fkamal000@citymail.cuny.edu

Faheem Kamal

Professor Gertner

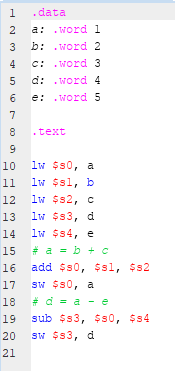
Take HOME TEST 1

CS342 and CS343

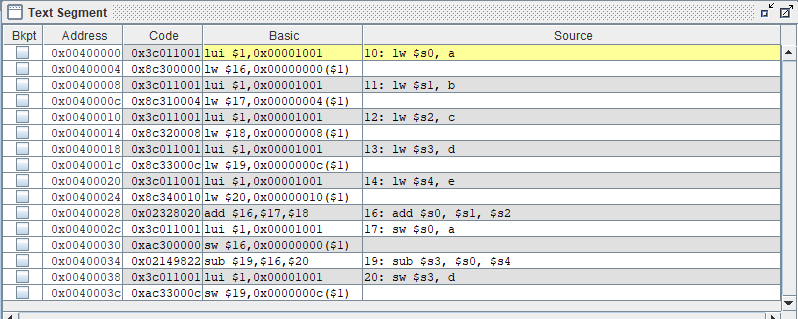
**PART I – MIPS**

* MIPS is Big Endian [Most Significant Bit is stored First]. MARS MIPS runs MIPS. Static variables are stored in memory and are accessed by their address in Memory. Local variables are stored at negative offset with respect to the Stack for all cases below.

**Problem 2.2-1**

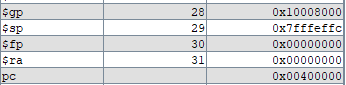


***Program in MIPS***



***Text Segment in MIPS***

* Program’s first step: The Stack Pointer ($sp) points to the address on top of the stack segment 0x7fffeffc. The Program Counter (pc) points to the starting address of the program. In this case, 0x00400000. This counter will change as it goes through the entire program.



* Each variable from A to E is declared and stored in an address within the Text Segment with the offset of 4.
  + a: .word 1 //declared as 1
  + b: .word 2 //declared as 2
  + c: .word 3 // declare c as 3
  + d: .word 4 // declare d as 4
  + e: .word 5 // declare e as 5
* The LUI (Load Upper Intermediate) instruction points to the address of 0x00001001 which is represented as 0x10010000 since MIPS is Big Endian. Furthermore, this address is loaded into the $at Register which is reserved for use in pseudoinstructions.



* The variables are then loaded into Saved Registers $s1 to $s5 for the 5 variables. With each load word operation, the program will repeatedly go to the starting address of the Text Segment before accessing the address of each variable where its value is stored. This is done 5 times. [lw ($s0 -> $s5), (a -> e)]











* After that, the values stored in Saved Registers $s1 and $s2 are added and then stored in SavedRegister $s0. [add $s0, $s1, $s2]



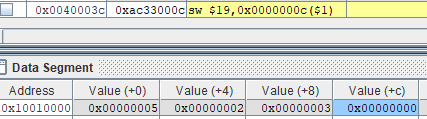
* We then write the new value of $s0 into the address of where the value of a is stored. Hence, overwriting a. [sw $s0, a]



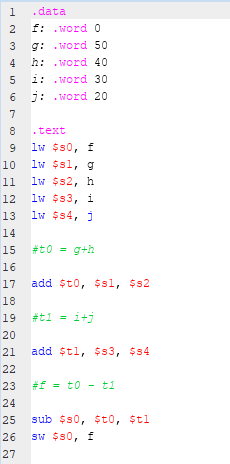
* The next instruction subtracts the values in Saved Registers $s4 and $s0 and then its stored in Saved Register $s3. [sub $s3, $s0, $s4]



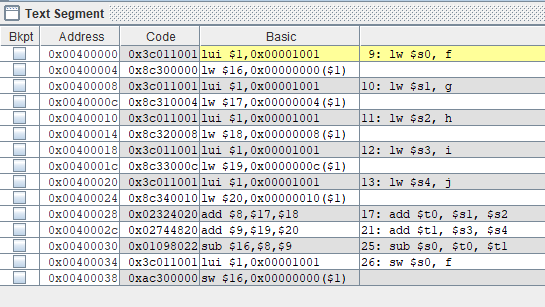
* We then write the new value of $s3 into the address of where the value of d is stored. Hence, overwriting d. [sw $s3, d]



**Problem 2.2-2**

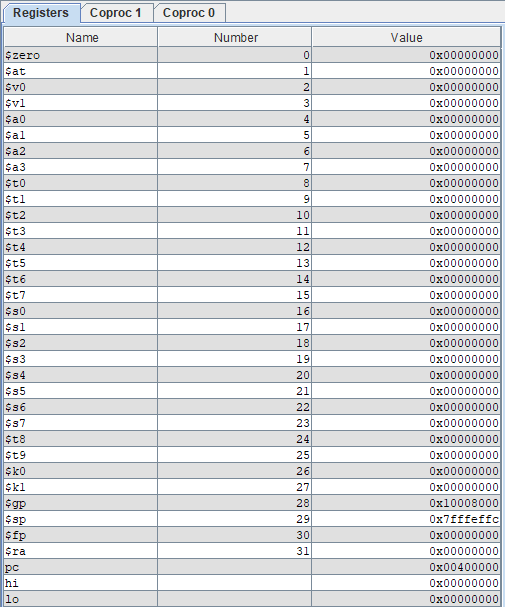


***Program in MIPS***



***Text Segment in MIPS***

* Program’s first step: The Stack Pointer ($sp) points to the address on top of the stack segment 0x7fffeffc. The Program Counter (pc) points to the starting address of the program. In this case, 0x00400000. This counter will change as it goes through the entire program.



***Registers in MIPS***

* Each variable from F to J is declared and stored in an address within the Text Segment with the offset of 4.
  + f: .word 0 //declared f as 0
  + g: .word 50 //declared g as 50
  + h: .word 40 // declare h as 40
  + i: .word 30 // declare i as 30
  + j: .word 20 // declare j as 20
* The LUI (Load Upper Intermediate) instruction points to the address of 0x00001001 which is represented as 0x10010000 since MIPS is Big Endian. Furthermore, this address is loaded into the $at Register which is reserved for use in pseudoinstructions.





* The variables are then loaded into Saved Registers $s1 to $s5 for the 5 variables. With each load word operation, the program will repeatedly go to the starting address of the Text Segment before accessing the address of each variable where its value is stored. This is done 5 times. [lw ($s0 -> $s5), (f -> j)]











* After that, the values stored in Saved Registers $s1 and $s2 are added and then stored in the Temporary Register $t0. [add $t0, $s1, $s2]



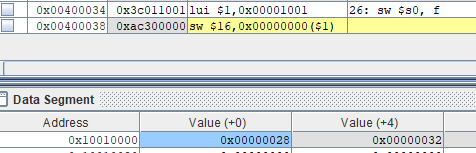
* The next instruction adds the values stored in Saved Registers $s3 and $s4 and then its stored in Temporary Register $t1. [add $t1, $s3, $s4]



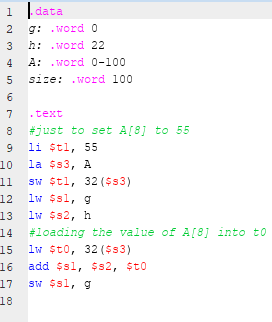
* We then subtract the values stored in Temporary Registers $t0 and $t1 and then its stored in the Saved Register $s0. [sub $s0, $t0, $t1]



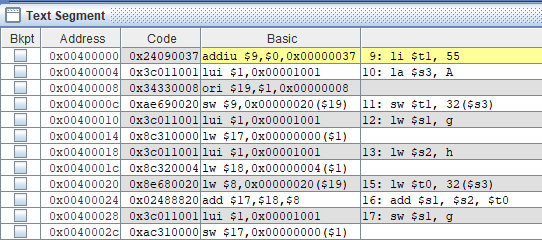
* We then write the new value of $s0 into the address of where the value of f is stored. Hence, overwriting f. [sw $s0, f]



**Problem 2.3-1**

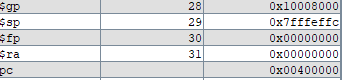


***Program in MIPS***



***Text Segment in MIPS***

* Program’s first step: The Stack Pointer ($sp) points to the address on top of the stack segment 0x7fffeffc. The Program Counter (pc) points to the starting address of the program. In this case, 0x00400000. This counter will change as it goes through the entire program. We also have a Group Point ($gp) pointing to the address 0x10008000.



* Each variable from g to size is declared and stored in an address within the Text Segment with the offset of 4.
  + g: .word 0 //declared g as 0
  + h: .word 22 //declared h as 22
  + A: .word 0-100 // declared A as an array that takes 0-100 (memory allocated that can store 0-100)
  + size: .word 100 // declare size as 100
* The LI (Load Immediate) instruction to store a value of 55 in Temporary Register $t1 is executed. This pseudoinstruction is translated as ADDIU (Add Immediate Unsigned) which will store 55 as a hexadecimal in the Temporary Register $t1,represented as $9. [li $t1, 55 -> addiu $9, $0, 0x00000037]



* Then, the address of A is loaded into the Save Register $s3. [la $s3, A]



* The LUI (Load Upper Intermediate) instruction points to the address of 0x00001001 which is represented as 0x10010000 since MIPS is Big Endian. Furthermore, this address is loaded into the $at Register which is reserved for use in pseudoinstructions.



* The ORI (Or Immediate) instruction loads the value of 0x00000008 (8) to the register $19 which is Save Register $s3. This register would take the value of LUI with the offset of 8.
* The next instruction will then store the value in $9 or Temporary Register $t1 in 0x00000020($19). The value is stored in 0x10010020 in the Data Segment with the offset of 8 to store 0x00000037. [sw $t1, 32($s3)]
* Afterwards, g and h variables are loaded into the respective Save Registers $s1 and $s2. [lw $s1, g] -> [lw $s2, h] Since g is 0 bytes, 0x00000000 is the value of $s1 and h which is 22 bytes is 0x00000016 in $s2.





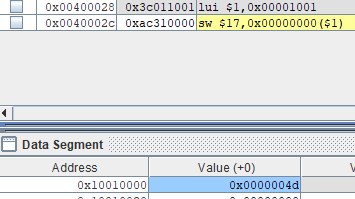
* Then, we load the value saved in Temporary Register $t1 into the Temporary Register $t0. [lw $t0, 32($s3)]



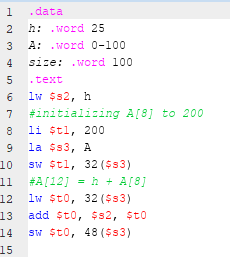
* Next, the values in Save Registers $s2 and $s0 are added and into Save Register $s1. [add $s1, $s2, $t0]



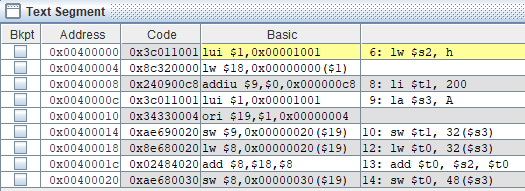
* Then, the value in Save Register $s1 is stored in the variable of g. Hence, overwriting the original value. [sw $s1, g]



**Problem 2.3-2**

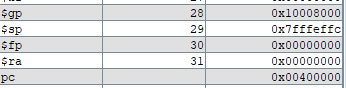


***Program in MIPS***



***Text Segment in MIPS***

* Program’s first step: The Stack Pointer ($sp) points to the address on top of the stack segment 0x7fffeffc. The Program Counter (pc) points to the starting address of the program. In this case, 0x00400000. This counter will change as it goes through the entire program. We also have a Group Point ($gp) pointing to the address 0x10008000.



* Each variable from h to size is declared and stored in an address within the Text Segment with the offset of 4.
  + h: .word 25 //declared h as 25
  + A: .word 0-100 // declared A as an array that takes 0-100 (memory allocated that can store 0-100 )
  + size: .word 100 // declare size as 100
* The LUI (Load Upper Intermediate) instruction points to the address of 0x00001001 which is represented as 0x10010000 since MIPS is Big Endian. Furthermore, this address is loaded into the $at Register which is reserved for use in pseudoinstructions.



* Next, the variable h’s value is loaded and stored in Save Register $s2. The value reflected is in Hexadecimal format (0x00000019). [lw $s2, h]



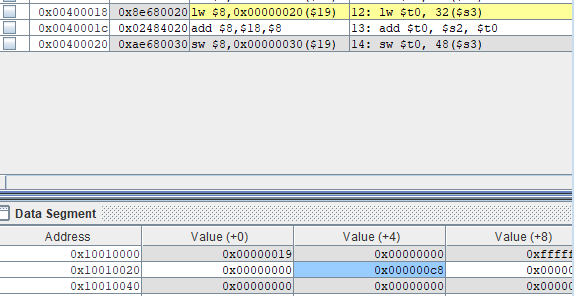
* The LI (Load Immediate) instruction to store a value of 200 in Temporary Register $t1 is executed. This pseudoinstruction is translated as ADDIU (Add Immediate Unsigned) which will store 200 as a hexadecimal in the Temporary Register $t1, represented as $9. [li $t1, 200 -> addiu $9, $0, 0x000000c8]
* Then, the address of A is loaded into the Save Register $s3. [la $s3, A]



* Then, we go to the $at Register and then, the ORI (Or Immediate) instruction loads the value of 0x00000004 (4) to the register $19 which is Save Register $s3. This register would take the value of LUI with the offset of 4. [ori $19, $1, 0x00000004]



* The next instruction will then store the value in $9 or Temporary Register $t1 in 0x00000020($19). The value is stored in 0x10010020 in the Data Segment with the offset of 32 to store 0x000000c8. [sw $t1, 32($s3)]



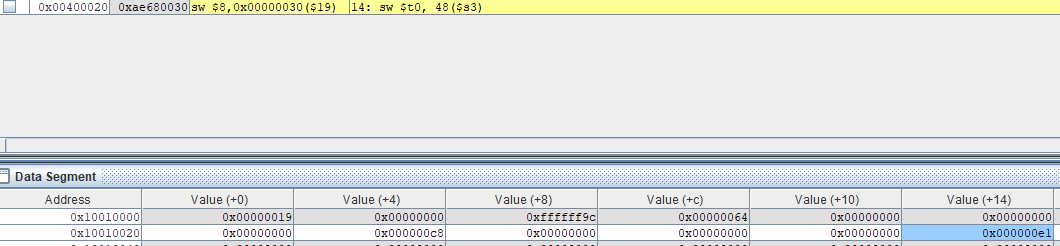
* Then, we load the value stored in 0x10010020 to the Temporary Register $t0. [lw $t0, 32($s3)]



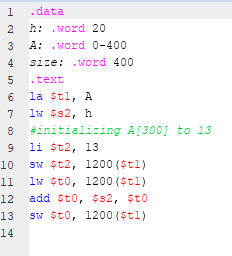
* Here, we add the values of $s2 and $t0 and push it to the Temporary Register $t0 (0x000000e1). [add $t0, $s2, $t0].



