

Take Home Test 1

MdShahid Bin Emdad

April 17th, 2022

CSC 34200

TABLE OF CONTENTS

| | |
|--|-----------|
| Objective: | 3 |
| Description of Specifications, and Functionality: | 3 |
| PART I: MARS Simulator..... | 3 |
| 2-2_1:..... | 3 |
| 2-2_2:..... | 8 |
| 2-3_1:..... | 11 |
| 2-3_2:..... | 13 |
| 2-5_2:..... | 16 |
| 2-6_1:..... | 19 |
| 2-7_1:..... | 22 |
| 2-8_1:..... | 24 |
| PART II: Visual Studio 2019 (Intel x86) | 26 |
| 2-2_1:..... | 26 |
| 2-2_2:..... | 29 |
| 2-3_1:..... | 31 |
| 2-3_2:..... | 32 |
| 2-5_1:..... | 33 |
| 2-6_1:..... | 35 |
| 2-6_2:..... | 36 |
| 2-7_1:..... | 38 |
| 2-8_1:..... | 39 |
| PART III: x86_64 ISA, Linux 64-bit | 41 |
| 2-2_1:..... | 41 |
| 2-2_2:..... | 43 |
| 2-3_1:..... | 45 |
| 2-3_2:..... | 47 |
| 2-5_1:..... | 49 |

| | |
|---------------------------|-----------|
| 2-6_1: | 51 |
| 2-6_2: | 53 |
| 2-7_1: | 55 |
| 2-8_1: | 57 |
| Explanation: | 59 |
| Conclusion: | 59 |

Objective:

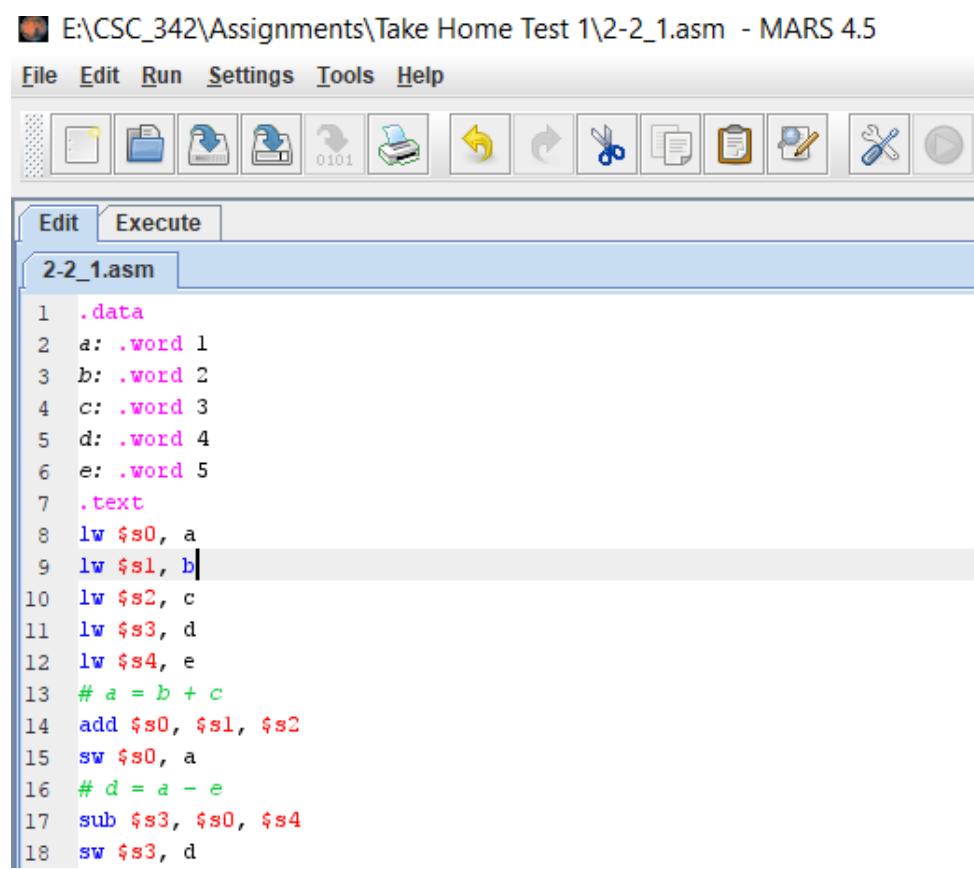
The objective of this test is to demonstrate, understand and ability to compare MIPS instructions set architecture, Intel x86 ISA using Windows MS 32-bit compiler and debugger and a Intel X86_64 bit ISA processor running Linux, 64 bit gcc and gdb. Some examples were provided by the professor, and we 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:



The screenshot shows the MARS 4.5 simulator interface. The title bar reads "E:\CSC_342\Assignments\Take Home Test 1\2-2_1.asm - MARS 4.5". The menu bar includes File, Edit, Run, Settings, Tools, and Help. Below the menu is a toolbar with various icons for file operations like Open, Save, and Run. The main window has tabs for "Edit" and "Execute", with "Edit" selected. The code editor displays the following assembly code:

```

1 .data
2 a: .word 1
3 b: .word 2
4 c: .word 3
5 d: .word 4
6 e: .word 5
7 .text
8 lw $s0, a
9 lw $s1, b
10 lw $s2, c
11 lw $s3, d
12 lw $s4, e
13 # a = b + c
14 add $s0, $s1, $s2
15 sw $s0, a
16 # d = a - e
17 sub $s3, $s0, $s4
18 sw $s3, d

```

The line "lw \$s1, b" is highlighted with a red cursor, indicating it is currently being edited.

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 successfully.

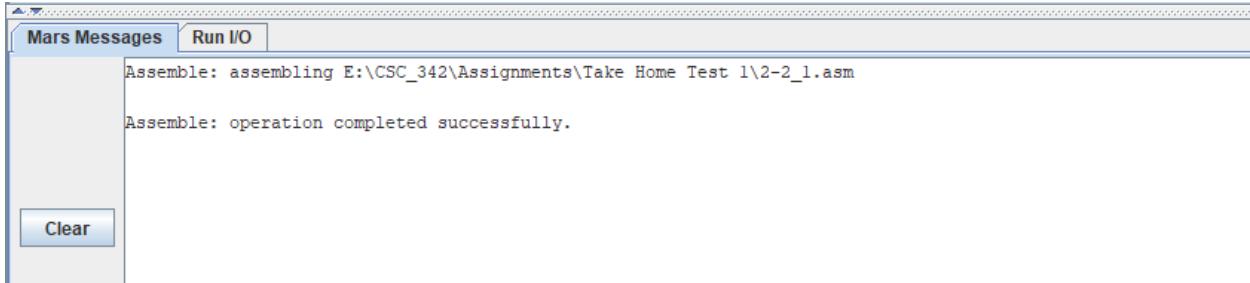


Figure 2: Operation Completion

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

| Text Segment | | | |
|--------------|------------|------------|-------------------------|
| Bkpt | Address | Code | Basic |
| | 0x00400000 | 0x3e011001 | lui \$1,0x00001001 |
| | 0x00400004 | 0x3c300000 | lw \$16,0x00000000(\$1) |
| | 0x00400008 | 0x3c011001 | lui \$1,0x00001001 |
| | 0x0040000c | 0x3c310004 | lw \$17,0x00000004(\$1) |
| | 0x00400010 | 0x3c011001 | lui \$1,0x00001001 |
| | 0x00400014 | 0x3c320008 | lw \$18,0x00000008(\$1) |
| | 0x00400018 | 0x3c011001 | lui \$1,0x00001001 |
| | 0x0040001c | 0x3c33000c | lw \$19,0x0000000c(\$1) |
| | 0x00400020 | 0x3c011001 | lui \$1,0x00001001 |
| | 0x00400024 | 0x3c340010 | lw \$20,0x00000010(\$1) |
| | 0x00400028 | 0x02328020 | add \$16,\$17,\$18 |
| | 0x0040002c | 0x3c011001 | lui \$1,0x00001001 |
| | 0x00400030 | 0xac300000 | sw \$16,0x00000000(\$1) |
| | 0x00400034 | 0x02149822 | sub \$19,\$16,\$20 |
| | 0x00400038 | 0x3c011001 | lui \$1,0x00001001 |
| | 0x0040003c | 0xac33000c | sw \$19,0x0000000c(\$1) |

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).

| Address | Value (+0) | Value (+4) | Value (+8) | Value (+c) | Value (+10) | Value (+14) | Value (+18) | Value (+1c) |
|------------|------------|------------|------------|------------|-------------|-------------|-------------|-------------|
| 0x10010000 | 0x00000001 | 0x00000002 | 0x00000003 | 0x00000004 | 0x00000005 | 0x00000000 | 0x00000000 | 0x00000000 |
| 0x10010020 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |
| 0x10010040 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |
| 0x10010060 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |
| 0x10010080 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |
| 0x100100a0 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |
| 0x100100c0 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |
| 0x100100e0 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |
| 0x10010100 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |
| 0x10010120 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |
| 0x10010140 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |
| 0x10010160 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |
| 0x10010180 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |
| 0x100101a0 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 | 0x00000000 |

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 | 0x7ffffeffc |
| \$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.

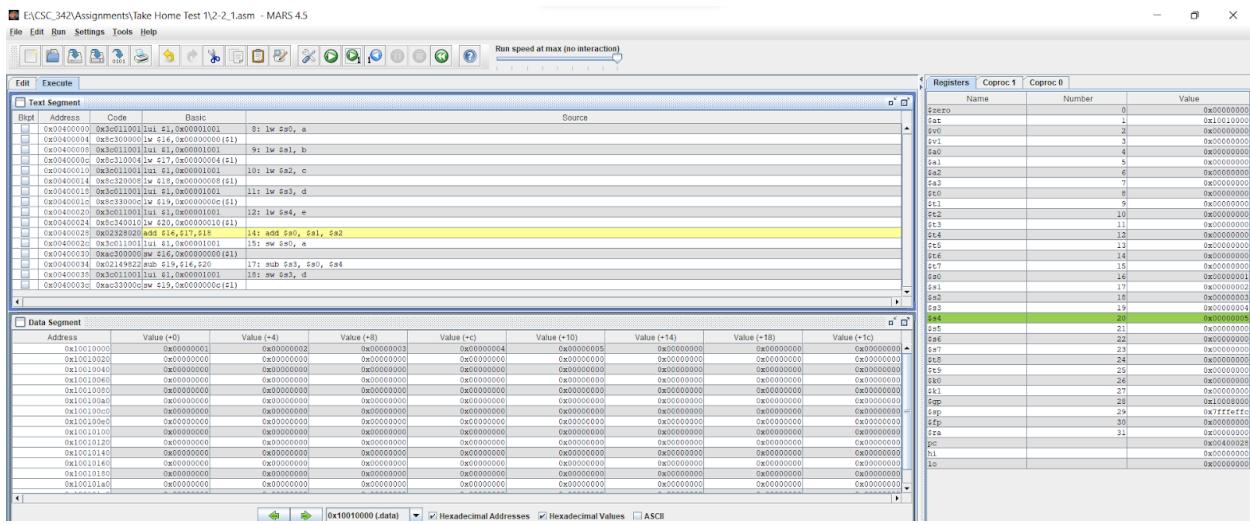


Figure 6: registers before arithmetic operations.

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

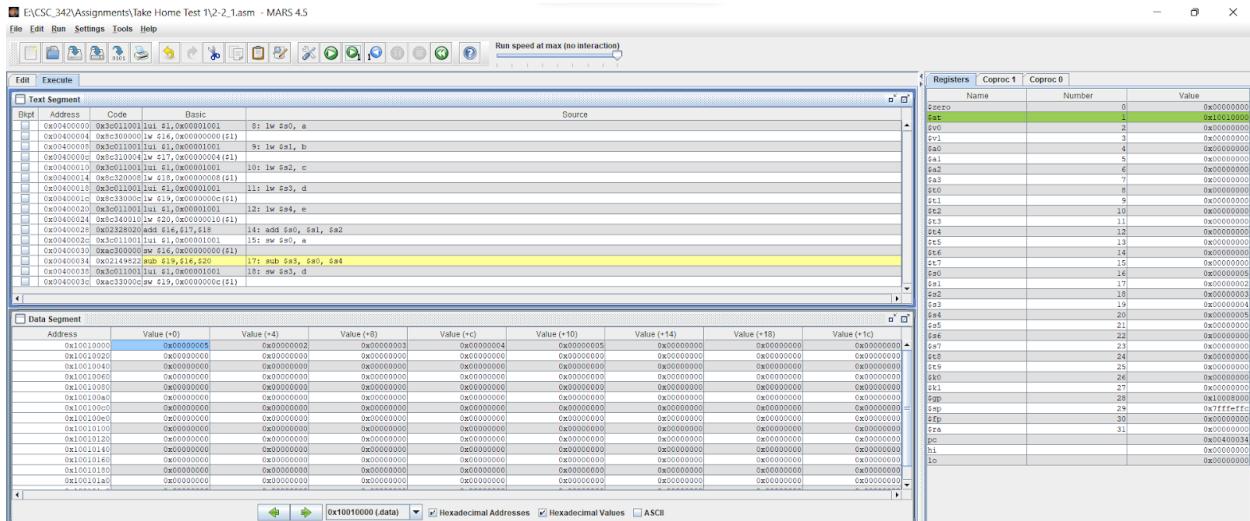
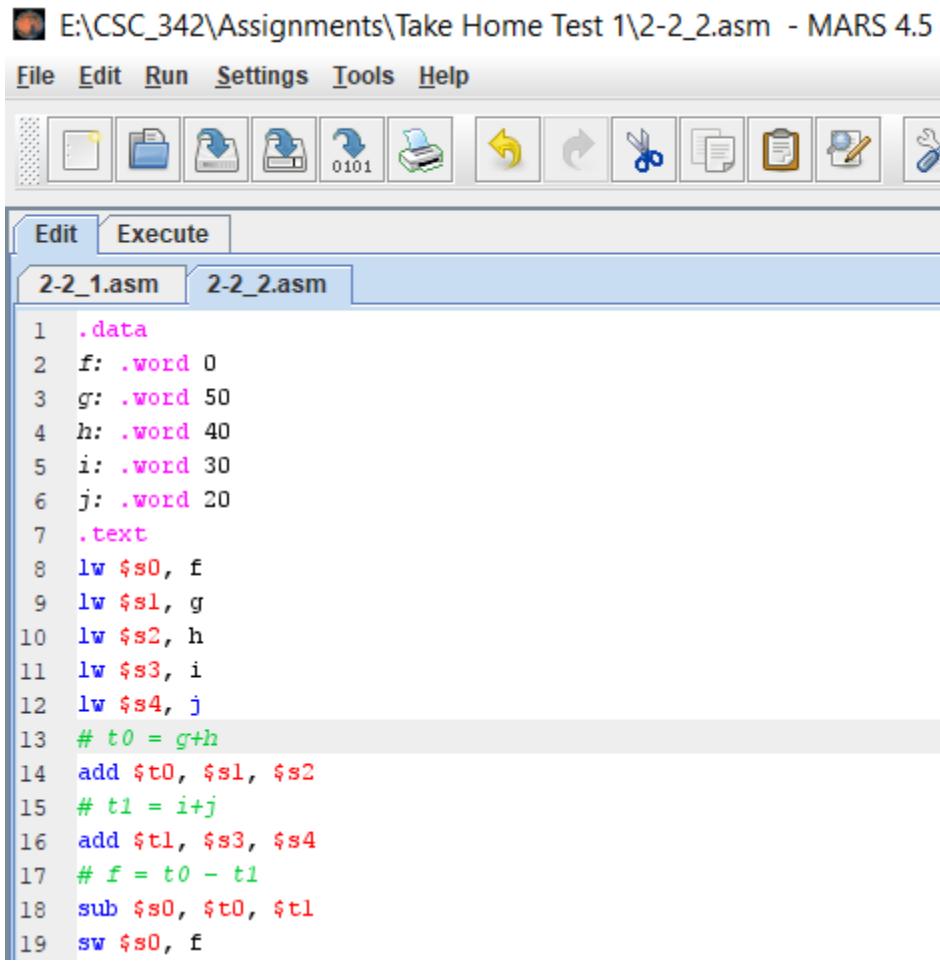


Figure 7: registers after arithmetic operations

We can see the register value stores under the name \$at and the value of (+0) offset is 0x0000000005.

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 subsectrations.



The screenshot shows the MARS 4.5 assembly editor interface. The title bar reads "E:\CSC_342\Assignments\Take Home Test 1\2-2_2.asm - MARS 4.5". The menu bar includes File, Edit, Run, Settings, Tools, and Help. Below the menu is a toolbar with various icons. The tabs at the top are "Edit" and "Execute", with "Edit" currently selected. The code window displays the following assembly code:

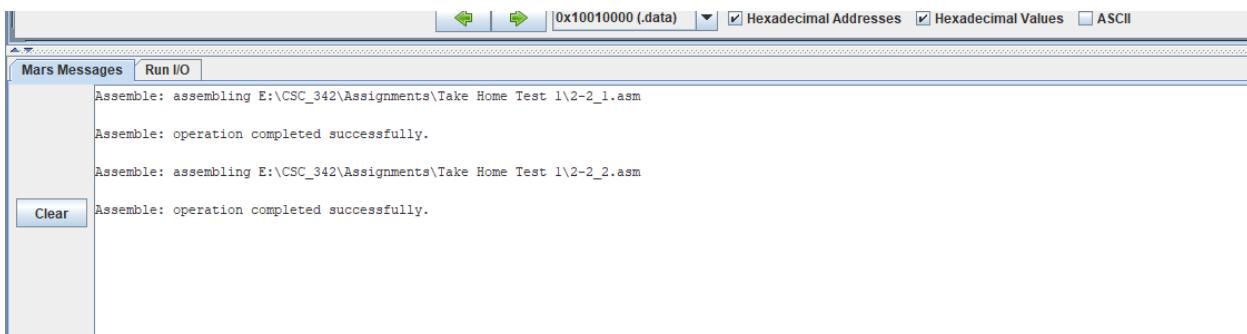
```

1 .data
2 f: .word 0
3 g: .word 50
4 h: .word 40
5 i: .word 30
6 j: .word 20
7 .text
8 lw $s0, f
9 lw $s1, g
10 lw $s2, h
11 lw $s3, i
12 lw $s4, j
13 # t0 = g+h
14 add $t0, $s1, $s2
15 # t1 = i+j
16 add $t1, $s3, $s4
17 # f = t0 - t1
18 sub $s0, $t0, $t1
19 sw $s0, f

```

Figure 8: ASM code for 2-2_2

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



The screenshot shows the "Mars Messages" window from the MARS 4.5 interface. The window title is "Mars Messages". The message log contains the following entries:

- Assemble: assembling E:\CSC_342\Assignments\Take Home Test 1\2-2_1.asm
- Assemble: operation completed successfully.
- Assemble: assembling E:\CSC_342\Assignments\Take Home Test 1\2-2_2.asm
- Assemble: operation completed successfully.

A "Clear" button is visible at the bottom left of the window.

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.

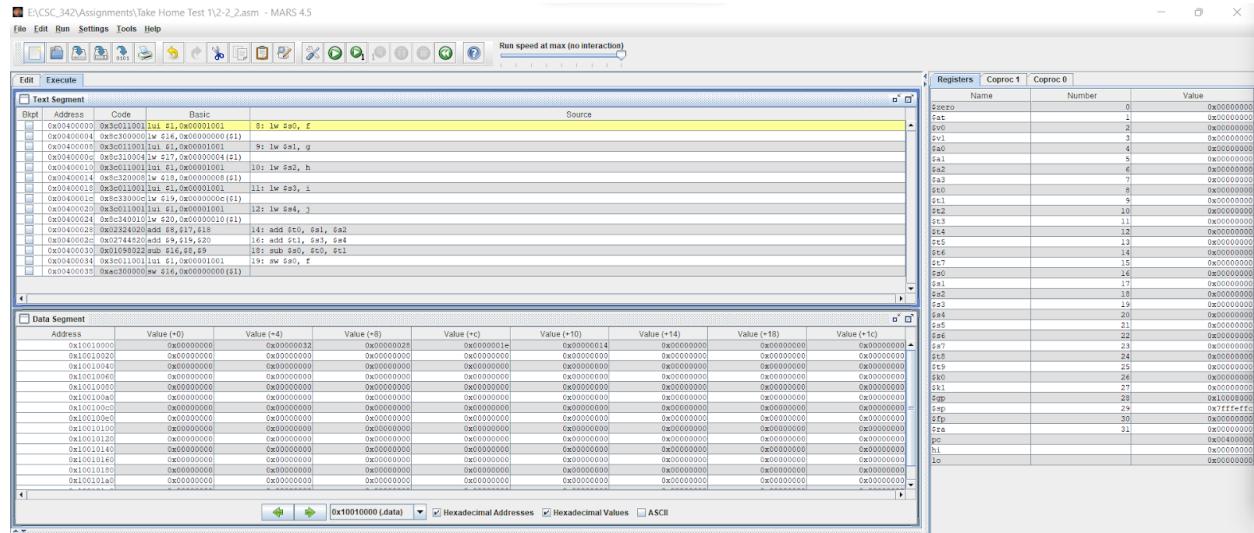


Figure 10: disassembly, memory, registers of 2-2_2.asm

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.

