Take Home Test 1

Chue Zhang

Csc343 Fall 2021

Professor Gertner

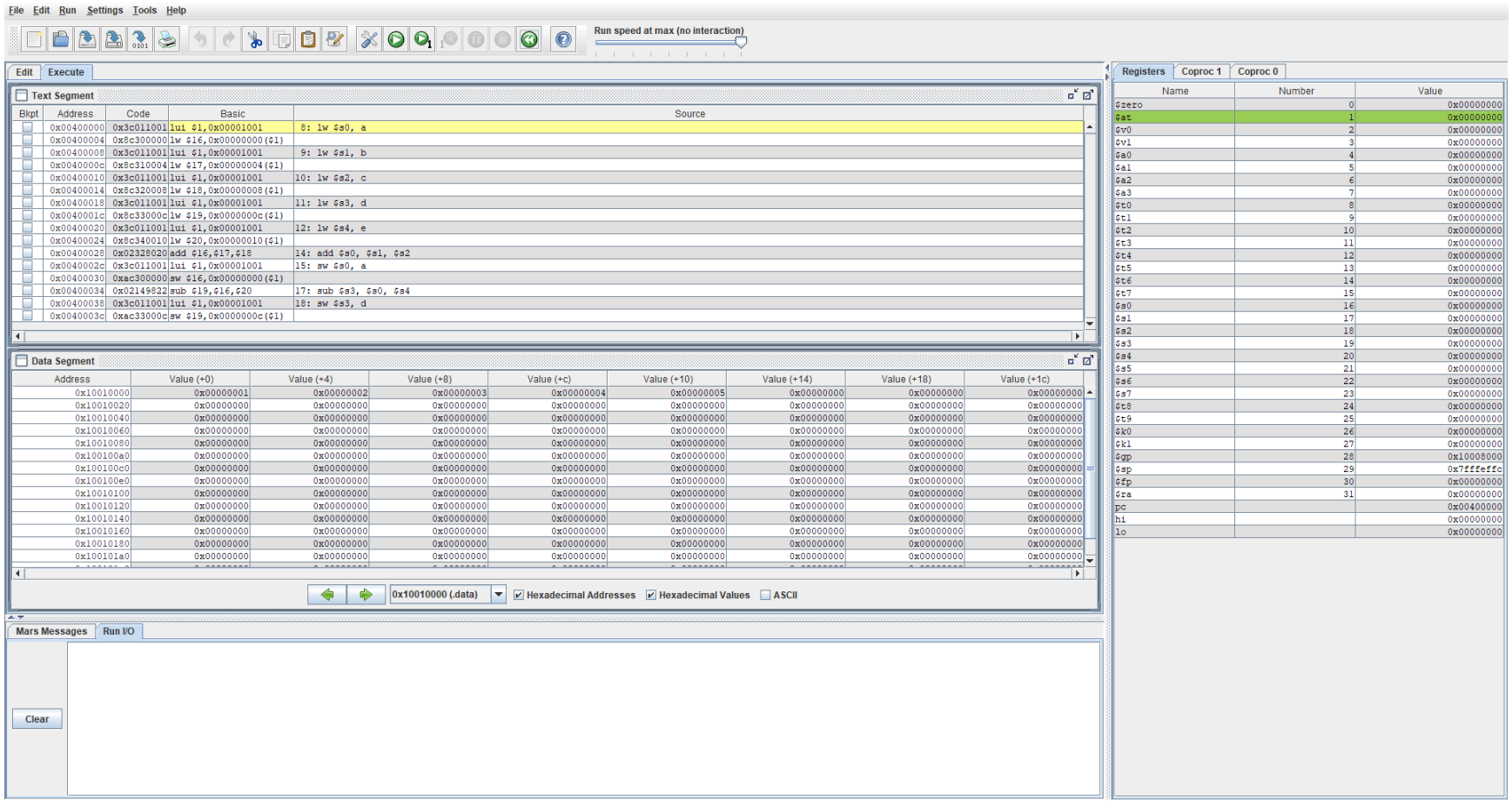
Start Time: Nov 6, 2021, 12:00PM

End Time: Nov 6, 2021, 10:00PM

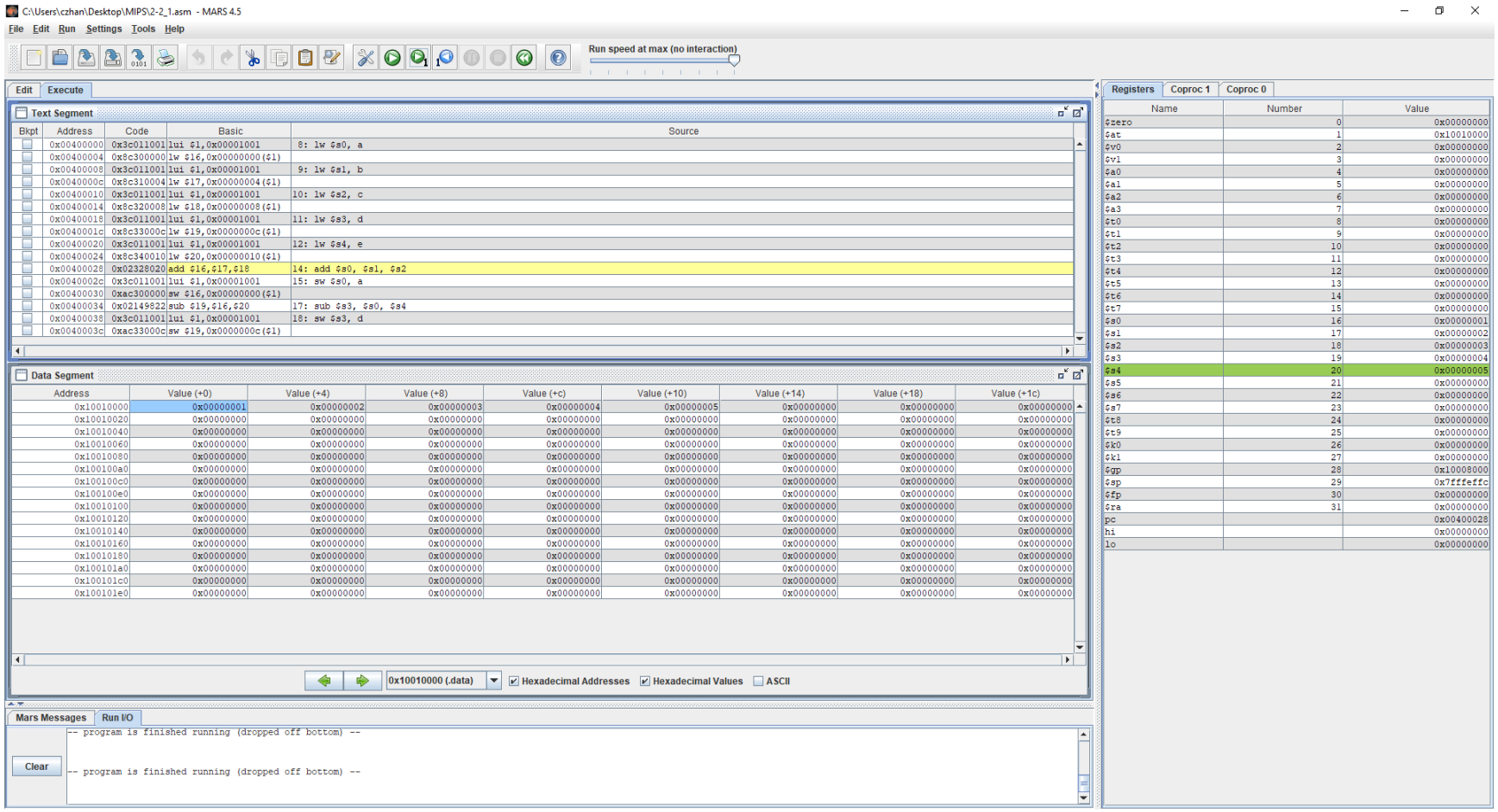
Table of contents

The Objective of this Take-home test is to test our understanding of the difference between the different processors that we use in class since we will be using them more in class over the next couple weeks. I will be conducting 3 different tests on many different codes. Once using the MIPS processor, Once