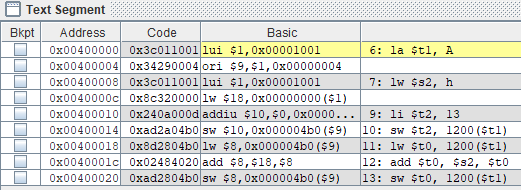
* Then, we store the new value of $t0 into 48($s3). [sw, $t0, 48($s3)]



**Problem 2.5-2**

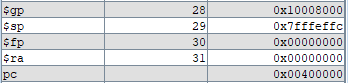


***Program in MIPS***



***Text Segment in MIPS***

* Program’s first step: The Stack Pointer ($sp) points to the address on top of the stack segment 0x7fffeffc. The Program Counter (pc) points to the starting address of the program. In this case, 0x00400000. This counter will change as it goes through the entire program. We also have a Group Point ($gp) pointing to the address 0x10008000.



* Each variable from h to size is declared and stored in an address within the Text Segment with the offset of 4.
  + h: .word 20 //declared h as 20
  + A: .word 0-400 // declared A as an array that takes 0-400 (memory allocated that can store 0-400 bytes)
  + size: .word 400 // declare size as 400
* The LUI (Load Upper Intermediate) instruction points to the address of 0x00001001 which is represented as 0x10010000 since MIPS is Big Endian. Furthermore, this address is loaded into the $at Register which is reserved for use in pseudoinstructions.



* Then, the address of A is loaded into the Temporary Register $t1. [la $t1, A]



* Then, we go to the $at Register and then, the ORI (Or Immediate) instruction loads the value of 0x00000004 (4) to the register $9 which is Temporary Register $t1. This register would take the value of LUI with the offset of 4. [ori $9, $1, 0x00000004]
* Next, the variable h’s value is loaded and stored in Save Register $s2. The value reflected is in Hexadecimal format (0x00000014). [lw $s2, h]



* The LI (Load Immediate) instruction to store a value of 13 in Temporary Register $t2 is executed. This pseudoinstruction is translated as ADDIU (Add Immediate Unsigned) which will store 13 as a hexadecimal in the Temporary Register $t2, represented as $10. [li $t2, 13 -> addiu $10, $0, 0x0000000d]
* The next instruction will then store the value in $10 or Temporary Register $t2 in 0x000004b0($19). The value is stored in 0x100104a0 in the Data Segment with the offset of 1200 from the top of the Data Segment to store 0x0000000d. [sw $t2, 1200($t1)]



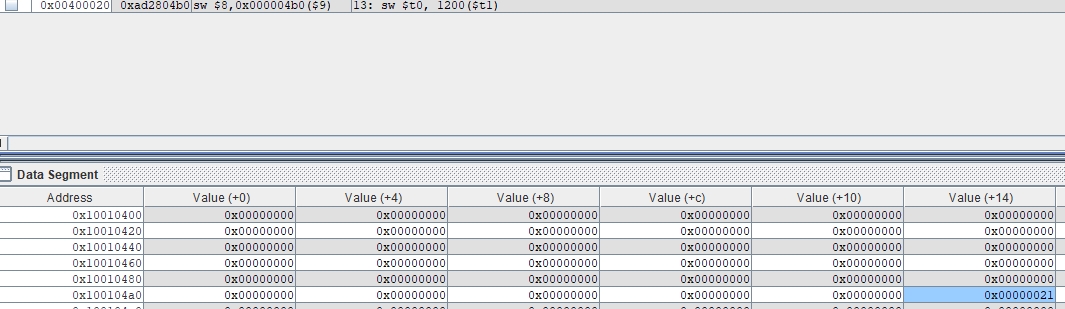
* Then, we load the value stored in 0x100104a0 to the Temporary Register $t0. [lw $t0, 1200($t1)]



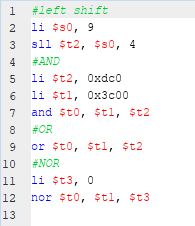
* Here, we add the values of $s2 and $t0 and push it to the Temporary Register $t0 (0x00000021). [add $t0, $s2, $t0]



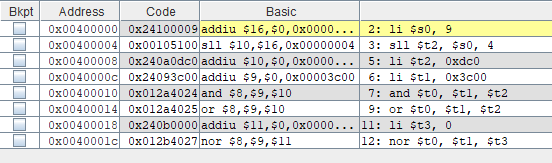
* Then, we store the new value of $t0 into 1200($t1). [sw, $t0, 1200($t1)]



**Problem 2.6-1**

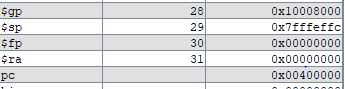


***Program in MIPS***



***Text Segment in MIPS***

* Program’s first step: The Stack Pointer ($sp) points to the address on top of the stack segment 0x7fffeffc. The Program Counter (pc) points to the starting address of the program. In this case, 0x00400000. This counter will change as it goes through the entire program. We also have a Group Point ($gp) pointing to the address 0x10008000.



* The LI (Load Immediate) instruction loads the immediate value (9) into the Save Register $s0.



* The SLL (Shift Left Logical) instruction shifts the value by a constant number of bits. In this case, the value stored in Save Register $s0 is shifted by 4 bits. The new value is 0x00000090 and then stored in the Temporary Register $t2.



* The LUI (Load Upper Intermediate) instruction points to the address of 0x00001001 which is represented as 0x10010000 since MIPS is Big Endian. Furthermore, this address is loaded into the $at Register which is reserved for use in pseudoinstructions.
* The LI (Load Immediate) instruction to store a value of 3264 in Temporary Register $t2 is executed. This pseudoinstruction is translated as ADDIU (Add Immediate Unsigned) which will store 3264 as a hexadecimal in the Temporary Register $t2, represented as $10. [li $t2, 0xdc0 -> addiu $10, $0, 0x00000dc0]



* The LI (Load Immediate) instruction to store a value of 0x3c00 in Temporary Register $t1 is executed. This pseudoinstruction is translated as ADDIU (Add Immediate Unsigned) which will store 0x3c00 in the Temporary Register $t1, represented as $10. [li $t1, 0x3c00 -> addiu $9, $0, 0x00003c00]



* Next, the AND (Bitwise AND) instruction will do an AND operation (If two source bits at a specific location are 1, the operation will yield 1. If not, then 0] on the values currently in Temporary Registers $t1 and $t2 and will store the resulting value in Temporary Register $t0 (0x00000c00). [and $t0, $t1, $t2]



* Next, the OR (Bitwise OR) instruction will do an OR operation (If two source bits at a specific location are 0, yield 0. If one is at least one, then yield 1] on the values currently in Temporary Registers $t1 and $t2 and will store the resulting value in Temporary Register $t0 (0x00003dc). [or $t0, $t1, $t2]



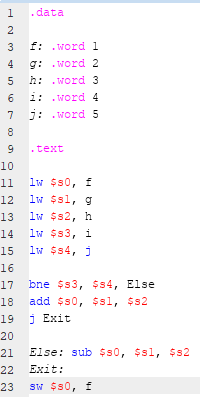
* Then, the LI (Load Immediate) instruction to store a value of 0 in Temporary Register $t3 is executed. This pseudoinstruction is translated as ADDIU (Add Immediate Unsigned) which will store 0 as a hexadecimal in the Temporary Register $t3, represented as $11. [li $t3, 0 -> addiu $10, $0, 0x00000000]



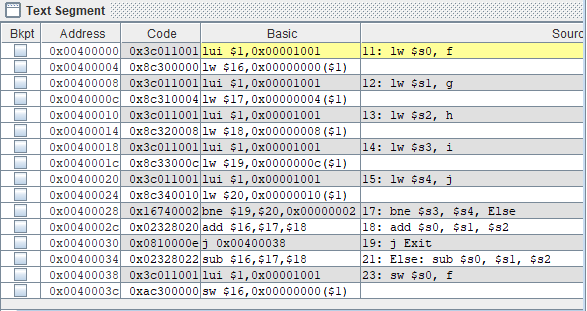
* Finally, the NOR (Bitwise NOR) instruction will do an NOR operation (If two source bits at a specific location are 1, then the operation will yield 0. If not, then 1 if both are 0] on the values currently in Temporary Register $t1 and $t3 and will store the resulting value in Temporary Register $t0 (0xffffc3ff). [nor $t0, $t1, $t3]



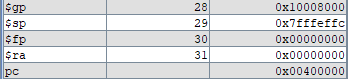
**Problem 2.7-1**



***Program in MIPS***



* Program’s first step: The Stack Pointer ($sp) points to the address on top of the stack segment 0x7fffeffc. The Program Counter (pc) points to the starting address of the program. In this case, 0x00400000. This counter will change as it goes through the entire program. We also have a Group Point ($gp) pointing to the address 0x10008000.



* Each variable from f to j is declared and stored in an address within the Text Segment with the offset of 4.
  + f: .word 1 // declared as 1
  + g: .word 2 // declared as 2
  + h: .word 3 // declared as 3
  + i: .word 4 // declared as 4
  + j: .word 5 // declared as 5
* The LUI (Load Upper Intermediate) instruction points to the address of 0x00001001 which is represented as 0x10010000 since MIPS is Big Endian. Furthermore, this address is loaded into the $at Register which is reserved for use in pseudoinstructions.



* Next, the variable f’s value is loaded and stored in Save Register $s0. The value reflected is in Hexadecimal format (0x00000001). [lw $s0, f]



* Next, the variable g’s value is loaded and stored in Save Register $s1. The value reflected is in Hexadecimal format (0x00000002). [lw $s1, g]



* Next, the variable h’s value is loaded and stored in Save Register $s2. The value reflected is in Hexadecimal format (0x00000003). [lw $s2, h]



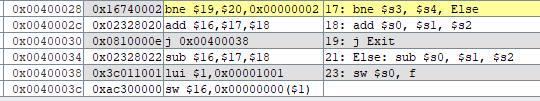
* Next, the variable i’s value is loaded and stored in Save Register $s3. The value reflected is in Hexadecimal format (0x00000004). [lw $si, h]

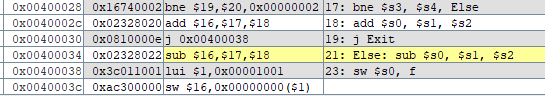


* Next, the variable j’s value is loaded and stored in Save Register $s4. The value reflected is in Hexadecimal format (0x00000005). [lw $s4, j]



* Then, we have an Branch on Not Equal instruction. Here, it suggests that the next instruction after this is at Else if the Saved Register values $s3 and $s4 are not equal. [bne $s3, $4, Else]

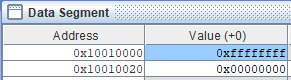




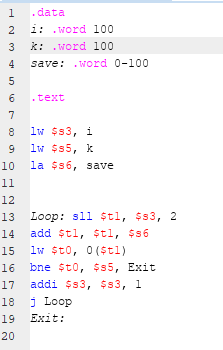
* Now, we are at the Else instruction which does a subtraction instruction. Here, we subtract the Saved Register value of h [$2] from the Saved Register value of g [$s1]. And we store this value in Saved Register value of f [$s0]. [Else: sub $s0, $s1, $s2]



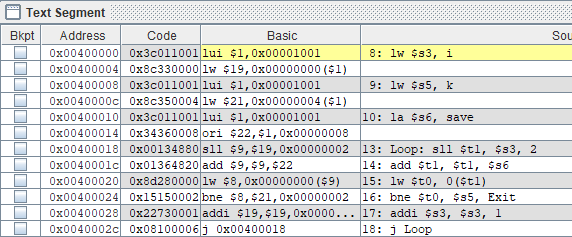
* The final instruction is to store this new value of f, overwritten by the sub instruction in the address of the $at Register which is 0x10010000. [sw $s0, f]



**Problem 2.7-2**

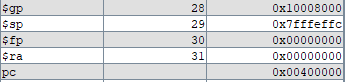


***Program in MIPS***



***Text Segment in MIPS***

* Program’s first step: The Stack Pointer ($sp) points to the address on top of the stack segment 0x7fffeffc. The Program Counter (pc) points to the starting address of the program. In this case, 0x00400000. This counter will change as it goes through the entire program. We also have a Group Point ($gp) pointing to the address 0x10008000.



* Each variable from i to k is declared and stored in an address within the Text Segment with the offset of 4.
  + i: .word 100 // declared as 100
  + k: .word 100 // declared as 100
  + save: .word 1-100 //declared as array that takes 0 – 100 (Memory allocated to take 0 – 100 bytes)
* The LUI (Load Upper Intermediate) instruction points to the address of 0x00001001 which is represented as 0x10010000 since MIPS is Big Endian. Furthermore, this address is loaded into the $at Register which is reserved for use in pseudoinstructions.



