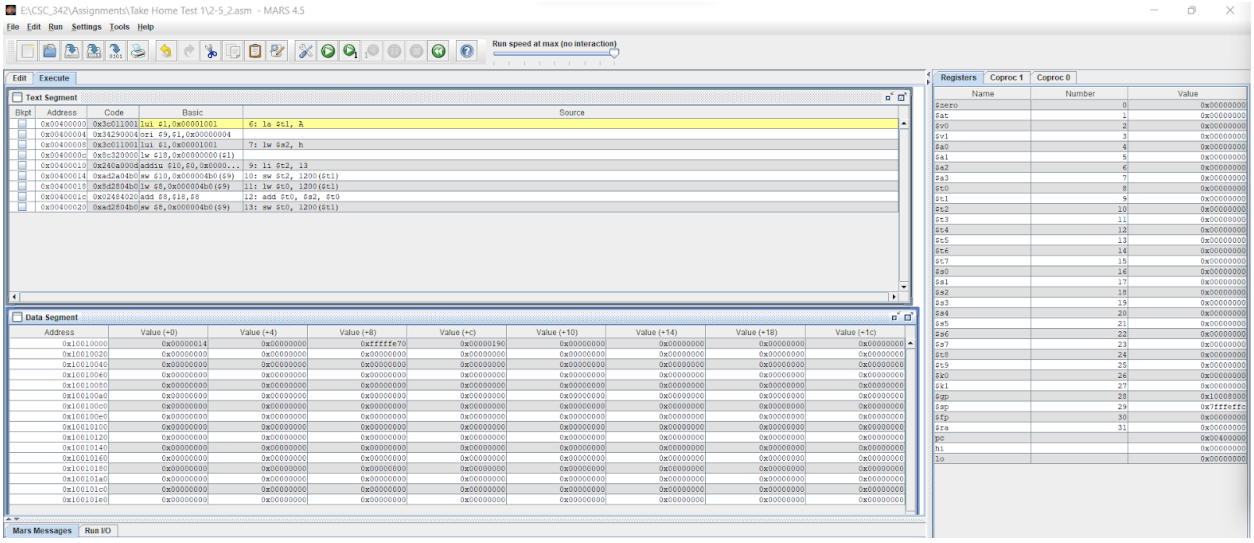

Figure 25: disassembly, memory, registers of MIPS file, 2-5_2.asm.

The very first address is 0x00400000. It is generated by the execution of the stored instruction followed by la $t1, A. The code column stores the address of every MIPS instruction, and it is 32bit long. The array address being assigned to register $s3. The previous stored the first address and listed all the addresses. The next column is the machines turns the MIPS instructions to assembly code and execute it. The basic code la $t1, A, generated to assembly code, lui $1, 0x00001001. The assembly code lw mean the load of the word and sw is to store the word. The instructions add and sub are the necessary operations code. Source column shows the original code writes in the MIPS simulator.

The value of 13 is in the A [300] and we need an offset of 1200 because 300*4 = 1200. This is not the start of the array.

The stack pointer is at value (+0), address is 0x00000014 and 20 in decimal. The value is stored in $t0. $t0: are fashionable registers and that they do now no longer want to be preserved. $s0: are stored registers and those want to be preserved for the duration of a call. $sp: that is our stack pointer. It factors to the pinnacle of the stack. $fp: This is our body pointer and unlike $sp it factors to the bottom of the stack. Pc: Program counter holds the deal with of the commands to be finished next.

In figure 26, we can see the registers before arithmetic operations.

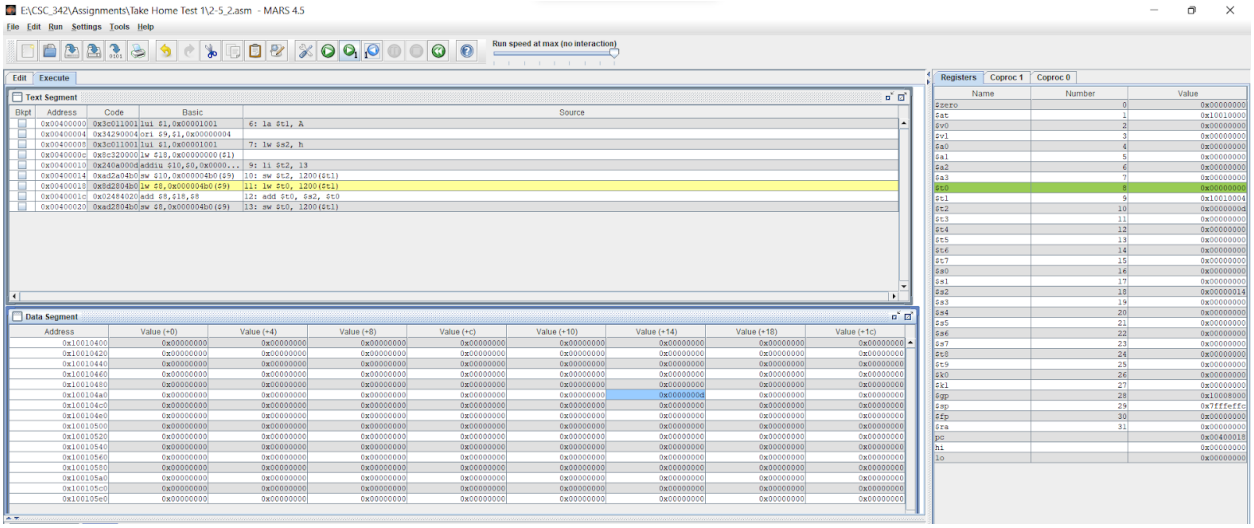

Figure 26: registers before arithmetic operations

In figure 27, we can see the registers after arithmetic operations.

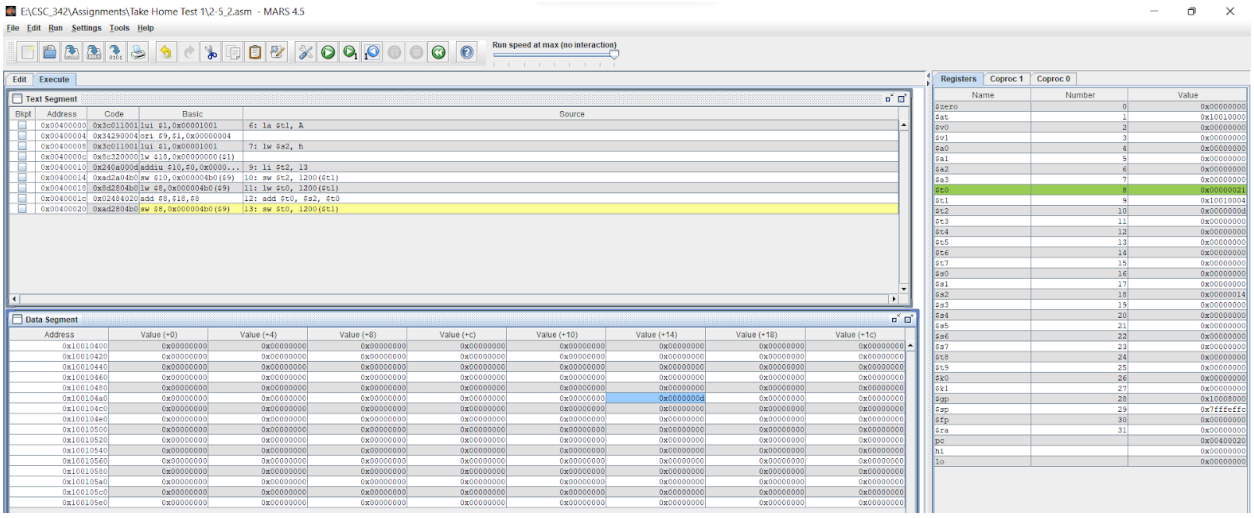

Figure 27: registers after arithmetic operations

We can see the register value stores under the name $t0, address is 0x00000021 and the value of (+14) offset is 0x000000d which 4.

2-6_1:

In figure 28, we can see the ASM code for 2-6_1. This ASM code for bitwise and logical operations on MIPS registers that contain the specified value given in the code.

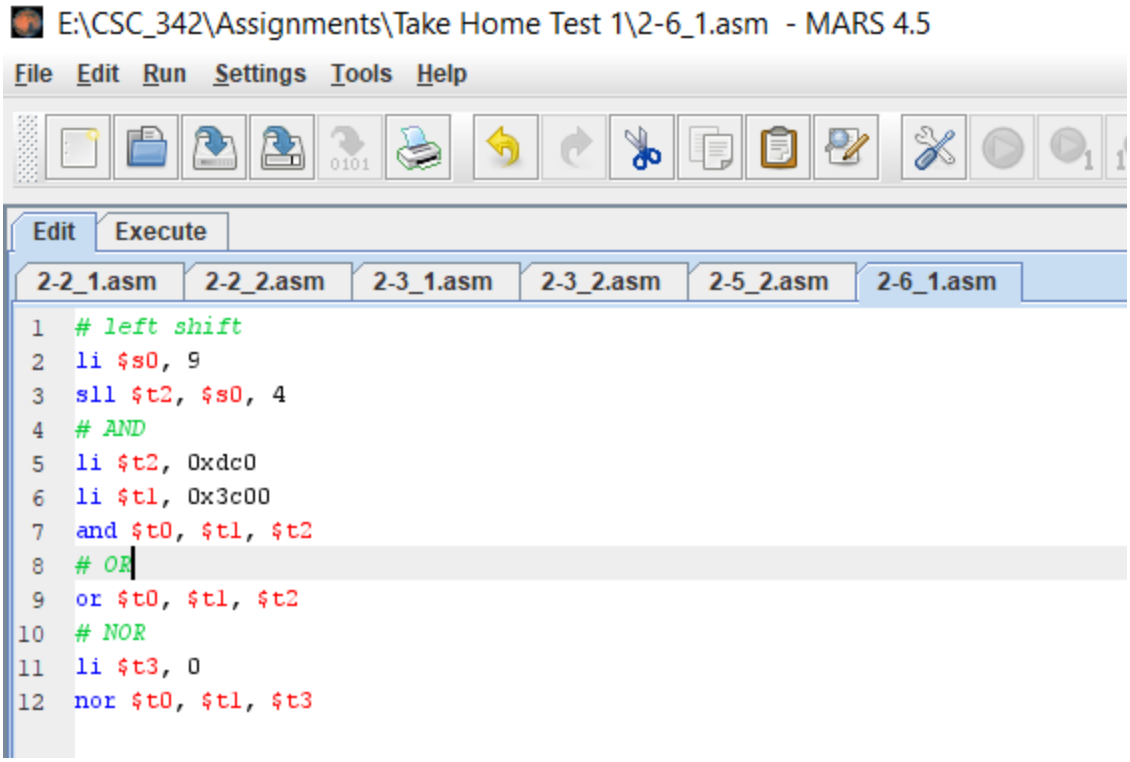

Figure 28: ASM code for 2-6_1

In figure 29, we can see that it executed and operation completed successfully.

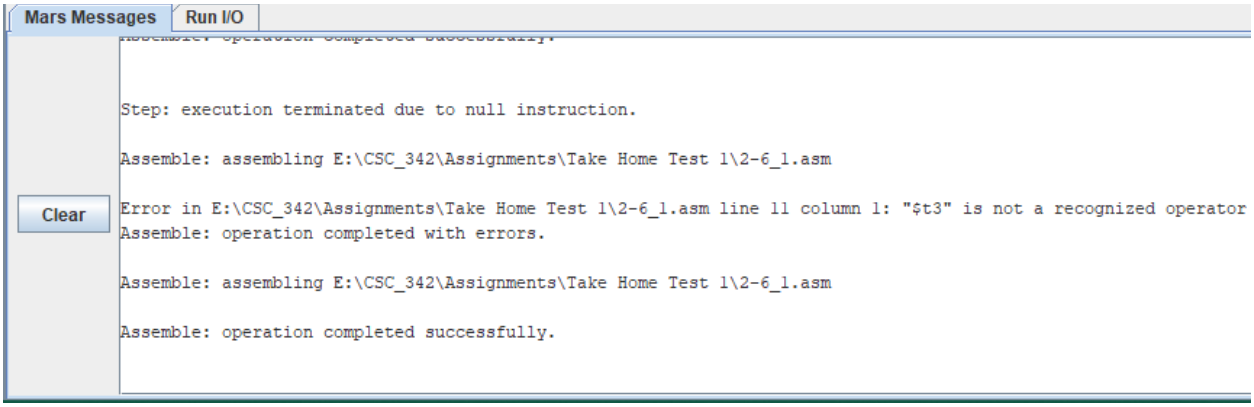

Figure 29: Operation Completion

In figure 30, we can see the disassembly, memory, registers of MIPS file, 2-6_1.asm.

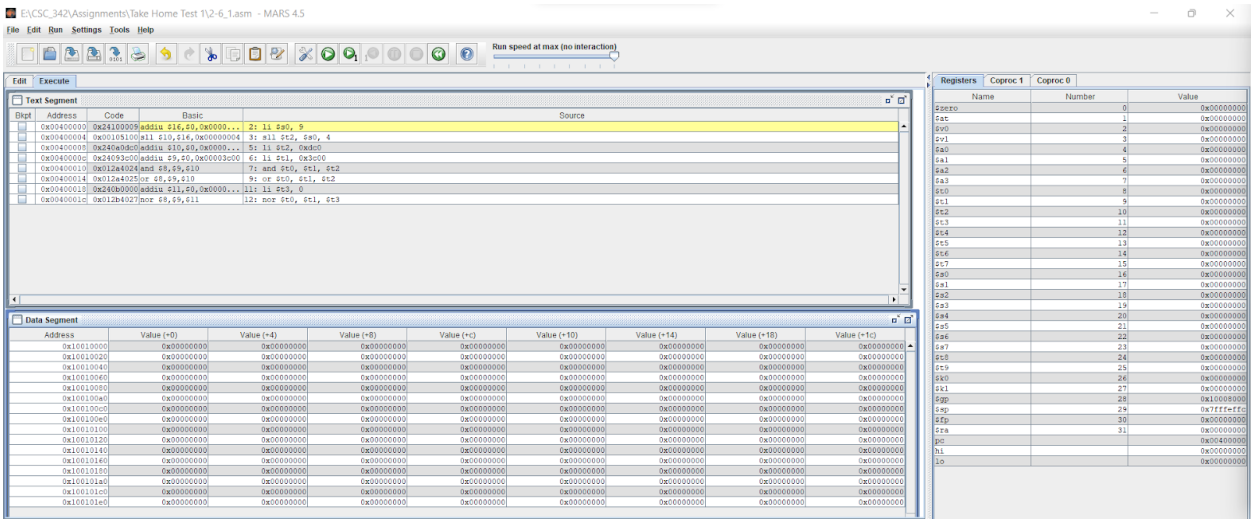

Figure 30: disassembly, memory, registers of MIPS file, 2-6_1.asm.

The very first address is 0x00400000. It is generated by the execution of the stored instruction followed by la $s0, 9. The code column stores the address of every MIPS instruction, and it is 32bit long. The array address being assigned to register $s0. The previous stored the first address and listed all the addresses. The next column is the machines turns the MIPS instructions to assembly code and execute it. The assembly code lw mean the load of the word and sw is to store the word. The instructions add and sub are the necessary operations code. Source column shows the original code writes in the MIPS simulator. As we do not have any variable in the data section, that's why all the data segments had 0 values. The value is stored in $t0.

In figure 31, we can see the registers before arithmetic operations.

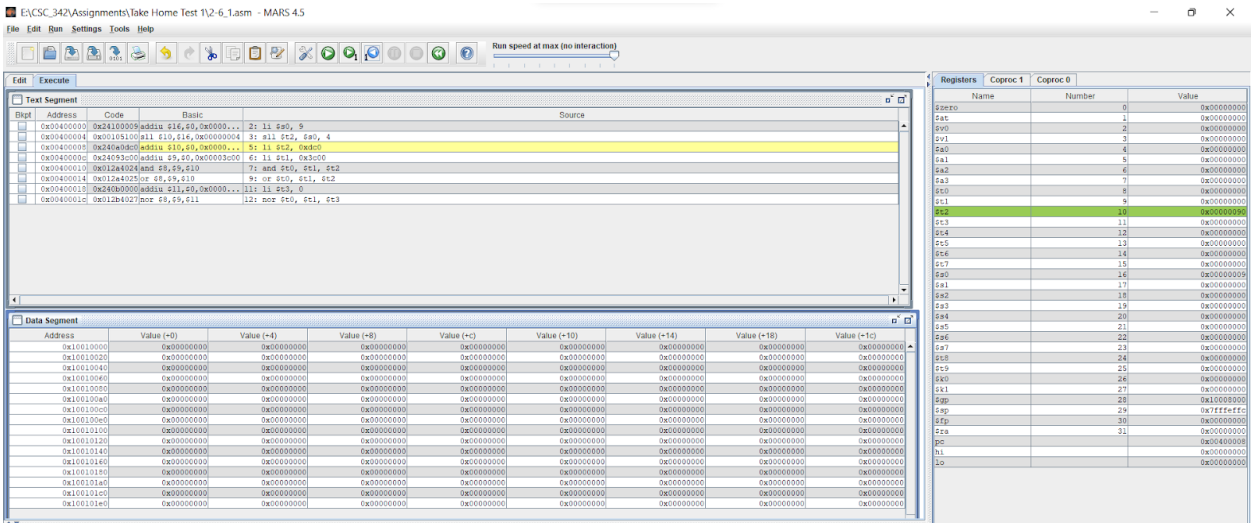

Figure 31: registers before arithmetic operations.

Line 5 shifts to the left to the contents of register $s0 and stores the result in register $t2. The address for $t2 is 0x00000090 and the value in decimal is 133.

In figure 32, we can see the registers after arithmetic operations.

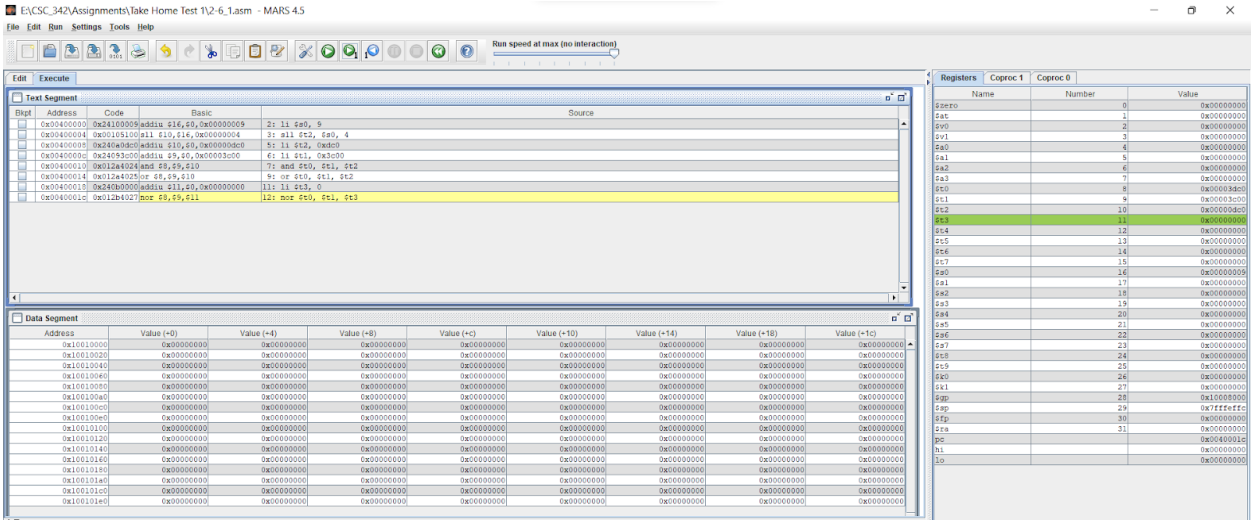

Figure 32: registers after arithmetic operations.

We can see the register value stores under the name $t3 and the value of (+0) offset is 0x0000000.

2-7_1:

In figure 33, we can see the ASM code for 2-7_1. This code is taken from the outside resources, and it is for if-else statements. Under certain circumstances, the code will decide and execute.