using the Intelx86 processor and once using the GCC compiler. Once tests are done, I will be talking about the differences and similarities between each processor and what makes them unique

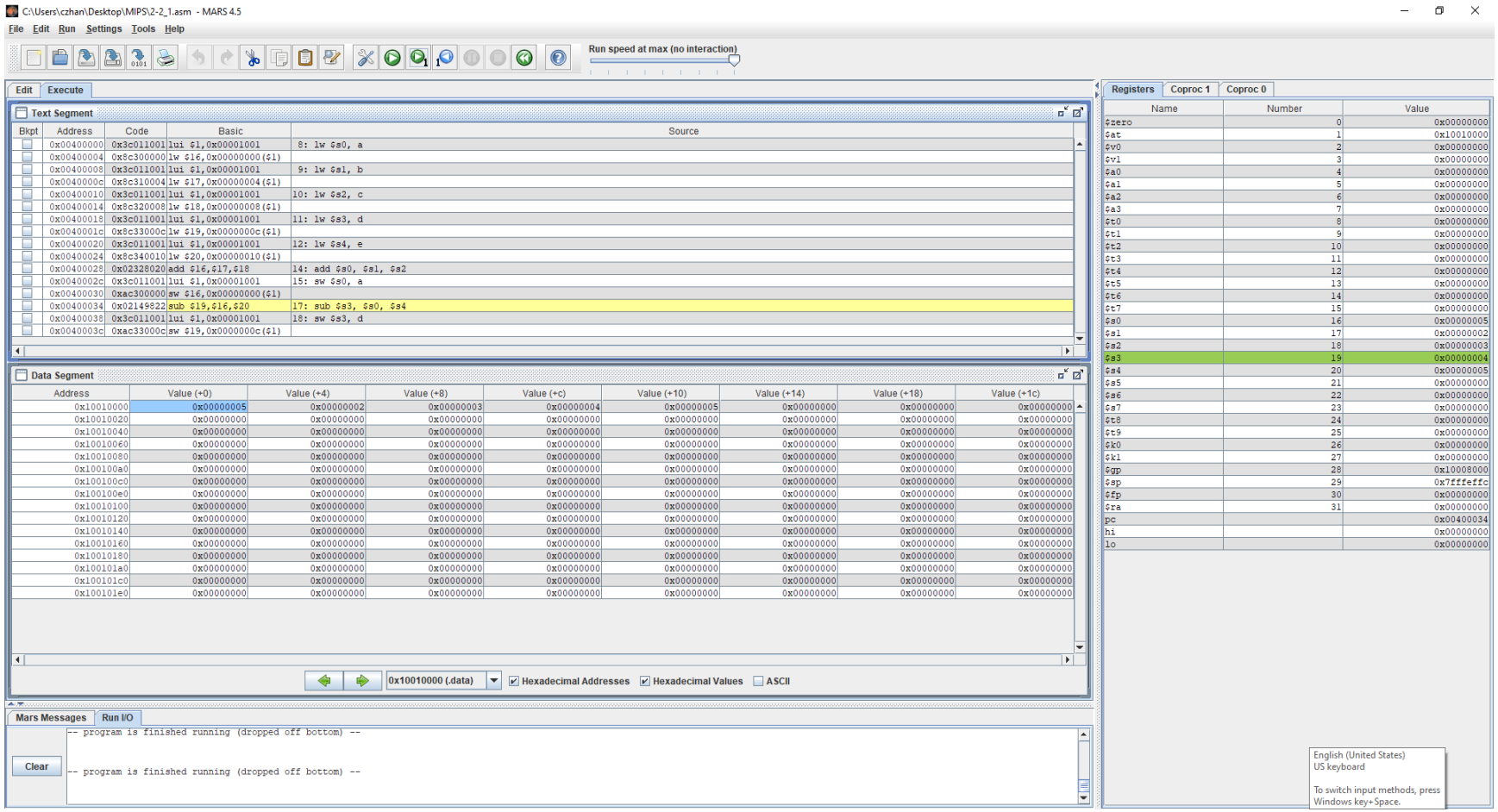
MARS simulator will be used to test assembly machine code whereas the intelx86 and GCC compiler will be performing using C code. Mars simulator provides instructions step by step showing where values are being loaded into, stored into, and written into. Intelx86 will have a disassembly window, a