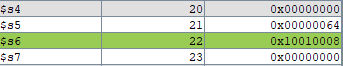
* Next, the variable i’s value is loaded and stored in Save Register $s3. The value reflected is in Hexadecimal format (0x00000064). [lw $s3, i]



* Next, the variable k’s value is loaded and stored in Save Register $s5. The value reflected is in Hexadecimal format (0x00000064). [lw $s5, k]



* Next, the address of save is loaded into the Saved Register $s6. [la $s6, save]



* Then, we go through the Loop. The first instruction is to shift the value stored in Saved Register $s3 by 2 bits to the left using the shift left operand. We then store this value in the Temporary Register $t1. [sll $t1, $s3, 2]



* Next, we add the address stored in $s6 and the value stored in Temporary Register $t1. This is then stored in Temporary Register $t1 which overwrites the original value. [add $t1, $t1, $s6]



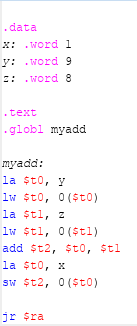
* Next, we load the value of $t1 stored in address 0 to $t0. [lw $t0, 0($t1)]



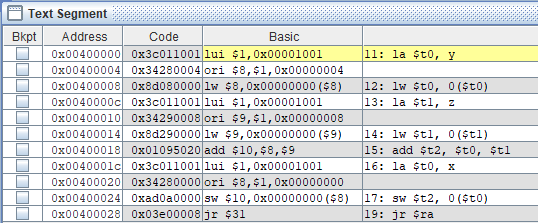
* Then, we have a Branch on Not Equal instruction which says that if values stored in Saved Register $s5 and Temporary Register $t0 are not equal, we will then Exit the program. Note: if i and j are defined as 0 bytes, then the program will not Exit!



**Problem 2.8-1**

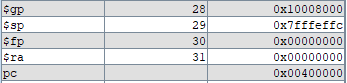


***Program in MIPS***



***Text Segment in MIPS***

* Program’s first step: The Stack Pointer ($sp) points to the address on top of the stack segment 0x7fffeffc. The Program Counter (pc) points to the starting address of the program. In this case, 0x00400000. This counter will change as it goes through the entire program. We also have a Group Point ($gp) pointing to the address 0x10008000.



* Each variable from x to z is declared and stored in an address within the Text Segment with the offset of 4.
  + x: .word 1 // declared as 1
  + y: .word 9 // declared as 9
  + z: .word 8 // declared as 8
* The LUI (Load Upper Intermediate) instruction points to the address of 0x00001001 which is represented as 0x10010000 since MIPS is Big Endian. Furthermore, this address is loaded into the $at Register which is reserved for use in pseudoinstructions.



* Next, we load the address of y into the Temporary Register $t0. [la $t0, y]



* Then, we load the word stored in y which is 9 and we store it in the Temporary Register $t0. [lw $t0, 0($t0)]



* Next, we load the address of z into the Temporary Register $t1. [la $t1, z]



* Then, we load the word stored in z which is 8 and we store it in the Temporary Register $t1. [lw $t1, 0($t1)]



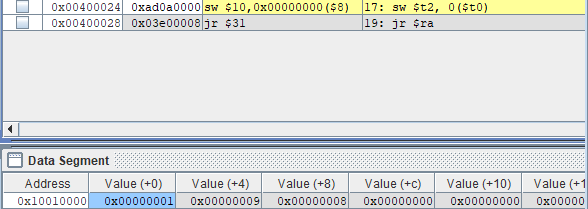
* Next, we use an Add Instruction to add the values stored in Temporary Registers $t0 and $t1. Then, the new value is stored (value is 17 in Binary) in Temporary Register $t2. [add $t2, $t0, $t1]



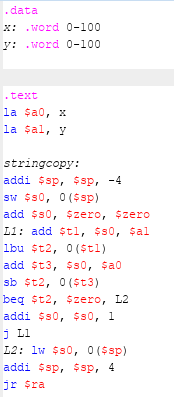
* Next, we load the address of x into the Temporary Register $t0. [la $t0, x]



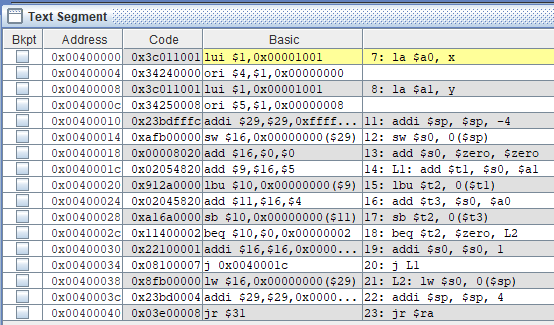
* Next, we stored the value stored in the address of the Temporary Register $t0 and we store it in the address of Temporary Register $t2. The value will be reflected in the Data Segment. [sw $t2, 0($t0)]



**Problem 2.9-1**

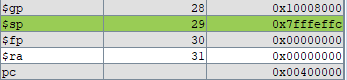


***Program in MIPS***



***Text Segment in MIPS***

* Program’s first step: The Stack Pointer ($sp) points to the address on top of the stack segment 0x7fffeffc. The Program Counter (pc) points to the starting address of the program. In this case, 0x00400000. This counter will change as it goes through the entire program. We also have a Group Point ($gp) pointing to the address 0x10008000.



* Each variable from x to z is declared and stored in an address within the Text Segment with the offset of 4.
  + x: .word 0-100 // declared as an Array that takes the value of 0-100
  + y: .word 0-100 // declared as an Array that takes the value of 0-100
* The LUI (Load Upper Intermediate) instruction points to the address of 0x00001001 which is represented as 0x10010000 since MIPS is Big Endian. Furthermore, this address is loaded into the $at Register which is reserved for use in pseudoinstructions.



* Next, we load the address of x into the Argument Register $a0. [la $a0, x]



* Then, we load the address of y into the Argument Register $a1. Note: The address of y reflects that the arrays of y and x are next to each other and that arrays can take 32 bits. [la $a1, y]



* We then use the Add Intermediate Instruction to add the value of Stack Pointer $sp and the value of -4. We then, move this new value to Stack Pointer $sp which overwrites its original value. [addi $sp, $sp, -4]



* We then store the value at the address of Stack Pointer $sp into the address of Save Register $s0. [sw $s0, 0($sp)]



* Next, we use an Add Instruction to add 0 and 0. This is then stored in the Save Register $s0. [add $s0, $zero, $zero]



* We then add the address stored in the Argument Register $a1 and the address stored in the Save Register $s0. This is then stored in the Temporary Register $t1. [add $t1, $s0, $a1]



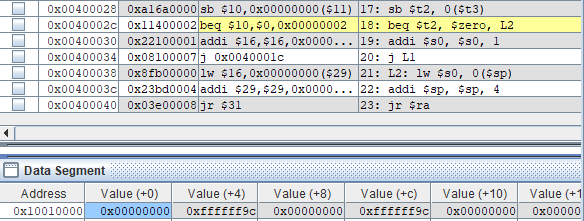
* We then use the Load Byte Unsigned Instruction which extracts a specified byte from a word and places it in the least significant byte position of the Target Register. Here, we load the value of 0 stored in the address in Temporary Register $t1. This is then stored in Temporary Register $t2. [lbu $t2, 0($t1)]



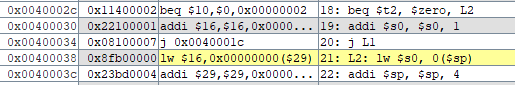
* Then, we the add the values stored in the Save Register $s0 and the Argument Register $a0. We would then store it in the Temporary Register $t3. [add $t3, $s0, $a0]



* Next, we stored the byte value of Temporary Register $t3 in the Temporary Register $t2. [sb $t2, 0($t3)]



* Next, we have a Branch if Equal Instruction which states that if the values of Temporary Register $t2 and Register $zero, we will then go to the Loop L2. [beq $t2, $zero, L2]



* The first Instruction of L2 is to load the value of the Stack Pointer (which is 0) in the Save Register $s0. [lw $s0, 0($sp)]



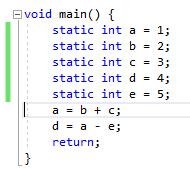
* We then have the Add Intermediate Instruction which adds the value of 4 to the address of the Stack Pointer $sp. [addi $sp, $sp, 4]



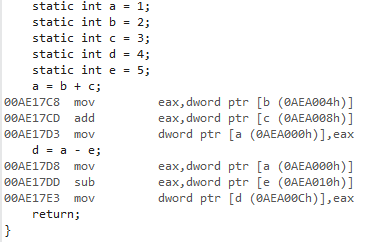
* Next, we exit the Program. [jr $ra]

**PART II – Intel**

* Intel is Little Endian [Most Significant Bit is stored Last]. Visual Studio supports Intel Architecture.



***Program in Visual Studio***



***Program in Disassembler***



***Static int a in Memory***



***Static int b, c, d and e in Memory***

* + Program: Contains 5 static variables/integers [a through e] with values [1 through 5].



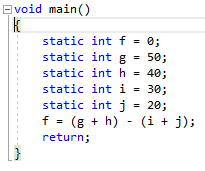
***Static int a in Memory after b + c***

* + The purpose of this program is to add b + c and store it in variable a. [a = b + c]
  + Here, static variable a’s value is overwritten and the new value 5 is stored. Evidence is shown in the address of a in memory as well as the EAX register.

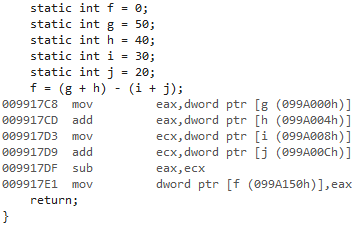


***Static int d in Memory after a – e***

* + The next step of the program is to subtract a – e and store it in variable d. [d = a -e]
  + Here, static variable d’s value is overwritten and the new value 0 is stored. Evidence is shown in the address of d in memory as well as the EAX register.



***Program in Visual Studio***



***Program in Disassembler***

* + Program: Contains 5 static variables/integers [f through j] with values [0, 50, 40, 30 and 20 respectively].



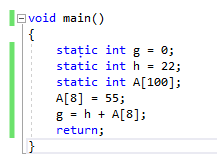
***Static int g, h, i and j in Memory***

* + The purpose of this program is to add g and h and i and j. We subtract (i + j) from (g + h) and store it in variable f. [f = (g+h) – (i+j)]

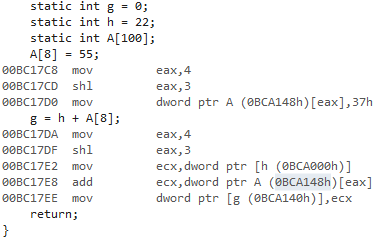


***Static int f in Memory***

* + The resulting value, 40 is stored as a Hexadecimal in the address of f in memory.



***Program in Visual Studio***



***Program in Disassembler***

* Program: Contains 2 static variables and 1 static array of the size 100. [g = 0, h = 22, and A[100] ]



***Static int h = 22 in Memory***



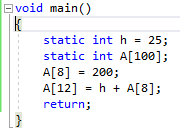
***A[8] = 55 in Memory***

* The start of this program is to have the number 55 be stored in the 8th position of Int Array A. (A[8] = 55)

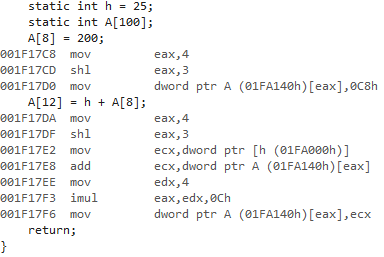


***Static int g = 77 in Memory***

* The following step is to add the value of h and A[8] which is 77 and store it in g. Value is stored as a Hexadecimal in the address of g in memory.



***Program in Visual Studio***



***Program in Disassembler***



***Static int h in Memory***

* Program: One static int variable h and One static int array A of size 100.



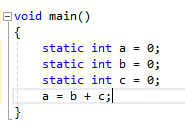
***A[8] = 200 in Memory***

* The first step of the program after declaring our static variables and array is to put 200 in the 8th place of the array A (A[8]). Represented as a Hexadecimal in Memory.

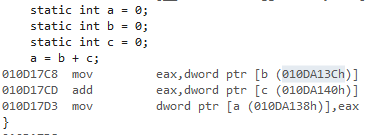


***A[12] = 225 in Memory***

* The second step was to add the contents of A[8] which is 200 and the value of h which is 25 and store the Hexadecimal value of 225 in the 12th of the array A. (A[12])



***Program in Visual Studio***



***Program in Disassembler***



***Static int a in Memory***

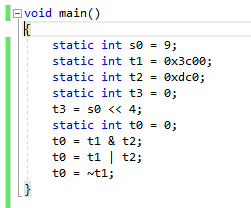


***Static int b in Memory***

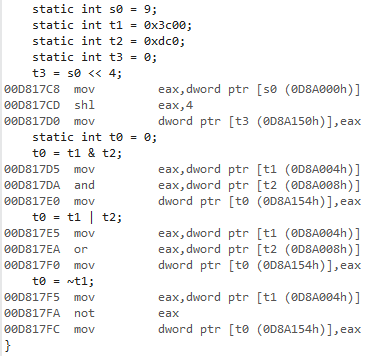


***Static int c in Memory***

* Program: Declare 3 static int variables a, b and c. All are initialized to 0.
* Then, we add b and c and its value is stored in variable a. This value is the Hexadecimal value of 0.



***Program in Visual Studio***



***Program in Disassembly***



***Static int s0, t1, t2 and t3***

* Program: Declare 4 static int variables s0, t1, t2 and t3.



***Static int t3***

* The next step is to store the value of static int variable s0 in static int variable t3.



***Static int t0***

* The next step is to store the bitwise AND operation of t1 and t2 in the static int variable t0. [Every binary position that is equal to 1 for both t1 and t2 is 1. Otherwise, 0]



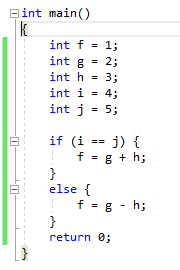
***Static int t0***

* The next step is to store the bitwise OR operation of t1 and t2 in the static int variable t0. [Every binary position that yields 1 will be 1 unless if the binary position for both is 0, then its 0.]