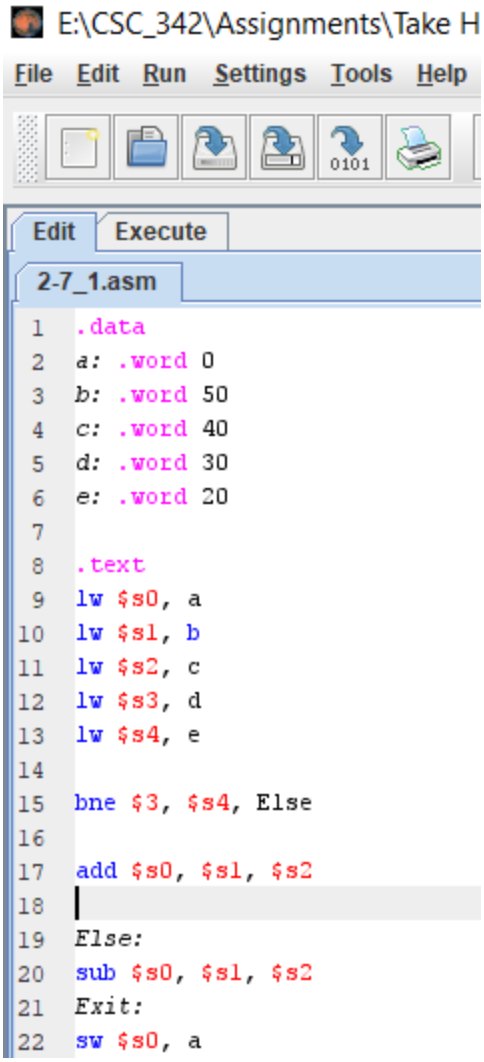

Figure 33: ASM code for 2-7_1

In figure 34, we can see that it executed and operation completed successfully.

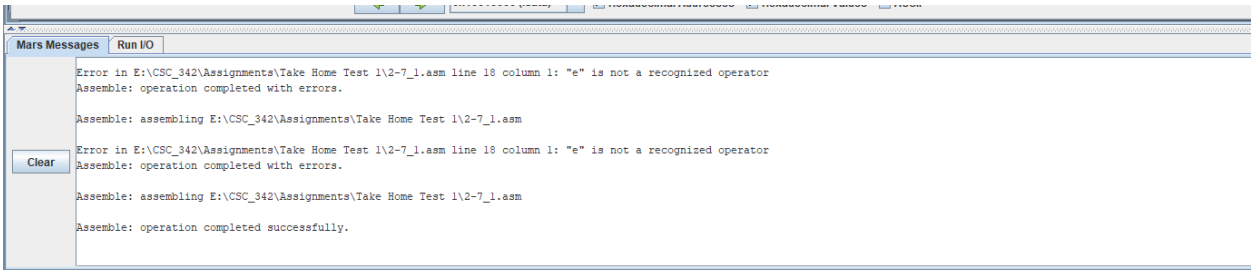

Figure 34: Operation Completion

In figure 35, we can see the disassembly, memory, registers of MIPS file, 2-7_1.asm.

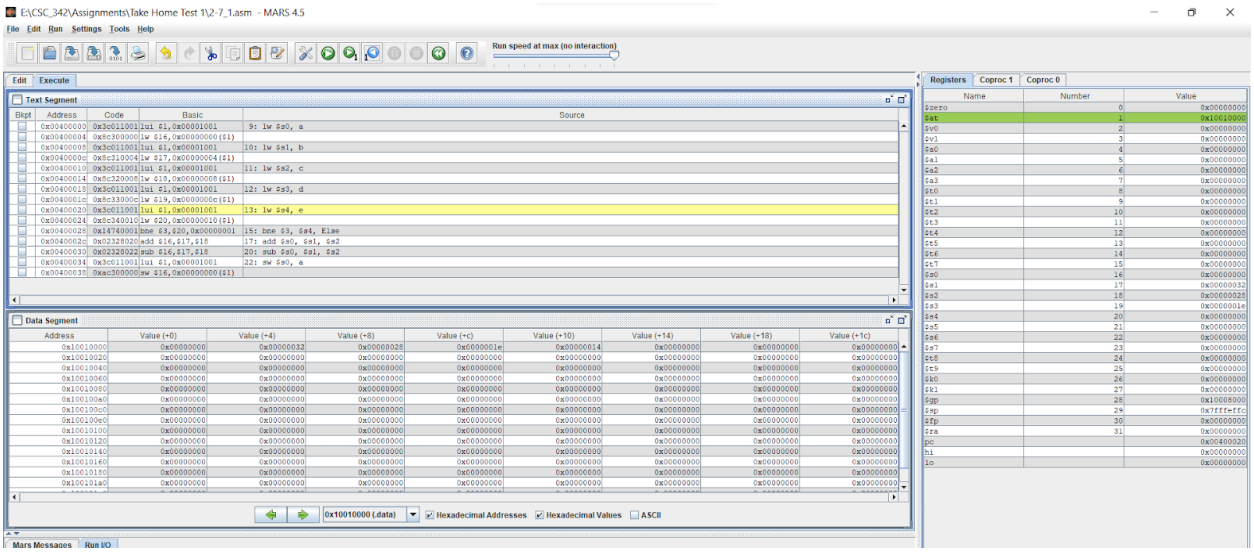

Figure 35: disassembly, memory, registers of MIPS file, 2-7_1.asm.

The address from 0x00400000 - 0x00400020, it is initializing the variables. It is generated by the execution of the stored instruction followed by lw $s0, a, lw $s0, b, lw $s0, c, lw $s0, d, lw $s0, e. The code column stores the address of every MIPS instruction, and it is 32bit long. The previous stored the first address and listed all the addresses. The next column is the machines turns the MIPS instructions to assembly code and execute it. The assembly code lw mean the load of the word and sw is to store the word. The instructions add and sub are the necessary operations code. Source column shows the original code writes in the MIPS simulator.

In figure 36, we can see the registers after arithmetic operations.

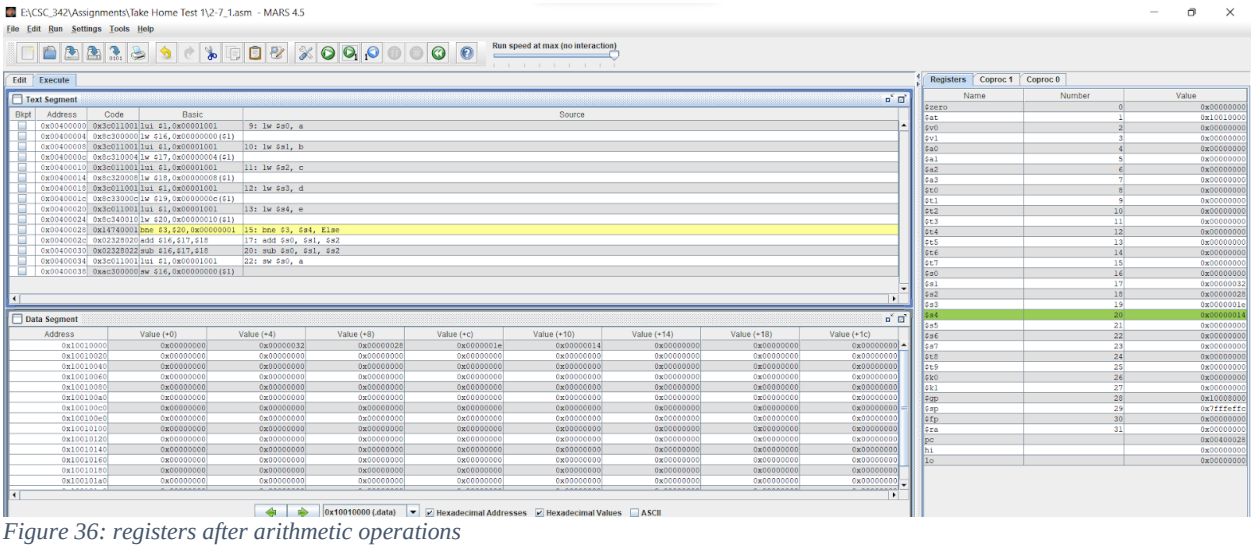

Figure 36: registers after arithmetic operations

After the line 15 executed, we can see the register and values in hexadecimal. After line 18, the register values in $s3 and $s4 did not equal which lead the code to line 25 and performed the arithmetic operation. We can see the We can see the register value stores under the name $s4 and the value of (+10) offset is 0x00000014.

2-8_1:

In figure 37, we can see the ASM code for 2-8_1. We are using main and myadd procedure to perform an arithmetic operation. The variable a, b will get call by the function and perform addition and return 30.

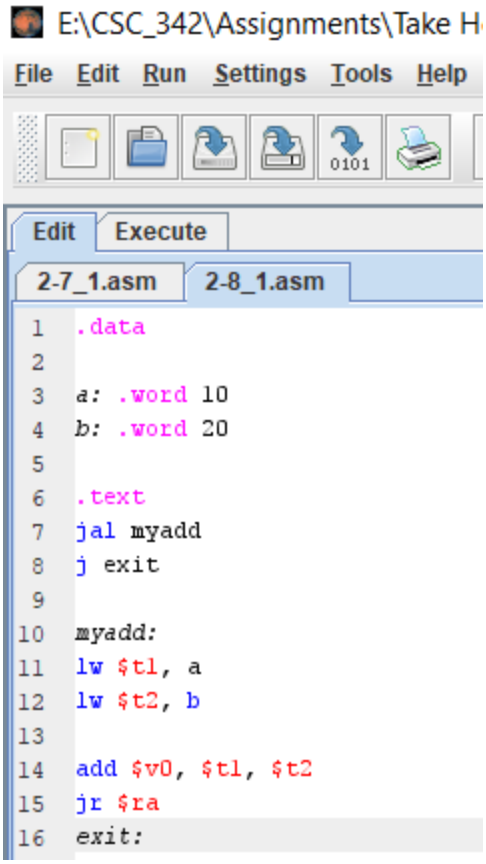

Figure 37: ASM code for 2-8_1

In figure 38, we can see that it executed, and operation completed successfully.

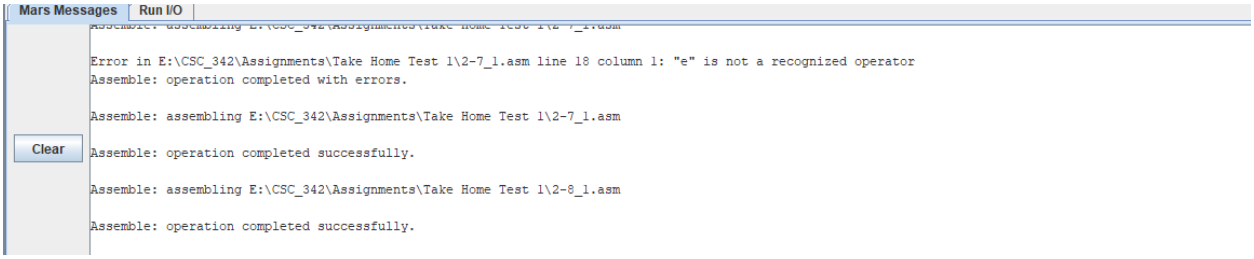

Figure 38: Operation Completion

In figure 39, we can see we can see the disassembly, memory, registers of MIPS file, 2-8_1.asm. It is the before arithmetic operations.

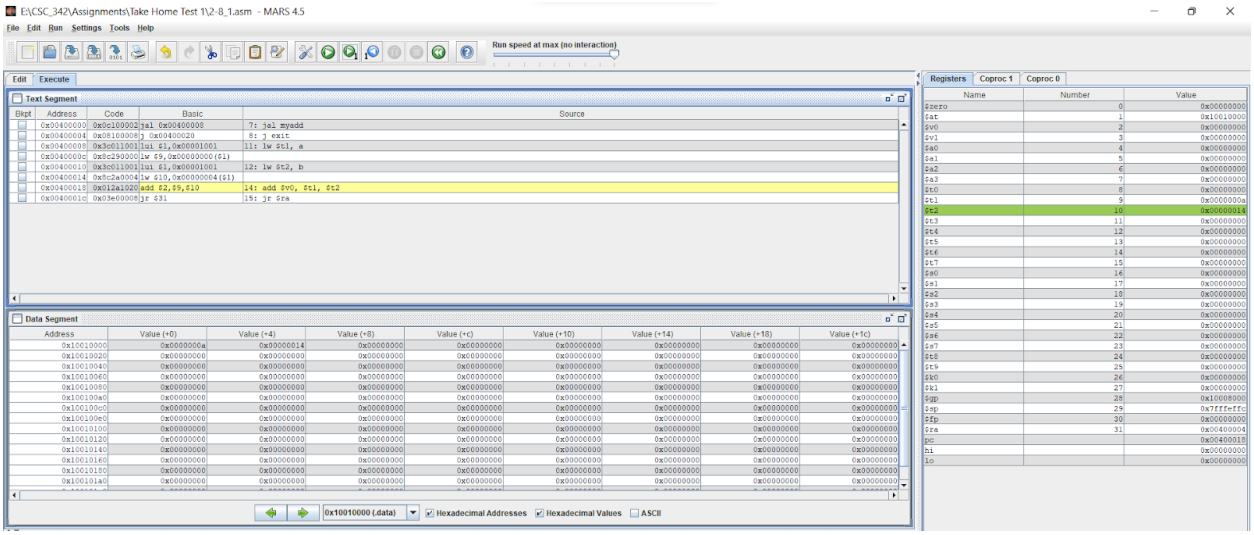

Figure 39: we can see the disassembly, memory, registers before arithmetic operations.

This statement creates a jump link between the myadd and main functions. Therefore, this allows you to call the myadd function. The add instruction calculates the sum of two numbers and stores them in $ v0. The jr statement returns from myadd function to the main function so that I can end the function call and end the code.

In figure 40, we can see the registers after arithmetic operations.

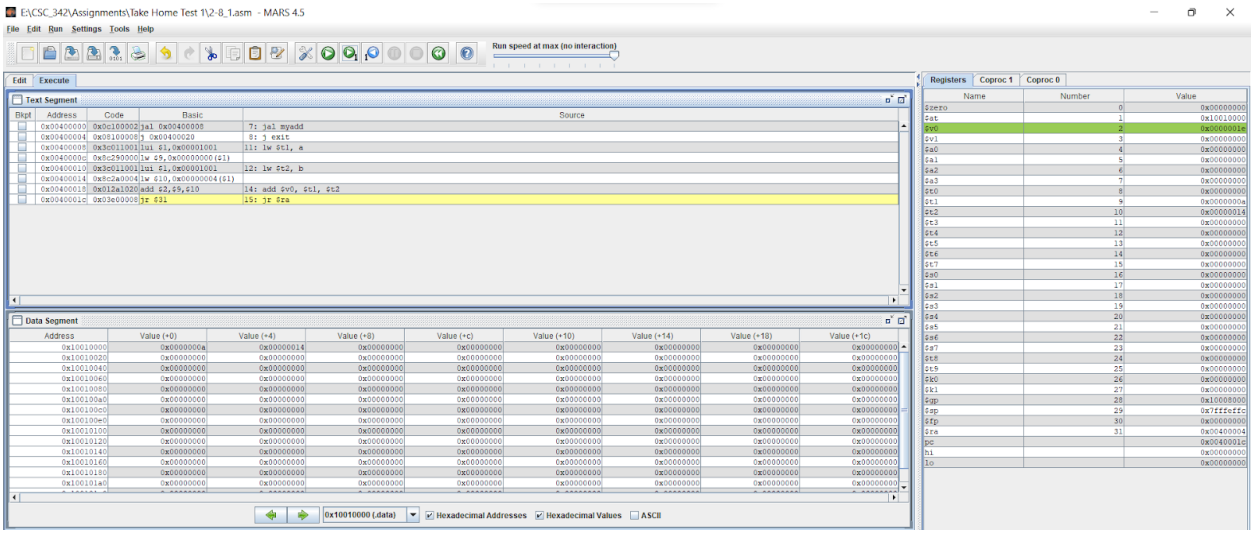

Figure 40: registers after arithmetic operations

We can see the register value stores under the name $v0 and address is 0x0000001e.

PART II: Visual Studio 2019 (Intel x86)

2-2_1:

In figure 41, we can see the project directory for the 2-2_1.c.

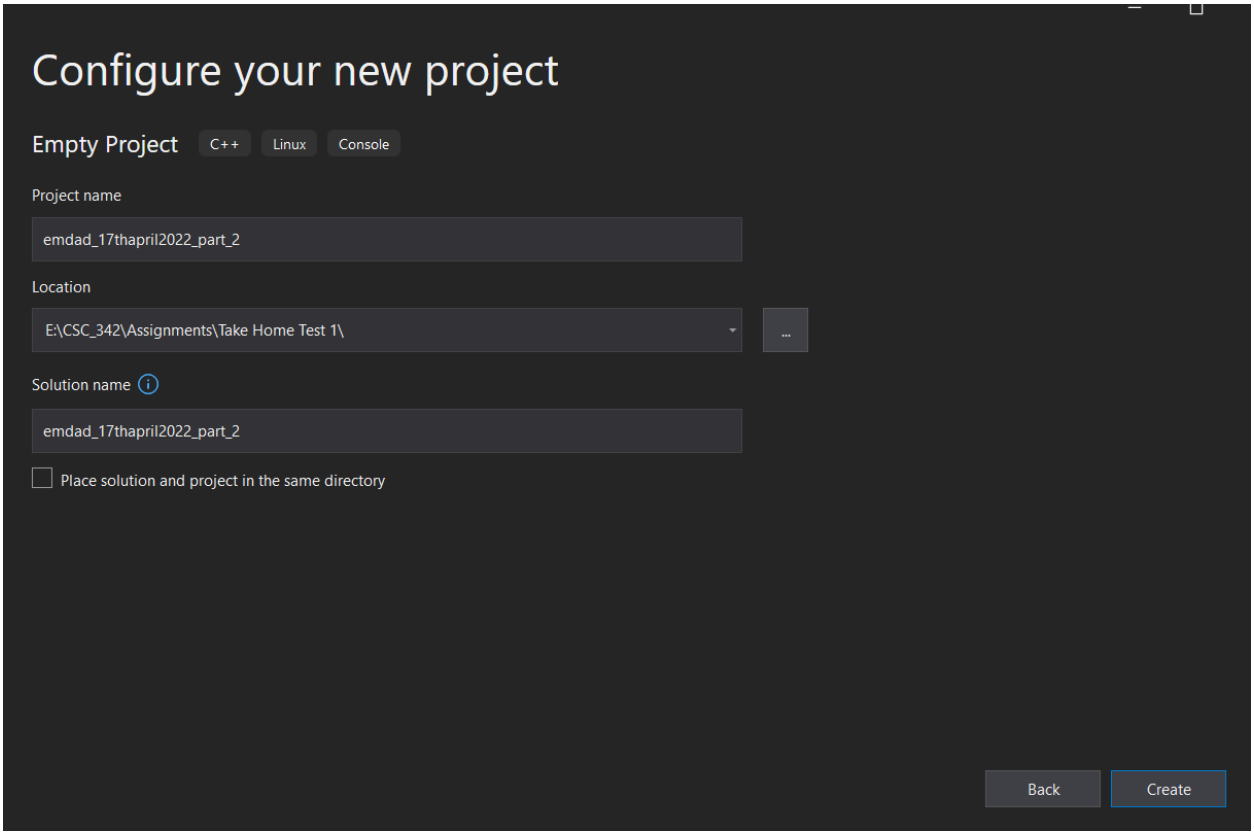

Figure 41: project directory for the 2-2_1.c.

In figure 42, we can see the file adding and naming 2-2_1.c it.