register window and several memory windows that help show what value is stored at what address. Lastly the GCC compiler will be debugging the C code using GDB and comparing things such as $ rip to the instructions listed in the dump file of the C code. What is also provided with GDB is similar to the intelx86 processor however, you have to manually print out each register and static variables



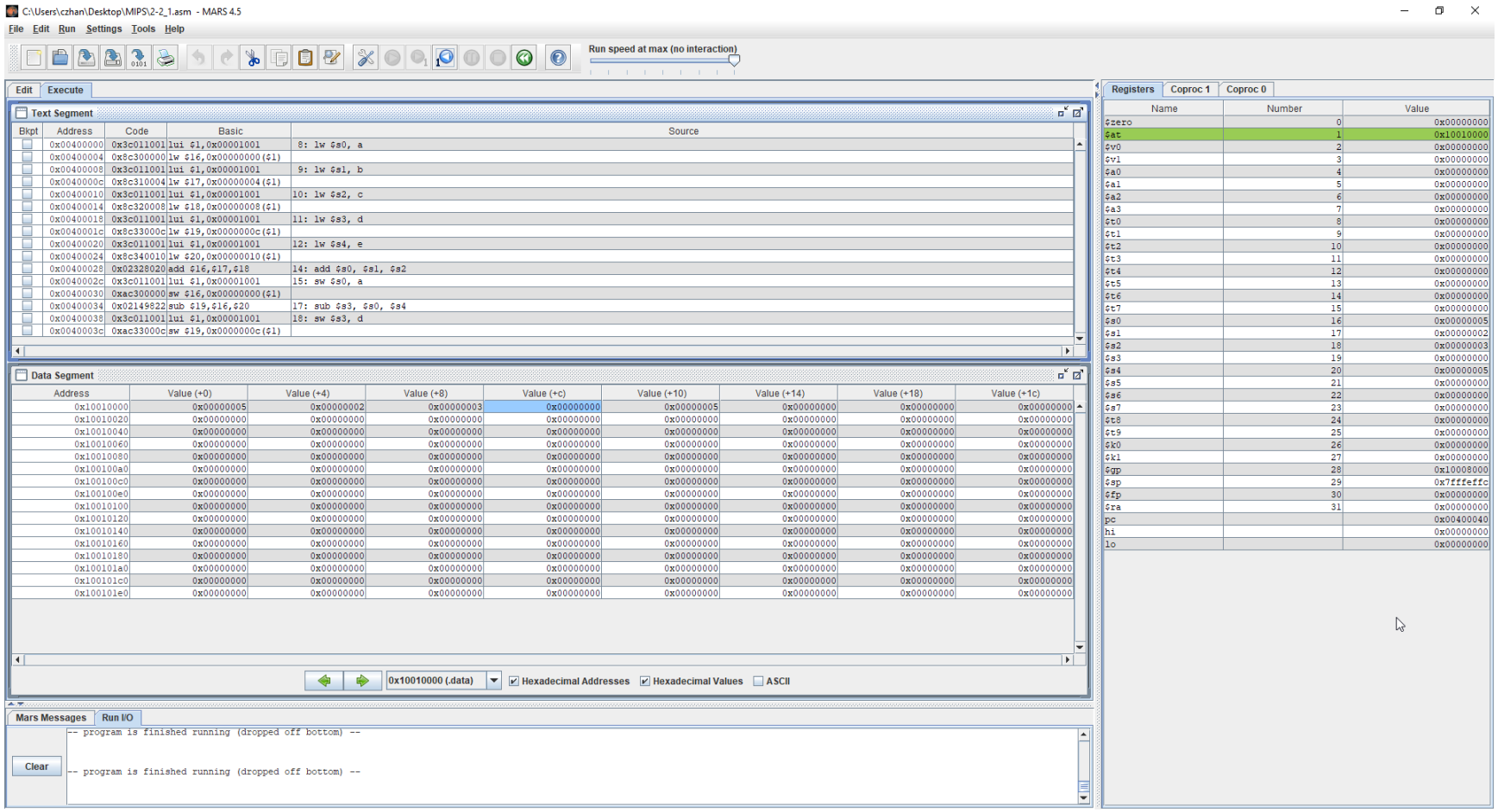
In the screenshot above, what can be observed is the MIPS file for 2.1 being run. Notice that Text segment, Data segment and Register are all present and available to be observed for any changes that occur during run time of the program. In the Registers table, we can notice that there is PC has the value 0x00400000 which indicates where the program is currently at. If we look at compare in the Text segment, we are indeed at the address 0x00400000. Please also notice that $sp or stack pointer which points at 0x7fffeffc which will NOT change due to the fact that all variables are static in this program therefore $sp will not change.