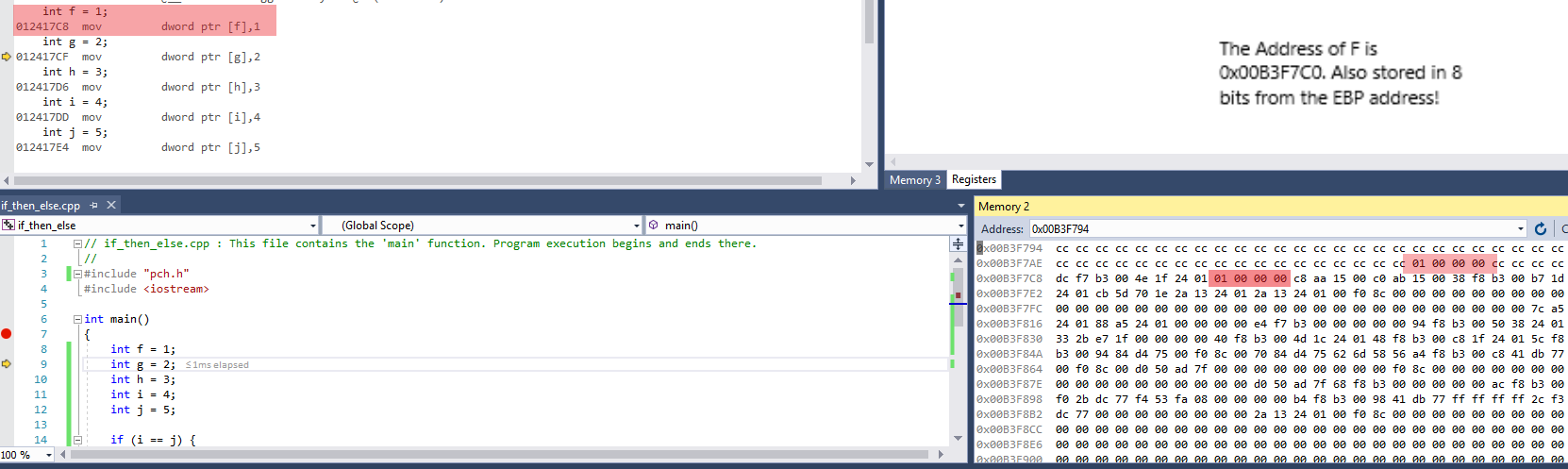
***Static int t0***

* The next step is to store the bitwise NOT operation of t1 in the static int variable t0. [Every binary position that yields 1 is 0 and vice versa]

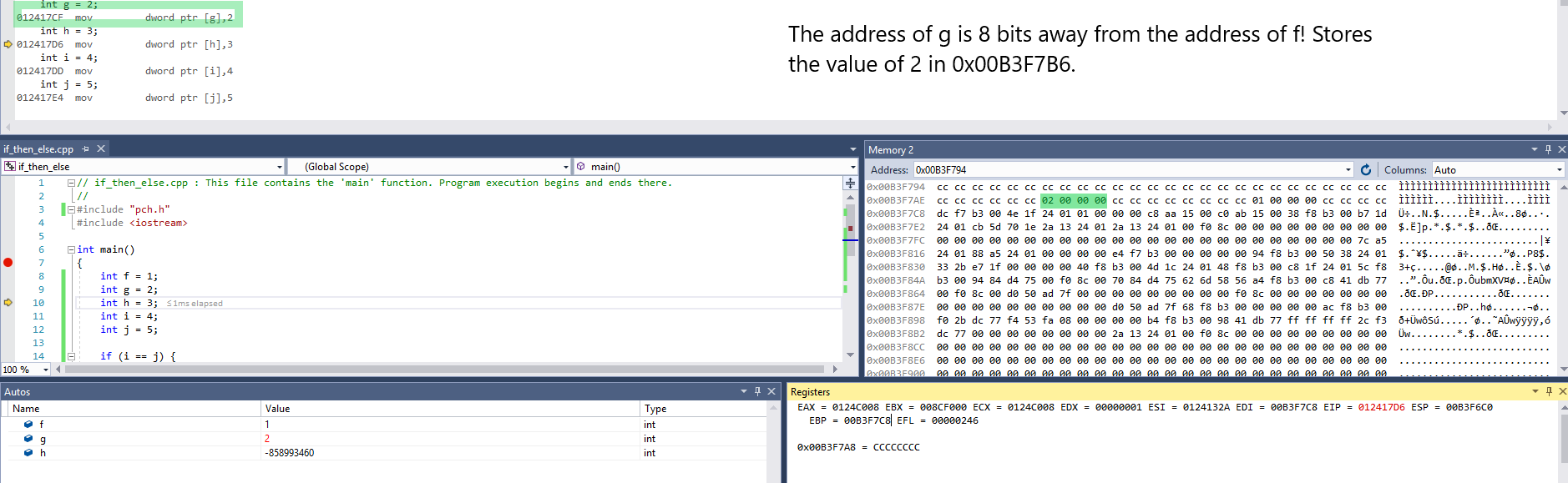


***Program in Visual Studios (if-then-else)***

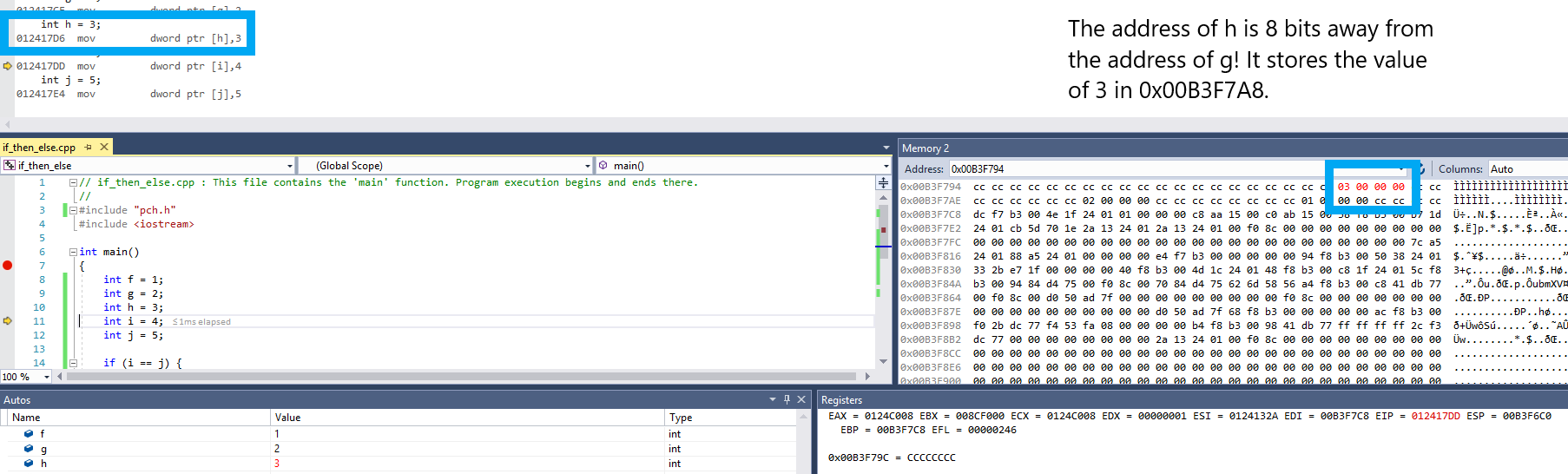
* This Program represents if-then-else in Visual Studios.
* We declare and store 5 variables: f, g, h, i and j as variable int.
* The Program will then execute the if statement. If i is equal to j, then it will add g and h and assign its value to f. Overwriting f.
* However, the next statement following this is when i is not equal to j. Then, it will subtract h from g. This value is then assigned to f. Overwriting f.
* For this program, the if statement will never execute since the values of i and j are not the same. Only the else statement executes



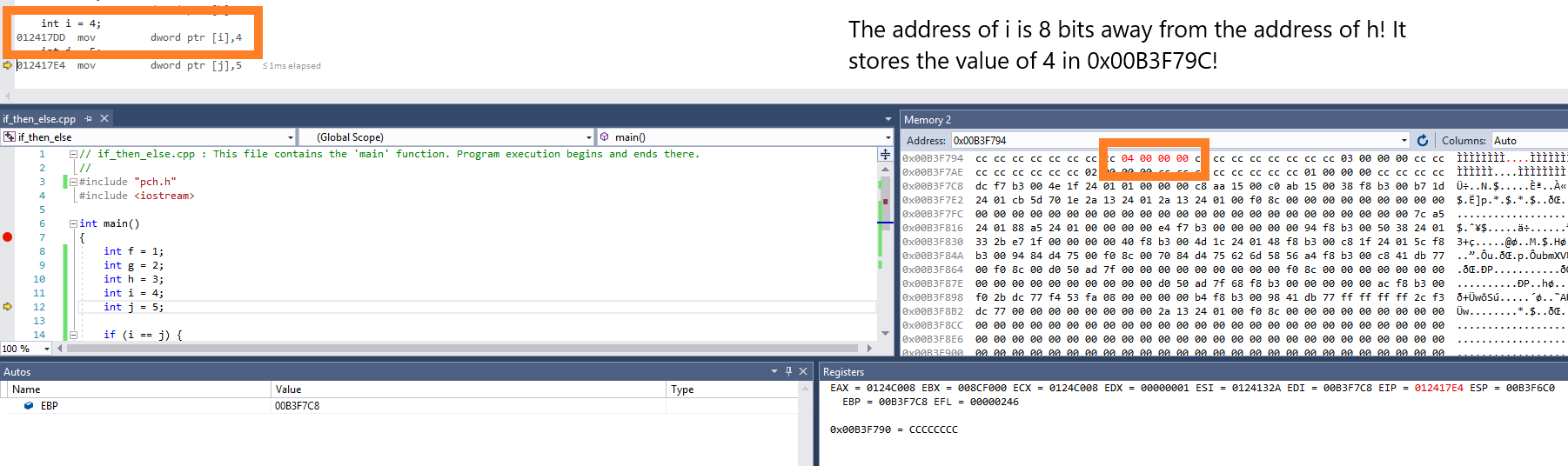
***Disassembly of the Program: F stored***

******

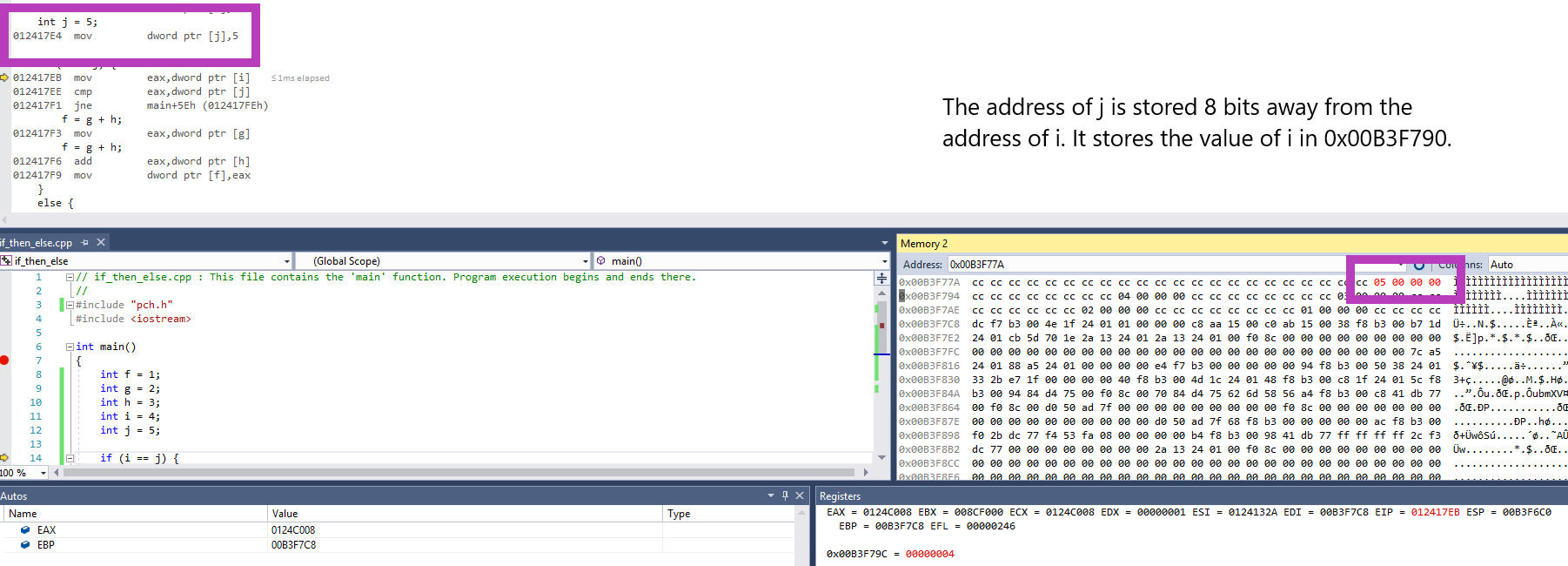
***Disassembly of the Program: G stored***