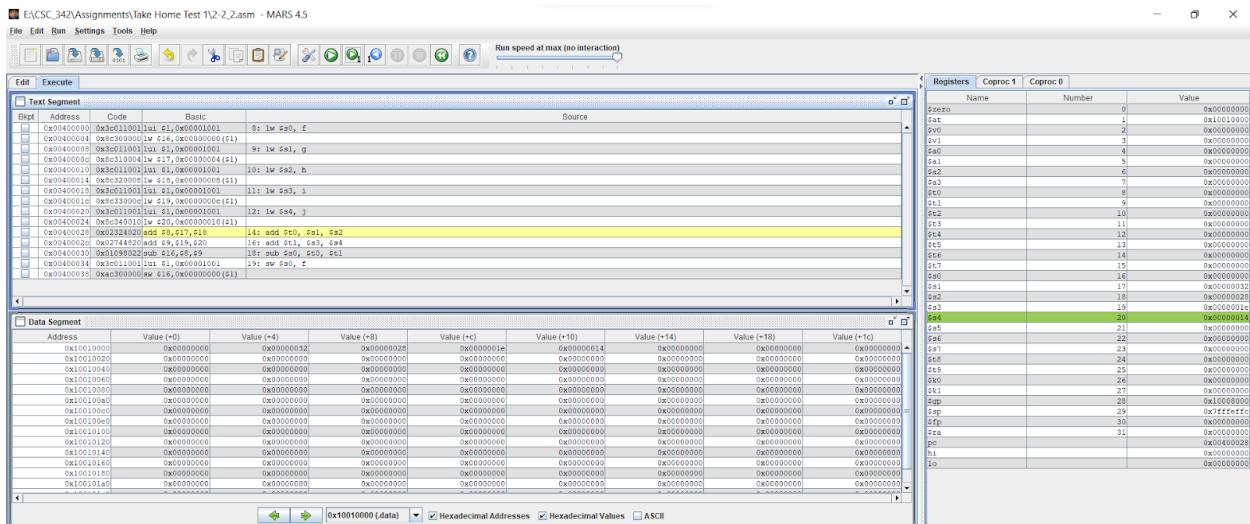


Figure 11: registers before arithmetic operations.

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

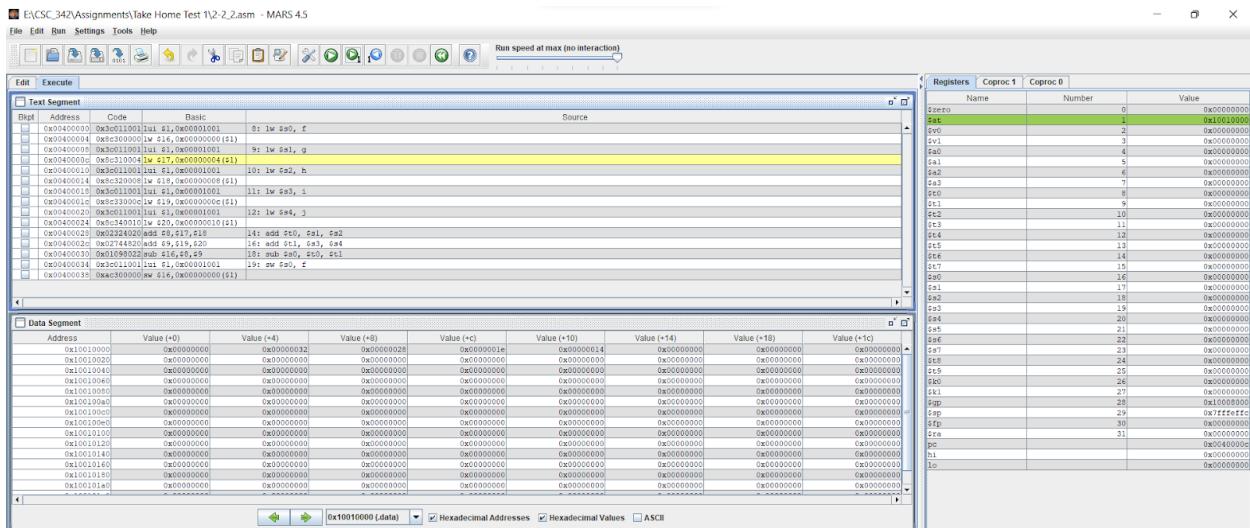
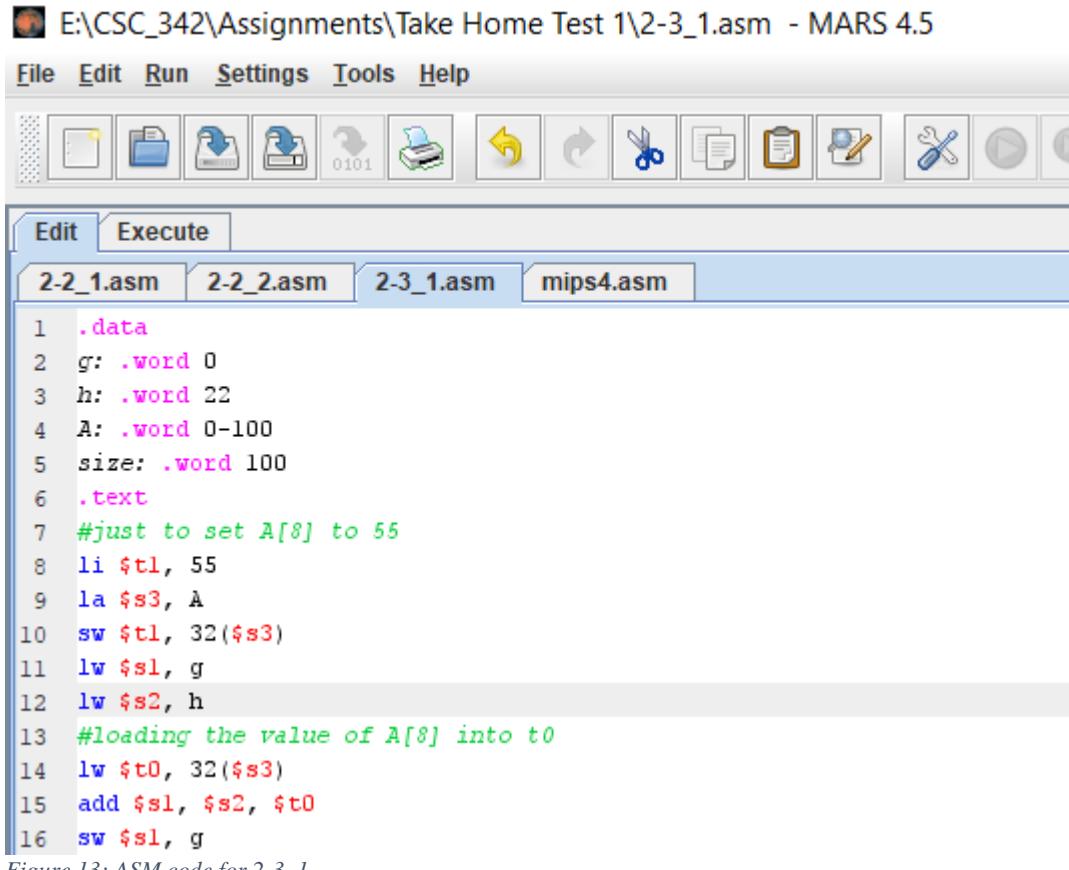


Figure 12: registers after arithmetic operations.

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 holds 4 bytes. If we multiple, we get $8 \times 4 = 32$.

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.



The screenshot shows the MARS 4.5 assembly editor interface. The title bar reads "E:\CSC_342\Assignments\Take Home Test 1\2-3_1.asm - MARS 4.5". The menu bar includes File, Edit, Run, Settings, Tools, and Help. Below the menu is a toolbar with various icons. The tabs at the top of the code area are labeled 2-2_1.asm, 2-2_2.asm, 2-3_1.asm (which is selected), and mips4.asm. The code listing is as follows:

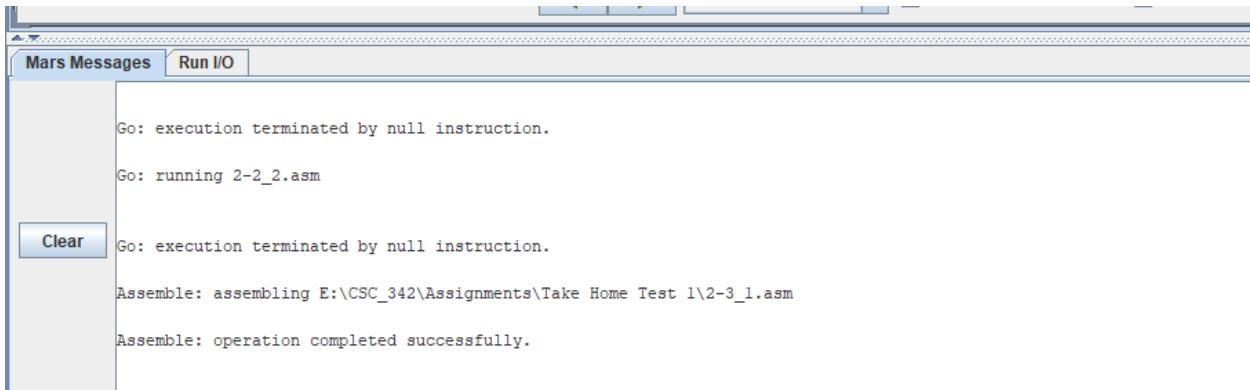
```

1 .data
2 g: .word 0
3 h: .word 22
4 A: .word 0-100
5 size: .word 100
6 .text
7 #just to set A[8] to 55
8 li $t1, 55
9 la $s3, A
10 sw $t1, 32($s3)
11 lw $s1, g
12 lw $s2, h
13 #loading the value of A[8] into t0
14 lw $t0, 32($s3)
15 add $s1, $s2, $t0
16 sw $s1, g

```

Figure 13: ASM code for 2-3_1

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

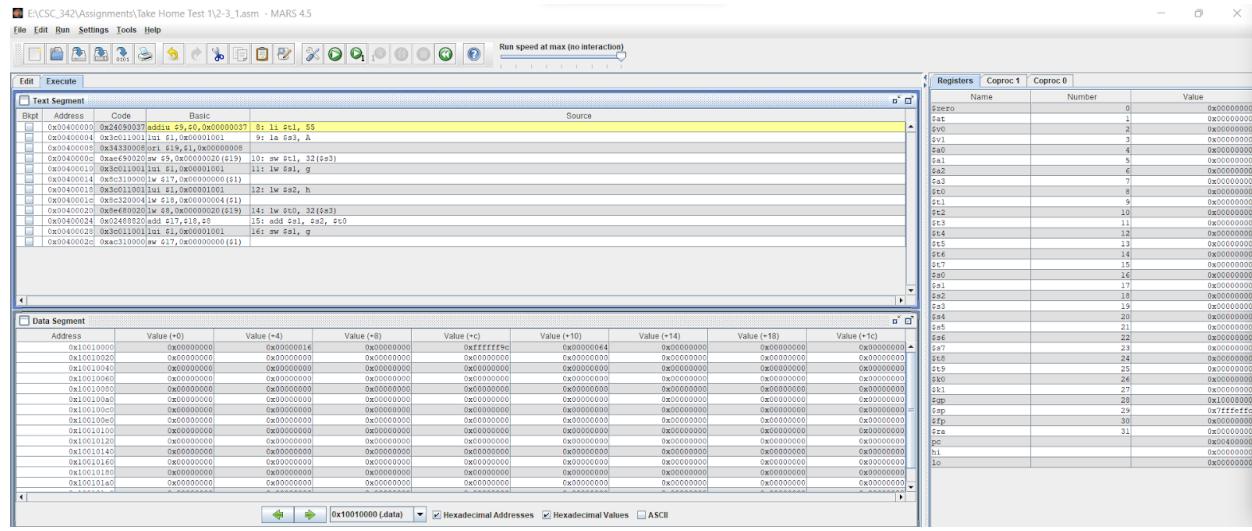


The screenshot shows the "Mars Messages" window. It displays the following log entries:

- Go: execution terminated by null instruction.
- Go: running 2-2_2.asm
- Clear
- Go: execution terminated by null instruction.
- Assemble: assembling E:\CSC_342\Assignments\Take Home Test 1\2-3_1.asm
- Assemble: operation completed successfully.

Figure 14: Operation Completion

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



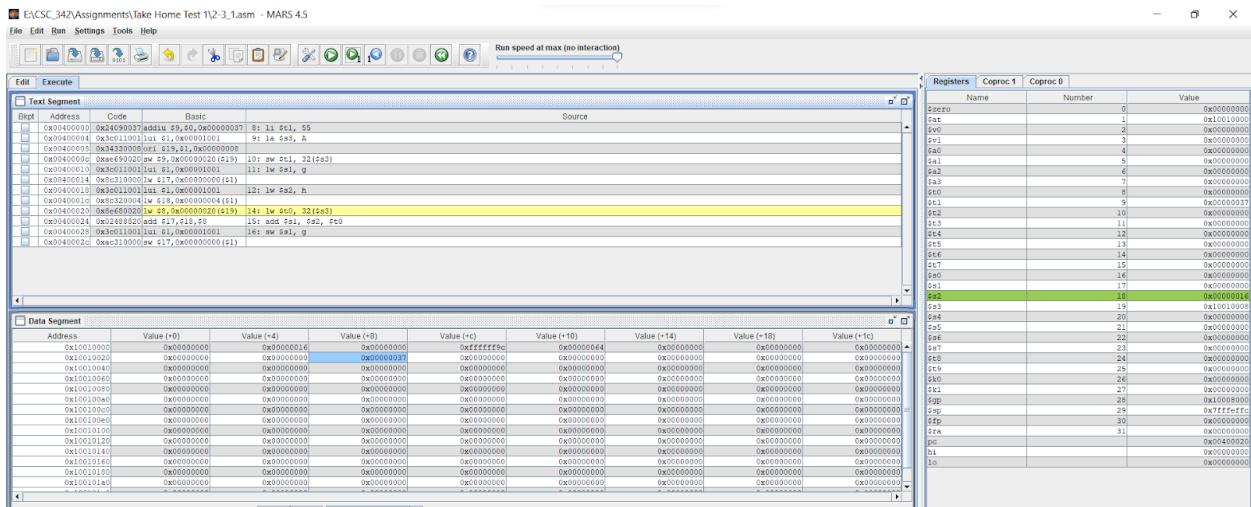


Figure 16: registers after arithmetic operations.

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.

```

E:\CSC_342\Assignments\Take Home Test 1\2-3_2.asm - MARS 4.5

File Edit Run Settings Tools Help

Edit Execute
2-2_1.asm 2-2_2.asm 2-3_1.asm 2-3_2.asm

1 .data
2 h: .word 25
3 A: .word 0-100
4 size: .word 100
5 .
6 lw $s2, h
7 #initializing A[8] to 200
8 li $t1, 200
9 la $s3, A
10 sw $t1, 32($s3)
11 #A[12] = h + A[8]
12 lw $t0, 32($s3)
13 add $t0, $s2, $t0
14 sw $t0, 48($s3)

```

Figure 17: ASM code for 2-3_2

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

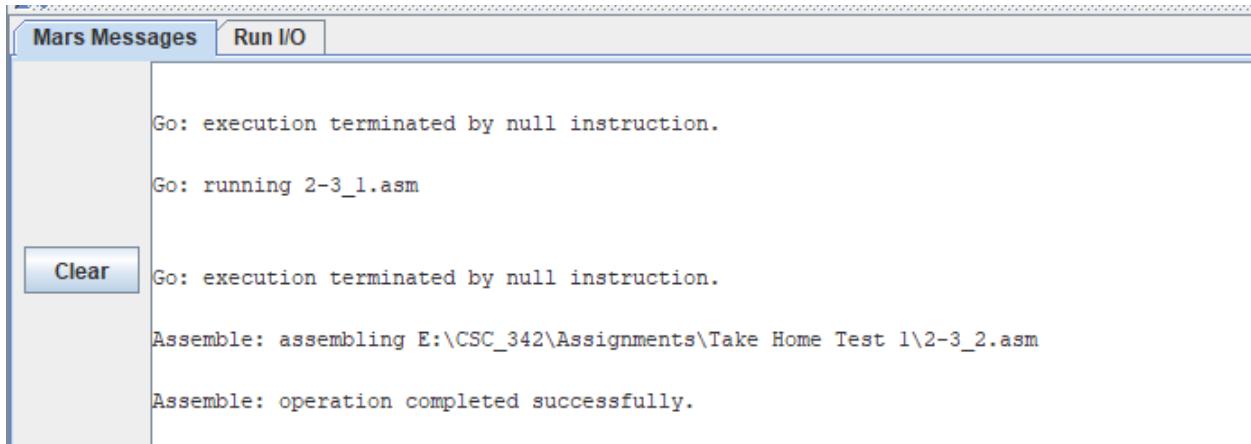


Figure 18: Operation Completion

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

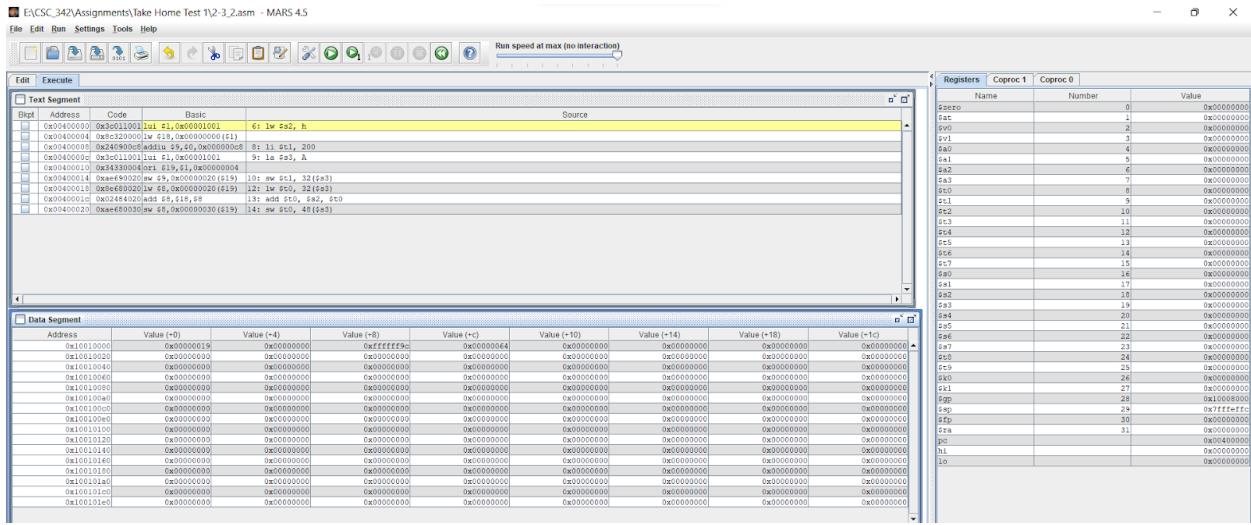


Figure 19: disassembly, memory, registers of MIPS file, 2-3_2.asm.

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.

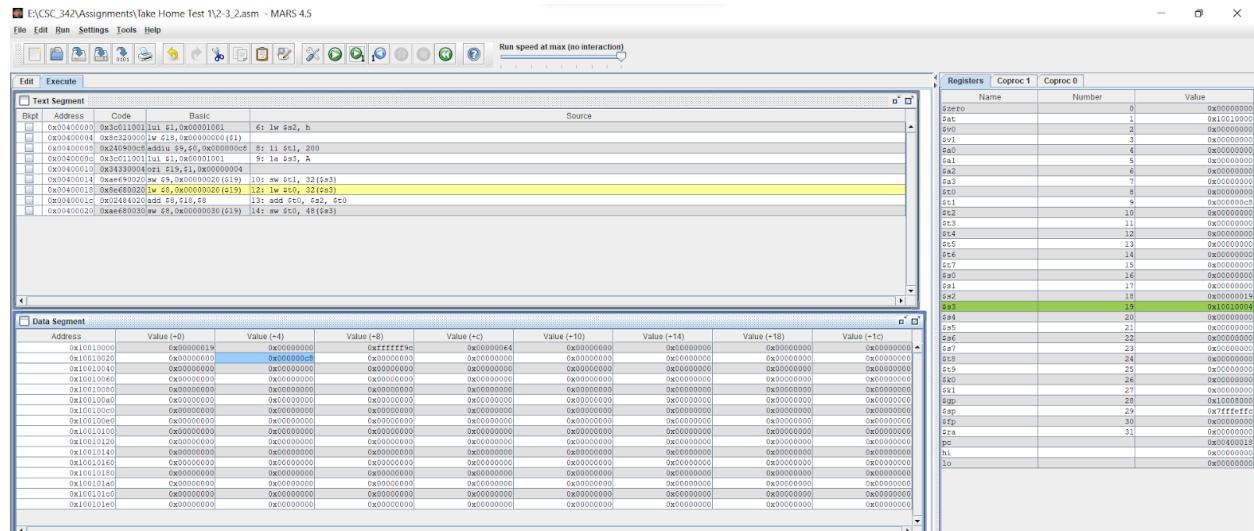


Figure 20: registers before arithmetic operations.

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

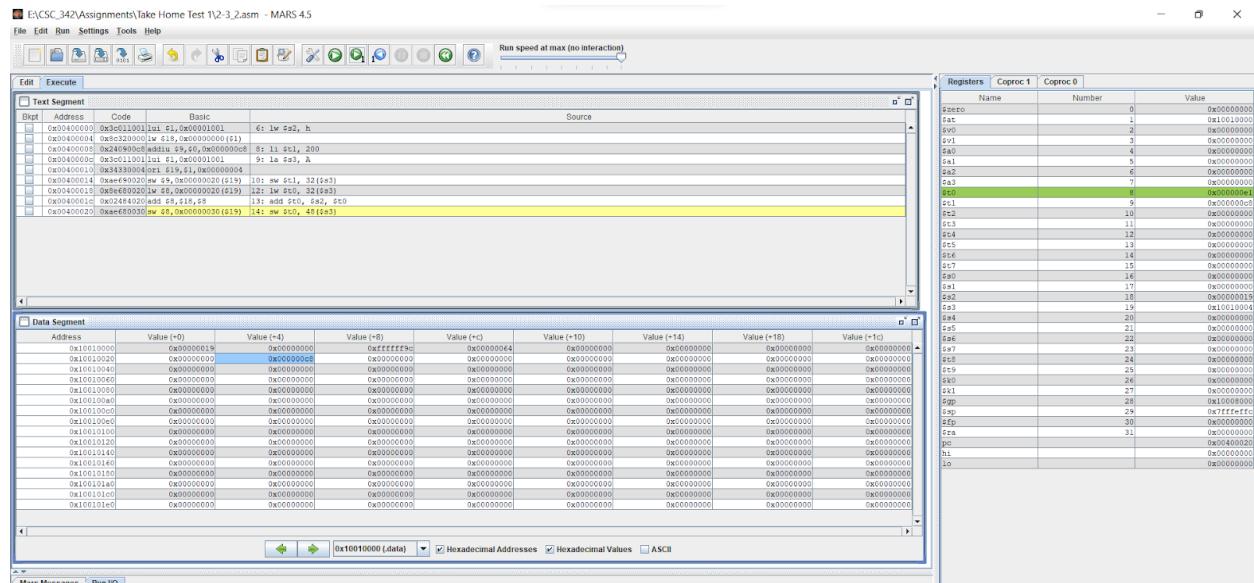


Figure 21: registers after arithmetic operations.