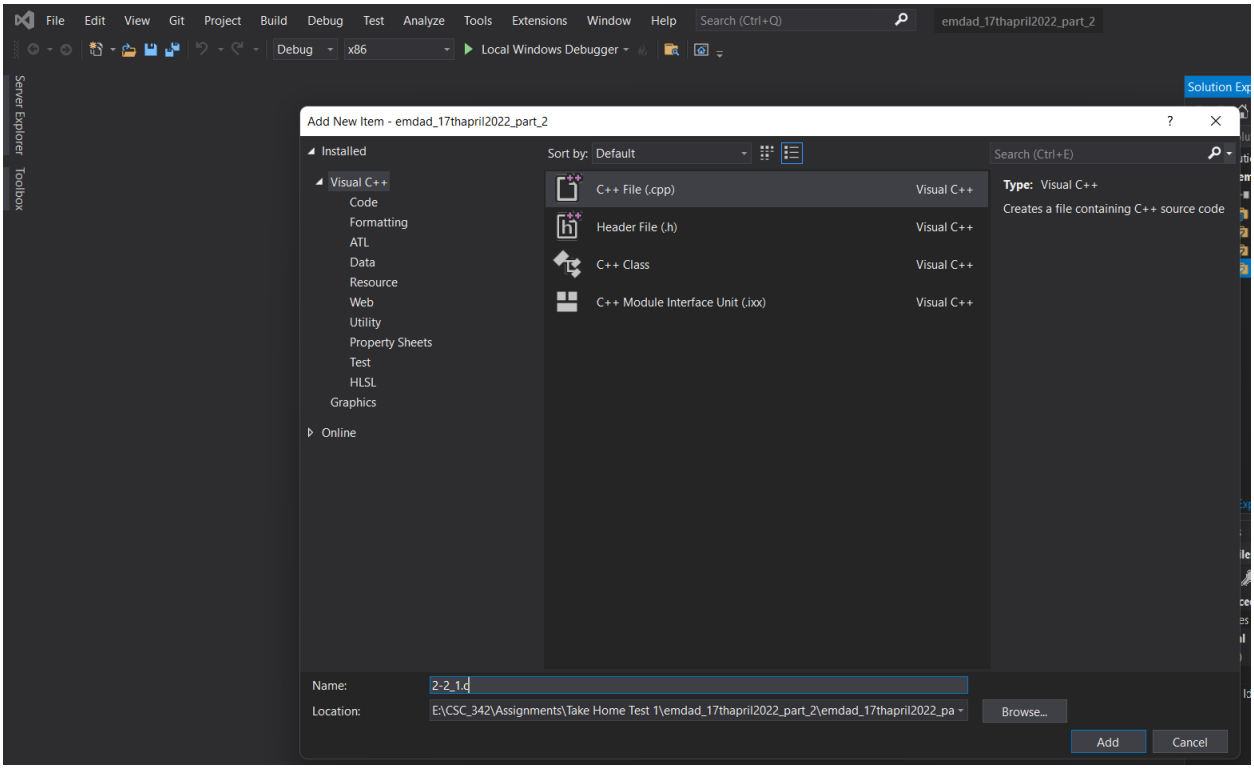

Figure 42: adding 2-2_1.c file in the project

In figure 43, we can see the source code for 2-2_1.c

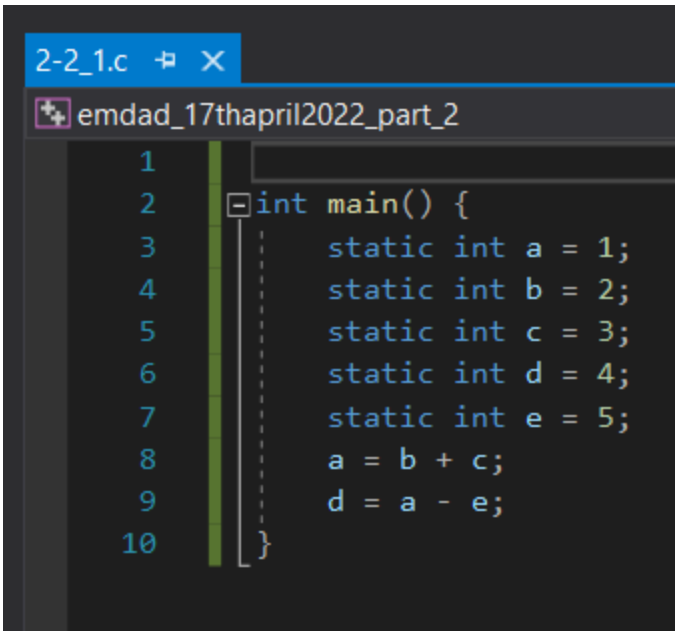

Figure 43: source code for 2-2_1.c

In figur 44, we can see that the project built successfully.

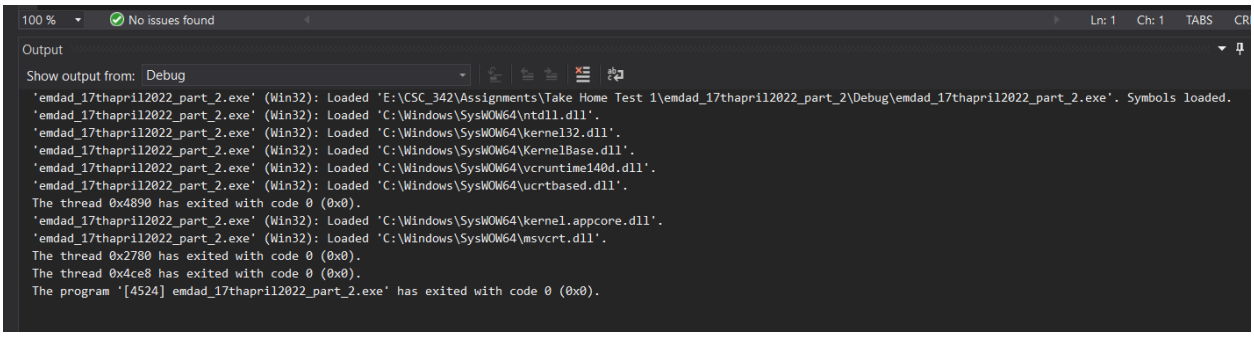

Figure 44: Built successfully

In figure 45, we can see the disassembly window code.

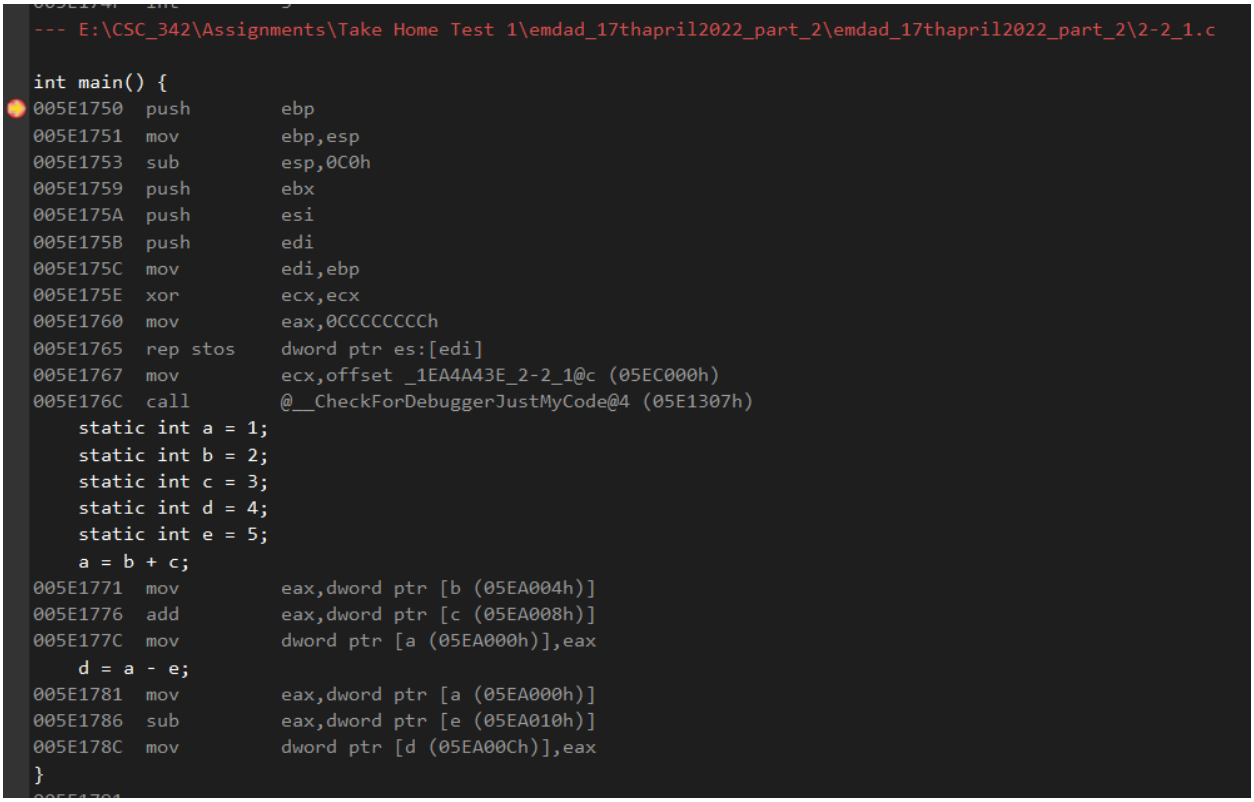

Figure 45: disassembly window code

The breakpoint arrow shows EBP is the base pointer for the current stack frame. So, the code starts off with setting a base pointer. The add instruction is done.

In figure 46, we can see the register window. In the register window, the EIP register contains the address of the next command to be executed. ESP is a stack pointer that points to the beginning of an address. As explained earlier, EBP indicates the top of the stack.

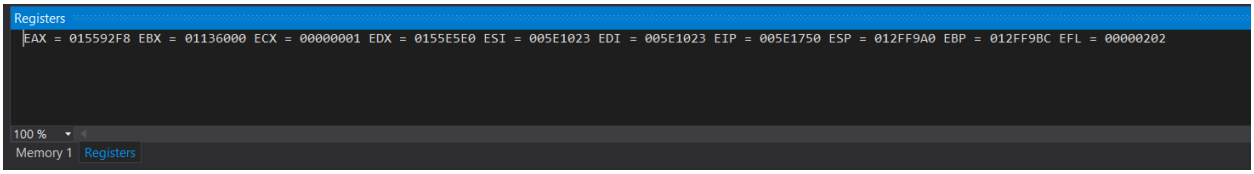

Figure 46: register window.

In figure 47, we can see the memory window and the address is 0x005E17E0. It is the memory of EBP, the instruction that executed in the code.

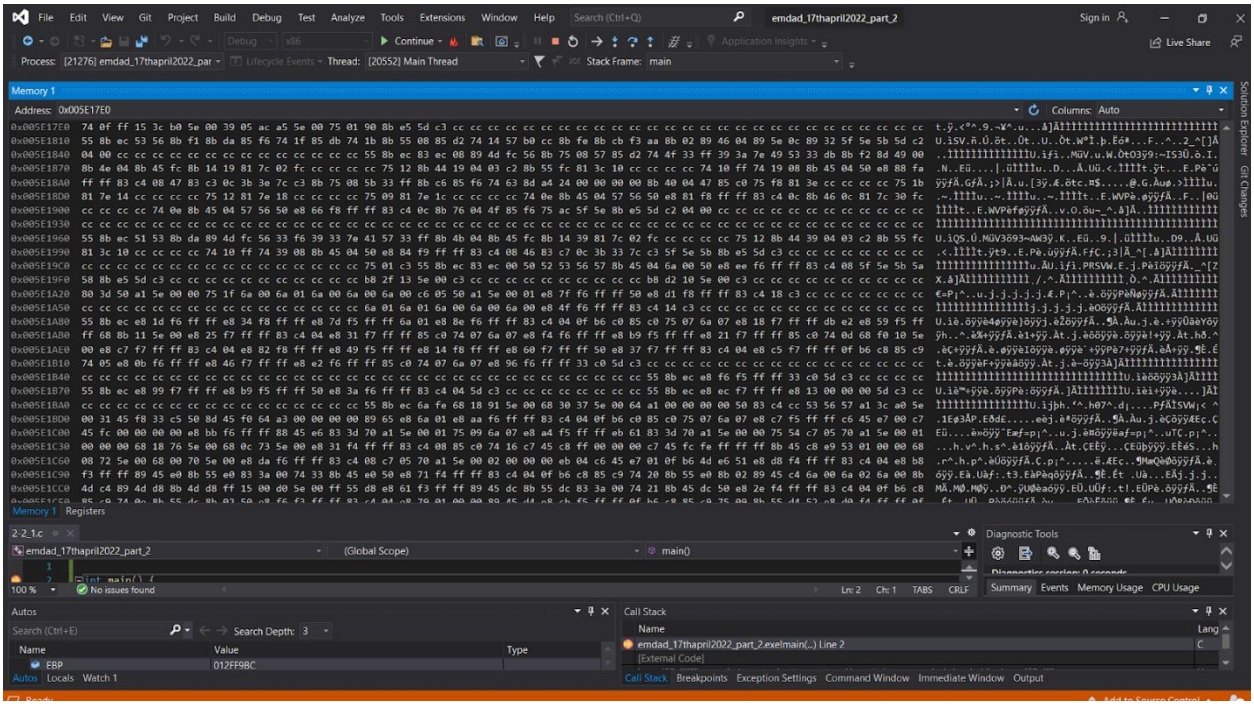

Figure 47: memory window

2-2_2:

In figure 41, we can see the source code for the 2-2_2.c.

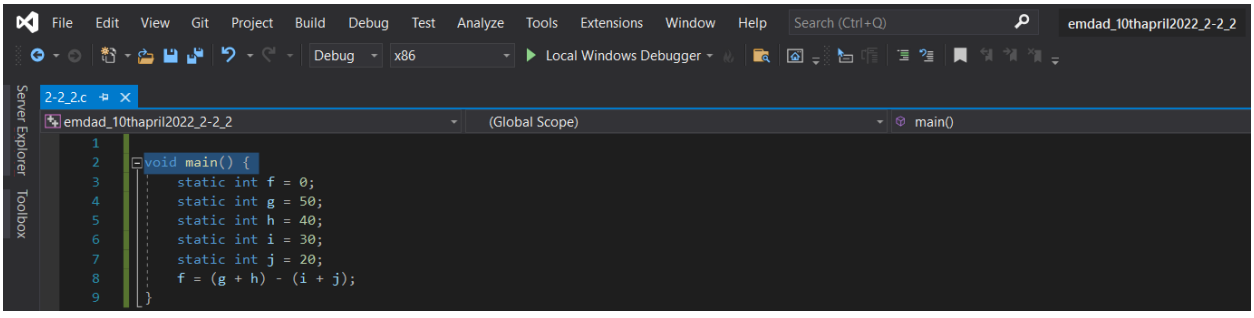

Figure 48: source code for the 2-2_2.c.

In figure 49, we can see that the project built successfully.

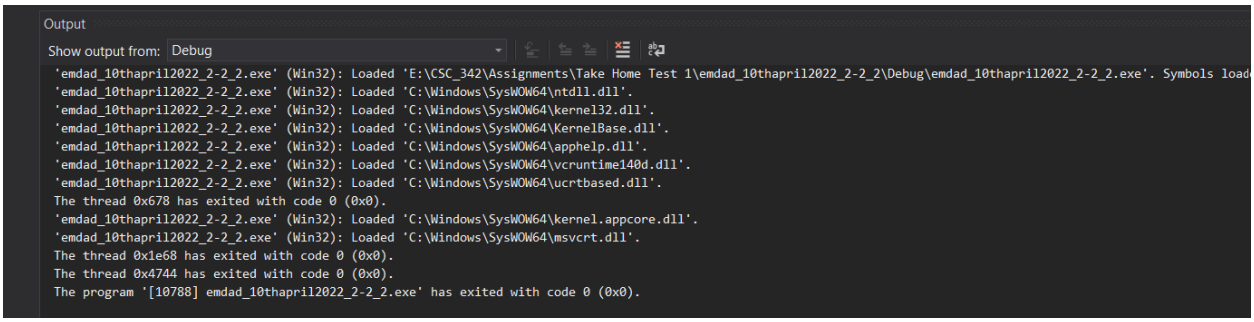

Figure 49: Built successfully

In figure 50, we can see the register window at the top, disassembly on the left and memory at the right side.

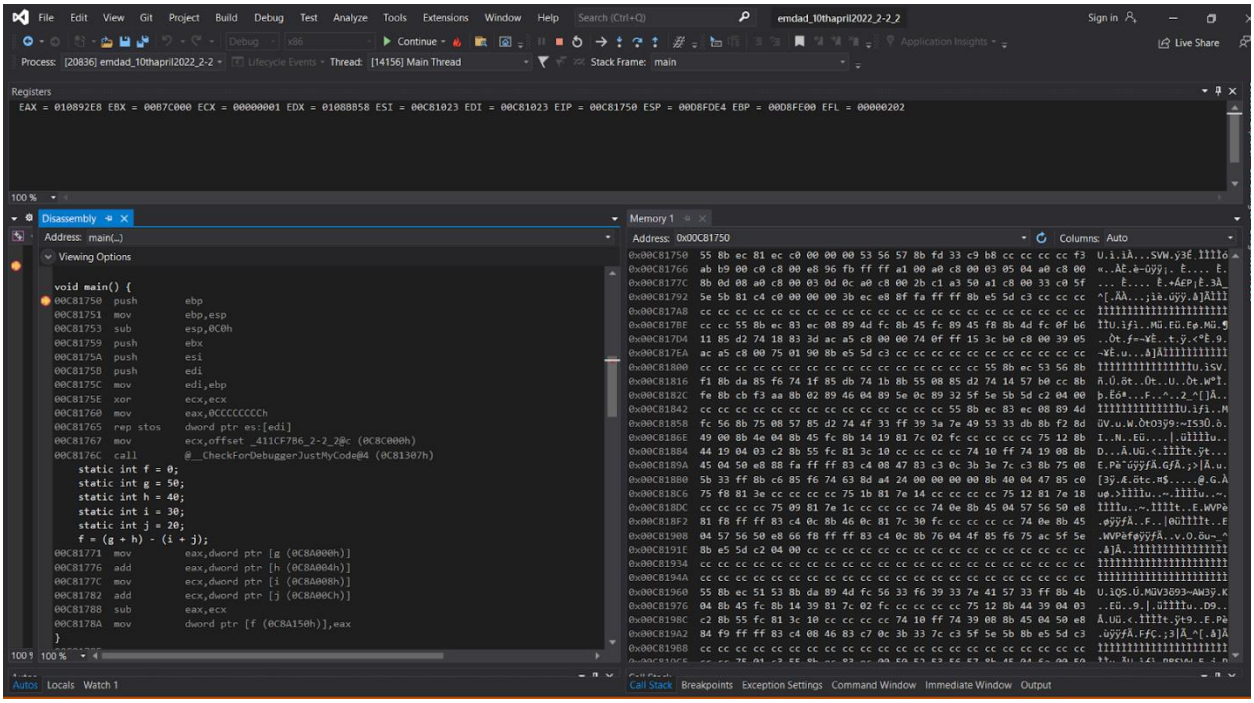

Figure 50: register, disassembly, memory window

We can see the static variables on the disassembly window. No assembler instructions are needed because these variables are placed in memory during the build. EAX: The value is not preserved to store the return value of the function. In the register window, the EIP register contains the address of the next command to be executed. ESP is a stack pointer that points to the beginning of an address. As explained earlier, EBP indicates the top of the stack. EBX: Not normally used and set to 0. ECX: Stores the loop count of the iterative program. You can also use the AX and DX registers in multiples. EDX: Often used for multiplication and division. EDI: Often used as a data destination or pointer. ESI: Same function as EDI register. EFL: This register stores various flags. We can see the memory window and the address is 0x00C81750. It is the memory of EBP, the instruction that executed in the code.

2-3_1:

In figure 51, we can see the source code for the 2-3_1.c.

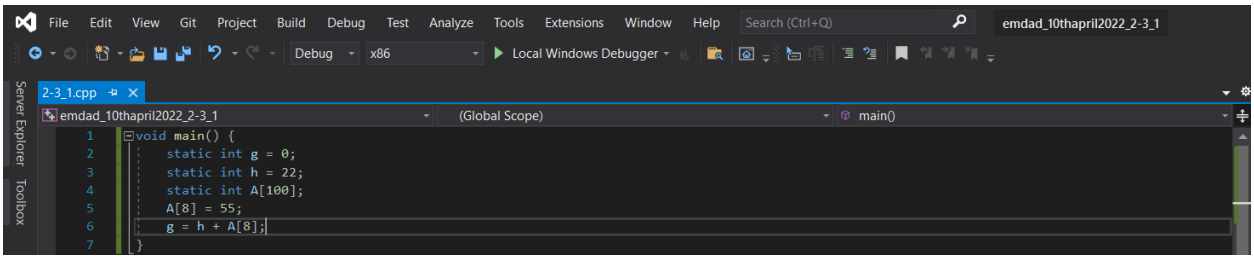

Figure 51: source code for the 2-3_1.c.

In figure 52, we can see that the project built successfully.