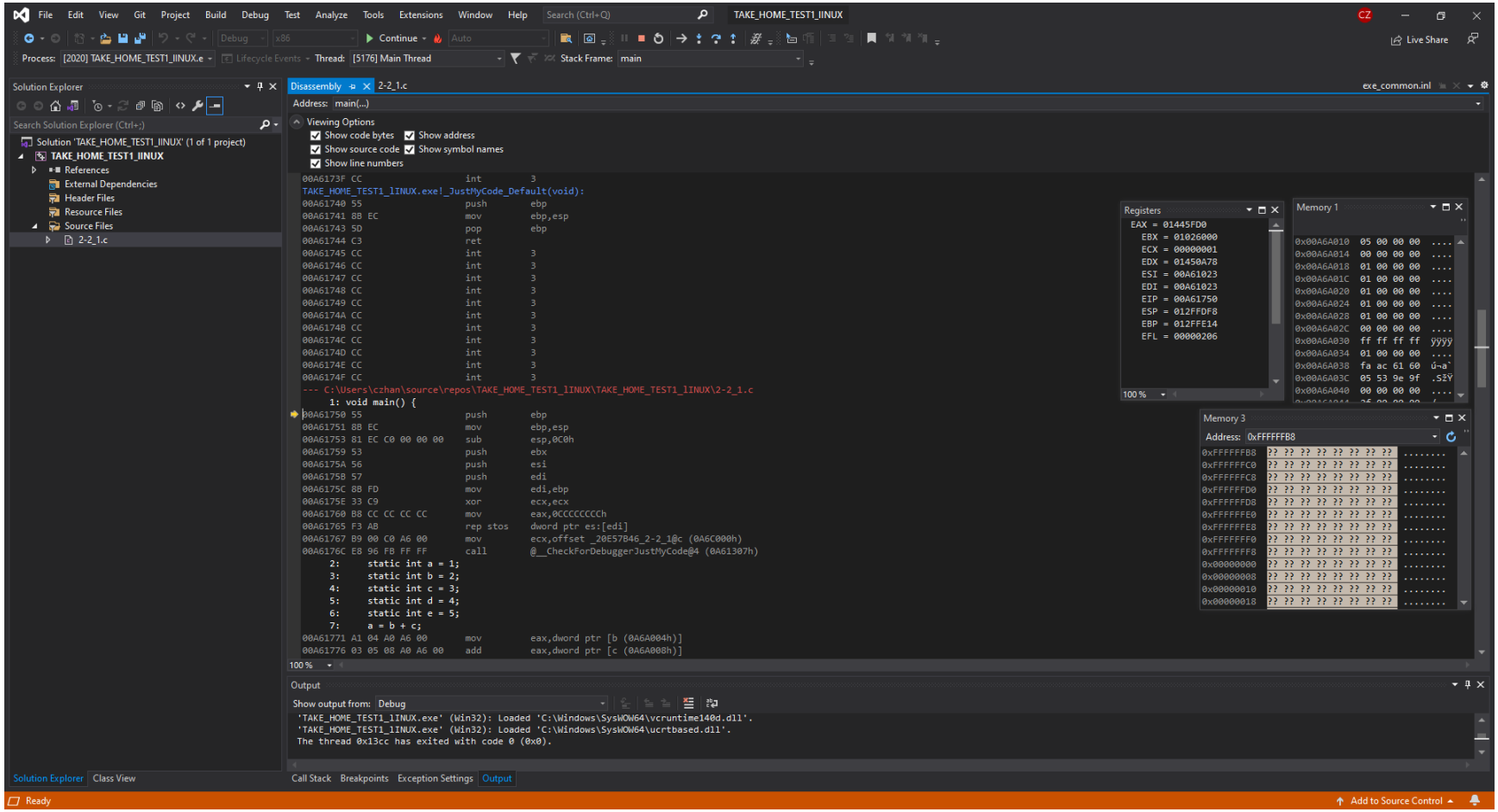
In this screenshot, I have moved several steps further ahead and I want you to notice the Text segment as well as the Register. Look at $s0, $s1, $s2, $s3, $s4 which when compared to the previous screenshot, there are not values stored into them. That is because in the asm file, we assigned values a,b,c,d,e to 1,2,3,4,5 at $s0 - $s4. You may also take a look at the Text segment which shows the actual machine instruction where a,b,c,d,e is assigned to 1,2,3,4,5.



