Take Home Test #1
Instruction Set Architecture Comparison

Name: Ismail Akram

Date: 11/10/2021

Course: CSC 34200-G & CSC34300-DE

Start Time and Date: 11/6/2021 4:20 PM

End Time and Date: 11/9/2021 12:00 AM (<12 hours total)

1. Please hand write and sign statements affirming that you will not cheat here and in your submission:

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

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

Ismail Akram

Contents

Objective:

The goal of this take-home test is to comprehend and compare various set architectures; from MIPS instruction set architecture, Intel x86 ISA (using Windows MS 32-bit compiler and debugger), and an Intel x86 64-bit ISA processor (running on Linux, 64-bit GCC and GDB).

I'm using MARS simulator to comprehend the MIPS instruction set architecture, Visual Studios' debugger to understand Intel x86 32-bit ISA, and Linux's GCC and GDB debugger to understand Intel x86 64-bit.

I'm running the various provided codes across these three platforms, as in, each code will be tested on each platform. For example, 2-2_1 will be running in MARS as assembly code, Visual Studios as C code, and Linux as C code. After compiling the code, we would use the debugger to analyze what occurs as each line of code runs. As a result, we would learn and understand how to read the debugger across these three platforms.

MIPS on MARS:

MIPS is a reduced instruction set computer instruction set architecture. It's a form of assembly language that produces an .asm file; in this showcase, I'm running MIPS on MARS. **MIPS operates using big endian**, so it stores its least significant byte (LSB) in the highest memory address.

2-2_1.asm:

The code written for 2-2_1 in MIPS is shown in Fig 1. It uses five static variables: a, b, c, d, and e. Using these five variables; arithmetic is performed and stored in memory.

In our case, we're adding the value of b and c and the sum is stored in a. We're also subtracting the values of a and e and storing that result in d. The results are stored in the data segment.

Akram_2-2_1.asm

```
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
```

Fig 1. Akram_2-2_1.asm code displayed in MARS

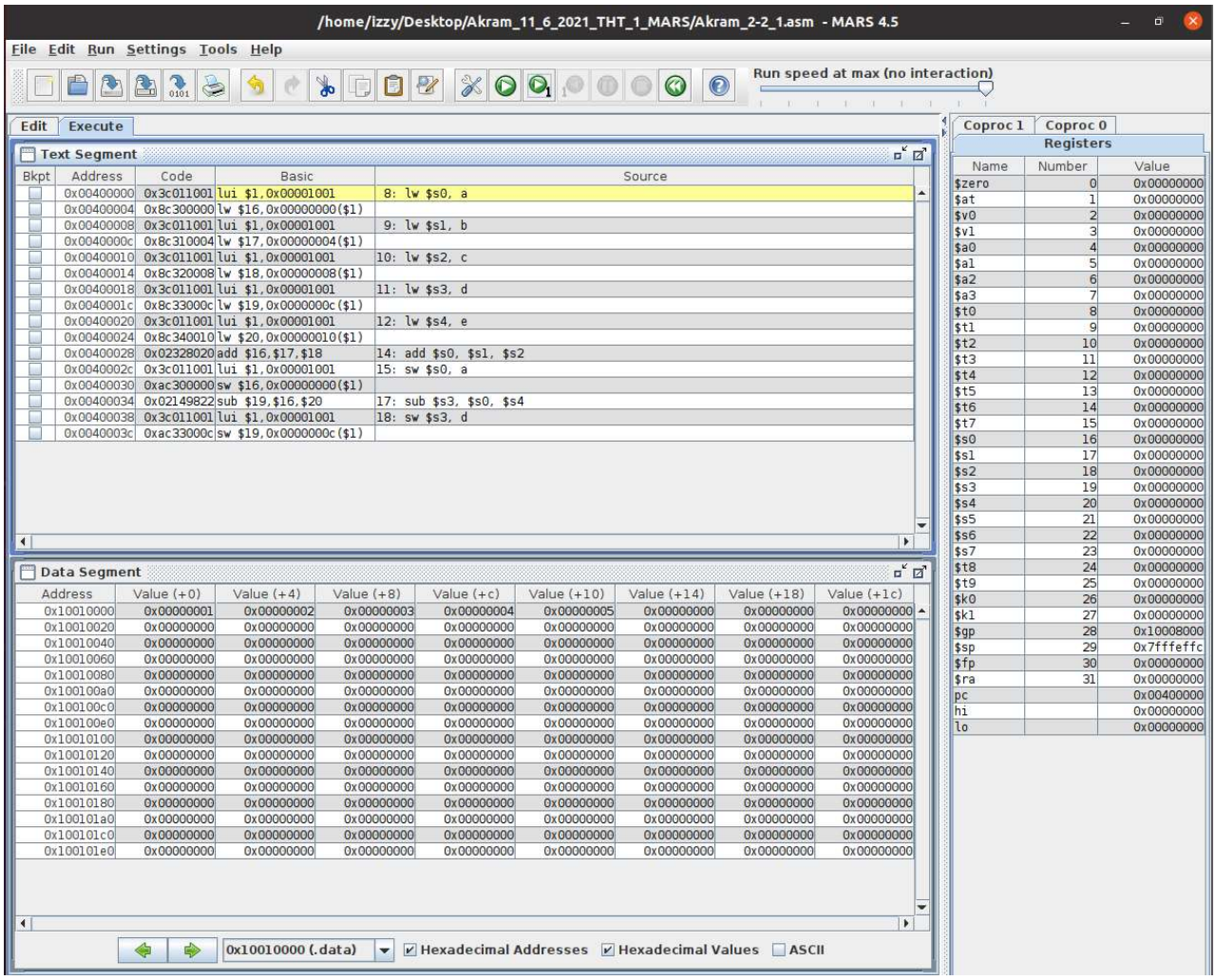

Fig 2. Akram_2-2_1.asm executed in MARS (lines 1-7)

Text Segment: this window shows that the code starts at line 8 since everything prior (line 1-7) doesn't require registers.

Registers: shows the value of the stack pointer, which is 0x7fffeffc. Nothing is stored in registers since no operations have been performed yet.

Data Segment: show the value of the static variables being stored into the addresses.

Variable a is stored in address: 0x10010000 containing the value: 0x00000001.
Variable b is stored in address: 0x10010004 containing the value: 0x00000002.
Variable c is stored in address: 0x10010008 containing the value: 0x00000003.
Variable d is stored in address: 0x1001000c containing the value: 0x00000004.
Variable e is stored in address: 0x10010010 containing the value: 0x00000005.

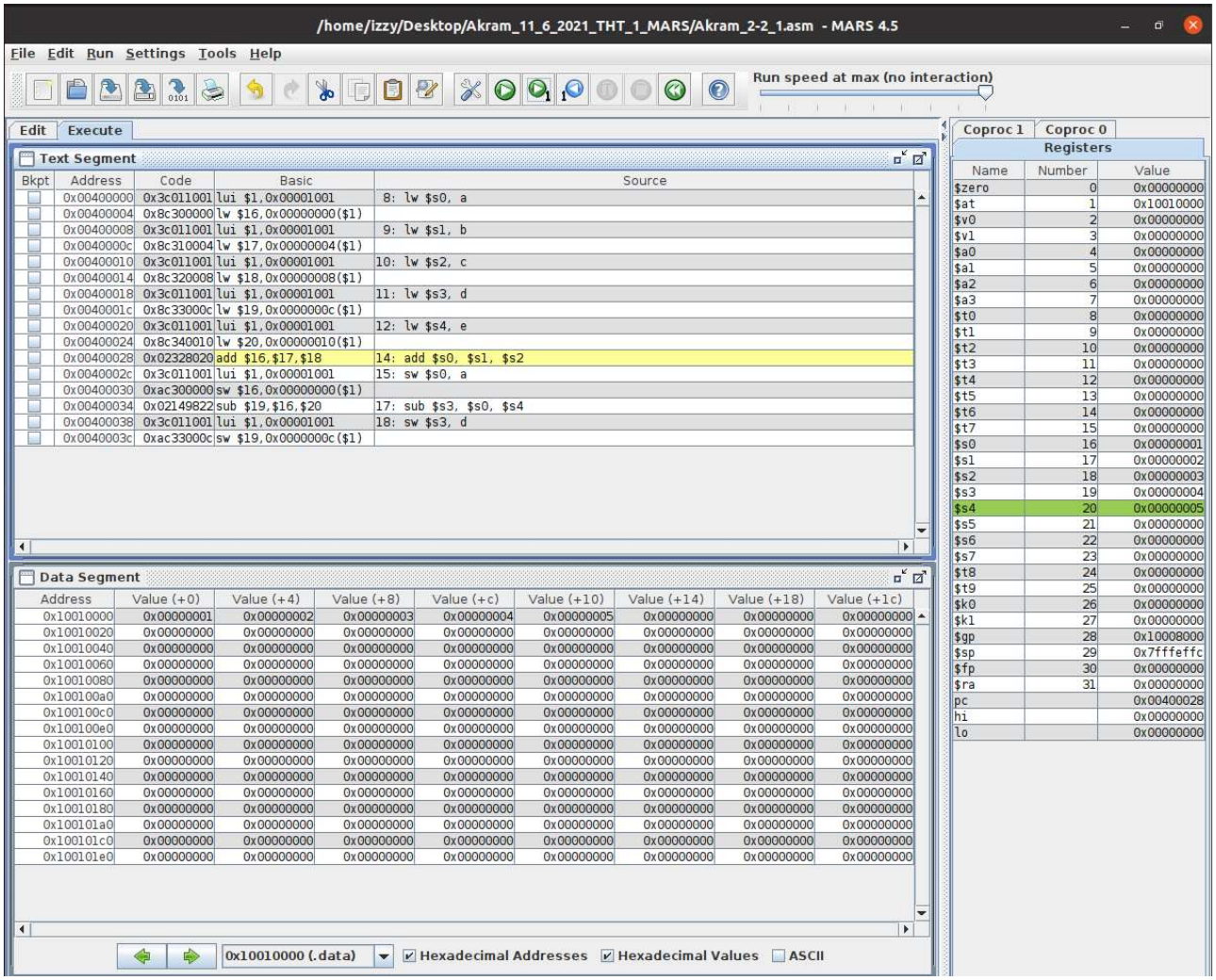

Fig 3. Akram_2-2_1.asm executed in MARS (lines 8-13)

Text Segment: this window shows that the code runs lines 8 to 13 and that the static variables are loaded into the registers during the run.

Registers: shows that the:

value of a is stored into $s0,
value of b is stored into $s1,
value of c is stored into $s2,
value of d is stored into $s3,
value of e is stored into $s4.

Data Segment: remains the same because lines 8 to 13 don't store anything into them or change any values.

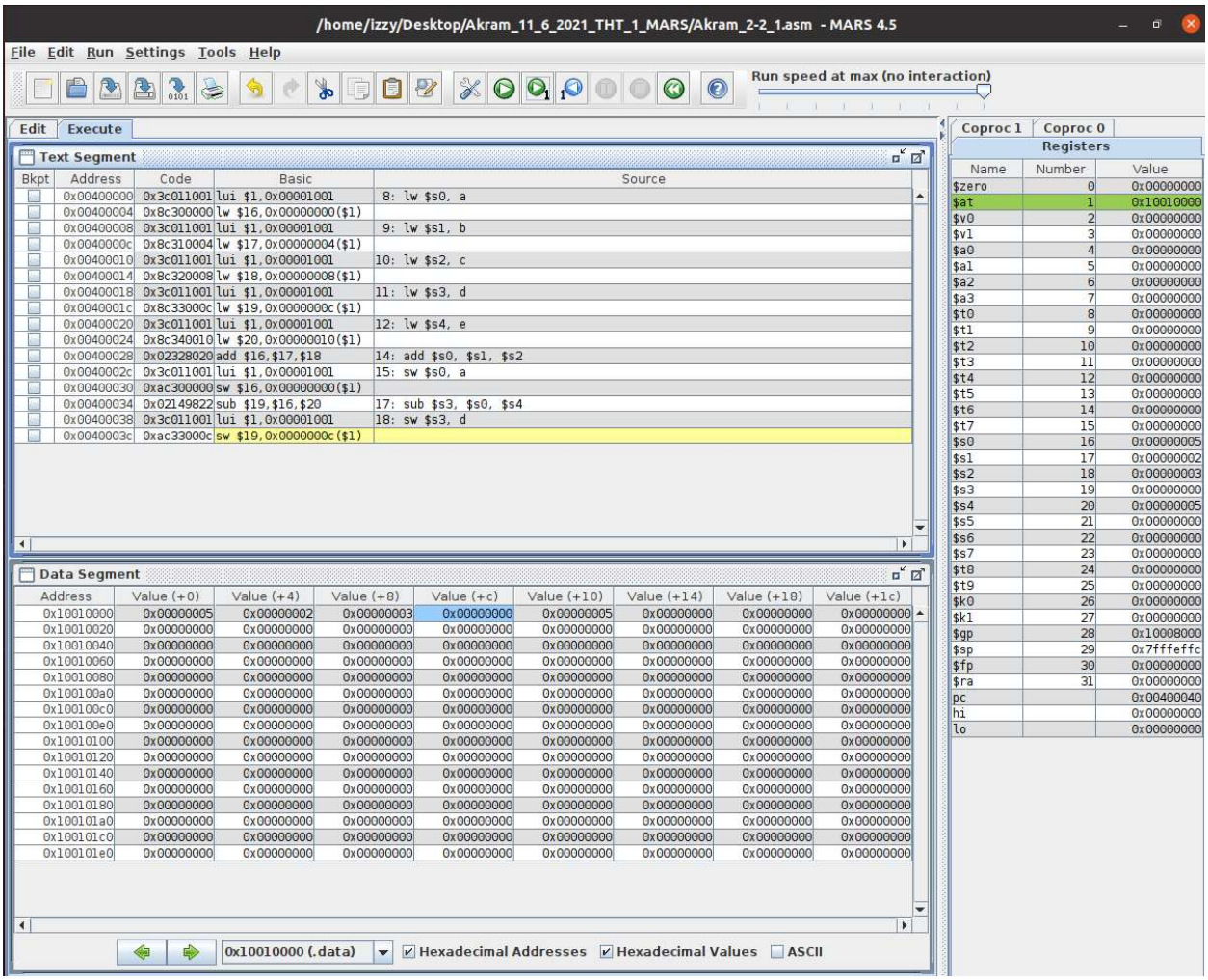

Fig 4. Akram_2-2_1.asm executed in MARS (lines 14-18)

Text Segment: this window shows that the code runs lines 14 to 18 and the effects on the registers and data segments during said run. We added and subtracted the values of the registers and stored the result into another register and an address of the data segment.

Registers: this window shows that the:

value of $s1 and $s2 are added, and the result is stored into $s0,
value of $s0 and $s4 are subtracted, and the result is stored into $s3.

Data Segment: shows that:

After the addition of $s1 and $s2, the result $s0 is stored into the address and allotted to the variable a. Address 0x1001000 contains the value 0x00000005.

After the subtraction of $s0 and $s4, the result $s3 is stored into the address and allotted to the variable d. Address 0x1001000c contains the value 0x00000000.

2-2_2.asm:

The code written for 2-2_2 in MIPS is shown in Fig 5. It uses five static variables: f, g, h, i, and j. Using these five variables; arithmetic is performed and stored in memory.

In our case, we're adding the values of g and h and the sum is stored in a register. We're also adding the values of i and j and storing that result in another register. We're then subtracting those results (the ones stored in registers t0 and t1 respectively) and storing this new result in f. Then that result will be stored in a data segment.

Akram_2-2_2.asm

```
1  .data
2  f: .word 1
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
```

Fig 5. Akram_2-2_2.asm code displayed in MARS

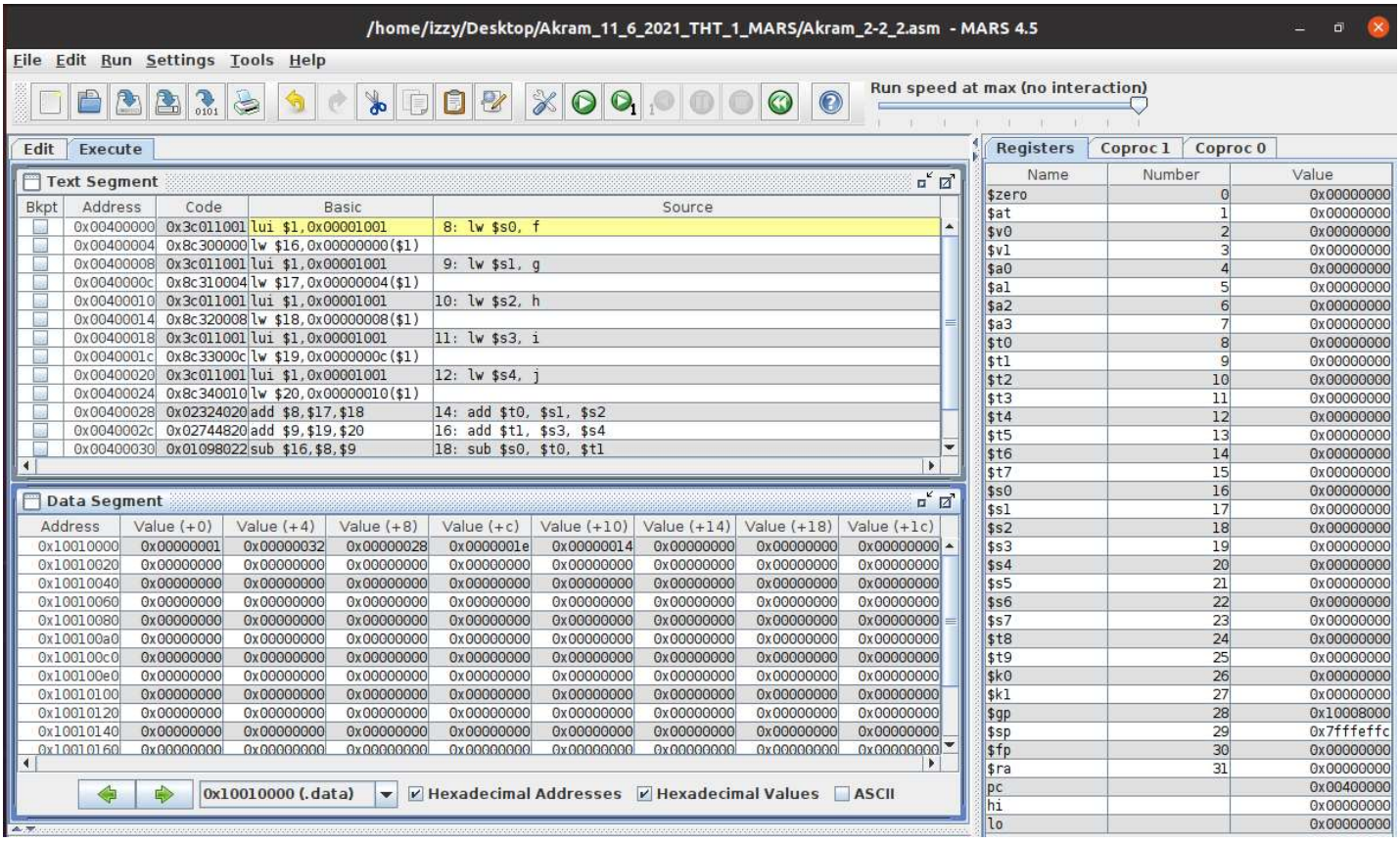

Fig 6. Akram_2-2_2.asm executed in MARS (lines 1-7)

Text Segment: this window shows that the code starts at line 8 since everything prior (line 1-7) doesn't require registers.

Registers: shows the value of the stack pointer, which is 0x7fffeffc. Nothing is stored in registers since no operations have been performed yet.

Data Segment: show the value of the static variables being stored into the addresses.

Variable f is stored in address: 0x10010000 containing the value: 0x00000001.
Variable g is stored in address: 0x10010004 containing the value: 0x00000032.
Variable h is stored in address: 0x10010008 containing the value: 0x00000028.
Variable i is stored in address: 0x1001000c containing the value: 0x0000001e.
Variable j is stored in address: 0x10010010 containing the value: 0x00000014.

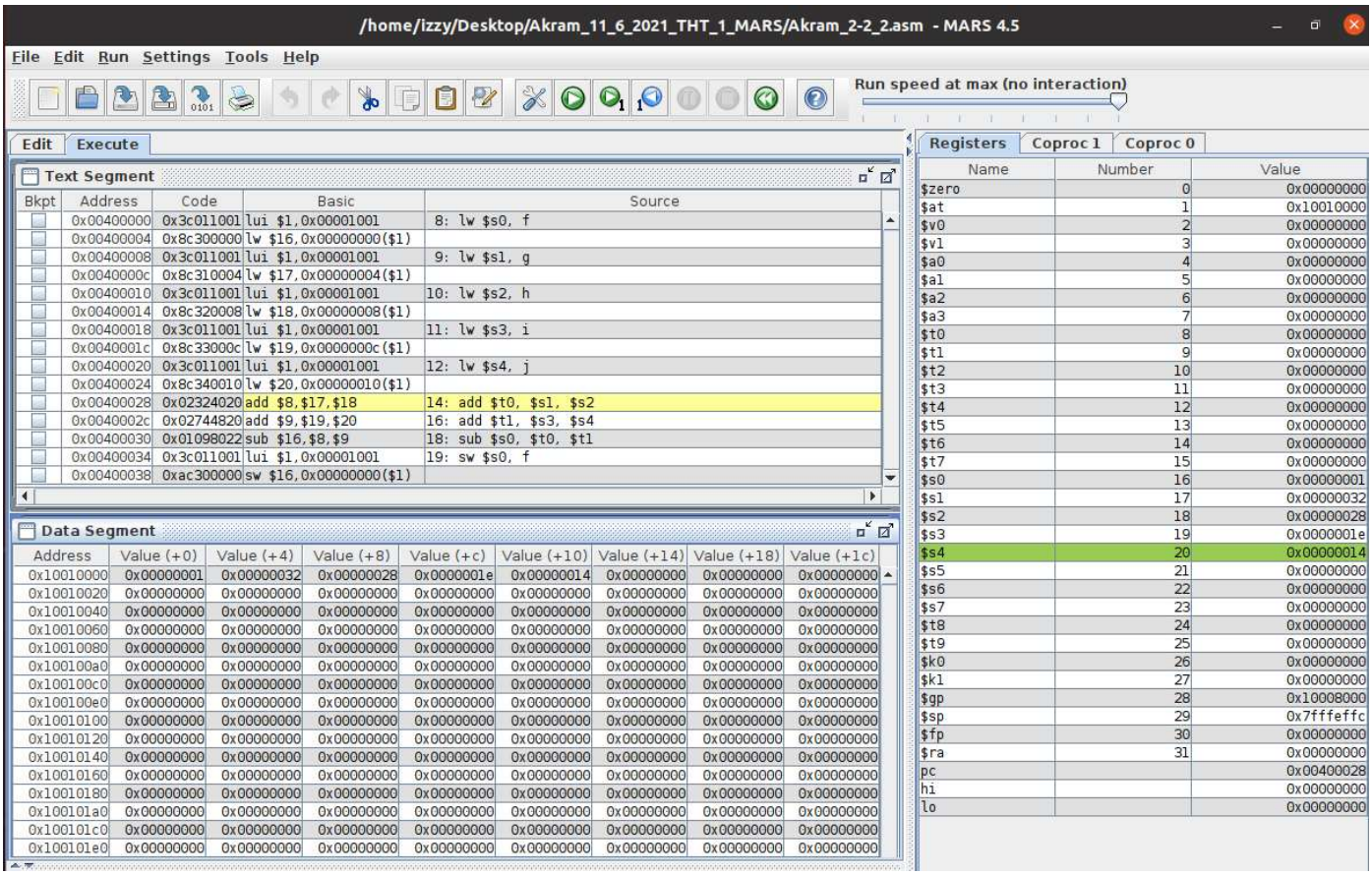

Fig 7. Akram_2-2_2.asm executed in MARS (lines 8-13)

Text Segment: this window shows that the code runs lines 8 to 13 and that the static variables are loaded into the registers during the run.

Registers: shows that the:

value of f is stored into $s0,
value of g is stored into $s1,
value of h is stored into $s2,
value of i is stored into $s3,
value of j is stored into $s4.

Data Segment: remains the same because lines 8 to 13 don't store anything into them or change any values.

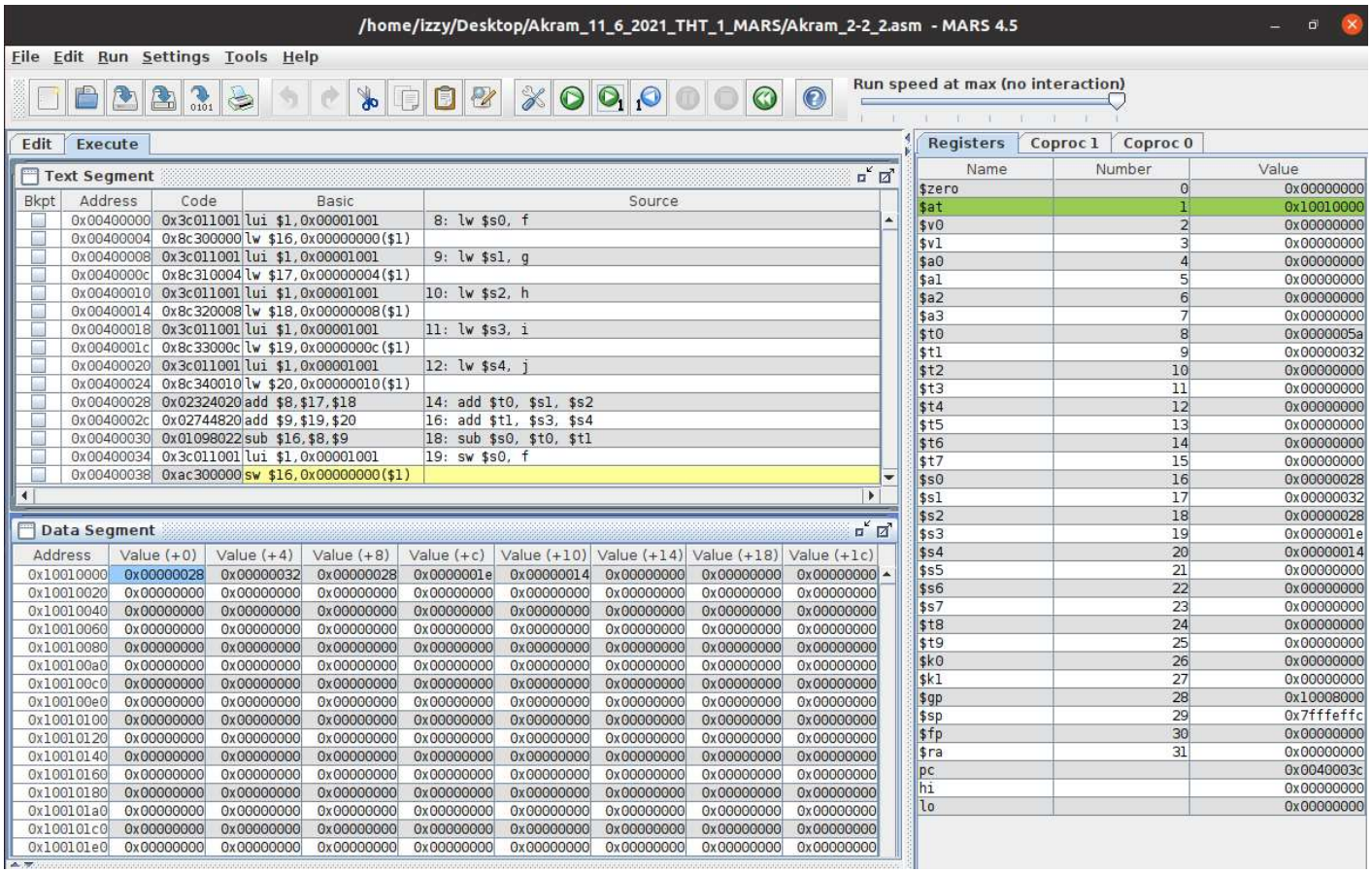

Fig 8. Akram_2-2_2.asm executed in MARS (lines 14-19)

Text Segment: this window shows that the code runs lines 14 to 19 and the effects on the registers and data segments during said run. We added and subtracted the values of the registers and stored the result into another register and an address of the data segment.

Registers: this window shows that the:

value of $s1 and $s2 are added, and the result is stored into $t0,
value of $s3 and $s4 are added, and the result is stored into $t1,
value of $t0 and $t1 are subtracted, and the result is stored into $s0.

Data Segment: shows that:

After the subtraction of $t0 and $t1, the result $s0 is stored into the address and allotted to the variable f. Address 0x10010000 contains the value 0x00000028.

2-3_1.asm:

The code written for 2-3_1 in MIPS is shown in Fig 9. It uses four static variables: g, h, A, and size. Using these four variables; an array is created, and addition is executed through its usage.

In our case, a value is stored in part of the array and is added to h. The result from that addition is stored into g and h and the data segment.

Akram_2-3_1.asm

```
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
```

Fig 9. Akram_2-3_1.asm code displayed in MARS

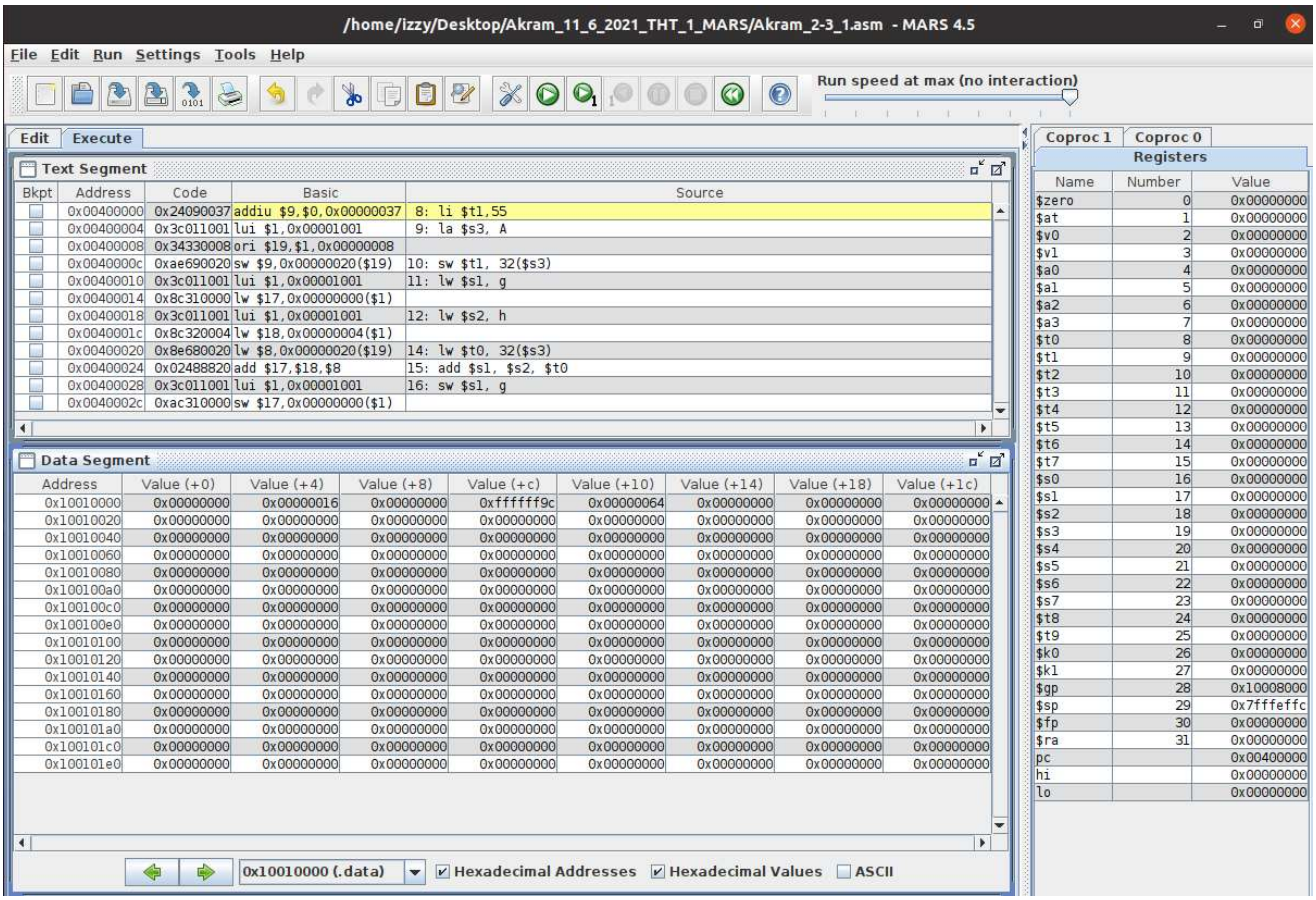

Fig 10. Akram_2-3_1.asm executed in MARS (lines 1-7)

Text Segment: this window shows that the code starts at line 8 since everything prior (line 1-7) doesn't require registers.

Registers: shows the value of the stack pointer, which is 0x7fffeffc. Nothing is stored in registers since no operations have been performed yet.

Data Segment: show the value of the static variables being stored into the addresses.

Variable g is stored in address: 0x10010000 containing the value: 0x00000000.
Variable h is stored in address: 0x10010004 containing the value: 0x00000016.
Variable A is stored in addresses: 0x10010008 and 0x1001000c
containing the values: 0x00000000 and 0xfffffff9c.
Variable size is stored in address: 0x10010010 containing the value: 0x00000064.

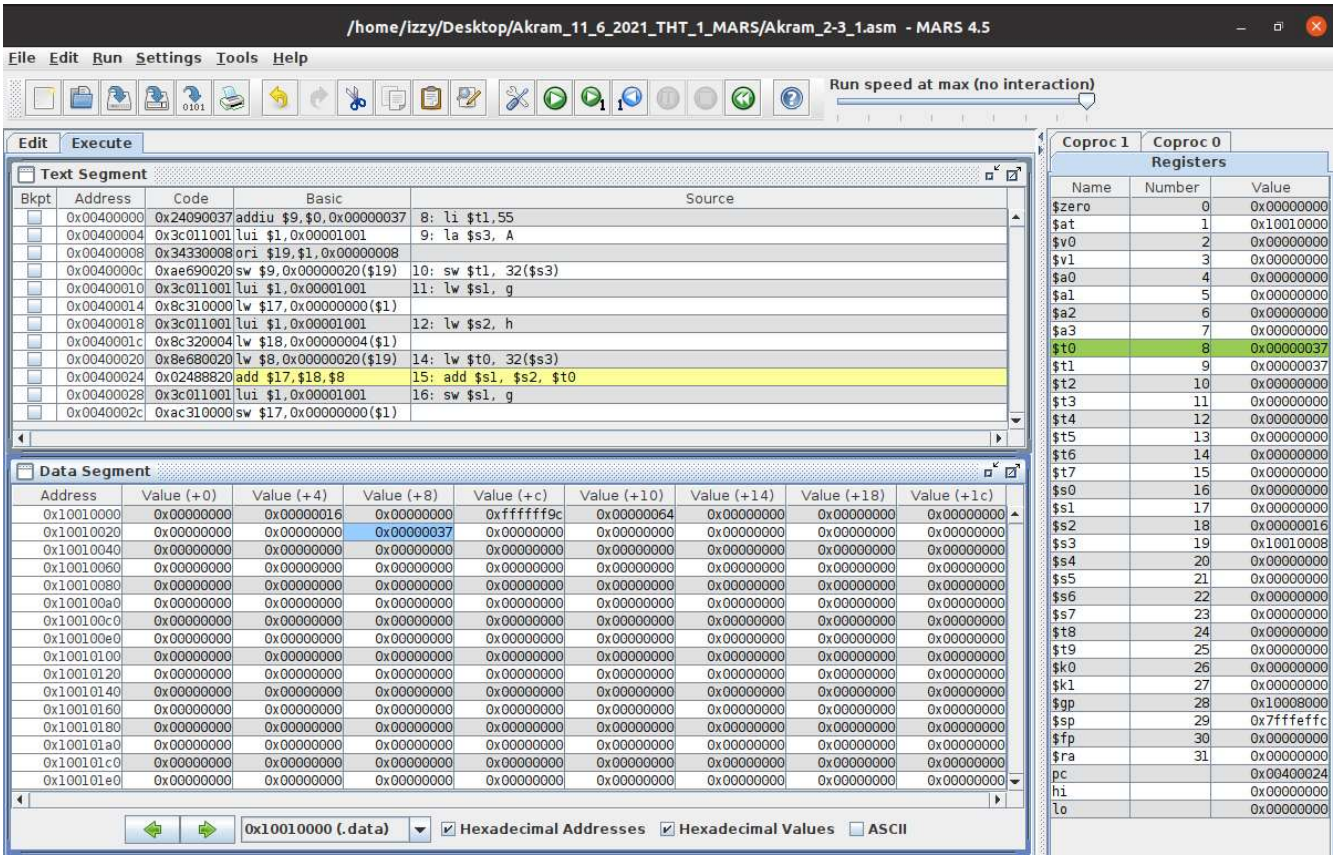

Fig 11. Akram_2-3_1.asm executed in MARS (lines 8-14)

Text Segment: this window shows that the code runs lines 8 to 14 and that the static variables and some values are loaded into the registers during the run. We're also storing the value of a register into the data segment (highlight blue).

Registers: shows that the:

value of 55 is stored into $t1 (hex value: 0x00000037),
address of A is stored into $s3,
value of g is stored into $s1,
value of h is stored into $s2,
value of 8th index in A is stored into $t0.

Data Segment: shows that the value of $t1; 55, is stored in the 8th index of array A.

Address 0x10010028 contains the value 0x00000037.

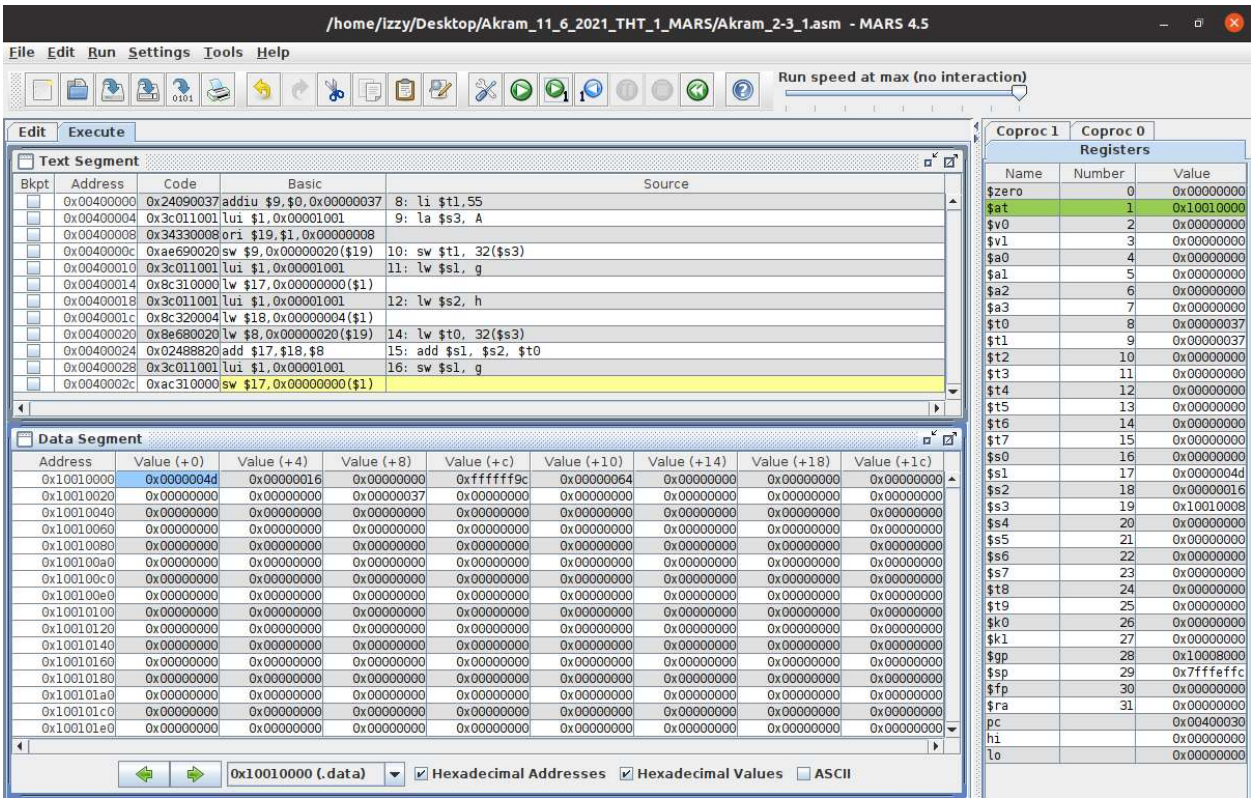

Fig 12. Akram_2-3_1.asm executed in MARS (lines 15-16)

Text Segment: this window shows that the code runs lines 15 to 16 and the effects on the registers and data segments during said run. We added the values of the registers and stored the result into another register and into the data segment.

Registers: this window shows that the:

value of $s2 and $t0 are added, and the result is stored into $s1.

Data Segment: shows that:

After the addition of $s2 and $t0, the result $s1 is stored into the address and allotted to the variable g. Address 0x10010000 contains the value 0x0000004d.

2-3_2.asm:

The code written for 2-3_2 in MIPS is shown in Fig 13. It uses three static variables: h, A, and size. Using these three variables; an array is created, and addition is executed through its usage.

In our case, a value is stored in the array and is added to h. The result from that addition is stored into another part of the array and the data segment.

Akram_2-3_2.asm

```
1  .data
2  h: .word 25
3  A: .word 0-100
4  size: .word 100
5  .text
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)
```

Fig 13. Akram_2-3_2.asm code displayed in MARS

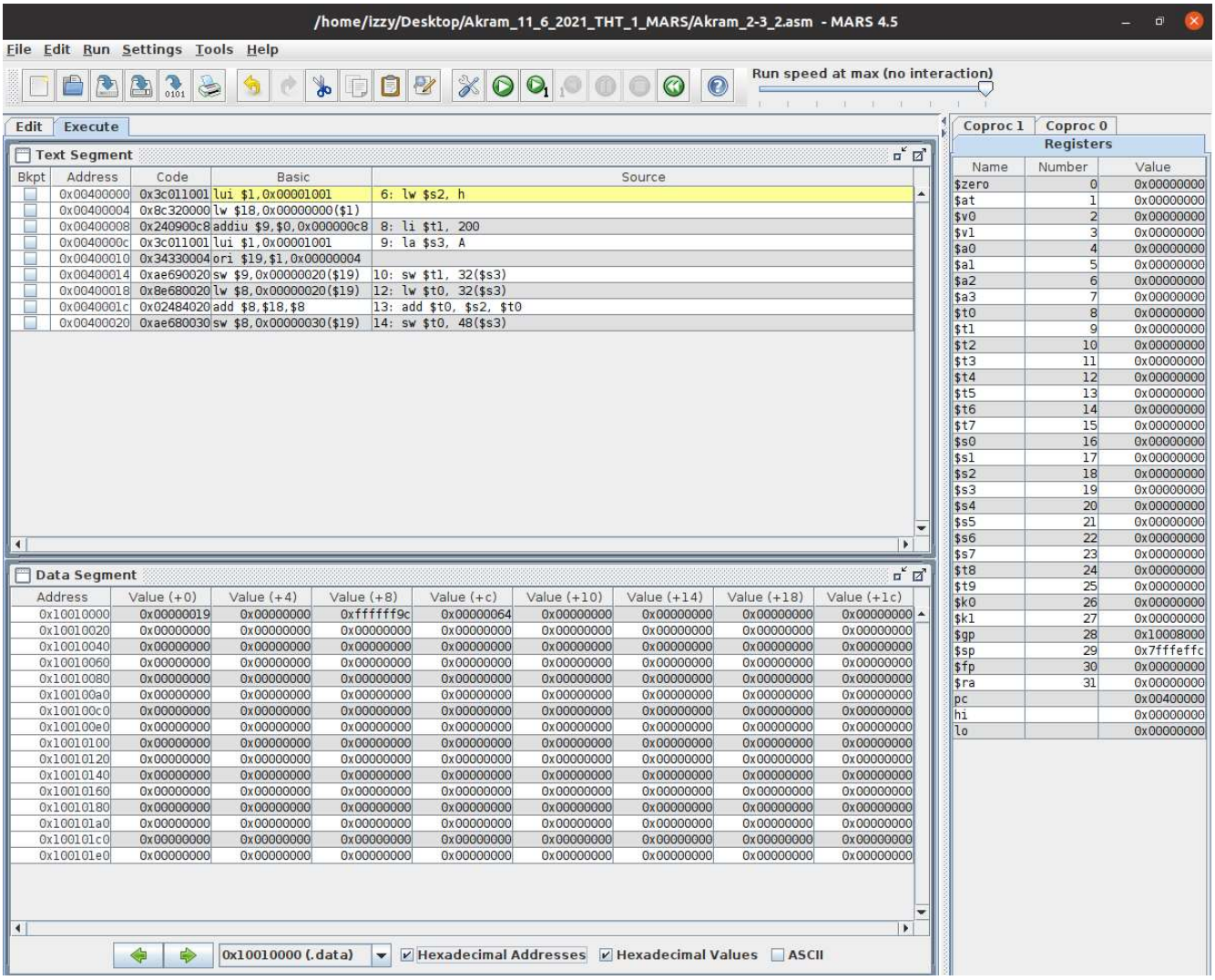

Fig 14. Akram_2-3_2.asm executed in MARS (lines 1-5)

Text Segment: this window shows that the code starts at line 6 since everything prior (line 1-5) doesn't require registers.

Registers: shows the value of the stack pointer, which is 0x7fffeffc. Nothing is stored in registers since no operations have been performed yet.

Data Segment: show the value of the static variables being stored into the addresses.

Variable h is stored in address: 0x10010000 containing the value: 0x00000019.
Variable A is stored in addresses: 0x10010004 and 0x10000008
containing the values: 0x00000000 and 0xffffff9c.
Variable size is stored in address: 0x1001000c containing the value: 0x00000064.

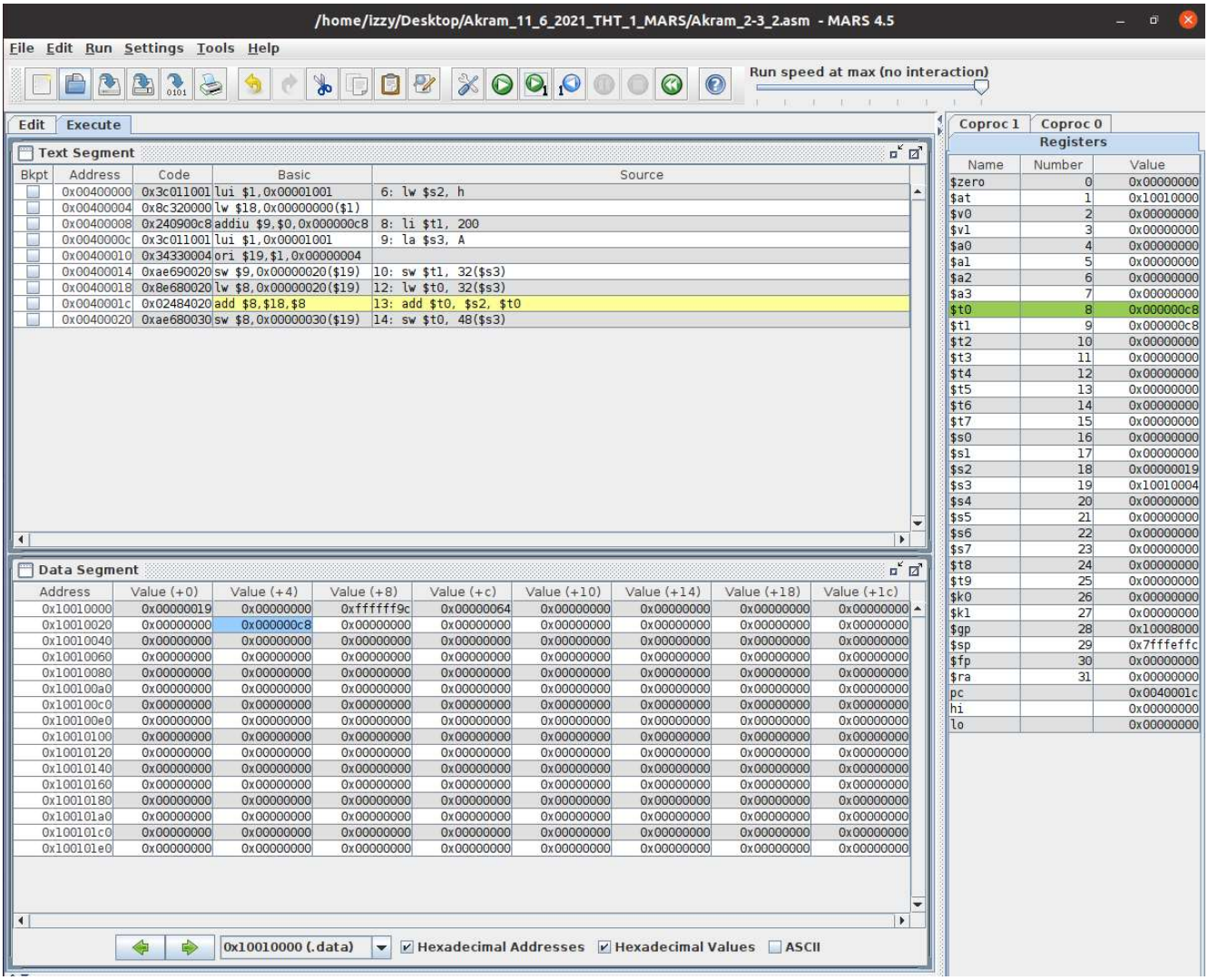

Fig 15. Akram_2-3_2.asm executed in MARS (lines 6-12)

Text Segment: this window shows that the code runs lines 6 to 12 and that the static variables and some values are loaded into the registers during the run. We're also storing the value of a register into the data segment (highlight blue).