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

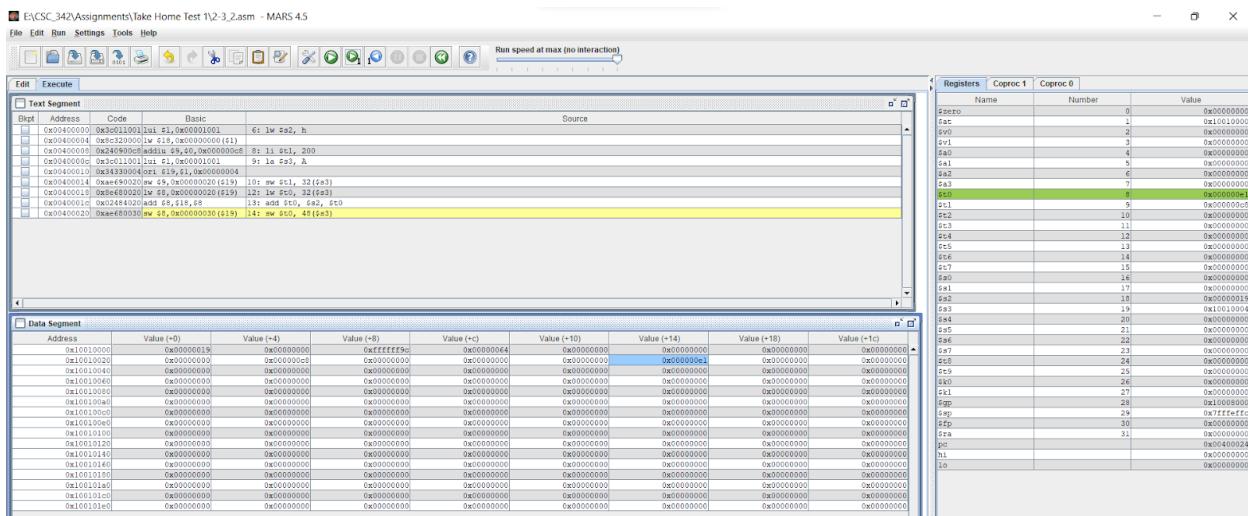


Figure 22: memory value changed

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.

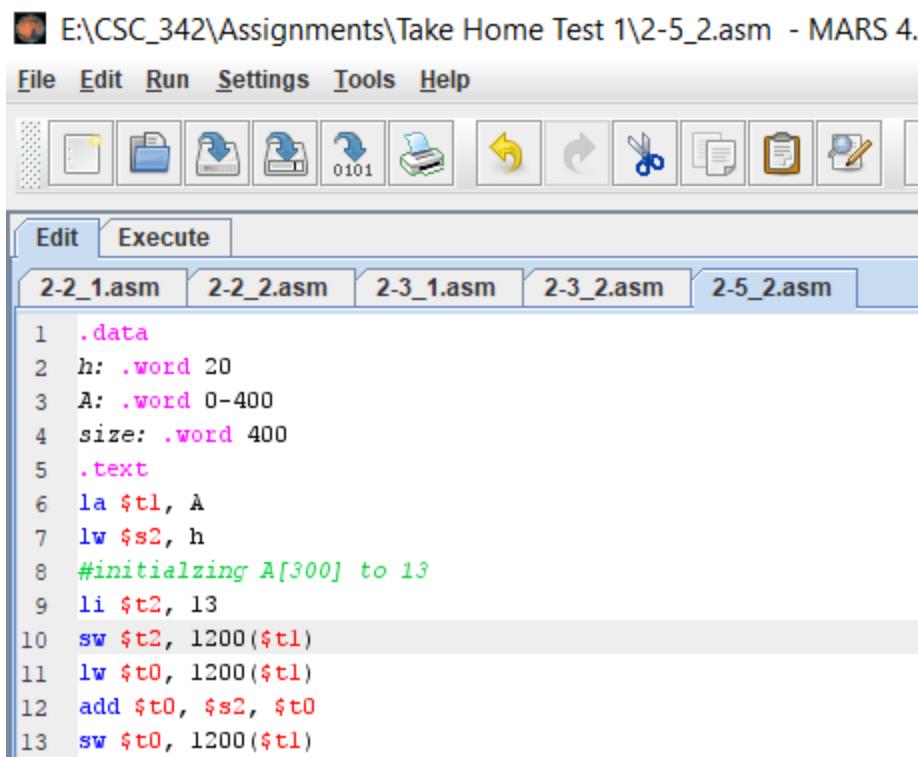


Figure 23: ASM code for 2-5_2

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

The screenshot shows the Mars Simulator interface with the following text in the message window:

```

Mars Messages Run IO

Step: execution terminated due to null instruction.

Assemble: assembling E:\CSC_342\Assignments\Take Home Test 1\2-5_2.asm

Error in E:\CSC_342\Assignments\Take Home Test 1\2-5_2.asm line 11 column 1: "lw": Too many or incorrectly formatted operands. Expected: lw $t1,-100($t2)
Error in E:\CSC_342\Assignments\Take Home Test 1\2-5_2.asm line 12 column 1: "$t0" is not a recognized operator
Assemble: operation completed with errors.

Assemble: assembling E:\CSC_342\Assignments\Take Home Test 1\2-5_2.asm

Assemble: operation completed successfully.

```

Figure 24: Operation Completion

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

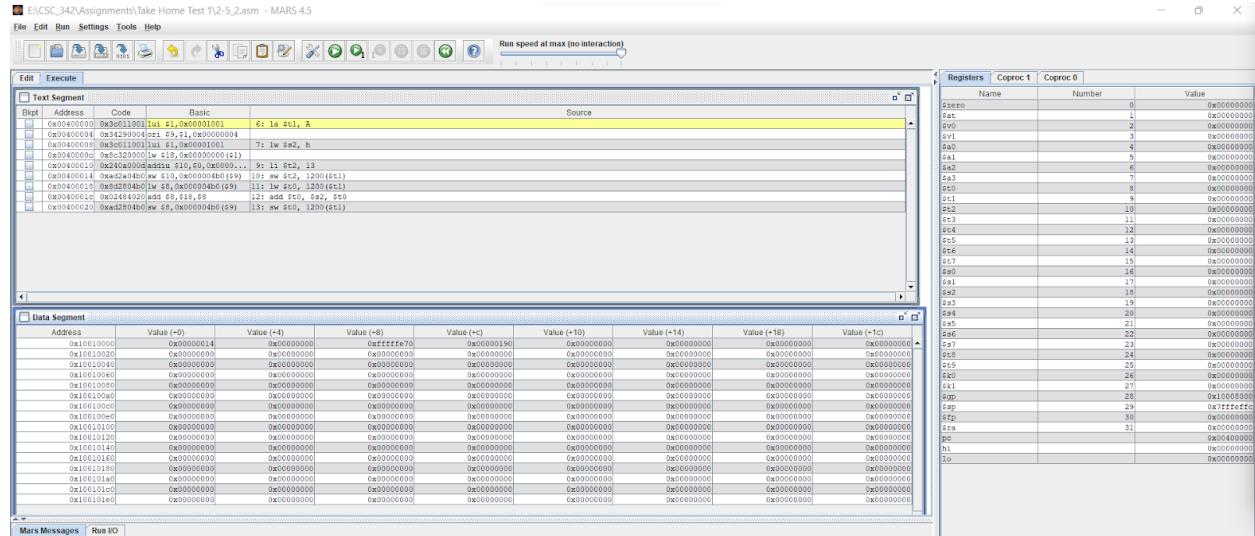


Figure 25: disassembly, memory, registers of MIPS file, 2-5_2.asm.

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.

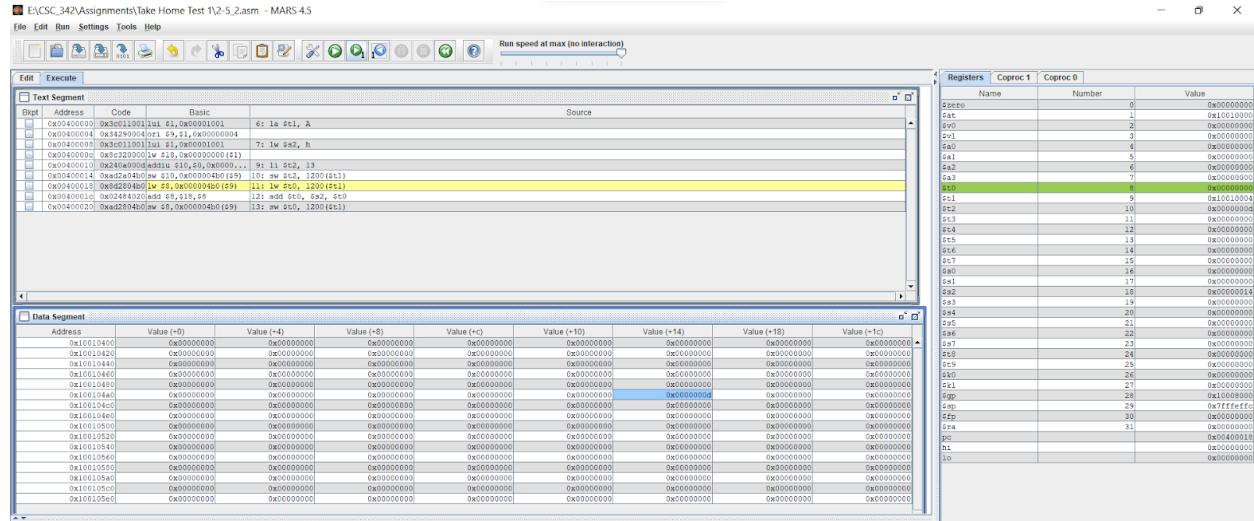


Figure 26: registers before arithmetic operations

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

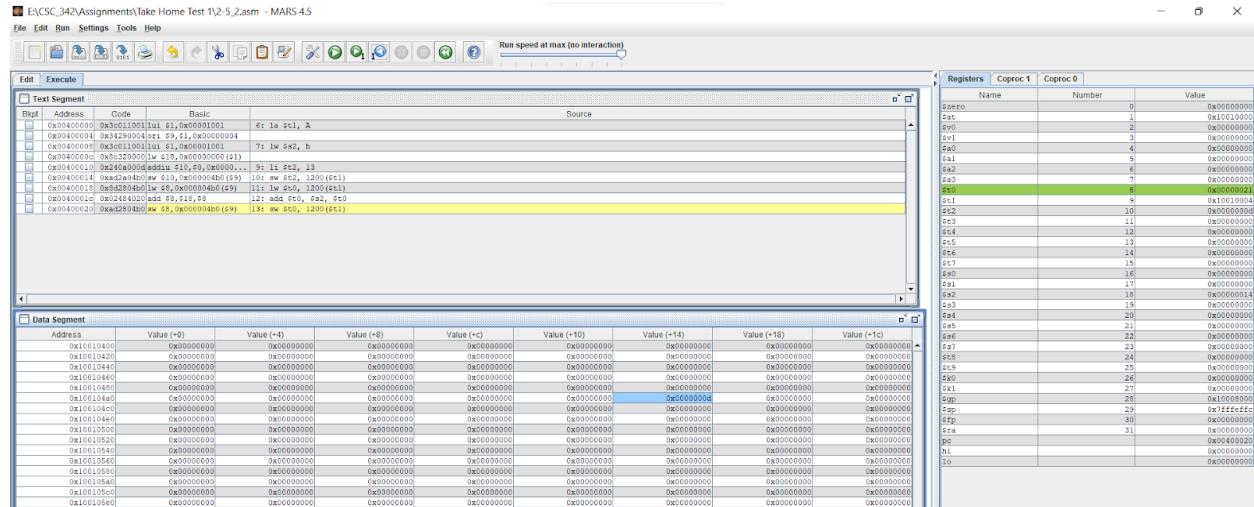
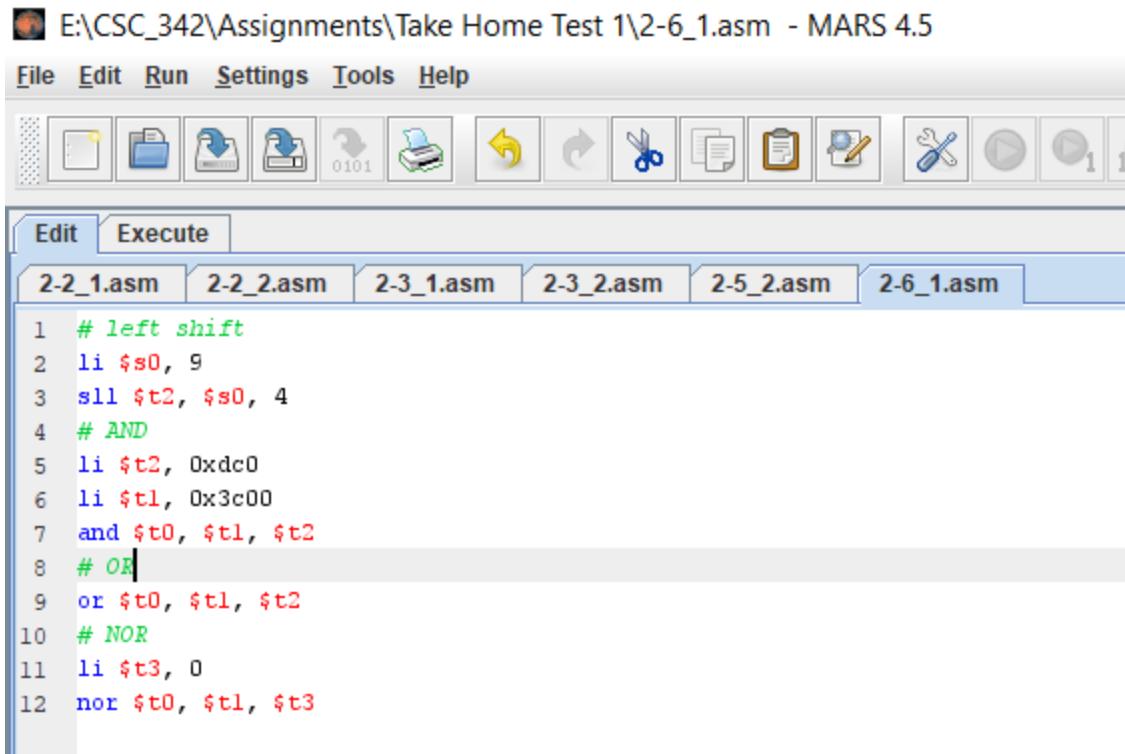


Figure 27: registers after arithmetic operations

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.



The screenshot shows the MARS 4.5 assembly editor interface. The title bar reads "E:\CSC_342\Assignments\Take Home Test 1\2-6_1.asm - MARS 4.5". The menu bar includes File, Edit, Run, Settings, Tools, and Help. Below the menu is a toolbar with various icons. The tabs at the top of the code area are labeled 2-2_1.asm, 2-2_2.asm, 2-3_1.asm, 2-3_2.asm, 2-5_2.asm, and 2-6_1.asm, with 2-6_1.asm being the active tab. The code itself is as follows:

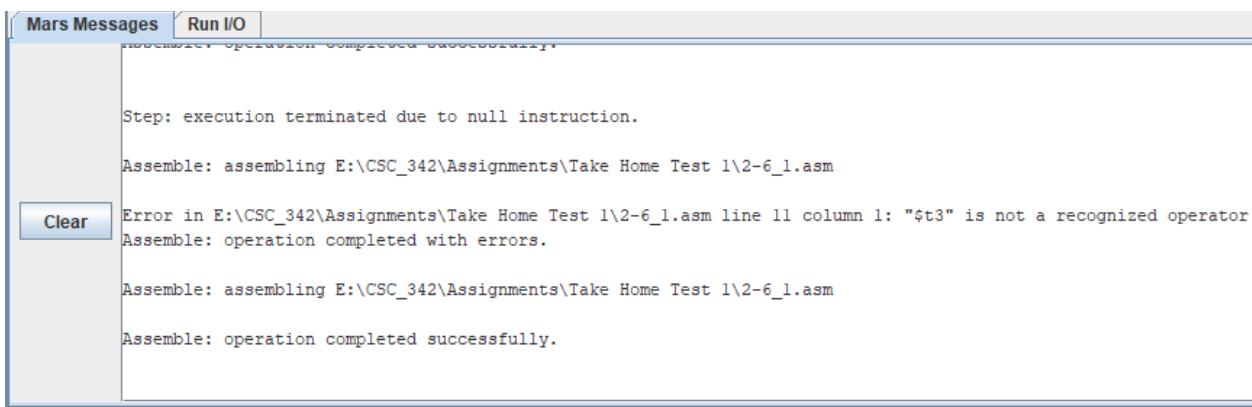
```

1 # left shift
2 li $s0, 9
3 sll $t2, $s0, 4
4 # AND
5 li $t2, 0xdc0
6 li $t1, 0x3c00
7 and $t0, $t1, $t2
8 # OR
9 or $t0, $t1, $t2
10 # NOR
11 li $t3, 0
12 nor $t0, $t1, $t3

```

Figure 28: ASM code for 2-6_1

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



The screenshot shows the Mars Messages window. It has two tabs: "Mars Messages" and "Run I/O". The "Mars Messages" tab contains the following log entries:

- Step: execution terminated due to null instruction.
- Assemble: assembling E:\CSC_342\Assignments\Take Home Test 1\2-6_1.asm
- Error in E:\CSC_342\Assignments\Take Home Test 1\2-6_1.asm line 11 column 1: "\$t3" is not a recognized operator
- Assemble: operation completed with errors.
- Assemble: assembling E:\CSC_342\Assignments\Take Home Test 1\2-6_1.asm
- Assemble: operation completed successfully.

Figure 29: Operation Completion

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

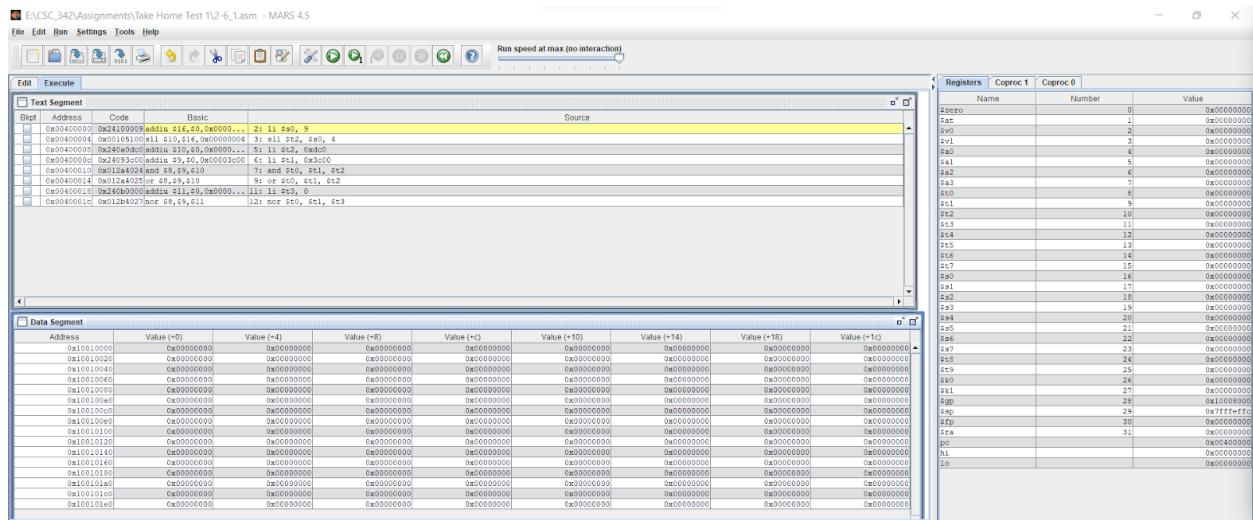


Figure 30: disassembly, memory, registers of MIPS file, 2-6_1.asm.

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.

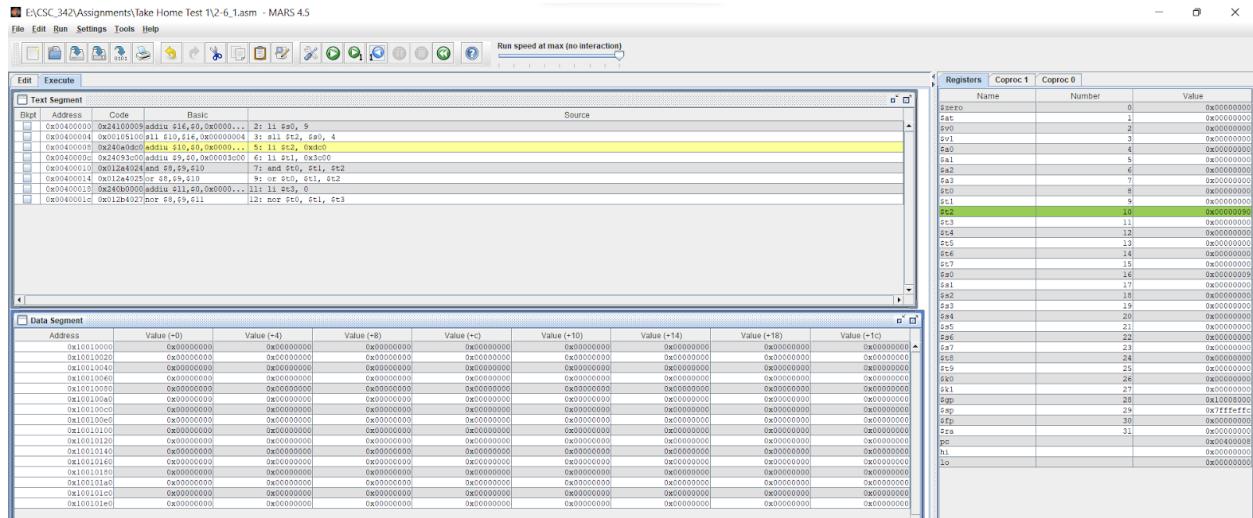


Figure 31: registers before arithmetic operations.