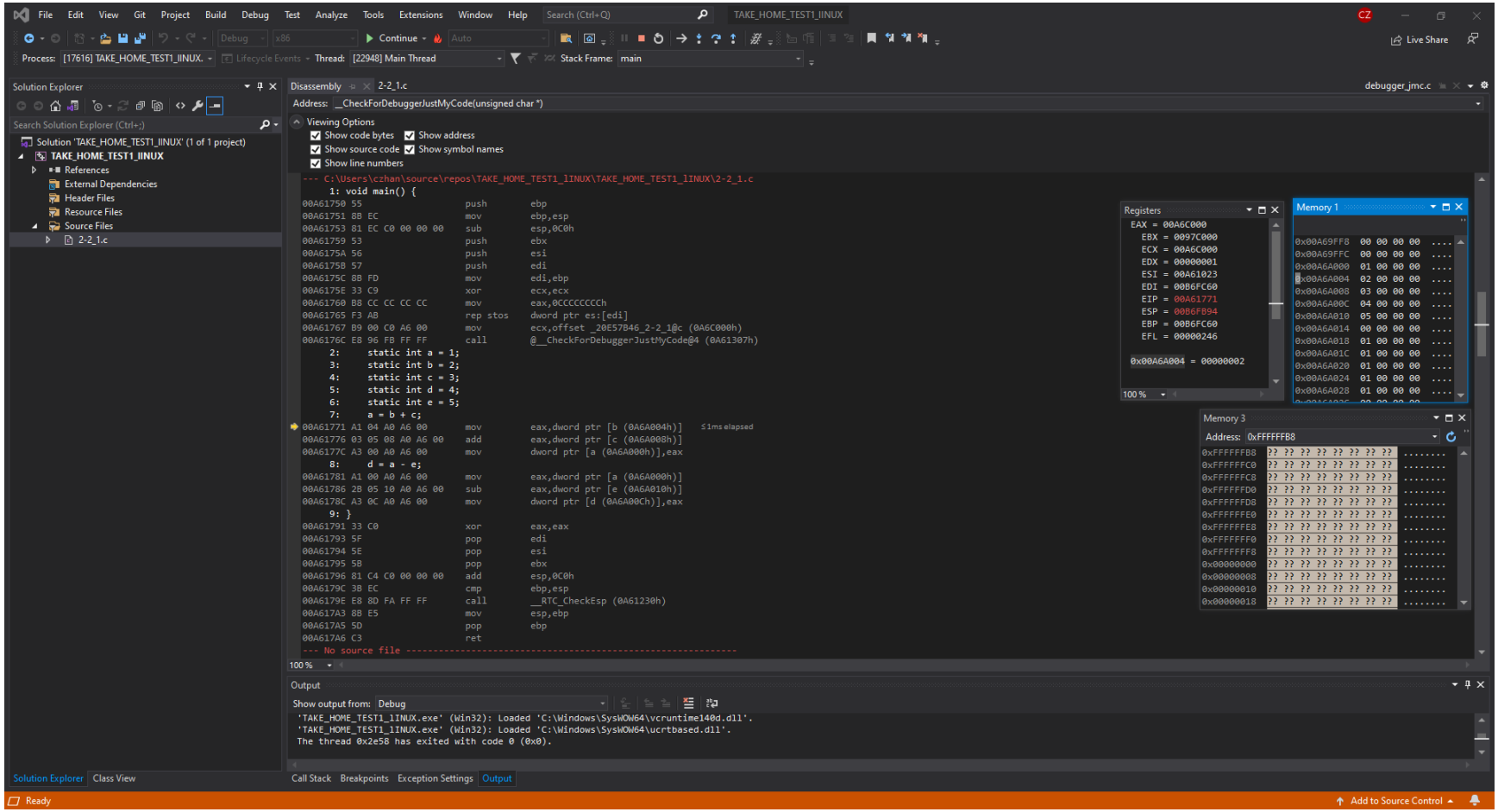
Please notice in this screenshot at $s0 where the value has now changed to 5 which should happen because the machine instruction says $s0 = $s1 + $s2 or a = b + c and we can observe that it has performed that instruction successfully. You can also further observe the Data segment as now value at offset +0 has now changed to value 5 where it was previously 1.