Registers: shows that the:

value of h is stored into $s2,
value of 200 is stored into $t1 (in hex: 0x000000c8),
address of A is stored into $s3,
value of 8th index in A is stored into $t0.

Data Segment: shows that the value of $t1; 200, is stored in the 8th index of array A.

Address 0x10010024 contains the value 0x000000c8.

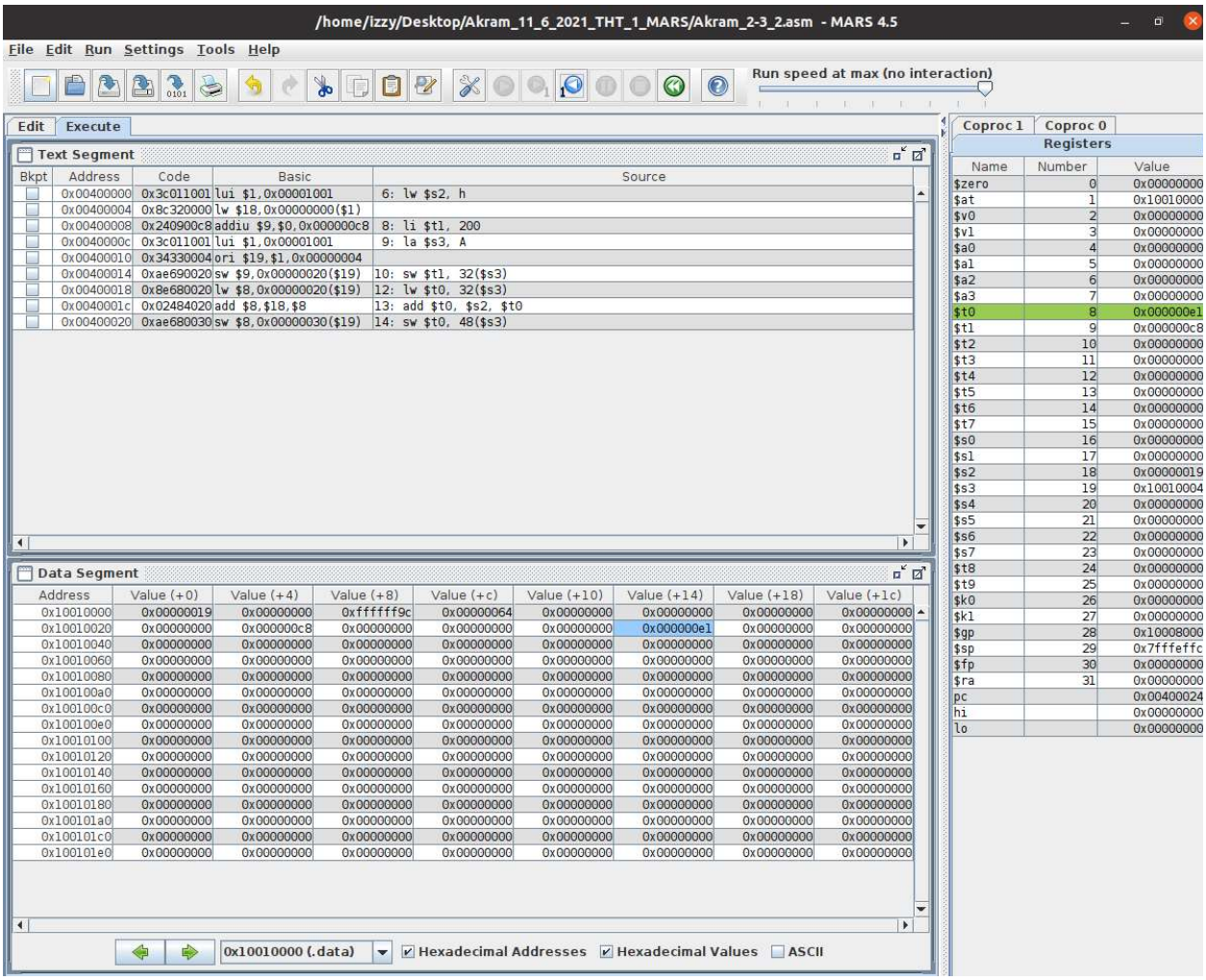

Fig 16. Akram_2-3_2.asm executed in MARS (lines 13-14)

Text Segment: this window shows that the code runs lines 13 to 14 and the effects on the registers and data segments during said run. We added the values of the registers and stored the result into another register and into the data segment.

Registers: this window shows that the:

value of $s2 and $t0 are added, and the result is stored into $t0.

Data Segment: shows that:

After the addition of $s2 and $t0, the result $t0 is stored into the address and allotted to the 12th index of the array A. Address 0x10010034 contains the value 0x000000e1.

2-5_1.asm:

The code written for 2-5_1 in MIPS is shown in Fig 17. It uses three static variables: h, A, and size. Using these three variables; an array A is created, and addition is executed through its usage.

In our case, a value is stored in the array and is added to h. The result from that addition is stored into another part of the array and the data segment.

Akram_2-5_1.asm

```
1  .data
2  h: .word 20
3  A: .word 0-400
4  size: .word 400
5  .text
6  la $t1, A
7  lw $s2, h
8  # initializing A[300] to 13
9  li $t2, 13
10 sw $t2, 1200($t1)
11 lw $t0, 1200($t1)
12 add $t0, $s2, $t0
13 sw $t0, 1200($t1)
```

Fig 17. Akram_2-5_1.asm code displayed in MARS

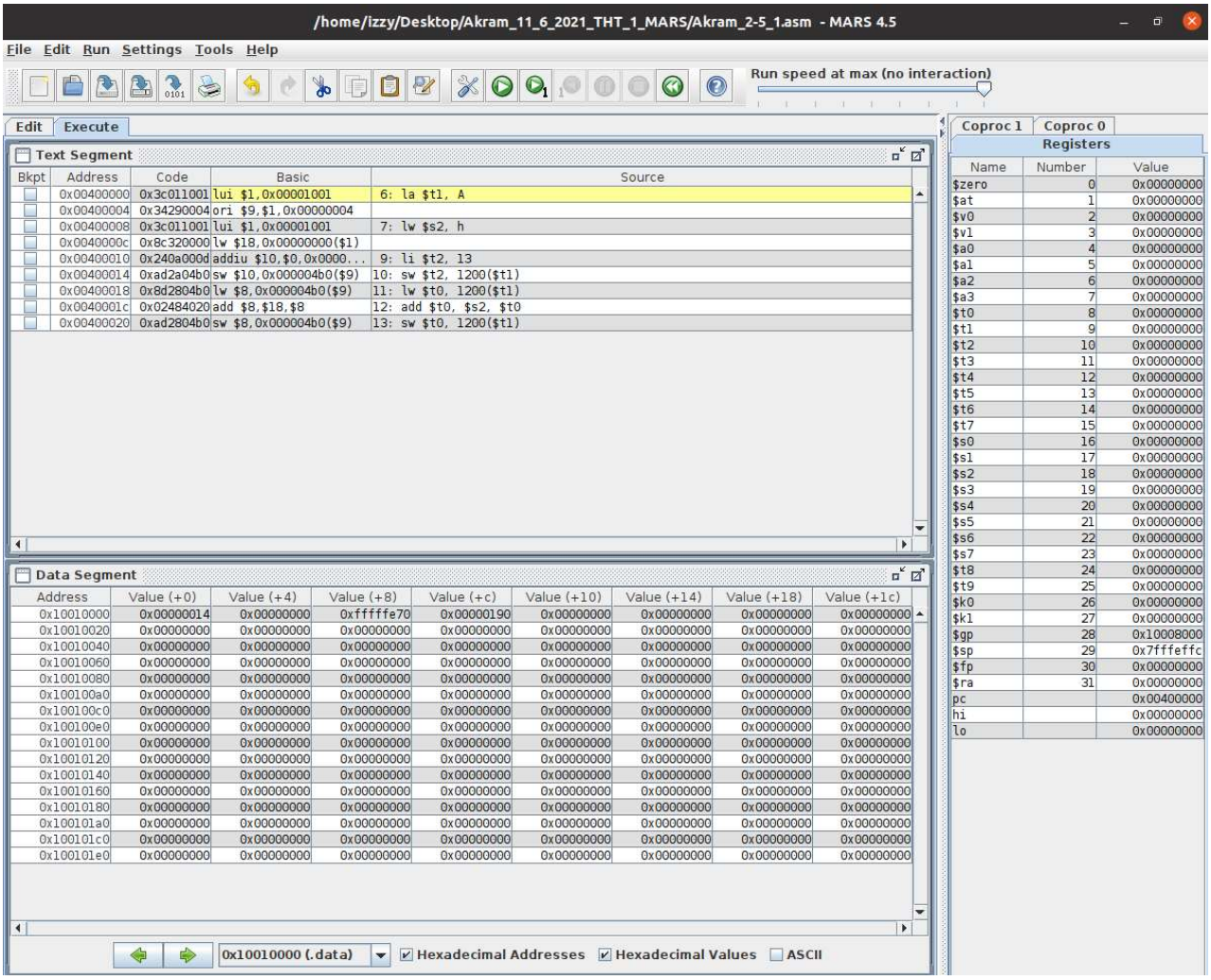

Fig 18. Akram_2-5_1.asm executed in MARS (lines 1-5)

Text Segment: this window shows that the code starts at line 6 since everything prior (line 1-5) doesn't require registers.

Registers: shows the value of the stack pointer, which is 0x7fffeffc. Nothing is stored in registers since no operations have been performed yet.

Data Segment: show the value of the static variables being stored into the addresses.

Variable h is stored in address: 0x10010000 containing the value: 0x00000014.
Variable A is stored in addresses: 0x10010004 and 0x10000008
containing the values: 0x00000000 and 0xfffffe70.
Variable size is stored in address: 0x1001000c containing the value: 0x00000190.

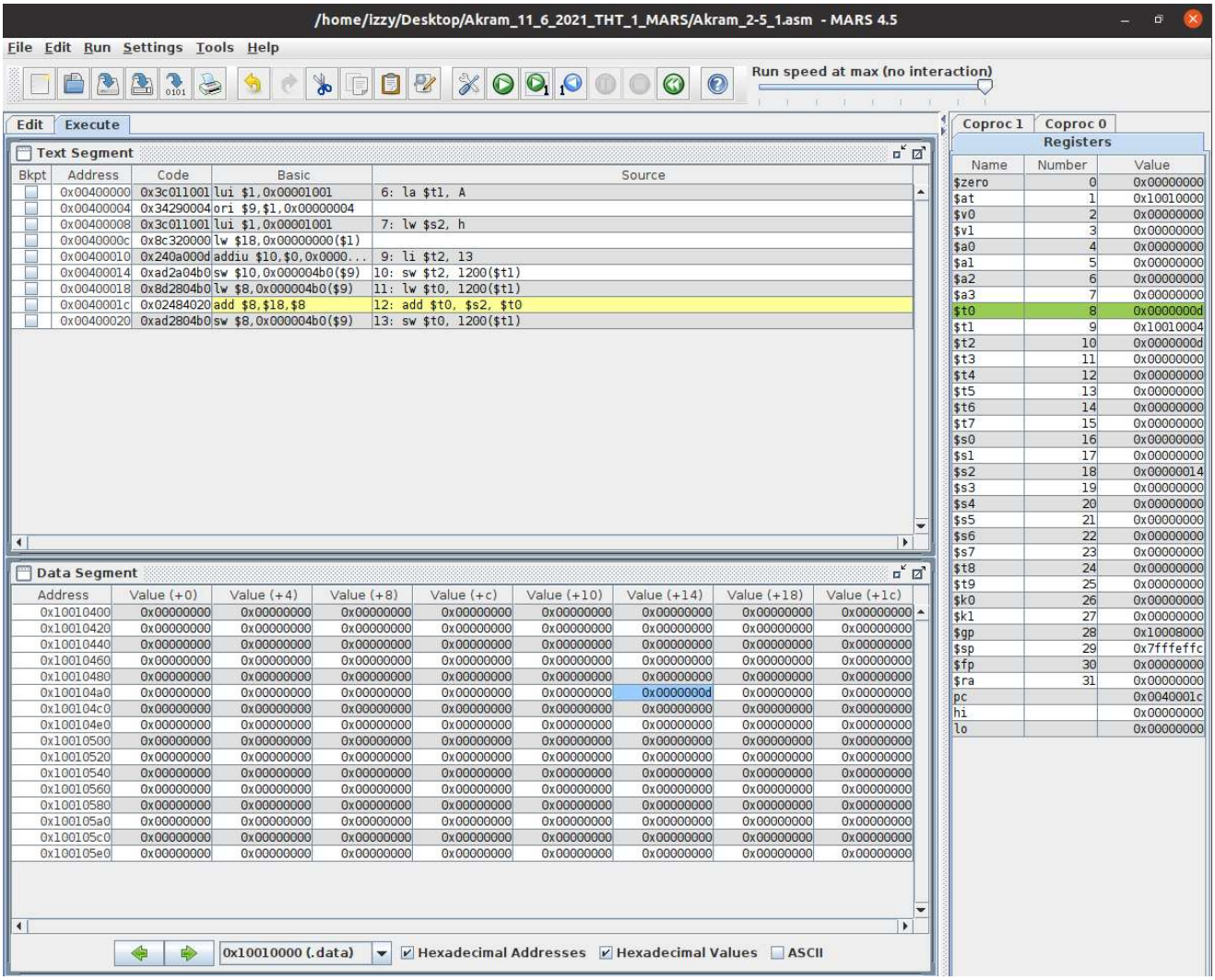

Fig 19. Akram_2-5_1.asm executed in MARS (lines 6-11)

Text Segment: this window shows that the code runs lines 6 to 11 and that the static variables and some values are loaded into the registers during the run. We're also storing the value of a register into the data segment (highlight blue).

Registers: shows that the:

value of h is stored into $s2,
value of 13 is stored into $t2 (in hex: 0x0000000d),
address of A is stored into $t1,
value of 300th index in A is stored into $t0.

Data Segment: shows that the value of $t2; 13, is stored in the 300th index of array A.

Address 0x100104b4 contains the value 0x0000000d.

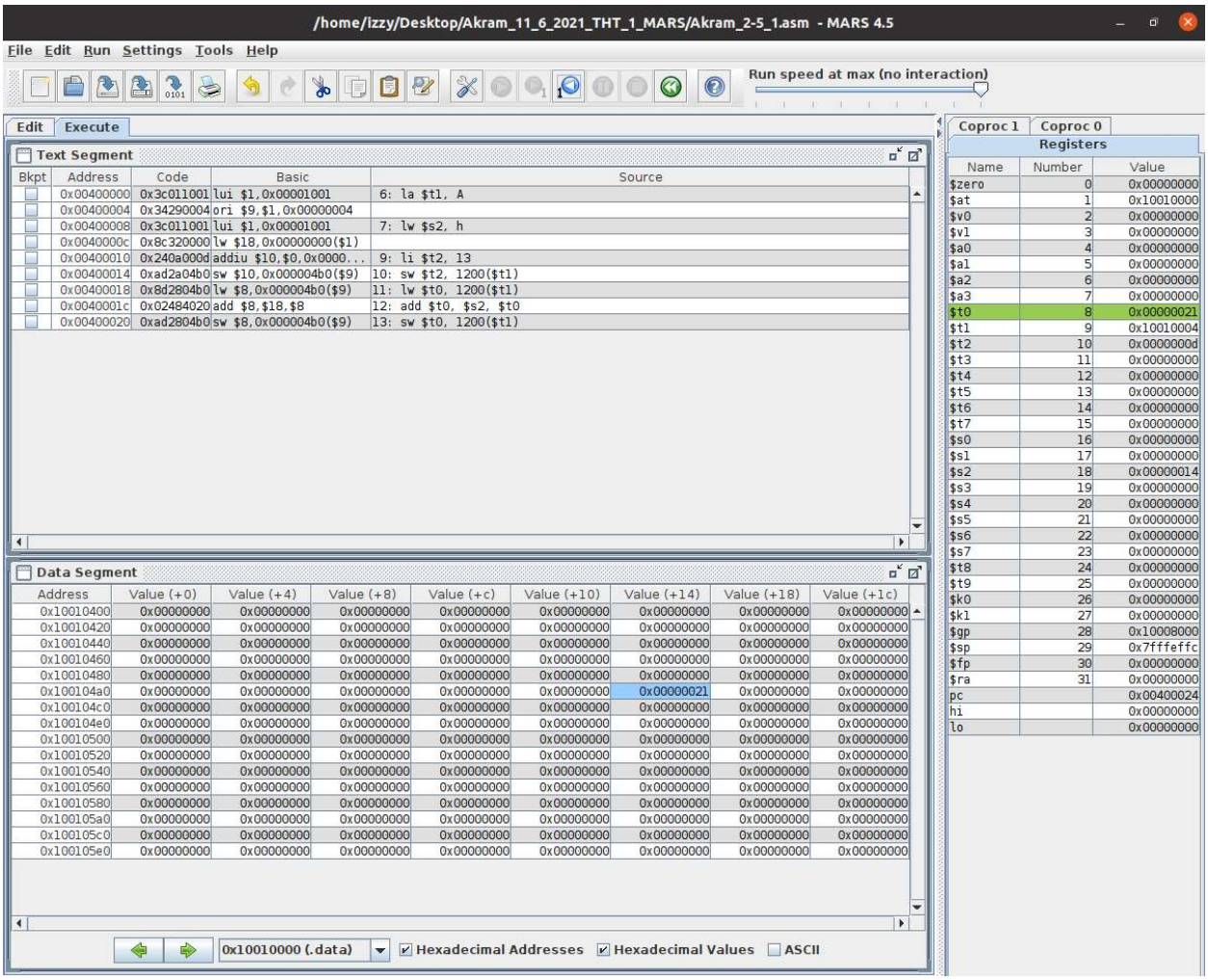

Fig 20. Akram_2-5_1.asm executed in MARS (lines 12-13)

Text Segment: this window shows that the code runs lines 12 to 13 and the effects on the registers and data segments during said run. We added the values of the registers and stored the result into another register and into the data segment.

Registers: this window shows that the:

value of $s2 and $t0 are added, and the result is stored into $t0.

Data Segment: shows that:

After the addition of $s2 and $t0, the result $t0 is stored into the address and allotted to the 300th index of the array A. Address 0x100104b4 contains the value 0x00000021.

2-6_1.asm:

The code written for 2-6_1 in MIPS is shown in Fig 21. It stores values into the registers and uses these values to perform three bitwise operations: sll (shift a register to the left), AND (compares two registers), and OR (also compares two registers).

Akram_2-6_1.asm

```
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
```

Fig 21. Akram_2-6_1.asm code displayed in MARS

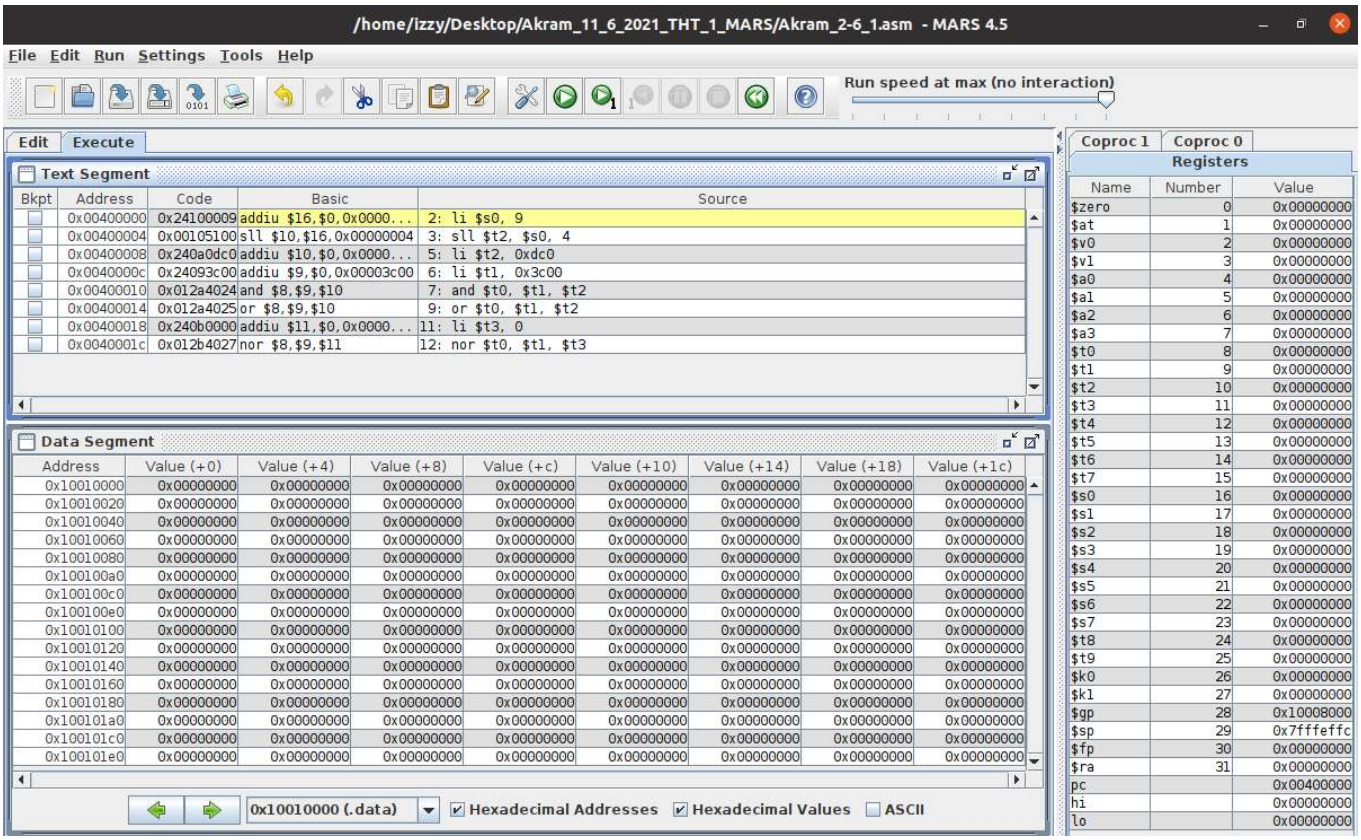

Fig 22. Akram_2-6_1.asm executed in MARS (lines 1)

Text Segment: this window shows that the code starts at line 2 since everything prior (line 1) doesn't require registers.

Registers: shows the value of the stack pointer, which is 0x7fffeffc. Nothing is stored in registers since no operations have been performed yet.

Data Segment: shows that nothing is in the addresses yet.

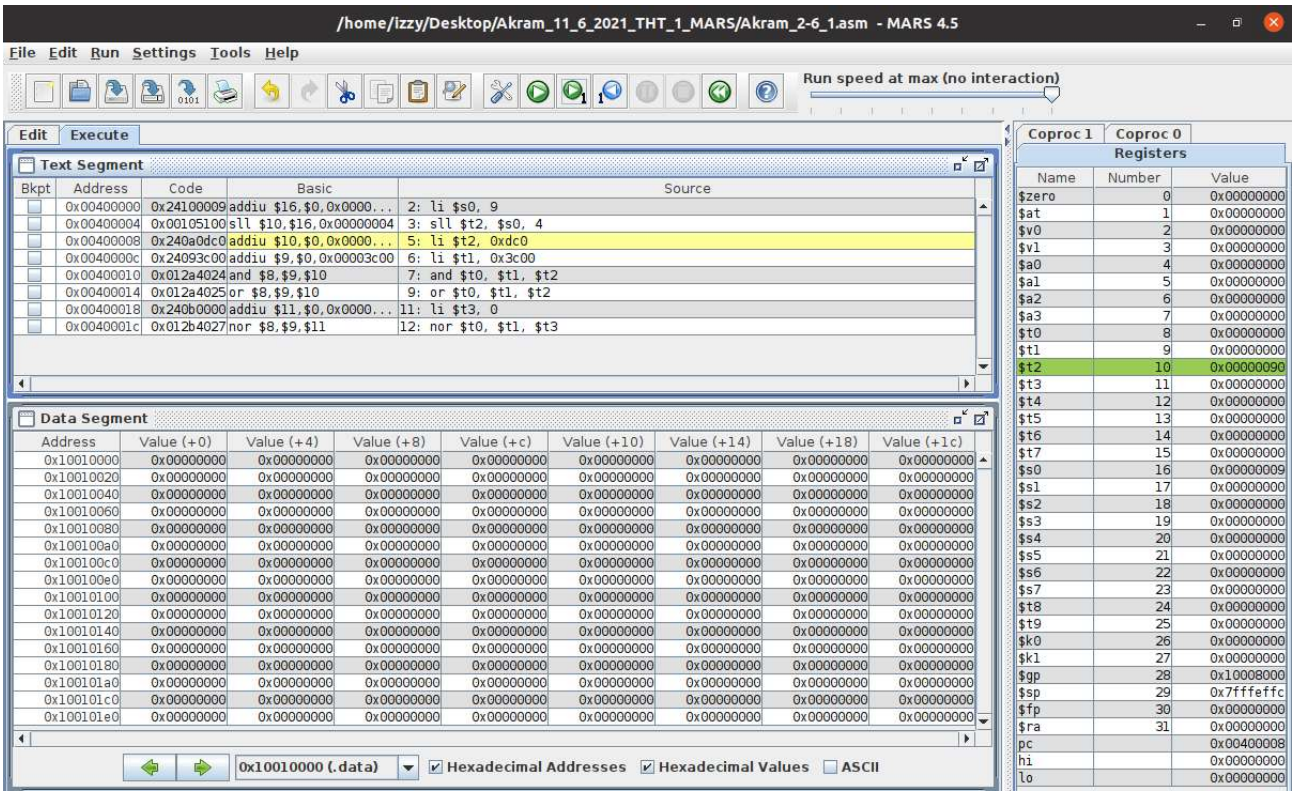

Fig 23. Akram_2-6_1.asm executed in MARS (lines 2-4)

Text Segment: this window shows that the code runs lines 2 to 4 and that the static variables and some values are loaded into the registers during the run. We're also performing bitwise operations to the registers

Registers: shows that the:

value of 9 is stored into $s0 (in hex: 0x00000009),
shift left of register $s0 by 4 bits is stored into $t2 (in hex: 0x00000090).

Data Segment: shows that nothing is in the addresses yet.

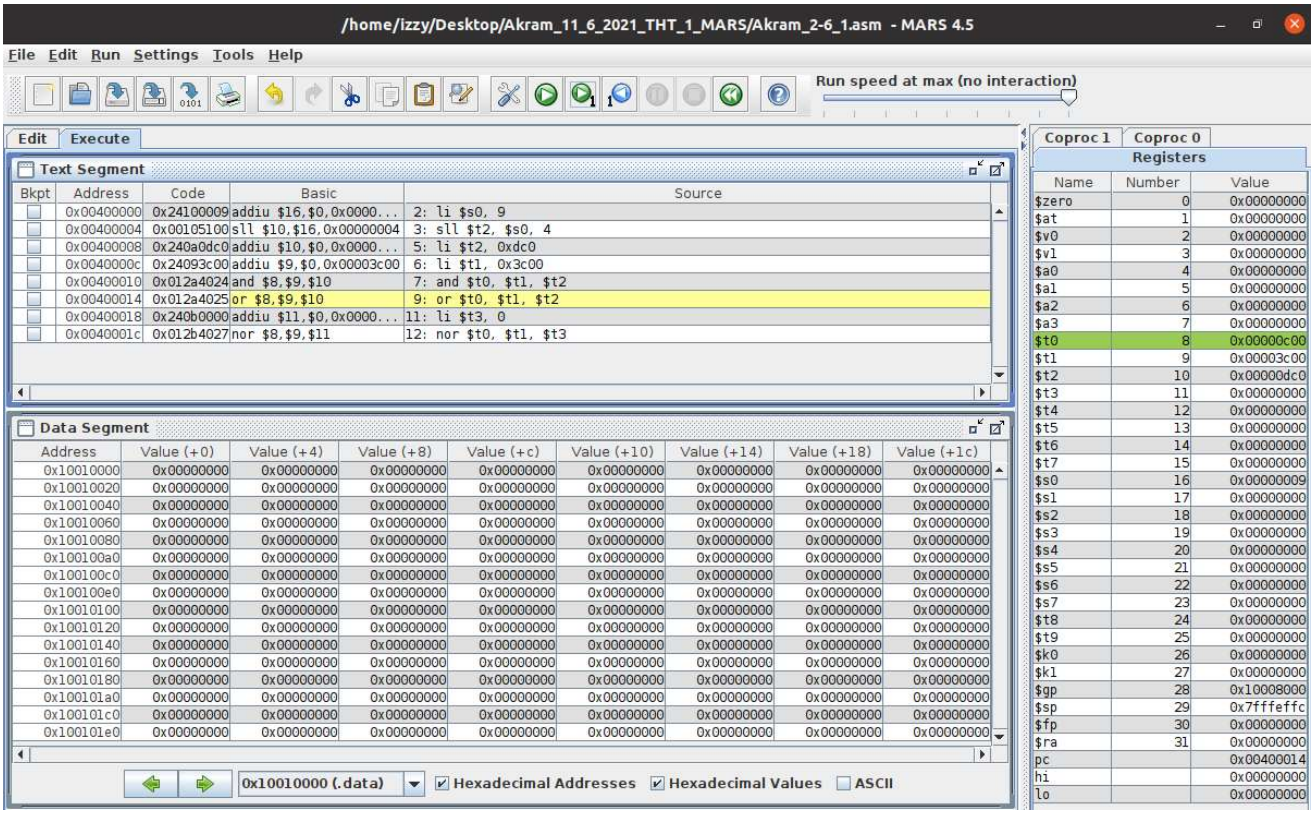

Fig 24. Akram_2-6_1.asm executed in MARS (lines 5-8)

Text Segment: this window shows that the code runs lines 5 to 8 and the effects on the registers and data segments during said run. We added the values of the registers and stored the result into another register. We're also performing bitwise operations to the registers.

Registers: this window shows that the:

hex value of 0xdc0 is stored into $t2,
hex value of 0x3c00 is stored into $t1,
AND (operator) comparison of $t1 and $t2 is stored into $t0, which is 0x00000c00

Data Segment: shows that nothing is in the addresses yet.

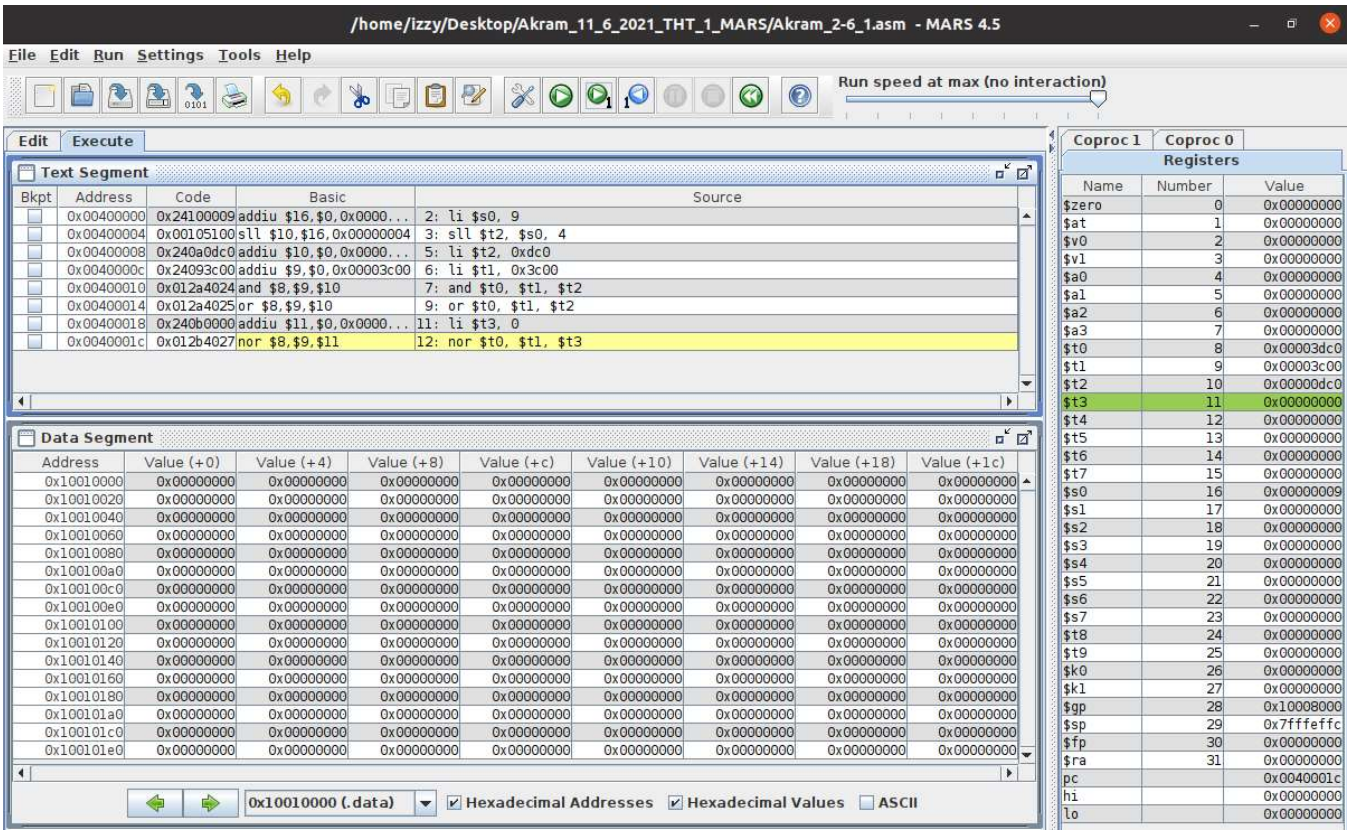

Fig 25. Akram_2-6_1.asm executed in MARS (lines 9-11)

Text Segment: this window shows that the code runs lines 9 to 11 and the effects on the registers and data segments during said run. We added the values of the registers and stored the result into another register. We're also performing bitwise operations to the registers.

Registers: this window shows that the:

> OR (operator) comparison of $t1 and $t2 is stored into $t0, which is 0x00003dc0, Value of 0 is stored into $t3.

Data Segment: shows that nothing is in the addresses yet.

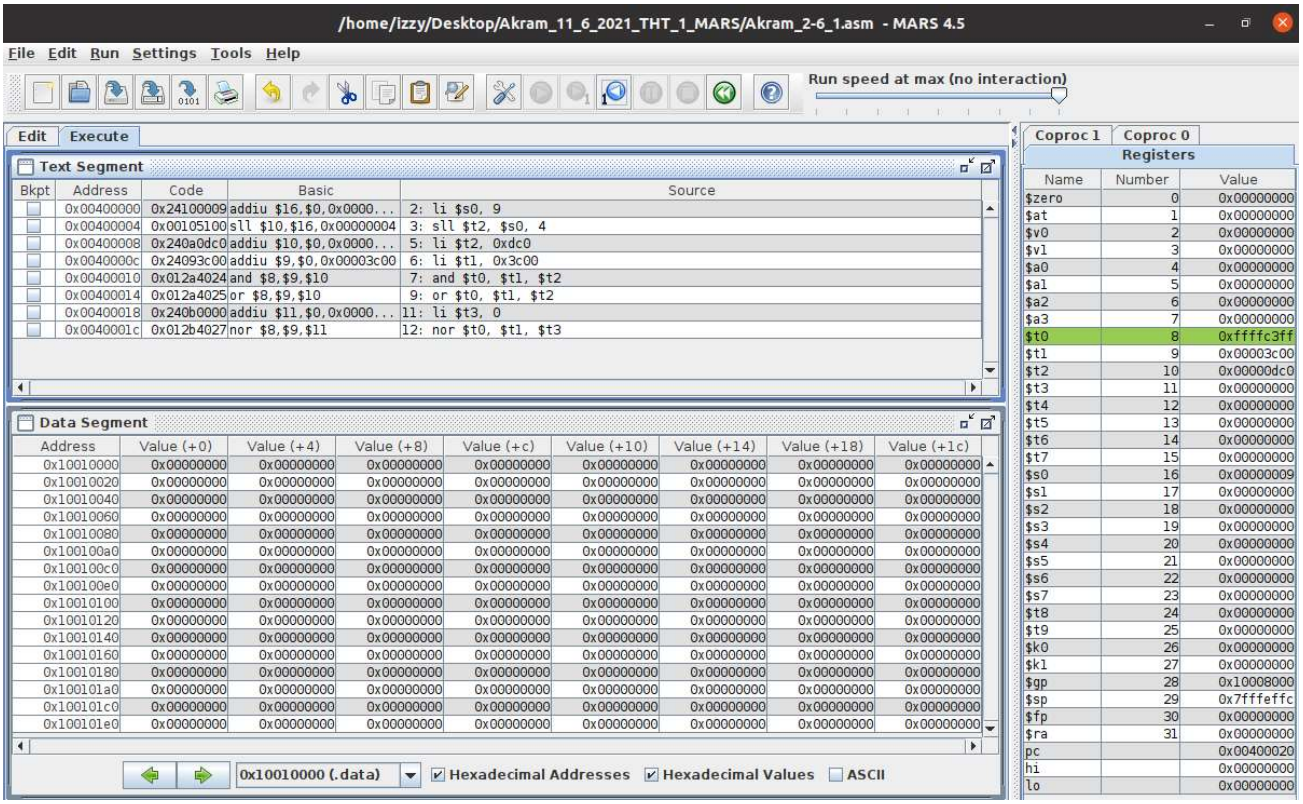

Fig 26. Akram_2-6_1.asm executed in MARS (lines 12)

Text Segment: this window shows that the code runs lines 9 to 11 and the effects on the registers and data segments during said run. We added the values of the registers and stored the result into another register. We're also performing bitwise operations to the registers.

Registers: this window shows that the:

NOR (operator) comparison of $t1 and $t3 is stored into $t0, which is 0xffffc3ff.

Data Segment: shows that nothing is in the addresses yet.

2-7_1.asm:

The code written for 2-7_1 in MIPS is shown in Fig 27. It uses five static variables: f, g, h, i, and j. Using these five variables; conditions are set up and if the variables satisfy those conditions; arithmetic operations are performed.

If i and j, then g and h will be added, and the result will be stored in f,
else,
g and h will be subtracted, and the result will be stored in f.

Akram_2-7_1.asm

```
1  .data
2  f: .word 1
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 bne $s3, $s4, Else
14 # go to Else if i != j
15 add $s0, $s1, $s2
16 # f = g + h (skipped if i = j)
17 j Exit
18 # go to Exit
19 Else:
20 sub $s0, $s1, $s2
21 # f = g - h (skipped if i = j)
22 Exit:
```

Fig 27. Akram_2-7_1.asm code displayed in MARS

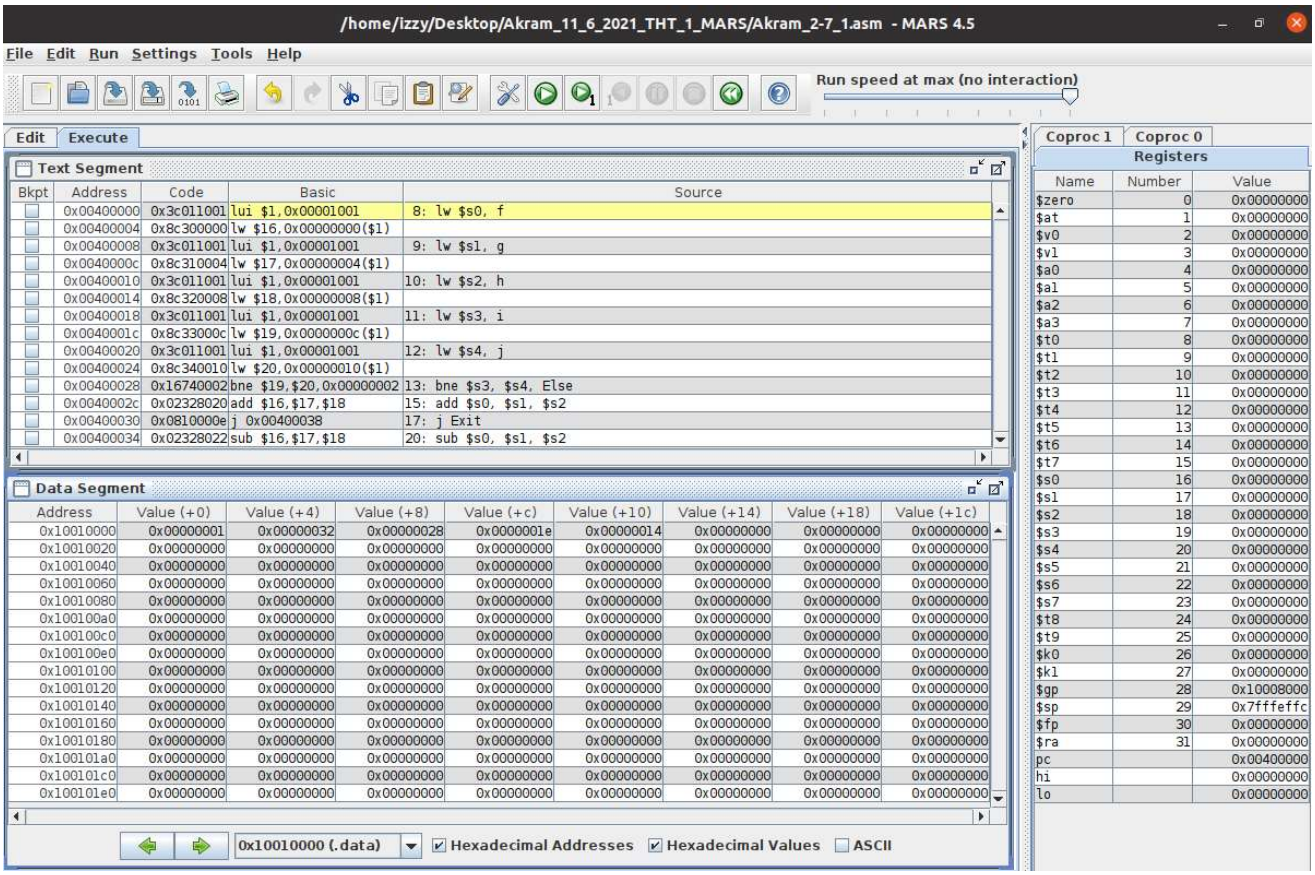

Fig 28. Akram_2-7_1.asm executed in MARS (lines 1-7)

Text Segment: this window shows that the code starts at line 8 since everything prior (lines 1-7) doesn't require registers.

Registers: shows the value of the stack pointer, which is 0x7fffeffc. Nothing is stored in registers since no operations have been performed yet.

Data Segment: show the value of the static variables being stored into the addresses.

Variable f is stored in address: 0x10010000 containing the value: 0x00000001.
Variable g is stored in address: 0x10010004 containing the value: 0x00000032.
Variable h is stored in address: 0x10010008 containing the value: 0x00000028.
Variable i is stored in address: 0x1001000c containing the value: 0x0000001e.
Variable j is stored in address: 0x10010010 containing the value: 0x00000014.

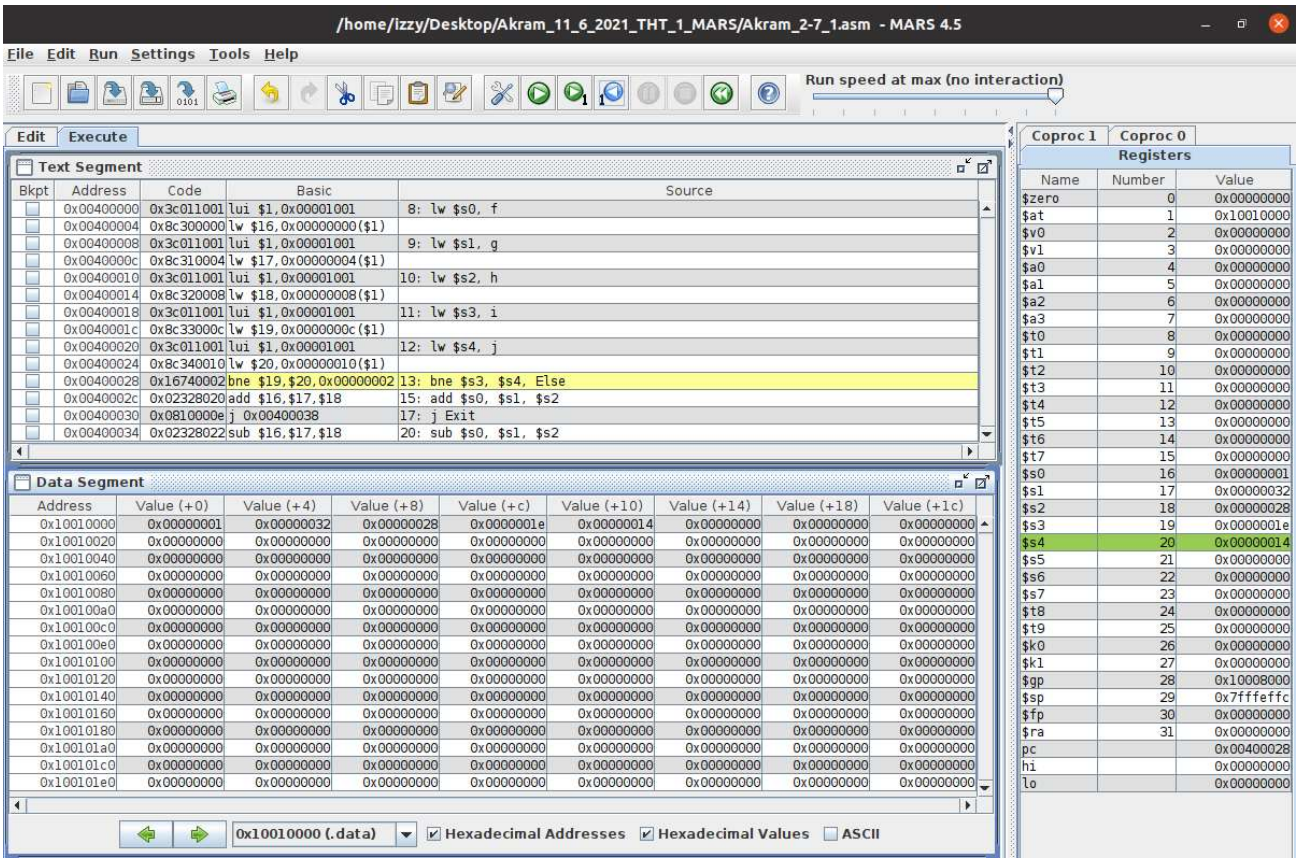

Fig 29. Akram_2-6_1.asm executed in MARS (lines 8-12)

Text Segment: this window shows that the code runs lines 8 to 12 and that the static variables and some values are loaded into the registers during the run.

Registers: this window shows that the:

value of f is stored into $s0,
value of g is stored into $s1,
value of h is stored into $s2,
value of i is stored into $s3,
value of j is stored into $s4,

Data Segment: remains the same because lines 8 to 12 don't store anything into them or change any values.

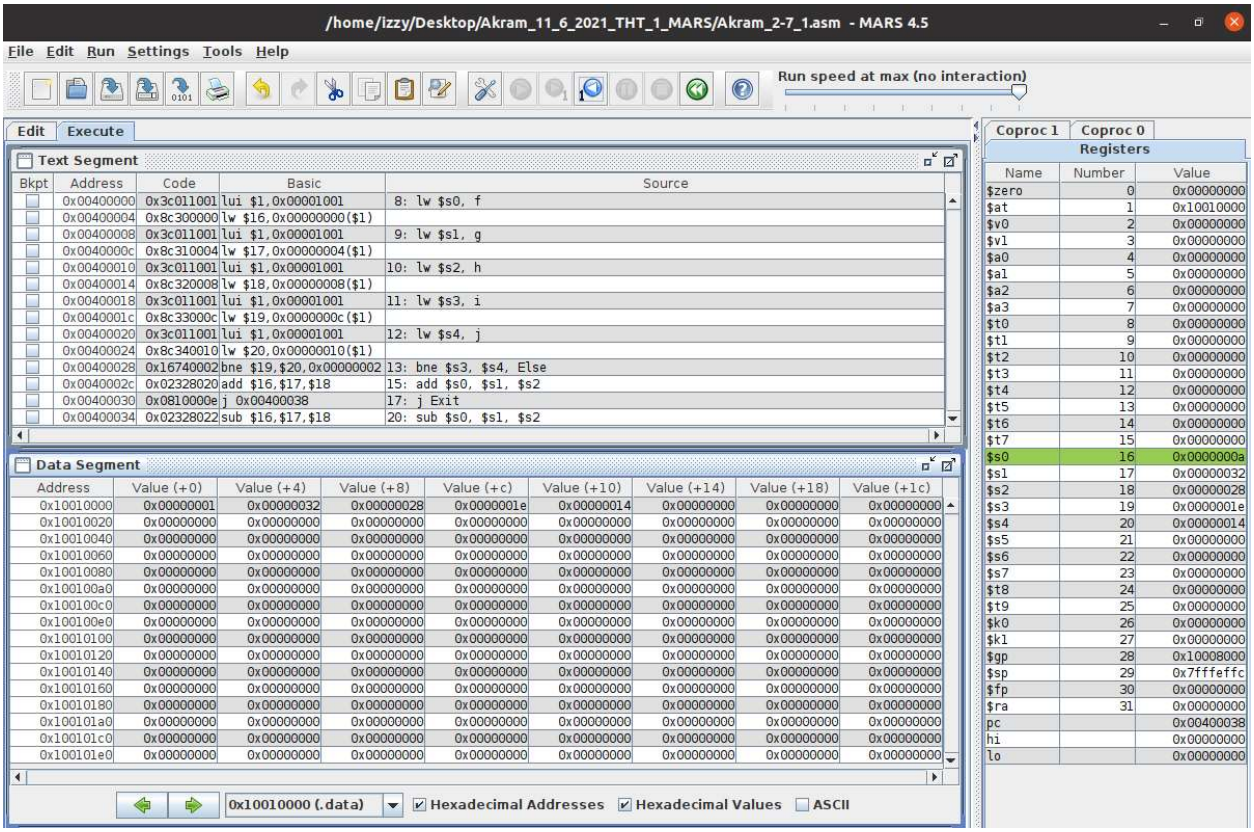

Fig 30. Akram_2-6_1.asm executed in MARS (lines 13-20)

Text Segment: this window shows that the code runs lines 13 to 20 and the effects on the registers and data segments during said run.