******

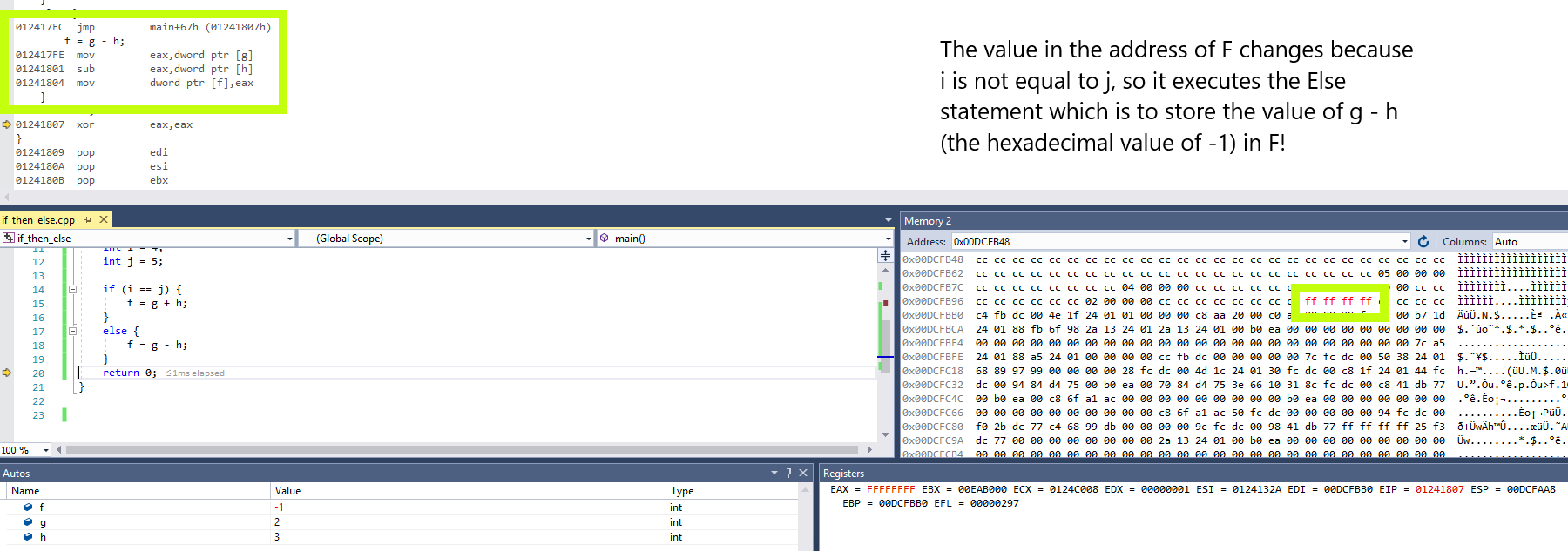
***Disassembly of the Program: H stored***

******

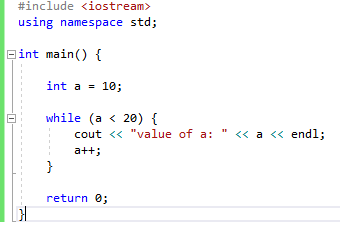
***Disassembly of the Program: I stored***

******

***Disassembly of the Program: J Stored***

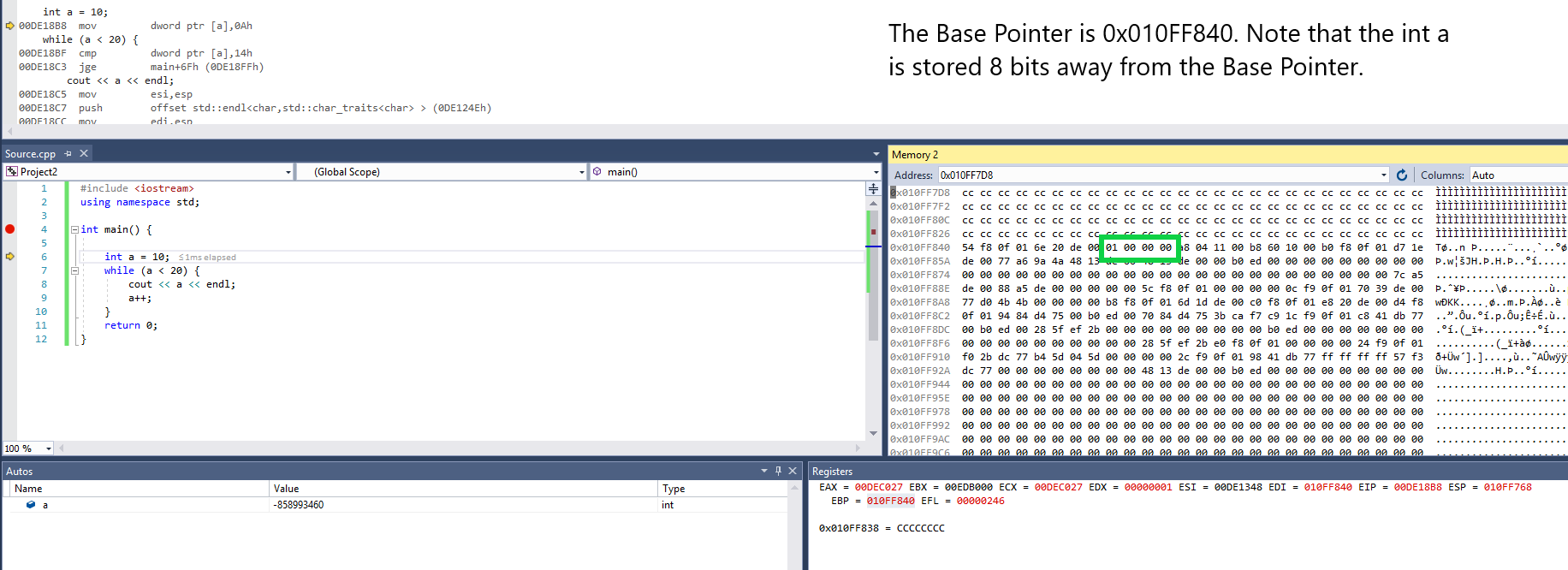
******

***Disassembly of the Program: Else statement***

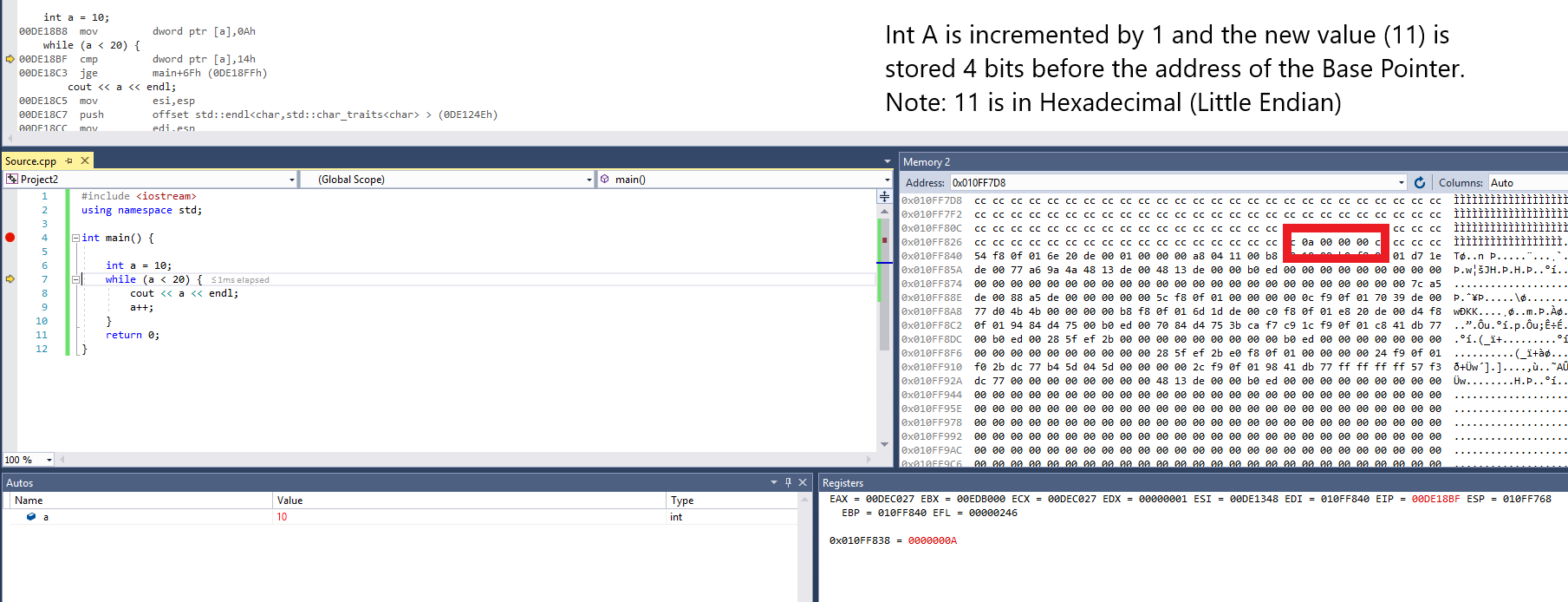
******

***Program in Visual Studios (while loop)***

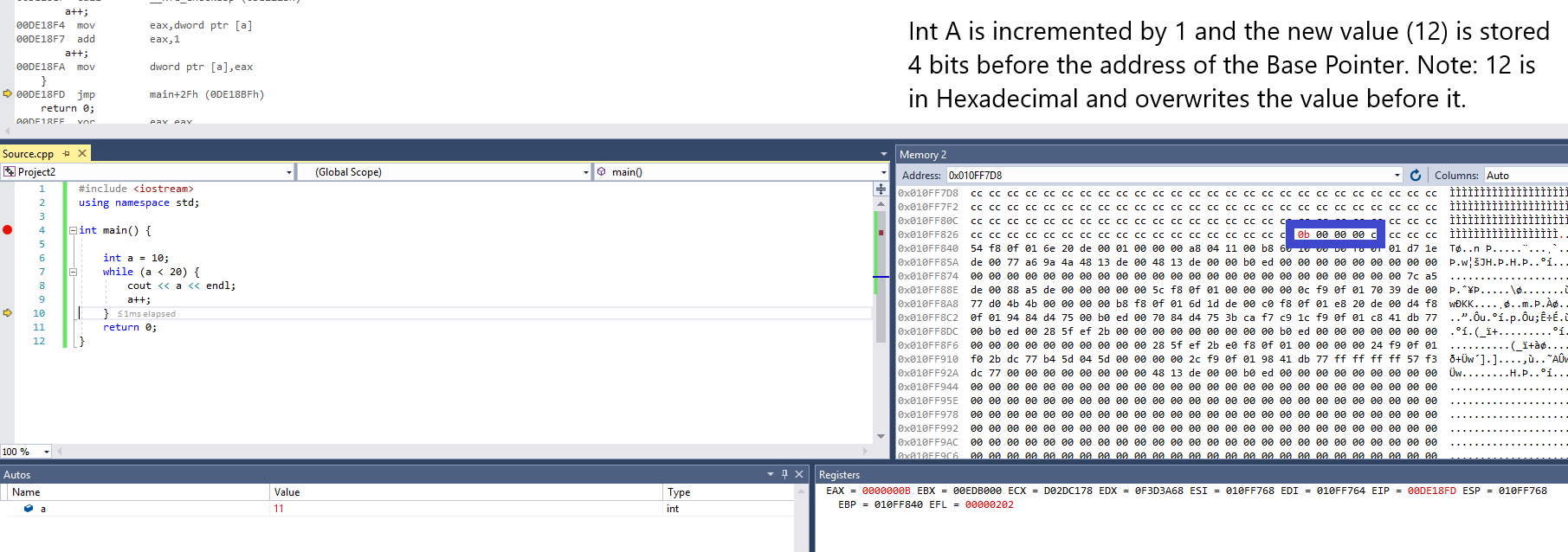
* This Program declares and stores the value of 10 in the int variable a.
* Next, we have a while statement that says that if a is less than 20 (which is true), then increment the value of a until it reaches the maximum potential value (19).
* Afterwards, the program is done.



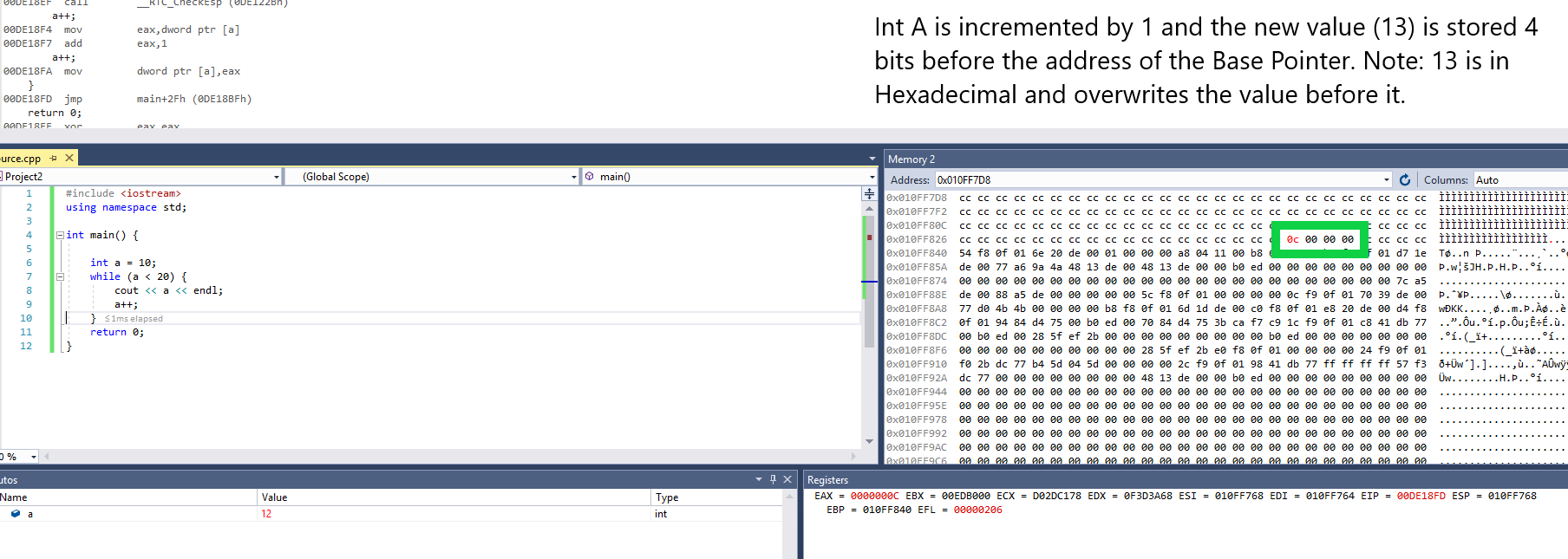
***Base Pointer Location and Location of int A***