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.

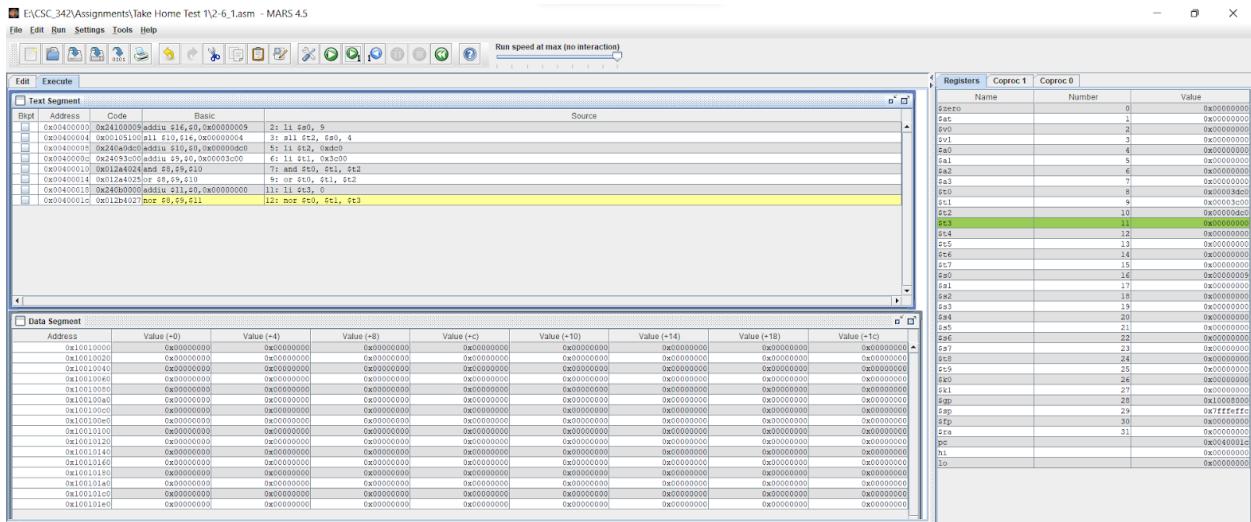
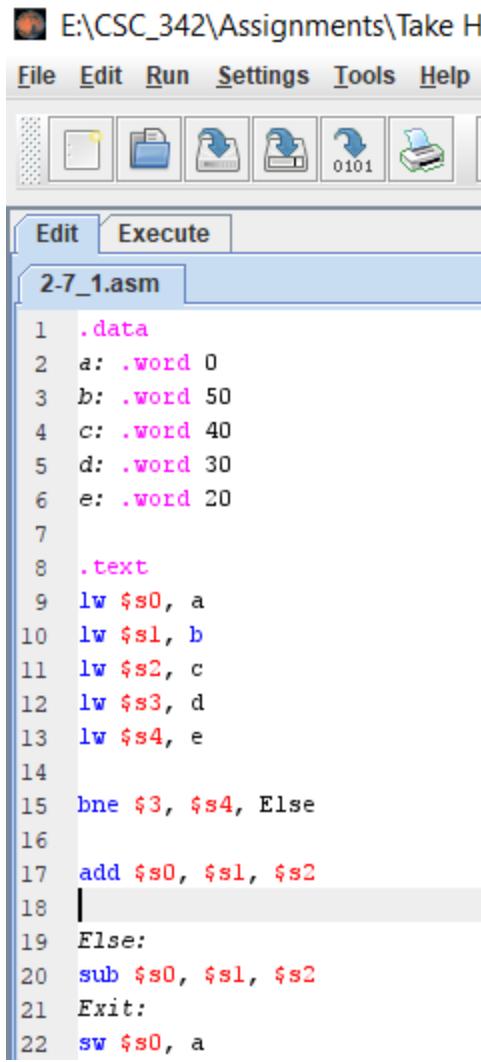


Figure 32: registers after arithmetic operations.

We can see the register value stores under the name \$t3 and the value of (+0) offset is 0x00000000.

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.



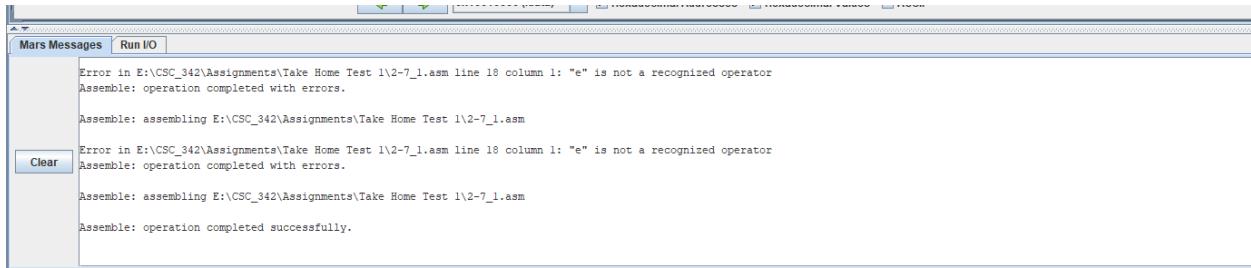
```

E:\CSC_342\Assignments\Take H
File Edit Run Settings Tools Help
Edit Execute
2-7_1.asm
1 .data
2 a: .word 0
3 b: .word 50
4 c: .word 40
5 d: .word 30
6 e: .word 20
7
8 .text
9 lw $s0, a
10 lw $s1, b
11 lw $s2, c
12 lw $s3, d
13 lw $s4, e
14
15 bne $s3, $s4, Else
16
17 add $s0, $s1, $s2
18 |
19 Else:
20 sub $s0, $s1, $s2
21 Exit:
22 sw $s0, a

```

Figure 33: ASM code for 2-7_1

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



Mars Messages | Run I/O

```

Error in E:\CSC_342\Assignments\Take Home Test 1\2-7_1.asm line 18 column 1: "e" is not a recognized operator
Assemble: operation completed with errors.

Assemble: assembling E:\CSC_342\Assignments\Take Home Test 1\2-7_1.asm

Error in E:\CSC_342\Assignments\Take Home Test 1\2-7_1.asm line 18 column 1: "e" is not a recognized operator
Assemble: operation completed with errors.

Assemble: assembling E:\CSC_342\Assignments\Take Home Test 1\2-7_1.asm
Assemble: operation completed successfully.

```

Clear

Figure 34: Operation Completion

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

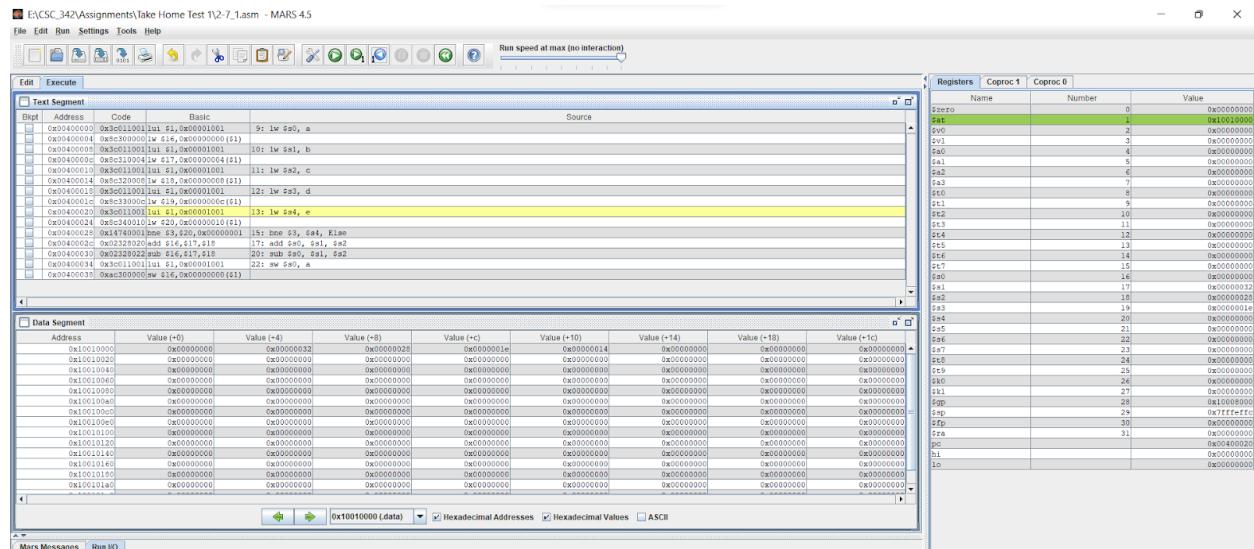


Figure 35: disassembly, memory, registers of MIPS file, 2-7_1.asm.

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.

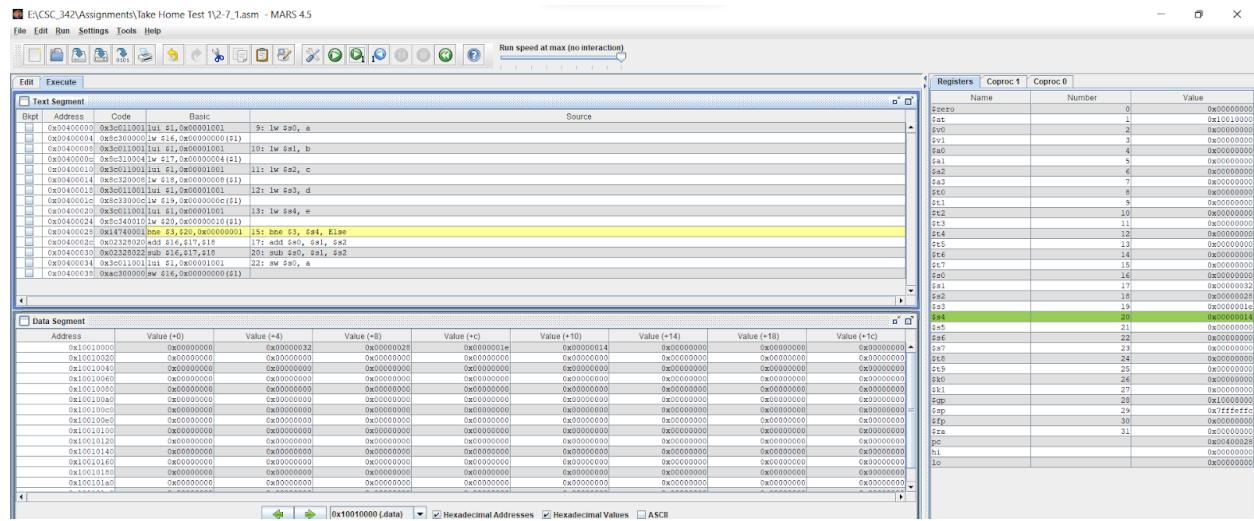
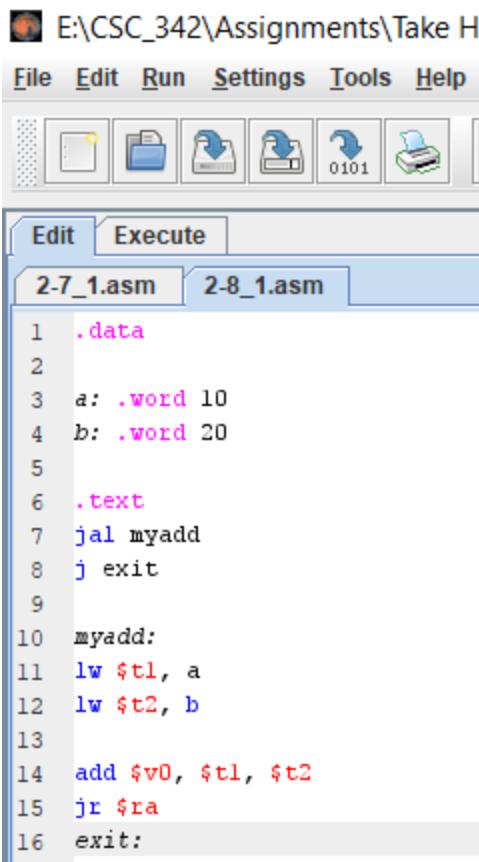


Figure 36: registers after arithmetic operations

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.



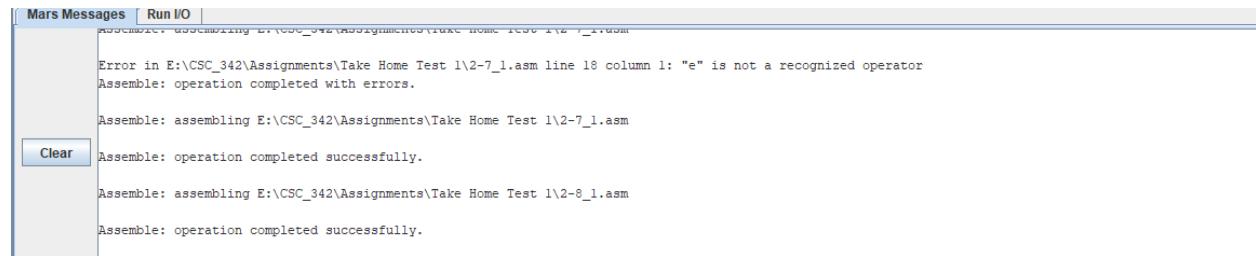
```

E:\CSC_342\Assignments\Take H
File Edit Run Settings Tools Help
[File, Edit, Run, Settings, Tools, Help icons]
Edit Execute
2-7_1.asm 2-8_1.asm
1 .data
2
3 a: .word 10
4 b: .word 20
5
6 .text
7 jal myadd
8 j exit
9
10 myadd:
11 lw $t1, a
12 lw $t2, b
13
14 add $v0, $t1, $t2
15 jr $ra
16 exit:

```

Figure 37: ASM code for 2-8_1

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



```

Mars Messages Run I/O
Error in E:\CSC_342\Assignments\Take Home Test 1\2-7_1.asm line 18 column 1: "e" is not a recognized operator
Assemble: operation completed with errors.

Assemble: assembling E:\CSC_342\Assignments\Take Home Test 1\2-7_1.asm
Clear Assemble: operation completed successfully.

Assemble: assembling E:\CSC_342\Assignments\Take Home Test 1\2-8_1.asm
Assemble: 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.

The screenshot shows the MARS 4.5 assembly editor interface. The top menu bar includes File, Edit, Run, Settings, Tools, and Help. The main window has two tabs: Text Segment and Data Segment. The Text Segment tab displays assembly code:

```

    .Biel Address Code Basic
    0x00400000 0x01000000 jal 0x00400008 7: jal myadd
    0x00400004 0x01000003 0x04000020 8: j exit
    0x00400009 0x01000001 0x00000001 11: lw $t1, a
    0x0040000c 0x01000000 0x00000000 12: add $v0, $t1, $t2
    0x00400010 0x01000001 0x00000000 13: lw $t2, b
    0x00400014 0x01000004 0x00000004 14: add $v0, $t1, $t2
    0x00400019 0x01000002 add $v0, $t1, $t2
    0x0040001c 0x01000000 jr $31 15: jr $t1
  
```

The Data Segment tab shows memory addresses from 0x10010000 to 0x10011000 with various values assigned. The Registers tab on the right lists the following register values:

| Name | Number | Value |
|--------|--------|------------|
| \$zero | 0 | 0x00000000 |
| \$at | 1 | 0x10010000 |
| \$v0 | 2 | 0x00000000 |
| \$v1 | 3 | 0x00000000 |
| \$a0 | 4 | 0x00000000 |
| \$a1 | 5 | 0x00000000 |
| \$t0 | 6 | 0x00000000 |
| \$t1 | 7 | 0x00000000 |
| \$t2 | 8 | 0x00000000 |
| \$t3 | 9 | 0x00000000 |
| \$t4 | 10 | 0x00000000 |
| \$t5 | 11 | 0x00000000 |
| \$t6 | 12 | 0x00000000 |
| \$t7 | 13 | 0x00000000 |
| \$t8 | 14 | 0x00000000 |
| \$t9 | 15 | 0x00000000 |
| \$t10 | 16 | 0x00000000 |
| \$t11 | 17 | 0x00000000 |
| \$t12 | 18 | 0x00000000 |
| \$t13 | 19 | 0x00000000 |
| \$t14 | 20 | 0x00000000 |
| \$t15 | 21 | 0x00000000 |
| \$t16 | 22 | 0x00000000 |
| \$t17 | 23 | 0x00000000 |
| \$t18 | 24 | 0x00000000 |
| \$t19 | 25 | 0x00000000 |
| \$t20 | 26 | 0x00000000 |
| \$t21 | 27 | 0x00000000 |
| \$t22 | 28 | 0x10010000 |
| \$t23 | 29 | 0x7fffffc0 |
| \$t24 | 30 | 0x00000000 |
| \$t25 | 31 | 0x00000004 |
| \$pc | | 0x00400010 |
| \$hi | | 0x00000000 |
| \$lo | | 0x00000000 |

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.

The screenshot shows the MARS 4.5 assembly editor interface. The Text Segment tab displays assembly code identical to Figure 39. The Data Segment tab shows memory addresses from 0x10010000 to 0x10011000 with various values assigned. The Registers tab on the right lists the following register values:

| Name | Number | Value |
|--------|--------|------------|
| \$zero | 0 | 0x00000000 |
| \$at | 1 | 0x10010000 |
| \$v0 | 2 | 0x00000001 |
| \$v1 | 3 | 0x00000000 |
| \$a0 | 4 | 0x00000000 |
| \$a1 | 5 | 0x00000000 |
| \$t0 | 6 | 0x00000000 |
| \$t1 | 7 | 0x00000000 |
| \$t2 | 8 | 0x00000000 |
| \$t3 | 9 | 0x00000000 |
| \$t4 | 10 | 0x00000014 |
| \$t5 | 11 | 0x00000000 |
| \$t6 | 12 | 0x00000000 |
| \$t7 | 13 | 0x00000000 |
| \$t8 | 14 | 0x00000000 |
| \$t9 | 15 | 0x00000000 |
| \$t10 | 16 | 0x00000000 |
| \$t11 | 17 | 0x00000000 |
| \$t12 | 18 | 0x00000000 |
| \$t13 | 19 | 0x00000000 |
| \$t14 | 20 | 0x00000000 |
| \$t15 | 21 | 0x00000000 |
| \$t16 | 22 | 0x00000000 |
| \$t17 | 23 | 0x00000000 |
| \$t18 | 24 | 0x00000000 |
| \$t19 | 25 | 0x00000000 |
| \$t20 | 26 | 0x00000000 |
| \$t21 | 27 | 0x00000000 |
| \$t22 | 28 | 0x10010000 |
| \$t23 | 29 | 0x7fffffc0 |
| \$t24 | 30 | 0x00000000 |
| \$t25 | 31 | 0x00000004 |
| \$pc | | 0x00400010 |
| \$hi | | 0x00000000 |
| \$lo | | 0x00000000 |

Figure 40: registers after arithmetic operations

We can see the register value stores under the name \$v0 and address is 0x00000001e.

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.

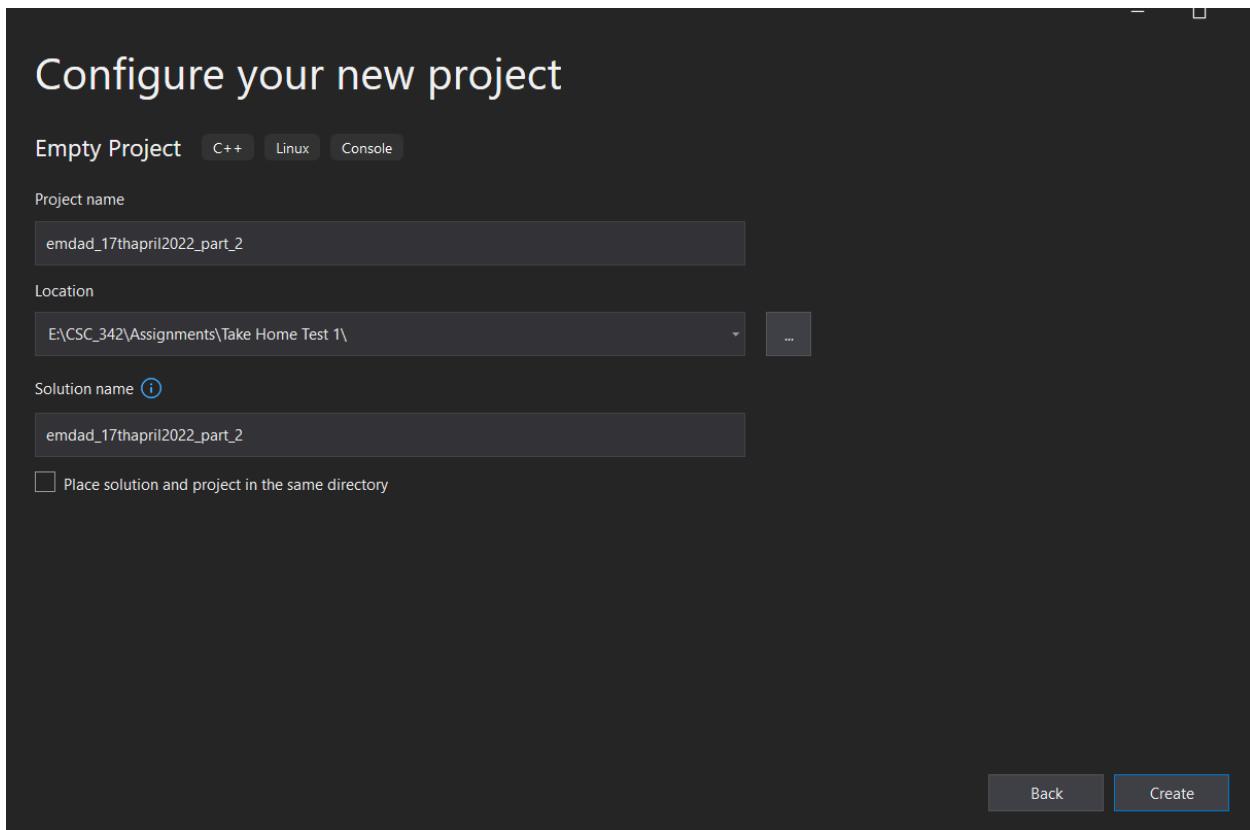


Figure 41: project directory for the 2-2_1.c.

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

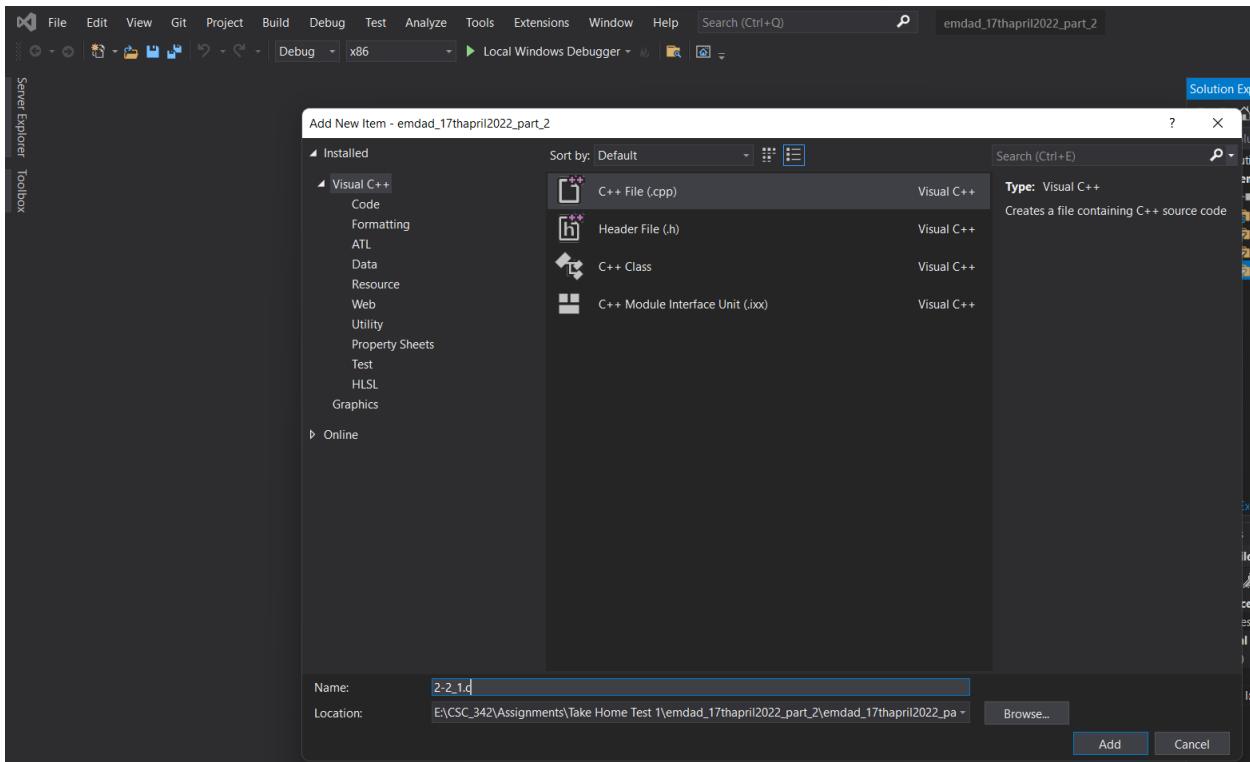


Figure 42: adding 2-2_1.c file in the project

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