We checked if the values of the two registers are equal, and if they are, they're added and the result is stored into another register. Else, they are subtracted and stored into another register.

Registers: this window shows that the:

value of $s1 and $s2 are added, and the result is stored into $s0

Data Segment: remains the same because lines 8 to 12 don't store anything into them or change any values.

2-7_2.asm:

The code written for 2-7_2 in MIPS is shown in Fig 31. It uses four static variables: i, k, save, and size. Using these four variables; conditions are set up and if the variables satisfy those conditions; loop operations are performed.

In our case and array called save is generated and the value 10 is saved into the 5th index. The loop is run and checked if the value 10 is saved in the ith index of that array:

If it is: increment i by 1,

Else,

Exit the loop.

Akram_2-7_1.asm | Akram_2-7_2.asm

```
1  .data
2  i: .word 5
3  k: .word 10
4  save: .word 0-100
5  size: .word 100
6
7  .text
8  lw $s3, i
9  lw $s5, k
10 la $s6, save
11 sw $s5, 20($s6)
12 # save 10 in save[5]
13 Loop:
14 sll $t1, $s3, 2
15 # Temp reg $t1 = i * 4
16 add $t1, $t1, $s6
17 # $t1 = address of save[i]
18 lw $t0, 0($t1)
19 # Temp reg $t0 - save[i]
20 bne $t0, $s5, Exit
21 # go to Exit if save[i] != k
22 addi $s3, $s3, 1
23 # i = i + 1
24 j Loop
25 # go to Loop
26 Exit:
```

Fig31. Akram_2-7_2.asm code displayed in MARS

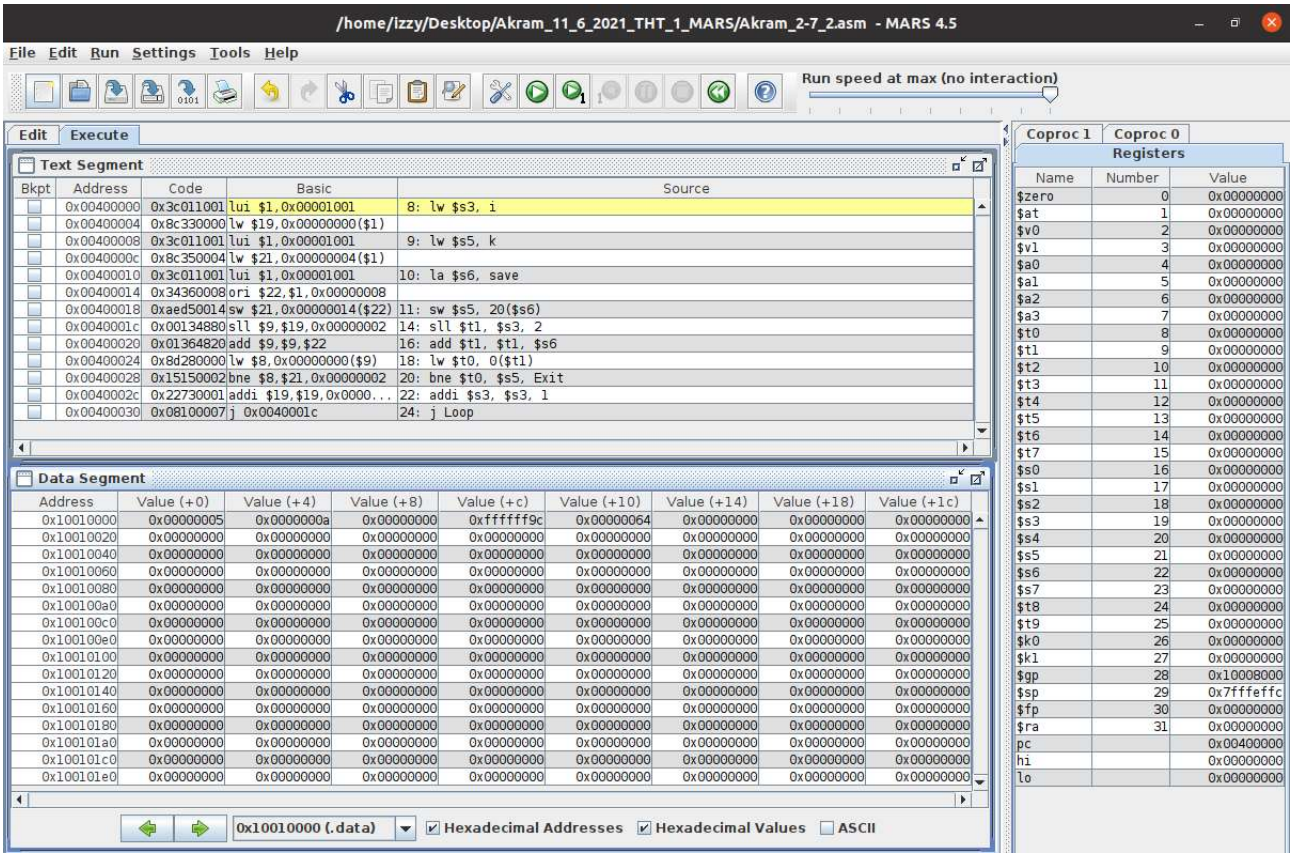

Fig 32. Akram_2-7_2.asm executed in MARS (lines 1-7)

Text Segment: this window shows that the code starts at line 8 since everything prior (lines 1-7) doesn't require registers.

Registers: shows the value of the stack pointer, which is 0x7fffeffc. Nothing is stored in registers since no operations have been performed yet.

Data Segment: show the value of the static variables being stored into the addresses.

Variable i is stored in address: 0x10010000 containing the value: 0x00000005.
Variable k is stored in address: 0x10010004 containing the value: 0x0000000a.
Variable save is stored in addresses: 0x10010008 and 0x1001000c
containing the values: 0x00000000 and 0xffffff9c.
Variable size is stored in address: 0x10010010 containing the value: 0x00000064.

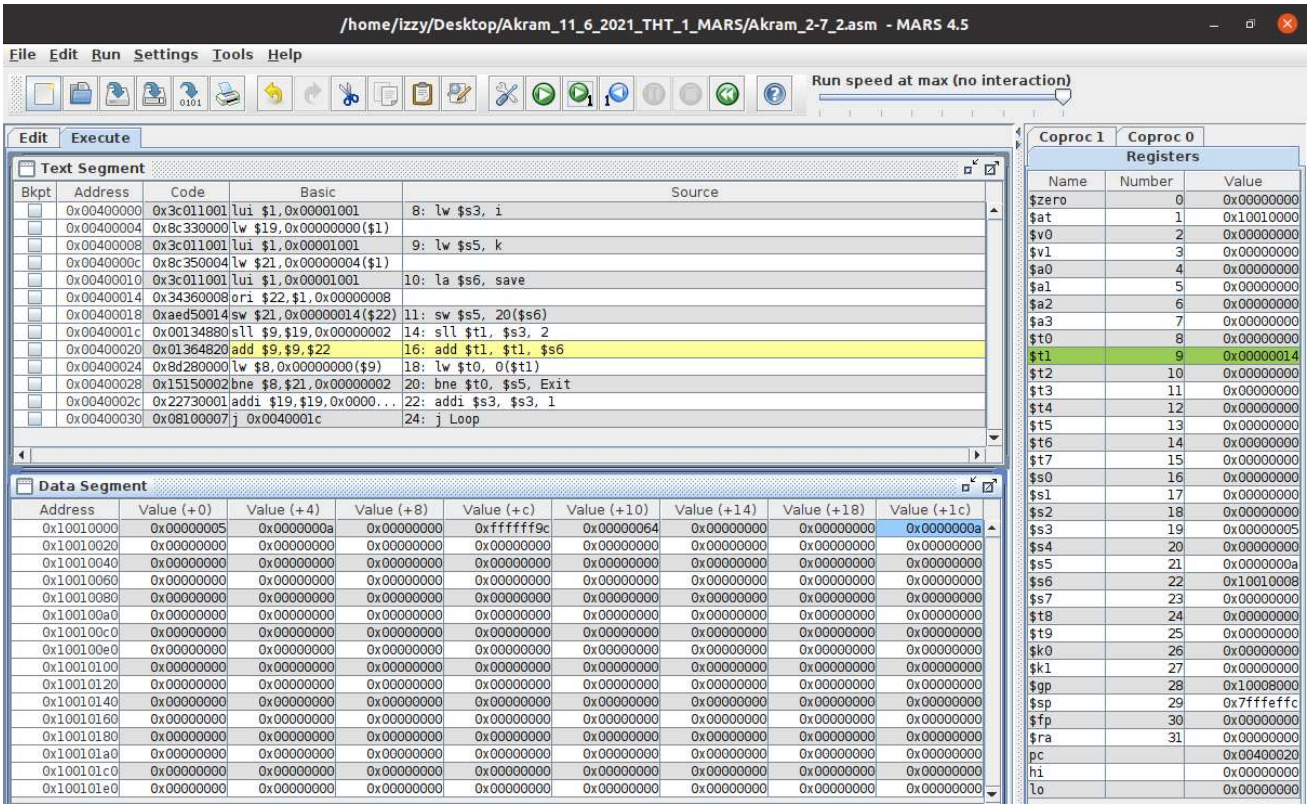

Fig 33. Akram_2-7_2.asm executed in MARS (lines 8-15)

Text Segment: this window shows that the code runs lines 8 to 15 and that the static variables and some values are loaded into the registers during the run. We're also storing the value of a register into the data segment (highlight blue).

Registers: this window shows that the:

value of i is stored into $s3,
value of k is stored into $s5,
address of save is stored into $s6,
shift left of register $s3 by 2 bits is stored into $t1 (in hex: 0x00000014).

Data Segment: shows that the value of $s5; 10, is stored in the 5th index of array save.

Address 0x1001001c contains the value 0x0000000a.

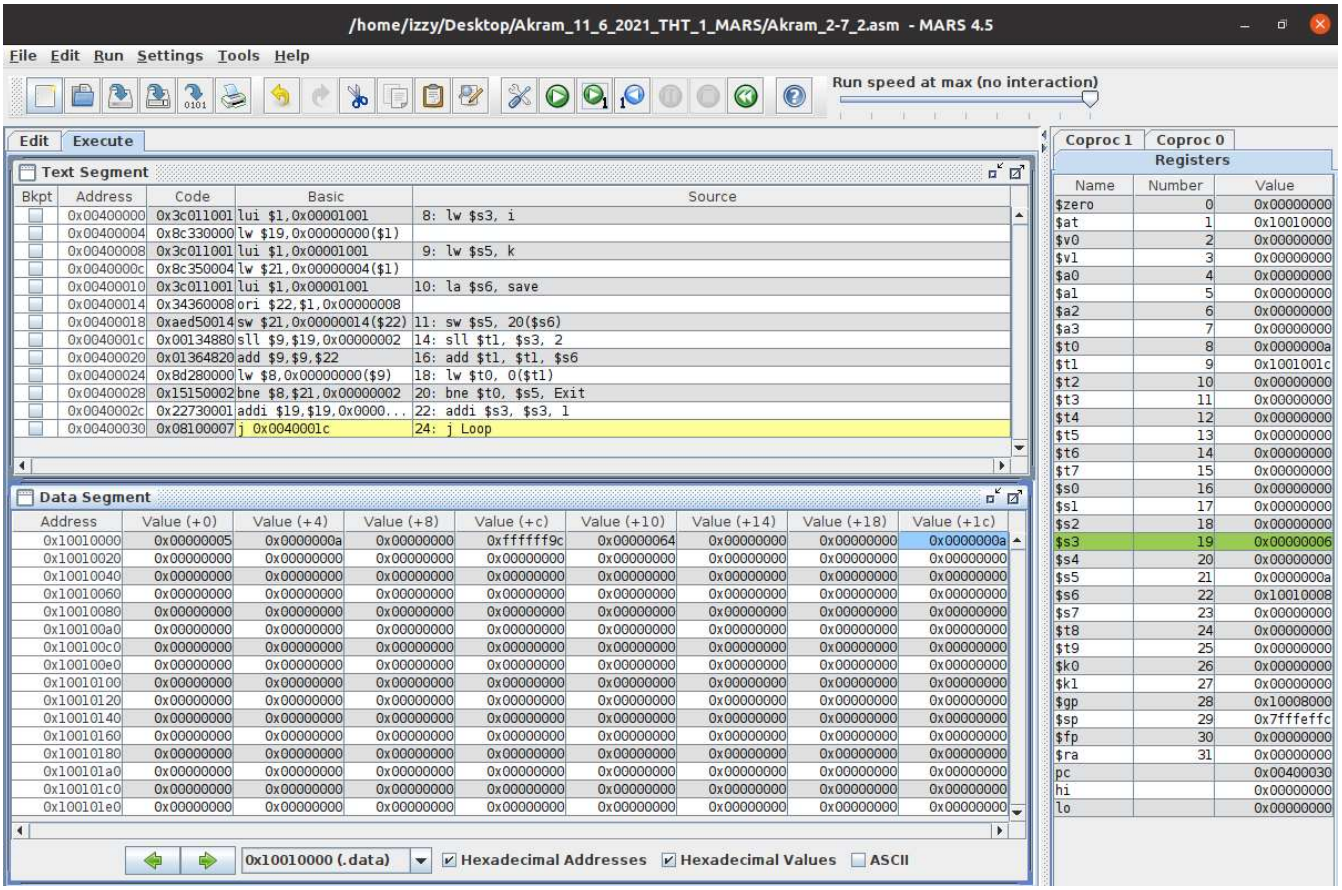

Fig 34. Akram_2-7_2.asm executed in MARS (lines 16-23)

Text Segment: this window shows that the code runs lines 16 to 23 and the effects on the registers and data segments during said run. We added the two registers to obtain the address of the 5th index of the array save and stored that address in a register. We also stored the value of that address into another register.

We checked if this value is equal to the value stored in $s5, and if they are, increment the value at $s3 by 1. Else, exit the loop.

Registers: this window shows that the:

address of 5th index of array save is stored in $t1,
value of the 5th index of array save is stored $t0,
incrementation of $s3 by 1.

Data Segment: remains the same because lines 16 to 23 don't store anything into them or change any values.

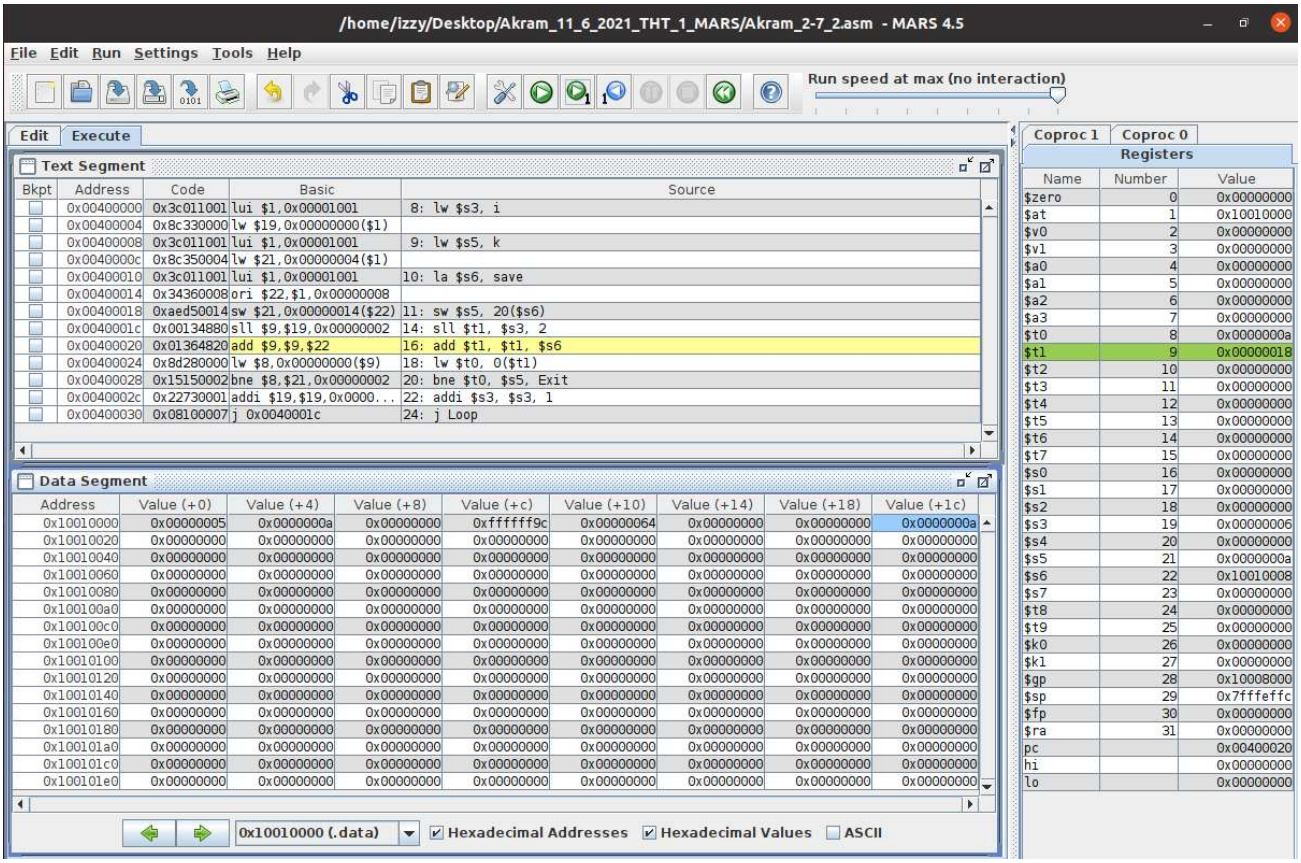

Fig 35. Akram_2-7_2.asm executed in MARS (lines 24 & 14)

Text Segment: this window shows that the code runs lines 24 and 14 and that we jumped back to address 0x004001c and its ran again as we're in a loop.

Registers: shows that the:

shift left of register $s3 by 2 bits is stored into $t1 (in hex: 0x00000018).

Data Segment: remains the same because lines 24 and 14 don't store anything into them or change any values.

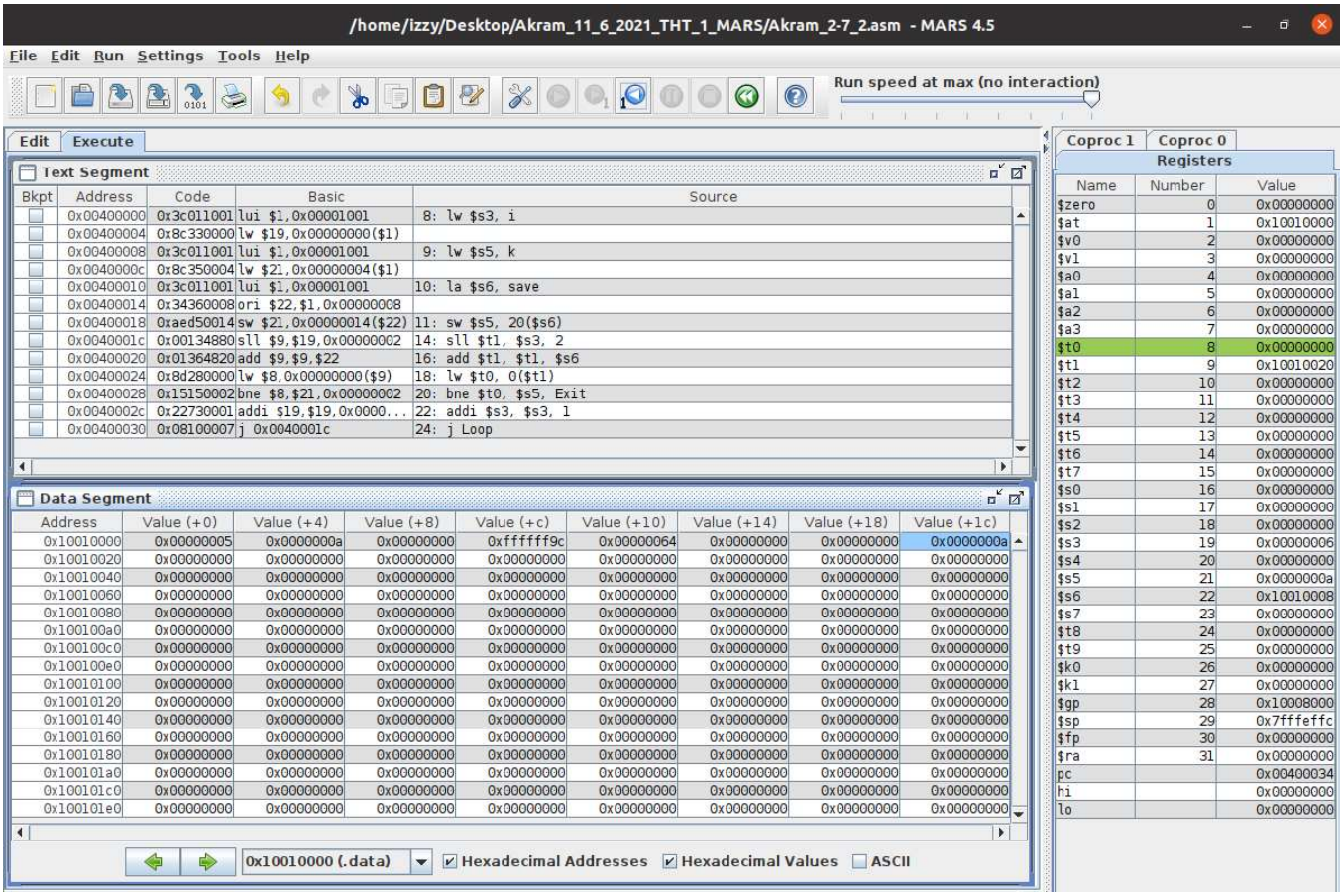

Fig 35. Akram_2-7_2.asm executed in MARS (lines 16-26)

Text Segment: this window shows that the code runs lines 16 to 26 and the effects on the registers and data segments during said run. We added the two registers to obtain the address of the 6th index of the array save and stored that address in a register. We also stored the value of that address into another register.

We checked if this value is equal to the value stored in $s5, and if they are, increment the value at $s3 by 1. Else, exit the loop. In this case, we're exiting.

Registers: this window shows that the:

address of 6th index of array save is stored in $t1,
value of the 6th index of array save is stored $t0.

Data Segment: remains the same because lines 16 to 26 don't store anything into them or change any values.

2-8_1.asm:

The code written for 2-8_1 in MIPS is shown in Fig 36. It uses three static variables: a, b, and c. Using these three variables; it'll enter a function called myadd, which will take two parameters and return their sum by storing the result into a register.

After the return, the value in that register is added by zero and is stored into a register. In our case, a and b are our parameters of the myadd function. c is equal to the zero added to the result stored in the return register and later into the data segment.

Akram_2-8_1.asm

```
1   .data
2   a: .word 4
3   b: .word 15
4   c: .word 0
5   .text
6   lw $a0, a
7   lw $a1, b
8   lw $s0, c
9   jal myadd
10  add $s0, $zero, $v0
11  sw $s0, c
12  j Exit
13  myadd:
14          add $v0, $a0, $a1
15          jr $ra
16  Exit:
```

Fig 37. Akram_2-8_1.asm code displayed in MARS

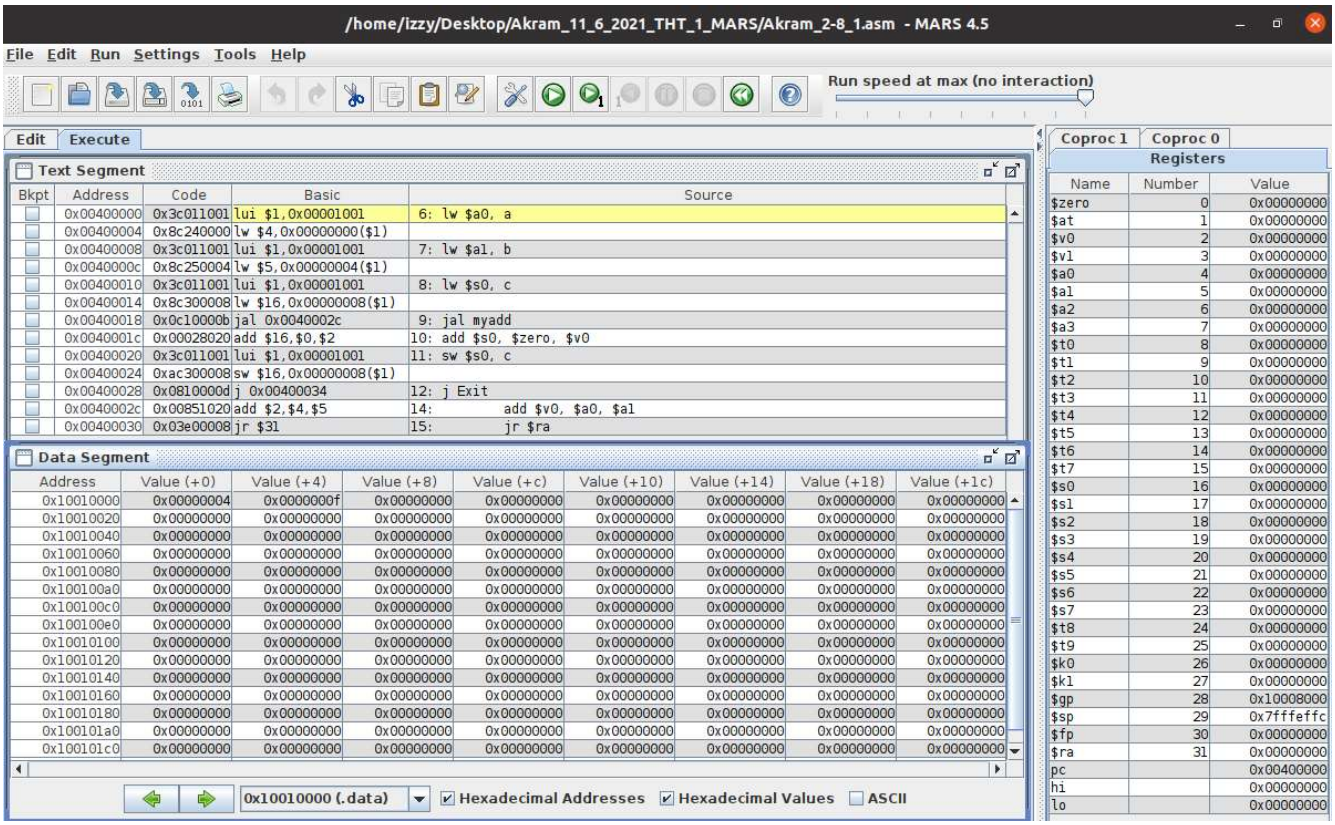

Fig 38. Akram_2-8_1.asm executed in MARS (lines 1-6)

Text Segment: this window shows that the code starts at line 6 since everything prior (lines 1-7) doesn't require registers.

Registers: shows the value of the stack pointer, which is 0x7fffeffc. Nothing is stored in registers since no operations have been performed yet.

Data Segment: show the value of the static variables being stored into the addresses.

Variable a is stored in address: 0x10010000 containing the value: 0x00000004,
Variable b is stored in address: 0x10010004 containing the value: 0x0000000f,
Variable c is stored in address: 0x10010008 containing the value: 0x00000000.

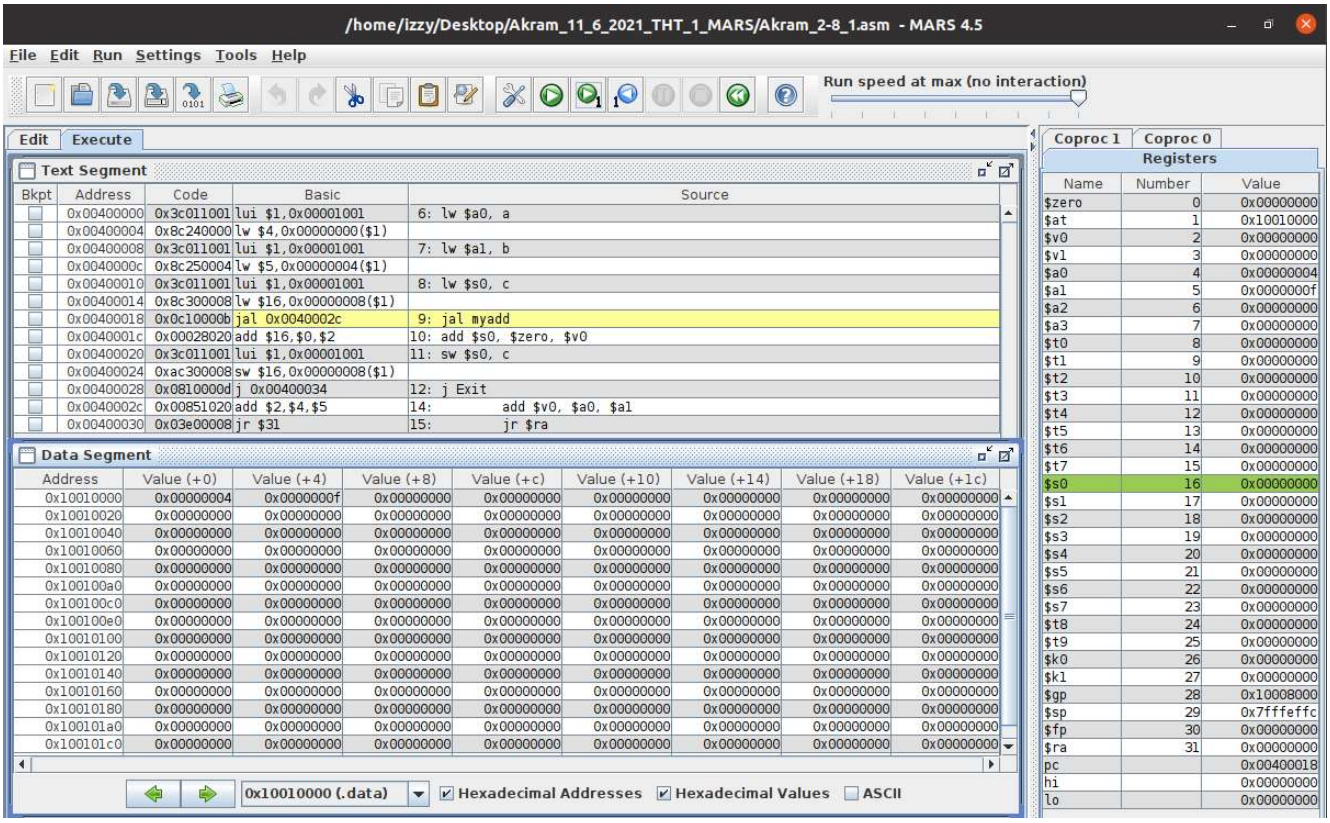

Fig 39. Akram_2-8_1.asm executed in MARS (lines 6-8)

Text Segment: this window shows that the code runs lines 6 to 8 and that the static variables and some values are loaded into the registers during the run.

Registers: this window shows that the:

value of a is stored into $a0,
value of b is stored into $a1,
value of c is stored into $s0,
a and b are stored into registers $a0 and $a1 since those are the registers for the parameters.

Data Segment: remains the same because lines 6 to 8 don't store anything into them or change any values.

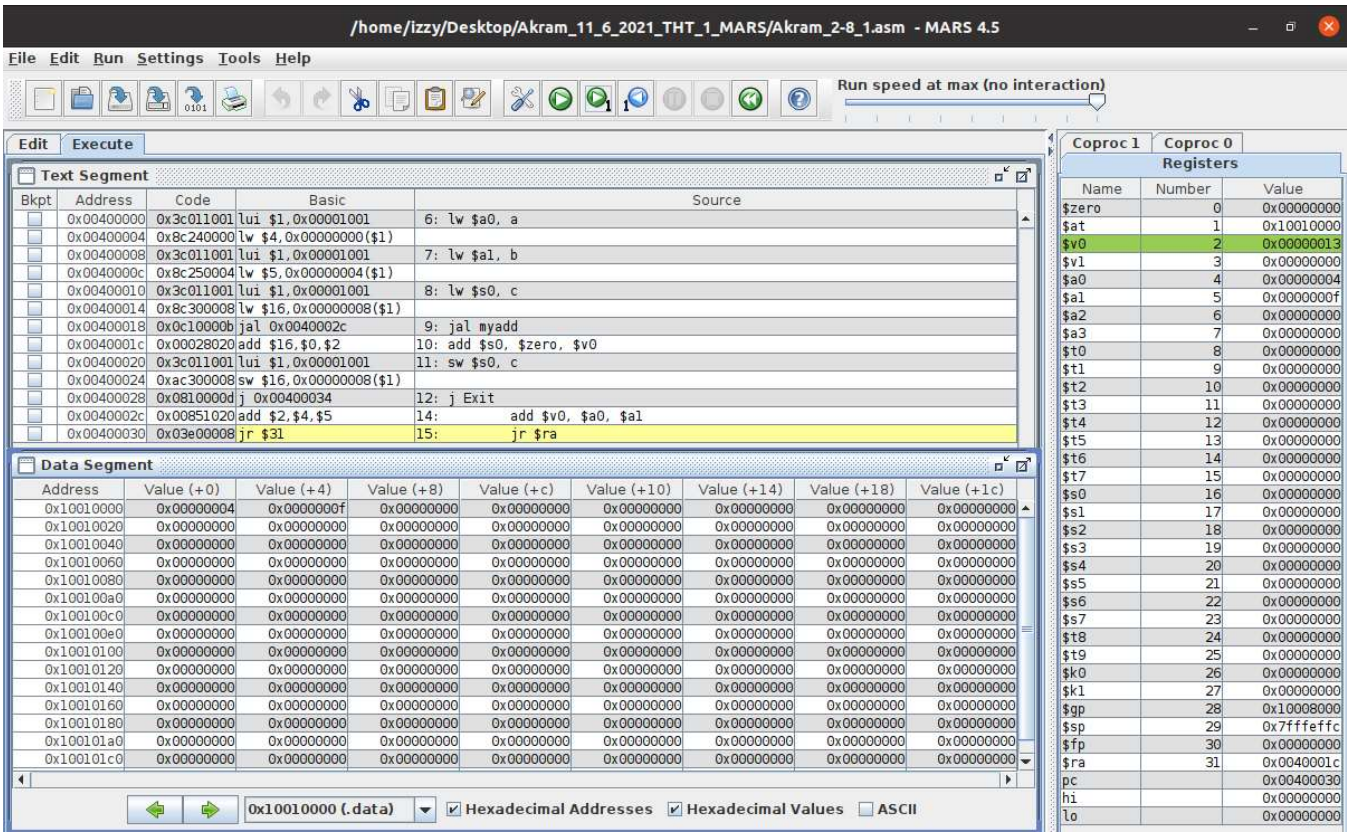

Fig 40. Akram_2-7_2.asm executed in MARS (lines 9 & 14)

Text Segment: this window shows that the code runs lines 9 and 14 and that we jumped to the function call and performed an addition on the two parameters. The result is stored in the return register of the function.

Registers: shows that the:

Values of $a0 and $a1 are added, and the result is stored into register $v0.

Data Segment: remains the same because lines 9 and 14 don't store anything into them or change any values.

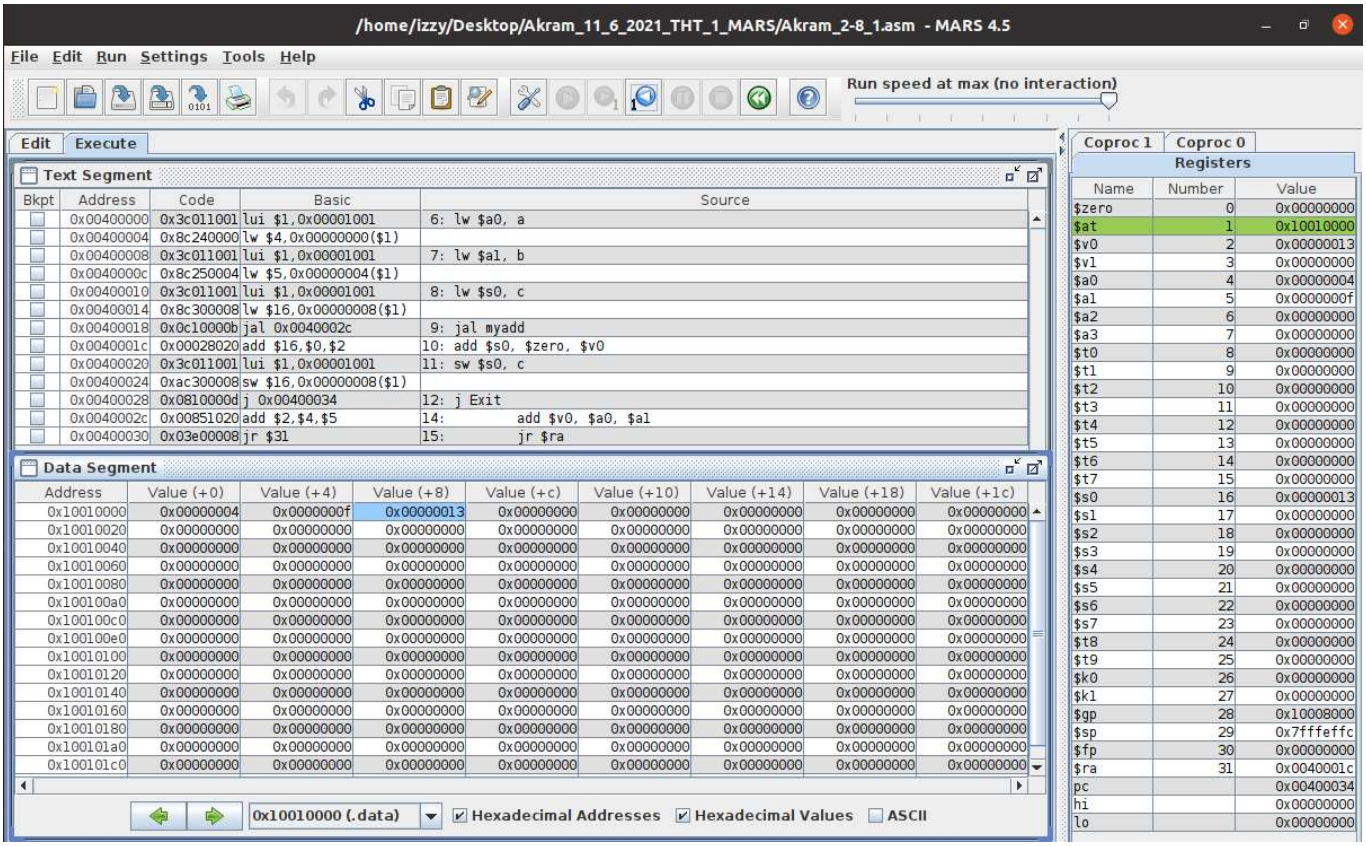

Fig 41. Akram_2-7_2.asm executed in MARS (lines 10-12, 15 & 16)

Text Segment: this window shows that the code runs lines 10-12, 15, and 16 and that we jumped out of the function call and back into the next line of code, which adds zero to the return of the function value and stores it into a register and the data segment.

Registers: shows that the:

Values of $v0 and $zero are added, and the result is stored into $s0.

Data Segment: shows that:

After the addition of $v0 and $zero, the result $s0 is stored into the address and allotted to the variable c. Address 0x1001008 contains the value 0x00000013.

Intel X86 ISA on Windows 32-bit Compiler:

C is a programing language that produces a .c file. For this demonstration we're running C on Visual Studios for the Intel X86 ISA on Windows 32-bit compiler. This compiler operates as little endian, so the least significant byte (LSB) is stored in the lowest memory address.

2-2_1.c:

The code written for 2-2_1 in C is shown in Fig 42. It uses five static variables: a, b, c, d, and e. Using these five variables; arithmetic is performed and stored in the variables.

In our case, we're adding the value of b and c and the sum is stored in a. We're also subtracting the values of a and e and storing that result in d.

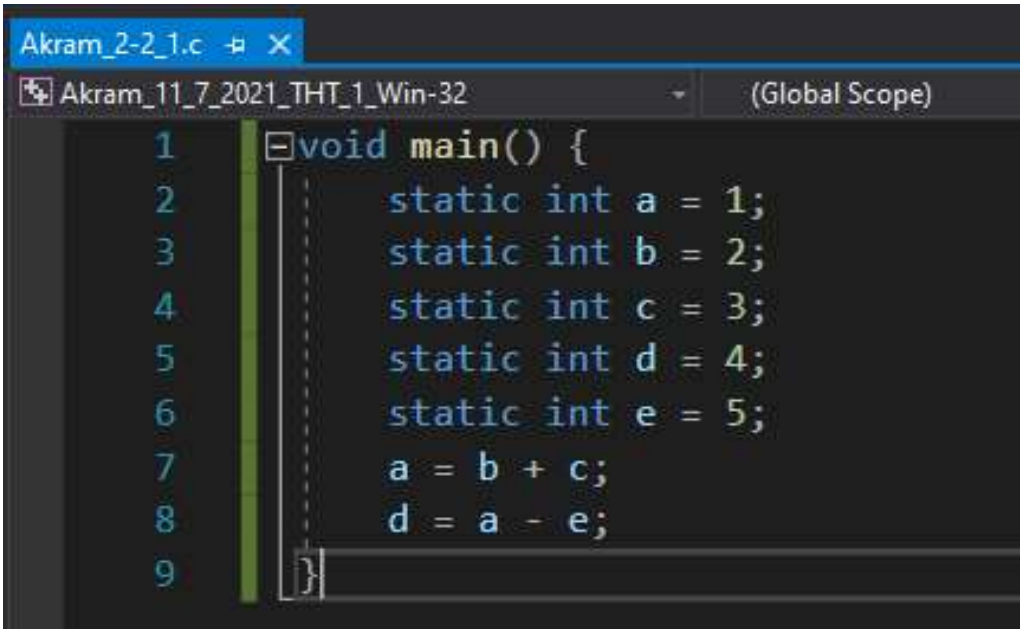

Fig 42. Akram_2-2_1.c executed in VS Win-32 bit

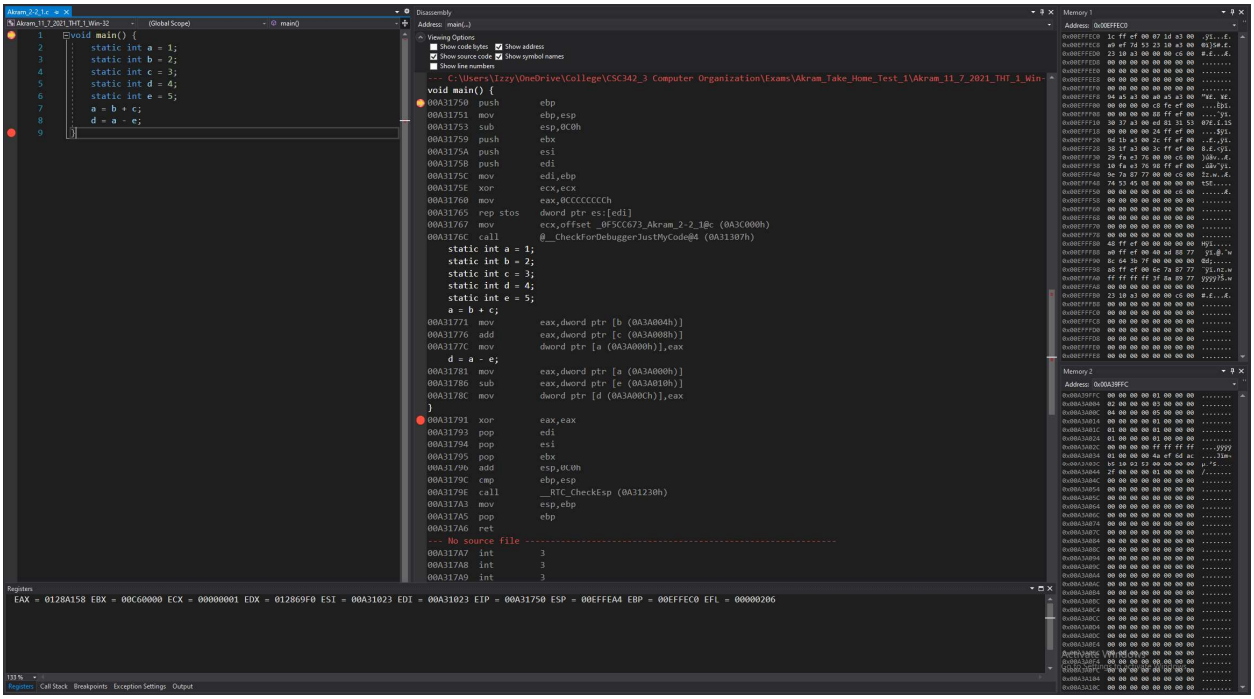

Figure 43: Debugger on Visual Studios (Before Line 1)

Disassembly: this window shows that the code starts at line 1, at main. No instructions have been performed yet to the registers or memories.

Registers: shows the value that's stored in each register, that the stack pointer, which is 0x00EFFEA4, is stored.

Memory: memory window 1 displays the values around the stack pointer (all random).

Memory window 2 displays the value of the static variables stored into the addresses.

Variable a is stored in address: 0x00A3A000 containing the value: 01 00 00 00
Variable b is stored in address: 0x00A3A004 containing the value: 02 00 00 00
Variable c is stored in address: 0x00A3A008 containing the value: 03 00 00 00
Variable d is stored in address: 0x00A3A00c containing the value: 04 00 00 00
Variable e is stored in address: 0x00A3A010 containing the value: 05 00 00 00

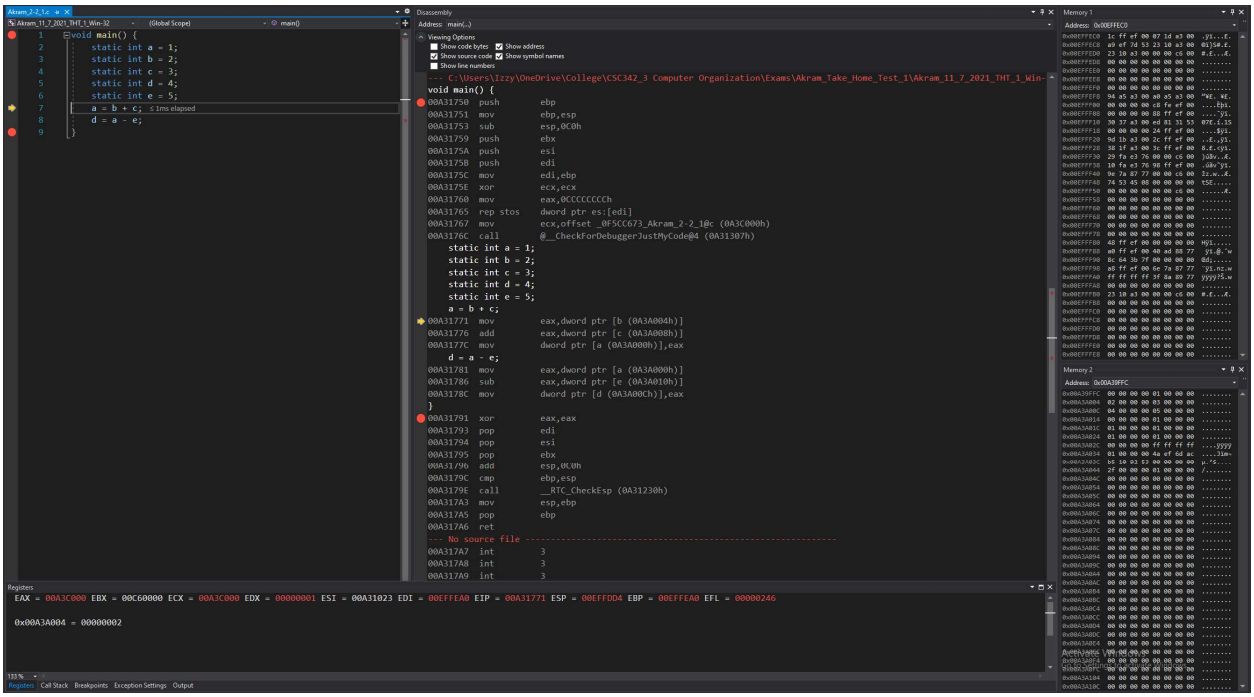

Figure 44: Debugger on Visual Studios (Lines 1-6)

Disassembly: this window shows that the code runs lines 1 to 6 and that the registers and memory are set to have the appropriate values to start the program's execution.

Registers: shows the value that's stored in each register. Some values have been changed from before, since this is setting up the registers for the program to run.

Memory: memory window 1 displays the values around the stack pointer (some now initialized).

Memory window 2 displays the value of the static variables stored into the addresses, remaining the same since lines 1 to 6 didn't store anything into the addresses or change any values.