******

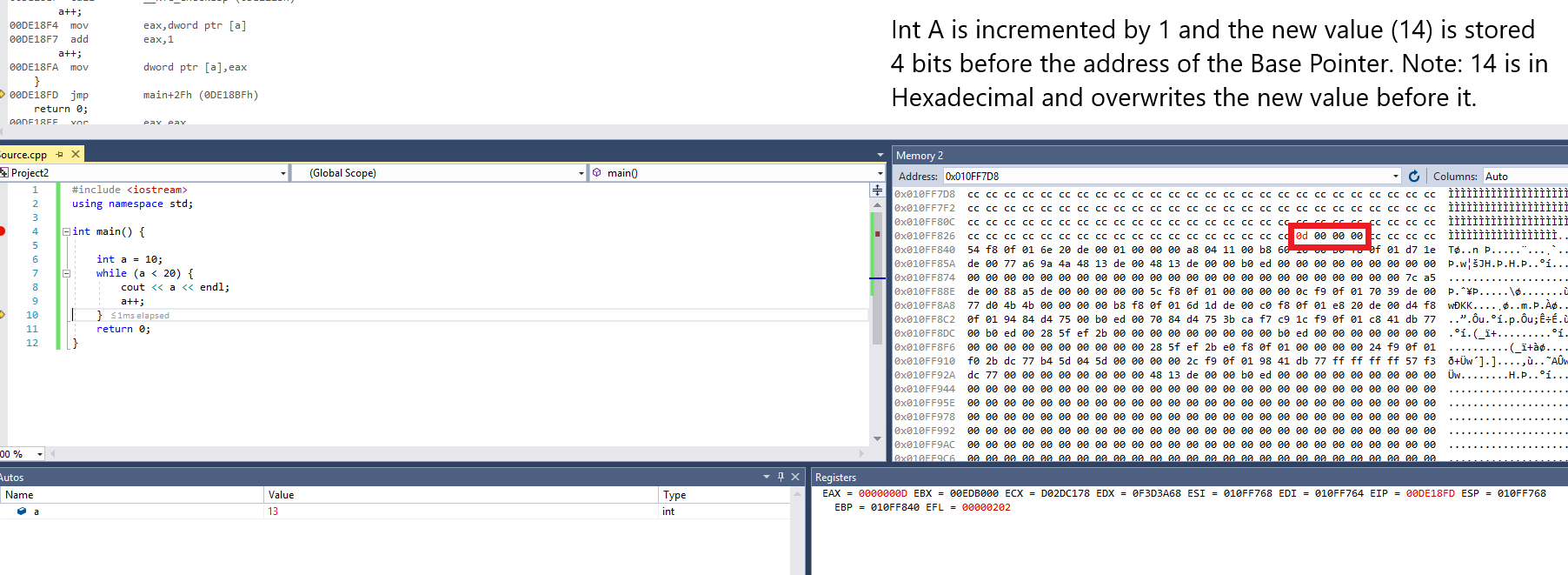
***The Value of A changes to 11***

******

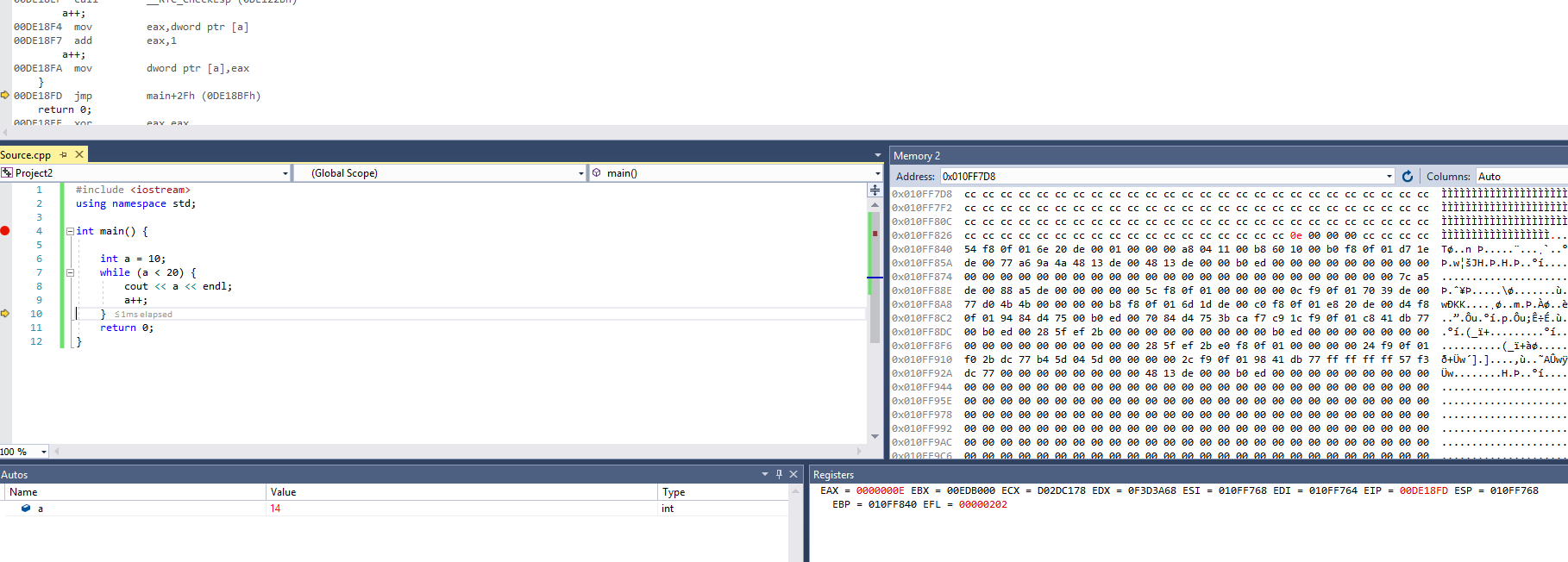
***The Value of A changes to 12***

******

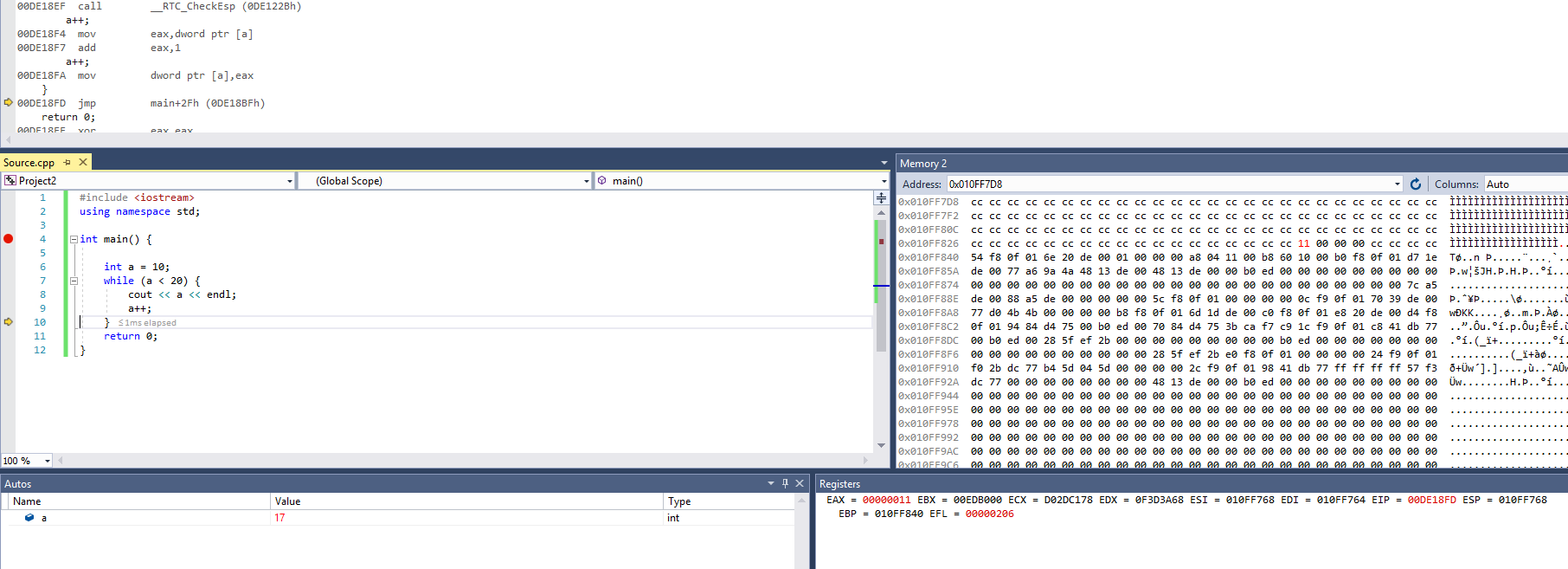
***The Value of A changes to 13***

******

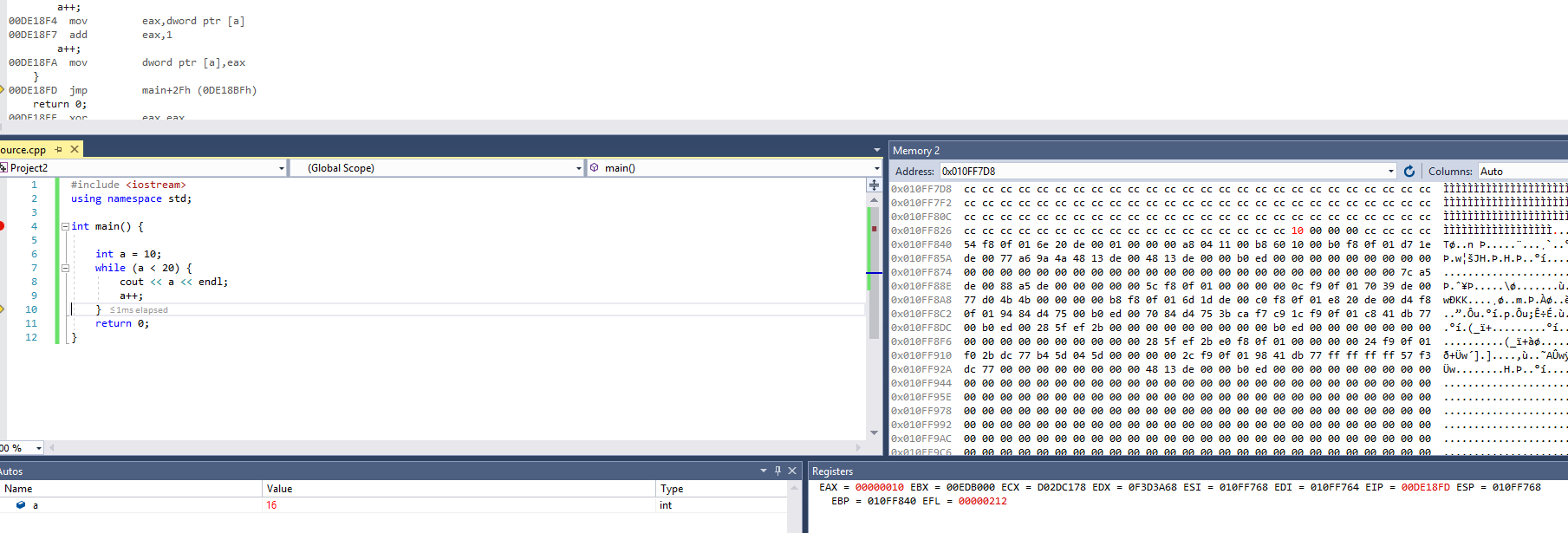
***The Value of A changes to 14***

******

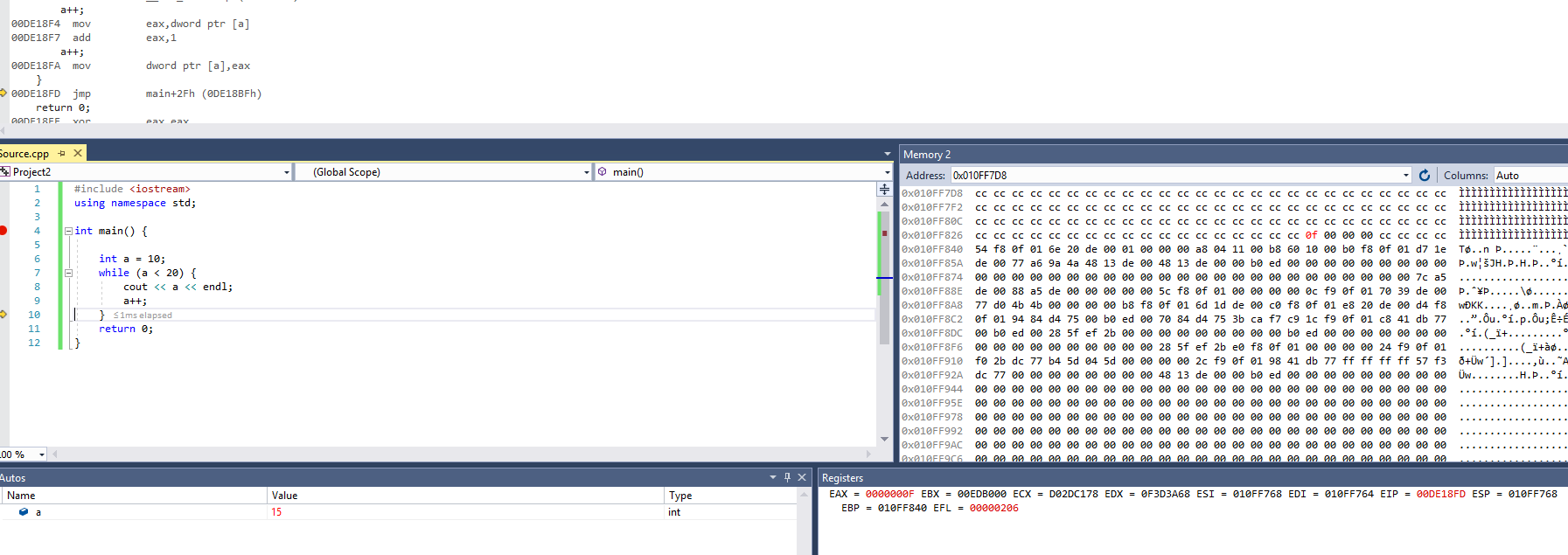
***The Value of A changes to 15***



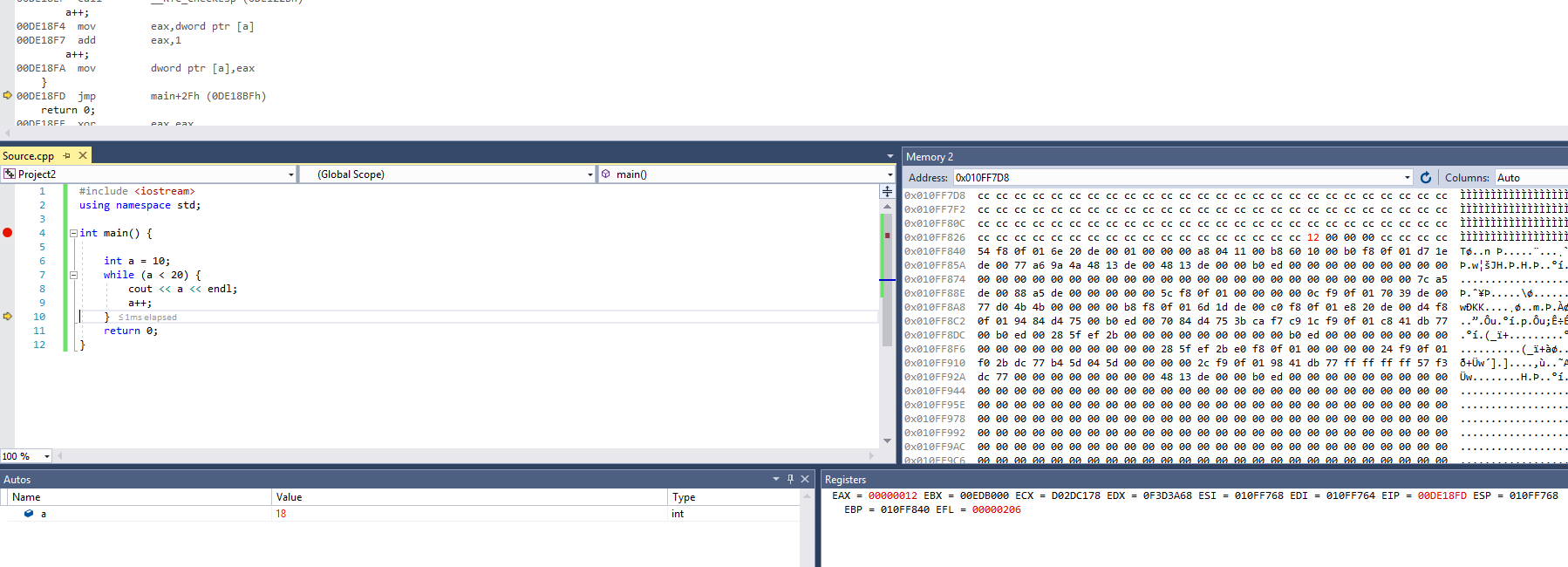
***The Value of A changes to 16***



***The Value of A changes to 17***



***The Value of A changes to 18***

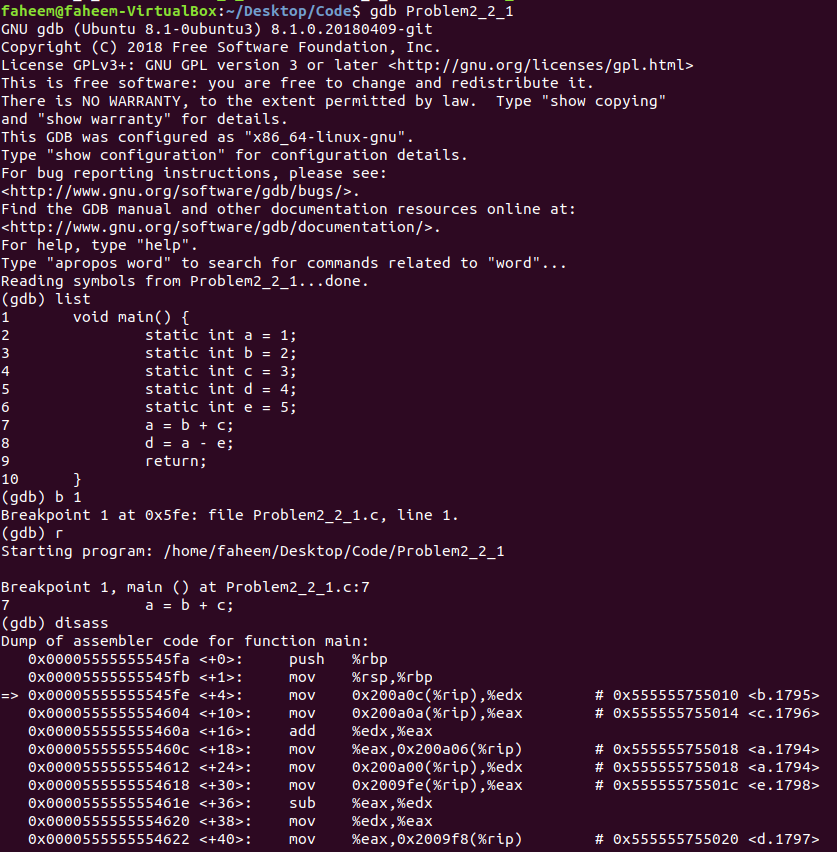
******

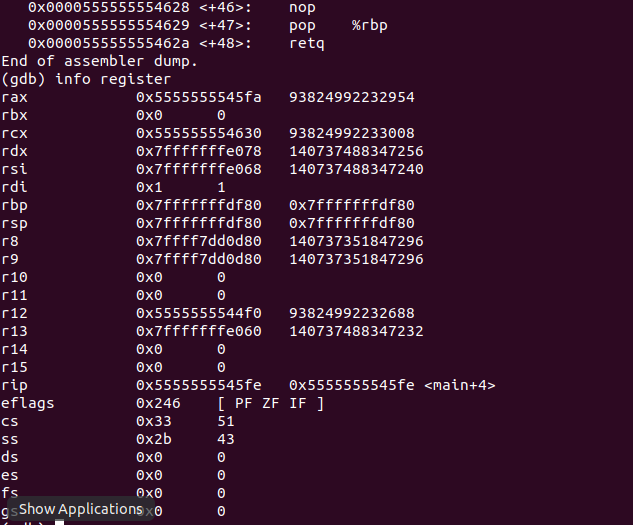
***The Value of A changes to 19***

**Part III. GCC and GDB (Intel x86-64 ISA)**

* Linux (GCC and GDB) is Big Endian.

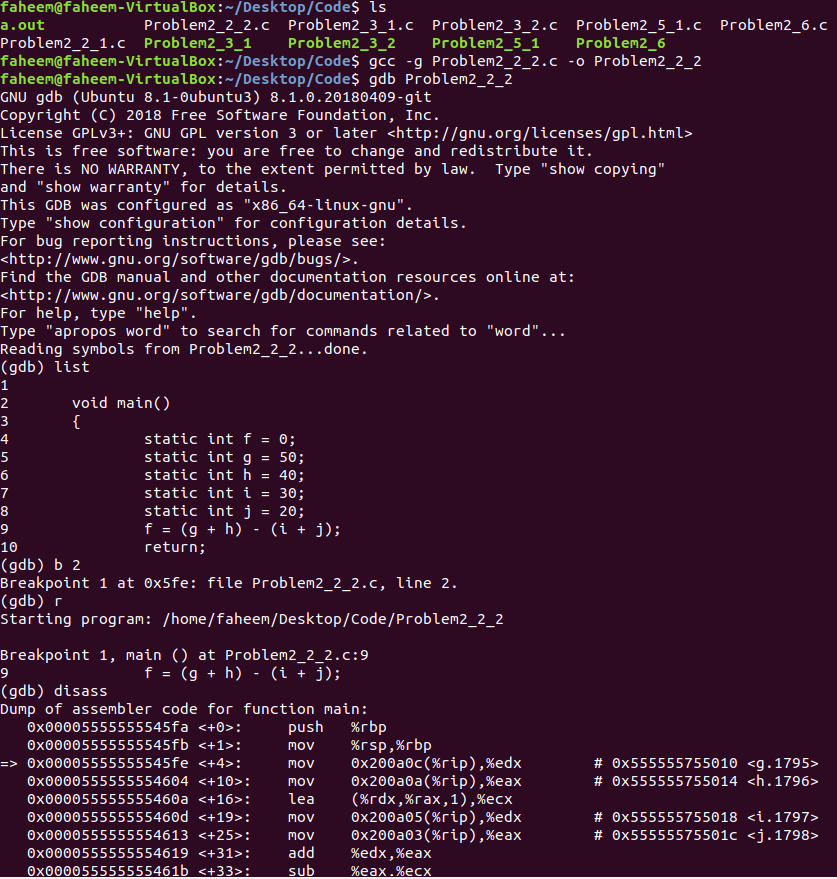
Problem 2.2.1:

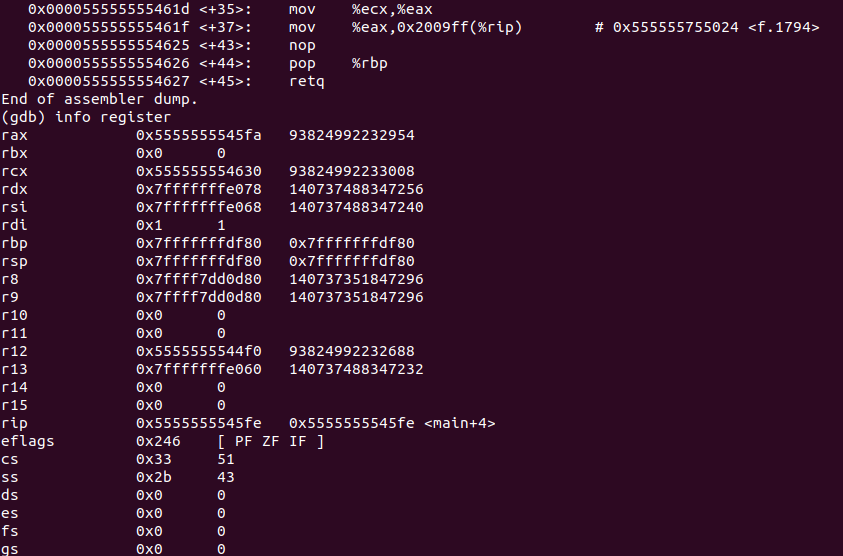




The first 2 instructions are called the function prologue. We push the old base pointer onto the stack (to return to later) and copy the value of the stack pointer to the base pointer. So, the base pointer points to the base of the main’s stack frame. The next instruction copiesthe value of 2 into the address 2099724 from the Instruction Pointer which is the next local variable slot. In other words, the value of b (which is 2) is stored at the offset value from the base pointer and that offset is the the offset of b. To verify this, we use gdb to get the location of b and the location of the $rip + offset and the address and value of both are the same. We do the same with c before both are