```

1 int main() {
2     static int a = 1;
3     static int b = 2;
4     static int c = 3;
5     static int d = 4;
6     static int e = 5;
7     a = b + c;
8     d = a - e;
9 }
10 }
```

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

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

The screenshot shows the 'Output' tab of a debugger interface. The status bar at the top indicates '100 %' and 'No issues found'. The main area displays the following text:

```
'emdad_17thapril2022_part_2.exe' (Win32): Loaded 'E:\CSC_342\Assignments\Take Home Test 1\emdad_17thapril2022_part_2\Debug\emdad_17thapril2022_part_2.exe'. Symbols loaded.
.emdad_17thapril2022_part_2.exe' (Win32): Loaded 'C:\Windows\SysWOW64\ntdll.dll'.
.emdad_17thapril2022_part_2.exe' (Win32): Loaded 'C:\Windows\SysWOW64\kernel32.dll'.
.emdad_17thapril2022_part_2.exe' (Win32): Loaded 'C:\Windows\SysWOW64\KernelBase.dll'.
.emdad_17thapril2022_part_2.exe' (Win32): Loaded 'C:\Windows\SysWOW64\vruntime140d.dll'.
.emdad_17thapril2022_part_2.exe' (Win32): Loaded 'C:\Windows\SysWOW64\ucrtbased.dll'.
The thread 0x4890 has exited with code 0 (0x0).
.emdad_17thapril2022_part_2.exe' (Win32): Loaded 'C:\Windows\SysWOW64\kernel.appcore.dll'.
.emdad_17thapril2022_part_2.exe' (Win32): Loaded 'C:\Windows\SysWOW64\msvcrt.dll'.
The thread 0x2780 has exited with code 0 (0x0).
The program '[4524] emdad_17thapril2022_part_2.exe' has exited with code 0 (0x0).
```

Figure 44: Built successfully

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

The screenshot shows the 'Disassembly' tab of a debugger interface. The assembly code is as follows:

```
--- E:\CSC_342\Assignments\Take Home Test 1\emdad_17thapril2022_part_2\emdad_17thapril2022_part_2\2-2_1.c

int main() {
    005E1750 push    ebp
    005E1751 mov     ebp,esp
    005E1753 sub    esp,0C0h
    005E1755 push    ebx
    005E175A push    esi
    005E175B push    edi
    005E175C mov     edi,ebp
    005E175E xor     ecx,ecx
    005E1760 mov     eax,0CCCCCCCCh
    005E1765 rep stos dword ptr es:[edi]
    005E1767 mov     ecx,offset _1EA4A43E_2-2_1@c (05EC000h)
    005E176C call    @_CheckForDebuggerJustMyCode@4 (05E1307h)

    static int a = 1;
    static int b = 2;
    static int c = 3;
    static int d = 4;
    static int e = 5;
    a = b + c;
    005E1771 mov     eax,dword ptr [b (05EA004h)]
    005E1776 add     eax,dword ptr [c (05EA008h)]
    005E177C mov     dword ptr [a (05EA000h)],eax
    d = a - e;
    005E1781 mov     eax,dword ptr [a (05EA000h)]
    005E1786 sub     eax,dword ptr [e (05EA010h)]
    005E178C mov     dword ptr [d (05EA00Ch)],eax
}
```

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.

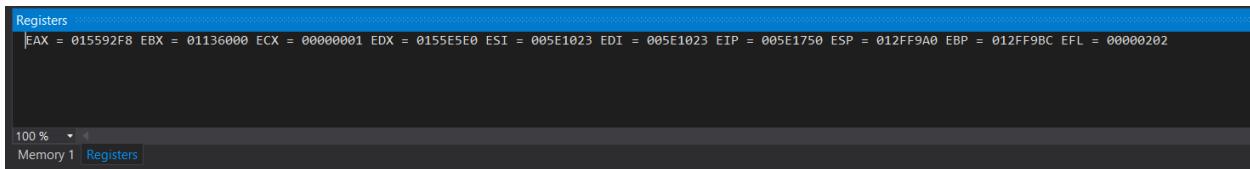


Figure 46: register window.

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.

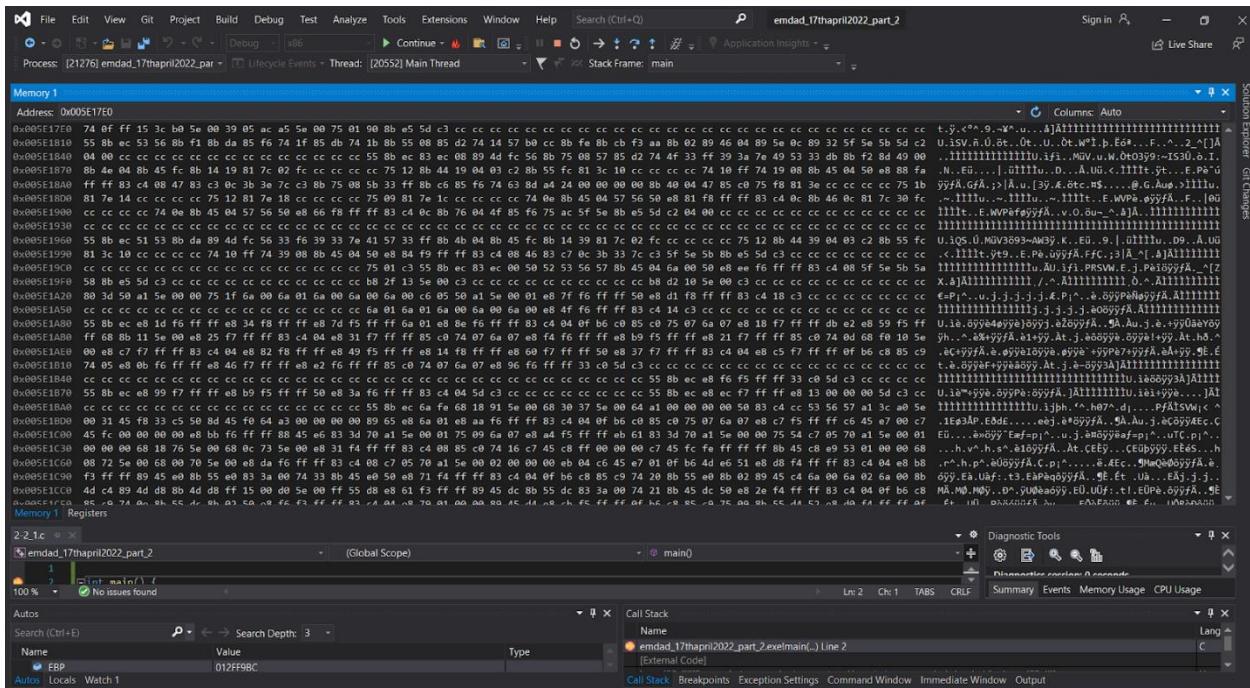


Figure 47: memory window

2-2_2:

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

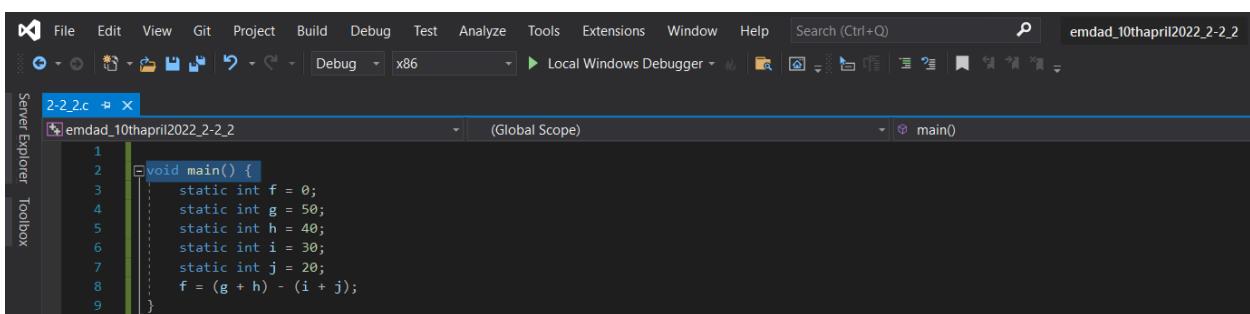


Figure 48: source code for the 2-2_2.c.

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

```

Output
Show output from: Debug
'emdad_10thapril2022_2-2_2.exe' (Win32): Loaded 'E:\CSC_342\Assignments\Take Home Test 1\emdad_10thapril2022_2-2_2\Debug\emdad_10thapril2022_2-2_2.exe'. Symbols loaded.
'emdad_10thapril2022_2-2_2.exe' (Win32): Loaded 'C:\Windows\SysWOW64\ntdll.dll'.
'emdad_10thapril2022_2-2_2.exe' (Win32): Loaded 'C:\Windows\SysWOW64\kernel32.dll'.
'emdad_10thapril2022_2-2_2.exe' (Win32): Loaded 'C:\Windows\SysWOW64\kernelBase.dll'.
'emdad_10thapril2022_2-2_2.exe' (Win32): Loaded 'C:\Windows\SysWOW64\apphelp.dll'.
'emdad_10thapril2022_2-2_2.exe' (Win32): Loaded 'C:\Windows\SysWOW64\vcruntime140d.dll'.
'emdad_10thapril2022_2-2_2.exe' (Win32): Loaded 'C:\Windows\SysWOW64\ucrtbased.dll'.
The thread 0x678 has exited with code 0 (0x0).
'emdad_10thapril2022_2-2_2.exe' (Win32): Loaded 'C:\Windows\SysWOW64\kernel.appcore.dll'.
'emdad_10thapril2022_2-2_2.exe' (Win32): Loaded 'C:\Windows\SysWOW64\msvcrtd.dll'.
The thread 0x1e68 has exited with code 0 (0x0).
The thread 0x4744 has exited with code 0 (0x0).
The program '[10788] emdad_10thapril2022_2-2_2.exe' has exited with code 0 (0x0).

```

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.

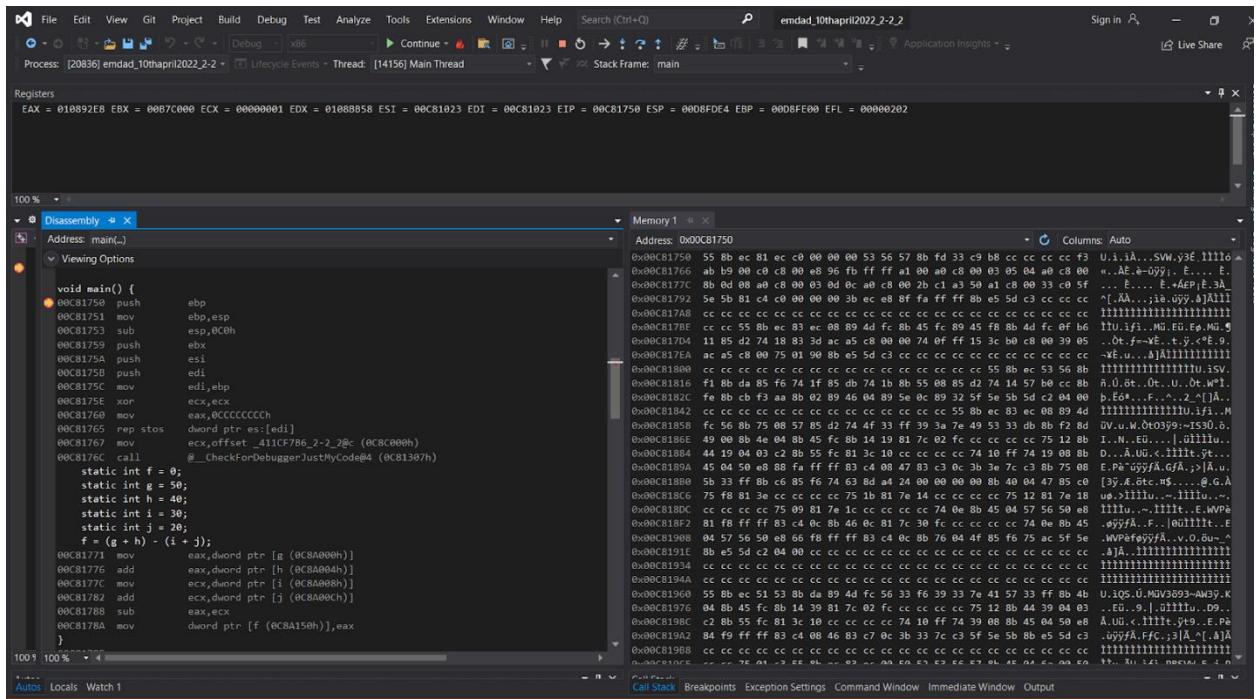


Figure 50: register, disassembly, memory window

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.

```

2-3_1.cpp 2-3_1
emdad_10thapril2022_2-3_1 (Global Scope) main()
1 void main() {
2     static int g = 0;
3     static int h = 22;
4     static int A[100];
5     A[8] = 55;
6     g = h + A[8];
7 }

```

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

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

```

Output
Show output from: Debug
emdad_10thapril2022_2-3_1.exe' (Win32): Loaded 'E:\CSC_342\Assignments\Take Home Test 1\emdad_10thapril2022_2-3_1\Debug\emdad_10thapril2022_2-3_1.exe'. Symbols loaded.
'emdad_10thapril2022_2-3_1.exe' (Win32): Loaded 'C:\Windows\SysWOW64\ntdll.dll'.
'emdad_10thapril2022_2-3_1.exe' (Win32): Loaded 'C:\Windows\SysWOW64\kernel32.dll'.
'emdad_10thapril2022_2-3_1.exe' (Win32): Loaded 'C:\Windows\SysWOW64\KernelBase.dll'.
'emdad_10thapril2022_2-3_1.exe' (Win32): Loaded 'C:\Windows\SysWOW64\apphelp.dll'.
'emdad_10thapril2022_2-3_1.exe' (Win32): Loaded 'C:\Windows\SysWOW64\vruntimei40d.dll'.
'emdad_10thapril2022_2-3_1.exe' (Win32): Loaded 'C:\Windows\SysWOW64\ucrtbased.dll'.
The thread 0x5668 has exited with code 0 (0x0).
'emdad_10thapril2022_2-3_1.exe' (Win32): Loaded 'C:\Windows\SysWOW64\kernel.appcore.dll'.
'emdad_10thapril2022_2-3_1.exe' (Win32): Loaded 'C:\Windows\SysWOW64\msvcrt.dll'.
The thread 0x4d30 has exited with code 0 (0x0).
The thread 0x9e4 has exited with code 0 (0x0).
The program '[14724] emdad_10thapril2022_2-3_1.exe' has exited with code 0 (0x0).

```

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.