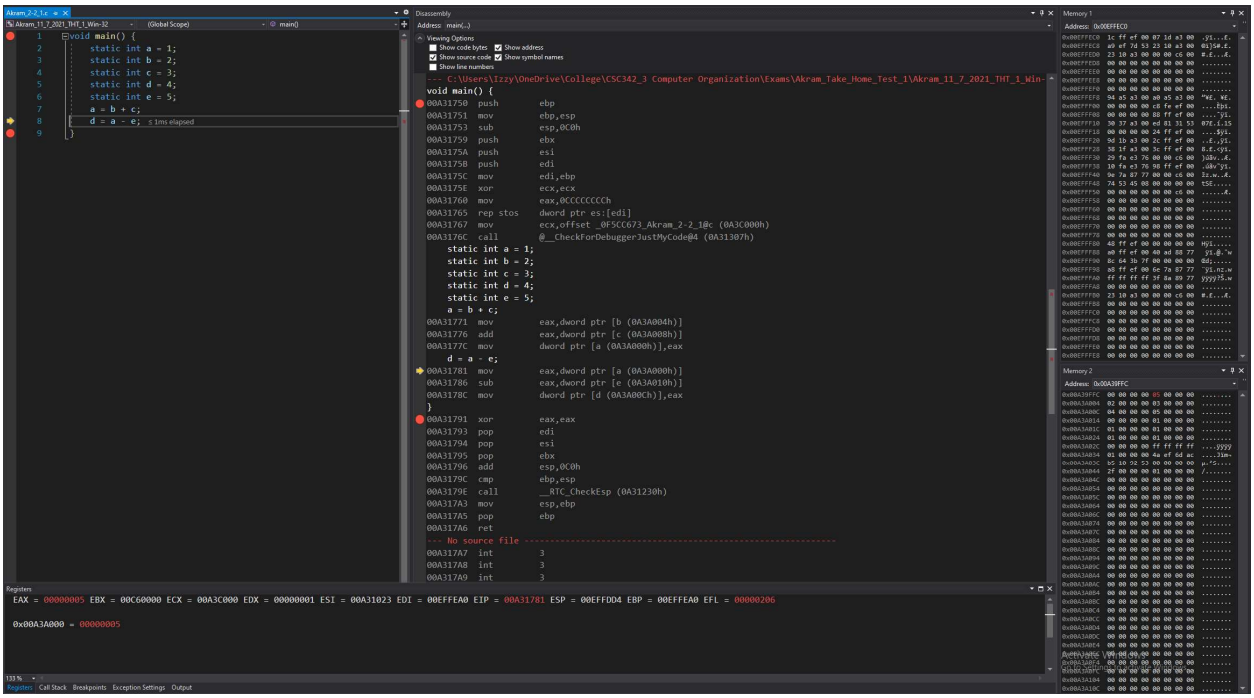

Figure 44: Debugger on Visual Studios (Lines 7)

Disassembly: this window shows that the code runs lines 7 and that the registers and memory moved a value into a register so it can be added to with another value and the result is to be saved into memory.

Registers: shows the value that's stored in each register:

Value of b (stored in memory), is moved to the register,

Addition of that the value in register with value of c (stored in memory), is performed and then stored to the same register.

Value in that register is moved into the memory, stored in the address of a and overwrites the value stored at a.

Memory: memory window 1 displays the values around the stack pointer, remains the same since line 7 didn't store anything into them or change any values.

Memory window 2 displays the value of the static variables stored into the addresses:

Variable a is stored in address: 0x00A3A000, now containing the value: 05 00 00 00.

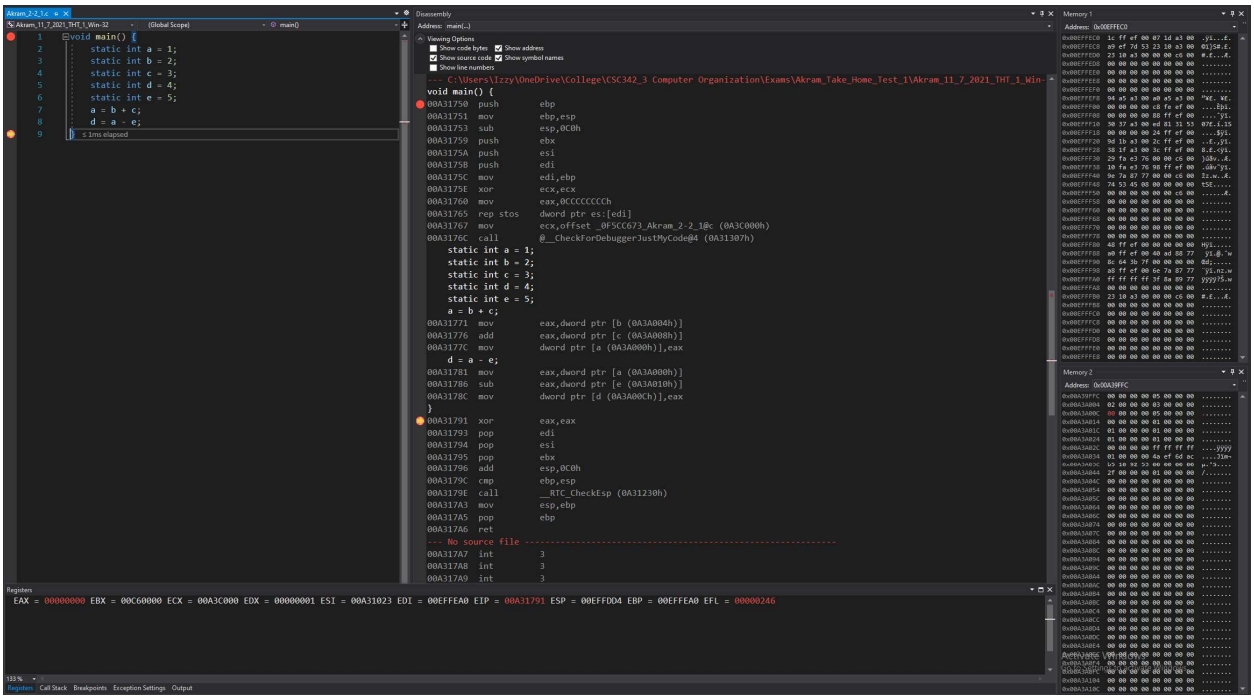

Figure 44: Debugger on Visual Studios (Lines 8-9)

Disassembly: this window shows that the code runs lines 8 to 9 and that a value was moved into a register so it can be subtracted with another value, the result would be saved into memory.

Registers: shows the value that's stored in each register:

Value of a (stored in memory), is moved to the register,

Subtraction of that the value in register with value of e (stored in memory), is performed and then stored to the same register.

Value in that register is moved into the memory, stored in the address of d and overwrites the value stored at d.

Memory: memory window 1 displays the values around the stack pointer, remains the same since line 8 to 9 didn't store anything into them or change any values.

Memory window 2 displays the value of the static variables stored into the addresses:

Variable d is stored in address: 0x00A3A00C, now containing the value: 00 00 00 00.

2-2_2.c:

The code written for 2-2_2 in C is shown in Fig 47. It uses five static variables: f, g, h, i, and j. Using these five variables; arithmetic is performed and stored in the variables.

In our case, we're adding the values of g and h, and we're also adding the values of i and j. We're then subtracting those results and storing this new result into f.

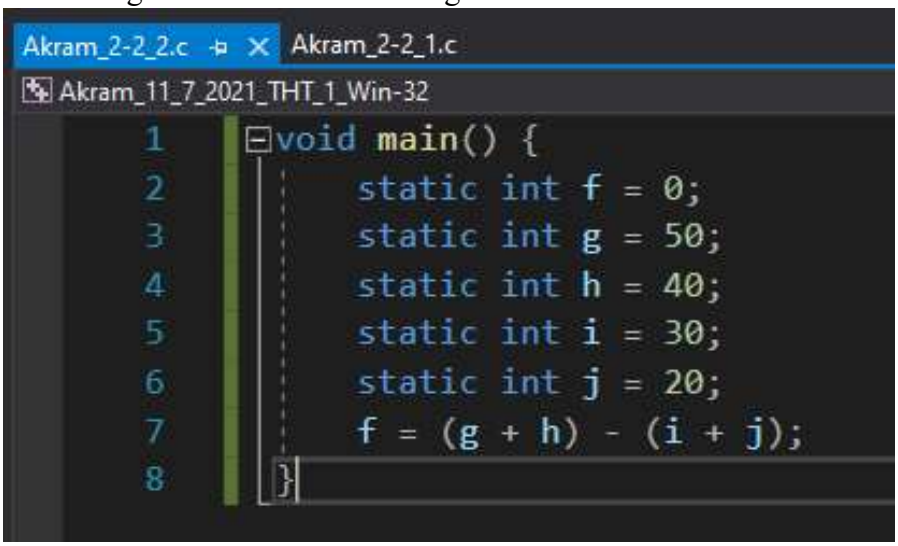

Fig 47. Akram_2-2_2.c executed in VS Win-32 bit

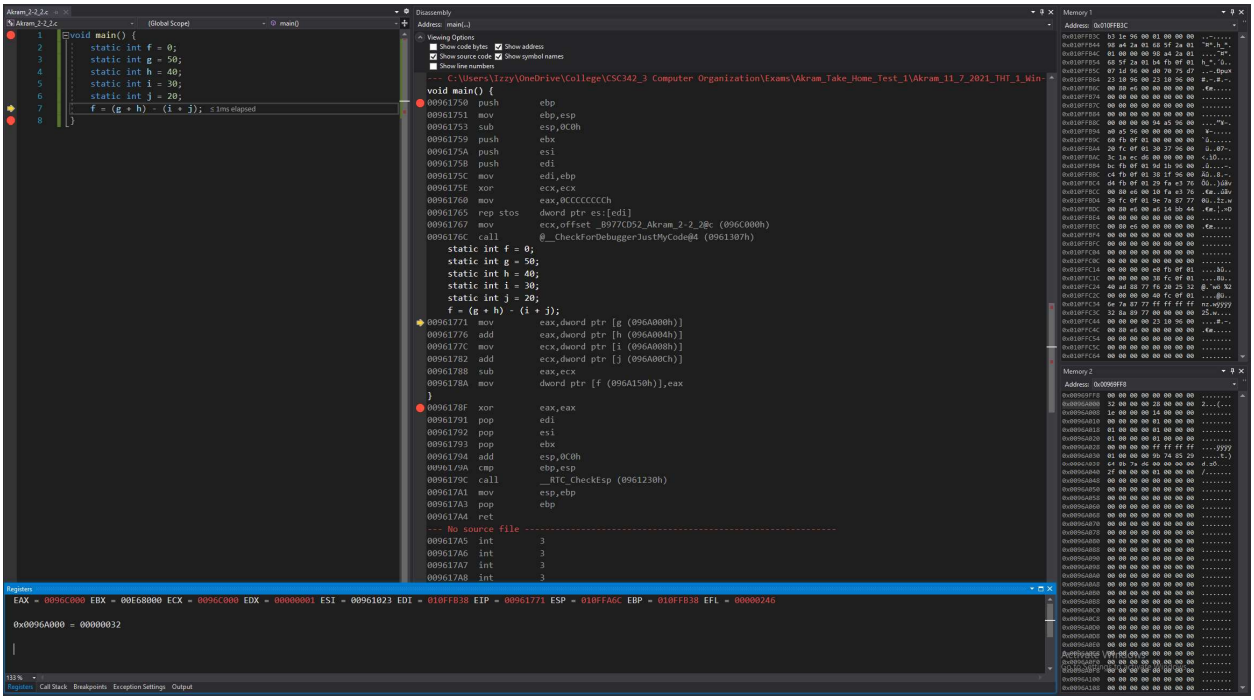

Figure 48: Debugger on Visual Studios (Before Line 1)

Disassembly: this window shows that the code starts at line 1, at main. No instructions have been performed yet to the registers or memories.

Registers: shows the value that's stored in each register, that the stack pointer, which is 0x010FFA6C, is stored.

Memory: memory window 1 displays the values around the stack pointer (all random).

Memory window 2 displays the value of the static variables stored into the addresses.

Variable f is stored in address: 0x00969FFC containing the value: 00 00 00 00
Variable g is stored in address: 0x0096A000 containing the value: 32 00 00 00
Variable h is stored in address: 0x0096A004 containing the value: 28 00 00 00
Variable i is stored in address: 0x0096A008 containing the value: 1e 00 00 00
Variable j is stored in address: 0x0096A00C containing the value: 14 00 00 00

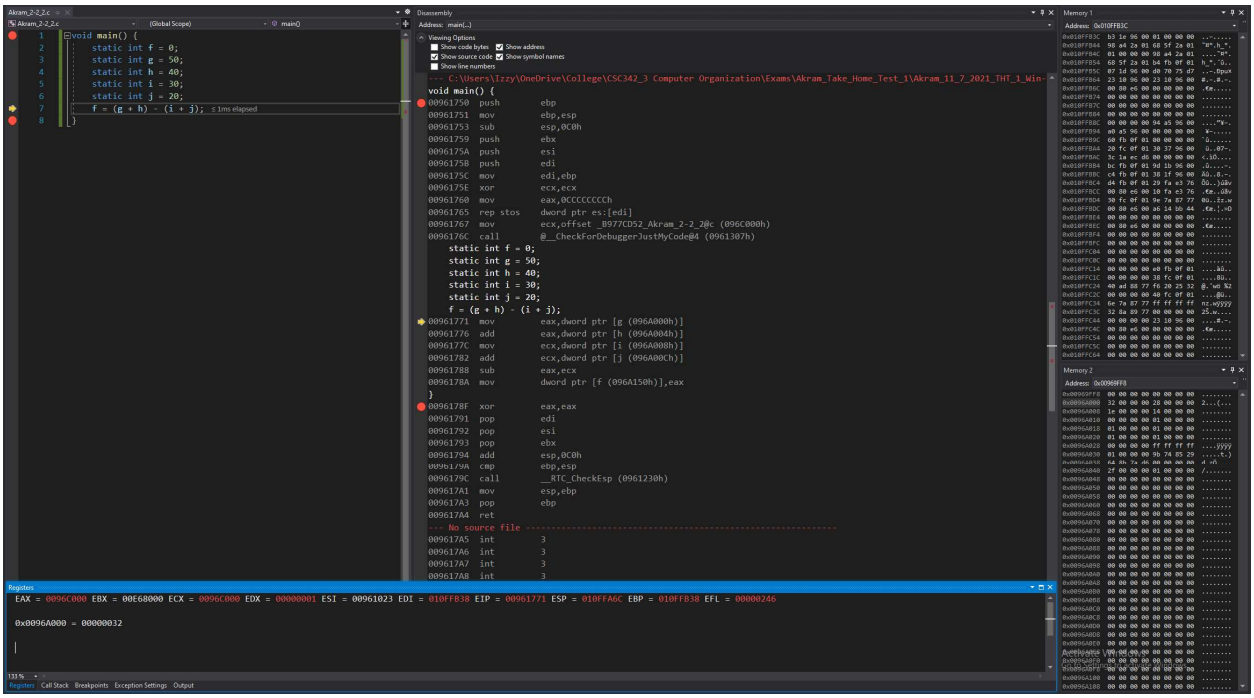

Figure 49: Debugger on Visual Studios (Lines 1-6)

Disassembly: this window shows that the code runs lines 1 to 6 and that the registers and memory are set to have the appropriate values to start the program's execution.

Registers: shows the value that's stored in each register. Some values have been changed from before, since this is setting up the registers for the program to run.

Memory: memory window 1 displays the values around the stack pointer (some now initialized).

Memory window 2 displays the value of the static variables stored into the addresses, remaining the same since lines 1 to 6 didn't store anything into the addresses or change any values.

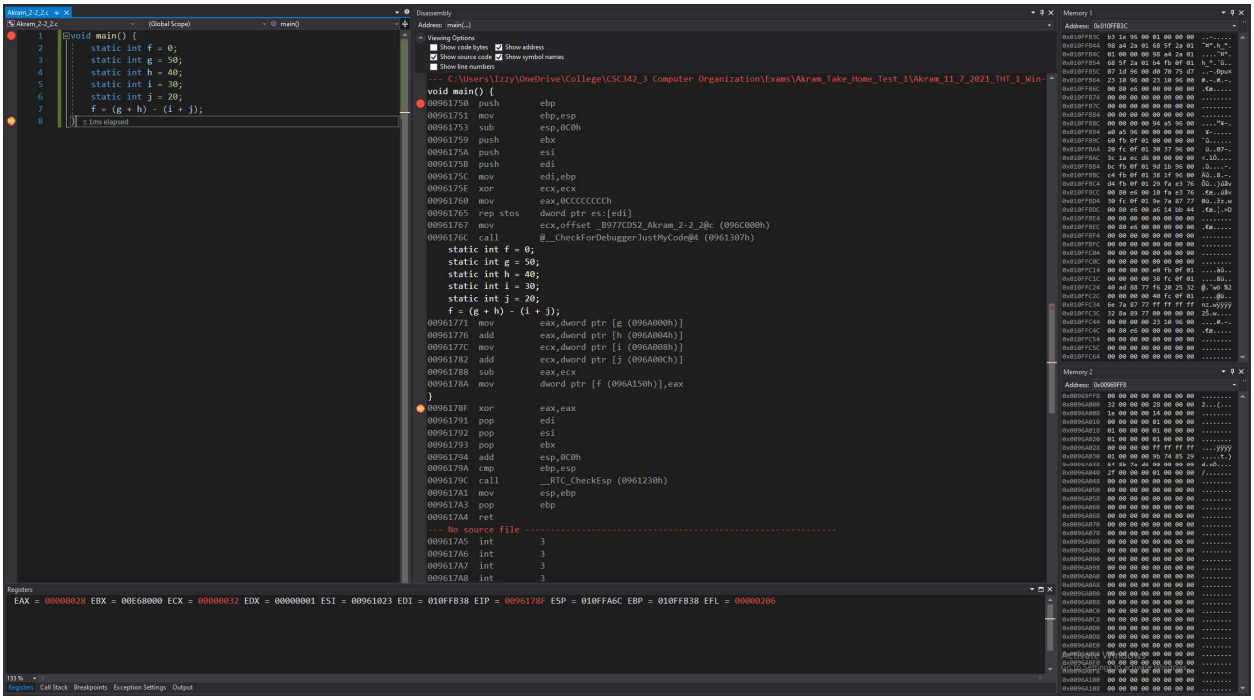

Figure 50: Debugger on Visual Studios (Lines 7-8)

Disassembly: this window shows that the code runs lines 7 to 8 and that the registers and memory moved a value into a register so it can be added and subtracted with another value and the result is to be saved into memory.

Registers: shows the value that's stored in each register:

Value of g (stored in memory), is moved to the register,

Addition of that the value in register with value of h (stored in memory), is performed and then stored to the same register.

Value of i (stored in memory), is moved to the register,

Addition of that the value in register with value of j (stored in memory), is performed and then stored to the same register.

Values in the two registers are subtracted is moved into the memory, stored in the address of f, and overwrites the value stored at f.

Memory: memory window 1 displays the values around the stack pointer, remains the same since line 7 to 8 didn't store anything into them or change any values.

Memory window 2 displays the value of the static variables stored into the addresses:

Variable f is stored in address: 0x00969FFC, now containing the value: 28 00 00 00.

2-3_1.c:

The code written for 2-3_1 in C is shown in Fig 51. It uses three static variables: g, h, and A. Using these three variables; an array is created, and addition is executed through its usage.

In our case, 55 is stored in the 8th index of array A and is added to h. The result from that addition is stored into g.

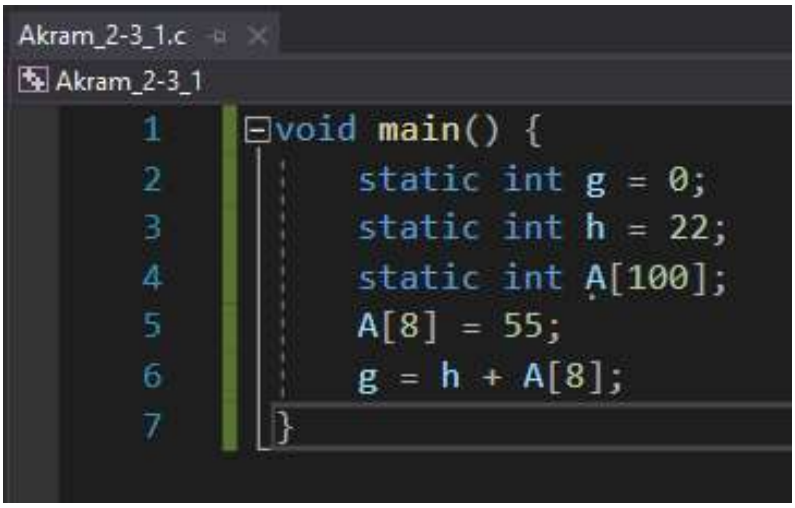

Fig 51. Akram_2-3_1.c executed in VS Win-32 bit

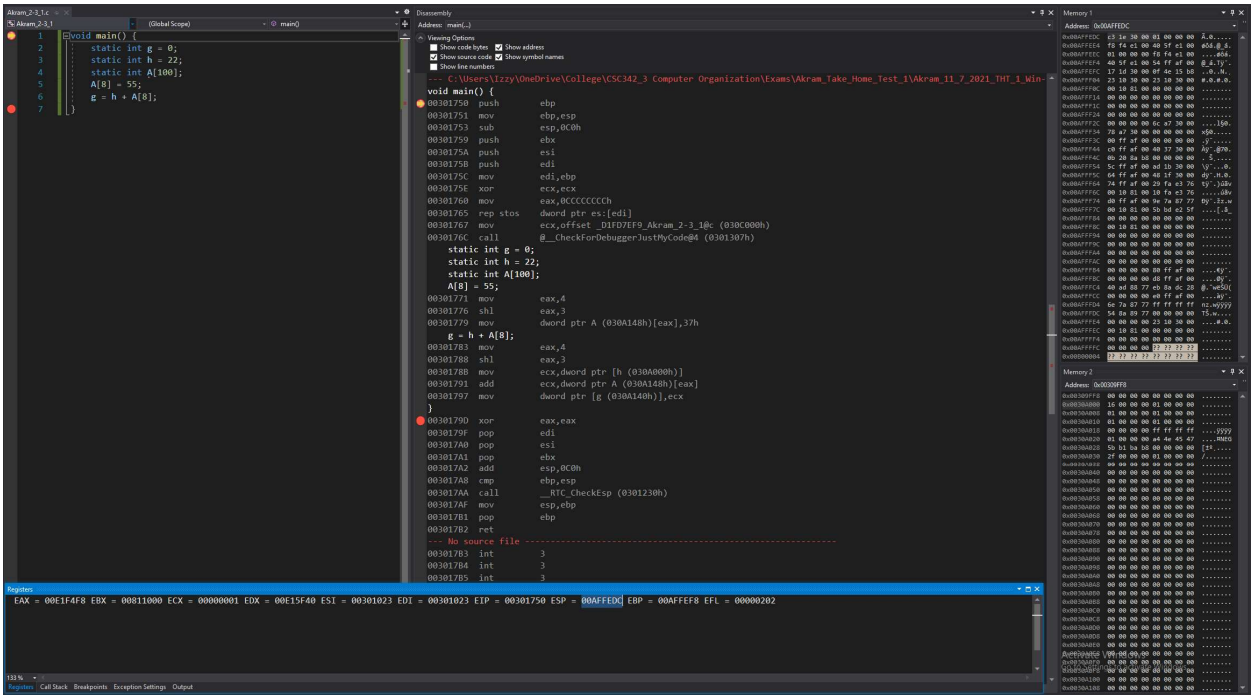

Figure 52: Debugger on Visual Studios (Before Line 1)

Disassembly: this window shows that the code starts at line 1, at main. No instructions have been performed yet to the registers or memories.

Registers: shows the value that's stored in each register, that the stack pointer, which is 0x00AFFEDC, is stored.

Memory: memory window 1 displays the values around the stack pointer (all random).

Memory window 2 displays the value of the static variables stored into the addresses.

Variable g is stored in address: 0x0030A140 containing the value: 00 00 00 00
Variable h is stored in address: 0x0030A000 containing the value: 16 00 00 00
Variable A is stored in address: 0x0030A148 containing the value: 00 00 00 00

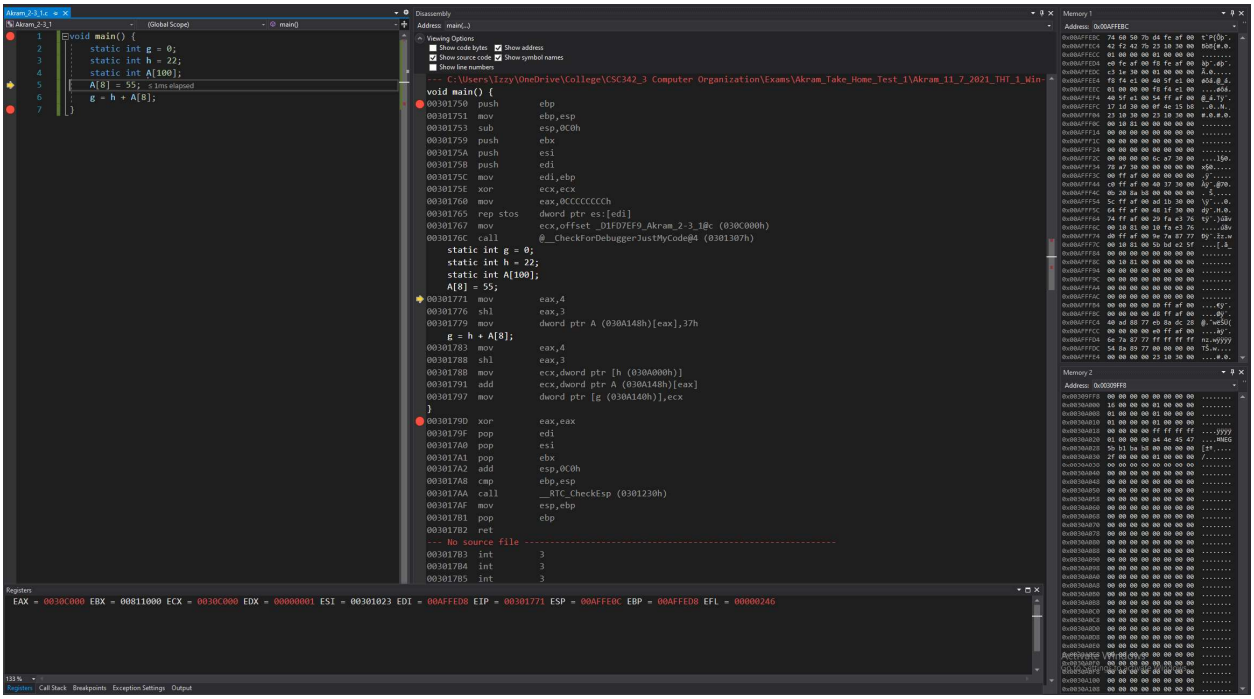

Figure 52: Debugger on Visual Studios (Lines 1-4)

Disassembly: this window shows that the code runs lines 1 to 4 and that the registers and memory are set to have the appropriate values to start the program's execution.

Registers: shows the value that's stored in each register. Some values have been changed from before, since this is setting up the registers for the program to run.

Memory: memory window 1 displays the values around the stack pointer (some now initialized).

Memory window 2 displays the value of the static variables stored into the addresses, remaining the same since lines 1 to 4 didn't store anything into the addresses or change any values.

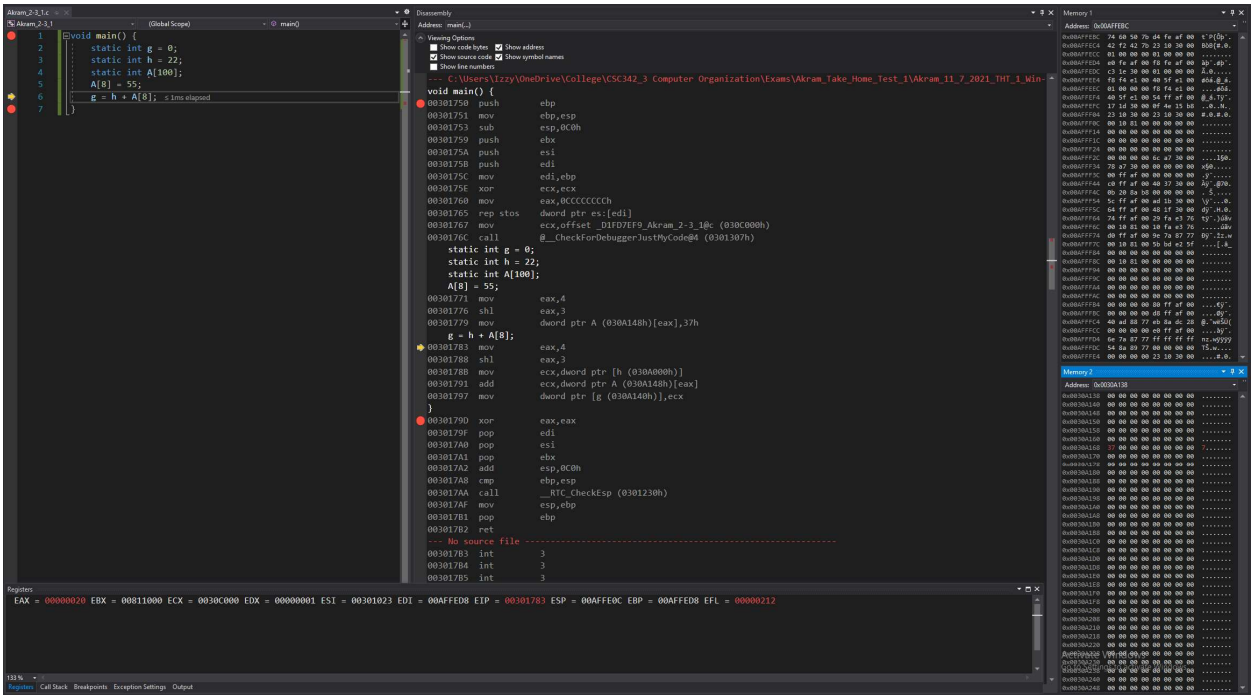

Figure 52: Debugger on Visual Studios (Line 5)

Disassembly: this window shows that the code runs lines 5 and that the value 4 is set into a register and then shifted to the left by 3 so the result would net us the 8th index.

Using the address of the array and the 8th index' value, we can calculate the address of the 8th index and store the value of 55 into that part of memory.

Registers: shows the value that's stored in each register:

Register EAX set to 00000004,
Then it is shifted by 3, becoming 00000020 = 4*8 = 8th index.

Memory: memory window 1 displays the values around the stack pointer, remains the same since line 5 didn't store anything into them or change any values.

Memory window 2 displays the values stored in array A.
Variable A's 8th index is stored in address: 0x0030A168, now containing the value: 37 00 00 00.

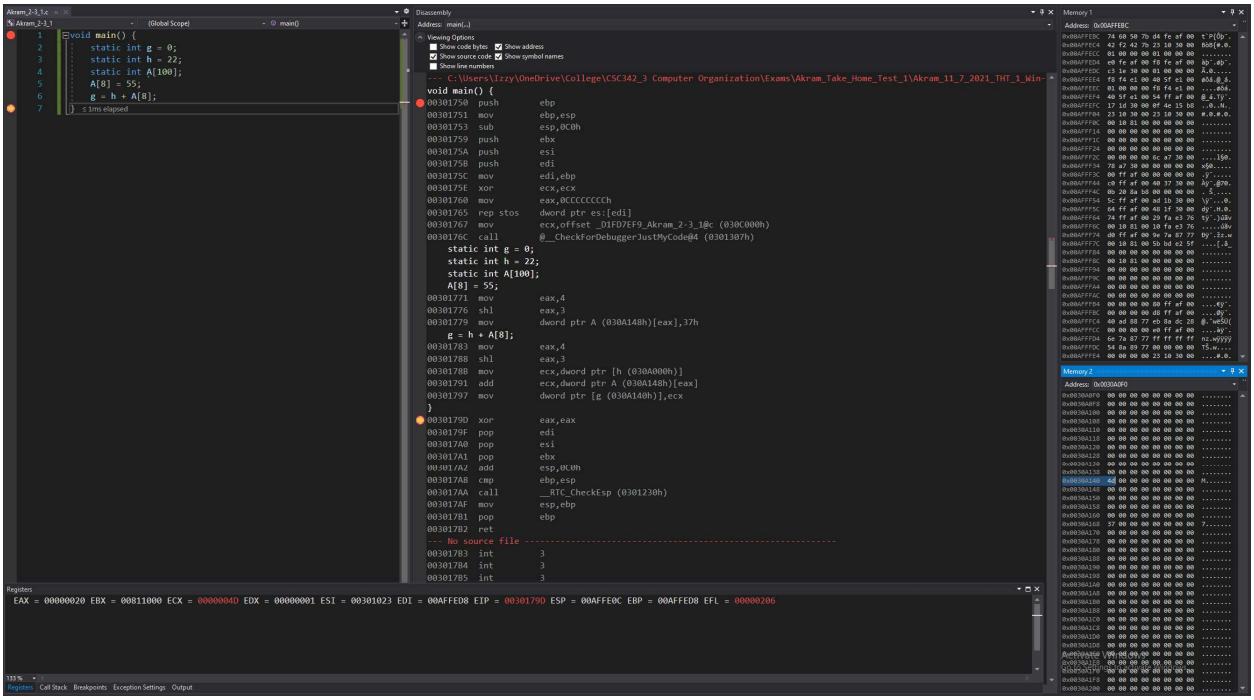

Figure 52: Debugger on Visual Studios (Line 6-7)

Disassembly: this window shows that the code runs lines 6 to 7 and that the value 4 is set into a register and then shifted to the left by 3 so the result would net us the 8th index.

Value of h (stored in memory), is moved to the register.

Addition of that the value in register with value of 8th index of the array (stored in memory), is performed and then stored to the same register.

Value in that register is moved into the memory, stored in the address of g and overwrites the value stored at g.

Registers: shows the value that's stored in each register:

Register EAX set to 00000004,
Then it is shifted by 3, becoming 00000020 = 4*8 = 8th index.
Register ECX set to value of h,
Then it is added by the 8th index of the array, and result is stored in ECX.

Memory: memory window 1 displays the values around the stack pointer, remains the same since line 6 to 7 didn't store anything into them or change any values.

Memory window 2 displays the value of the static variables stored into the addresses:

Variable g is stored in address: 0x0030A140, now containing the value: 4d 00 00 00.

2-3_2.c:

The code written for 2-3_2 in C is shown in Fig 56. It uses two static variables: h and A. Using these two variables; an array is created, and addition is executed through its usage.

In our case, 200 is stored in the 8th index of A and is added to h. The result from that addition is stored into the 12th index of A.

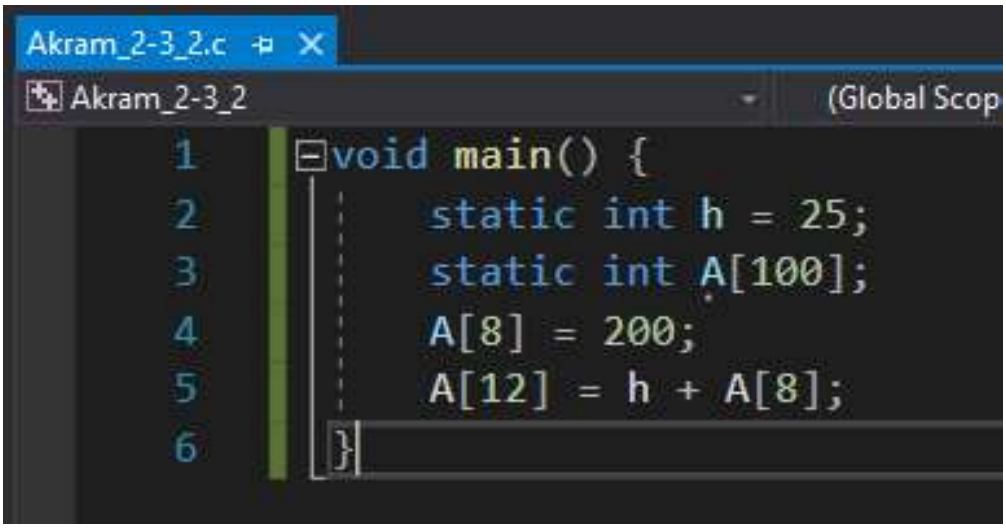

Fig 56. Akram_2-3_2.c executed in VS Win-32 bit

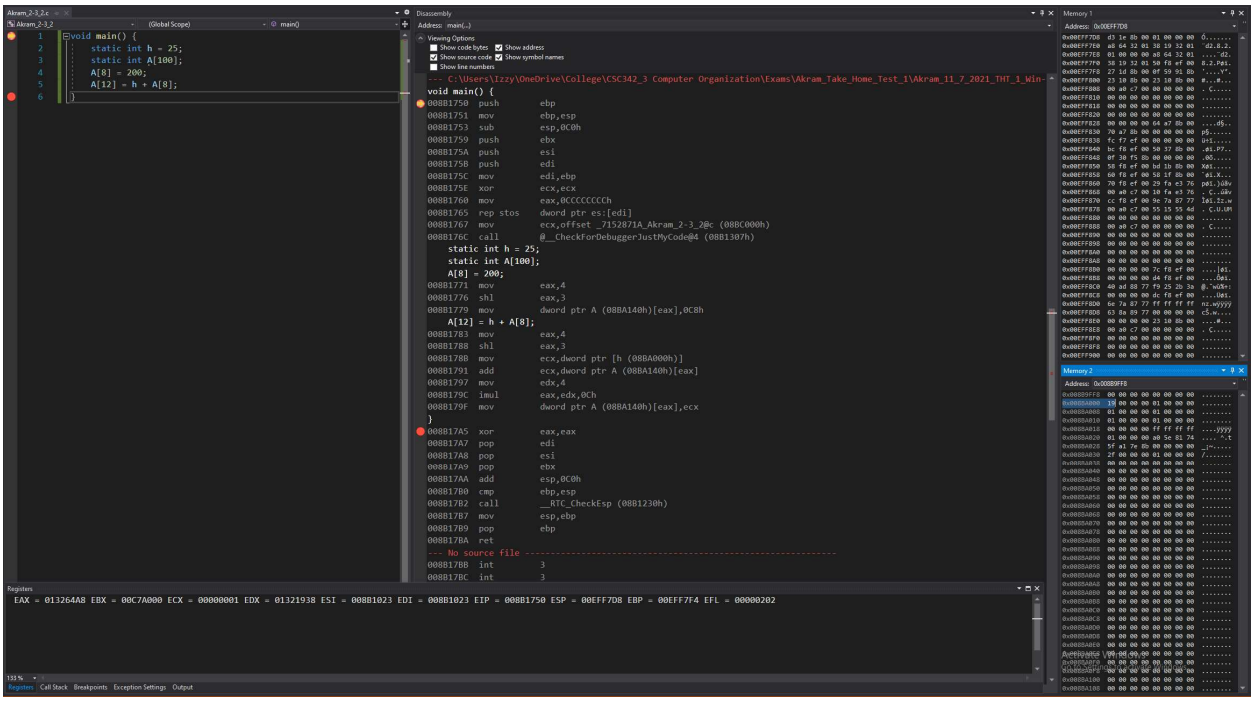

Figure 57: Debugger on Visual Studios (Before Line 1)

Disassembly: this window shows that the code starts at line 1, at main. No instructions have been performed yet to the registers or memories.

Registers: shows the value that's stored in each register, that the stack pointer, which is 0x00EFF708, is stored.

Memory: memory window 1 displays the values around the stack pointer (all random).

Memory window 2 displays the value of the static variables stored into the addresses.

Variable h is stored in address: 0x008BA000 containing the value: 19 00 00 00
Variable A is stored in address: 0x008BA140 containing the value: 00 00 00 00

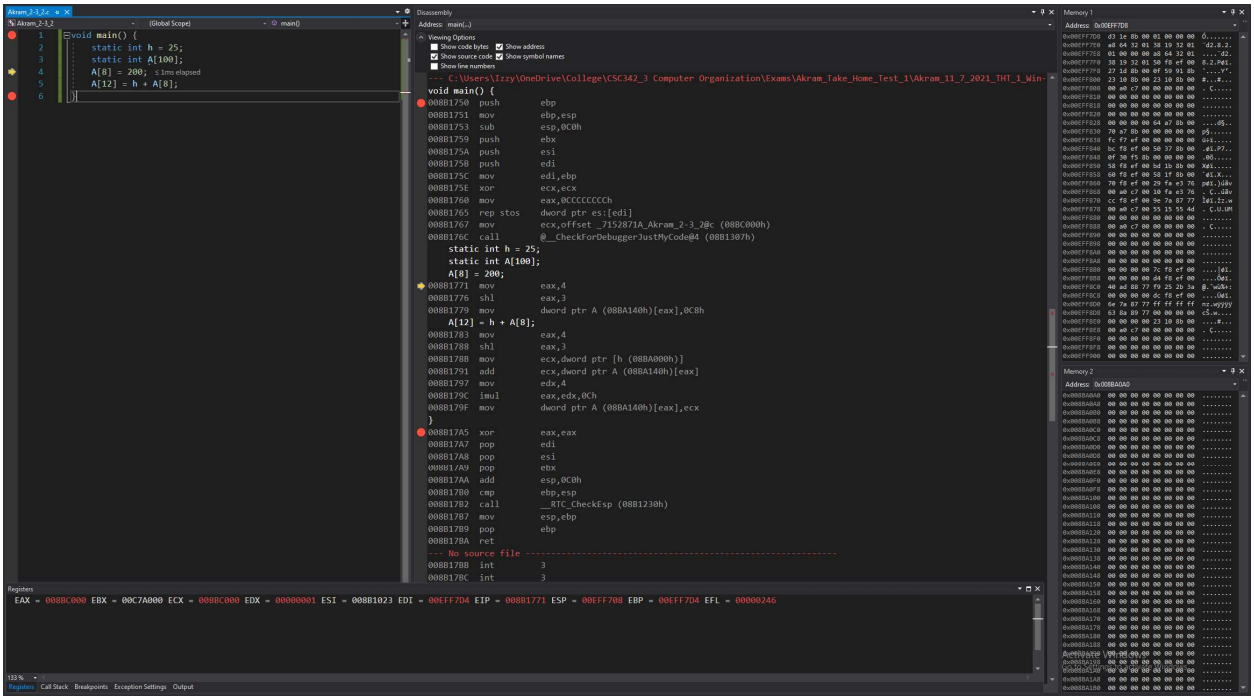

Figure 58. Debugger on Visual Studios (Lines 1-3)

Disassembly: this window shows that the code runs lines 1 to 3 and that the registers and memory are set to have the appropriate values to start the program's execution.

Registers: shows the value that's stored in each register. Some values have been changed from before, since this is setting up the registers for the program to run.

Memory: memory window 1 displays the values around the stack pointer (some now initialized).

Memory window 2 displays the value of the static variables stored into the addresses, remaining the same since lines 1 to 3 didn't store anything into the addresses or change any values.

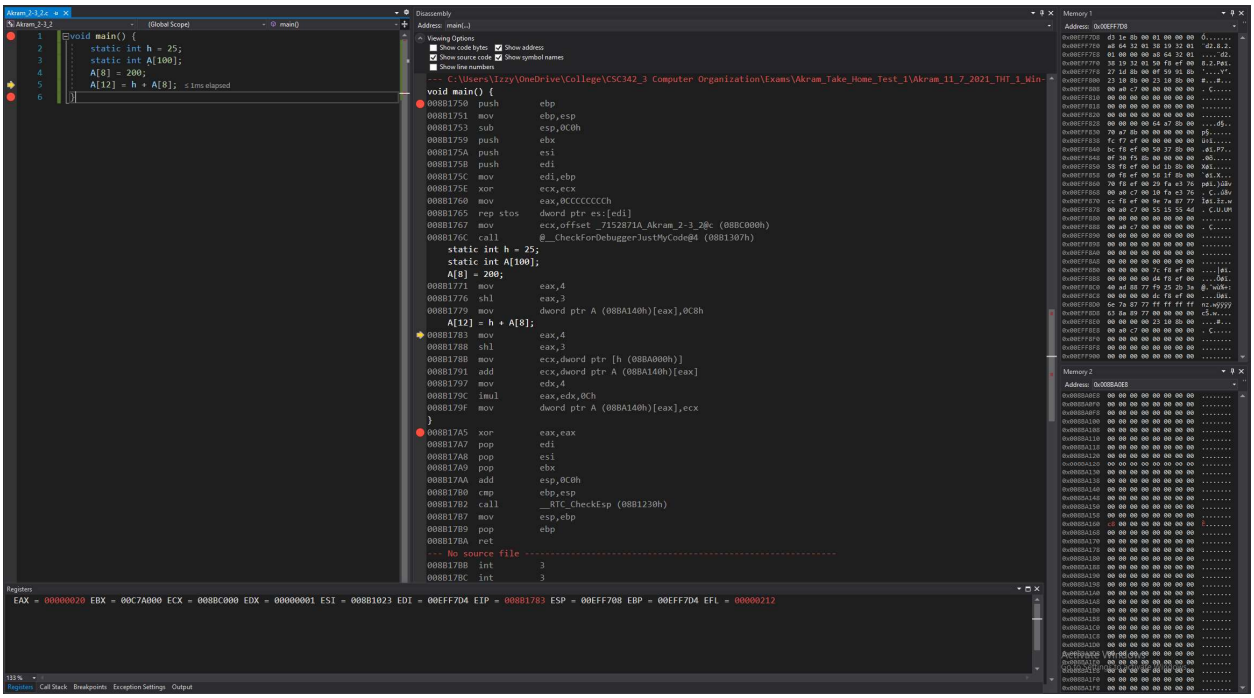

Figure 59. Debugger on Visual Studios (Line 4)

Disassembly: this window shows that the code runs line 5 and that the value 4 is set into a register and then shifted to the left by 3 so the result would net us the 8th index.

Using the address of the array and the 8th index' value, we can calculate the address of the 8th index and store the value of 200 into that part of memory.

Registers: shows the value that's stored in each register:

Register EAX set to 00000004,
Then it is shifted by 3, becoming 00000020 = 4*8 = 8th index.

Memory: memory window 1 displays the values around the stack pointer, remains the same since line 5 didn't store anything into them or change any values.

Memory window 2 displays the values stored in array A.
Variable A's 8th index is stored in address: 0x008BA160, now containing the value: c8 00 00 00.

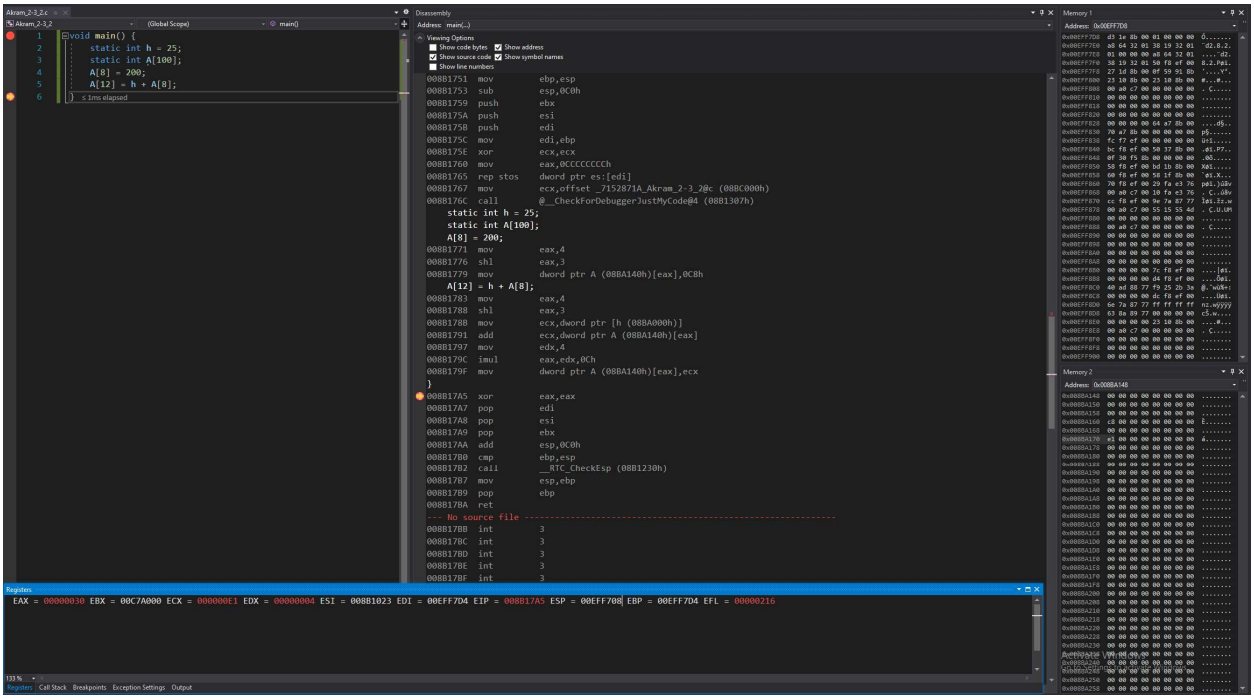

Figure 60. Debugger on Visual Studios (Lines 5 to 6)

Disassembly: this window shows that the code runs lines 5 to 6 and that the value 4 is set into a register and then shifted to the left by 3 so the result would net us the 8th index.

Value of h (stored in memory), is moved to the register.

Addition of that the value in register with value of 8th index of the array (stored in memory), is performed and then stored to the same register.

Value of 4 (stored in memory), is set to another register.

Perform an operation to net us the 12th index.

Using the address of the array and the 12th index' value, we can calculate the address of the 12th index and store the value of the addition into that part of memory.

Registers: shows the value that's stored in each register:

Register EAX set to 00000004,
Then it is shifted by 3, becoming 00000020 = 4*8 = 8th index.
Register ECX set to value of h,
Then it is added by the 8th index of the array, and result is stored in ECX.
Register EDX set to 00000004,
EDX is used to perform an operation and the result is saved in EAX.

Memory: memory window 1 displays the values around the stack pointer, remains the same since line 5 to 6 didn't store anything into them or change any values.

Memory window 2 displays the value of the static variables stored into the addresses:

Variable 12th index of A is stored in address: 0x008BA170, now containing the value: e1 00 00 00.