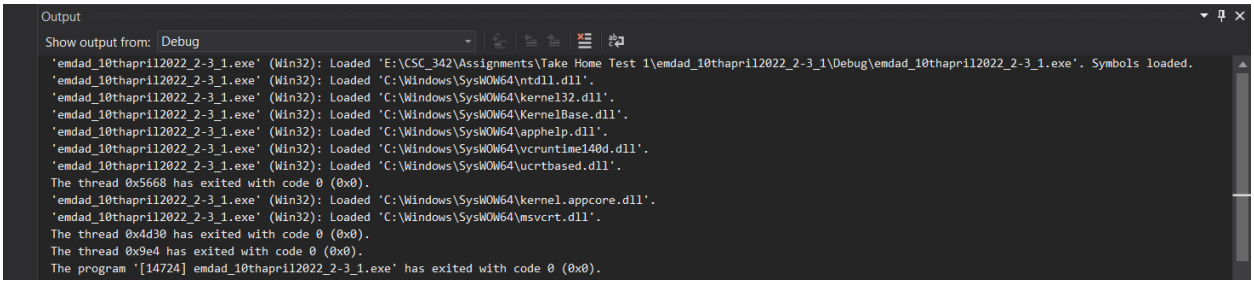

Figure 52: Built successfully

In figure 53, we can see the register window at the top, disassembly on the left and memory at the right side.

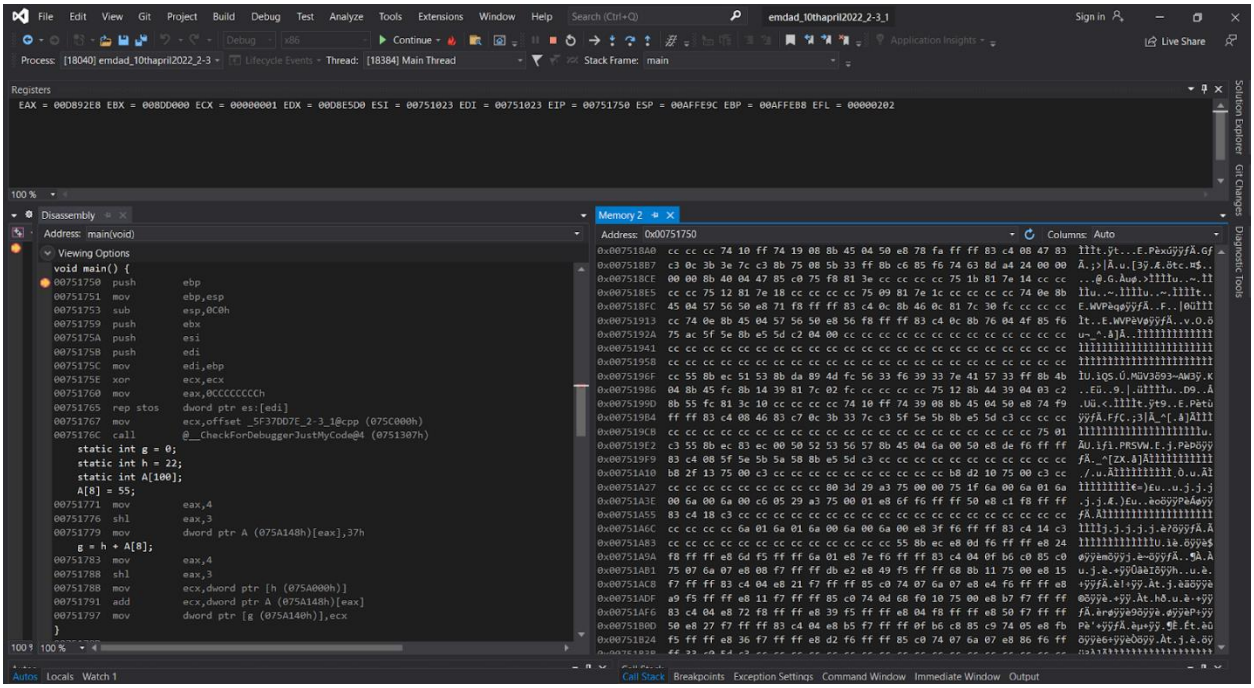

Figure 53: register, disassembly, memory window

We can see the static variables on the disassembly window. No assembler instructions are needed because these variables are placed in memory during the build. In the register window, the EIP register contains the address of the next command to be executed. ESP is a stack pointer that points to the beginning of an address. As explained earlier, EBP indicates the top of the stack. EBX: Not normally used and set to 0. ECX: Stores the loop count of the iterative program. You can also use the AX and DX registers in multiples. EDX: Often used for multiplication and division. EDI: Often used as a data destination or pointer. ESI: Same function as EDI register. EFL: This register stores various flags. We can see the memory window and the address is 0x00751750. It is the memory of EBP, the instruction that executed in the code.

2-3_2:

In figure 54, we can see the source code for the 2-3_2.c.

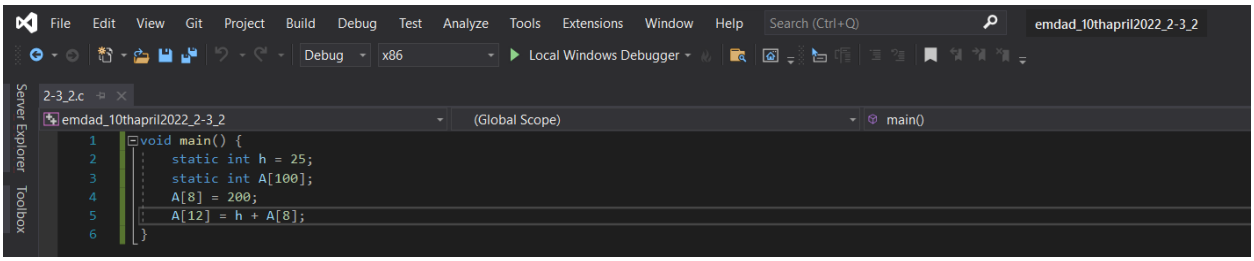

Figure 54: source code for the 2-3_2.c.

In figure 55, we can see that the project built successfully.

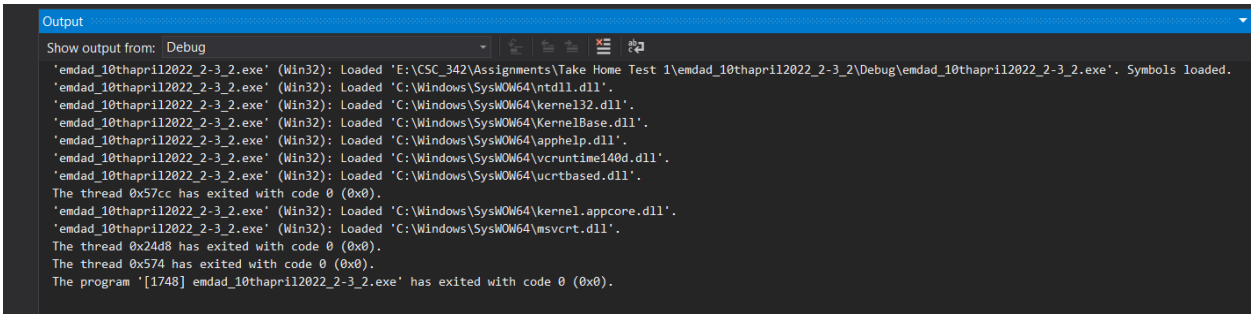

Figure 55: Built successfully

In figure 56, we can see the register window at the top, disassembly on the left and memory at the right side.

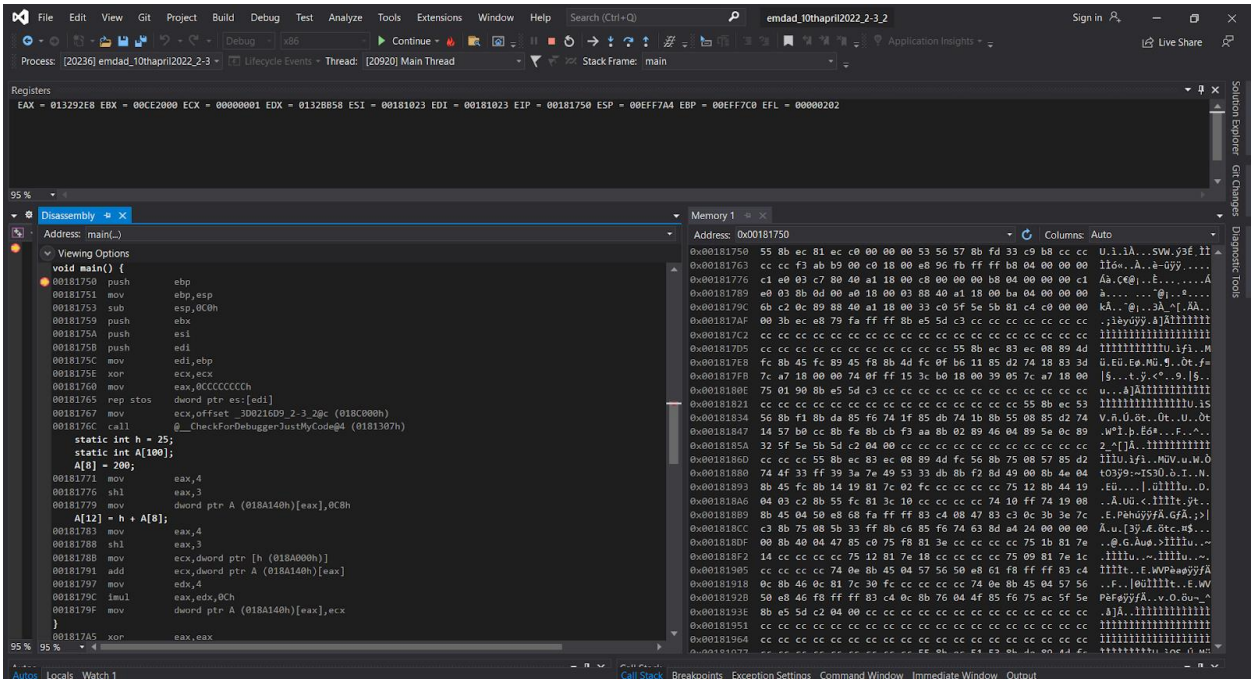

Figure 56: register, disassembly, memory window

We can see the static variables on the disassembly window. No assembler instructions are needed because these variables are placed in memory during the build. In the register window, the EIP register contains the address of the next command to be executed. ESP is a stack pointer that points to the beginning of an address. As explained earlier, EBP indicates the top of the stack. EBX: Not normally used and set to 0. ECX: Stores the loop count of the iterative program. You can also use the AX and DX registers in multiples. EDX: Often used for multiplication and division. EDI: Often used as a data destination or pointer. ESI: Same function as EDI register. EFL: This register stores various flags. We can see the memory window and the address is 0x00C81750. It is the memory of EBP, the instruction that executed in the code.

2-5_1:

In figure 57, we can see the source code for the 2-5_1.c.

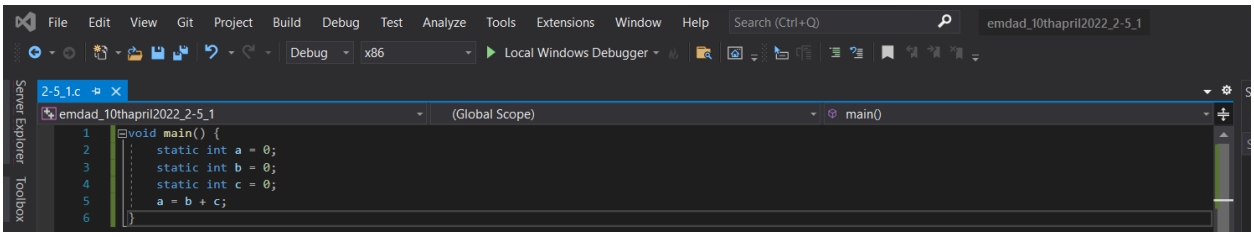

Figure 57: source code for the 2-5_1.c.

In figure 58, we can see that the project built successfully.

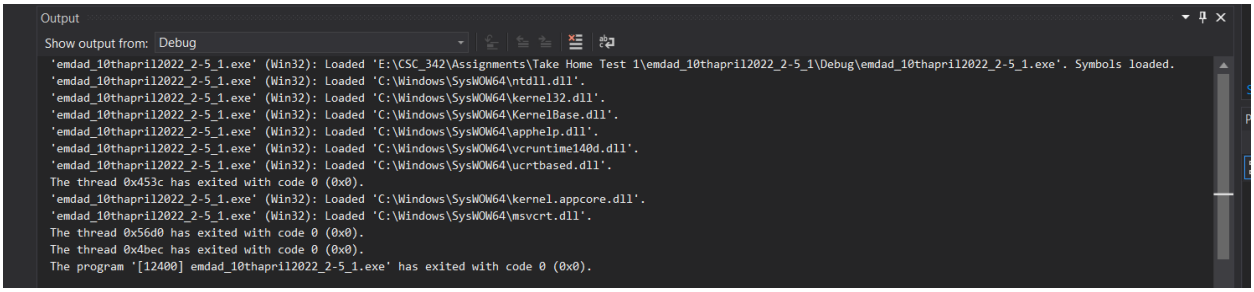

Figure 58: Built successfully

In figure 59, we can see the register window at the top, disassembly on the left and memory at the right side.

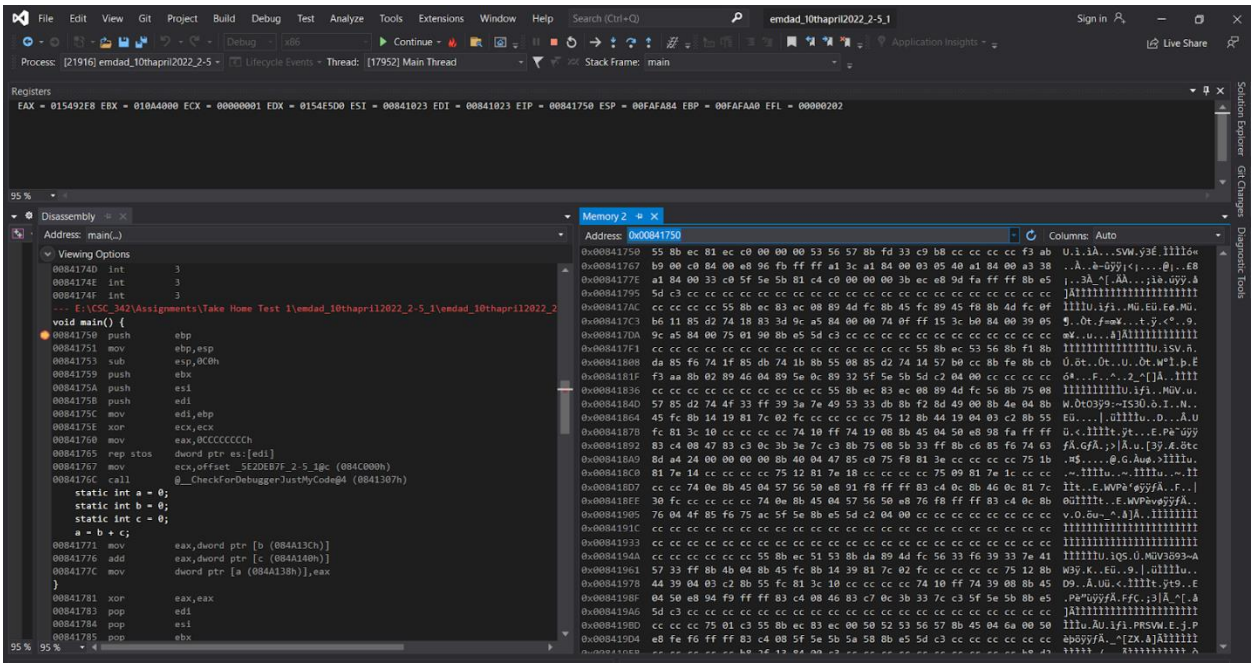

Figure 59: register, disassembly, memory window

We can see the static variables on the disassembly window. The address next to the debugger pointer which says “b (084A13Ch)” is the address for our static variable b. In the register window, the EIP register contains the address of the next command to be executed. ESP is a stack pointer that points to the beginning of an address. As explained earlier, EBP indicates the top of the stack. EBX: Not normally used and set to 0. ECX: Stores the loop count of the iterative program. You can also use the AX and DX registers in multiples. EDX: Often used for multiplication and division. EDI: Often used as a data destination or pointer. ESI: Same function as EDI register. EFL: This register stores various flags. We can see the memory window and the address is 0x00841750. It is the memory of EBP, the instruction that executed in the code.

2-6_1:

In figure 60, we can see the source code for the 2-6_1.c.

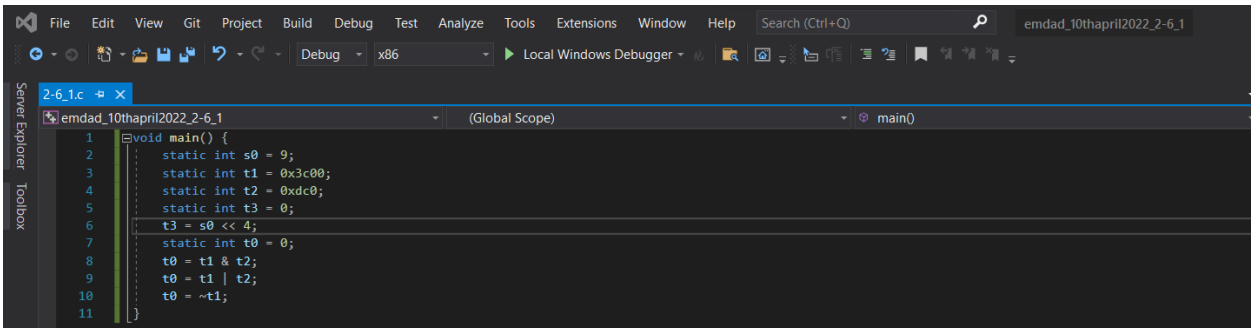

Figure 60: source code for the 2-6_1.c.

In figure 61, we can see that the project built successfully.

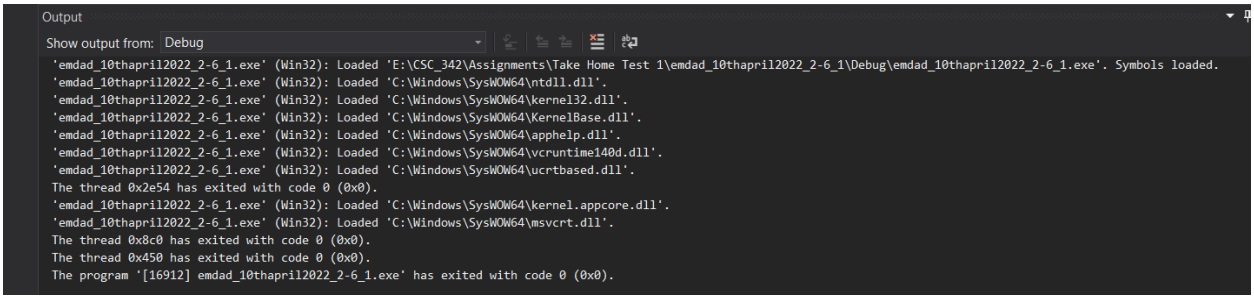

Figure 61: Built successfully

In figure 62, we can see the register window at the top, disassembly on the left and memory at the right side.