moved the edx and eax register. Here an addition occurs based on the add instruction. Then the resulting value is moved to a new address from the eax register. We repeat the similar instructions for a and e before the subtraction instruction. This instruction will store the new value of d (which is 0). The last 2 steps, also known as the function epilogue, pop the old base pointer off the stack and stores it back into $rbp and “retq” jumps back to the return address (stored on stack).

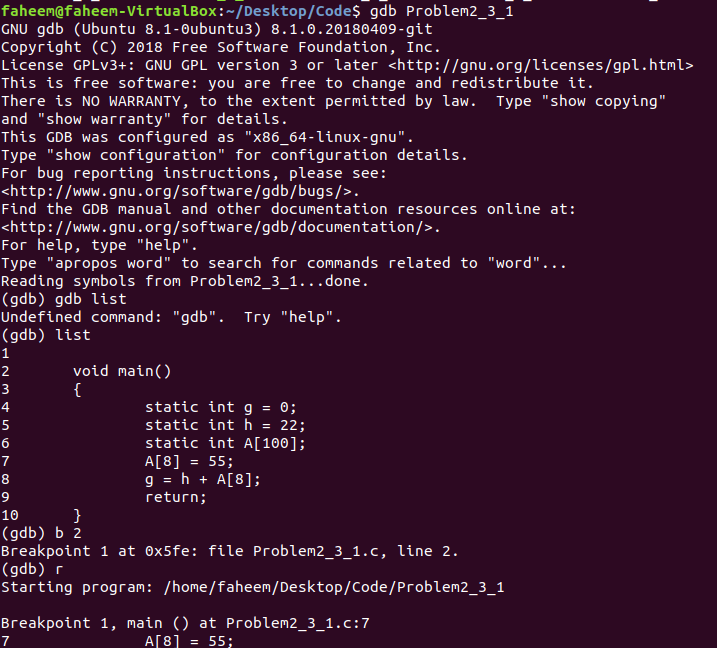
Problem 2.2.2:

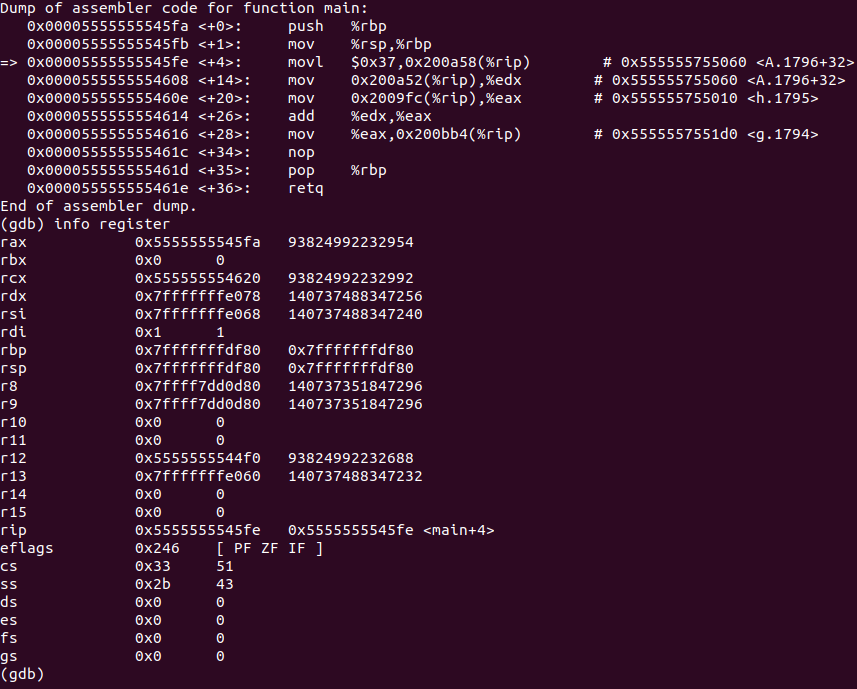




The first 2 instructions are called the function prologue. We push the old base pointer onto the stack (to return to later) and copy the value of the stack pointer to the base pointer. The base pointer serves as the base of the stack frame. We will then move two addresses (both of which are 2 bits away from each other from the Instruction Pointer $rip) to edx and eax respectively. We then use a load effective address which will compute the sum of rdx and rax. This new value will be stored in ecx (Counter Register). From here, another two addresses, also 2 bits away from each other from the Instruction Pointer will be moved into edx and eax. We then add these new values and then subtract ecx from eax to obtain a new value. We can check the value in GDB to give us the new value in hexadecimal. The last 2 steps, also known as the function epilogue, pop the old base pointer off the stack and store is back into $rbp and “retq” jumps back to the return address (stored on the stack frame).