2-5_1.c:

The code written for 2-5_1 in MIPS is shown in Fig 61. It uses two static variables: h and A. Using these two variables; an array A is created, and addition is executed through its usage.

In our case, 13 is stored in the 300th index of the array and is added to h. The result from that addition is stored back into the 300th index.

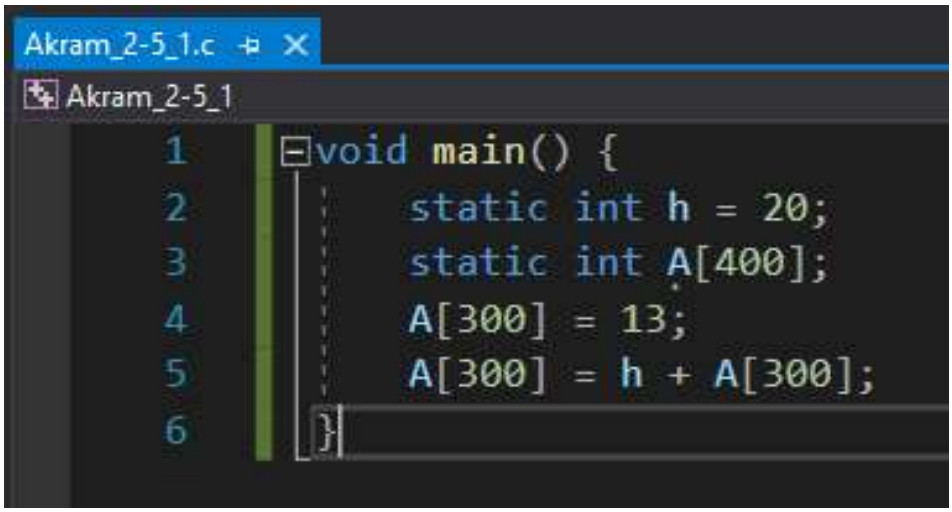

Fig 61. Akram_2-5_1.c executed in VS Win-32 bit

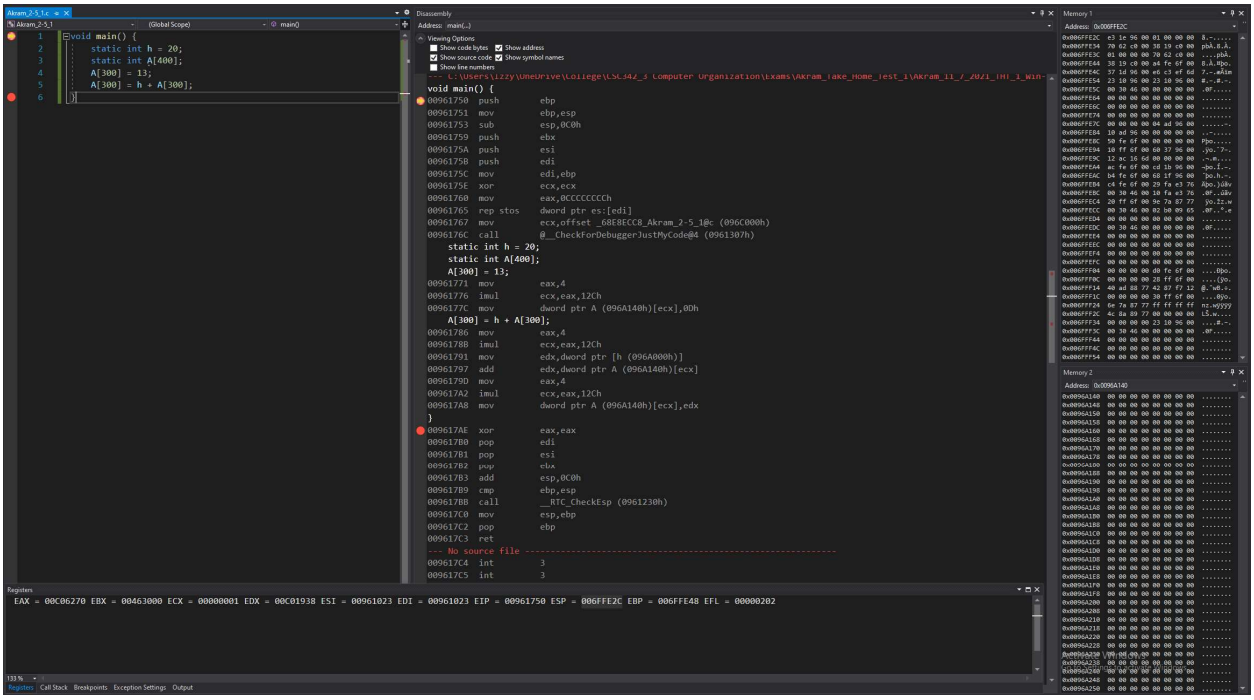

Figure 62: Debugger on Visual Studios (Before Line 1)

Disassembly: this window shows that the code starts at line 1, at main. No instructions have been performed yet to the registers or memories.

Registers: shows the value that's stored in each register, that the stack pointer, which is 0x006FFE2C, is stored.

Memory: memory window 1 displays the values around the stack pointer (all random).

Memory window 2 displays the value of the static variables stored into the addresses.

Variable h is stored in address: 0x0096A000 containing the value: 14 00 00 00
Variable A is stored in address: 0x0096A140 containing the value: 00 00 00 00

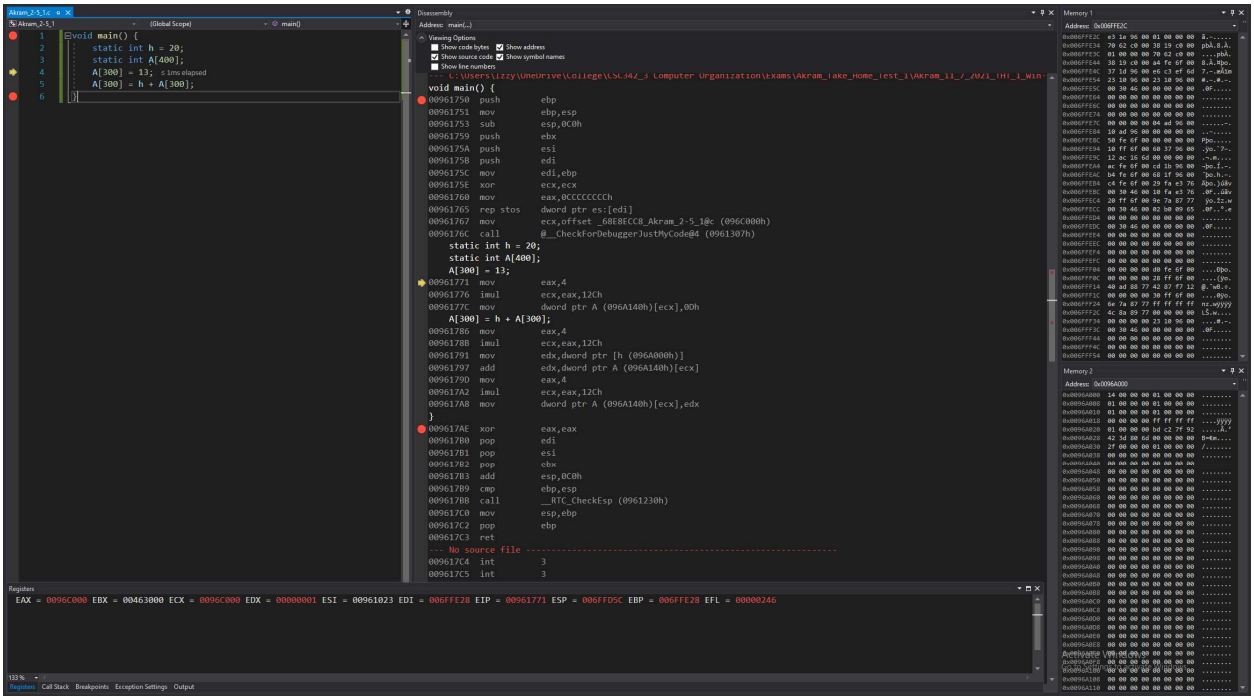

Figure 63: Debugger on Visual Studios (Before Lines 1-3)

Disassembly: this window shows that the code runs lines 1 to 3 and that the registers and memory are set to have the appropriate values to start the program's execution.

Registers: shows the value that's stored in each register. Some values have been changed from before, since this is setting up the registers for the program to run.

Memory: memory window 1 displays the values around the stack pointer (some now initialized).

Memory window 2 displays the value of the static variables stored into the addresses, remaining the same since lines 1 to 3 didn't store anything into the addresses or change any values.

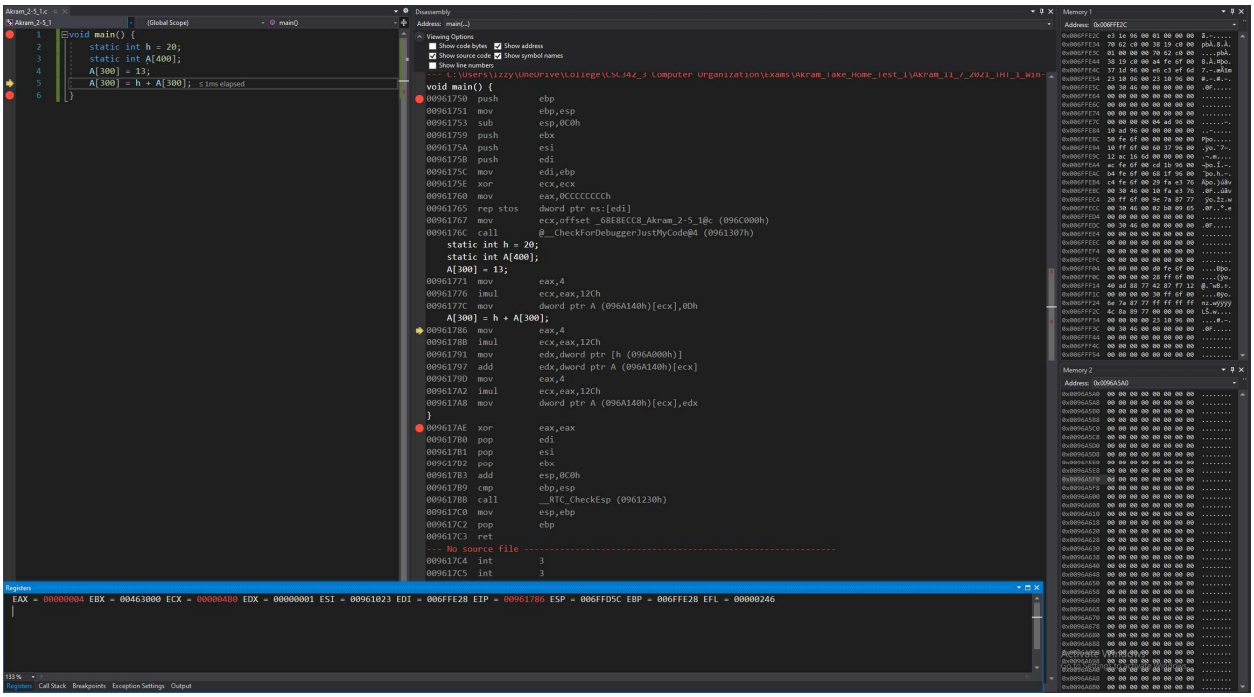

Figure 64: Debugger on Visual Studios (Before Line 5)

Disassembly: this window shows that the code runs line 5 and that the value 4 is set into a register and then an operation is performed to net us the 300th index.

Using the address of the array and the 300th index' value, we can calculate the address of the 300th index and store the value of 13 into that part of memory.

Registers: shows the value that's stored in each register:

Register EAX set to 00000004,
Operation is performed on it to net us the register: 000004B0,
giving us the 300th index (saved in ECX)

Memory: memory window 1 displays the values around the stack pointer, remains the same since line 4 didn't store anything into them or change any values.

Memory window 2 displays the values stored in array A.
Variable A's 300th index is stored in address: 0x0096A5F0, now containing the value: 0d 00 00 00.

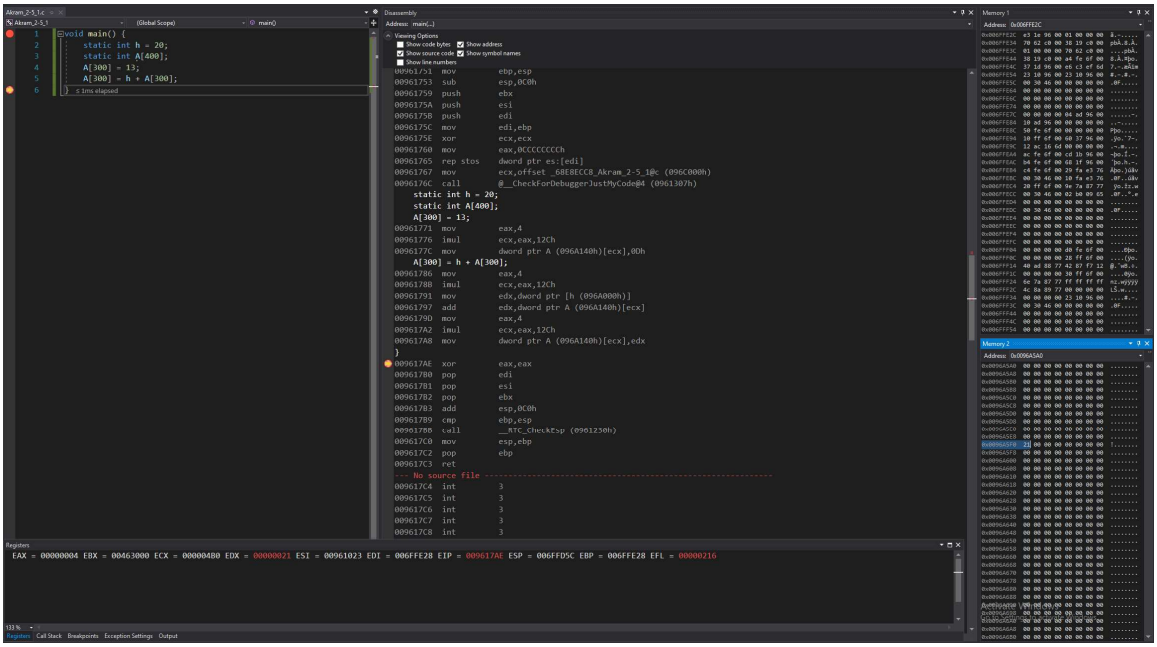

Figure 65: Debugger on Visual Studios (Before Lines 5-6)

Disassembly: this window shows that the code runs lines 5 to 6 and that the value 4 is set into a register and then an operation is performed to net us the 300th index.

Value of h (stored in memory), is moved to the register.

Addition of that the value in register with value of 300th index of the array (stored in memory), is performed and then stored to the same register.

Value of 4 (stored in memory), is set to another register.

Perform an operation to net us the 300th index.

Using the address of the array and the 300th index' value, we can calculate the address of the 300th index and store the value of the addition into that part of memory.

Registers: shows the value that's stored in each register:

Register EAX set to 00000004,

Then it is used to perform an operation and the result is stored in ECX, netting us the 300th index,

Register EDX set to value of h,

Then it is added by the 300th index of the array, and result is stored in EDX.

Register EAX set to 00000004,

EAX is used to perform an operation and the result is saved in ECX.

Memory: memory window 1 displays the values around the stack pointer, remains the same since line 5 to 6 didn't store anything into them or change any values.

Memory window 2 displays the value of the static variables stored into the addresses:

Variable 300th index of A is stored in address: 0x00D295F0, now containing the value: 21 00 00 00.

2-6_1.c:

The code written for 2-6_1 in C is shown in Fig 66. It stores values into the registers and uses these values to perform three bitwise operations: shift a register to the left, AND (compares two registers), and OR (also compares two registers).

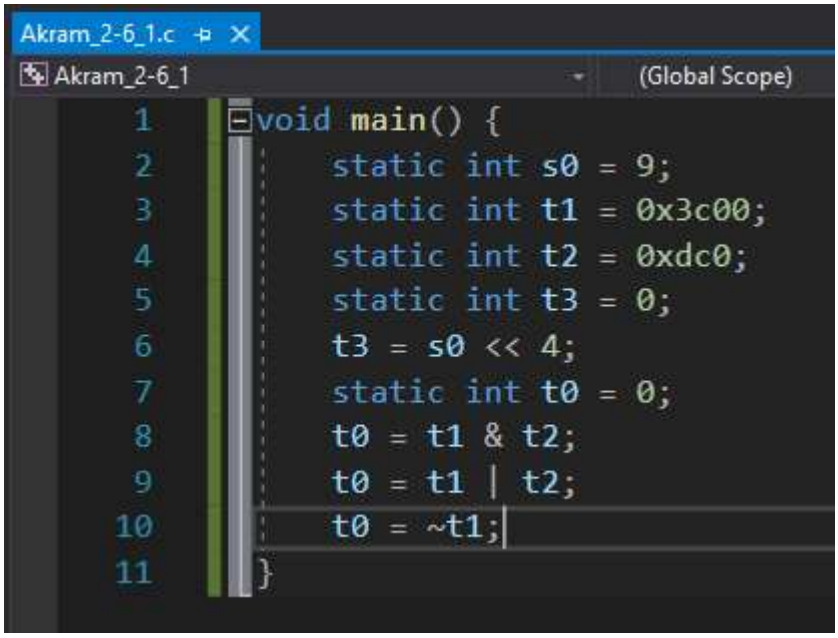

Fig 66. Akram_2-6_1.c executed in VS Win-32 bit

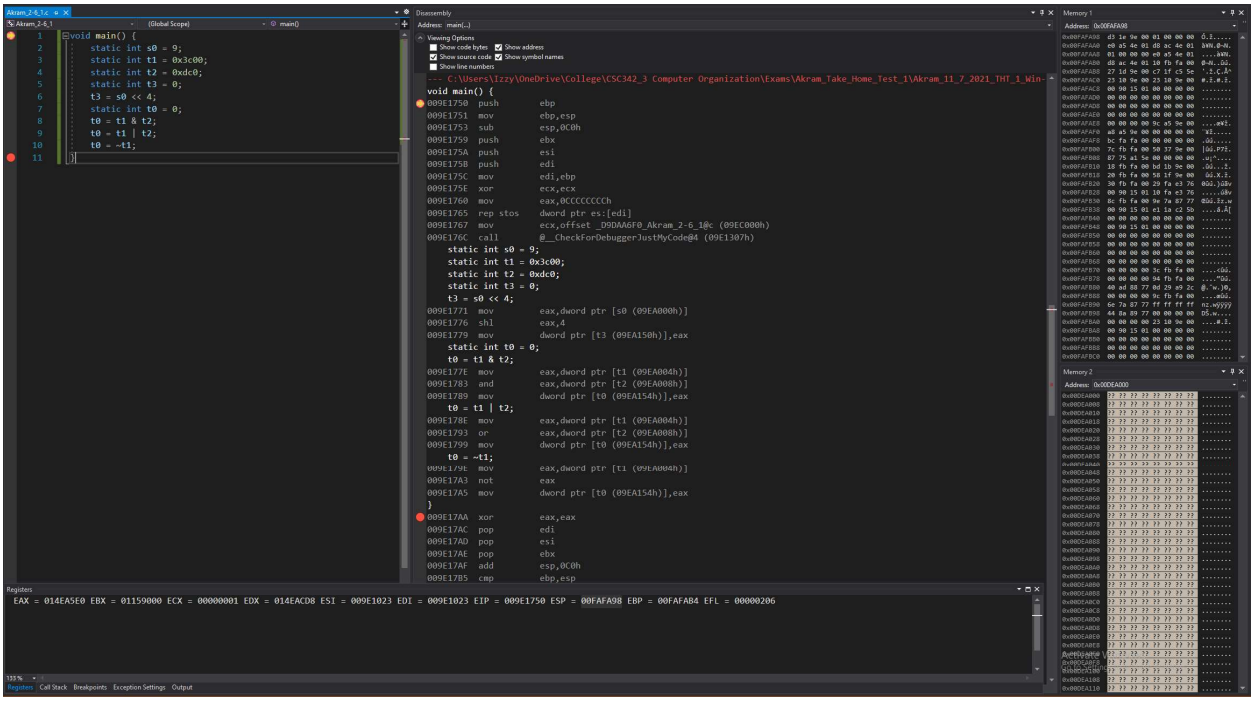

Figure 67: Debugger on Visual Studios (Before Line 1)

Disassembly: this window shows that the code starts at line 1, at main. No instructions have been performed yet to the registers or memories.

Registers: shows the value that's stored in each register, that the stack pointer, which is 0x00FAF9C8, is stored.

Memory: memory window 1 displays the values around the stack pointer (all random).

Memory window 2 displays the value of the static variables stored into the addresses.

Variable s0 is stored in address: 0x009EA000 containing the value: 09 00 00 00
Variable t1 is stored in address: 0x009EA004 containing the value: 00 3c 00 00
Variable t2 is stored in address: 0x009EA008 containing the value: c0 0d 00 00
Variable t3 is stored in address: 0x009EA150 containing the value: 00 00 00 00
Variable t0 is stored in address: 0x009EA154 containing the value: 00 00 00 00

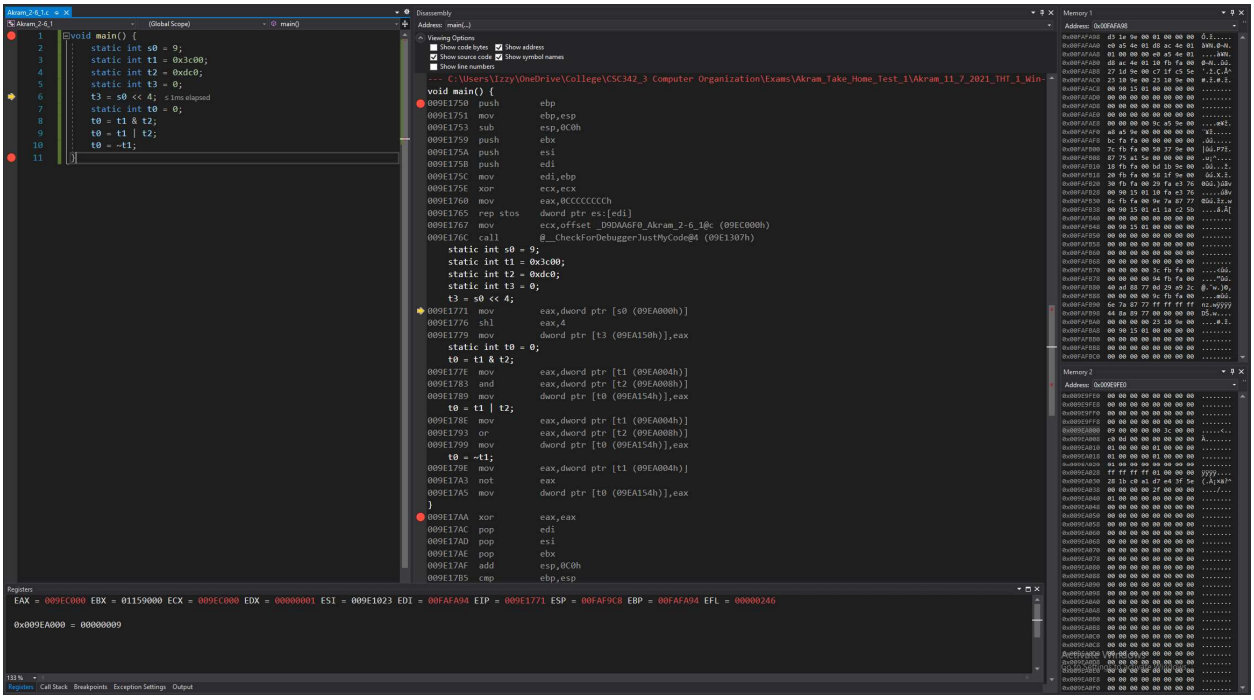

Figure 68: Debugger on Visual Studios (Lines 1-5)

Disassembly: this window shows that the code runs lines 1 to 5 and that the registers and memory are set to have the appropriate values to start the program's execution.

Registers: shows the value that's stored in each register. Some values have been changed from before, since this is setting up the registers for the program to run.

Memory: memory window 1 displays the values around the stack pointer (some now initialized).

Memory window 2 displays the value of the static variables stored into the addresses, remaining the same since lines 1 to 5 didn't store anything into the addresses or change any values.

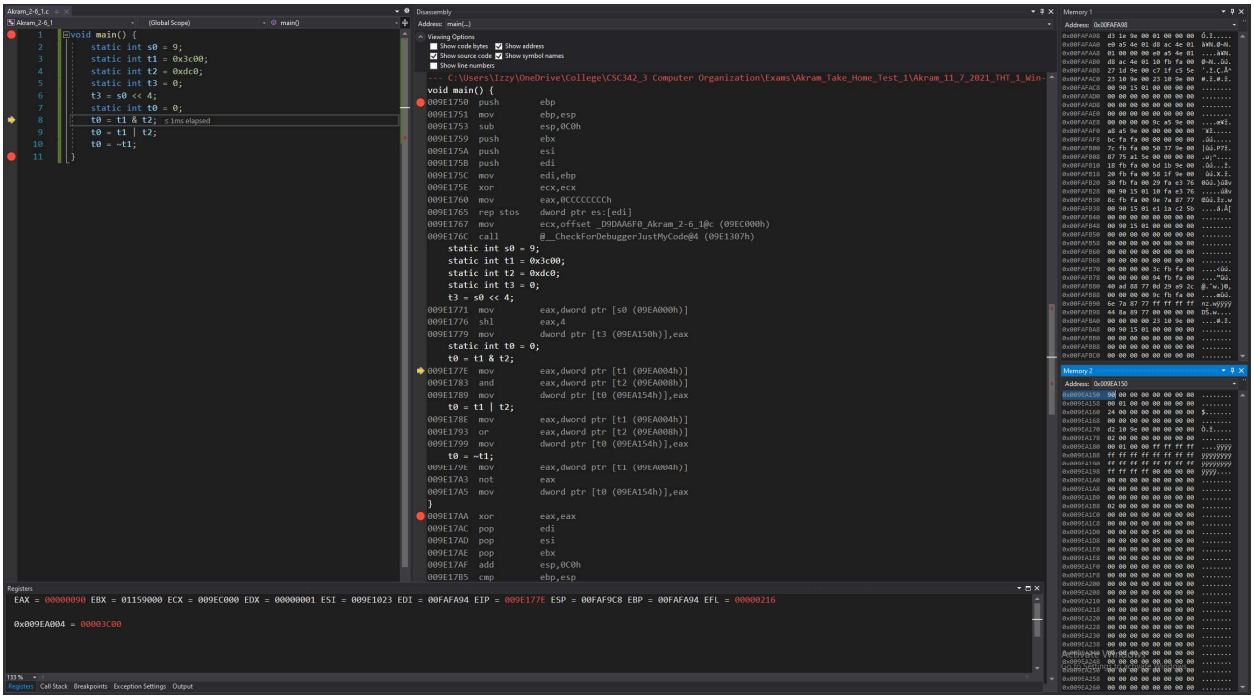

Figure 69: Debugger on Visual Studios (Lines 6-7)

Disassembly: this window shows that the code runs lines 6 to 7 and that the value 4 is moved into a register so it can be shifted to the left by 4 and the result is to be stored in memory.

Using the address of the array and the 300th index' value, we can calculate the address of the 300th index and store the value of 13 into that part of memory.

Registers: shows the value that's stored in each register:

Value of s0 (stored in memory), is moved to the register.
Value is shifted to the left by 4 and then stored to the same register.
Value in that register is moved into the memory, stored in the address of t3 and overwrites the value stored at t3.

Memory: memory window 1 displays the values around the stack pointer, remains the same since line 8 didn't store anything into them or change any values.

Memory window 2 displays the value of the static variables stored into the addresses,

Variable t3 is stored in address: 0x009EA150, now containing the value: 90 00 00 00.

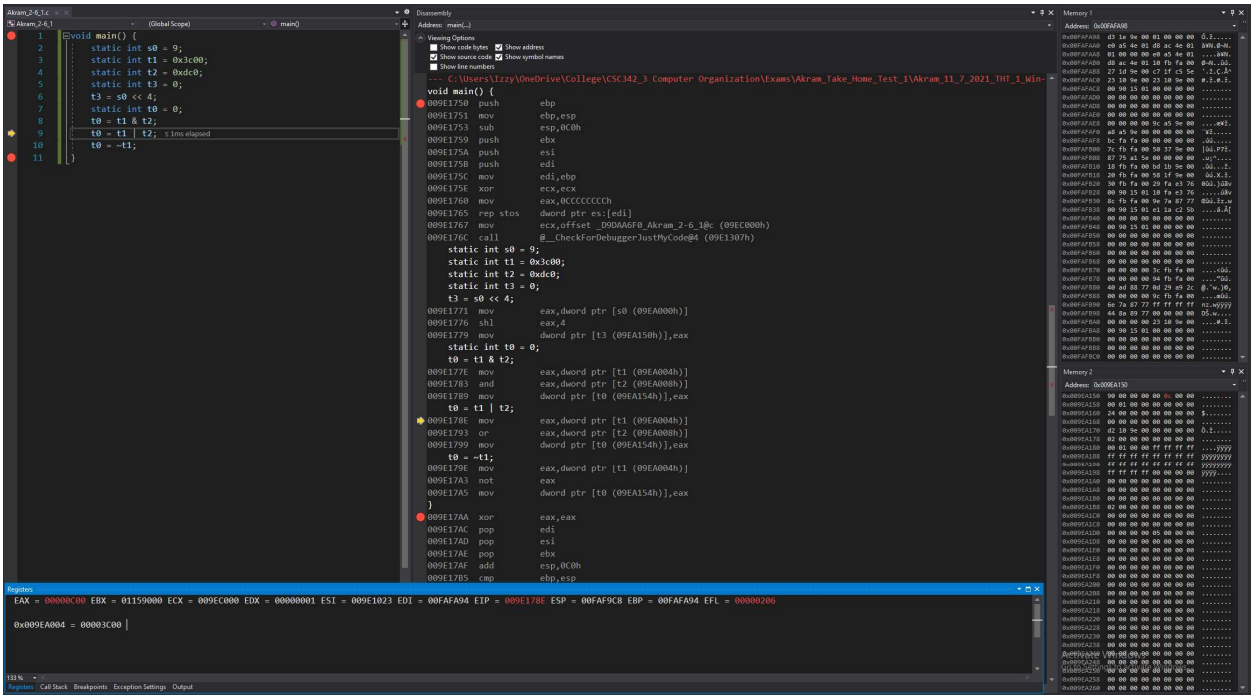

Figure 70: Debugger on Visual Studios (Line 8)

Disassembly: this window shows that the code runs line 8 and that a value is moved into a register so it can be compared to another value using bitwise operation. The result is to be stored in memory.

Registers: shows the value that's stored in each register:

Value of t1 (stored in memory), is moved to the register.

Then that value is compared with t2 using bitwise operation, result is stored to the same register.

Value in that register is moved into the memory, stored in the address of t0 and overwrites the value stored at t0.

Memory: memory window 1 displays the values around the stack pointer, remains the same since line 8 didn't store anything into them or change any values.

Memory window 2 displays the value of the static variables stored into the addresses:

Variable t0 is stored in address: 0x009EA155, now containing the value: 0c 00 00 00.

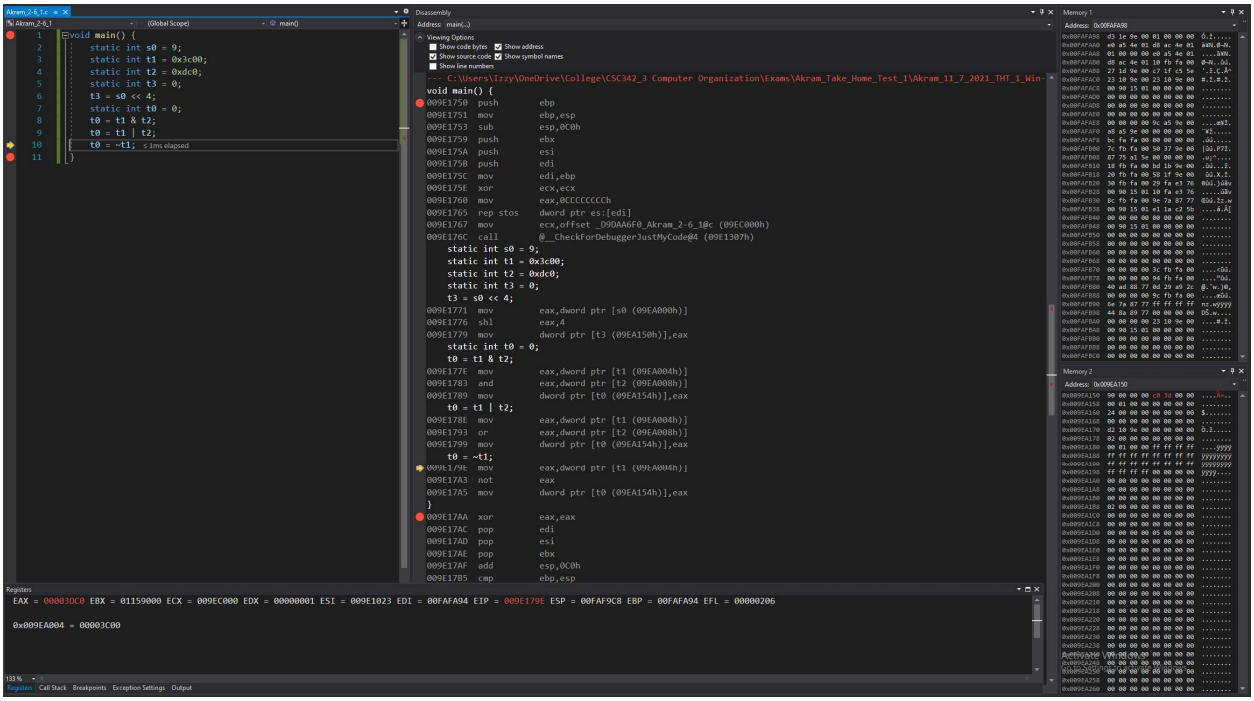

Figure 71: Debugger on Visual Studios (Line 9)

Disassembly: this window shows that the code runs line 9 and that a value is moved into a register so it can be compared to another value using bitwise operation. The result is to be stored in memory.

Registers: shows the value that's stored in each register:

Value of t1 (stored in memory), is moved to the register.

Then that value is compared with t2 using bitwise operation, result is stored to the same register.

Value in that register is moved into the memory, stored in the address of t0 and overwrites the value stored at t0.

Memory: memory window 1 displays the values around the stack pointer, remains the same since line 9 didn't store anything into them or change any values.

Memory window 2 displays the value of the static variables stored into the addresses:

Variable t0 is stored in address: 0x009EA155, now containing the value: c0 3d 00 00.

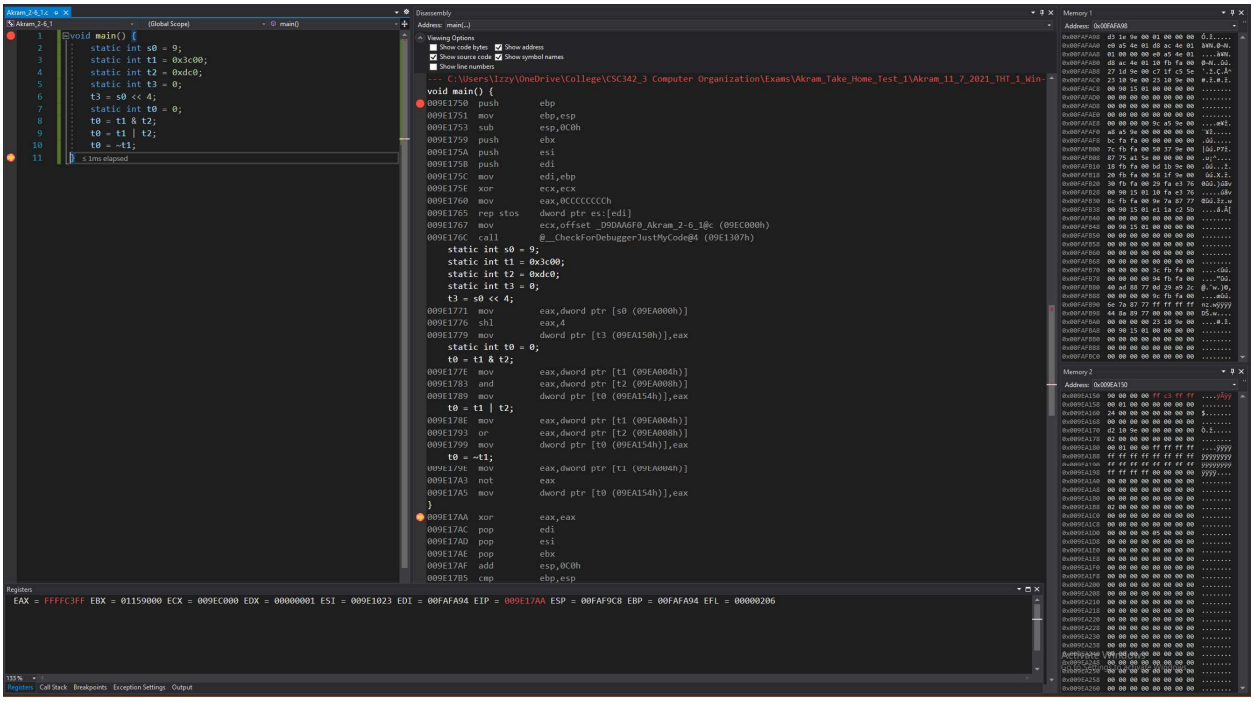

Figure 72: Debugger on Visual Studios (Lines 10 to 11)

Disassembly: this window shows that the code runs lines 10 to 11 and that a value is moved into a register so it can be compared to another value using bitwise operation. The result is to be stored in memory.

Registers: shows the value that's stored in each register:

Value of t1 (stored in memory), is moved to the register.

Then that value is compared with t2 using bitwise operation, result is stored to the same register.

Value in that register is moved into the memory, stored in the address of t0 and overwrites the value stored at t0.

Memory: memory window 1 displays the values around the stack pointer, remains the same since lines 10 to 11 didn't store anything into them or change any values.

Memory window 2 displays the value of the static variables stored into the addresses:

Variable t0 is stored in address: 0x009EA155, now containing the value: ff c3 ff ff.

2-7_1.c:

The code written for 2-7_1 in C is shown in Fig 73. It uses five static variables: f, g, h, i, and j. Using these five variables; conditions are set up and if the variables satisfy those conditions; arithmetic operations are performed.

If i and j, then g and h will be added, and the result will be stored in f,
else,
g and h will be subtracted, and the result will be stored in f.

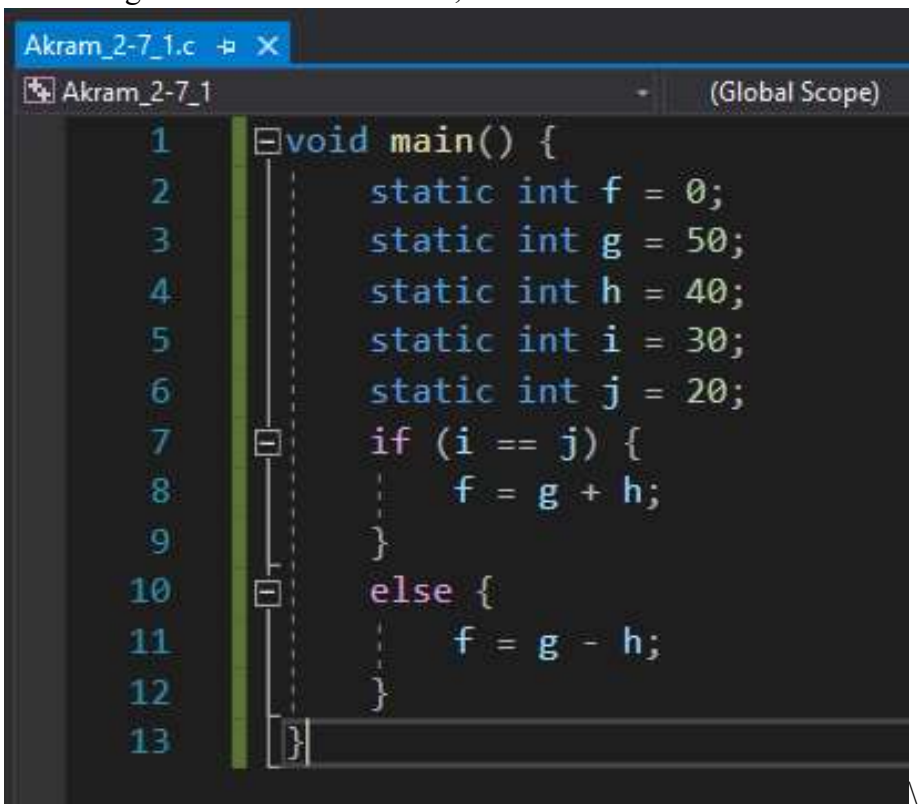

Fig 73. Akram_2-7_1.c executed in VS Win-32 bit

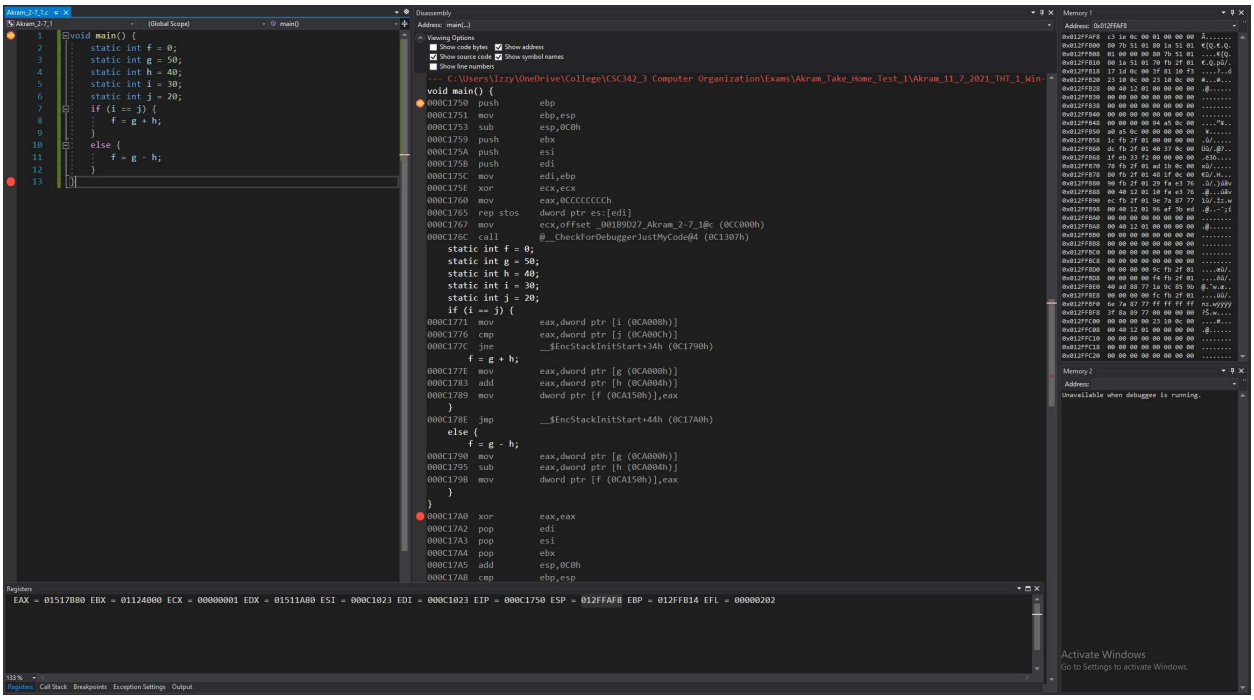

Figure 74: Debugger on Visual Studios (Before Line 1)

Disassembly: this window shows that the code starts at line 1, at main. No instructions have been performed yet to the registers or memories.

Registers: shows the value that's stored in each register, that the stack pointer, which is 0x012FFA28, is stored.

Memory: memory window 1 displays the values around the stack pointer (all random).

Memory window 2 displays the value of the static variables stored into the addresses.

Variable f is stored in address: 0x00C9FFC containing the value: 00 00 00 00
Variable g is stored in address: 0x000CA000 containing the value: 32 00 00 00
Variable h is stored in address: 0x000CA004 containing the value: 28 00 00 00
Variable i is stored in address: 0x000CA008 containing the value: 1e 00 00 00
Variable j is stored in address: 0x000CA00c containing the value: 14 00 00 00

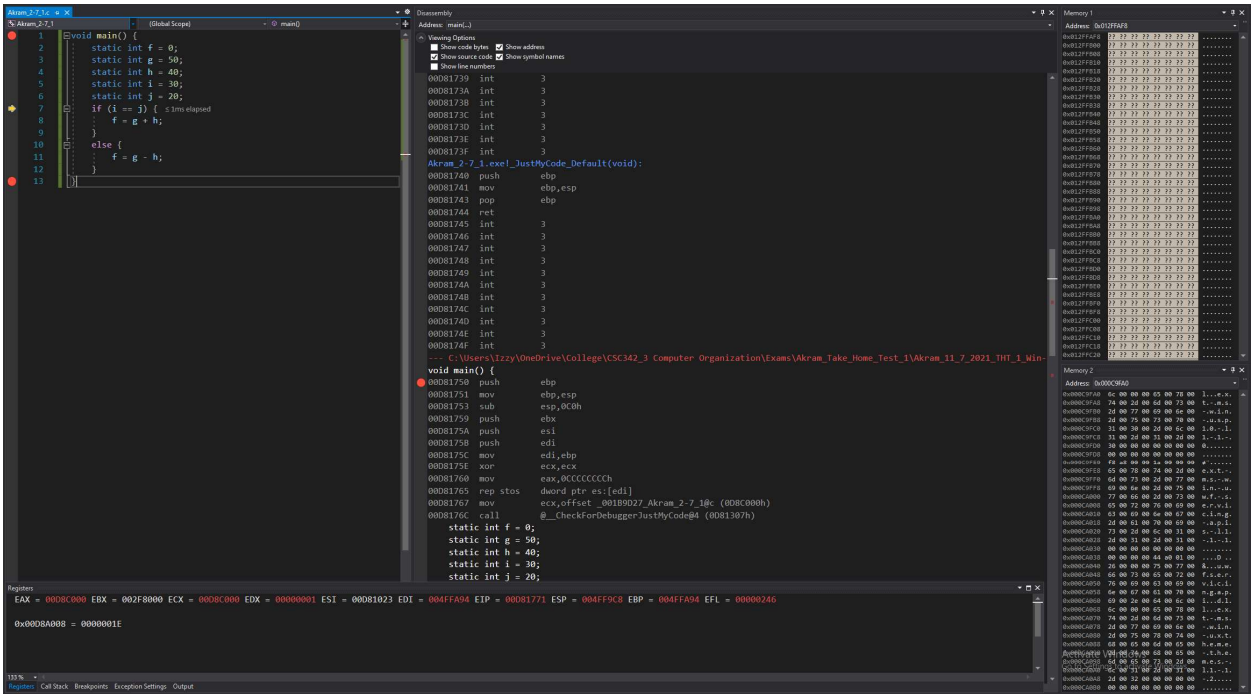

Figure 75: Debugger on Visual Studios (Lines 1 to 6)

Disassembly: this window shows that the code runs lines 1 to 6 and that the registers and memory are set to have the appropriate values to start the program's execution.

Registers: shows the value that's stored in each register. Some values have been changed from before, since this is setting up the registers for the program to run.

Memory: memory window 1 displays the values around the stack pointer (some now initialized).

Memory window 2 displays the value of the static variables stored into the addresses, remaining the same since lines 1 to 6 didn't store anything into the addresses or change any values.

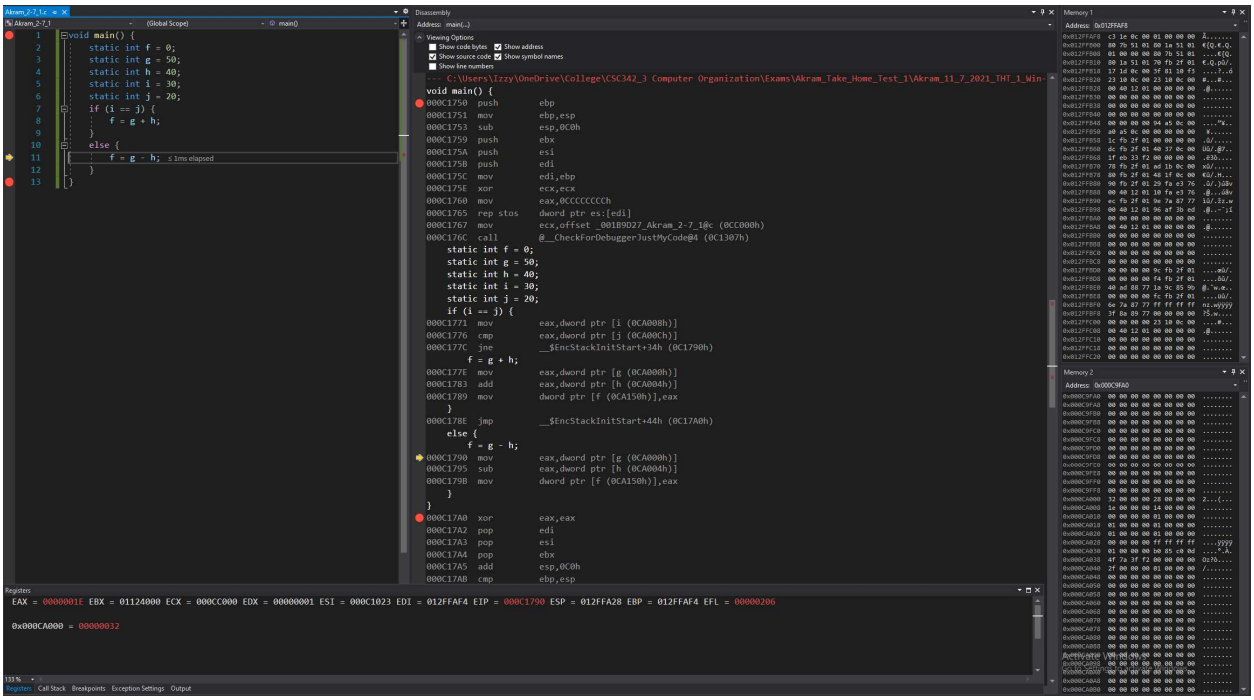

Figure 76: Debugger on Visual Studios (Lines 7 to 10)

Disassembly: this window shows that the code runs lines 7 to 10 and that the values are moved into registers and checked if they meet certain conditions. If it did, it'll run through that, else, jump to another part of the program.