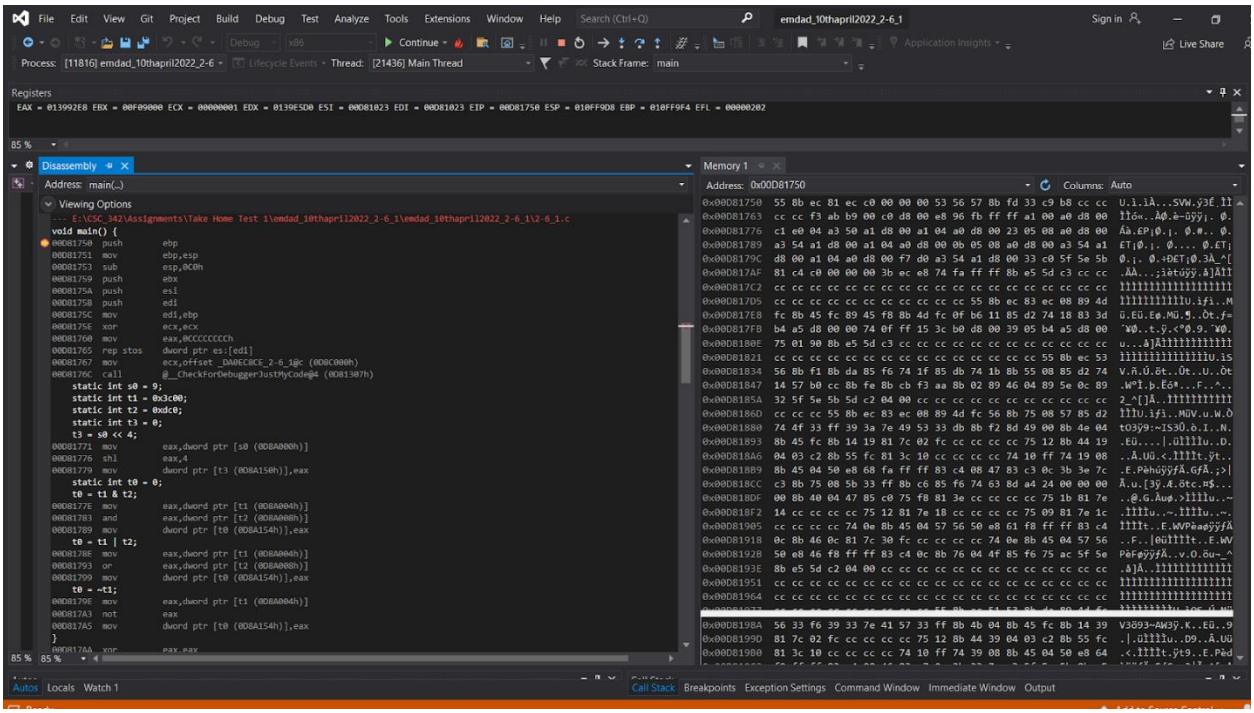

Figure 62: register, disassembly, memory window

We can see the static variables on the disassembly window. The “shl” instruction is the shift left instructions that we do in our bit wise operation. [s0 (0D08A000h)] is the little endian which is the least significant value in the hexadecimal is stored at the lowest bit. In the register window, the EIP register contains the address of the next command to be executed. ESP is a stack pointer that points to the beginning of an address. As explained earlier, EBP indicates the top of the stack. EBX: Not normally used and set to 0. ECX: Stores the loop count of the iterative program. You can also use the AX and DX registers in multiples. EDX: Often used for multiplication and division. EDI: Often used as a data destination or pointer. ESI: Same function as EDI register. EFL: This register stores various flags. We can see the memory window and the address is 0x00D81750. It is the memory of EBP, the instruction that executed in the code.

2-6_2:

In figure 63, we can see the source code for the 2-6_2.c.

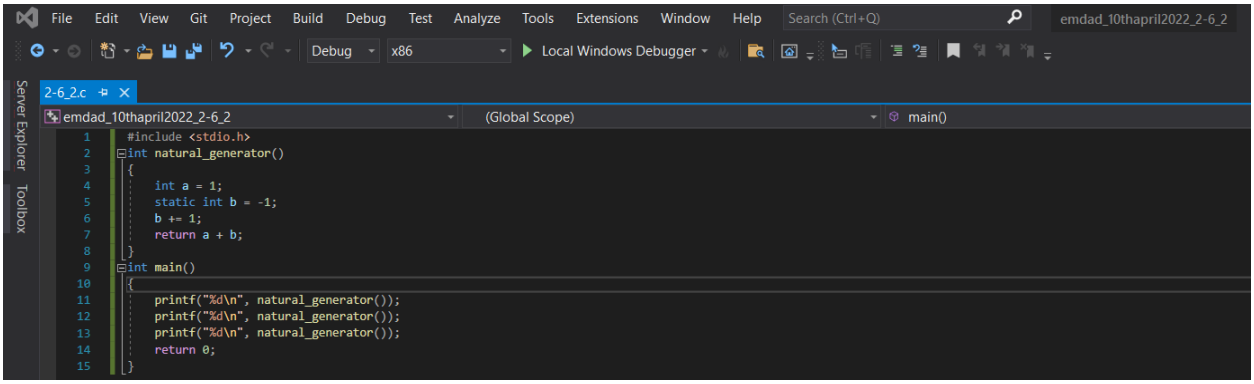

Figure 63: source code for the 2-6_2.c.

In figure 64, we can see that the project built successfully.

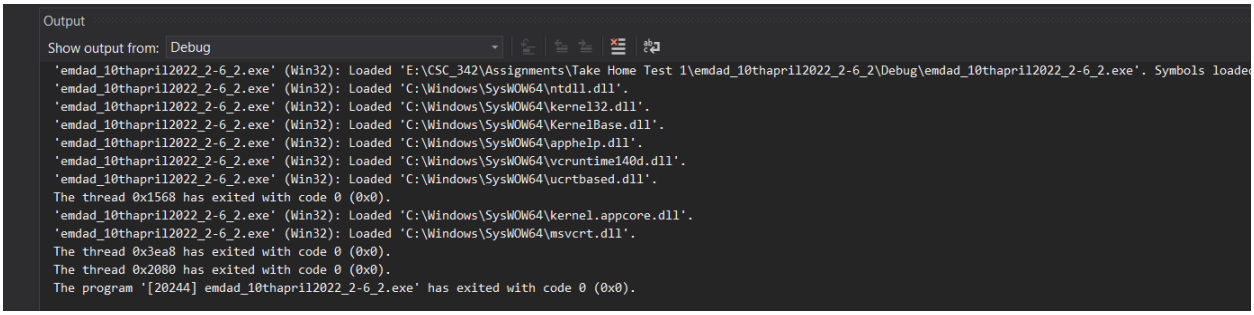

Figure 64: Built successfully

In figure 65, we can see the register window at the top, disassembly on the left and memory at the right side.

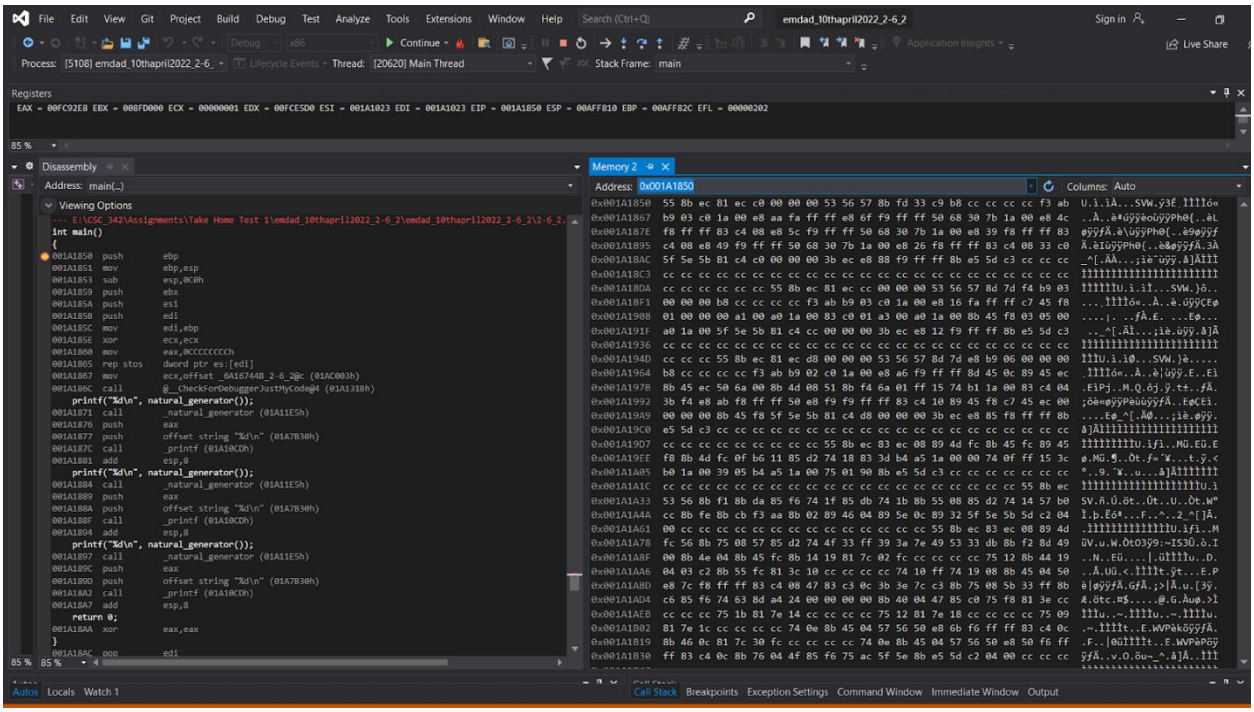

Figure 65: register, disassembly, memory window

We can see the disassembly window; no assembler instructions are needed because these variables are placed in memory during the build. There is no local variable created so local is empty. In the register window, the EIP register contains the address of the next command to be executed. ESP is a stack pointer that points to the beginning of an address. As explained earlier, EBP indicates the top of the stack. EBX: Not normally used and set to 0. ECX: Stores the loop count of the iterative program. You can also use the AX and DX registers in multiples. EDX: Often used for multiplication and division. EDI: Often used as a data destination or pointer. ESI: Same function as EDI register. EFL: This register stores various flags. We can see the memory window and the address is 0x001A1850. It is the memory of EBP, the instruction that executed in the code.

2-7_1:

In figure 66, we can see the source code for the 2-7_1.c.

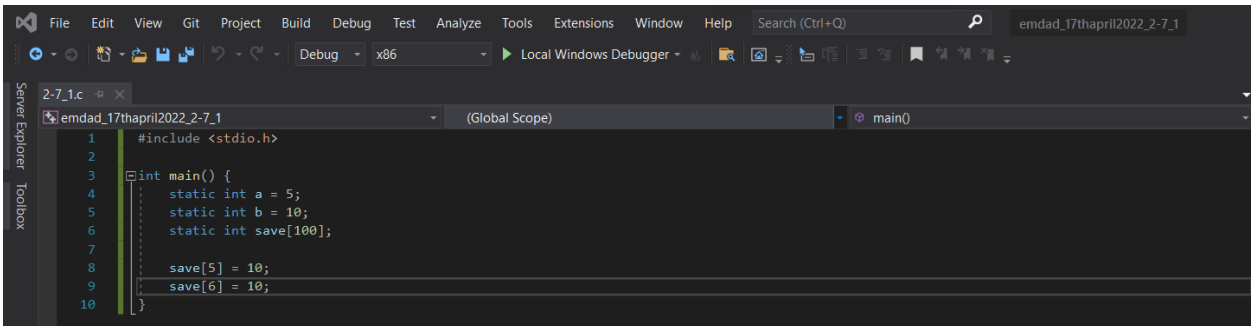

Figure 66: source code for the 2-7_1.c.

In figure 67, we can see that the project built successfully.

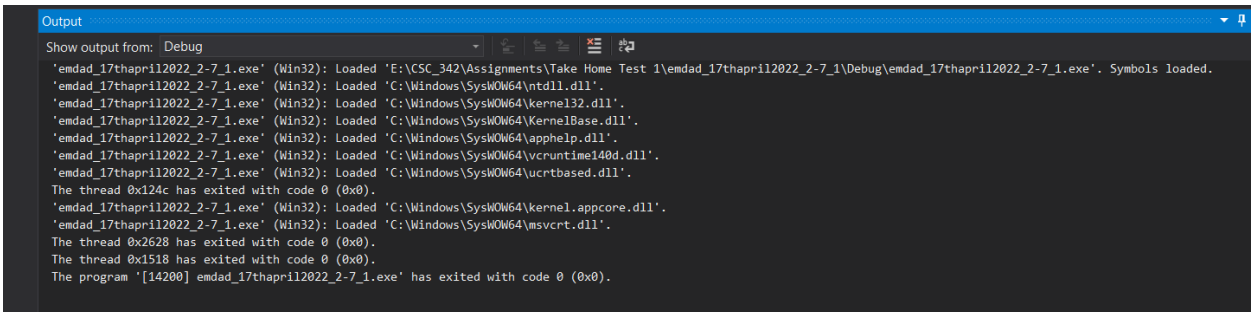

Figure 67: Built successfully

In figure 68, we can see the register window at the top, disassembly on the left and memory at the right side.

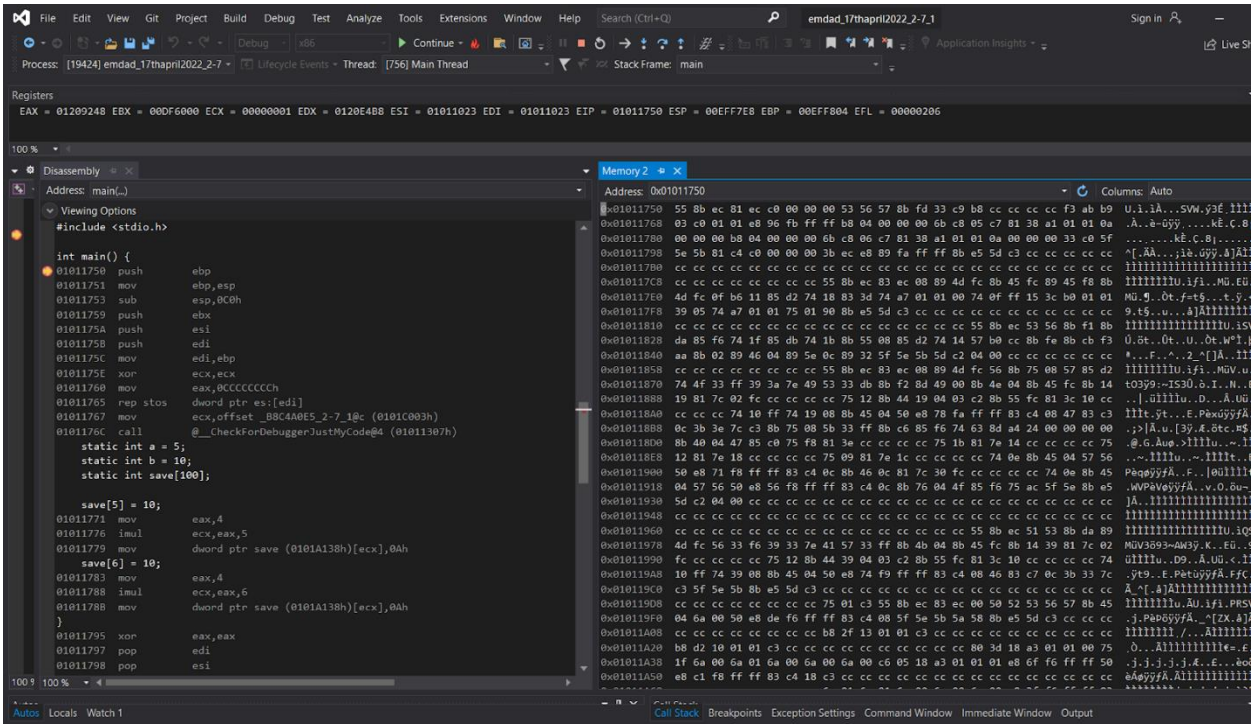

Figure 68: register, disassembly, memory window

We can see the static variables on the disassembly window. In the register window, the EIP register contains the address of the next command to be executed. ESP is a stack pointer that points to the beginning of an address. As explained earlier, EBP indicates the top of the stack. EBX: Not normally used and set to 0. ECX: Stores the loop count of the iterative program. You can also use the AX and DX registers in multiples. EDX: Often used for multiplication and division. EDI: Often used as a data destination or pointer. ESI: Same function as EDI register. EFL: This register stores various flags. We can see the memory window and the address is 0x001011750. It is the memory of EBP, the instruction that executed in the code.

2-8_1:

In figure 41, we can see the source code for the 2-8_1.c.

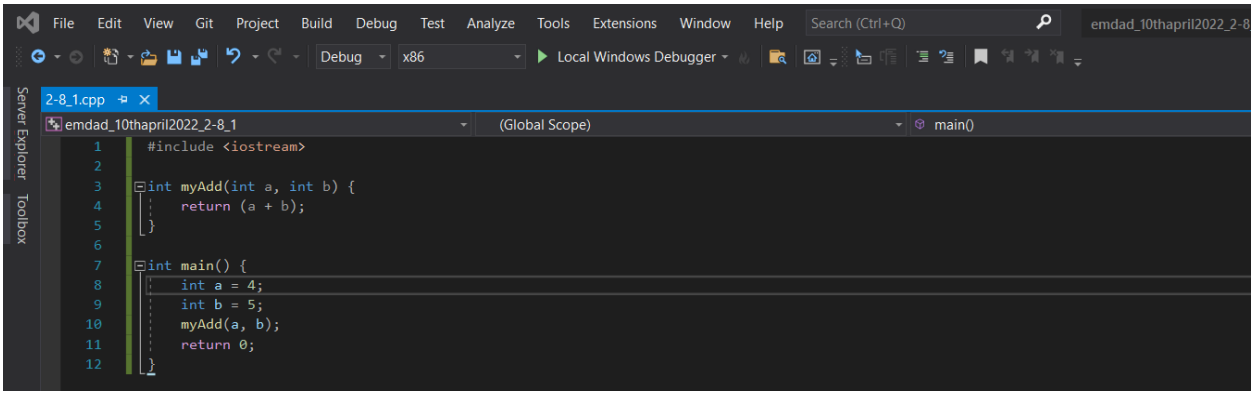

Figure 69: source code for the 2-8_1.c.

In figure 44, we can see that the project built successfully.

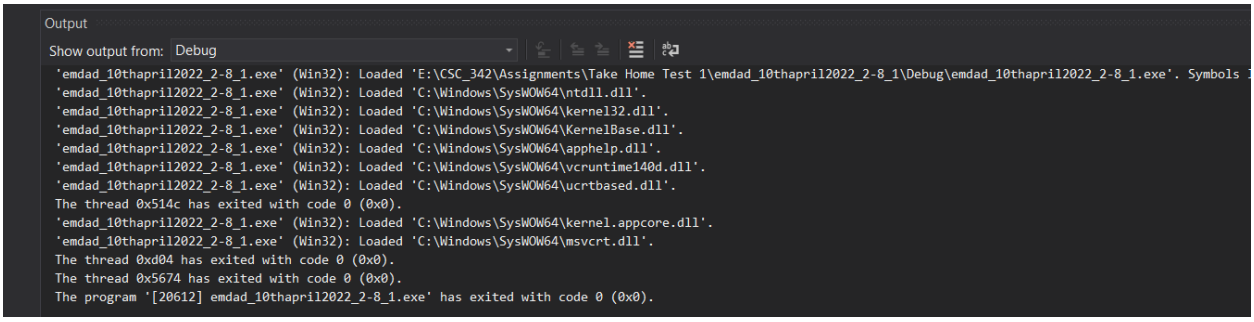

Figure 70: Built successfully

In figure 50, we can see the register window at the top, disassembly on the left and memory at the right side.