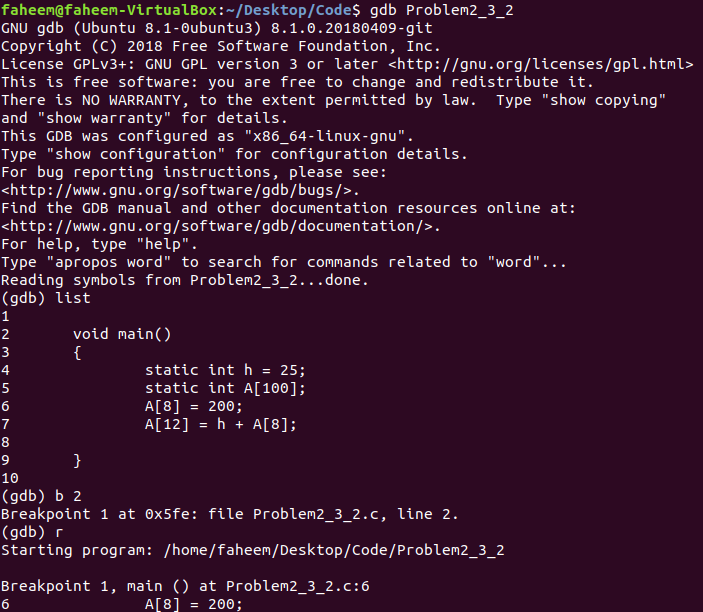
Problem 2.3.1:

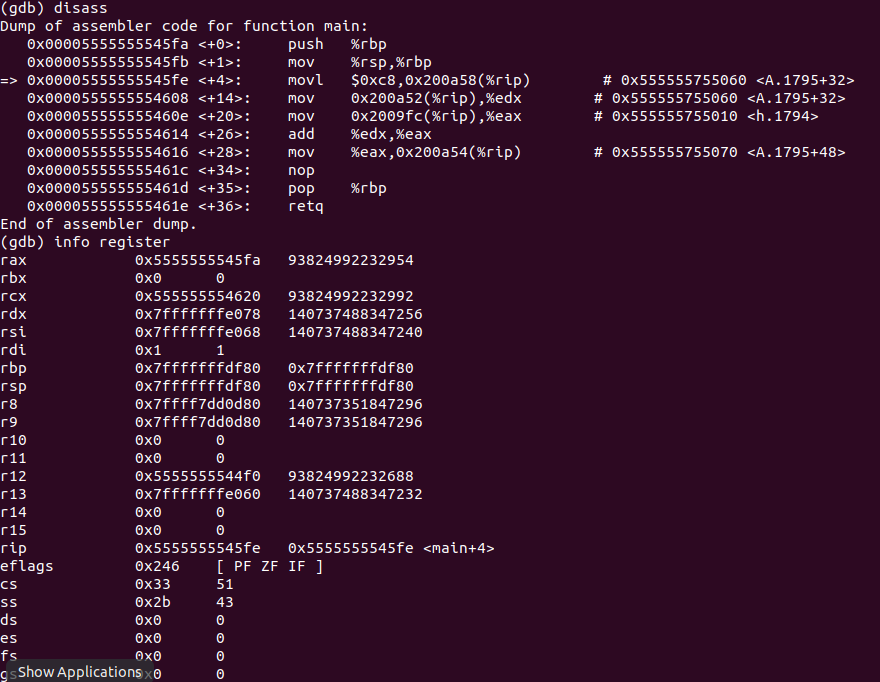




The first 2 instructions are called the function prologue. We push the old base pointer onto the stack (to return to later) and copy the value of the stack pointer to the base pointer. The base pointer serves as the base of the stack frame. The movl instruction will overwrite the value in the address of 0x200a58 from the Instruction Pointer. We then move, two values 0x200a52 and 0x2009fc to the edx and eax instructions respectively. We will then add these values and then move this particular value to the address 0x200bb4 from the Instruction Pointer. The last 2 steps, also known as the function epilogue, pop the old base pointer off the stack and store is back into $rbp and “retq” jumps back to the return address (stored on the stack).

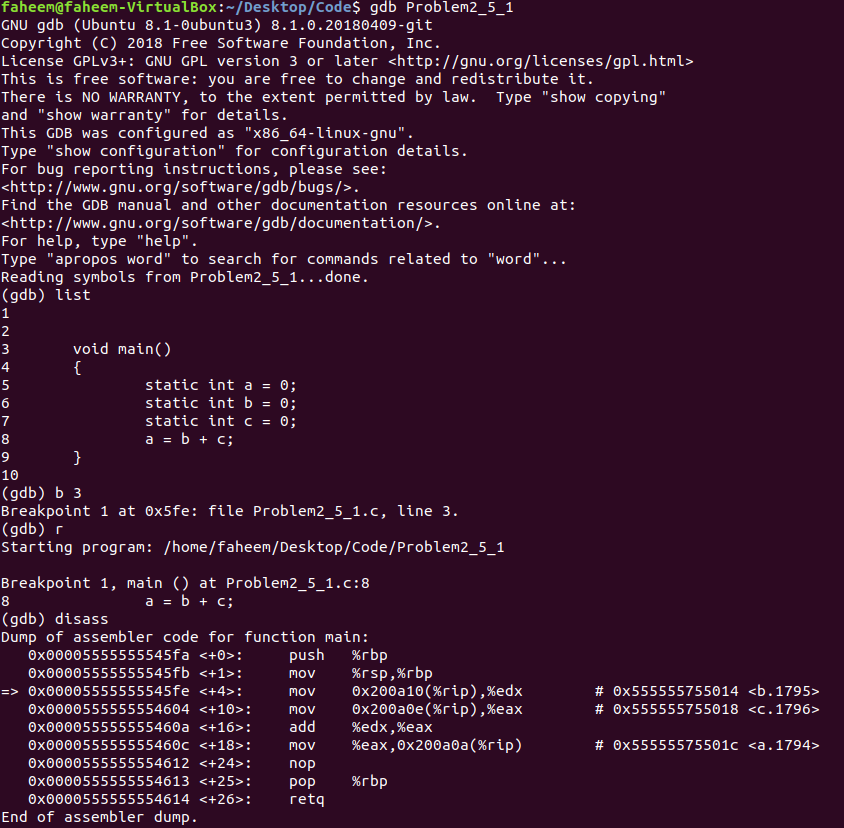
Problem 2.3.2:

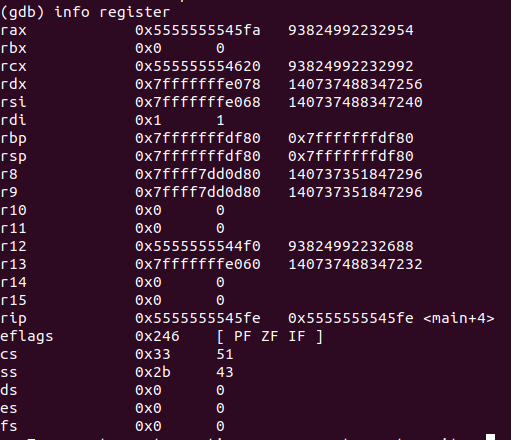




The first 2 instructions are called the function prologue. We push the old base pointer onto the stack (to return to later) and copy the value of the stack pointer to the base pointer. The base pointer serves as the base of the stack frame. We then move the values stored in 0x200a52 and 0x2009fc from the Instruction Pointer to the edx and eax registers. Then, we add these values and store it in the address 0x200a54 from the Instruction Pointer. The last 2 steps, also known as the function epilogue, pop the old base pointer off the stack and store is back into $rbp and “retq” jumps back to the return address (stored on the stack frame).

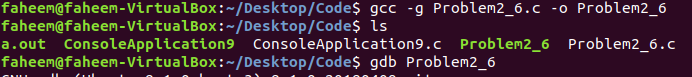
* Problem 2.5.1:

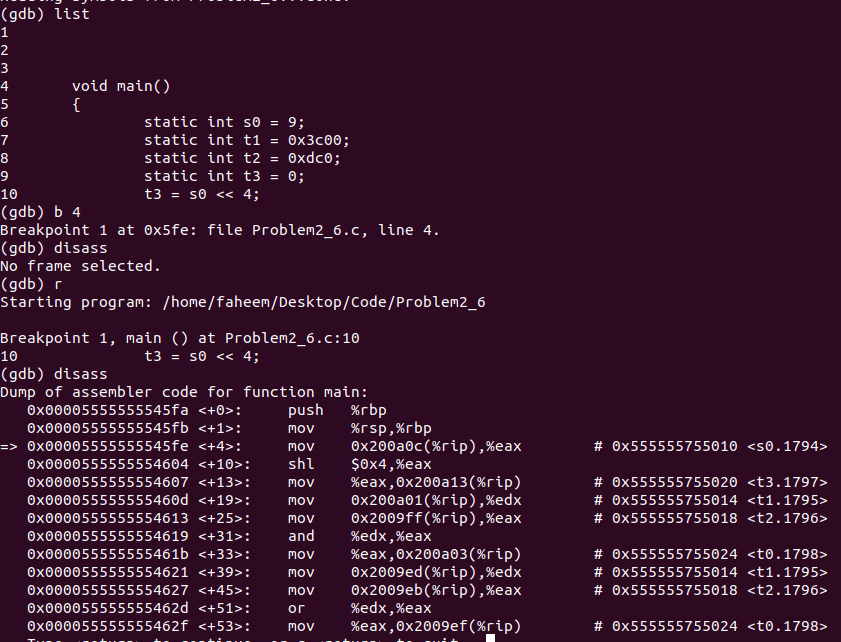


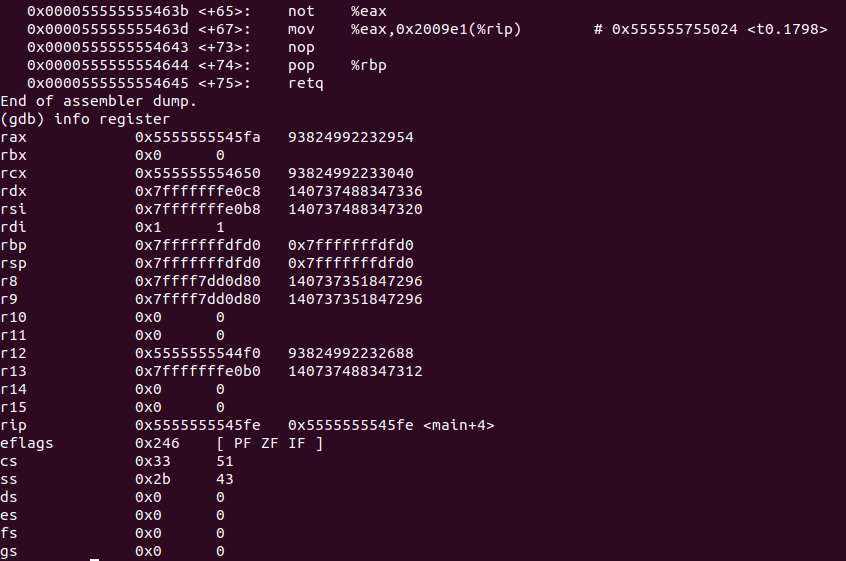


The first 2 instructions are called the function prologue. We push the old base pointer onto the stack (to return to later) and copy the value of the stack pointer to the base pointer. The base pointer serves as the base of the stack frame. We then move the values in the address 0x200a10 and 0x200a0e (2 bit from each other) from the Instruction Pointer to the edx and eax registers. We then add these values and move it to the address 0x200a0a from the Instruction Pointer. The last 2 steps, also known as the function epilogue, pop the old base pointer off the stack and store is back into $rbp and “retq” jumps back to the return address (stored on the stack frame).

* Problem 2.6:







The first 2 instructions are called the function prologue. We push the old base pointer onto the stack (to return to later) and copy the value of the stack pointer to the base pointer. The base pointer serves as the base of the stack frame. We then move the value stored in the address 0x200a0c from the Instruction Pointer to eax. Then, we shift the value by 4 bits to the left. Next, we move this value to the address 0x200a13 in the Instruction Pointer. We then move the value in the address 0x200a01 and 0x2009ff from the Instruction Pointer to edx and eax registers. Here, we use the bitwise and operation on eax and edx. We then move the value contained in the address of 0x200a03 from the Instruction Pointer to eax. Then, we move values from the addresses 0x2009ed and 0x2009eb from the Instruction Pointer to the edx and eax register. We then use the bitwise OR operation. Finally, we move the value contained in the address of 0x2009ef from the Instruction Pointer to eax register. Here, we use the bitwise not operation and move the resulting value to the address 0x2009e1 from the Instruction Pointer. The last 2 steps, also known as the function epilogue, pop the old base pointer off the stack and store is back into $rbp and “retq” jumps back to the return address (stored on the stack frame).