In this case, we jumped to another part.

Registers: shows the value that's stored in each register:

Value of i (stored in memory), is moved to the register,
Value of j (stored in memory), is moved to another register,
Two registers are compared, not satisfying the condition, so it jumped back to another part.

Memory: memory window 1 displays the values around the stack pointer, remains the same since line 7 to 10 didn't store anything into them or change any values.

Memory window 2 displays the value of the static variables stored into the addresses, remaining the same since lines 7 to 10 didn't store anything into the addresses or change any values.

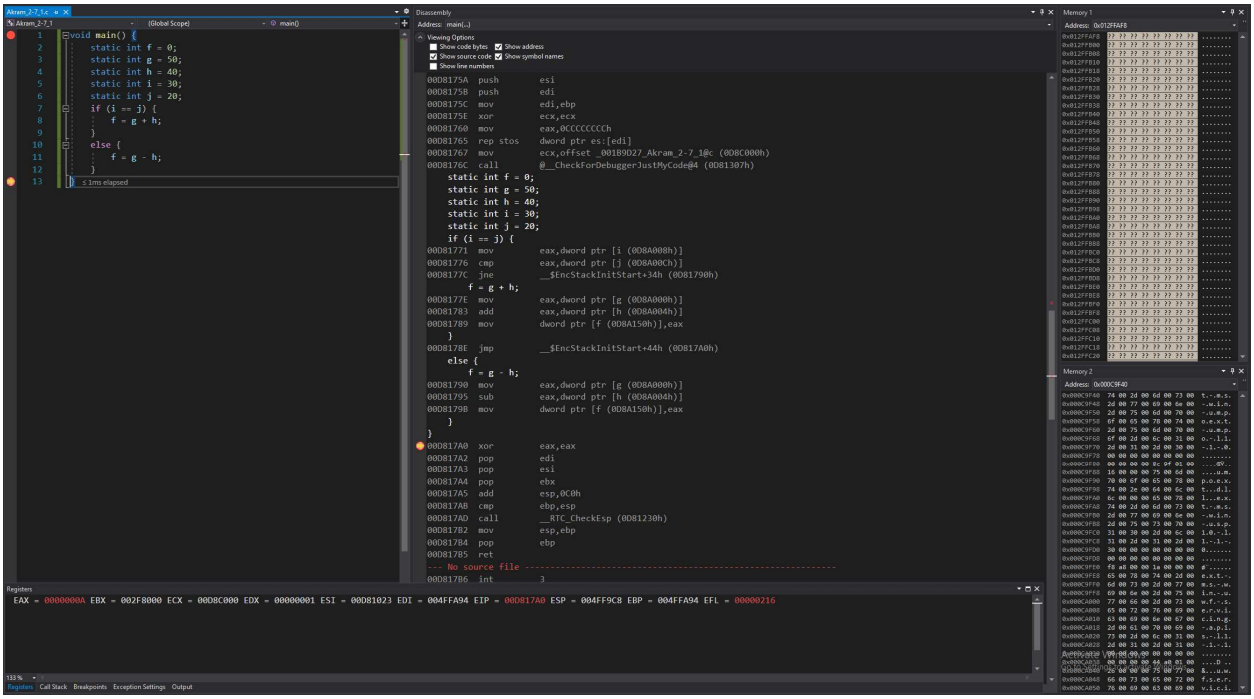

Figure 77: Debugger on Visual Studios (Lines 11 to 13)

Disassembly: this window shows that the code runs lines 11 to 13 and that a value is moved into a register so it can be subtracted from another value. The result is to be stored in memory.

Registers: shows the value that's stored in each register:

Value of g (stored in memory), is moved to the register,

Then that value is subtracted from the value of h (stored in memory), is performed, and the result is stored to the same register.

Value in that register is moved into the memory, stored in the address of f and overwrites the value stored at f.

Memory: memory window 1 displays the values around the stack pointer, remains the same since lines 11 to 13 didn't store anything into them or change any values.

Memory window 2 displays the value of the static variables stored into the addresses:

Variable f is stored in address: 0x000CA150, now containing the value: 0a 00 00 00.

2-7_2.c:

The code written for 2-7_2 in C is shown in Fig 78. It uses three static variables: i, k, and save. Using these three variables; conditions are set up and if the variables satisfy those conditions; loop operations are performed.

In our case and array called save is generated and the value 10 is saved into the 5^{th} index. The loop is run and checked if the value 10 is saved in the i^{th} index of that array:

If it is: increment i by 1,

Else,

Exit the loop.

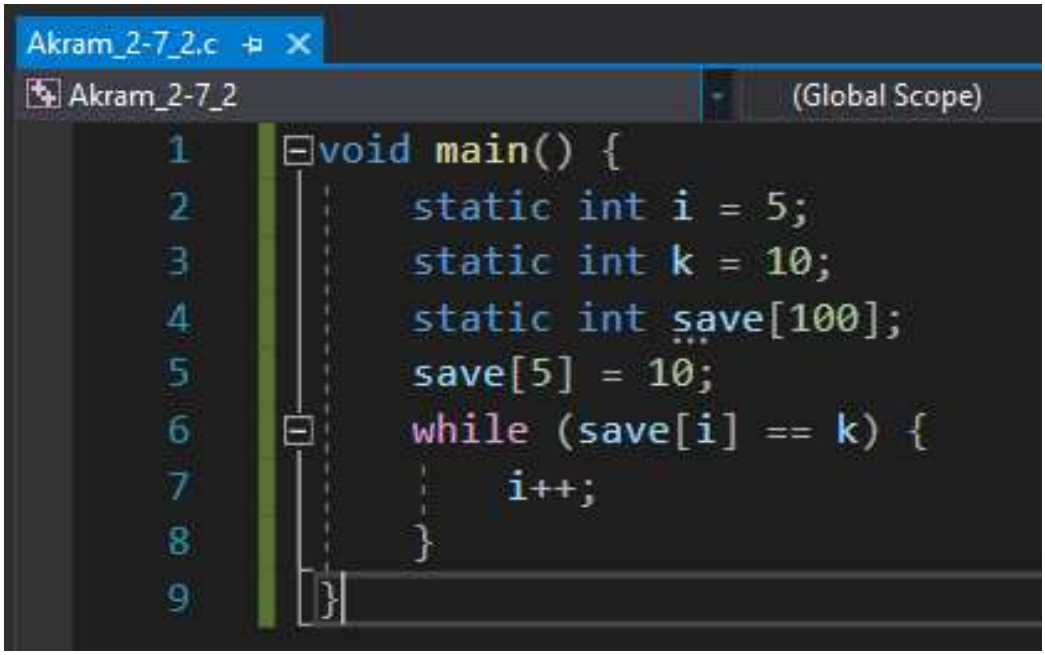

Fig 78. Akram_2-7_2.c executed in VS Win-32 bit

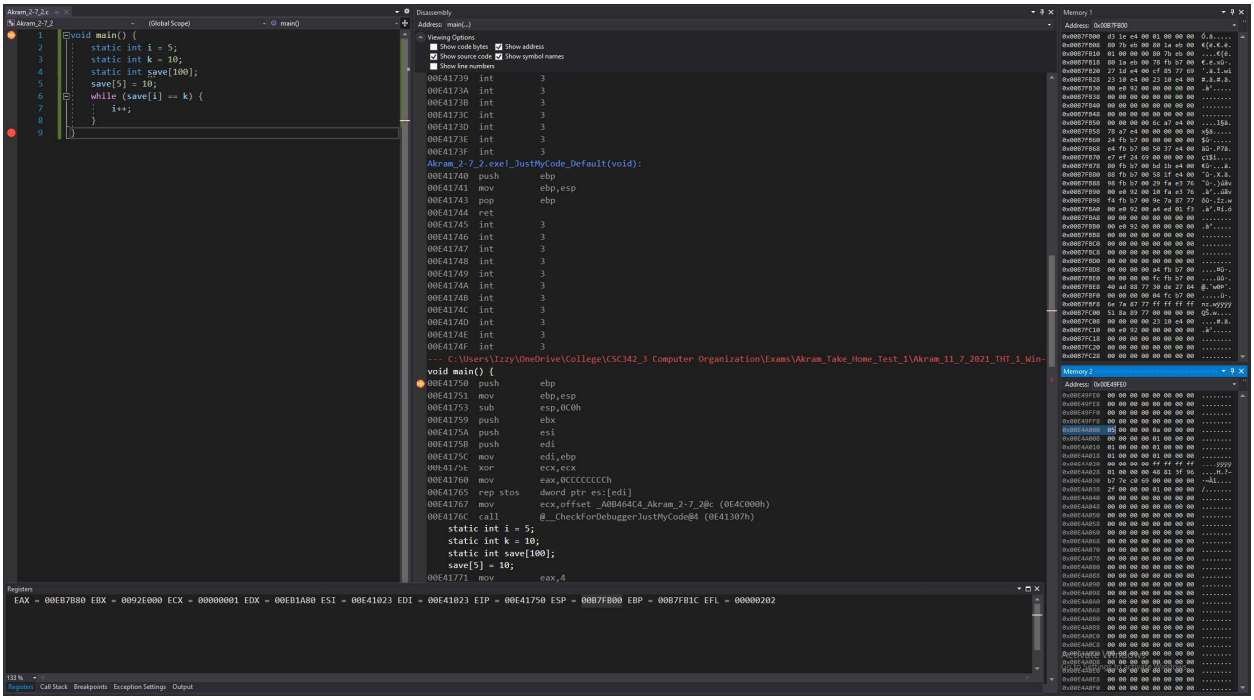

Figure 79: Debugger on Visual Studios (Before Line 1)

Disassembly: this window shows that the code starts at line 1, at main. No instructions have been performed yet to the registers or memories.

Registers: shows the value that's stored in each register, that the stack pointer, which is 0x00B7FB00, is stored.

Memory: memory window 1 displays the values around the stack pointer (all random).

Memory window 2 displays the value of the static variables stored into the addresses.

Variable i is stored in address: 0x00E4A000 containing the value: 05 00 00 00
Variable j is stored in address: 0x00E4A004 containing the value: 0a 00 00 00
Variable save is stored in address: 0x00EA0148 containing the value: 00 00 00 00

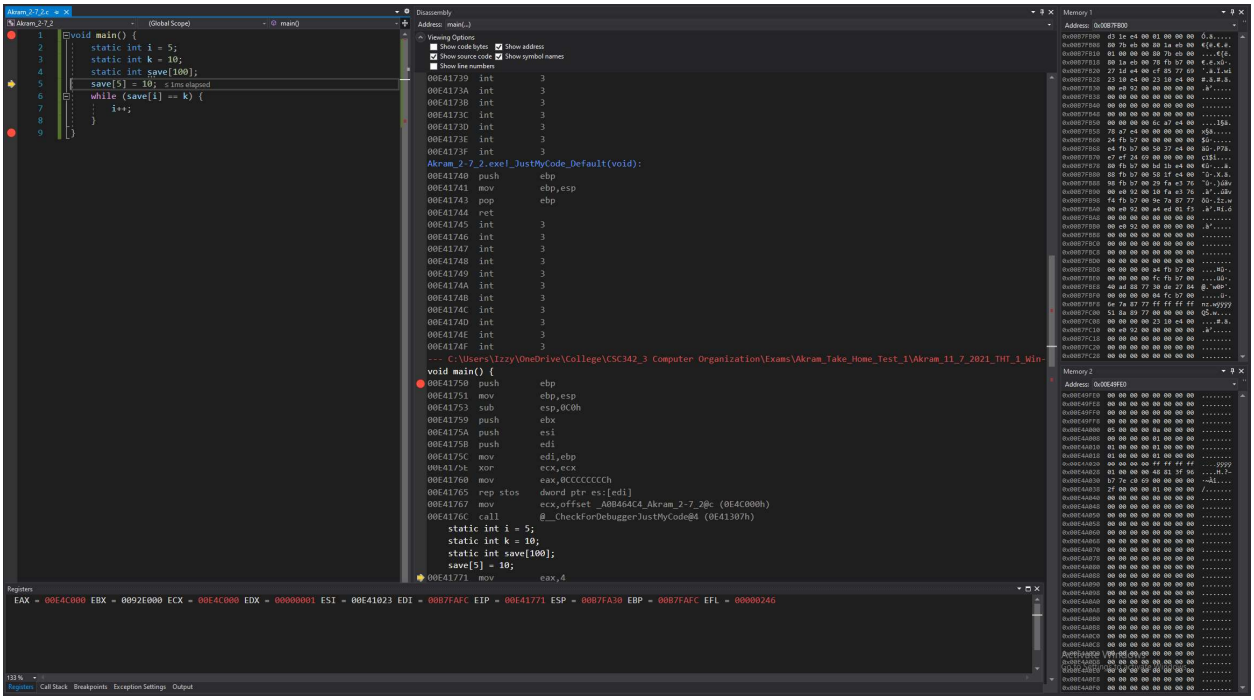

Figure 80: Debugger on Visual Studios (Lines 1 to 4)

Disassembly: this window shows that the code runs lines 1 to 4 and that the registers and memory are set to have the appropriate values to start the program's execution.

Registers: shows the value that's stored in each register. Some values have been changed from before, since this is setting up the registers for the program to run.

Memory: memory window 1 displays the values around the stack pointer (some now initialized).

Memory window 2 displays the value of the static variables stored into the addresses, remaining the same since lines 1 to 4 didn't store anything into the addresses or change any values.

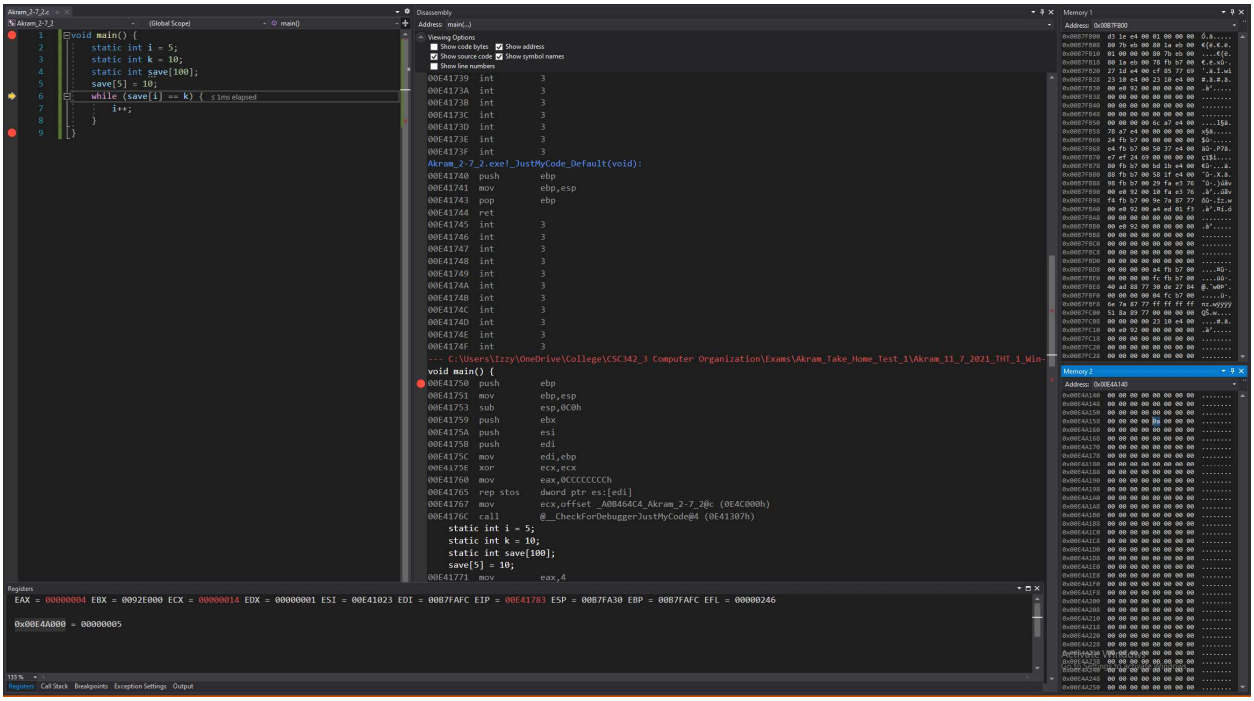

Figure 81: Debugger on Visual Studios (Line 5)

Disassembly: this window shows that the code runs line 5 and that the value 10 is set into a register and then an operation is performed to net us the 5th index.

Using the address of the array and the 5th index' value, we can calculate the address of the 5th index and store the value of 10 into that part of memory.

Registers: shows the value that's stored in each register:

Register EAX set to 00000004,
Operation is performed on it to net us the register: 00000014,
giving us the 5th index (saved in ECX)

Memory: memory window 1 displays the values around the stack pointer, remains the same since line 4 didn't store anything into them or change any values.

Memory window 2 displays the values stored in array A.
Variable A's 5th index is stored in address: 0x00E4A15c, now containing the value: 0a 00 00 00.

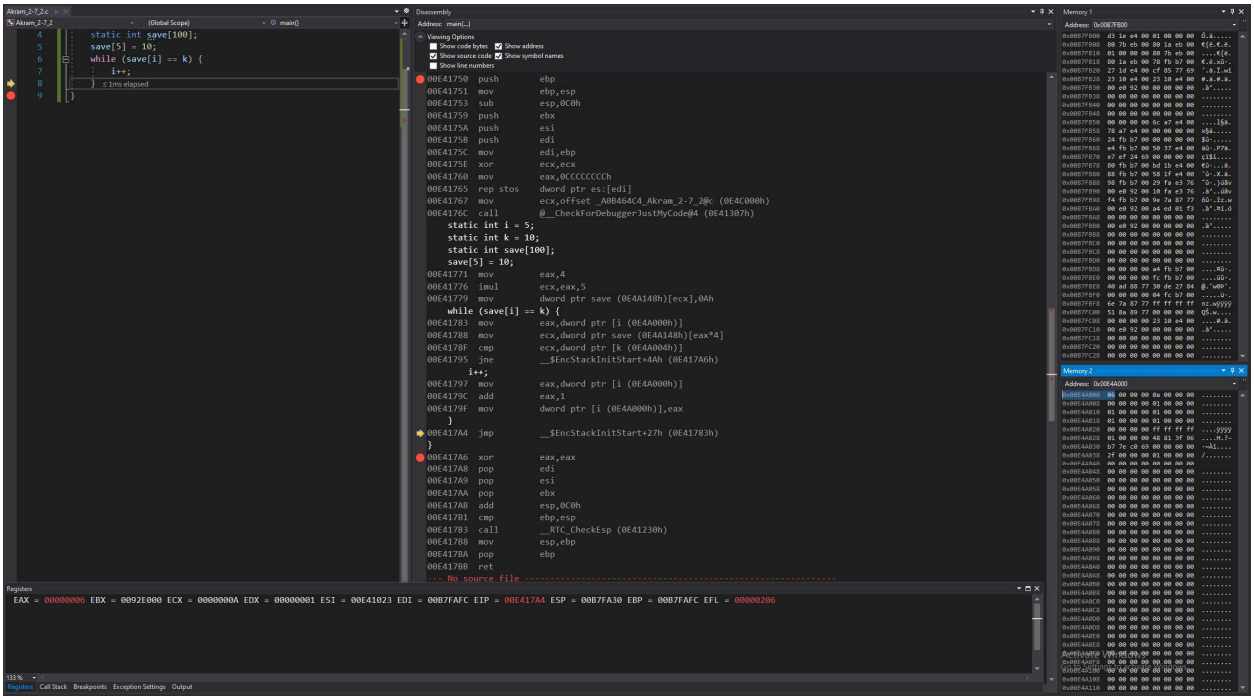

Figure 82: Debugger on Visual Studios (Lines 6 to 7)

Disassembly: this window shows that the code runs lines 6 to 7 and that the values are moved into registers and checked if they meet certain conditions. If it did, it'll run through that, else, jump to another part of the program.

In this case, condition is satisfied, so we increment i by 1.

Registers: shows the value that's stored in each register:

Value of i (stored in memory), is moved to the register.
Then that value is compared with the value of the i^{th} index of the array (stored in memory), is performed and then stored to the same register.

Value of k (stored in memory), is compared with the register to check if they're equal,
Since they are equal, i is stored in a register, incremented by 1, and then stored back to the address of i.

Memory: memory window 1 displays the values around the stack pointer, remains the same since line 6 to 7 didn't store anything into them or change any values.

Memory window 2 displays the value of the static variables stored into the addresses:

Variable i is stored in address: 0x00E4A000, now containing the value: 06 00 00 00.

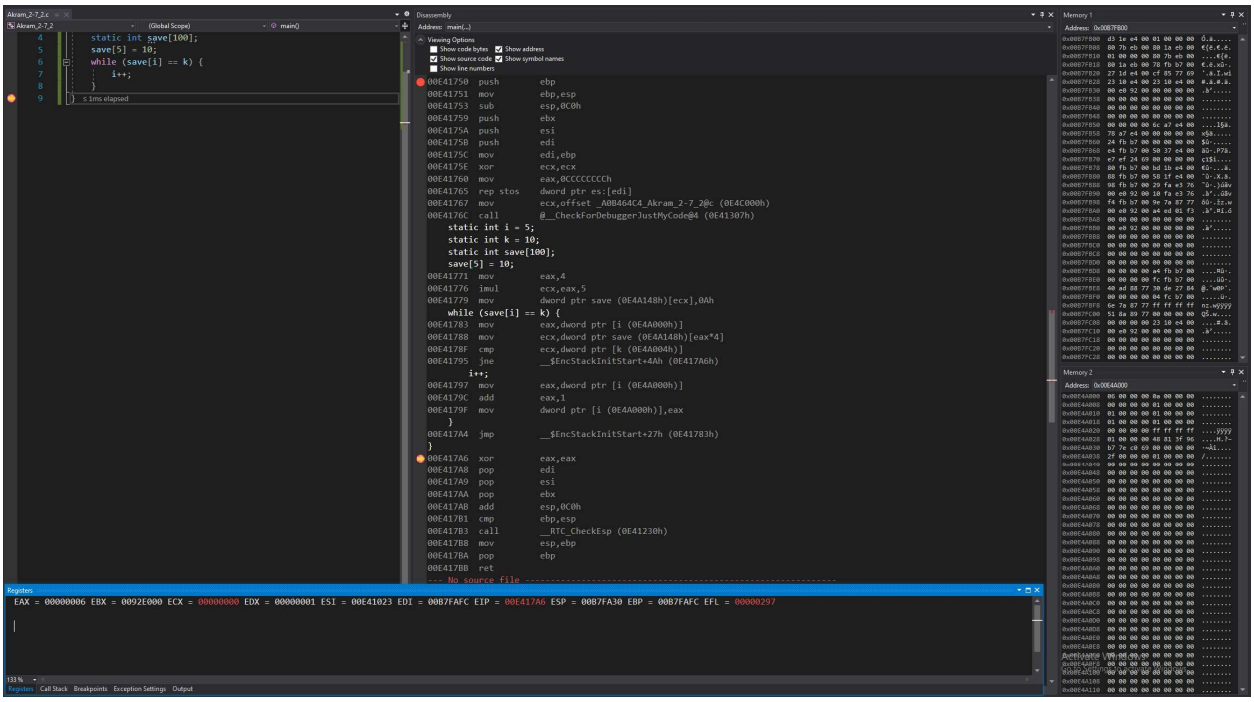

Figure 83: Debugger on Visual Studios (Lines 6 to 9)

Disassembly: this window shows that the code runs lines 6 to 9 and that we jumped back to line 6 because of the while loop. Values are moved into registers and checked if they meet certain conditions. If it did, it'll run through that, else, jump to another part of the program.

In this case, condition is not satisfied, so we jump to the end of the program.

Registers: shows the value that's stored in each register:

Value of i (stored in memory), is moved to the register.
Then that value is compared with the value of the i^{th} index of the array (stored in memory), is performed and then stored to the same register.

Value of k (stored in memory), is compared with the register to check if they're equal, Since they are not equal, we jump to the end of the code.

Memory: memory window 1 displays the values around the stack pointer, remains the same since line 6 to 9 didn't store anything into them or change any values.

Memory window 2 displays the value of the static variables stored into the addresses, remaining the same since lines 6 to 9 didn't store anything into the addresses or change any values.

2-8_1.c:

The code written for 2-8_1 in C is shown in Fig 84. It uses three static variables: a, b, and c. Using these three variables; it'll enter a function called myadd, which will take two parameters a and b and return their sum by storing the result into c.

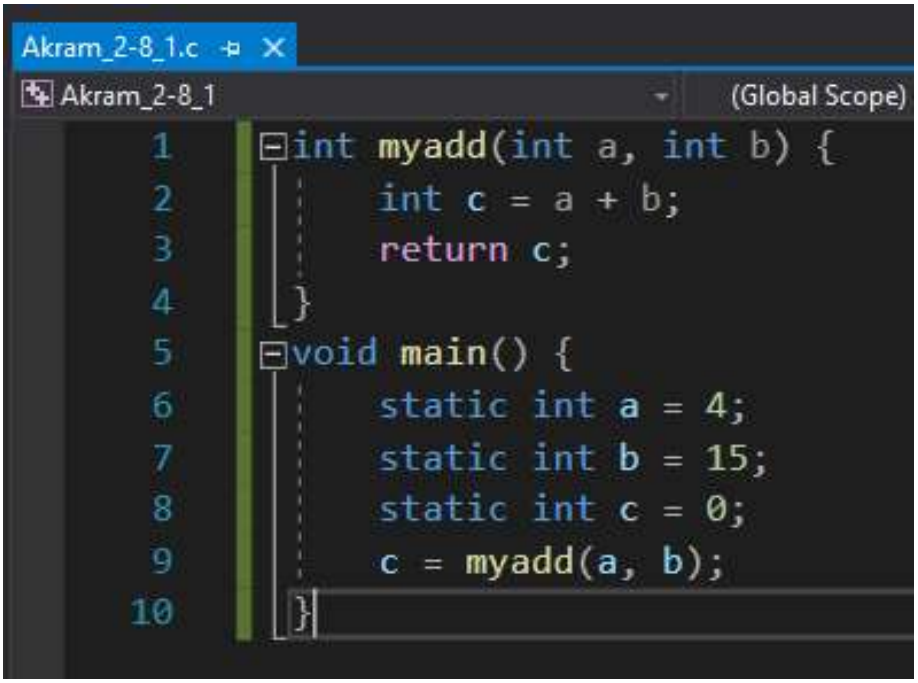

Fig 84. Akram_2-8_1.c executed in VS Win-32 bit

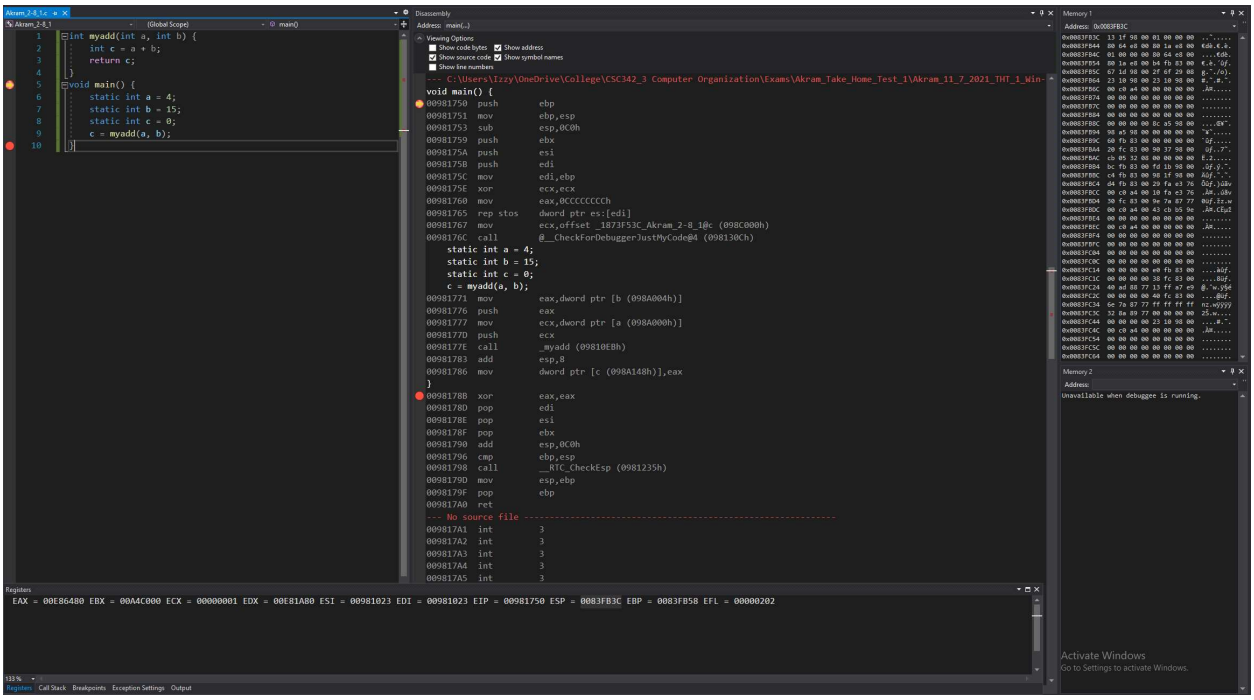

Figure 85: Debugger on Visual Studios (Before Line 1)

Disassembly: this window shows that the code starts at line 1, at main. No instructions have been performed yet to the registers or memories.

Registers: shows the value that's stored in each register, that the stack pointer, which is 0x0083FB3C, is stored.

Memory: memory window 1 displays the values around the stack pointer (all random).

Memory window 2 displays the value of the static variables stored into the addresses.

Variable a is stored in address: 0x0098A000 containing the value: 04 00 00 00
Variable b is stored in address: 0x0098A004 containing the value: 0f 00 00 00
Variable c is stored in address: 0x0098A148 containing the value: 00 00 00 00

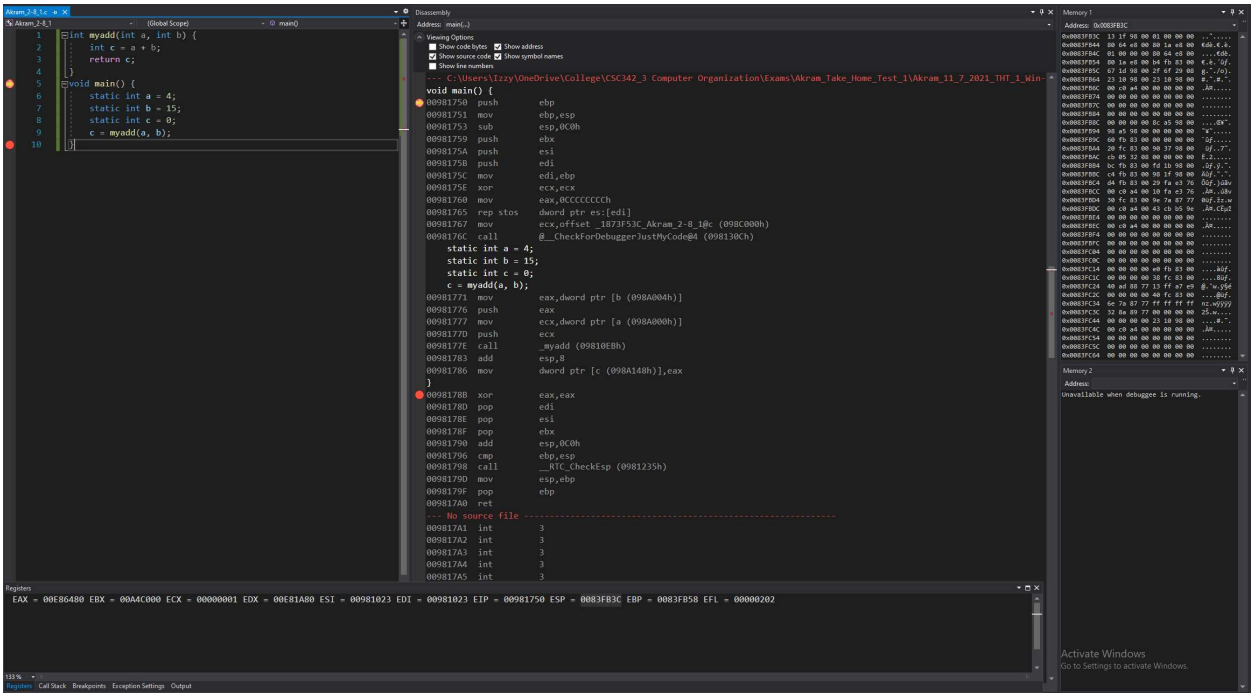

Figure 86: Debugger on Visual Studios (Lines 5 to 8)

Disassembly: this window shows that the code runs lines 5 to 8 and that the registers and memory are set to have the appropriate values to start the program's execution.

Registers: shows the value that's stored in each register. Some values have been changed from before, since this is setting up the registers for the program to run.

Memory: memory window 1 displays the values around the stack pointer (some now initialized).

Memory window 2 displays the value of the static variables stored into the addresses, remaining the same since lines 5 to 8 didn't store anything into the addresses or change any values.

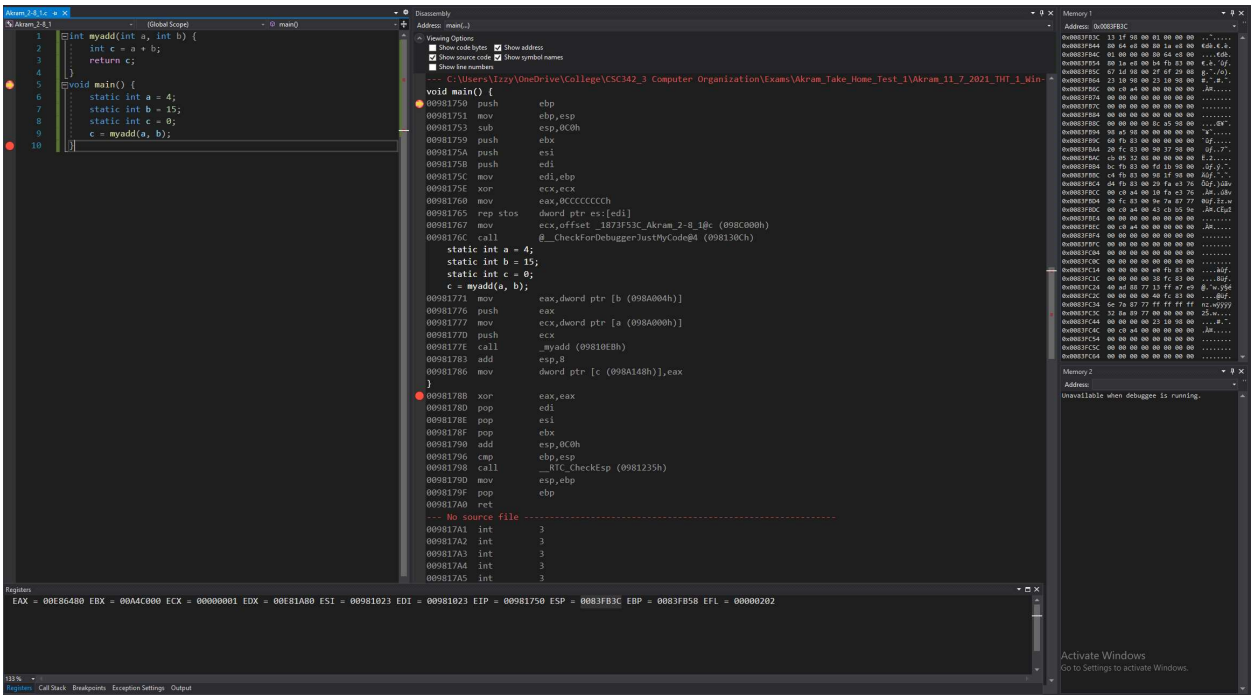

Figure 87: Debugger on Visual Studios (Lines 9)

Disassembly: this window shows that the code runs line 9 and that the values are moved into registers to be used for a function call.

Registers: shows the value that's stored in each register:

Value of a (stored in memory), is moved to the register,
Value of b (stored in memory), is moved to another register.

Memory: memory window 1 displays the values around the stack pointer, remains the same since line 9 didn't store anything into them or change any values.

Memory window 2 displays the value of the static variables stored into the addresses, remaining the same since line 9 didn't store anything into the addresses or change any values.

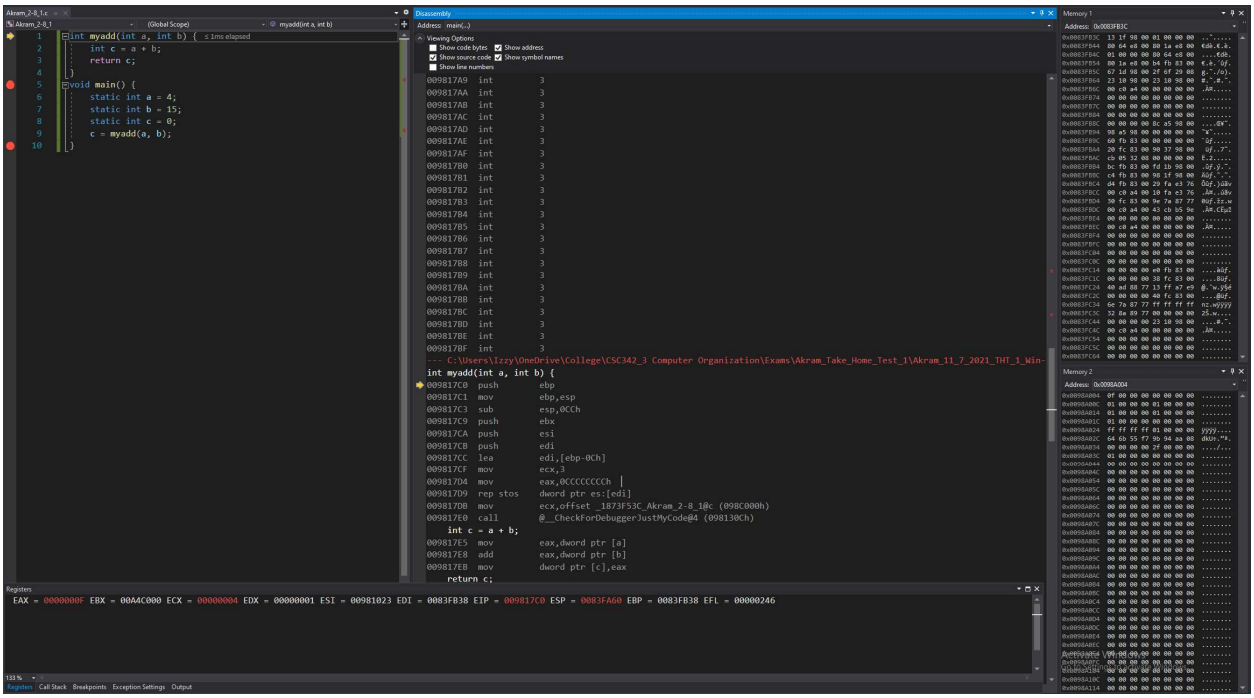

Figure 88: Debugger on Visual Studios (Before Line 1)

Disassembly: this window shows that the code jumped to line 1, at the function call. No instructions have been performed yet to the registers or memories.

Registers: shows the value that's stored in each register, that the stack pointer, which is now 0x00AFF8F4, is stored.

Memory: memory window 1 displays the values around the stack pointer, remains the same since line 9 didn't store anything into them or change any values.

Memory window 2 displays the value of the static variables stored into the addresses, remaining the same since line 9 didn't store anything into the addresses or change any values.

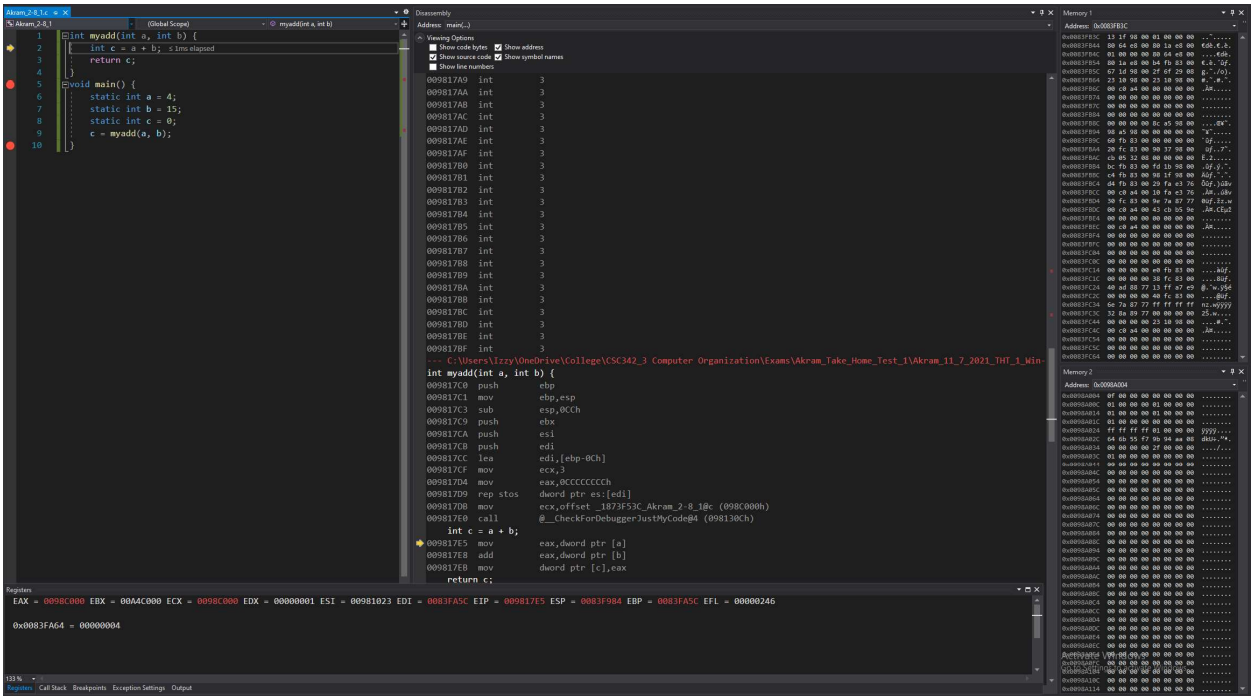

Figure 89: Debugger on Visual Studios (Line 1)

Disassembly: this window shows that the code runs line 1, and that the registers and memory are set to have the appropriate values to start the program's execution.

Registers: shows the value that's stored in each register. Some values have been changed from before, since this is setting up the registers for the program to run.

Memory: memory window 1 displays the values around the new stack pointer (some now are not all c's).

Memory window 2 displays the value of the static variables stored into the addresses, remaining the same since line 1 didn't store anything into the addresses or change any values.

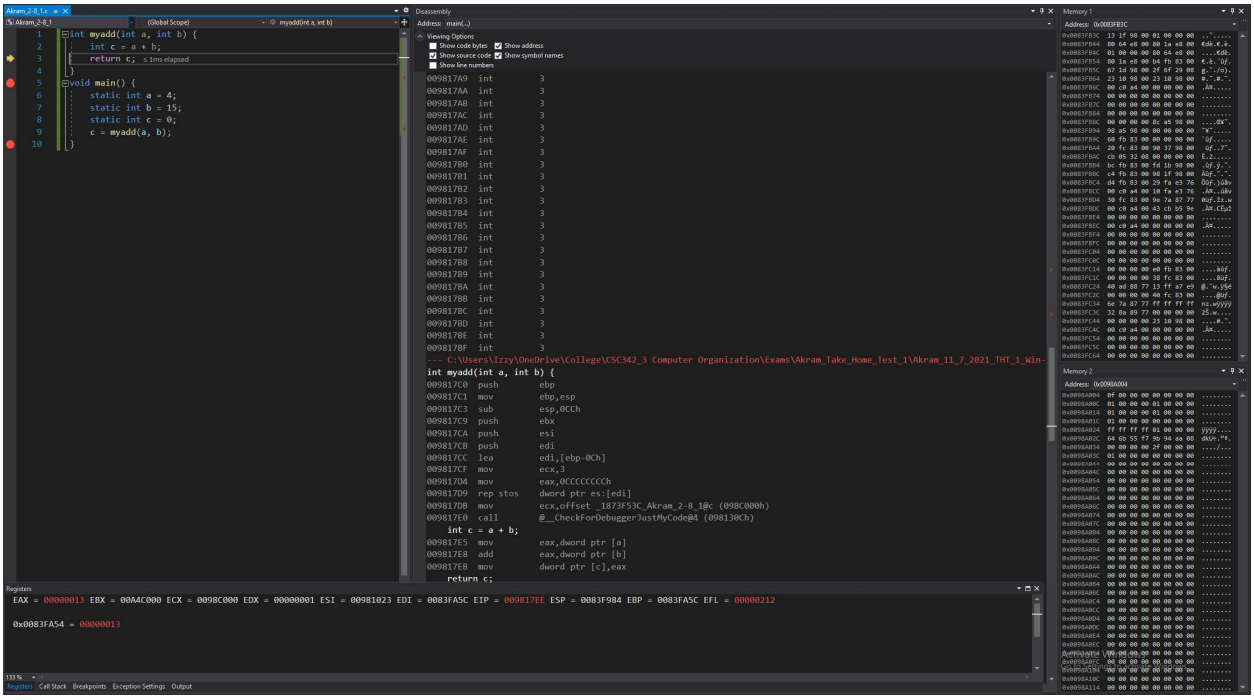

Figure 90: Debugger on Visual Studios (Line 2)

Disassembly: this window shows that the code runs line 2 and that the registers and memory moved a value into a register so it can be added to with another value and the result is to be saved into a pointer.

Registers: shows the value that's stored in each register:

Value that points to a, is moved to the register,

Addition of that the value in register with value that points to b, is performed, and then stored to the same register.

Value in that register is moved into the value that points to c.

Memory: memory window 1 displays the values around the stack pointer, remains the same since line 2 didn't store anything into them or change any values.

Memory window 2 displays the value of the static variables stored into the addresses, remaining the same since line 2 didn't store anything into the addresses or change any values.

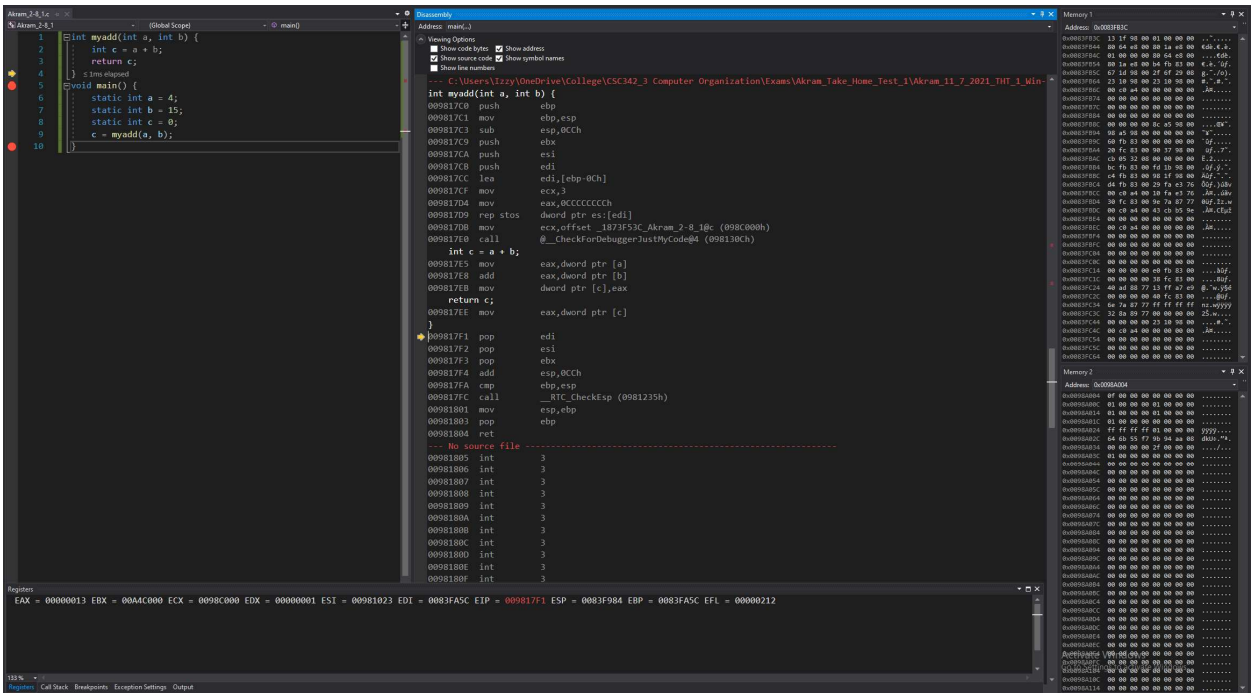

Figure 91: Debugger on Visual Studios (Lines 3 to 4)

Disassembly: this window shows that the code runs lines 3 to 4 and that a value is moved to a register so it can be the return of a function call.

Registers: shows the value that's stored in each register:

Value that points to c, is moved to the register EAX.

Memory: memory window 1 displays the values around the stack pointer, remains the same since lines 3 to 4 didn't store anything into them or change any values.