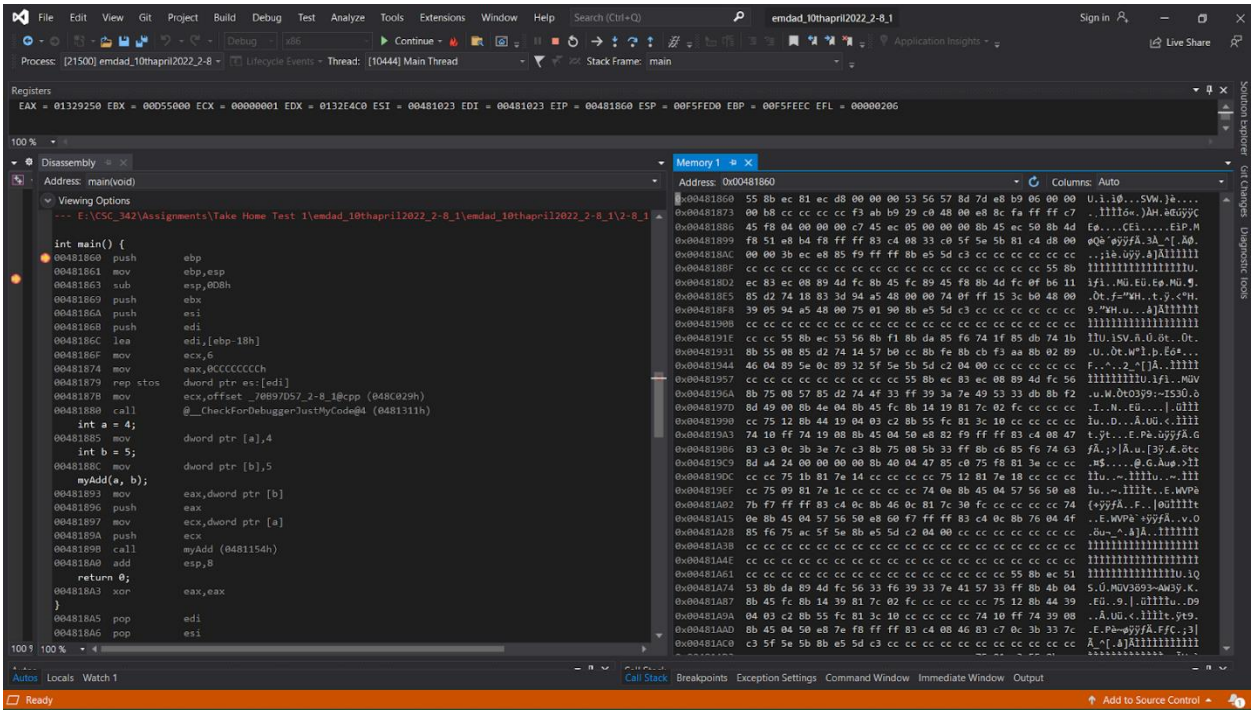

Figure 71: register, disassembly, memory window

We can see on the disassembly window, the call instruction making a call to the function myAdd. In the register window, the EIP register contains the address of the next command to be executed. ESP is a stack pointer that points to the beginning of an address. As explained earlier, EBP indicates the top of the stack. EBX: Not normally used and set to 0. ECX: Stores the loop count of the iterative program. You can also use the AX and DX registers in multiples. EDX: Often used for multiplication and division. EDI: Often used as a data destination or pointer. ESI: Same function as EDI register. EFL: This register stores various flags. We can see the memory window and the address is 0x00481860. It is the memory of EBP, the instruction that executed in the code.

PART III: x86_64 ISA, Linux 64-bit

2-2_1:

In figure 41, we can see the source code for the 2-2_1.c in linux editor.

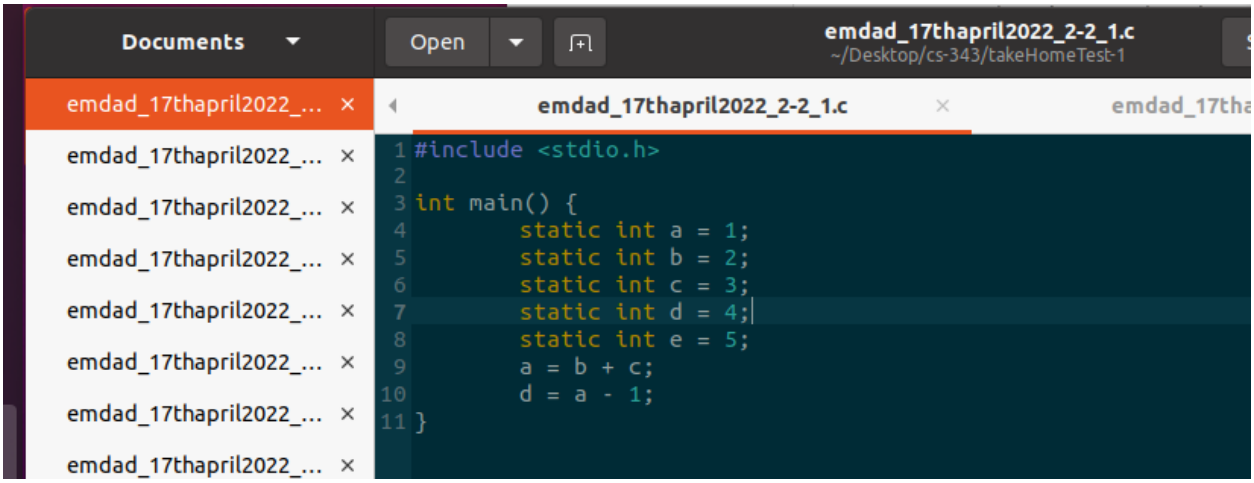

Figure 72: source code for the 2-2_1.c

The main function started at line 3 and then there are some variables initialized and finally the code performs two arithmatic operations. One is addition (a=b+c) and other one is summation (d=a-1).

In figure 73, we can see that the code output, disassemble, info registers.

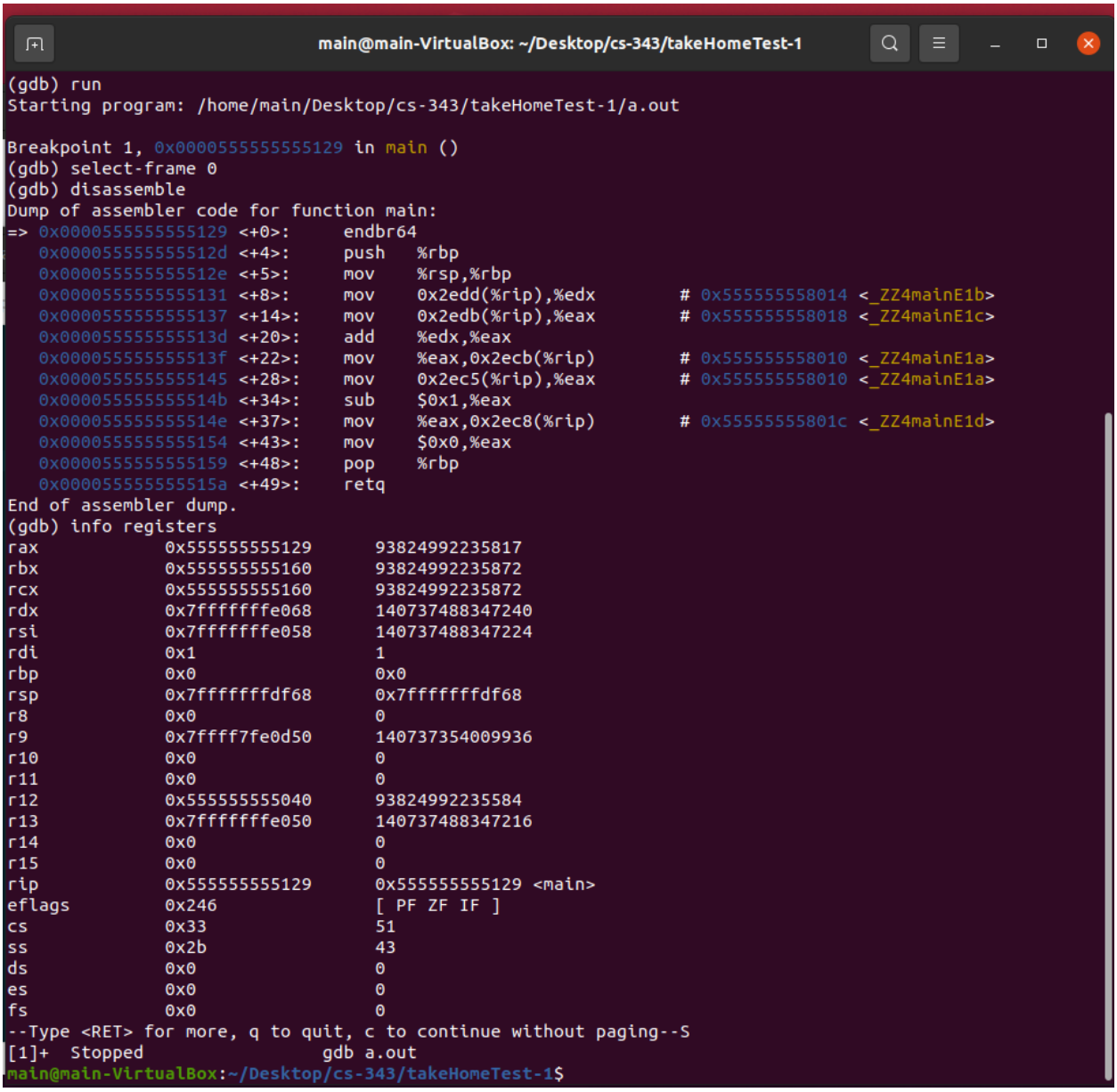

Figure 73: code output, disassemble, info registers.

First, I did gdb run to start the program and then b main to put a breakpoint at the main function line. After running the disassemble code, we get the assembler code of function main. 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. The mov instruction stores all local values in registers. The 64-bit register contains the base pointer which is pushed onto the stack and value get copied is $rsp in $rbp.

2-2_2:

In figure 74, we can see the source code for the 2-2_2.c in linux editor.

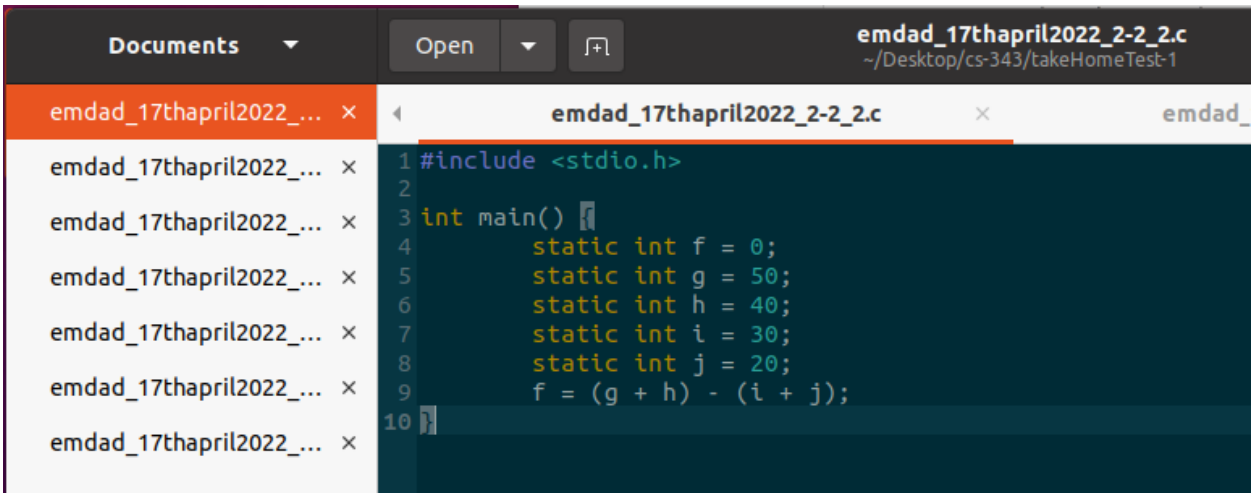

Figure 74: source code for the 2-2_2.c

The main function started at line 3 and then there are some variables initialized and finally the code performs two arithmatic operations. One is addition (g+h) and other one is summation.

In figure 75, we can see that the code output, disassemble, info registers.

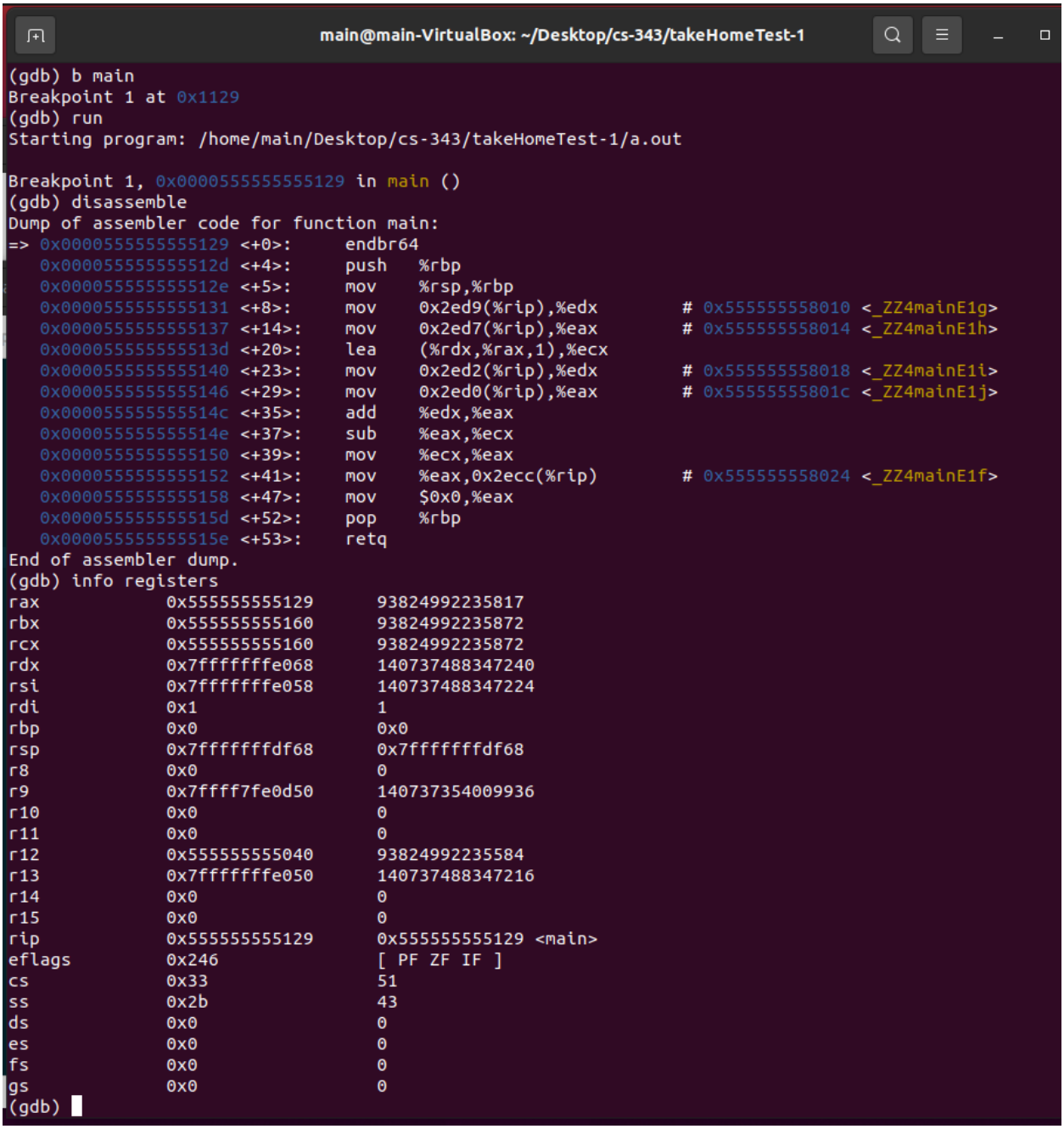

Figure 75: code output, disassemble, info registers.

First, I did gdb run to start the program and then b main to put a breakpoint at the main function line. 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.

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. The mov instruction stores all local values in registers.

2-3_1:

In figure 76, we can see the source code for the 2-3_1.c in linux editor.

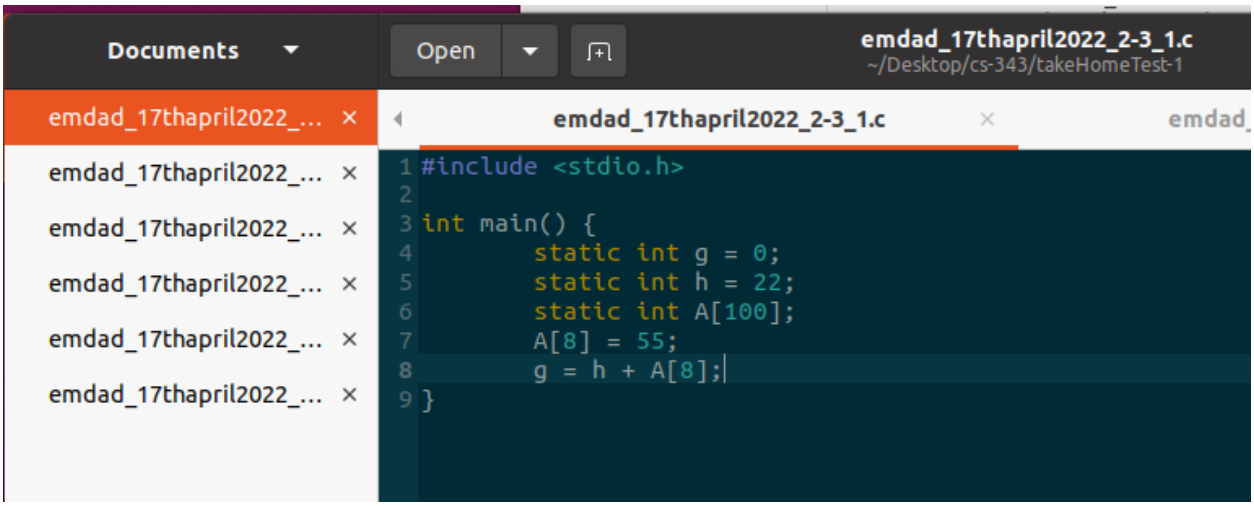

Figure 76: source code for the 2-3_1.c

The main function started at line 3 and then there are some variables initialized and finally the code performs arithmetics operations: addition and subtraction on static variables and arrays initialized.

In figure 77, we can see that the code output, disassemble, info registers.

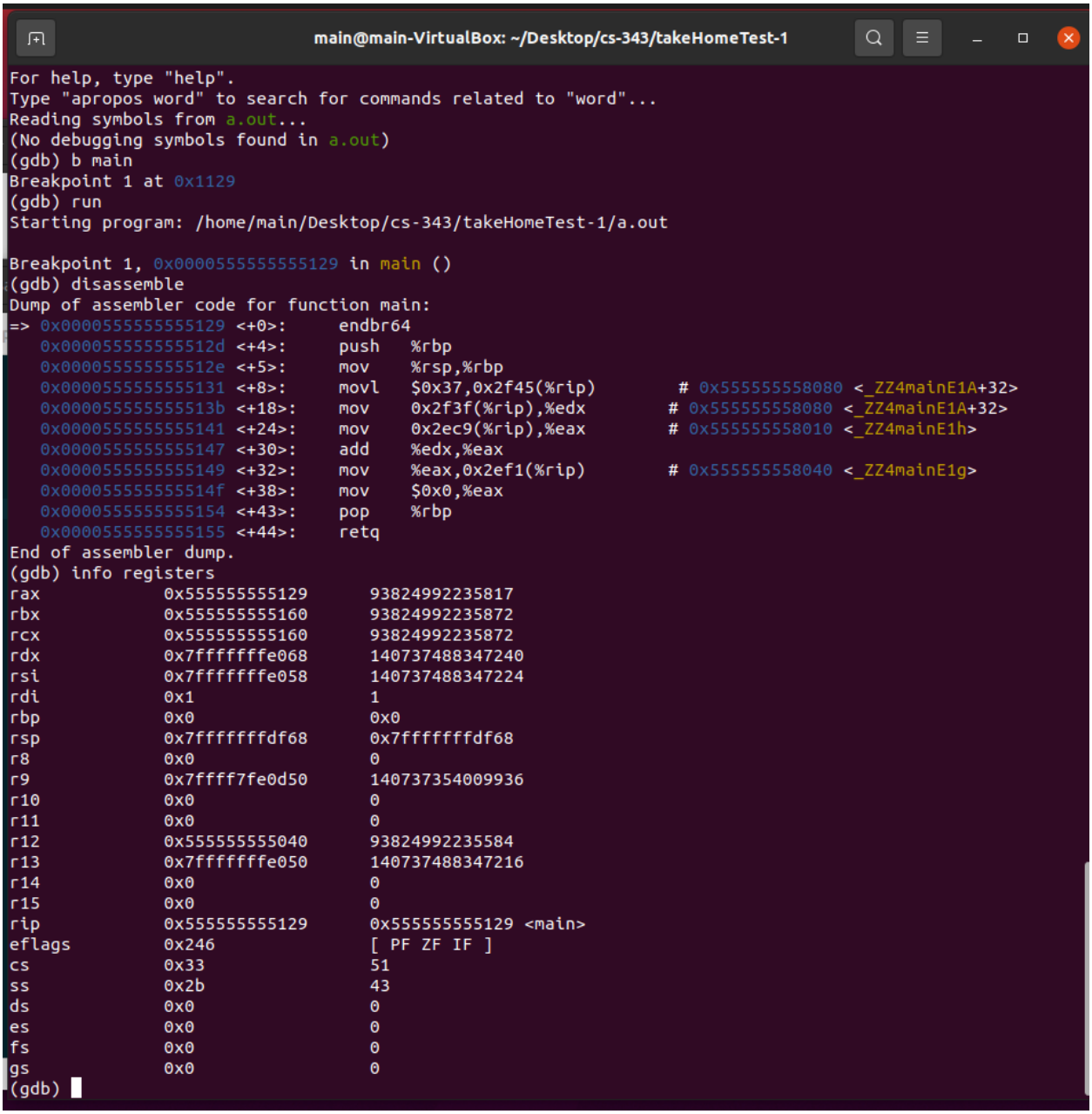

Figure 77: code output, disassemble, info registers.