```
2-3_2.c
void main() {
    static int h = 25;
    static int A[100];
    A[8] = 200;
    A[12] = h + A[8];
}
```

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

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

```
Show output from: Debug
' emdad_10thapril2022_2-3_2.exe' (Win32): Loaded 'E:\CSC_342\Assignments\Take Home Test 1\emdad_10thapril2022_2-3_2\Debug\emdad_10thapril2022_2-3_2.exe'. Symbols loaded.
' emdad_10thapril2022_2-3_2.exe' (Win32): Loaded 'C:\Windows\SysWOW64\ntdll.dll'.
' emdad_10thapril2022_2-3_2.exe' (Win32): Loaded 'C:\Windows\SysWOW64\kernel32.dll'.
' emdad_10thapril2022_2-3_2.exe' (Win32): Loaded 'C:\Windows\SysWOW64\kernelbase.dll'.
' emdad_10thapril2022_2-3_2.exe' (Win32): Loaded 'C:\Windows\SysWOW64\apphelp.dll'.
' emdad_10thapril2022_2-3_2.exe' (Win32): Loaded 'C:\Windows\SysWOW64\vcruntime140d.dll'.
' emdad_10thapril2022_2-3_2.exe' (Win32): Loaded 'C:\Windows\SysWOW64\ucrtbased.dll'.
The thread 0x57cc has exited with code 0 (0x0).
' emdad_10thapril2022_2-3_2.exe' (Win32): Loaded 'C:\Windows\SysWOW64\kernel.appcore.dll'.
' emdad_10thapril2022_2-3_2.exe' (Win32): Loaded 'C:\Windows\SysWOW64\msvcrt.dll'.
The thread 0x2408 has exited with code 0 (0x0).
The thread 0x574 has exited with code 0 (0x0).
The program '[1748] emdad_10thapril2022_2-3_2.exe' has exited with code 0 (0x0).
```

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.

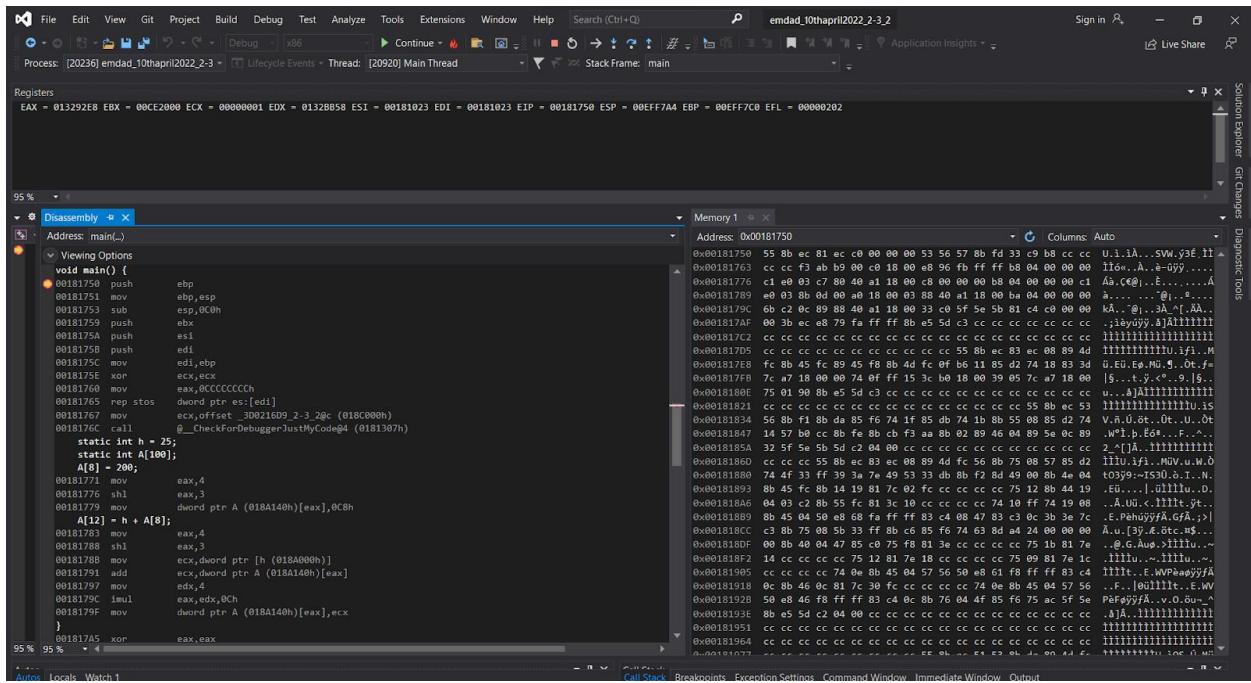


Figure 56: register, disassembly, memory window

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.

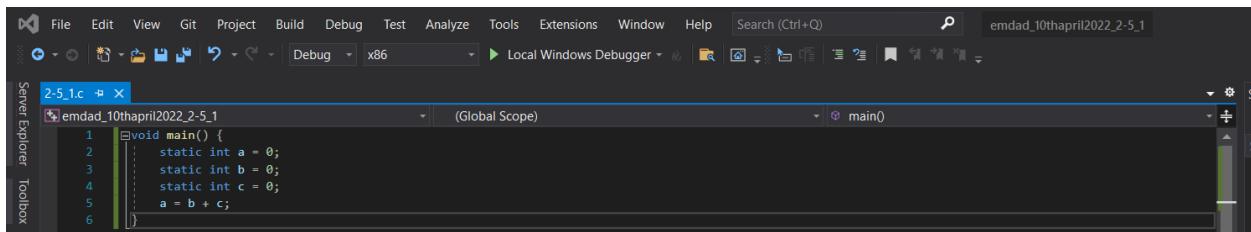


Figure 57: source code for the 2-5_1.c

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

```

Output
Show output from: Debug
'emdad_10thapril2022_2-5_1.exe' (Win32): Loaded 'E:\CSC_342\Assignments\Take Home Test 1\emdad_10thapril2022_2-5_1\Debug\emdad_10thapril2022_2-5_1.exe'. Symbols loaded.
'emdad_10thapril2022_2-5_1.exe' (Win32): Loaded 'C:\Windows\SysWOW64\ntdll.dll'.
'emdad_10thapril2022_2-5_1.exe' (Win32): Loaded 'C:\Windows\SysWOW64\kernel32.dll'.
'emdad_10thapril2022_2-5_1.exe' (Win32): Loaded 'C:\Windows\SysWOW64\kernelbase.dll'.
'emdad_10thapril2022_2-5_1.exe' (Win32): Loaded 'C:\Windows\SysWOW64\apphelp.dll'.
'emdad_10thapril2022_2-5_1.exe' (Win32): Loaded 'C:\Windows\SysWOW64\vruntime.dll'.
'emdad_10thapril2022_2-5_1.exe' (Win32): Loaded 'C:\Windows\SysWOW64\ucrtbased.dll'.

The thread 0x453c has exited with code 0 (0x0).
'emdad_10thapril2022_2-5_1.exe' (Win32): Loaded 'C:\Windows\SysWOW64\kernel.appcore.dll'.
'emdad_10thapril2022_2-5_1.exe' (Win32): Loaded 'C:\Windows\SysWOW64\msvcrtd.dll'.
The thread 0x56d0 has exited with code 0 (0x0).
The thread 0x4bec has exited with code 0 (0x0).

The program '[12400] emdad_10thapril2022_2-5_1.exe' has exited with code 0 (0x0).

```

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.

The screenshot shows the Visual Studio debugger interface with the following details:

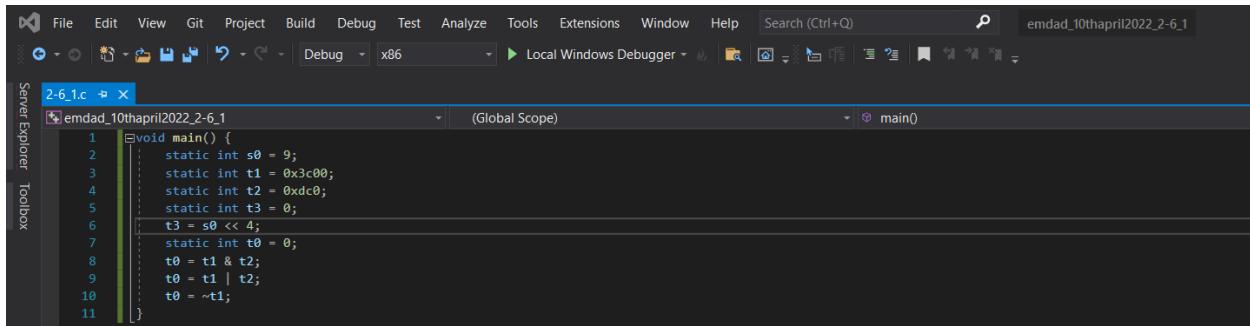
- Registers:** Shows CPU registers (EAX=015492E8, ECX=010A4900, EDX=0154E5D0, etc.)
- Disassembly:** Shows assembly code for the main() function, including instructions like push, mov, add, and call.
- Memory:** Shows memory dump at address 0x00841750, displaying binary data and ASCII representation.

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.



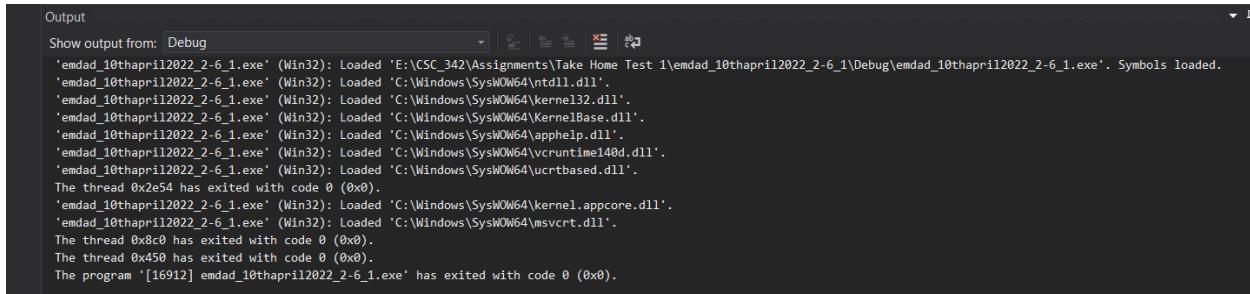
```

2-6_1.c 2-6_1
emdad_10thapril2022_2-6_1 (Global Scope) main()
1 void main() {
2     static int s0 = 9;
3     static int t1 = 0x3c00;
4     static int t2 = 0xdc0;
5     static int t3 = 0;
6     t3 = s0 << 4;
7     static int t0 = 0;
8     t0 = t1 & t2;
9     t0 = t1 | t2;
10    t0 = ~t1;
11 }

```

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

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



```

Output
Show output from: Debug
emdad_10thapril2022_2-6_1.exe' (Win32): Loaded 'E:\CSC_342\Assignments\Take Home Test 1\emdad_10thapril2022_2-6_1\Debug\emdad_10thapril2022_2-6_1.exe'. Symbols loaded.
'emdad_10thapril2022_2-6_1.exe' (Win32): Loaded 'C:\Windows\SysWOW64\ntdll.dll'.
'emdad_10thapril2022_2-6_1.exe' (Win32): Loaded 'C:\Windows\SysWOW64\kernel32.dll'.
'emdad_10thapril2022_2-6_1.exe' (Win32): Loaded 'C:\Windows\SysWOW64\kernelBase.dll'.
'emdad_10thapril2022_2-6_1.exe' (Win32): Loaded 'C:\Windows\SysWOW64\apphelp.dll'.
'emdad_10thapril2022_2-6_1.exe' (Win32): Loaded 'C:\Windows\SysWOW64\vcruntime140d.dll'.
'emdad_10thapril2022_2-6_1.exe' (Win32): Loaded 'C:\Windows\SysWOW64\ucrtbased.dll'.
The thread 0x2e54 has exited with code 0 (0x0).
'emdad_10thapril2022_2-6_1.exe' (Win32): Loaded 'C:\Windows\SysWOW64\kernel.appcore.dll'.
'emdad_10thapril2022_2-6_1.exe' (Win32): Loaded 'C:\Windows\SysWOW64\msvcrtd.dll'.
The thread 0x8c0 has exited with code 0 (0x0).
The thread 0x450 has exited with code 0 (0x0).
The program '[16912] emdad_10thapril2022_2-6_1.exe' has exited with code 0 (0x0).

```

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.

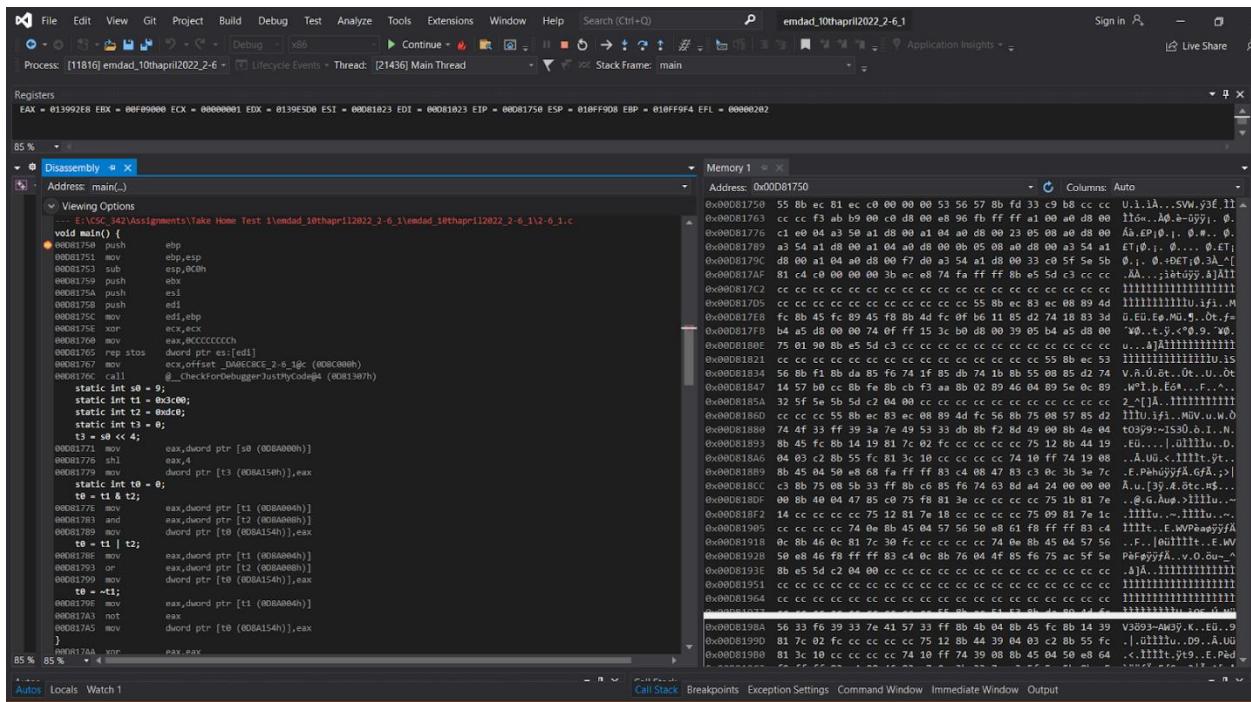


Figure 62: register, disassembly, memory window

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.

```

2-6_2.c
1 #include <stdio.h>
2
3 {
4     int a = 1;
5     static int b = -1;
6     b += 1;
7     return a + b;
8 }
9
10 int main()
11 {
12     printf("%d\n", natural_generator());
13     printf("%d\n", natural_generator());
14     printf("%d\n", natural_generator());
15     return 0;
}

```

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

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

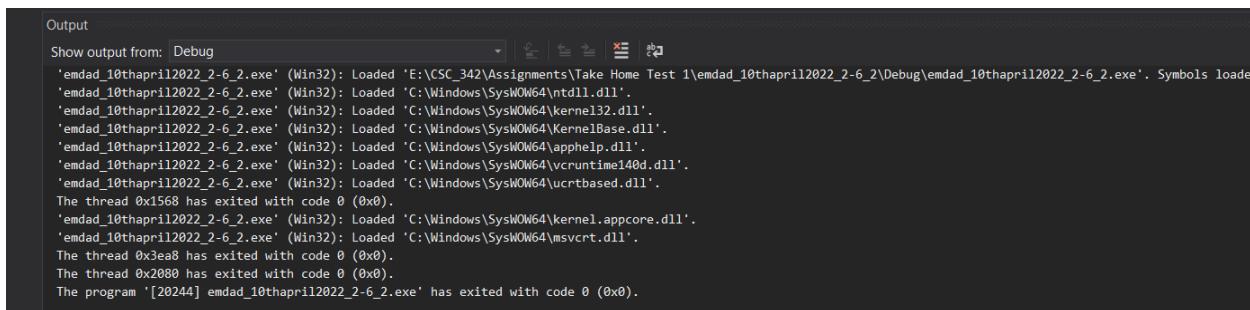


Figure 64: Built successfully

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

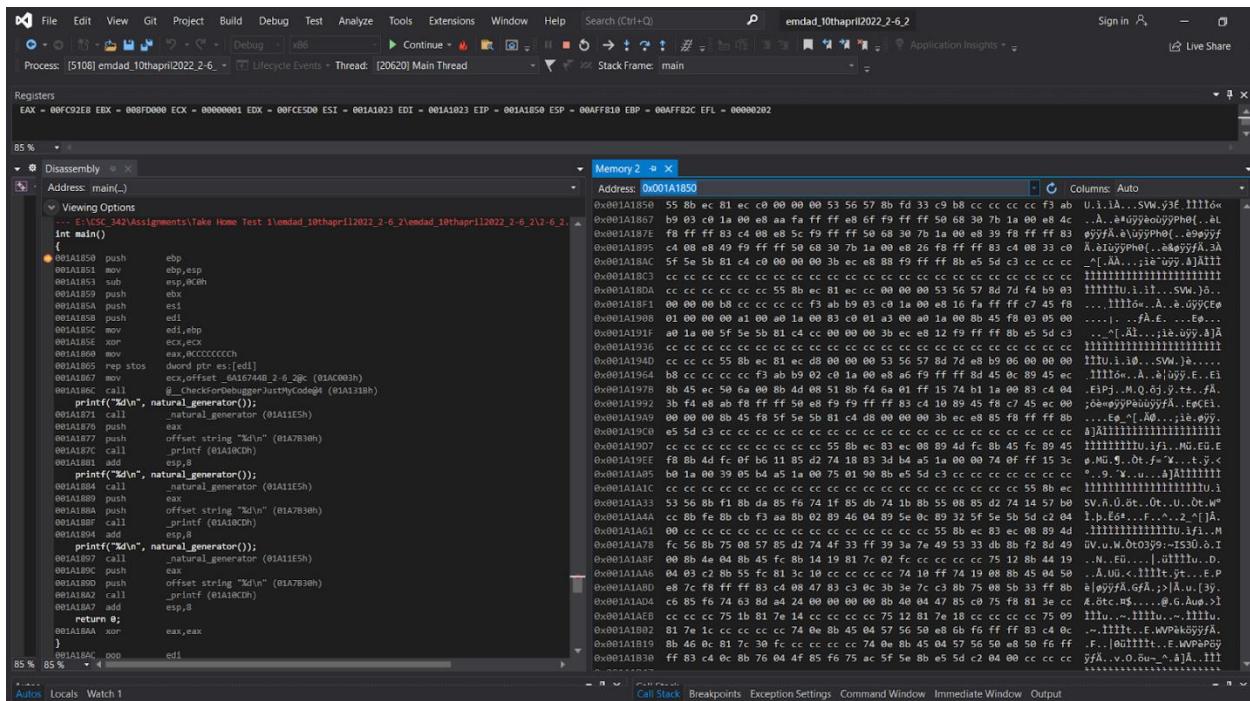
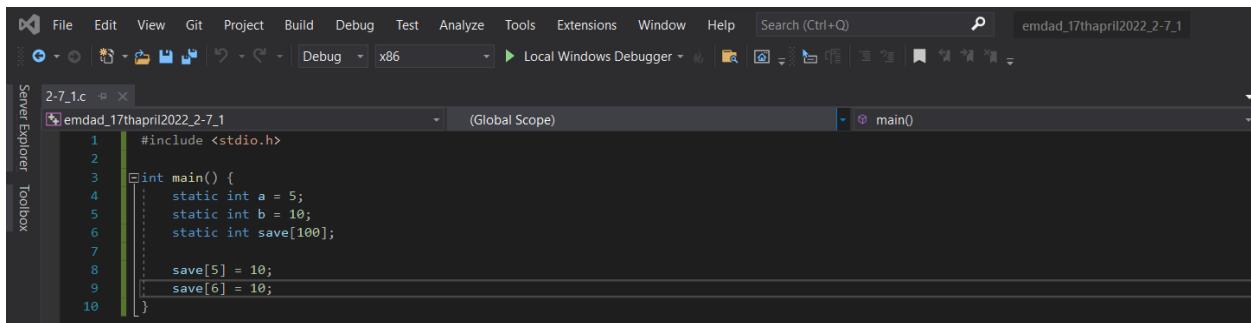


Figure 65: register, disassembly, memory window

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.



```

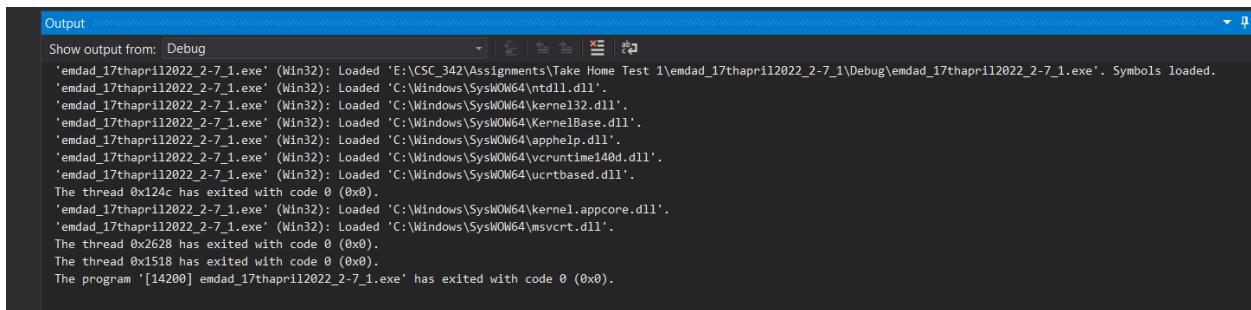
2-7_1.c  emdad_17thapril2022_2-7_1
2-7_1.c  emdad_17thapril2022_2-7_1.h
2-7_1.c  Global Scope
2-7_1.c  main()

1 #include <stdio.h>
2
3 int main() {
4     static int a = 5;
5     static int b = 10;
6     static int save[100];
7
8     save[5] = 10;
9     save[6] = 10;
10 }

```

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

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