Memory window 2 displays the value of the static variables stored into the addresses, remaining the same since lines 3 to 4 didn't store anything into the addresses or change any values.

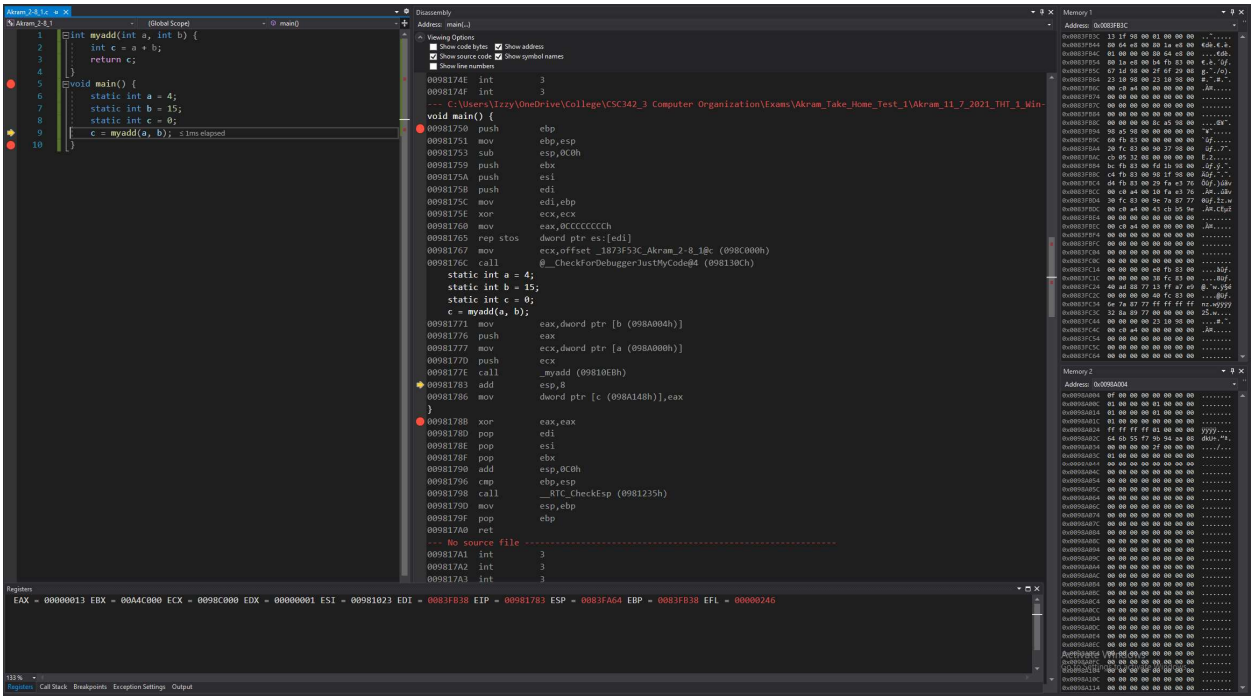

Figure 92: Debugger on Visual Studios (Line 9)

Disassembly: this window shows that the code jumped to line 9, at main.

Registers: shows the value that's stored in each register, that the stack pointer, which is now 0x0083FA6C, is stored.

Memory: memory window 1 displays the values around the stack pointer, remains the same since line 4 didn't store anything into them or change any values.

Memory window 2 displays the value of the static variables stored into the addresses, remaining the same since line 4 didn't store anything into the addresses or change any values.

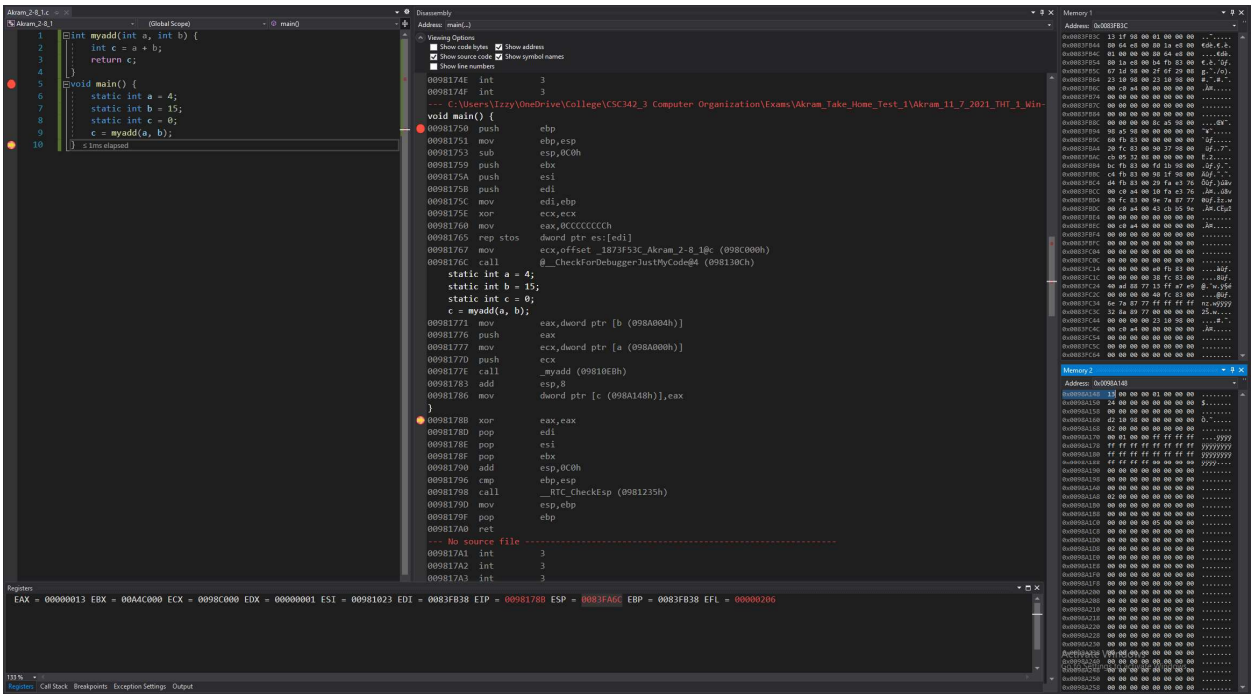

Figure 93: Debugger on Visual Studios (Lines 9 to 10)

Disassembly: this window shows that the code runs lines 9 to 10 and we add 8 to the stack pointer to return to the original stack pointer and moved the value at the register into an address.

Registers: shows the value that's stored in each register, that the stack pointer, which is now 0x0083FA6C, is stored.

Memory: memory window 1 displays the values around the stack pointer, remains the same since line 9 to 10 didn't store anything into them or change any values.

Memory window 2 displays the value of the static variables stored into the addresses:

Variable c is stored in address: 0x0098A148, now containing the value: 13 00 00 00.

Intel X86 ISA on Linux 64-bit gcc and gdb:

C is a programing language that produces a .c file. For this demonstration we're running C on Linux for its gcc and gdb, Intel X86 ISA on Linux 64-bit compiler. This compiler operates as little endian, so the least significant byte (LSB) is stored in the lowest memory address.

2-2_1.c:

The code written for 2-2_1 in C is shown in Fig 94. It uses five static variables: a, b, c, d, and e. Using these five variables; arithmetic is performed and stored in the variables.

In our case, we're adding the value of b and c and the sum is stored in a. We're also subtracting the values of a and e and storing that result in d.

```
(gdb) list
1       void 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       }
(gdb)
```

Fig 94. Akram_2-2_1.c executed in Linux 64-32 bit

```
(gdb) disassemble
Dump of assembler code for function main:
=> 0x0000555555555129 <+0>:     endbr64
   0x000055555555512d <+4>:     push   %rbp
   0x000055555555512e <+5>:     mov    %rsp,%rbp
   0x0000555555555131 <+8>:     mov    0x2ed9(%rip),%edx        # 0x555555558010 <b.1913>
   0x0000555555555137 <+14>:    mov    0x2ed7(%rip),%eax        # 0x555555558014 <c.1914>
   0x000055555555513d <+20>:    add    %edx,%eax
   0x000055555555513f <+22>:    mov    %eax,0x2ed3(%rip)        # 0x555555558018 <a.1912>
   0x0000555555555145 <+28>:    mov    0x2ecd(%rip),%edx        # 0x555555558018 <a.1912>
   0x000055555555514b <+34>:    mov    0x2ecb(%rip),%eax        # 0x55555555801c <e.1916>
   0x0000555555555151 <+40>:    sub    %eax,%edx
   0x0000555555555153 <+42>:    mov    %edx,%eax
   0x0000555555555155 <+44>:    mov    %eax,0x2ec5(%rip)        # 0x555555558020 <d.1915>
   0x000055555555515b <+50>:    nop
   0x000055555555515c <+51>:    pop    %rbp
   0x000055555555515d <+52>:    retq
End of assembler dump.
(gdb) print /x $rsp
$1 = 0x7fffffffdf78
(gdb) print /x $rbp
$2 = 0x0
(gdb) info registers
rax            0x555555555129      93824992235817
rbx            0x555555555160      93824992235872
rcx            0x555555555160      93824992235872
rdx            0x7fffffffe078      140737488347256
rsi            0x7fffffffe068      140737488347240
rdi            0x1                 1
rbp            0x0                 0x0
rsp            0x7fffffffdf78      0x7fffffffdf78
r8             0x0                 0
r9             0x7ffff7fe0d50      140737354009936
r10            0x7ffff7ffcf68      140737354125160
r11            0x206               518
r12            0x555555555040      93824992235584
r13            0x7fffffffe060      140737488347232
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)
```

Fig 95. Akram_2-2_1.c GDB on Linux (Before Line 1)

Disassembly: this window shows that the code runs line 1 and that stack pointer and base pointer are set.

Registers: shows the value that's stored in each register, that the:

stack pointer is at 0x7fffffffdf78,
base pointer is at 0x0,
there are other registers containing random values.

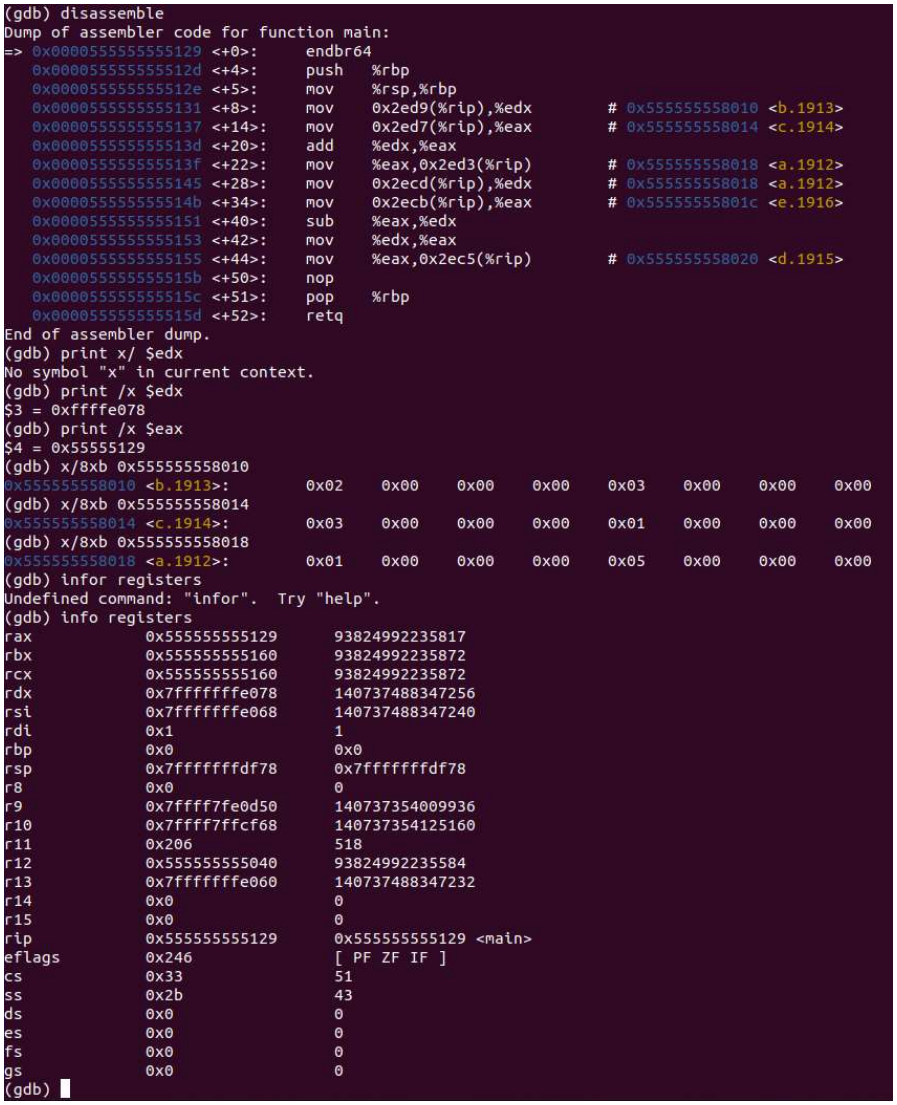

Fig 96. Akram_2-2_1.c GDB on Linux (Lines 1-8)

Disassembly: this window shows that the code runs lines 1 to 8 and that values are moved into registers for addition purposes. The result is to be stored in memory.

Registers: shows the value that's stored in each register, that the:

Value of b (stored in memory), is moved to edx,
Value of c (stored in memory), is moved to eax,
Addition is performed on the two registers and result is stored in eax.
Value in that register is moved into the memory, stored in the address of a and overwrites the value stored at a.

Memory: shows the values stored in an address:
0x555555558010 contains 0x02 0x00 0x00 0x00 0x03 0x00 0x00 0x00
0x555555558014 contains 0x03 0x00 0x00 0x00 0x01 0x00 0x00 0x00
0x555555558018 contains 0x01 0x00 0x00 0x00 0x05 0x00 0x00 0x00

```
(gdb) disassemble
Dump of assembler code for function main:
=> 0x0000555555555129 <+0>:     endbr64
   0x000055555555512d <+4>:     push   %rbp
   0x000055555555512e <+5>:     mov    %rsp,%rbp
   0x0000555555555131 <+8>:     mov    0x2ed9(%rip),%edx        # 0x555555558010 <b.1913>
   0x0000555555555137 <+14>:    mov    0x2ed7(%rip),%eax        # 0x555555558014 <c.1914>
   0x000055555555513d <+20>:    add    %edx,%eax
   0x000055555555513f <+22>:    mov    %eax,0x2ed3(%rip)        # 0x555555558018 <a.1912>
   0x0000555555555145 <+28>:    mov    0x2ecd(%rip),%edx        # 0x555555558018 <a.1912>
   0x000055555555514b <+34>:    mov    0x2ecb(%rip),%eax        # 0x55555555801c <e.1916>
   0x0000555555555151 <+40>:    sub    %eax,%edx
   0x0000555555555153 <+42>:    mov    %edx,%eax
   0x0000555555555155 <+44>:    mov    %eax,0x2ec5(%rip)        # 0x555555558020 <d.1915>
   0x000055555555515b <+50>:    nop
   0x000055555555515c <+51>:    pop    %rbp
   0x000055555555515d <+52>:    retq
End of assembler dump.
(gdb) print /x $edx
$5 = 0xffffe078
(gdb) print /x $eax
$6 = 0x55555129
(gdb) x/8xb 0x555555558018
0x555555558018 <a.1912>:        0x01    0x00    0x00    0x00    0x05    0x00    0x00    0x00
(gdb) x/8xb 0x55555555801c
0x55555555801c <e.1916>:        0x05    0x00    0x00    0x00    0x04    0x00    0x00    0x00
(gdb) x/8xb 0x555555558020
0x555555558020 <d.1915>:        0x04    0x00    0x00    0x00    0x00    0x00    0x00    0x00
(gdb) info registers
rax            0x555555555129      93824992235817
rbx            0x555555555160      93824992235872
rcx            0x555555555160      93824992235872
rdx            0x7fffffffe078      140737488347256
rsi            0x7fffffffe068      140737488347240
rdi            0x1                 1
rbp            0x0                 0x0
rsp            0x7fffffffdf78      0x7fffffffdf78
r8             0x0                 0
r9             0x7ffff7fe0d50      140737354009936
r10            0x7ffff7ffcf68      140737354125160
r11            0x206               518
r12            0x555555555040      93824992235584
r13            0x7fffffffe060      140737488347232
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)
```

Fig 97. Akram_2-2_1.c GDB on Linux (Lines 8-9)

Disassembly: this window shows that the code runs lines 8 to 9 and that values are moved into registers for subtraction purposes. The result is to be stored in memory.

Registers: shows the value that's stored in each register, that the:

Value of a (stored in memory), is moved to edx,
Value of e (stored in memory), is moved to eax,
Subtraction is performed on the two registers and result is stored in edx.
eax is set to the same value as edx,
Value in that register is moved into the memory, stored in the address of f and overwrites the value stored at f.

Memory: shows the values stored in an address:

0x555555558018 contains 0x01 0x00 0x00 0x00 0x04 0x00 0x00 0x00
0x55555555801c contains 0x05 0x00 0x00 0x00 0x04 0x00 0x00 0x00
0x555555558020 contains 0x04 0x00 0x00 0x00 0x00 0x00 0x00 0x00

2-2_2.c:

The code written for 2-2_1 in C is shown in Fig 98. It uses five static variables: a, b, c, d, and e. Using these five variables; arithmetic is performed and stored in the variables.

In our case, we're adding the value of b and c and the sum is stored in a. We're also subtracting the values of a and e and storing that result in d.

```
(gdb) list
1       void main() {
2               static int f = 0;
3               static int g = 50;
4               static int h = 40;
5               static int i = 30;
6               static int j = 20;
7               f = (g + h) - (i + j);
8       }
(gdb)
```

Fig 98. Akram_2-2_2.c executed in Linux 64-32 bit

```
(gdb) disassemble
Dump of assembler code for function main:
=> 0x0000555555555129 <+0>:     endbr64
   0x000055555555512d <+4>:     push   %rbp
   0x000055555555512e <+5>:     mov    %rsp,%rbp
   0x0000555555555131 <+8>:     mov    0x2ed9(%rip),%edx        # 0x555555558010 <g.1913>
   0x0000555555555137 <+14>:    mov    0x2ed7(%rip),%eax        # 0x555555558014 <h.1914>
   0x000055555555513d <+20>:    lea    (%rdx,%rax,1),%ecx
   0x0000555555555140 <+23>:    mov    0x2ed2(%rip),%edx        # 0x555555558018 <i.1915>
   0x0000555555555146 <+29>:    mov    0x2ed0(%rip),%eax        # 0x55555555801c <j.1916>
   0x000055555555514c <+35>:    add    %edx,%eax
   0x000055555555514e <+37>:    sub    %eax,%ecx
   0x0000555555555150 <+39>:    mov    %ecx,%eax
   0x0000555555555152 <+41>:    mov    %eax,0x2ecc(%rip)        # 0x555555558024 <f.1912>
   0x0000555555555158 <+47>:    nop
   0x0000555555555159 <+48>:    pop    %rbp
   0x000055555555515a <+49>:    retq
End of assembler dump.
(gdb) print /x $rsp
$1 = 0x7fffffffdf78
(gdb) print /x rbp
No symbol "rbp" in current context.
(gdb) print /x $rbp
$2 = 0x0
(gdb) info registers
rax            0x555555555129      93824992235817
rbx            0x555555555160      93824992235872
rcx            0x555555555160      93824992235872
rdx            0x7fffffffe078      140737488347256
rsi            0x7fffffffe068      140737488347240
rdi            0x1                 1
rbp            0x0                 0x0
rsp            0x7fffffffdf78      0x7fffffffdf78
r8             0x0                 0
r9             0x7ffff7fe0d50      140737354009936
r10            0x7ffff7ffcf68      140737354125160
r11            0x206               518
r12            0x555555555040      93824992235584
r13            0x7fffffffe060      140737488347232
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)
```

Fig 99. Akram_2-2_2.c GDB on Linux (Before Line 1)

Disassembly: this window shows that the code runs line 1 and that stack pointer and base pointer are set.

Registers: shows the value that's stored in each register, that the:

stack pointer is at 0x7fffffffdf78,
base pointer is at 0x0,
there are other registers containing random values.

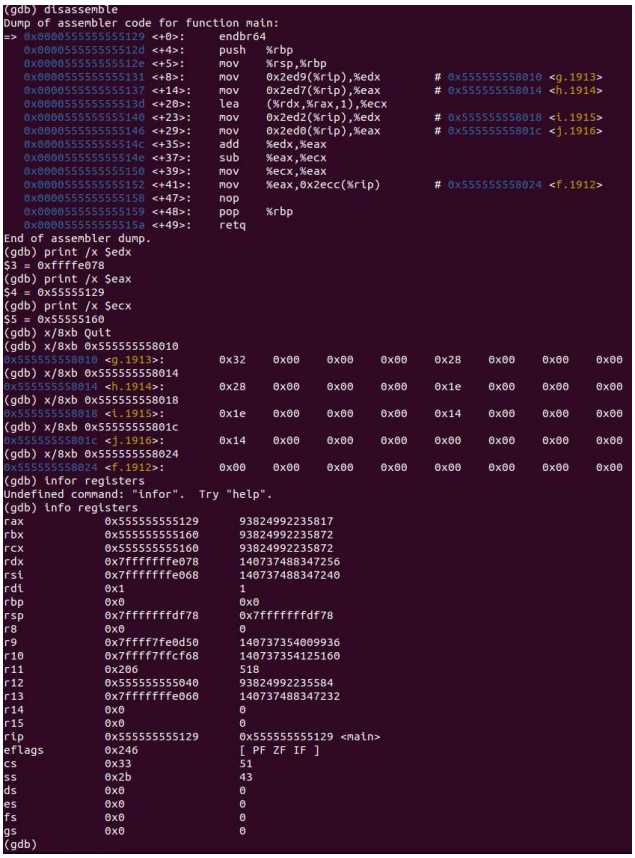

Fig 100. Akram_2-2_2.c GDB on Linux (Lines 1-8)

Disassembly: this window shows that the code runs lines 1 to 8 and that values are moved into registers for addition and subtraction purposes. The result is to be stored in memory.

Registers: shows the value that's stored in each register, that the:

Value of g (stored in memory), is moved to edx,
Value of h (stored in memory), is moved to eax,
operation is performed on the two registers and result is stored in ecx.
Value of i (stored in memory), is moved to edx,
Value of j (stored in memory), is moved to eax,
operation is performed on the two registers and result is stored in eax.
addition is performed on the two registers and result is stored in eax.
subtraction is performed on the eax and ecx and result is stored in ecx.
Value in that register is moved into the memory, stored in the address of f and overwrites the value stored at f.

Memory: shows the values stored in an address:

0x555555558010 contains 0x32 0x00 0x00 0x00 0x28 0x00 0x00 0x00
0x555555558014 contains 0x28 0x00 0x00 0x00 0x1e 0x00 0x00 0x00
0x555555558018 contains 0x1e 0x00 0x00 0x00 0x14 0x00 0x00 0x00
0x55555555801c contains 0x14 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x555555558020 contains 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00

2-3_1.c:

The code written for 2-3_1 in C is shown in Fig 101. It uses three static variables: g, h, and A. Using these three variables; an array is created, and addition is executed through its usage.

In our case, 55 is stored in the 8th index of array A and is added to h. The result from that addition is stored into g.

```
(gdb) list
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       }
(gdb)
```

Fig 101. Akram_2-3_1.c executed in Linux 64-32 bit

```
Dump of assembler code for function main:
=> 0x0000555555555129 <+0>:     endbr64
   0x000055555555512d <+4>:     push   %rbp
   0x000055555555512e <+5>:     mov    %rsp,%rbp
   0x0000555555555131 <+8>:     movl   $0x37,0x2f25(%rip)        # 0x555555558060 <A.1914+32>
   0x000055555555513b <+18>:    mov    0x2f1f(%rip),%edx         # 0x555555558060 <A.1914+32>
   0x0000555555555141 <+24>:    mov    0x2ec9(%rip),%eax         # 0x555555558010 <h.1913>
   0x0000555555555147 <+30>:    add    %edx,%eax
   0x0000555555555149 <+32>:    mov    %eax,0x3081(%rip)         # 0x5555555581d0 <g.1912>
   0x000055555555514f <+38>:    nop
   0x0000555555555150 <+39>:    pop    %rbp
   0x0000555555555151 <+40>:    retq
End of assembler dump.
(gdb) print /x $rsp
$1 = 0x7fffffffdf78
(gdb) print /x $rbp
$2 = 0x0
(gdb) info registers
rax            0x555555555129      93824992235817
rbx            0x555555555160      93824992235872
rcx            0x555555555160      93824992235872
rdx            0x7fffffffe078      140737488347256
rsi            0x7fffffffe068      140737488347240
rdi            0x1                 1
rbp            0x0                 0x0
rsp            0x7fffffffdf78      0x7fffffffdf78
r8             0x0                 0
r9             0x7ffff7fe0d50      140737354009936
r10            0x7ffff7ffcf68      140737354125160
r11            0x206               518
r12            0x555555555040      93824992235584
r13            0x7fffffffe060      140737488347232
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)
```

Fig 102. Akram_2-3_1.c GDB on Linux (Before Line 1)

Disassembly: this window shows that the code runs line 1 and that stack pointer and base pointer are set.

Registers: shows the value that's stored in each register, that the:

stack pointer is at 0x7fffffffdf78,
base pointer is at 0x0,
there are other registers containing random values.

```
(gdb) disassemble
Dump of assembler code for function main:
=> 0x0000555555555129 <+0>:     endbr64
   0x000055555555512d <+4>:     push   %rbp
   0x000055555555512e <+5>:     mov    %rsp,%rbp
   0x0000555555555131 <+8>:     movl   $0x37,0x2f25(%rip)        # 0x555555558060 <A.1914+32>
   0x000055555555513b <+18>:    mov    0x2f1f(%rip),%edx         # 0x555555558060 <A.1914+32>
   0x0000555555555141 <+24>:    mov    0x2ec9(%rip),%eax         # 0x555555558010 <h.1913>
   0x0000555555555147 <+30>:    add    %edx,%eax
   0x0000555555555149 <+32>:    mov    %eax,0x3081(%rip)         # 0x5555555581d0 <g.1912>
   0x000055555555514f <+38>:    nop
   0x0000555555555150 <+39>:    pop    %rbp
   0x0000555555555151 <+40>:    retq
End of assembler dump.
(gdb) print /x $0x37
$3 = 0x0
(gdb) x/8xb 0x555555558060
0x555555558060 <A.1914+32>:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
(gdb) info registers
rax            0x555555555129      93824992235817
rbx            0x555555555160      93824992235872
rcx            0x555555555160      93824992235872
rdx            0x7fffffffe078      140737488347256
rsi            0x7fffffffe068      140737488347240
rdi            0x1                 1
rbp            0x0                 0x0
rsp            0x7fffffffdf78      0x7fffffffdf78
r8             0x0                 0
r9             0x7ffff7fe0d50      140737354009936
r10            0x7ffff7ffcf68      140737354125160
r11            0x206               518
r12            0x555555555040      93824992235584
r13            0x7fffffffe060      140737488347232
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)
```

Fig 103. Akram_2-3_1.c GDB on Linux (Lines 1-5)

Disassembly: this window shows that the code runs lines 1 to 5 and that the 8th index of the array A is set to the value 55.

Registers: shows the value that's stored in each register, that the:

Values remain the same since lines 1 to 5 don't change anything.

Memory: shows the values stored in an address:
0x555555558060 contains 0x37 0x00 0x00 0x00 0x00 0x00 0x00 0x00

```
(gdb) disassemble
Dump of assembler code for function main:
=> 0x0000555555555129 <+0>:     endbr64
   0x000055555555512d <+4>:     push   %rbp
   0x000055555555512e <+5>:     mov    %rsp,%rbp
   0x0000555555555131 <+8>:     movl   $0x37,0x2f25(%rip)        # 0x555555558060 <A.1914+32>
   0x000055555555513b <+18>:    mov    0x2f1f(%rip),%edx         # 0x555555558060 <A.1914+32>
   0x0000555555555141 <+24>:    mov    0x2ec9(%rip),%eax         # 0x555555558010 <h.1913>
   0x0000555555555147 <+30>:    add    %edx,%eax
   0x0000555555555149 <+32>:    mov    %eax,0x3081(%rip)         # 0x5555555581d0 <g.1912>
   0x000055555555514f <+38>:    nop
   0x0000555555555150 <+39>:    pop    %rbp
   0x0000555555555151 <+40>:    retq
End of assembler dump.
(gdb) print /x $edx
$1 = 0xffffe078
(gdb) print /x $eax
$2 = 0x55555129
(gdb) x/8xb 0x555555558060
0x555555558060 <A.1914+32>:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
(gdb) x/8xb 0x555555558010
0x555555558010 <h.1913>:        0x16    0x00    0x00    0x00    0x00    0x00    0x00    0x00
(gdb) x/8xb 0x5555555581d0
0x5555555581d0 <g.1912>:        0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
(gdb) info registers
rax            0x555555555129      93824992235817
rbx            0x555555555160      93824992235872
rcx            0x555555555160      93824992235872
rdx            0x7fffffffe078      140737488347256
rsi            0x7fffffffe068      140737488347240
rdi            0x1                 1
rbp            0x0                 0x0
rsp            0x7fffffffdf78      0x7fffffffdf78
r8             0x0                 0
r9             0x7ffff7fe0d50      140737354009936
r10            0x7ffff7ffcf68      140737354125160
r11            0x206               518
r12            0x555555555040      93824992235584
r13            0x7fffffffe060      140737488347232
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)
```

Fig 104. Akram_2-3_1.c GDB on Linux (Lines 6-7)

Disassembly: this window shows that the code runs lines 1 to 8 and that values are moved into registers for addition purposes. The result is to be stored in memory.

Registers: shows the value that's stored in each register, that the:

Value of 8th index of array A is stored in edx,
Value of h (stored in memory), is moved to eax,
addition is performed on the two registers and result is stored in eax.
Value in that register is moved into the memory, stored in the address of f and overwrites the value stored at f.

Memory: shows the values stored in an address:

0x555555558060 contains 0x37 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x555555558010 contains 0x16 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x5555555581d0 contains 0x4d 0x00 0x00 0x00 0x00 0x00 0x00 0x00

2-3_2.c:

The code written for 2-3_2 in C is shown in Fig 105. It uses two static variables: h and A. Using these two variables; an array is created, and addition is executed through its usage.

In our case, 200 is stored in the 8th index of A and is added to h. The result from that addition is stored into the 12th index of A.

```
(gdb) list
1       void main() {
2               static int h = 25;
3               static int A[100];
4               A[8] = 200;
5               A[12] = h + A[8];
6       }
(gdb)
```

Fig 105. Akram_2-3_2.c executed in Linux 64-32 bit

```
(gdb) disassemble
Dump of assembler code for function main:
=> 0x0000555555555129 <+0>:     endbr64
   0x000055555555512d <+4>:     push   %rbp
   0x000055555555512e <+5>:     mov    %rsp,%rbp
   0x0000555555555131 <+8>:     movl   $0xc8,0x2f25(%rip)        # 0x555555558060 <A.1913+32>
   0x000055555555513b <+18>:    mov    0x2f1f(%rip),%edx         # 0x555555558060 <A.1913+32>
   0x0000555555555141 <+24>:    mov    0x2ec9(%rip),%eax         # 0x555555558010 <h.1912>
   0x0000555555555147 <+30>:    add    %edx,%eax
   0x0000555555555149 <+32>:    mov    %eax,0x2f21(%rip)         # 0x555555558070 <A.1913+48>
   0x000055555555514f <+38>:    nop
   0x0000555555555150 <+39>:    pop    %rbp
   0x0000555555555151 <+40>:    retq
End of assembler dump.
(gdb) print /x $rsp
$1 = 0x7fffffffdf78
(gdb) print /x $rbp
$2 = 0x0
(gdb) info registers
rax            0x555555555129      93824992235817
rbx            0x555555555160      93824992235872
rcx            0x555555555160      93824992235872
rdx            0x7fffffffe078      140737488347256
rsi            0x7fffffffe068      140737488347240
rdi            0x1                 1
rbp            0x0                 0x0
rsp            0x7fffffffdf78      0x7fffffffdf78
r8             0x0                 0
r9             0x7ffff7fe0d50      140737354009936
r10            0x7ffff7ffcf68      140737354125160
r11            0x206               518
r12            0x555555555040      93824992235584
r13            0x7fffffffe060      140737488347232
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)
```

Fig 106. Akram_2-3_2.c GDB on Linux (Before Line 1)

Disassembly: this window shows that the code runs line 1 and that stack pointer and base pointer are set.

Registers: shows the value that's stored in each register, that the:

stack pointer is at 0x7fffffffdf78,
base pointer is at 0x0,
there are other registers containing random values.

```
(gdb) disassemble
Dump of assembler code for function main:
=> 0x0000555555555129 <+0>:     endbr64
   0x000055555555512d <+4>:     push   %rbp
   0x000055555555512e <+5>:     mov    %rsp,%rbp
   0x0000555555555131 <+8>:     movl   $0xc8,0x2f25(%rip)        # 0x555555558060 <A.1913+32>
   0x000055555555513b <+18>:    mov    0x2f1f(%rip),%edx        # 0x555555558060 <A.1913+32>
   0x0000555555555141 <+24>:    mov    0x2ec9(%rip),%eax        # 0x555555558010 <h.1912>
   0x0000555555555147 <+30>:    add    %edx,%eax
   0x0000555555555149 <+32>:    mov    %eax,0x2f21(%rip)        # 0x555555558070 <A.1913+48>
   0x000055555555514f <+38>:    nop
   0x0000555555555150 <+39>:    pop    %rbp
   0x0000555555555151 <+40>:    retq
End of assembler dump.
(gdb) x/8xb 0x555555558060
0x555555558060 <A.1913+32>:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
(gdb) info registers
rax            0x555555555129      93824992235817
rbx            0x555555555160      93824992235872
rcx            0x555555555160      93824992235872
rdx            0x7fffffffe078      140737488347256
rsi            0x7fffffffe068      140737488347240
rdi            0x1                 1
rbp            0x0                 0x0
rsp            0x7fffffffdf78      0x7fffffffdf78
r8             0x0                 0
r9             0x7ffff7fe0d50      140737354009936
r10            0x7ffff7ffcf68      140737354125160
r11            0x206               518
r12            0x555555555040      93824992235584
r13            0x7fffffffe060      140737488347232
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)
```

Fig 107. Akram_2-3_2.c GDB on Linux (Lines 1-4)

Disassembly: this window shows that the code runs lines 1 to 4 and that the 8th index of the array A is set to the value 200.

Registers: shows the value that's stored in each register, that the:

Values remain the same since lines 1 to 4 don't change anything.

Memory: shows the values stored in an address:

0x555555558060 contains 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00

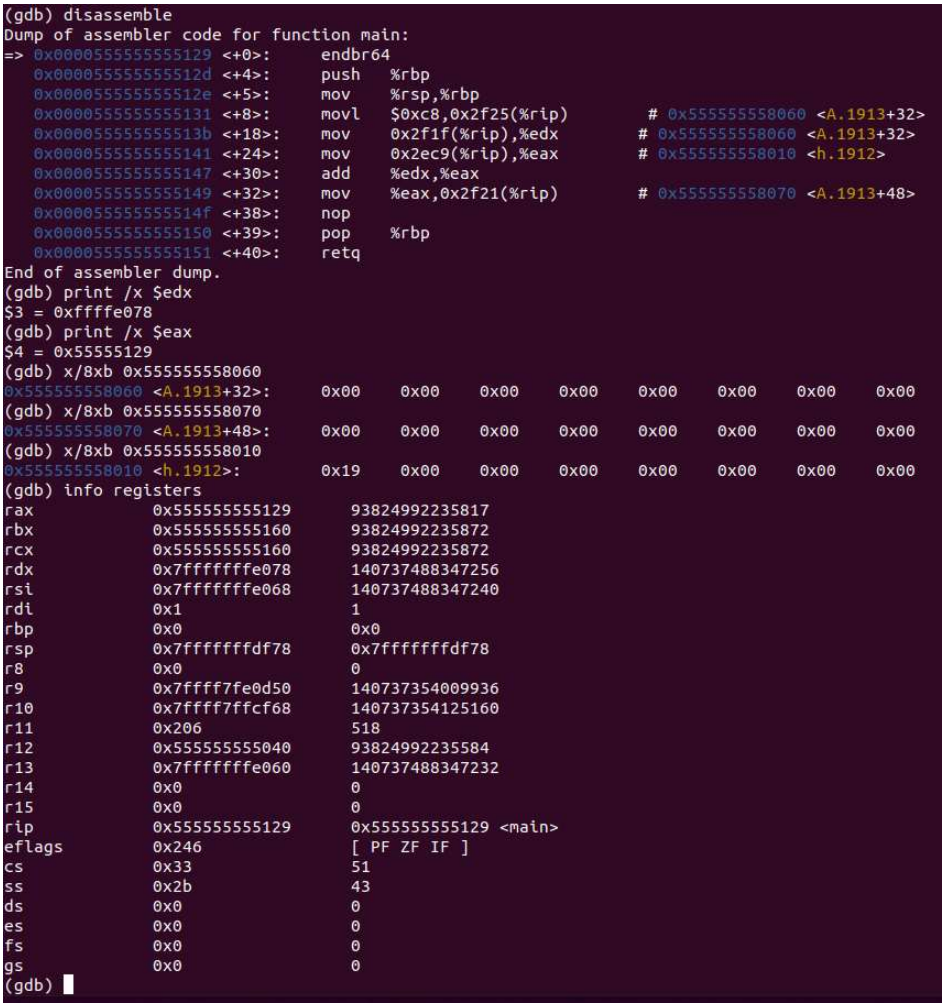

Fig 108. Akram_2-3_2.c GDB on Linux (Lines 5-6)

Disassembly: this window shows that the code runs lines 5 to 6 and that values are moved into registers for addition purposes. The result is to be stored in memory.

Registers: shows the value that's stored in each register, that the:

Value of 8th index of array A is stored in edx,
Value of h (stored in memory), is moved to eax,
addition is performed on the two registers and result is stored in eax.
Value in that register is moved into the memory, stored in the address of the 12th index of the array.

Memory: shows the values stored in an address:
0x555555558060 contains 0xc8 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x555555558010 contains 0x19 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x5555555581d0 contains 0xe1 0x00 0x00 0x00 0x00 0x00 0x00 0x00

2-5_1.c:

The code written for 2-5_1 in MIPS is shown in Fig 109. It uses two static variables: h and A. Using these two variables; an array A is created, and addition is executed through its usage.

In our case, 13 is stored in the 300th index of the array and is added to h. The result from that addition is stored back into the 300th index.

```
(gdb) list
1       void main() {
2               static int h = 20;
3               static int A[400];
4               A[300] = 13;
5               A[300] = h + A[300];
6       }
(gdb)
```

Fig 109. Akram_2-5_1.c executed in Linux 64-32 bit

```
(gdb) disassemble
Dump of assembler code for function main:
=> 0x0000555555555129 <+0>:     endbr64
   0x000055555555512d <+4>:     push   %rbp
   0x000055555555512e <+5>:     mov    %rsp,%rbp
   0x0000555555555131 <+8>:     movl   $0xd,0x33b5(%rip)        # 0x5555555584f0 <A.1913+1200>
   0x000055555555513b <+18>:    mov    0x33af(%rip),%edx        # 0x5555555584f0 <A.1913+1200>
   0x0000555555555141 <+24>:    mov    0x2ec9(%rip),%eax        # 0x555555558010 <h.1912>
   0x0000555555555147 <+30>:    add    %edx,%eax
   0x0000555555555149 <+32>:    mov    %eax,0x33a1(%rip)        # 0x5555555584f0 <A.1913+1200>
   0x000055555555514f <+38>:    nop
   0x0000555555555150 <+39>:    pop    %rbp
   0x0000555555555151 <+40>:    retq
End of assembler dump.
(gdb) print /x $rsp
$1 = 0x7fffffffdf98
(gdb) print /x $rbp
$2 = 0x0
(gdb) info registers
rax            0x555555555129      93824992235817
rbx            0x555555555160      93824992235872
rcx            0x555555555160      93824992235872
rdx            0x7fffffffe098      140737488347288
rsi            0x7fffffffe088      140737488347272
rdi            0x1                 1
rbp            0x0                 0x0
rsp            0x7fffffffdf98      0x7fffffffdf98
r8             0x0                 0
r9             0x7ffff7fe0d50      140737354009936
r10            0x7ffff7ffcf68      140737354125160
r11            0x206               518
r12            0x555555555040      93824992235584
r13            0x7fffffffe080      140737488347264
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)
```

Fig 110. Akram_2-5_1.c GDB on Linux (Before Line 1)

Disassembly: this window shows that the code runs line 1 and that stack pointer and base pointer are set.

Registers: shows the value that's stored in each register, that the:

stack pointer is at 0x7fffffffdf98,
base pointer is at 0x0,
there are other registers containing random values.

```
(gdb) disassemble
Dump of assembler code for function main:
=> 0x0000555555555129 <+0>:     endbr64
   0x000055555555512d <+4>:     push   %rbp
   0x000055555555512e <+5>:     mov    %rsp,%rbp
   0x0000555555555131 <+8>:     movl   $0xd,0x33b5(%rip)        # 0x5555555584f0 <A.1913+1200>
   0x000055555555513b <+18>:    mov    0x33af(%rip),%edx        # 0x5555555584f0 <A.1913+1200>
   0x0000555555555141 <+24>:    mov    0x2ec9(%rip),%eax        # 0x555555558010 <h.1912>
   0x0000555555555147 <+30>:    add    %edx,%eax
   0x0000555555555149 <+32>:    mov    %eax,0x33a1(%rip)        # 0x5555555584f0 <A.1913+1200>
   0x000055555555514f <+38>:    nop
   0x0000555555555150 <+39>:    pop    %rbp
   0x0000555555555151 <+40>:    retq
End of assembler dump.
(gdb) x/8xb 0x5555555584f0
0x5555555584f0 <A.1913+1200>:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
(gdb) info register
rax            0x555555555129      93824992235817
rbx            0x555555555160      93824992235872
rcx            0x555555555160      93824992235872
rdx            0x7fffffffe098      140737488347288
rsi            0x7fffffffe088      140737488347272
rdi            0x1                 1
rbp            0x0                 0x0
rsp            0x7fffffffdf98      0x7fffffffdf98
r8             0x0                 0
r9             0x7ffff7fe0d50      140737354009936
r10            0x7ffff7ffcf68      140737354125160
r11            0x206               518
r12            0x555555555040      93824992235584
r13            0x7fffffffe080      140737488347264
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)
```

Fig 111. Akram_2-5_1.c GDB on Linux (Lines 1-4)

Disassembly: this window shows that the code runs lines 1 to 4 and that the 300th index of the array A is set to the value 13.

Registers: shows the value that's stored in each register, that the:

Values remain the same since lines 1 to 4 don't change anything.

Memory: shows the values stored in an address:

0x5555555584f0 contains 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00

```
(gdb) disassemble
Dump of assembler code for function main:
=> 0x0000555555555129 <+0>:     endbr64
   0x000055555555512d <+4>:     push   %rbp
   0x000055555555512e <+5>:     mov    %rsp,%rbp
   0x0000555555555131 <+8>:     movl   $0xd,0x33b5(%rip)        # 0x5555555584f0 <A.1913+1200>
   0x000055555555513b <+18>:    mov    0x33af(%rip),%edx        # 0x5555555584f0 <A.1913+1200>
   0x0000555555555141 <+24>:    mov    0x2ec9(%rip),%eax        # 0x555555558010 <h.1912>
   0x0000555555555147 <+30>:    add    %edx,%eax
   0x0000555555555149 <+32>:    mov    %eax,0x33a1(%rip)        # 0x5555555584f0 <A.1913+1200>
   0x000055555555514f <+38>:    nop
   0x0000555555555150 <+39>:    pop    %rbp
   0x0000555555555151 <+40>:    retq
End of assembler dump.
(gdb) print /x $edx
$3 = 0xffffe098
(gdb) print /x $eax
$4 = 0x55555129
(gdb) x/8xb 0x5555555584f0
0x5555555584f0 <A.1913+1200>:   0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
(gdb) x/8xb 0x555555558010
0x555555558010 <h.1912>:        0x14    0x00    0x00    0x00    0x00    0x00    0x00    0x00
(gdb) info registers
rax            0x555555555129      93824992235817
rbx            0x555555555160      93824992235872
rcx            0x555555555160      93824992235872
rdx            0x7fffffffe098      140737488347288
rsi            0x7fffffffe088      140737488347272
rdi            0x1                 1
rbp            0x0                 0x0
rsp            0x7fffffffdf98      0x7fffffffdf98
r8             0x0                 0
r9             0x7ffff7fe0d50      140737354009936
r10            0x7ffff7ffcf68      140737354125160
r11            0x206               518
r12            0x555555555040      93824992235584
r13            0x7fffffffe080      140737488347264
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)
```

Fig 112. Akram_2-5_1.c GDB on Linux (Lines 5-6)

Disassembly: this window shows that the code runs lines 5 to 6 and that values are moved into registers for addition purposes. The result is to be stored in memory.

Registers: shows the value that's stored in each register, that the:

Value of 300th index of array A is stored in edx,
Value of h (stored in memory), is moved to eax,
addition is performed on the two registers and result is stored in eax.
Value in that register is moved into the memory, stored in the address of the 300th index of the array.

Memory: shows the values stored in an address:

0x5555555584f0 contains 0xc8 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x555555558010 contains 0x19 0x00 0x14 0x00 0x00 0x00 0x00 0x00

2-6_1.c:

The code written for 2-6_1 in C is shown in Fig 113. It stores values into the registers and uses these values to perform three bitwise operations: shift a register to the left, AND (compares two registers), and OR (also compares two registers).

```
(gdb) list
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;
(gdb)
```

Fig 113. Akram_2-6_1.c executed in Linux 64-32 bit

```
(gdb) disassemble
Dump of assembler code for function main:
=> 0x0000555555555129 <+0>:     endbr64
   0x000055555555512d <+4>:     push   %rbp
   0x000055555555512e <+5>:     mov    %rsp,%rbp
   0x0000555555555131 <+8>:     mov    0x2ed9(%rip),%eax        # 0x555555558010 <s0.1912>
   0x0000555555555137 <+14>:    shl    $0x4,%eax
   0x000055555555513a <+17>:    mov    %eax,0x2ee0(%rip)        # 0x555555558020 <t3.1915>
   0x0000555555555140 <+23>:    mov    0x2ece(%rip),%edx        # 0x555555558014 <t1.1913>
   0x0000555555555146 <+29>:    mov    0x2ecc(%rip),%eax        # 0x555555558018 <t2.1914>
   0x000055555555514c <+35>:    and    %edx,%eax
   0x000055555555514e <+37>:    mov    %eax,0x2ed0(%rip)        # 0x555555558024 <t0.1916>
   0x0000555555555154 <+43>:    mov    0x2eba(%rip),%edx        # 0x555555558014 <t1.1913>
   0x000055555555515a <+49>:    mov    0x2eb8(%rip),%eax        # 0x555555558018 <t2.1914>
   0x0000555555555160 <+55>:    or     %edx,%eax
   0x0000555555555162 <+57>:    mov    %eax,0x2ebc(%rip)        # 0x555555558024 <t0.1916>
   0x0000555555555168 <+63>:    mov    0x2ea6(%rip),%eax        # 0x555555558014 <t1.1913>
   0x000055555555516e <+69>:    not    %eax
   0x0000555555555170 <+71>:    mov    %eax,0x2eae(%rip)        # 0x555555558024 <t0.1916>
   0x0000555555555176 <+77>:    nop
   0x0000555555555177 <+78>:    pop    %rbp
   0x0000555555555178 <+79>:    retq
End of assembler dump.
(gdb) print /x $rsp
$1 = 0x7fffffffdf98
(gdb) print /x $rbp
$2 = 0x0
(gdb) info registers
rax            0x555555555129      93824992235817
rbx            0x555555555180      93824992235904
rcx            0x555555555180      93824992235904
rdx            0x7fffffffe098      140737488347288
rsi            0x7fffffffe088      140737488347272
rdi            0x1                 1
rbp            0x0                 0x0
rsp            0x7fffffffdf98      0x7fffffffdf98
r8             0x0                 0
r9             0x7ffff7fe0d50      140737354009936
r10            0x7ffff7ffcf68      140737354125160
r11            0x206               518
r12            0x555555555040      93824992235584
r13            0x7fffffffe080      140737488347264
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)
```

Fig 114. Akram_2-6_1.c GDB on Linux (Before Line 1)

Disassembly: this window shows that the code runs line 1 and that stack pointer and base pointer are set.

Registers: shows the value that's stored in each register, that the:

stack pointer is at 0x7fffffffdf98,
base pointer is at 0x0,
there are other registers containing random values.