First, I did gdb run to start the program and then b main to put a breakpoint at the main function line. 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. The value 0x37 is moved to location 0x555555558060. This is the location of A [8], which is an offset of 32 from the base array address. A [8] now contains 0x37 (55). Memory address 0x555555558010 contains 0x16. The data from memory is copied to two registers and added between them. The result is then saved in memory. 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. The mov instruction stores all local values in registers.

2-3_2:

In figure 78, we can see the source code for the 2-3_2.c in linux editor.

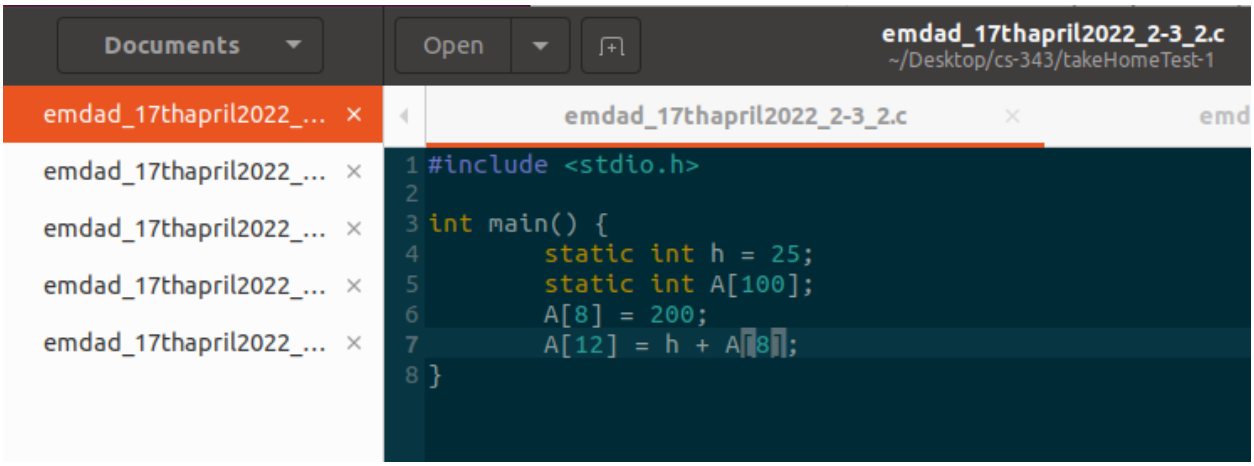

Figure 78: source code for the 2-3_2.c

The main function started at line 3 and then there are some variables initialized and finally the code performs arithmetics operations: addition and subtraction on static variables and arrays initialized.

In figure 79, we can see that the code output, disassemble, info registers.

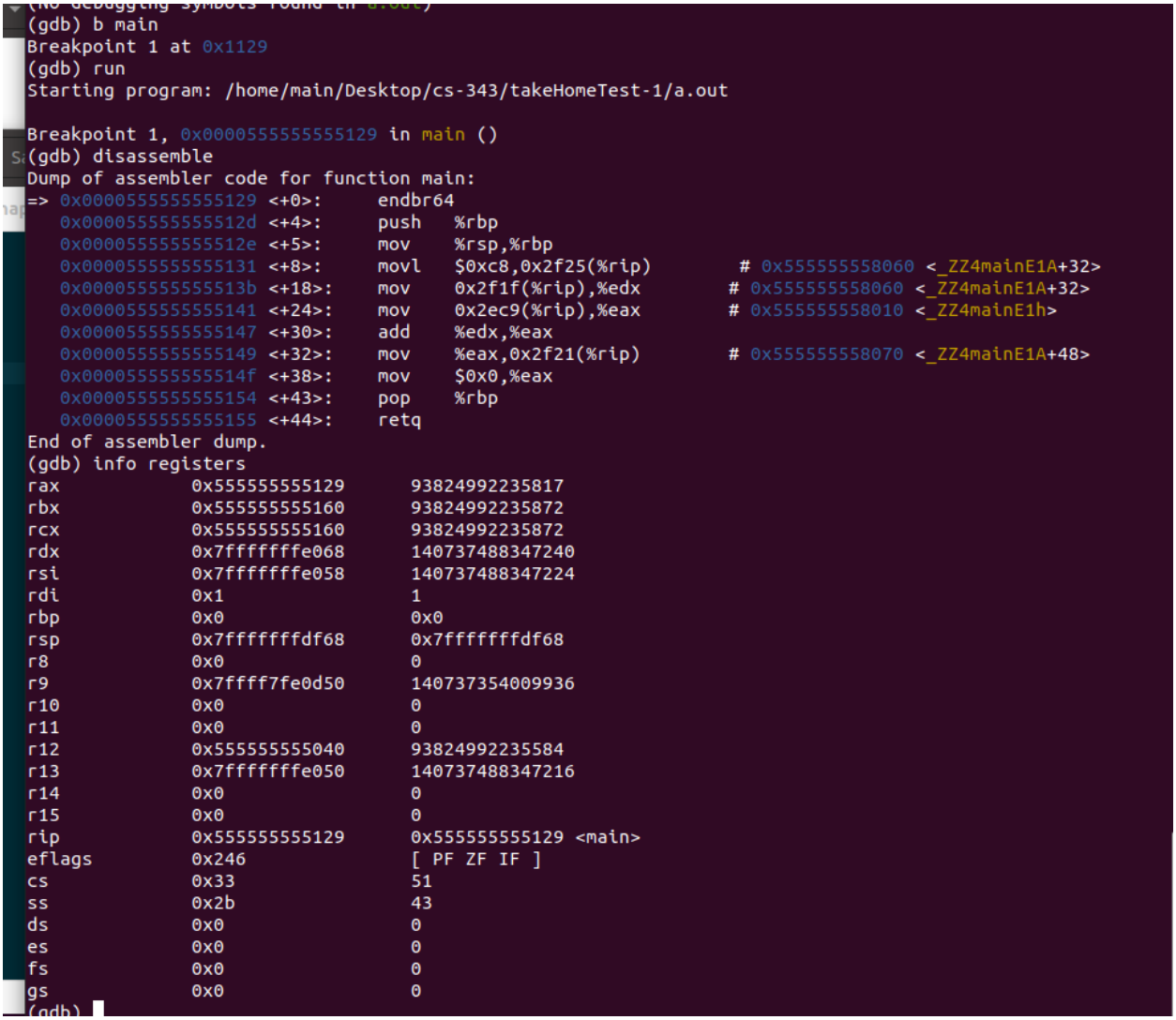

Figure 79: code output, disassemble, info registers.

First, I did gdb run to start the program and then b main to put a breakpoint at the main function line. 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. The value 0x37 is moved to location 0x555555558060. This is the location of A [8], which is an offset of 32 from the base array address. A [8] now contains 0x37 (55). Memory address 0x555555558010 contains 0x16. The data from memory is copied to two registers and added between them. The result is then saved in memory. 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. The mov instruction stores all local values in registers.

2-5_1:

In figure 80, we can see the source code for the 2-5_1.c in linux editor.

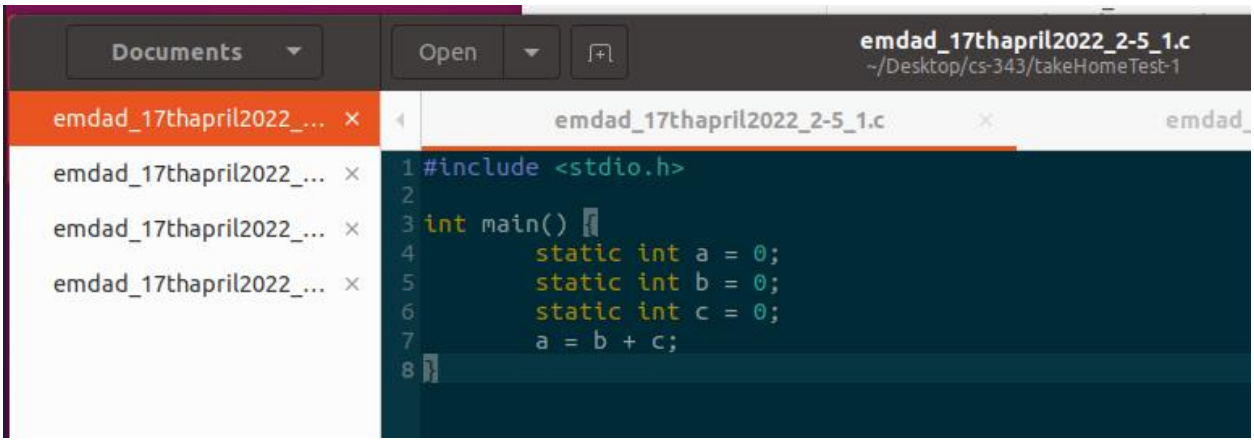

Figure 80: source code for the 2-5_1.c

The main function started at line 3 and then there are some variables initialized and finally the code performs arithmatic operations.

In figure 81, we can see that the code output, disassemble, info registers.

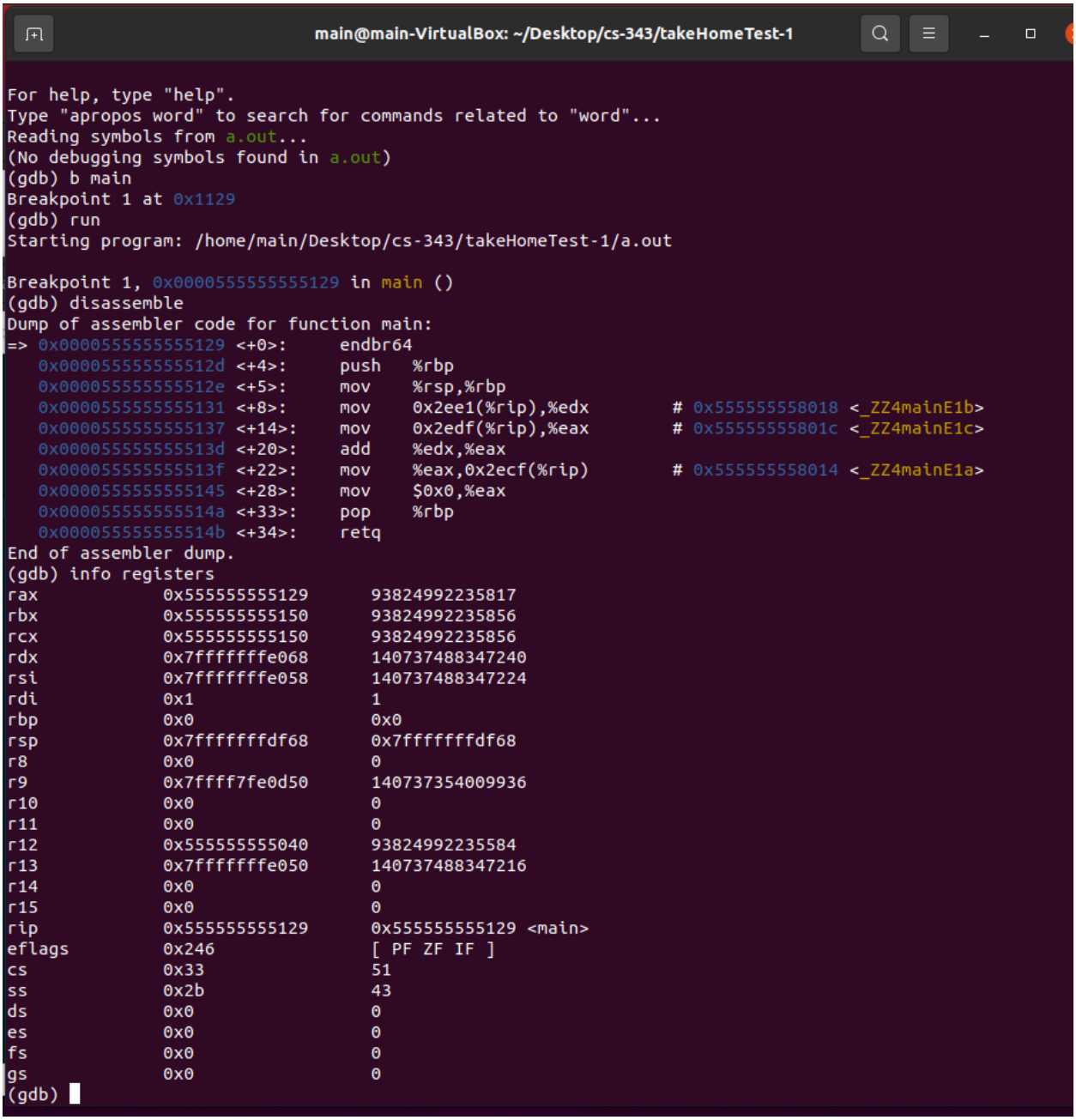

Figure 81: code output, disassemble, info registers.

First, I did gdb run to start the program and then b main to put a breakpoint at the main function line. 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. The endbr64 stands for terminate any indirect branch. The data from memory is copied to two registers and added between them. The result is then saved in memory. 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. The mov instruction stores all local values in registers.

2-6_1:

In figure 82, we can see the source code for the 2-6_1.c in linux editor.

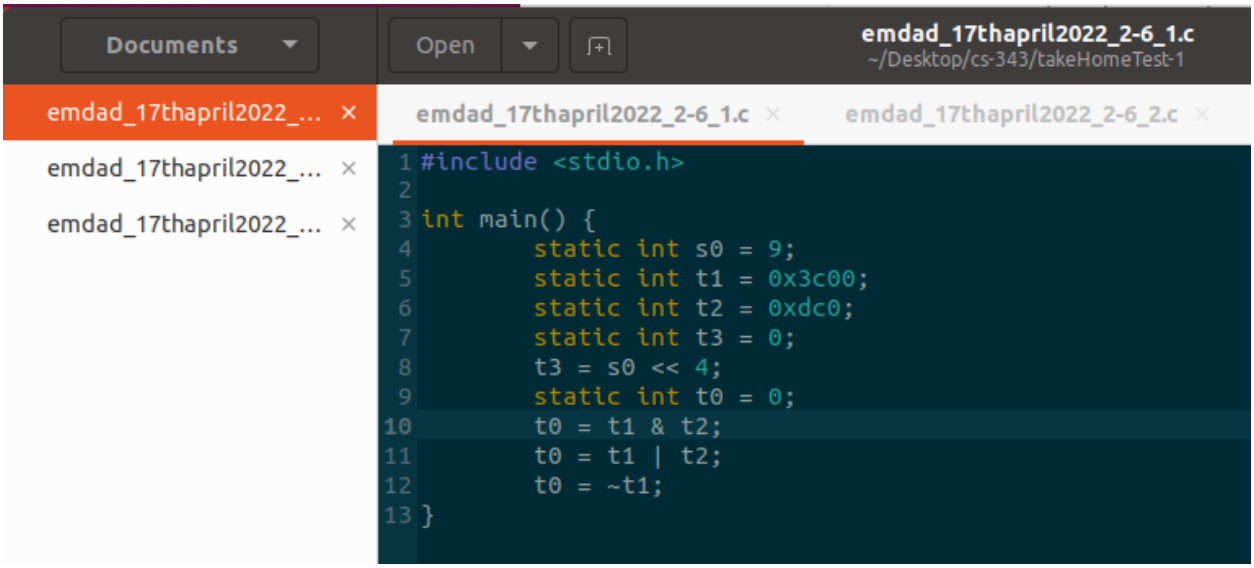

Figure 82: source code for the 2-6_1.c

The main function started at line 3 and then there are some variables initialized . It performs decision making: if-else statement on static variables initialized.

In figure 83, we can see that the code output, disassemble, info registers.

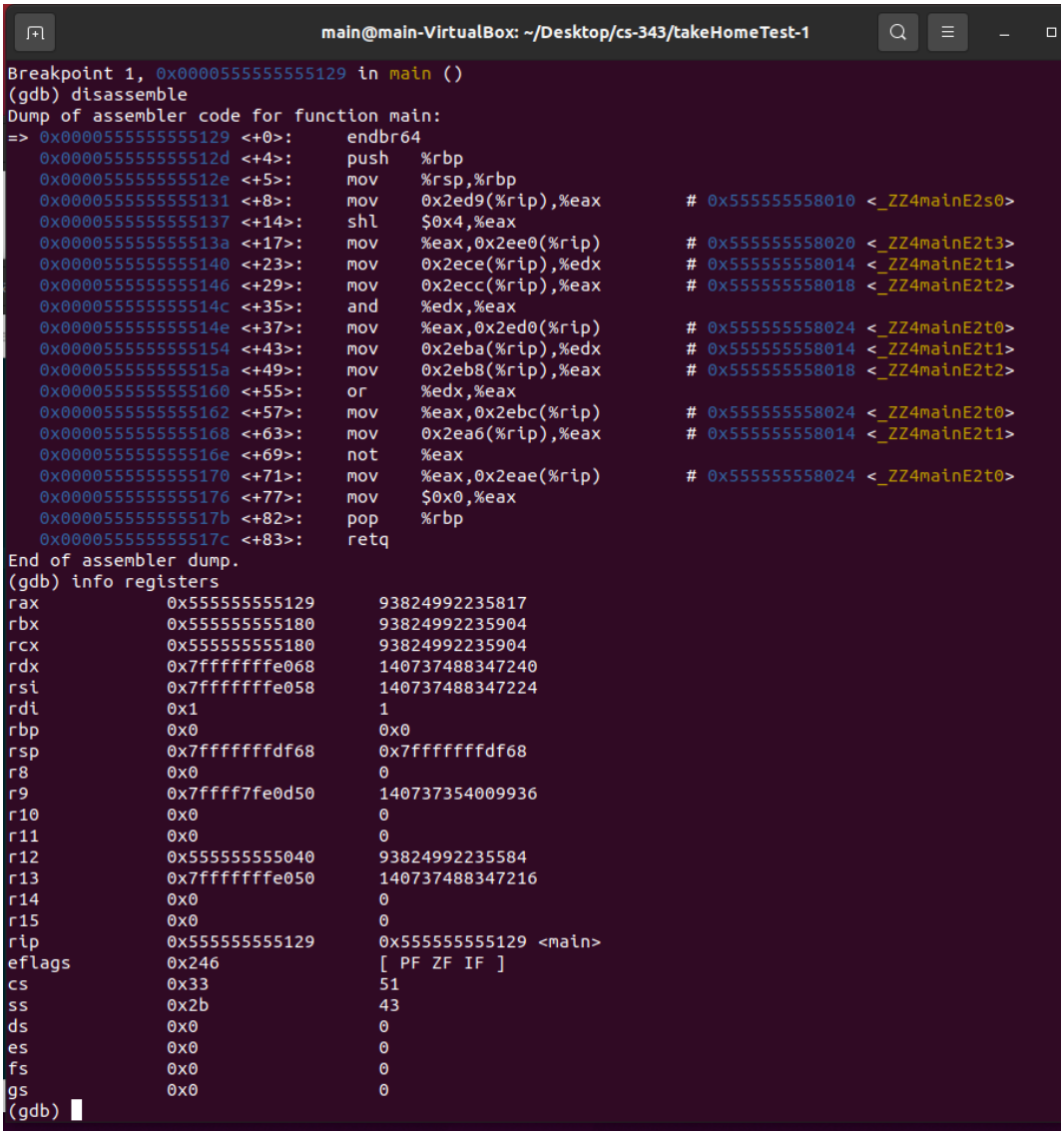

Figure 83: code output, disassemble, info registers.

First, I did gdb run to start the program and then b main to put a breakpoint at the main function line. 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. Memory address 0x555555558010 contains 0x09 and is the location for s0. Memory address 0x555555558014 contains 0x3C00 and is the location of t1. Memory address 0x555555558018 contains 0x0DC0 and is the location for t2. 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. The mov instruction stores all local values in registers.