```

Output
Show output from: Debug
' emdad_17thapril2022_2-7_1.exe' (Win32): Loaded 'E:\CSC_342\Assignments\Take Home Test 1\emdad_17thapril2022_2-7_1\Debug\emdad_17thapril2022_2-7_1.exe'. Symbols loaded.
' emdad_17thapril2022_2-7_1.exe' (Win32): Loaded 'C:\Windows\SysWOW64\ntdll.dll'.
' emdad_17thapril2022_2-7_1.exe' (Win32): Loaded 'C:\Windows\SysWOW64\kernel32.dll'.
' emdad_17thapril2022_2-7_1.exe' (Win32): Loaded 'C:\Windows\SysWOW64\kernelBase.dll'.
' emdad_17thapril2022_2-7_1.exe' (Win32): Loaded 'C:\Windows\SysWOW64\apphelp.dll'.
' emdad_17thapril2022_2-7_1.exe' (Win32): Loaded 'C:\Windows\SysWOW64\vcruntime140d.dll'.
' emdad_17thapril2022_2-7_1.exe' (Win32): Loaded 'C:\Windows\SysWOW64\ucrtbased.dll'.
The thread 0x124c has exited with code 0 (0x0).
' emdad_17thapril2022_2-7_1.exe' (Win32): Loaded 'C:\Windows\SysWOW64\kernel.appcore.dll'.
' emdad_17thapril2022_2-7_1.exe' (Win32): Loaded 'C:\Windows\SysWOW64\msvcrt.dll'.
The thread 0x2628 has exited with code 0 (0x0).
The thread 0x1518 has exited with code 0 (0x0).
The program '[14200] emdad_17thapril2022_2-7_1.exe' has exited with code 0 (0x0).

```

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.

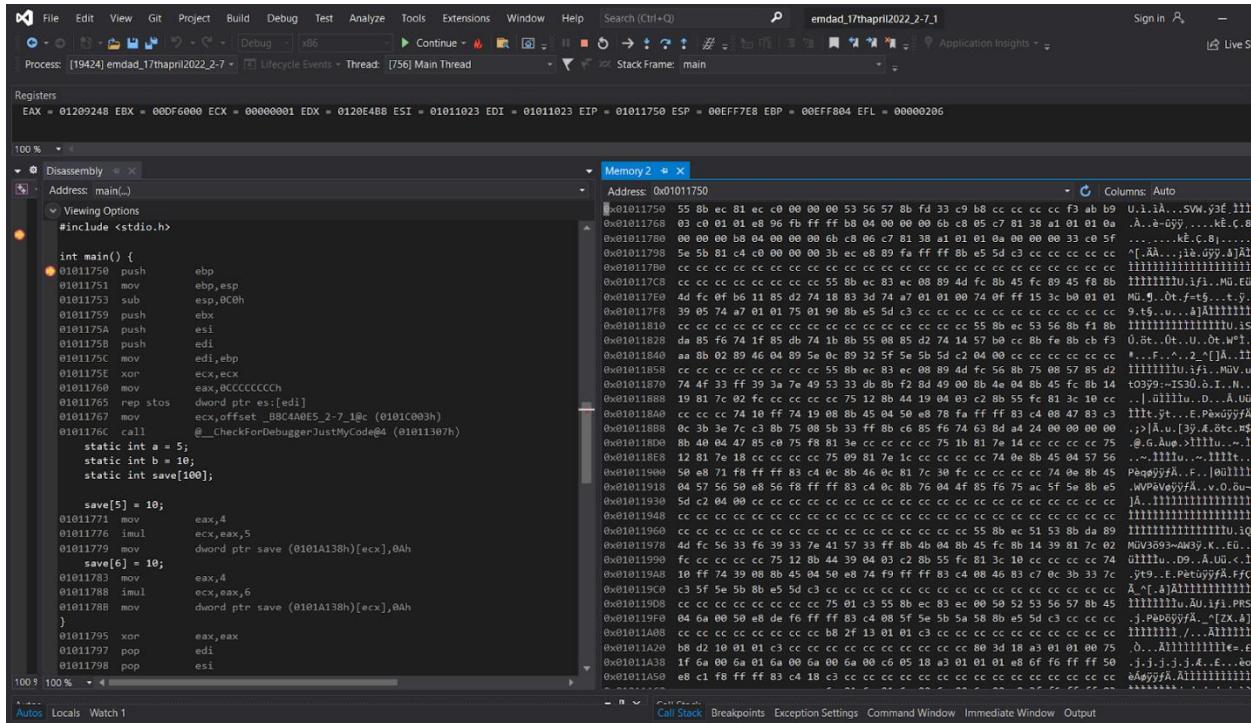


Figure 68: register, disassembly, memory window

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.

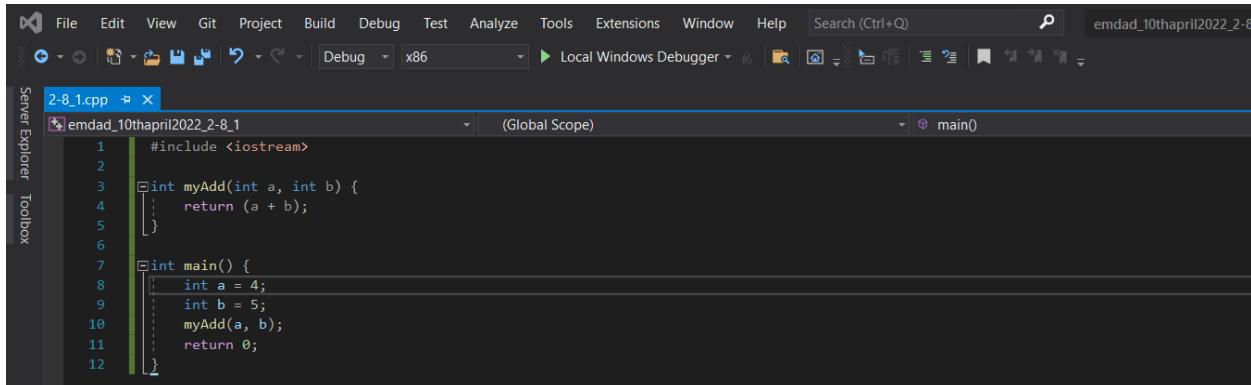


Figure 69: source code for the 2-8_1.c.

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

```

Output
Show output from: Debug
'emedad_10thapril2022_2-8_1.exe' (Win32): Loaded 'E:\CSC_342\Assignments\Take Home Test 1\emedad_10thapril2022_2-8_1\Debug\emedad_10thapril2022_2-8_1.exe', Symbols loaded.
'emedad_10thapril2022_2-8_1.exe' (Win32): Loaded 'C:\Windows\SysWOW64\ntdll.dll'.
'emedad_10thapril2022_2-8_1.exe' (Win32): Loaded 'C:\Windows\SysWOW64\kernel32.dll'.
'emedad_10thapril2022_2-8_1.exe' (Win32): Loaded 'C:\Windows\SysWOW64\KernelBase.dll'.
'emedad_10thapril2022_2-8_1.exe' (Win32): Loaded 'C:\Windows\SysWOW64\apphelp.dll'.
'emedad_10thapril2022_2-8_1.exe' (Win32): Loaded 'C:\Windows\SysWOW64\vcruntime140d.dll'.
'emedad_10thapril2022_2-8_1.exe' (Win32): Loaded 'C:\Windows\SysWOW64\ucrtbased.dll'.
The thread 0x514c has exited with code 0 (0x0).
'emedad_10thapril2022_2-8_1.exe' (Win32): Loaded 'C:\Windows\SysWOW64\kernel.appcore.dll'.
'emedad_10thapril2022_2-8_1.exe' (Win32): Loaded 'C:\Windows\SysWOW64\msvcrtd.dll'.
The thread 0xd04 has exited with code 0 (0x0).
The thread 0x5674 has exited with code 0 (0x0).
The program '[20612] emdad_10thapril2022_2-8_1.exe' has exited with code 0 (0x0).

```

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.

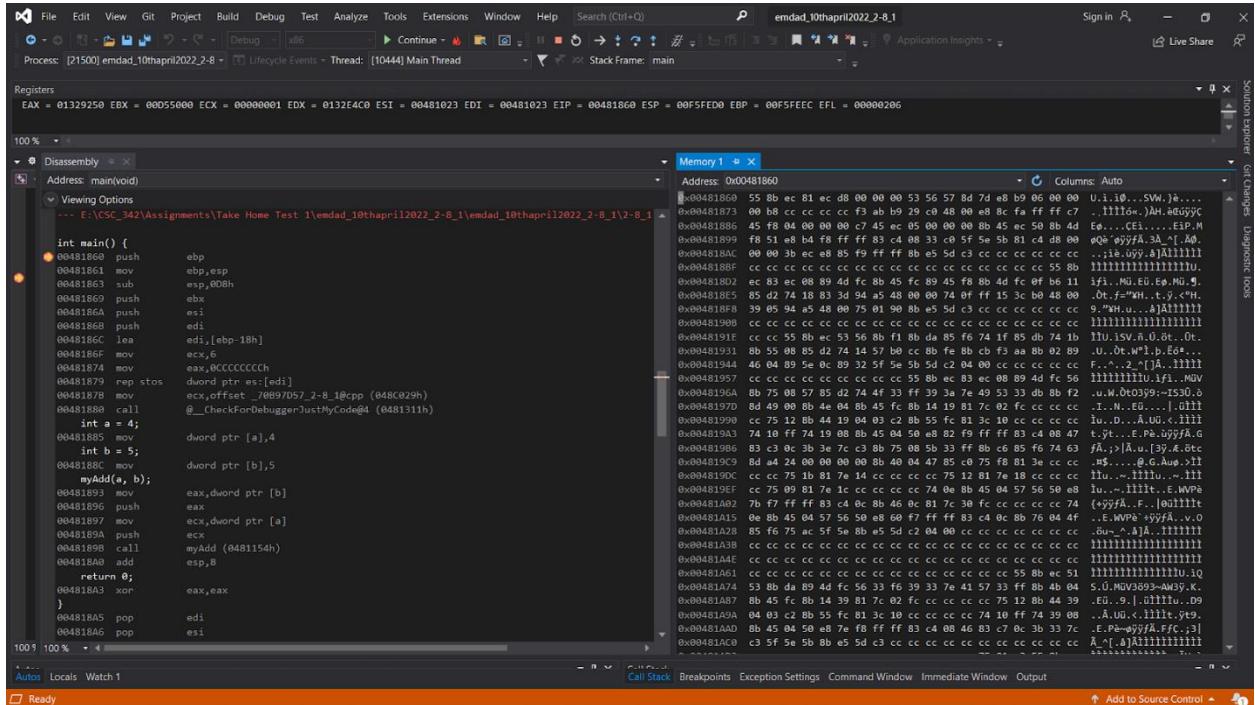


Figure 71: register, disassembly, memory window

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.

The screenshot shows a code editor interface with multiple tabs. The active tab is titled "emdad_17thapril2022_2-2_1.c". The code within this tab is as follows:

```
1 #include <stdio.h>
2
3 int main() {
4     static int a = 1;
5     static int b = 2;
6     static int c = 3;
7     static int d = 4;
8     static int e = 5;
9     a = b + c;
10    d = a - 1;
11 }
```

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 arithmetic 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.

```
(gdb) run
Starting program: /home/main/Desktop/cs-343/takeHomeTest-1/a.out

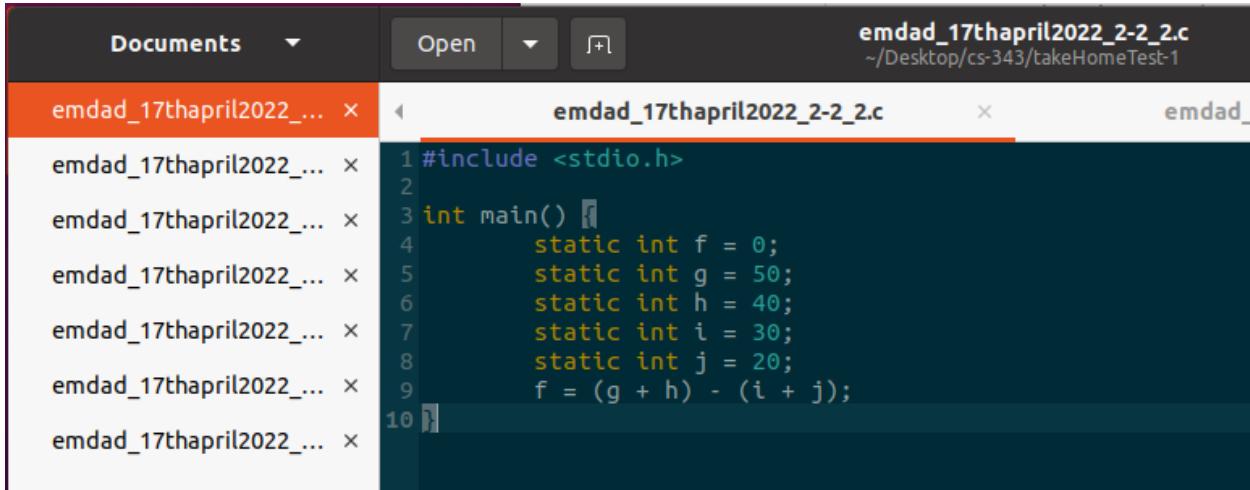
Breakpoint 1, 0x00005555555555129 in main ()
(gdb) select-frame 0
(gdb) disassemble
Dump of assembler code for function main:
=> 0x00005555555555129 <+0>:    endbr64
    0x0000555555555512d <+4>:    push   %rbp
    0x0000555555555512e <+5>:    mov    %rsp,%rbp
    0x00005555555555131 <+8>:    mov    0x2edd(%rip),%edx      # 0x55555558014 <_ZZ4mainE1b>
    0x00005555555555137 <+14>:   mov    0x2edb(%rip),%eax      # 0x55555558018 <_ZZ4mainE1c>
    0x0000555555555513d <+20>:   add    %edx,%eax
    0x0000555555555513f <+22>:   mov    %eax,0x2ecb(%rip)    # 0x55555558010 <_ZZ4mainE1a>
    0x00005555555555145 <+28>:   mov    0x2ec5(%rip),%eax      # 0x55555558010 <_ZZ4mainE1a>
    0x0000555555555514b <+34>:   sub    $0x1,%eax
    0x0000555555555514e <+37>:   mov    %eax,0x2ec8(%rip)    # 0x5555555801c <_ZZ4mainE1d>
    0x00005555555555154 <+43>:   mov    $0x0,%eax
    0x00005555555555159 <+48>:   pop    %rbp
    0x0000555555555515a <+49>:   retq
End of assembler dump.
(gdb) info registers
rax          0x5555555555129      93824992235817
rbx          0x5555555555160      93824992235872
rcx          0x5555555555160      93824992235872
rdx          0x7fffffff068        140737488347240
rsi          0x7fffffff058        140737488347224
rdi          0x1                  1
rbp          0x0                  0x0
rsp          0x7fffffffdf68       0x7fffffffdf68
r8           0x0                  0
r9           0x7fffff7fe0d50       140737354009936
r10          0x0                  0
r11          0x0                  0
r12          0x5555555555040      93824992235584
r13          0x7fffffff050        140737488347216
r14          0x0                  0
r15          0x0                  0
rip          0x5555555555129      0x5555555555129 <main>
eflags        0x246              [ PF ZF IF ]
cs            0x33                51
ss            0x2b                43
ds            0x0                 0
es            0x0                 0
fs            0x0                 0
--Type <RET> for more, q to quit, c to continue without paging--S
[1]+  Stopped                  gdb a.out
main@main-VirtualBox: ~/Desktop/cs-343/takeHomeTest-1$
```

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.



The screenshot shows a terminal window titled "Documents" with an "Open" button. The current file is "emdad_17thapril2022_2-2_2.c" located at "~/Desktop/cs-343/takeHomeTest-1". The code is as follows:

```
1 #include <stdio.h>
2
3 int main() {
4     static int f = 0;
5     static int g = 50;
6     static int h = 40;
7     static int i = 30;
8     static int j = 20;
9     f = (g + h) - (i + j);
10 }
```

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 arithmetic operations. One is addition ($g+h$) and other one is summation.

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

```

(gdb) b main
Breakpoint 1 at 0x1129
(gdb) run
Starting program: /home/main/Desktop/cs-343/takeHomeTest-1/a.out

Breakpoint 1, 0x000055555555129 in main ()
(gdb) disassemble
Dump of assembler code for function main:
=> 0x000055555555129 <+0>:    endbr64
 0x00005555555512d <+4>:    push   %rbp
 0x00005555555512e <+5>:    mov    %rsp,%rbp
 0x000055555555131 <+8>:    mov    0x2ed9(%rip),%edx      # 0x555555558010 <_ZZ4mainE1g>
 0x000055555555137 <+14>:   mov    0x2ed7(%rip),%eax      # 0x555555558014 <_ZZ4mainE1h>
 0x00005555555513d <+20>:   lea    (%rdx,%rax,1),%ecx
 0x000055555555140 <+23>:   mov    0x2ed2(%rip),%edx      # 0x555555558018 <_ZZ4mainE1i>
 0x000055555555146 <+29>:   mov    0x2ed0(%rip),%eax      # 0x55555555801c <_ZZ4mainE1j>
 0x00005555555514c <+35>:   add    %edx,%eax
 0x00005555555514e <+37>:   sub    %eax,%ecx
 0x000055555555150 <+39>:   mov    %ecx,%eax
 0x000055555555152 <+41>:   mov    %eax,0x2ecc(%rip)     # 0x555555558024 <_ZZ4mainE1f>
 0x000055555555158 <+47>:   mov    $0x0,%eax
 0x00005555555515d <+52>:   pop    %rbp
 0x00005555555515e <+53>:   retq
End of assembler dump.
(gdb) info registers
rax            0x5555555555129      93824992235817
rbx            0x5555555555160      93824992235872
rcx            0x5555555555160      93824992235872
rdx            0x7fffffff068        140737488347240
rsi            0x7fffffff058        140737488347224
rdi            0x1                  1
rbp            0x0                  0x0
rsp            0x7fffffffdf68      0x7fffffffdf68
r8              0x0                  0
r9              0x7fffff7fe0d50      140737354009936
r10             0x0                  0
r11             0x0                  0
r12             0x555555555040      93824992235584
r13             0x7fffffff0e050      140737488347216
r14             0x0                  0
r15             0x0                  0
rip            0x5555555555129      0x5555555555129 <main>
eflags          0x246                [ PF ZF IF ]
cs              0x33                 51
ss              0x2b                 43
ds              0x0                  0
es              0x0                  0
fs              0x0                  0
gs              0x0                  0
(gdb)

```

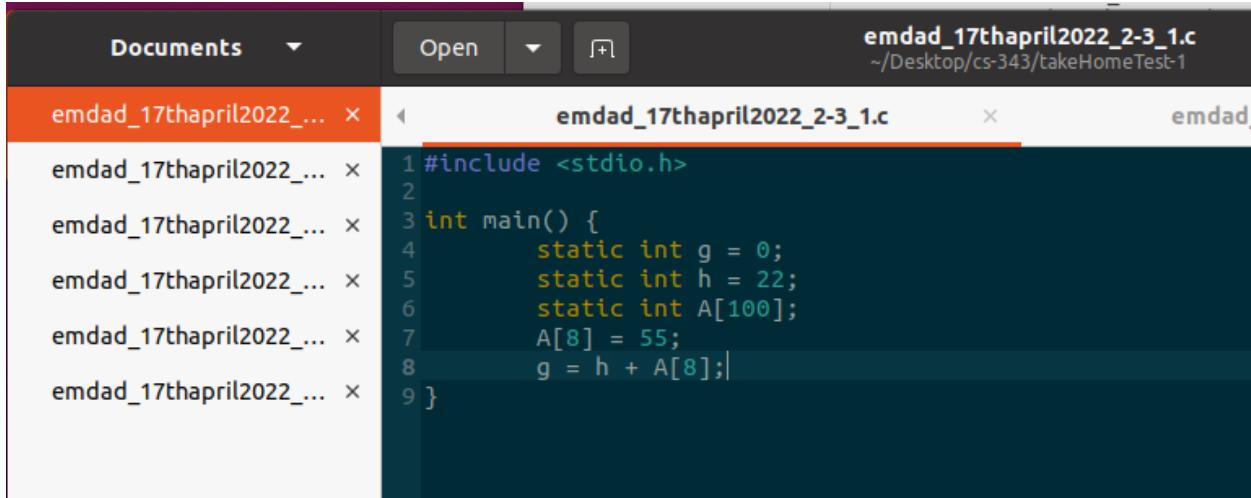
Figure 75: code output, disassemble, info registers.

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.



```
1 #include <stdio.h>
2
3 int main() {
4     static int g = 0;
5     static int h = 22;
6     static int A[100];
7     A[8] = 55;
8     g = h + A[8];
9 }
```

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.

```

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from a.out...
(No debugging symbols found in a.out)
(gdb) b main
Breakpoint 1 at 0x1129
(gdb) run
Starting program: /home/main/Desktop/cs-343/takeHomeTest-1/a.out

Breakpoint 1, 0x000055555555129 in main ()
(gdb) disassemble
Dump of assembler code for function main:
=> 0x000055555555129 <+0>:    endbr64
  0x00005555555512d <+4>:    push   %rbp
  0x00005555555512e <+5>:    mov    %rsp,%rbp
  0x000055555555131 <+8>:    movl   $0x37,0x2f45(%rip)      # 0x555555558080 <_ZZ4mainE1A+32>
  0x00005555555513b <+18>:   mov    0x2f3f(%rip),%edx      # 0x555555558080 <_ZZ4mainE1A+32>
  0x000055555555141 <+24>:   mov    0x2ec9(%rip),%eax      # 0x555555558010 <_ZZ4mainE1h>
  0x000055555555147 <+30>:   add    %edx,%eax
  0x000055555555149 <+32>:   mov    %eax,0x2ef1(%rip)      # 0x555555558040 <_ZZ4mainE1g>
  0x00005555555514f <+38>:   mov    $0x0,%eax
  0x000055555555154 <+43>:   pop    %rbp
  0x000055555555155 <+44>:   retq  
End of assembler dump.
(gdb) info registers
rax          0x5555555555129      93824992235817
rbx          0x5555555555160      93824992235872
rcx          0x5555555555160      93824992235872
rdx          0x7fffffff0e68      140737488347240
rsi          0x7fffffff0e58      140737488347224
rdi          0x1                  1
rbp          0x0                  0x0
rsp          0x7fffffffdf68      0x7fffffffdf68
r8           0x0                  0
r9           0x7fff7fe0d50      140737354009936
r10          0x0                  0
r11          0x0                  0
r12          0x555555555040      93824992235584
r13          0x7fffffff0e50      140737488347216
r14          0x0                  0
r15          0x0                  0
rip          0x5555555555129      0x5555555555129 <main>
eflags        0x246              [ PF ZF IF ]
cs            0x33                51
ss            0x2b                43
ds            0x0                  0
es            0x0                  0
fs            0x0                  0
gs            0x0                  0
(gdb) 
```

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.

The screenshot shows a code editor interface with multiple tabs. The current tab is titled "emdad_17thapril2022_2-3_2.c". The code inside the tab is as follows:

```
#include <stdio.h>
int main() {
    static int h = 25;
    static int A[100];
    A[8] = 200;
    A[12] = h + A[8];
}
```

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.