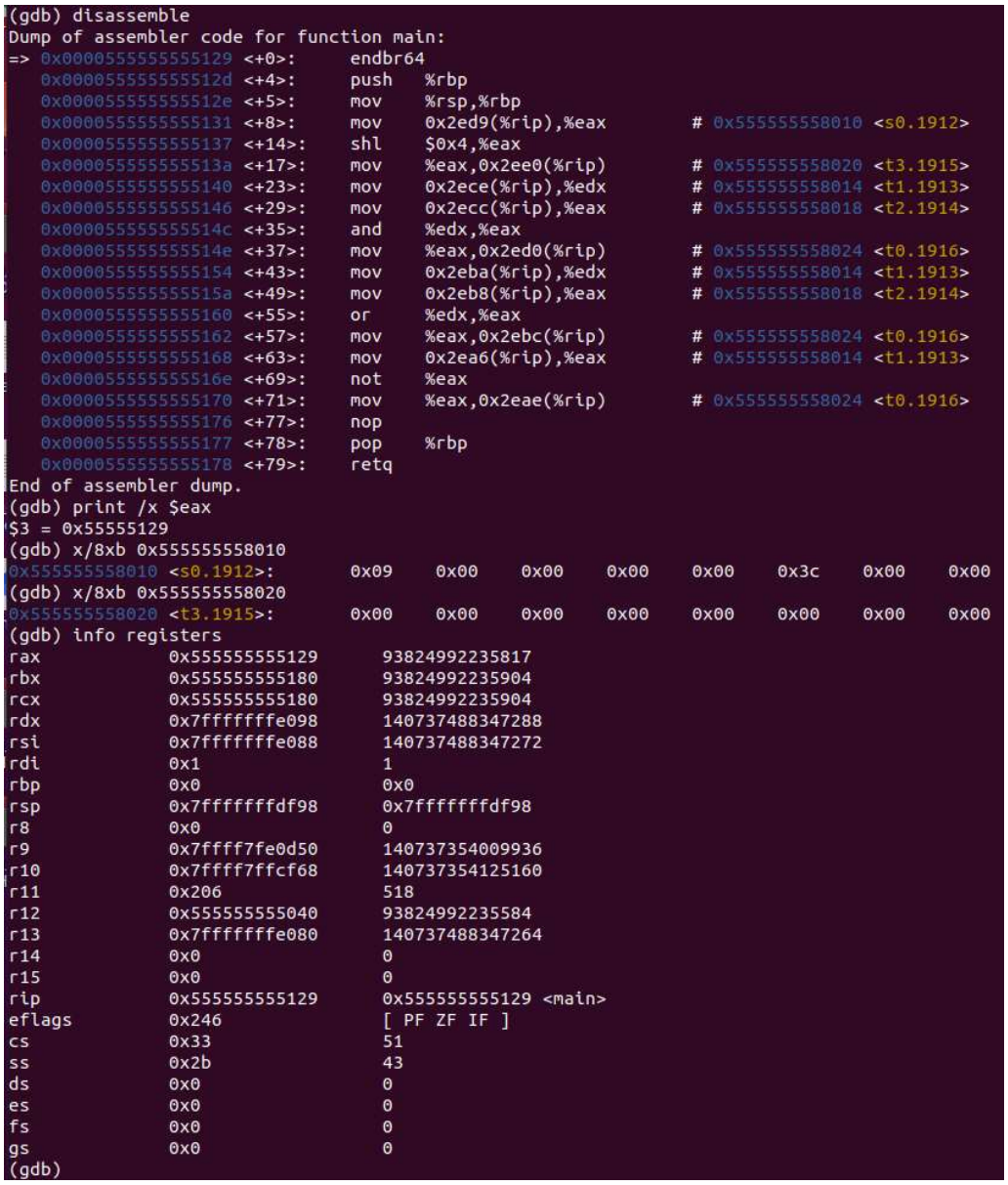

Fig 115. Akram_2-6_1.c GDB on Linux (Lines 1-6)

Disassembly: this window shows that the code runs lines 1 to 6 and that the value is moved to the register so can be shifted to the left by 4, and the result is to be stored in memory.

Registers: shows the value that's stored in each register, that the:

Value of s0 (stored in memory) is moved to eax,

Value is shifted to the left by 4, result is stored in the same register,

Value in that register is moved into the memory, stored in the address of t0 and overwrites the value stored at t0.

Memory: shows the values stored in an address:

0x555555558010 contains 0x09 0x00 0x00 0x00 0x00 0x3c 0x00 0x00
0x555555558020 contains 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00

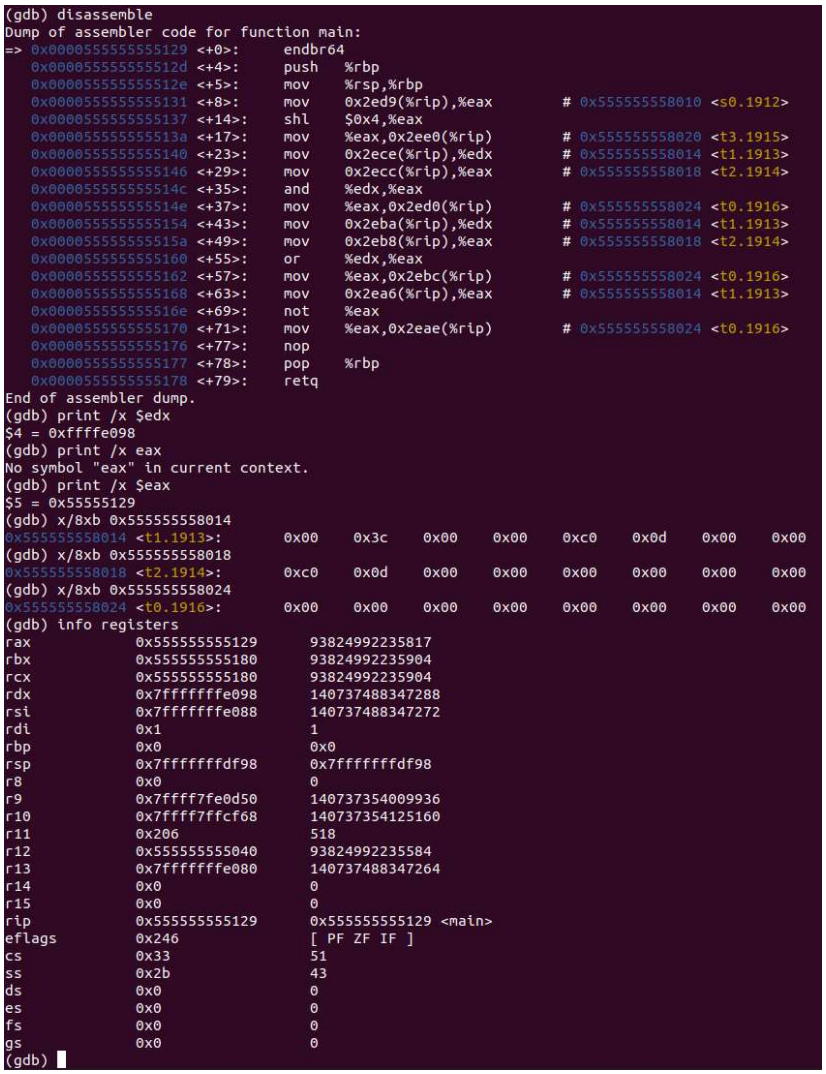

Fig 116. Akram_2-6_1.c GDB on Linux (Lines 7-8)

Disassembly: this window shows that the code runs lines 7 to 8 and that values are moved into registers for bitwise comparison purposes. The result is to be stored in memory.

Registers: shows the value that's stored in each register, that the:

Value of t1 (stored in memory), is moved to edx,
Value of t2 (stored in memory), is moved to eax,
Bitwise comparison is performed on the two registers and result is stored in eax.
Value in that register is moved into the memory, stored in the address of t0 and overwrites the value stored at t0.

Memory: shows the values stored in an address:

0x555555558014 contains 0x00 0x3c 0x00 0x00 0xc0 0x0d 0x00 0x00
0x555555558018 contains 0xc0 0x0d 0x00 0x00 0x00 0x00 0x00 0x00
0x555555558024 contains 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00

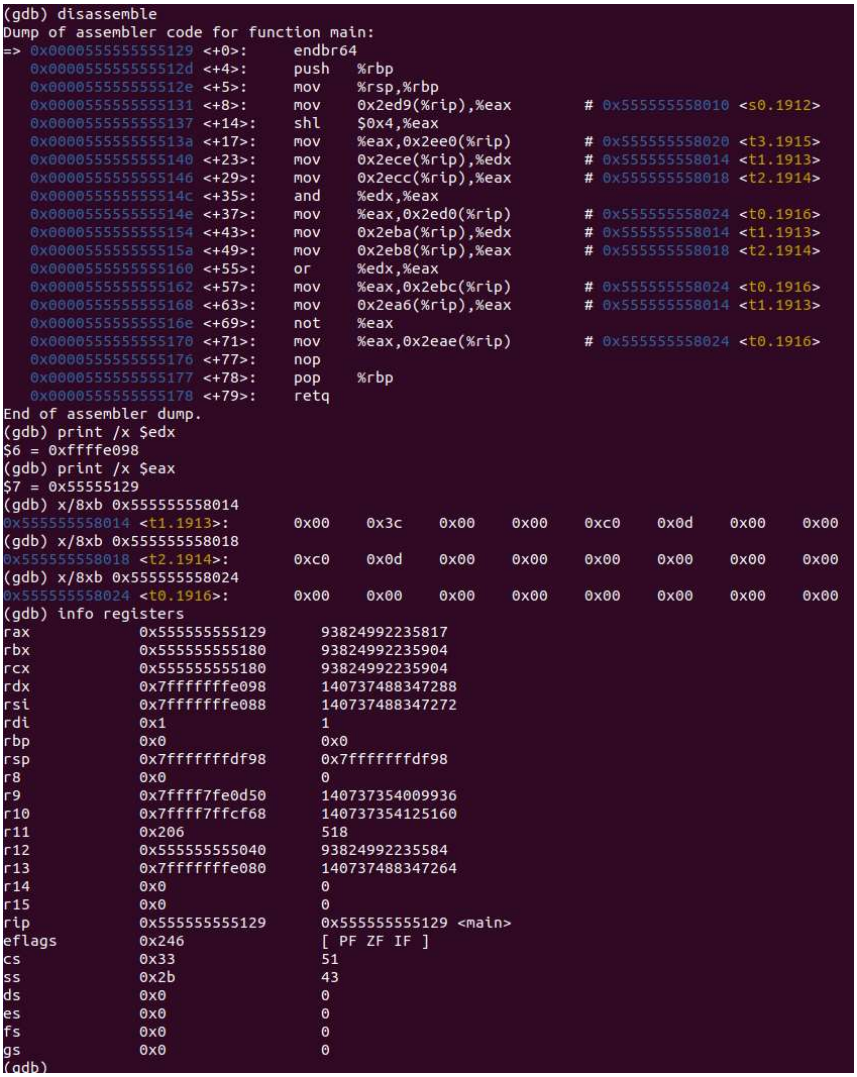

Fig 117. Akram_2-6_1.c GDB on Linux (Line 9)

Disassembly: this window shows that the code runs line 9 and that values are moved into registers for bitwise comparison purposes. The result is to be stored in memory.

Registers: shows the value that's stored in each register, that the:

Value of t1 (stored in memory), is moved to edx,
Value of t2 (stored in memory), is moved to eax,
Bitwise comparison is performed on the two registers and result is stored in eax.
Value in that register is moved into the memory, stored in the address of t0 and overwrites the value stored at t0.

Memory: shows the values stored in an address:

0x555555558014 contains 0x00 0x3c 0x00 0x00 0xc0 0x0d 0x00 0x00
0x555555558018 contains 0xc0 0x0d 0x00 0x00 0x00 0x00 0x00 0x00
0x555555558024 contains 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00

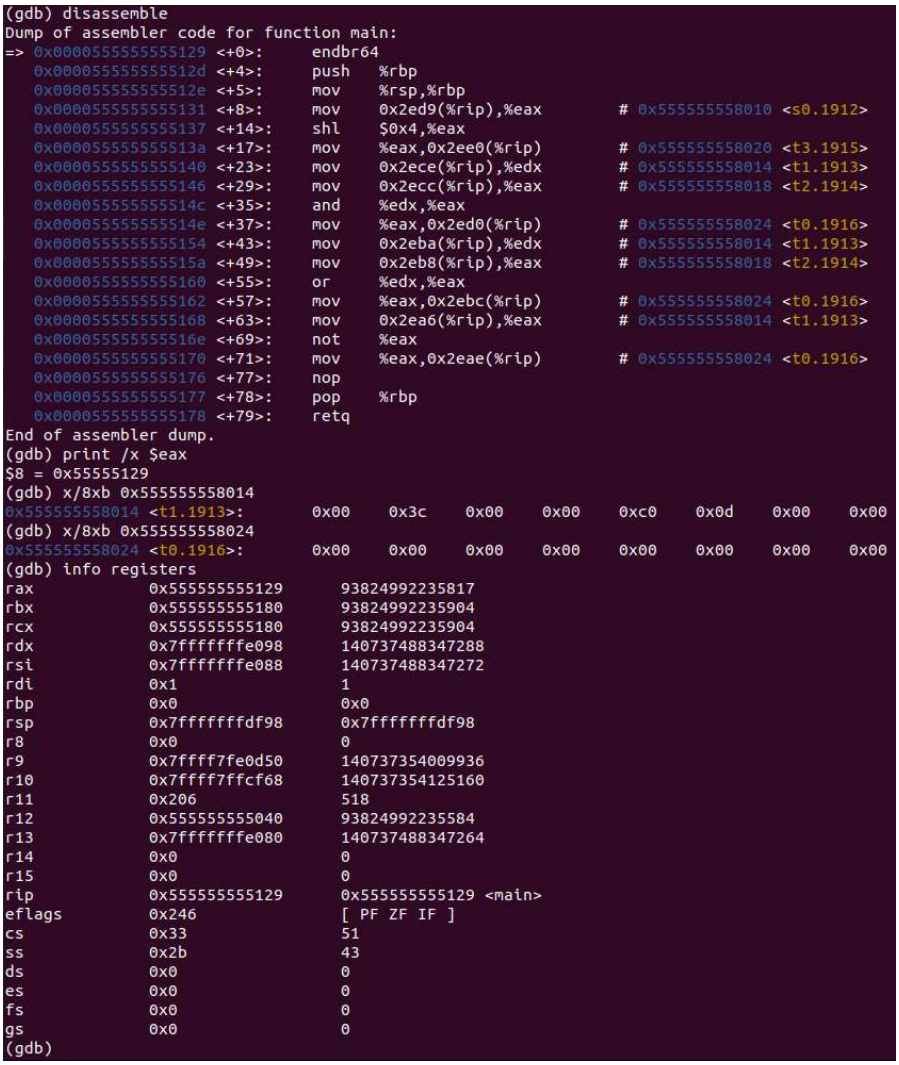

Fig 118. Akram_2-6_1.c GDB on Linux (Lines 10-11)

Disassembly: this window shows that the code runs lines 10 to 11 and that values are moved into registers for bitwise comparison purposes. The result is to be stored in memory.

Registers: shows the value that's stored in each register, that the:

Value of t1 (stored in memory), is moved to edx,
Value of t2 (stored in memory), is moved to eax,
Bitwise comparison is performed on the two registers and result is stored in eax.
Value in that register is moved into the memory, stored in the address of t0 and overwrites the value stored at t0.

Memory: shows the values stored in an address:

0x555555558014 contains 0x00 0x3c 0x00 0x00 0xc0 0x0d 0x00 0x00
0x555555558024 contains 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00

2-7_1.c:

The code written for 2-7_1 in C is shown in Fig 119. It uses five static variables: f, g, h, i, and j. Using these five variables; conditions are set up and if the variables satisfy those conditions; arithmetic operations are performed.

If i and j, then g and h will be added, and the result will be stored in f,
else,
g and h will be subtracted, and the result will be stored in f.

```
(gdb) list
1       void main() {
2               static int f = 0;
3               static int g = 50;
4               static int h = 40;
5               static int i = 30;
6               static int j = 20;
7               if (i == j) {
8                       f = g + h;
9               }
10              else {
(gdb)
```

Fig 119. Akram_2-7_1.c executed in Linux 64-32 bit

```
(gdb) disassemble
Dump of assembler code for function main:
=> 0x0000555555555129 <+0>:     endbr64
   0x000055555555512d <+4>:     push   %rbp
   0x000055555555512e <+5>:     mov    %rsp,%rbp
   0x0000555555555131 <+8>:     mov    0x2ed9(%rip),%edx        # 0x555555558010 <i.1915>
   0x0000555555555137 <+14>:    mov    0x2ed7(%rip),%eax        # 0x555555558014 <j.1916>
   0x000055555555513d <+20>:    cmp    %eax,%edx
   0x000055555555513f <+22>:    jne    0x555555555157 <main+46>
   0x0000555555555141 <+24>:    mov    0x2ed1(%rip),%edx        # 0x555555558018 <g.1913>
   0x0000555555555147 <+30>:    mov    0x2ecf(%rip),%eax        # 0x55555555801c <h.1914>
   0x000055555555514d <+36>:    add    %edx,%eax
   0x000055555555514f <+38>:    mov    %eax,0x2ecf(%rip)        # 0x555555558024 <f.1912>
   0x0000555555555155 <+44>:    jmp    0x55555555516d <main+68>
   0x0000555555555157 <+46>:    mov    0x2ebb(%rip),%edx        # 0x555555558018 <g.1913>
   0x000055555555515d <+52>:    mov    0x2eb9(%rip),%eax        # 0x55555555801c <h.1914>
   0x0000555555555163 <+58>:    sub    %eax,%edx
   0x0000555555555165 <+60>:    mov    %edx,%eax
   0x0000555555555167 <+62>:    mov    %eax,0x2eb7(%rip)        # 0x555555558024 <f.1912>
   0x000055555555516d <+68>:    nop
   0x000055555555516e <+69>:    pop    %rbp
   0x000055555555516f <+70>:    retq
End of assembler dump.
(gdb) print /x $rsp
$1 = 0x7fffffffdf78
(gdb) print /x $rbo
$2 = 0x0
(gdb) info registers
rax            0x555555555129      93824992235817
rbx            0x555555555170      93824992235888
rcx            0x555555555170      93824992235888
rdx            0x7fffffffe078      140737488347256
rsi            0x7fffffffe068      140737488347240
rdi            0x1                 1
rbp            0x0                 0x0
rsp            0x7fffffffdf78      0x7fffffffdf78
r8             0x0                 0
r9             0x7ffff7fe0d50      140737354009936
r10            0x7ffff7ffcf68      140737354125160
r11            0x206               518
r12            0x555555555040      93824992235584
r13            0x7fffffffe060      140737488347232
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)
```

Fig 120. Akram_2-7_1.c GDB on Linux (Before Line 1)

Disassembly: this window shows that the code runs line 1 and that stack pointer and base pointer are set.

Registers: shows the value that's stored in each register, that the:

stack pointer is at 0x7fffffffdf78,
base pointer is at 0x0,
there are other registers containing random values.

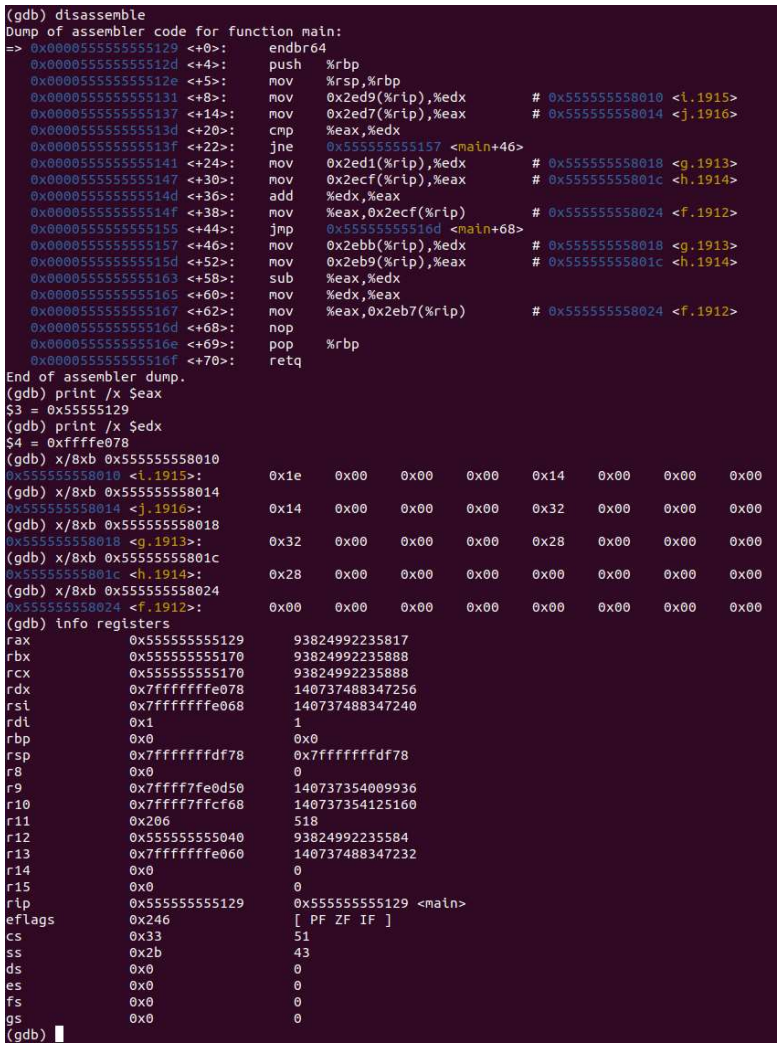

Fig 121. Akram_2-7_1.c GDB on Linux (Lines 1-9)

Disassembly: this window shows that the code runs lines 1 to 9 and that the value is moved to the register so can checked if they satisfy certain conditions. If they do, it'll run that, otherwise jump to another part of the program.

Registers: shows the value that's stored in each register, that the:

Value of i (stored in memory) is moved to edx,
Value of j (stored in memory) is moved to eax,
The two registers are compared, condition is not satisfied, so program jumps to another part.

Memory: shows the values stored in an address:

0x555555558010 contains 0x1e 0x00 0x00 0x00 0x14 0x00 0x00 0x00
0x555555558014 contains 0x14 0x00 0x00 0x00 0x32 0x00 0x00 0x00
0x555555558018 contains 0x32 0x00 0x00 0x00 0x28 0x00 0x00 0x00
0x55555555801c contains 0x28 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x555555558024 contains 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00

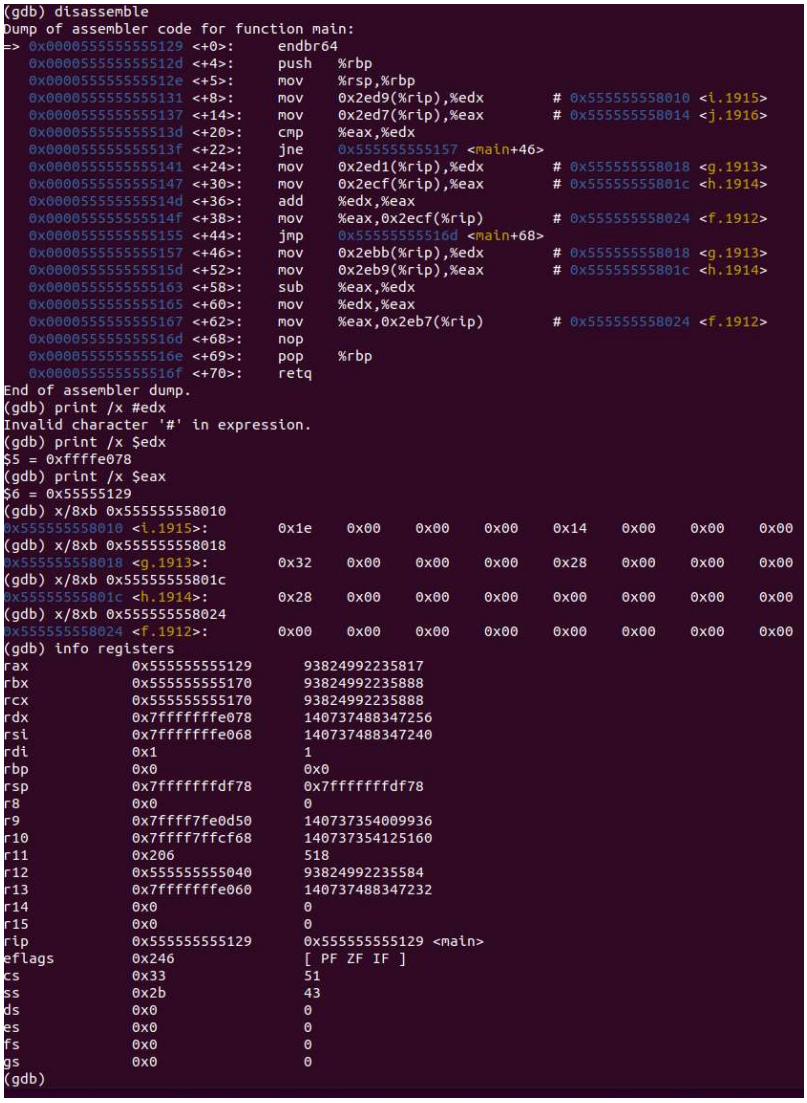

Fig 122. Akram_2-7_1.c GDB on Linux (Lines 10-13)

Disassembly: this window shows that the code runs lines 10 to 13 and that the values is moved to the register so subtraction purposes. The result is to be stored in memory.

Registers: shows the value that's stored in each register, that the:

Value of g (stored in memory) is moved to edx,
Value of h (stored in memory) is moved to eax,
Subtraction of the two registers is performed, the result is stored in edx,
Value in that register is moved into the memory, stored in the address of f and overwrites the value stored at f.

Memory: shows the values stored in an address:

0x555555558018 contains 0x32 0x00 0x00 0x00 0x28 0x00 0x00 0x00
0x55555555801c contains 0x28 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x555555558024 contains 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00

2-7_2.c:

The code written for 2-7_2 in C is shown in Fig 123. It uses three static variables: i, k, and save. Using these three variables; conditions are set up and if the variables satisfy those conditions; loop operations are performed.

In our case and array called save is generated and the value 10 is saved into the 5^{th} index. The loop is run and checked if the value 10 is saved in the i^{th} index of that array:

If it is: increment i by 1,

Else,

Exit the loop.

```
(gdb) list
1       void main() {
2               static int i = 5;
3               static int k = 10;
4               static int save[100];
5               save[5] = 10;
6               while (save[i] == k) {
7                       i++;
8               }
9       }
(gdb)
```

Fig 123. Akram_2-7_2.c executed in Linux 64-32 bit

```
(gdb) disassemble
Dump of assembler code for function main:
=> 0x0000555555555129 <+0>:     endbr64
   0x000055555555512d <+4>:     push   %rbp
   0x000055555555512e <+5>:     mov    %rsp,%rbp
   0x0000555555555131 <+8>:     movl   $0xa,0x2f19(%rip)        # 0x555555558054 <save.1914+20>
   0x000055555555513b <+18>:    jmp    0x55555555514c <main+35>
   0x000055555555513d <+20>:    mov    0x2ecd(%rip),%eax        # 0x555555558010 <i.1912>
   0x0000555555555143 <+26>:    add    $0x1,%eax
   0x0000555555555146 <+29>:    mov    %eax,0x2ec4(%rip)        # 0x555555558010 <i.1912>
   0x000055555555514c <+35>:    mov    0x2ebe(%rip),%eax        # 0x555555558010 <i.1912>
   0x0000555555555152 <+41>:    cltq
   0x0000555555555154 <+43>:    lea    0x0(,%rax,4),%rdx
   0x000055555555515c <+51>:    lea    0x2edd(%rip),%rax        # 0x555555558040 <save.1914>
   0x0000555555555163 <+58>:    mov    (%rdx,%rax,1),%edx
   0x0000555555555166 <+61>:    mov    0x2ea8(%rip),%eax        # 0x555555558014 <k.1913>
   0x000055555555516c <+67>:    cmp    %eax,%edx
   0x000055555555516e <+69>:    je     0x55555555513d <main+20>
   0x0000555555555170 <+71>:    nop
   0x0000555555555171 <+72>:    nop
   0x0000555555555172 <+73>:    pop    %rbp
   0x0000555555555173 <+74>:    retq
End of assembler dump.
(gdb) print /x $rsp
$1 = 0x7fffffffdf78
(gdb) print /x $rbp
$2 = 0x0
(gdb) info registers
rax            0x555555555129      93824992235817
rbx            0x555555555180      93824992235904
rcx            0x555555555180      93824992235904
rdx            0x7fffffffe078      140737488347256
rsi            0x7fffffffe068      140737488347240
rdi            0x1                 1
rbp            0x0                 0x0
rsp            0x7fffffffdf78      0x7fffffffdf78
r8             0x0                 0
r9             0x7ffff7fe0d50      140737354009936
r10            0x7ffff7ffcf68      140737354125160
r11            0x206               518
r12            0x555555555040      93824992235584
r13            0x7fffffffe060      140737488347232
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)
```

Fig 124. Akram_2-7_2.c GDB on Linux (Before Line 1)

Disassembly: this window shows that the code runs line 1 and that stack pointer and base pointer are set.

Registers: shows the value that's stored in each register, that the:

stack pointer is at 0x7fffffffdf78,
base pointer is at 0x0,
there are other registers containing random values.

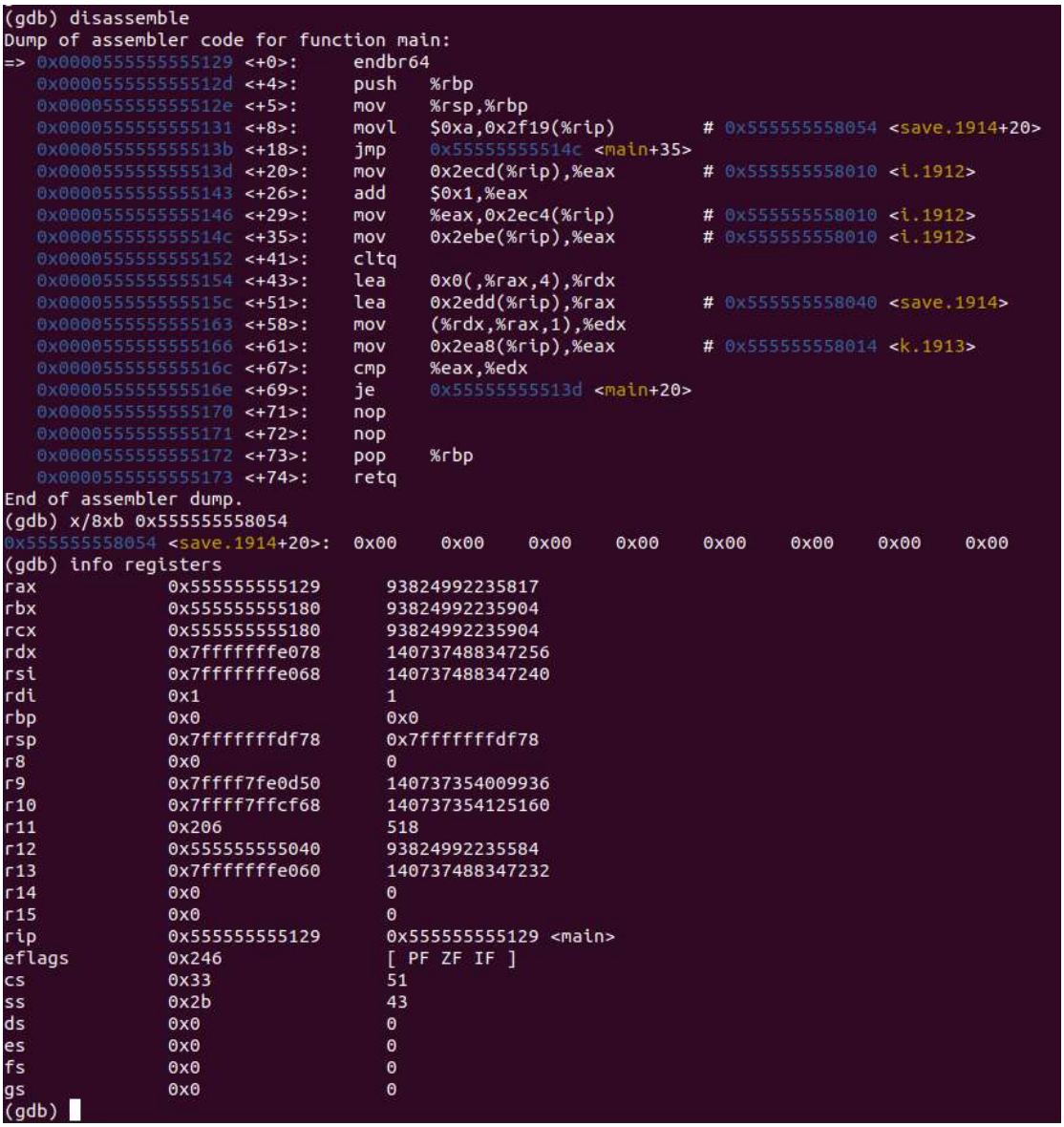

Fig 125. Akram_2-7_2.c GDB on Linux (Lines 1-5)

Disassembly: this window shows that the code runs lines 1 to 5 and that the 5th index of the array is set equal to the value 10.

Registers: remains the same because lines 1 to 5 don't store anything into them or change any values.

Memory: shows the values stored in an address:

0x555555558054 contains 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00

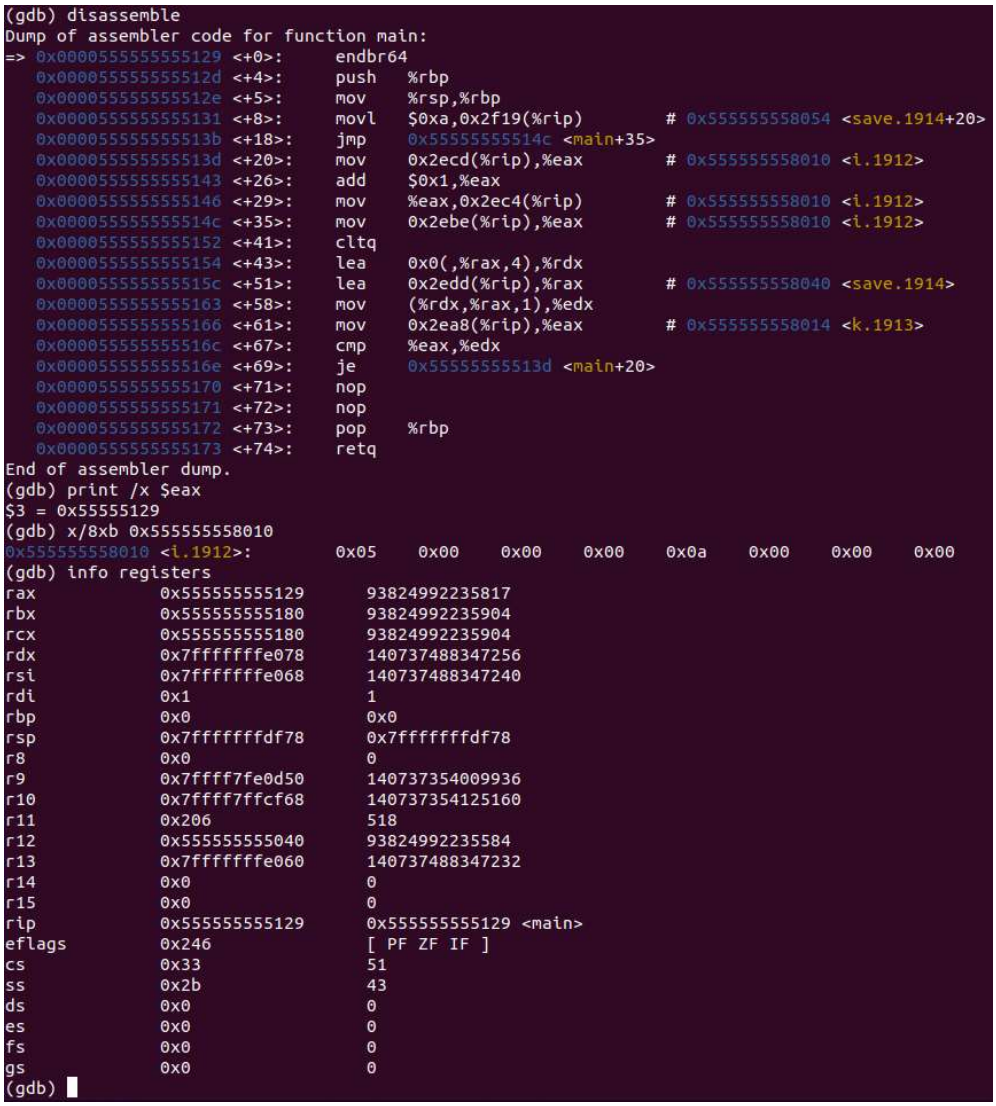

Fig 126. Akram_2-7_2.c GDB on Linux (Lines 6-8)

Disassembly: this window shows that the code runs lines 6 to 8 and that values are moved into registers to check if they meet certain conditions. If they do, then execute, otherwise jump to another part of the program.

Registers: shows the value that's stored in each register, that the:

Value of i^{th} index of array A is stored in edx,
Value of k (stored in memory), is moved to edx,

The two registers are compared, if they are equal, then jump to another part of the disassemble and increment i by 1. It does this by storing the value of i in a register and adding 1 to it and then storing it back to the address of i.

Memory: shows the values stored in an address:
0x555555558010 contains 0x05 0x00 0x00 0x00 0x0a 0x00 0x00 0x00

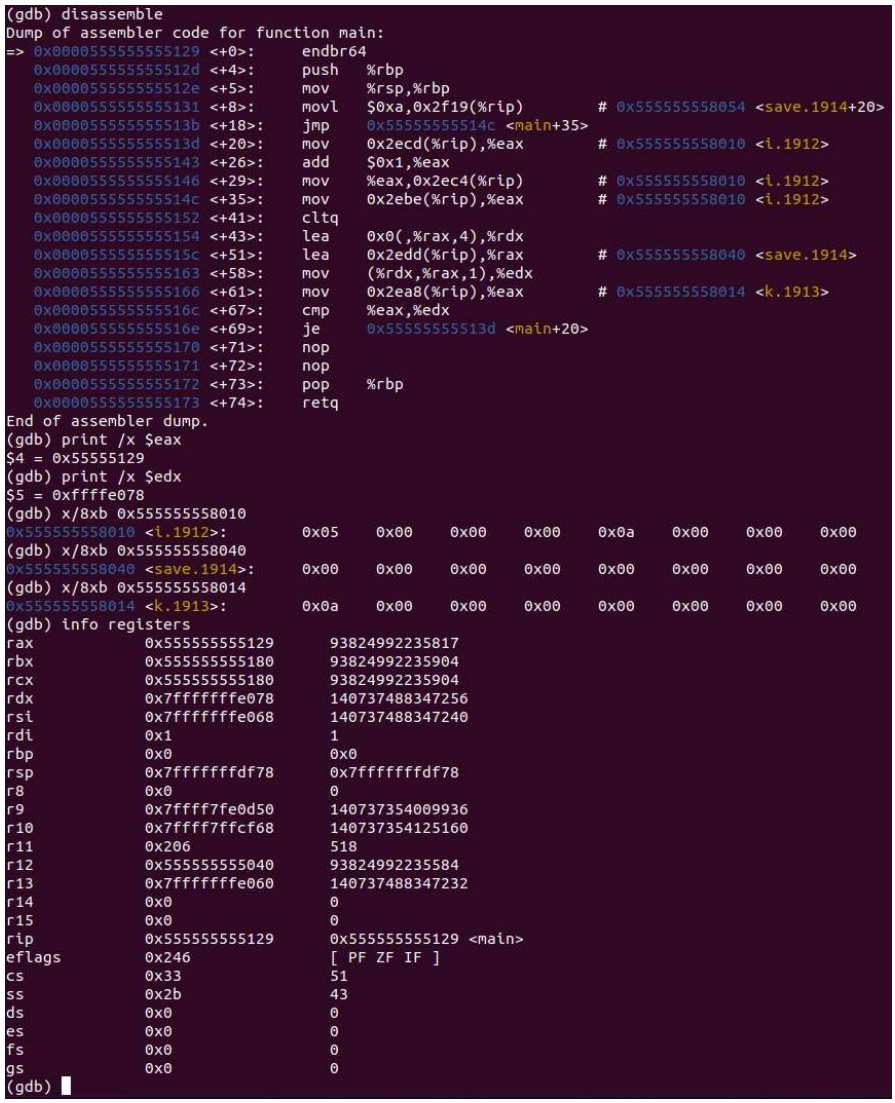

Fig 126. Akram_2-7_2.c GDB on Linux (Lines 6-9)

Disassembly: this window shows that the code runs lines 6 to 9 and that values are moved into registers to check if they meet certain conditions. If they do, then execute, otherwise jump to another part of the program. In this case, it did not satisfy the condition, so it jumped to the end.

Registers: shows the value that's stored in each register, that the:

Value of i^{th} index of array A is stored in edx,
Value of k (stored in memory), is moved to edx,
The two registers are compared, if they are equal, since they're not, the program jumps to the end of the code.

Memory: shows the values stored in an address:
0x555555558010 contains 0x05 0x00 0x00 0x00 0x0a 0x00 0x00 0x00
0x555555558040 contains 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x555555558014 contains 0x0a 0x00 0x00 0x00 0x00 0x00 0x00 0x00

2-8_1.c:

The code written for 2-8_1 in C is shown in Fig 127. It uses three static variables: a, b, and c. Using these three variables; it'll enter a function called myadd, which will take two parameters a and b and return their sum by storing the result into c.

```
(gdb) list
1       int myadd(int a, int b) {
2               int c = a + b;
3               return c;
4       }
5       void main() {
6               static int a = 4;
7               static int b = 15;
8               static int c = 0;
9               c = myadd(a, b);
10      }
(gdb)
```

Fig 128. Akram_2-8_1.c executed in Linux 64-32 bit

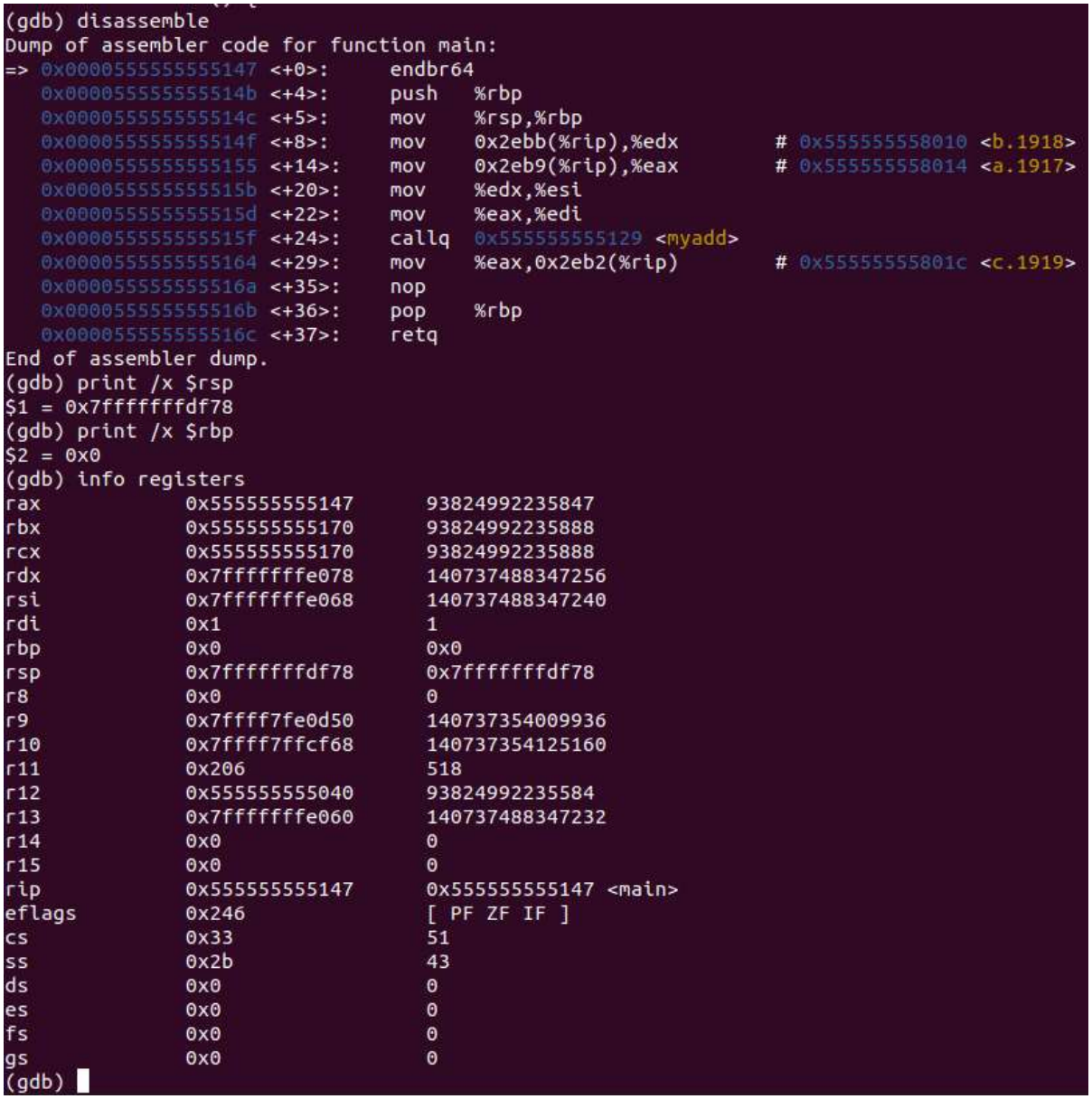

Fig 129. Akram_2-7_2.c GDB on Linux (Before Line 5)

Disassembly: this window shows that the code starts at line 5 and that stack pointer and base pointer are set.

Registers: shows the value that's stored in each register, that the:

stack pointer is at 0x7fffffffdf78,
base pointer is at 0x0,
there are other registers containing random values.

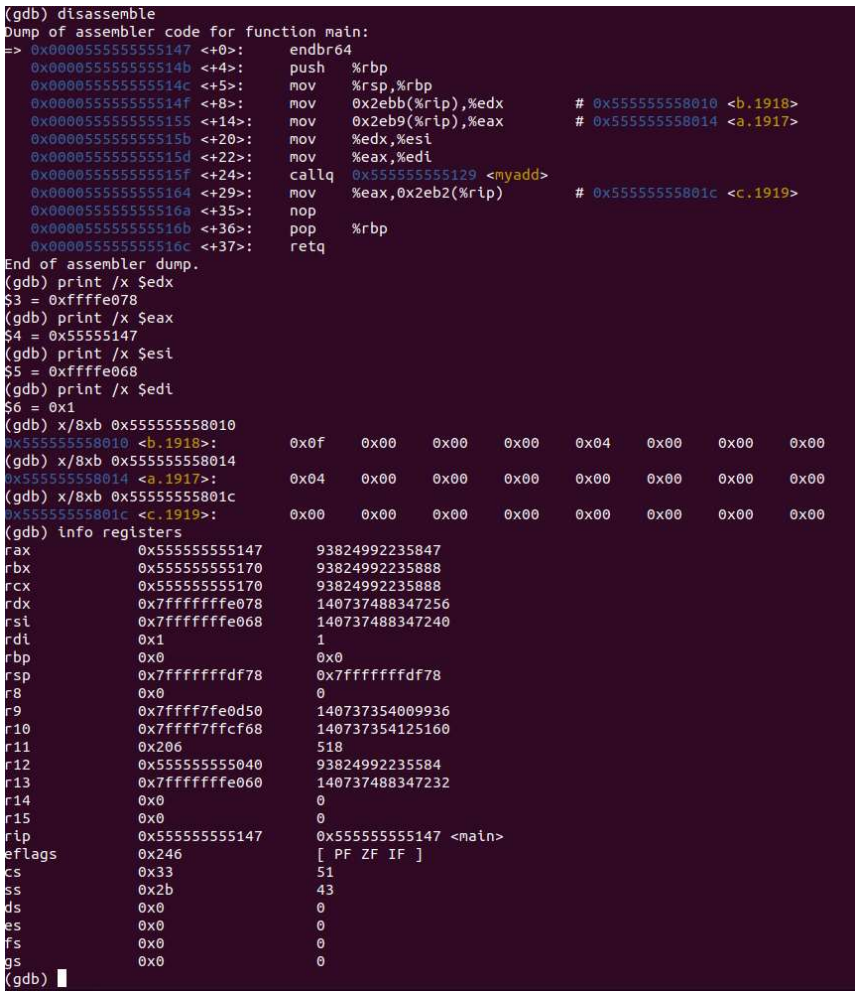

Fig 130. Akram_2-8_1.c GDB on Linux (Line 1-10)

Disassembly: this window shows that the code runs lines 1 to 10 and that values are moved into registers for function call purposes. The return value is to be stored in memory.

Registers: shows the value that's stored in each register, that the:

Value of b (stored in memory), is moved to edx,
Value of a (stored in memory), is moved to eax,
esi is set equal to the same value as edx,
edi is set equal to the same value as eax, (to be in the registers for the function call), function call is made,
Value in that return register is moved into the memory, stored in the address of c and overwrites the value stored at c.

Memory: shows the values stored in an address:

0x555555558010 contains 0x0f 0x00 0x00 0x00 0x04 0x00 0x00 0x00
0x555555558014 contains 0x04 0x00 0x14 0x00 0x00 0x00 0x00 0x00
0x55555555801c contains 0x19 0x00 0x14 0x00 0x00 0x00 0x00 0x00

Conclusion:

Add read.me

This take-home test taught me to compare, contrast and understand:

- MIPS instruction set architecture,
- Intel x86 ISA using Windows VS 32-bit compiler and debugger,
- and Intel x886 64-bit GCC and GDB compiler.

We used **MARS simulator to run MIPS code** to see the registers and data segments changing in each line of code:

- the loading of values into the registers
- and their computation and loading into the data segments.

We used **Visual Studio to run C code** and see the registers and memories changing through each disassembly:

- We saw each value moving into the registers
- and exactly how it moves back into the memory, changing the values in the memory.

We used Linux to run C code to see the registers and memories change in each disassemble:

- We saw each value moving into the registers
- and exactly how it moves back into the memory, changing the values in the memory.

As we ran several sample codes across these three platforms, we notice similarities and differences in these platforms. **Intel x86 32-bit ISA and Intel x86 64-bit ISA both use little endian. MIPS uses big endian.** All three platforms use their registers.

In conclusion, although we're running the same form of code on different platforms, the disassembly and their inner workings within their respective machine, all function in a similar manner albeit noticeable minor differences.