2-6_2:

In figure 84, we can see the source code for the 2-6_2.cpp in linux editor.

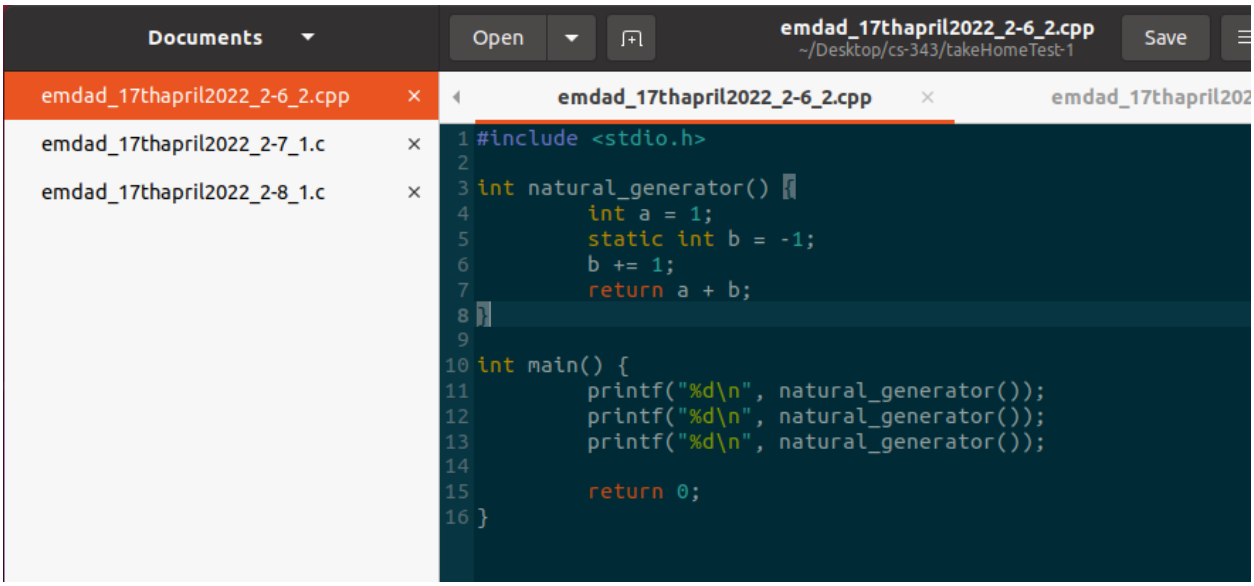

Figure 84: source code for the 2-6_2.cpp

The main function started at line 10 and then there are some variables initialized. We called the natural_generator function in the main function and printed the value of it. One thing to be noted is this file is .cpp.

In figure 85, we can see that the code output, disassemble, info registers.

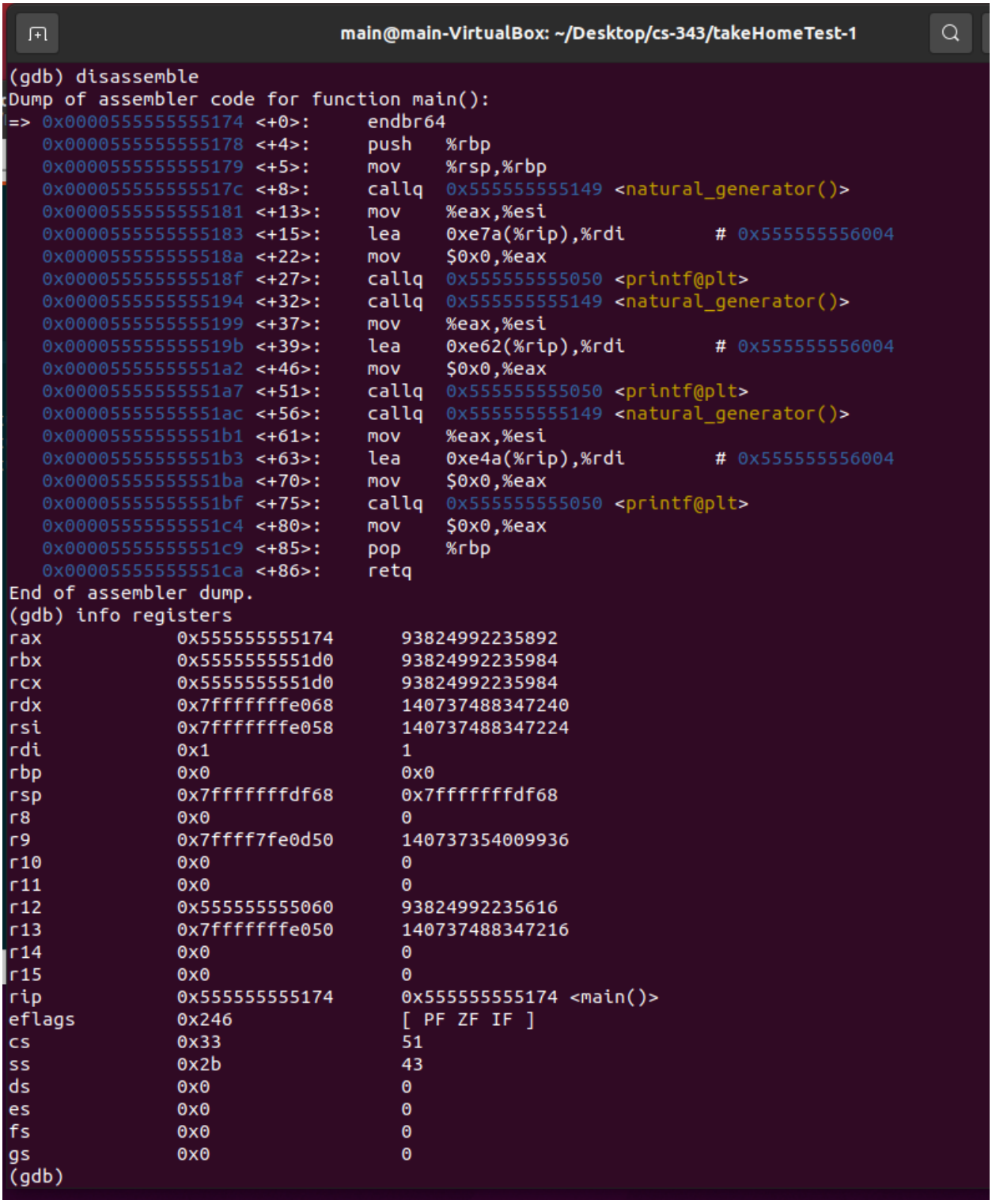

Figure 85: code output, disassemble, info registers.

First, I did gdb run to start the program and then b main to put a breakpoint at the main function line. 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 natural generator and then storing the address of the new generated value. 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. The mov instruction stores all local values in registers.

2-7_1:

In figure 86, we can see the source code for the 2-7_1.c in linux editor.

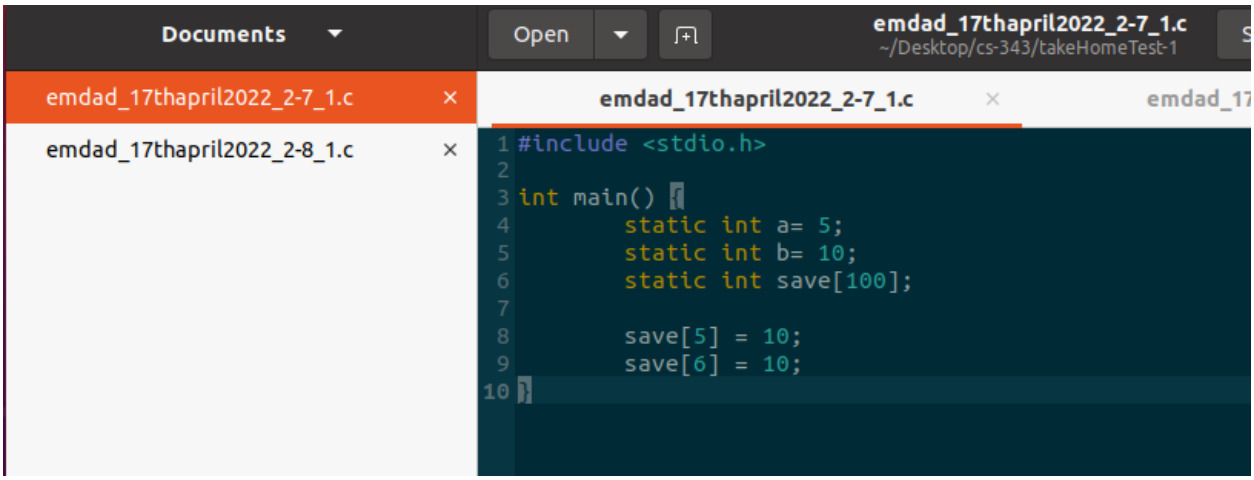

Figure 86: source code for the 2-7_1.c

The main function started at line 3 and then there are some variables initialized . It performs decision making: if-else statement on static variables initialized.

In figure 87, we can see that the code output, disassemble, info registers.

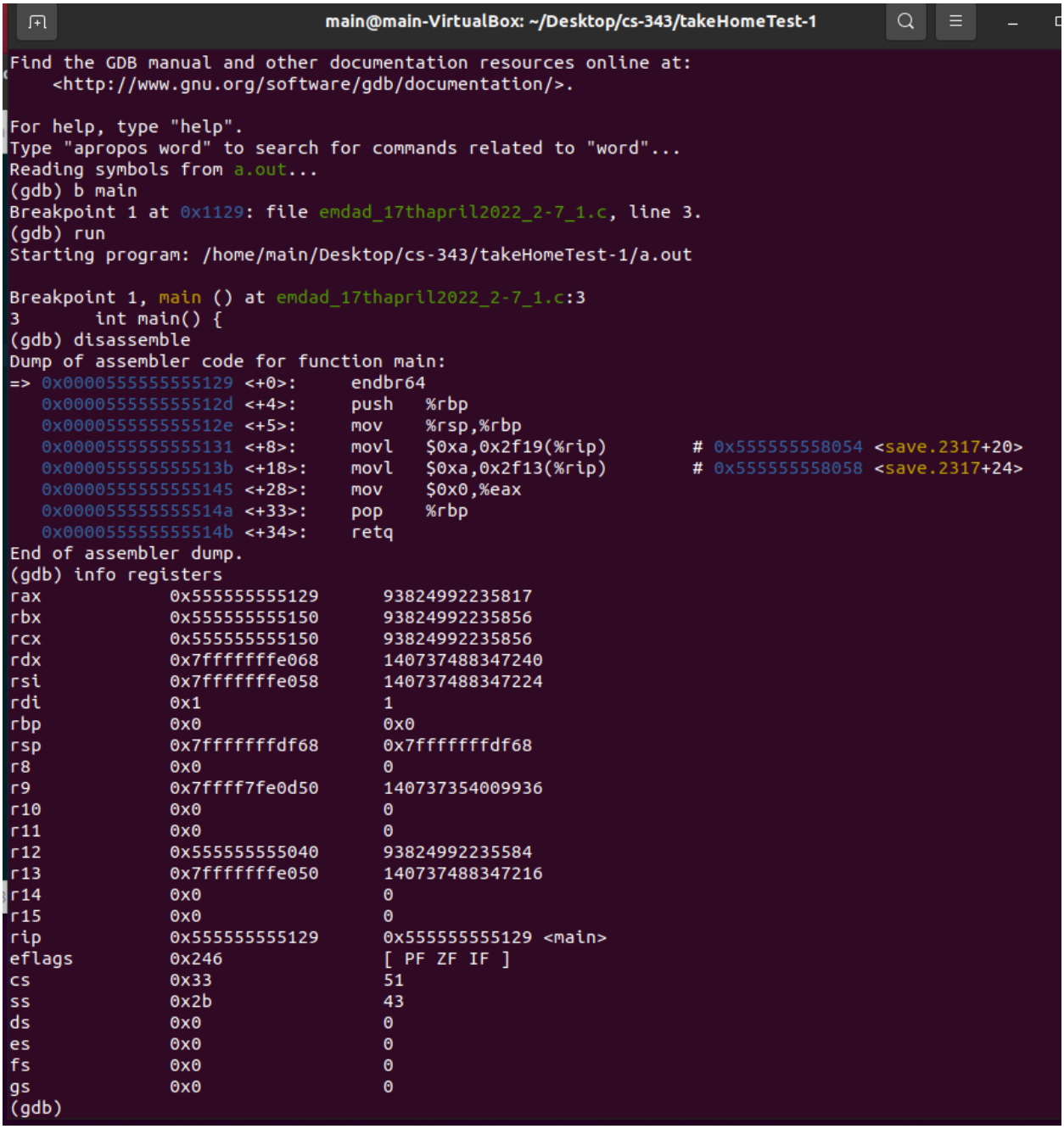

Figure 87: code output, disassemble, info registers.

First, I did gdb run to start the program and then b main to put a breakpoint at the main function line. 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. 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. The mov instruction stores all local values in registers.

2-8_1:

In figure 88, we can see the source code for the 2-8_1.cpp in linux editor.

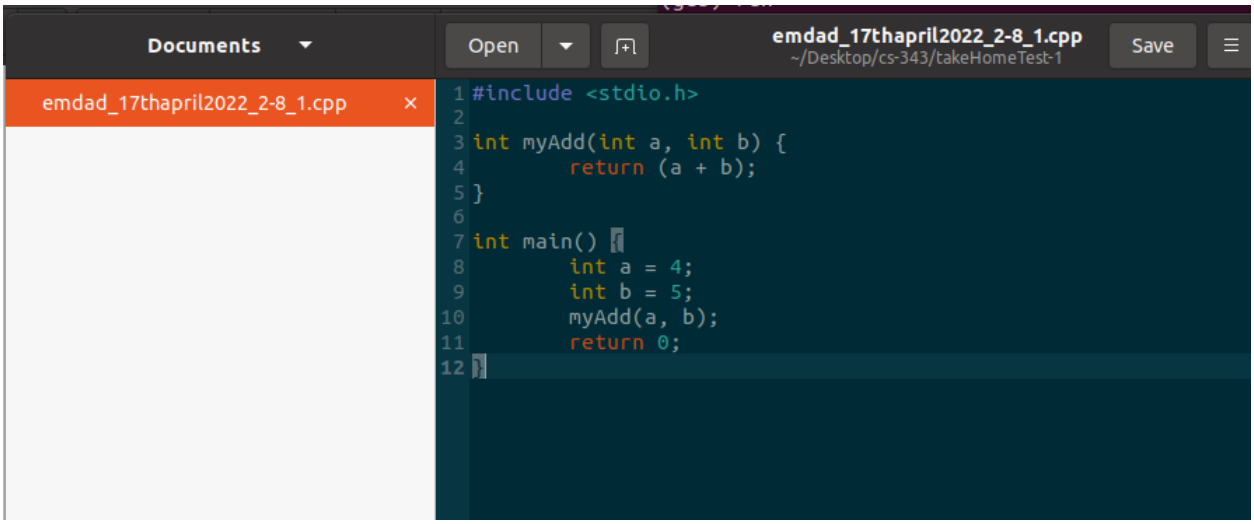

Figure 88: source code for the 2-8_1.cpp

The main function started at line 7 and then there are some variables initialized. We called the myadd function in the main function and printed the value of it. One thing to be noted is this file is .cpp.

In figure 89, we can see that the code output, disassemble, info registers.

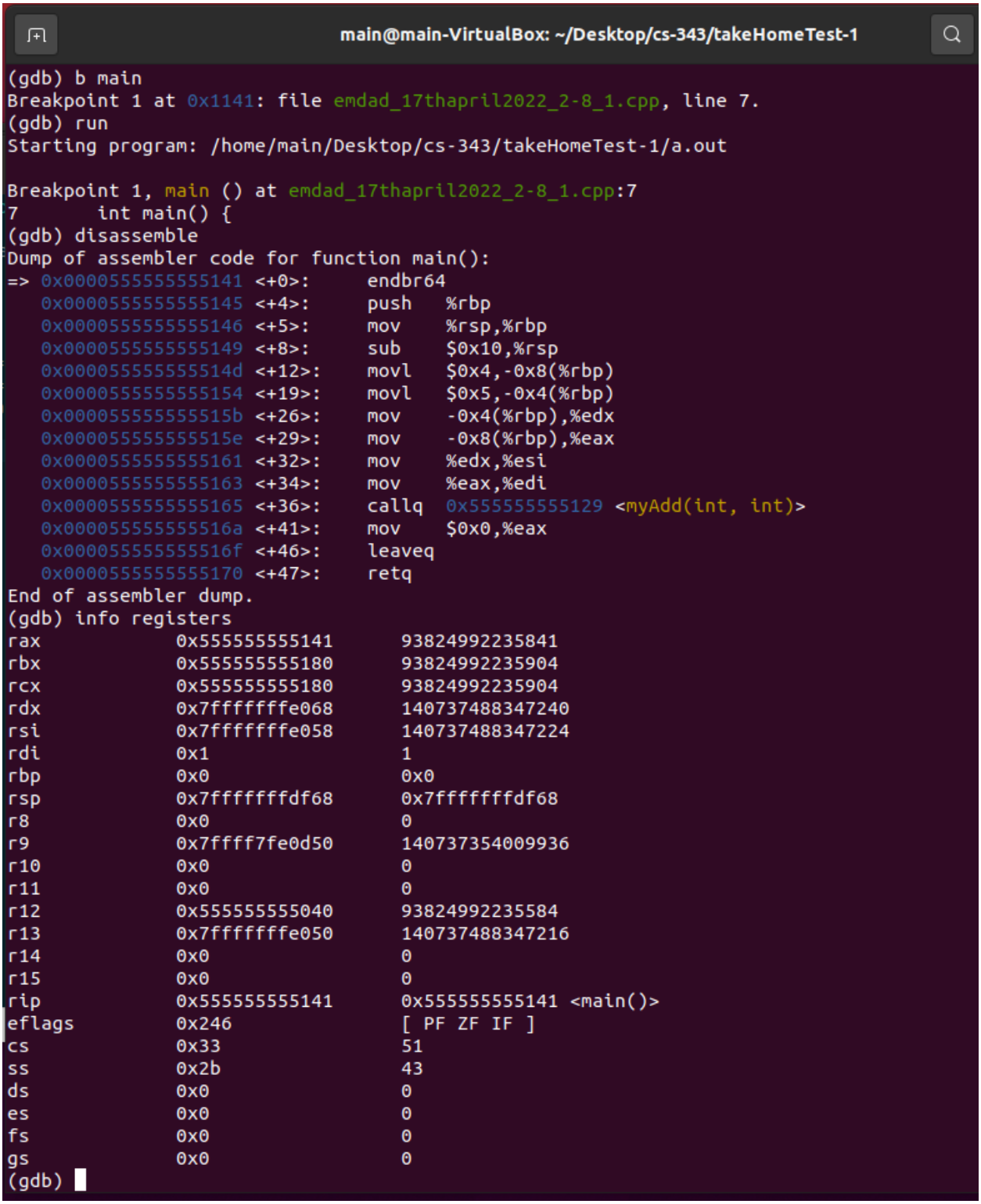

Figure 89: code output, disassemble, info registers.

First, I did gdb run to start the program and then b main to put a breakpoint at the main function line. 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. The callq instruction is the main difference from other functions we did, and we discussed callq in previous example. 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. The mov instruction stores all local values in registers.

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. The stack pointer points to the top of the stack to register. The frame pointer points to the back of the stack. The simulator uses big endian notation to store the data. This basically means that you can read the data from left to right and the most significant byte is stored at the least significant address in memory. In the MARS-MIPS simulator, arithmetic operations are registers for storing inputs, inputs 1, and inputs 2. Logical registers store and, nor, or, and other logical bit operations. Data is communicated between registers and memory by the "lw" and "sw" instructions, which represent load and store words. Loads word copy data from memory into a register and stores word copy data from register into memory. MIPS compiles loop statements with bne, branch not equal, and statements. If the two operands are not equal, the instruction decoder jumps to the marked label. MIPS function calls use "jal". It stores the return address that the register returns after a particular call is made.

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.

Conclusion:

This take-home test taught me how to demonstrate, understand and ability to compare 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. I used the professor’s and book examples were provided and analyzed them. I also learned about little endian, and big

endian where appropriate, demonstrate if-then-else statements on each platform, describe differences and similarities in each case. Further, I learned and understood about registers, memory, and disassembly and how to debug code overall.