```

(gdb) b main
Breakpoint 1 at 0x1129
(gdb) run
Starting program: /home/main/Desktop/cs-343/takeHomeTest-1/a.out

Breakpoint 1, 0x000055555555129 in main ()
(gdb) disasemble
Dump of assembler code for function main:
=> 0x000055555555129 <+0>:    endbr64
  0x00005555555512d <+4>:    push   %rbp
  0x00005555555512e <+5>:    mov    %rsp,%rbp
  0x000055555555131 <+8>:    movl   $0xc8,0x2f25(%rip)      # 0x555555558060 <_ZZ4mainE1A+32>
  0x000055555555135 <+18>:   mov    0x2f1f(%rip),%edx      # 0x555555558060 <_ZZ4mainE1A+32>
  0x000055555555141 <+24>:   mov    0x2ec9(%rip),%eax      # 0x555555558010 <_ZZ4mainE1h>
  0x000055555555147 <+30>:   add    %edx,%eax
  0x000055555555149 <+32>:   mov    %eax,0x2f21(%rip)      # 0x555555558070 <_ZZ4mainE1A+48>
  0x00005555555514f <+38>:   mov    $0x0,%eax
  0x000055555555154 <+43>:   pop    %rbp
  0x000055555555155 <+44>:   retq
End of assembler dump.
(gdb) info registers
rax            0x5555555555129      93824992235817
rbx            0x5555555555160      93824992235872
rcx            0x5555555555160      93824992235872
rdx            0x7fffffff068       140737488347240
rsi            0x7fffffff058       140737488347224
rdi            0x1                  1
rbp            0x0                  0x0
rsp            0x7fffffffdf68      0x7fffffffdf68
r8              0x0                  0
r9              0x7ffff7fe0d50      140737354009936
r10             0x0                  0
r11             0x0                  0
r12             0x555555555040      93824992235584
r13             0x7fffffff050      140737488347216
r14             0x0                  0
r15             0x0                  0
rip            0x5555555555129      0x5555555555129 <main>
eflags          0x246                [ PF ZF IF ]
cs              0x33                 51
ss              0x2b                 43
ds              0x0                  0
es              0x0                  0
fs              0x0                  0
gs              0x0                  0
(gdb)

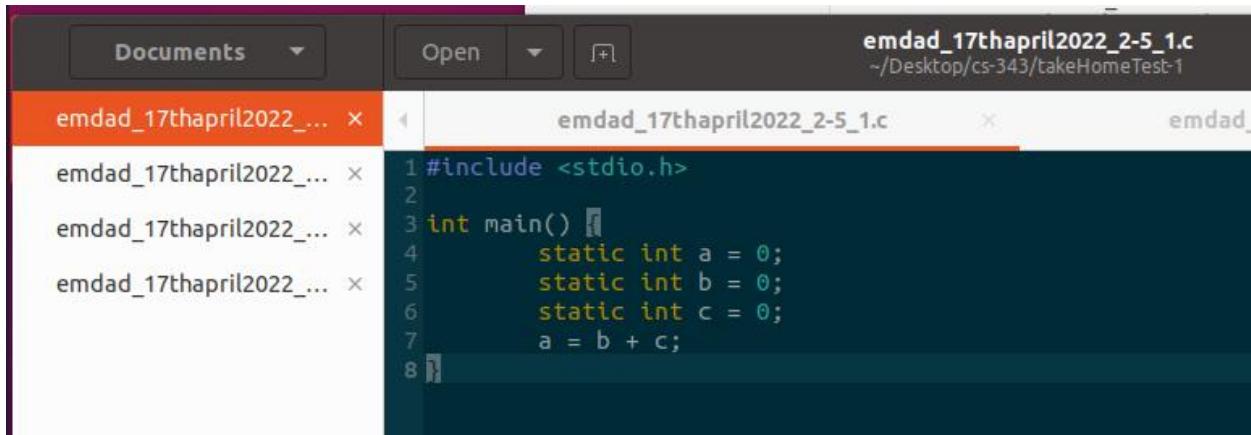
```

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.



The screenshot shows a terminal window with a dark background. At the top, there are tabs for "Documents" and "Open". The current tab is titled "emdad_17thapril2022_2-5_1.c" with the path "~/Desktop/cs-343/takeHomeTest-1". Below the tabs, there are several other tabs labeled "emdad_17thapril2022_..." followed by ellipses. The main area of the terminal displays the following C code:

```
1 #include <stdio.h>
2
3 int main() {
4     static int a = 0;
5     static int b = 0;
6     static int c = 0;
7     a = b + c;
8 }
```

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 arithmetic operations.

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

```

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from a.out...
(No debugging symbols found in a.out)
(gdb) b main
Breakpoint 1 at 0x1129
(gdb) run
Starting program: /home/main/Desktop/cs-343/takeHomeTest-1/a.out

Breakpoint 1, 0x00005555555555129 in main ()
(gdb) disassemble
Dump of assembler code for function main:
=> 0x00005555555555129 <+0>:    endbr64
 0x0000555555555512d <+4>:    push   %rbp
 0x0000555555555512e <+5>:    mov    %rsp,%rbp
 0x00005555555555131 <+8>:    mov    0x2ee1(%rip),%edx      # 0x55555558018 <_ZZ4mainE1b>
 0x00005555555555137 <+14>:   mov    0x2edf(%rip),%eax      # 0x5555555801c <_ZZ4mainE1c>
 0x0000555555555513d <+20>:   add    %edx,%eax
 0x0000555555555513f <+22>:   mov    %eax,0x2ecf(%rip)     # 0x55555558014 <_ZZ4mainE1a>
 0x00005555555555145 <+28>:   mov    $0x0,%eax
 0x0000555555555514a <+33>:   pop    %rbp
 0x0000555555555514b <+34>:   retq
End of assembler dump.
(gdb) info registers
rax          0x5555555555129      93824992235817
rbx          0x5555555555150      93824992235856
rcx          0x5555555555150      93824992235856
rdx          0x7fffffff068        140737488347240
rsi          0x7fffffff058        140737488347224
rdi          0x1                  1
rbp          0x0                  0x0
rsp          0x7fffffffdf68       0x7fffffffdf68
r8           0x0                  0
r9           0x7ffff7fe0d50       140737354009936
r10          0x0                  0
r11          0x0                  0
r12          0x5555555555040      93824992235584
r13          0x7fffffff050        140737488347216
r14          0x0                  0
r15          0x0                  0
rip          0x5555555555129      0x5555555555129 <main>
eflags        0x246              [ PF ZF IF ]
cs            0x33                51
ss            0x2b                43
ds            0x0                  0
es            0x0                  0
fs            0x0                  0
gs            0x0                  0
(gdb) 
```

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.

```

Documents Open emdad_17thApril2022_2-6_1.c ~Desktop/cs-343/takeHomeTest-1
emdad_17thApril2022_2-6_1.c × emdad_17thApril2022_2-6_2.c ×
emdad_17thApril2022_2-6_1.c ×
1 #include <stdio.h>
2
3 int main() {
4     static int s0 = 9;
5     static int t1 = 0x3c00;
6     static int t2 = 0xdc0;
7     static int t3 = 0;
8     t3 = s0 << 4;
9     static int t0 = 0;
10    t0 = t1 & t2;
11    t0 = t1 | t2;
12    t0 = ~t1;
13 }

```

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.

```

Breakpoint 1, 0x000055555555129 in main ()
(gdb) disasemb
Dump of assembler code for function main:
=> 0x000055555555129 <+0>:    endbr64
 0x00005555555512d <+4>:    push   %rbp
 0x00005555555512e <+5>:    mov    %rsp,%rbp
 0x000055555555131 <+8>:    mov    0x2ed9(%rip),%eax      # 0x555555558010 <_ZZ4mainE2s0>
 0x000055555555137 <+14>:   shl    $0x4,%eax
 0x00005555555513a <+17>:   mov    %eax,0x2ee0(%rip)    # 0x555555558020 <_ZZ4mainE2t3>
 0x000055555555140 <+23>:   mov    0x2ece(%rip),%edx      # 0x555555558014 <_ZZ4mainE2t1>
 0x000055555555146 <+29>:   mov    0x2ecc(%rip),%eax      # 0x555555558018 <_ZZ4mainE2t2>
 0x00005555555514c <+35>:   and    %edx,%eax
 0x00005555555514e <+37>:   mov    %eax,0x2ed0(%rip)    # 0x555555558024 <_ZZ4mainE2t0>
 0x000055555555154 <+43>:   mov    0x2eba(%rip),%edx      # 0x555555558014 <_ZZ4mainE2t1>
 0x00005555555515a <+49>:   mov    0x2eb8(%rip),%eax      # 0x555555558018 <_ZZ4mainE2t2>
 0x000055555555160 <+55>:   or     %edx,%eax
 0x000055555555162 <+57>:   mov    %eax,0xebc(%rip)    # 0x555555558024 <_ZZ4mainE2t0>
 0x000055555555168 <+63>:   mov    0x2e6(%rip),%eax      # 0x555555558014 <_ZZ4mainE2t1>
 0x00005555555516e <+69>:   not    %eax
 0x000055555555170 <+71>:   mov    %eax,0x2eae(%rip)    # 0x555555558024 <_ZZ4mainE2t0>
 0x000055555555176 <+77>:   mov    $0x0,%eax
 0x00005555555517b <+82>:   pop    %rbp
 0x00005555555517c <+83>:   retq 
End of assembler dump.
(gdb) info registers
rax          0x5555555555129      93824992235817
rbx          0x5555555555180      93824992235904
rcx          0x5555555555180      93824992235904
rdx          0x7fffffff fe068      140737488347240
r8l          0x7fffffff fe058      140737488347224
rdi          0x1                  1
rbp          0x0                  0x0
rsp          0x7fffffff fd68      0x7fffffff fd68
r8           0x0                  0
r9           0x7fffff7fe0d50      140737354009936
r10          0x0                  0
r11          0x0                  0
r12          0x5555555555040      93824992235584
r13          0x7fffffff fe050      140737488347216
r14          0x0                  0
r15          0x0                  0
rip          0x5555555555129      0x5555555555129 <main>
eflags        0x246              [ PF ZF IF ]
cs            0x33                51
ss            0x2b                43
ds            0x0                  0
es            0x0                  0
fs            0x0                  0
gs            0x0                  0
(gdb)

```

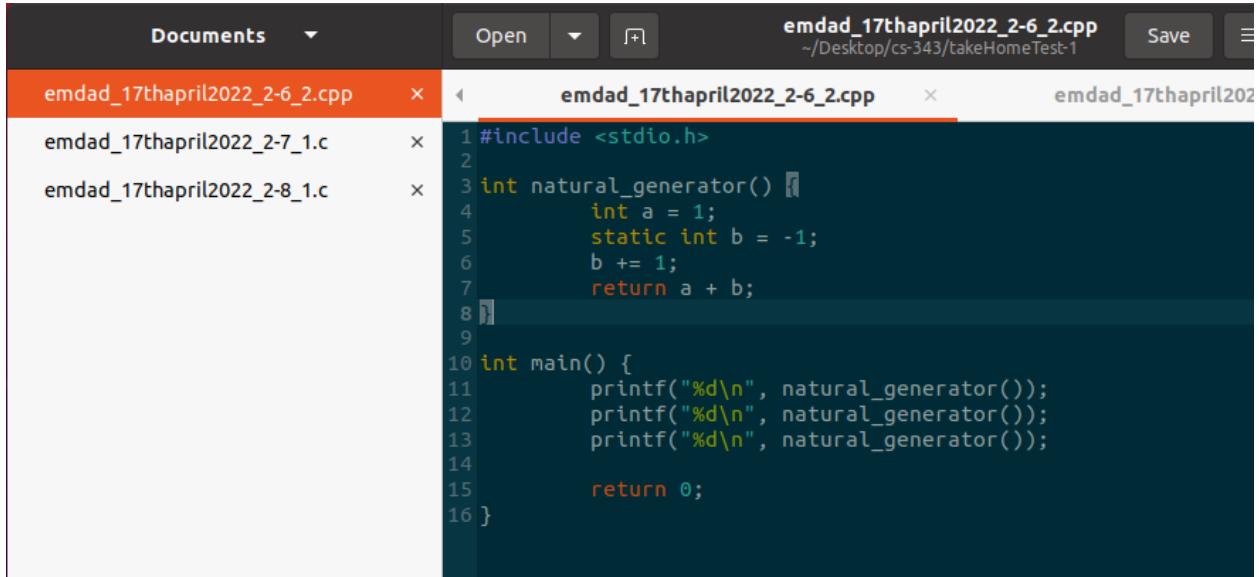
Figure 83: code output, disassemble, info registers.

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.



```

Documents Open emdad_17thapril2022_2-6_2.cpp ~/Desktop/cs-343/takeHomeTest-1 Save
emdad_17thapril2022_2-6_2.cpp x emdad_17thapril2022_2-6_2.cpp x emdad_17thapril2022_2-6_2.cpp x
emdad_17thapril2022_2-7_1.c x
emdad_17thapril2022_2-8_1.c x

1 #include <stdio.h>
2
3 int natural_generator() {
4     int a = 1;
5     static int b = -1;
6     b += 1;
7     return a + b;
8 }
9
10 int main() {
11     printf("%d\n", natural_generator());
12     printf("%d\n", natural_generator());
13     printf("%d\n", natural_generator());
14
15     return 0;
16 }

```

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.

```
main@main-VirtualBox: ~/Desktop/cs-343/takeHomeTest-1
(gdb) disassemble
Dump of assembler code for function main():
=> 0x000055555555174 <+0>:    endbr64
  0x000055555555178 <+4>:    push   %rbp
  0x000055555555179 <+5>:    mov    %rsp,%rbp
  0x00005555555517c <+8>:    callq  0x5555555555149 <natural_generator()>
  0x000055555555181 <+13>:   mov    %eax,%esi
  0x000055555555183 <+15>:   lea    0xe7a(%rip),%rdi          # 0x555555556004
  0x00005555555518a <+22>:   mov    $0x0,%eax
  0x00005555555518f <+27>:   callq  0x555555555050 <printf@plt>
  0x000055555555194 <+32>:   callq  0x555555555149 <natural_generator()>
  0x000055555555199 <+37>:   mov    %eax,%esi
  0x00005555555519b <+39>:   lea    0xe62(%rip),%rdi          # 0x555555556004
  0x0000555555551a2 <+46>:   mov    $0x0,%eax
  0x0000555555551a7 <+51>:   callq  0x555555555050 <printf@plt>
  0x0000555555551ac <+56>:   callq  0x555555555149 <natural_generator()>
  0x0000555555551b1 <+61>:   mov    %eax,%esi
  0x0000555555551b3 <+63>:   lea    0xe4a(%rip),%rdi          # 0x555555556004
  0x0000555555551ba <+70>:   mov    $0x0,%eax
  0x0000555555551bf <+75>:   callq  0x555555555050 <printf@plt>
  0x0000555555551c4 <+80>:   mov    $0x0,%eax
  0x0000555555551c9 <+85>:   pop    %rbp
  0x0000555555551ca <+86>:   retq
End of assembler dump.
(gdb) info registers
rax          0x5555555555174      93824992235892
rbx          0x5555555551d0       93824992235984
rcx          0x5555555551d0       93824992235984
rdx          0x7fffffff0e068     140737488347240
rsi          0x7fffffff0e058     140737488347224
rdi          0x1                  1
rbp          0x0                  0x0
rsp          0x7fffffffdf68     0x7fffffffdf68
r8           0x0                  0
r9           0x7fffff7fe0d50     140737354009936
r10          0x0                  0
r11          0x0                  0
r12          0x555555555060     93824992235616
r13          0x7fffffff0e050     140737488347216
r14          0x0                  0
r15          0x0                  0
rip          0x5555555555174      0x5555555555174 <main()>
eflags        0x246              [ PF ZF IF ]
cs            0x33                51
ss            0x2b                43
ds            0x0                  0
es            0x0                  0
fs            0x0                  0
gs            0x0                  0
(gdb)
```

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.

```

1 #include <stdio.h>
2
3 int main() {
4     static int a= 5;
5     static int b= 10;
6     static int save[100];
7
8     save[5] = 10;
9     save[6] = 10;
10 }

```

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.

```

Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from a.out...
(gdb) b main
Breakpoint 1 at 0x1129: file emdad_17thapril2022_2-7_1.c, line 3.
(gdb) run
Starting program: /home/main/Desktop/cs-343/takeHomeTest-1/a.out

Breakpoint 1, main () at emdad_17thapril2022_2-7_1.c:3
3      int main() {
(gdb) disassemble
Dump of assembler code for function main:
=> 0x000055555555129 <+0>:    endbr64
  0x00005555555512d <+4>:    push   %rbp
  0x00005555555512e <+5>:    mov    %rsp,%rbp
  0x000055555555131 <+8>:    movl   $0xa,0x2f19(%rip)          # 0x555555558054 <save.2317+20>
  0x00005555555513b <+18>:   movl   $0xa,0x2f13(%rip)          # 0x555555558058 <save.2317+24>
  0x000055555555145 <+28>:   mov    $0x0,%eax
  0x00005555555514a <+33>:   pop    %rbp
  0x00005555555514b <+34>:   retq  
End of assembler dump.
(gdb) info registers
rax            0x5555555555129      93824992235817
rbx            0x5555555555150      93824992235856
rcx            0x5555555555150      93824992235856
rdx            0x7fffffff068        140737488347240
rsi            0x7fffffff058        140737488347224
rdi            0x1                1
rbp            0x0                0x0
rsp            0x7fffffffdf68      0x7fffffffdf68
r8             0x0                0
r9             0x7ffff7fe0d50      140737354009936
r10            0x0                0
r11            0x0                0
r12            0x555555555040      93824992235584
r13            0x7fffffff050        140737488347216
r14            0x0                0
r15            0x0                0
rip            0x555555555129      0x555555555129 <main>
eflags          0x246            [ PF ZF IF ]
cs             0x33             51
ss             0x2b             43
ds             0x0              0
es             0x0              0
fs             0x0              0
gs             0x0              0
(gdb)

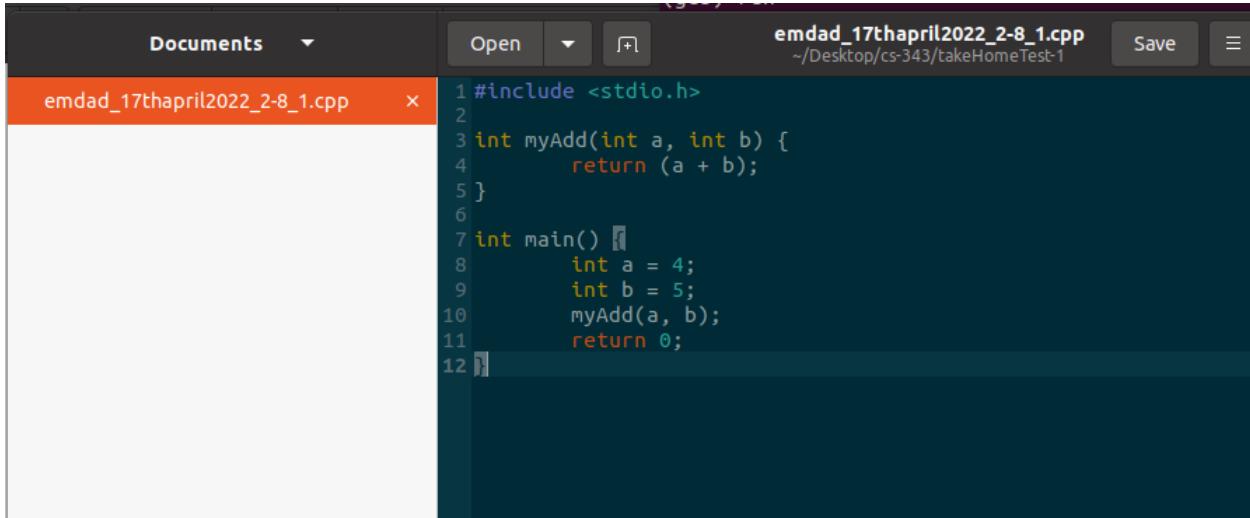
```

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.



The screenshot shows a terminal window with a dark theme. The title bar says "Documents" and the current file is "emdad_17thapril2022_2-8_1.cpp". The code editor area contains the following C++ code:

```
1 #include <stdio.h>
2
3 int myAdd(int a, int b) {
4     return (a + b);
5 }
6
7 int main() {
8     int a = 4;
9     int b = 5;
10    myAdd(a, b);
11    return 0;
12 }
```

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.

```
main@main-VirtualBox: ~/Desktop/cs-343/takeHomeTest-1
(gdb) b main
Breakpoint 1 at 0x1141: file emdad_17thapril2022_2-8_1.cpp, line 7.
(gdb) run
Starting program: /home/main/Desktop/cs-343/takeHomeTest-1/a.out

Breakpoint 1, main () at emdad_17thapril2022_2-8_1.cpp:7
7      int main() {
(gdb) disassemble
Dump of assembler code for function main():
=> 0x00000555555555141 <+0>:    endbr64
  0x00000555555555145 <+4>:    push   %rbp
  0x00000555555555146 <+5>:    mov    %rsp,%rbp
  0x00000555555555149 <+8>:    sub    $0x10,%rsp
  0x0000055555555514d <+12>:   movl   $0x4,-0x8(%rbp)
  0x00000555555555154 <+19>:   movl   $0x5,-0x4(%rbp)
  0x0000055555555515b <+26>:   mov    -0x4(%rbp),%edx
  0x0000055555555515e <+29>:   mov    -0x8(%rbp),%eax
  0x00000555555555161 <+32>:   mov    %edx,%esi
  0x00000555555555163 <+34>:   mov    %eax,%edi
  0x00000555555555165 <+36>:   callq  0x555555555129 <myAdd(int, int)>
  0x0000055555555516a <+41>:   mov    $0x0,%eax
  0x0000055555555516f <+46>:   leaveq 
  0x00000555555555170 <+47>:   retq 

End of assembler dump.
(gdb) info registers
rax          0x555555555141      93824992235841
rbx          0x555555555180      93824992235904
rcx          0x555555555180      93824992235904
rdx          0x7fffffff068       140737488347240
rsi          0x7fffffff058       140737488347224
rdi          0x1                  1
rbp          0x0                  0x0
rsp          0x7fffffffdf68      0x7fffffffdf68
r8           0x0                  0
r9           0x7fffff7fe0d50     140737354009936
r10          0x0                  0
r11          0x0                  0
r12          0x555555555040      93824992235584
r13          0x7fffffff050       140737488347216
r14          0x0                  0
r15          0x0                  0
rip          0x555555555141      0x555555555141 <main()>
eflags        0x246              [ PF ZF IF ]
cs            0x33                51
ss            0x2b                43
ds            0x0                 0
es            0x0                 0
fs            0x0                 0
gs            0x0                 0
(gdb)
```

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.