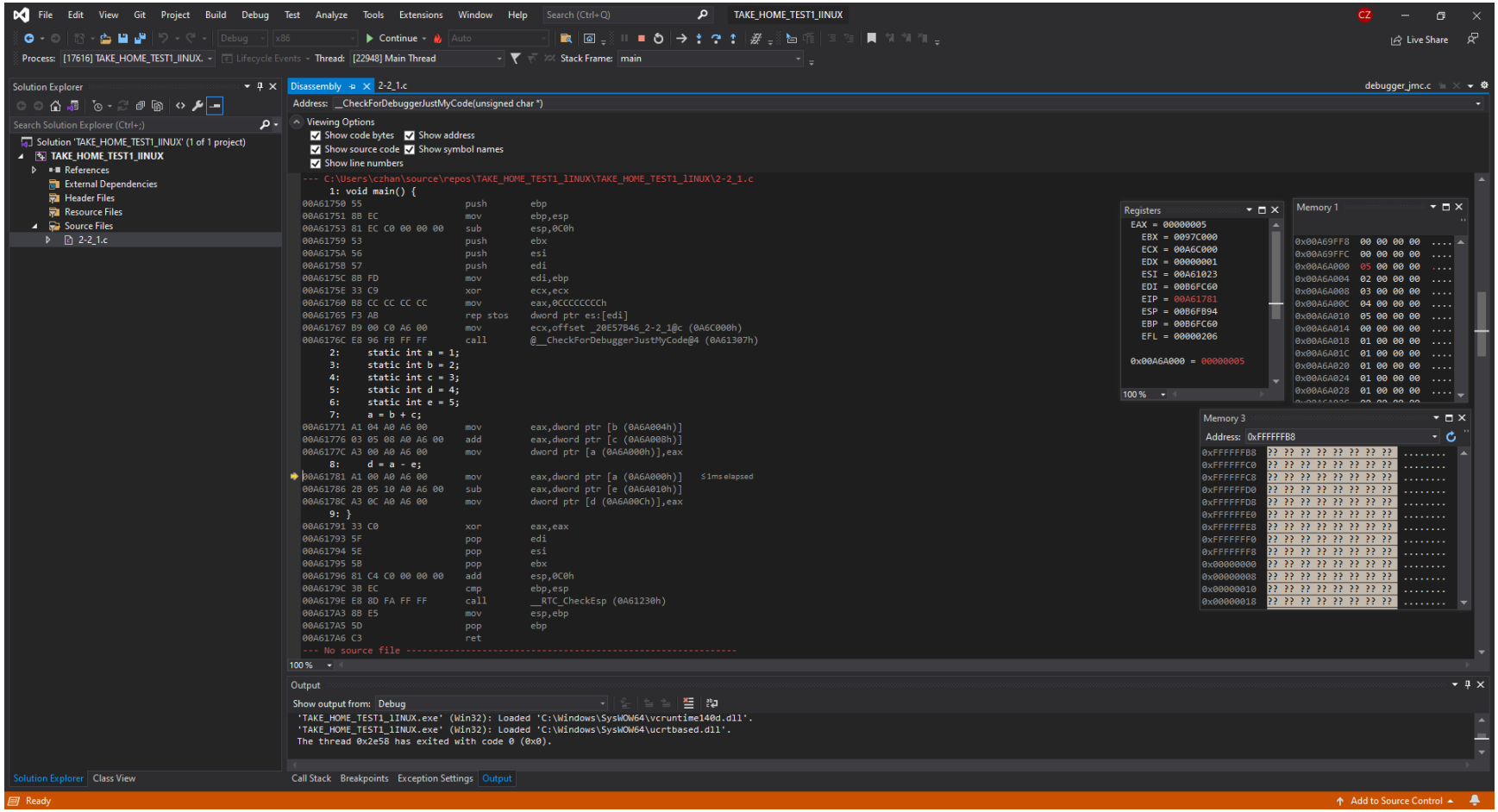
Lastly, this is the final instruction that is being done where d = a -e. If you look at the PC, it is pointing towards the last instruction, and you can also observe in $s3 that its value is now 0 as well as Data segments 0x10010000 at offset + c where the value is now 0 because the instruction of d = a – e has taken place. Lastly, as I mentioned previously, $sp has not changed at all because all variables instantiated are static variables and there are no local variables therefore the $sp does not change throughout the span of the program run time.



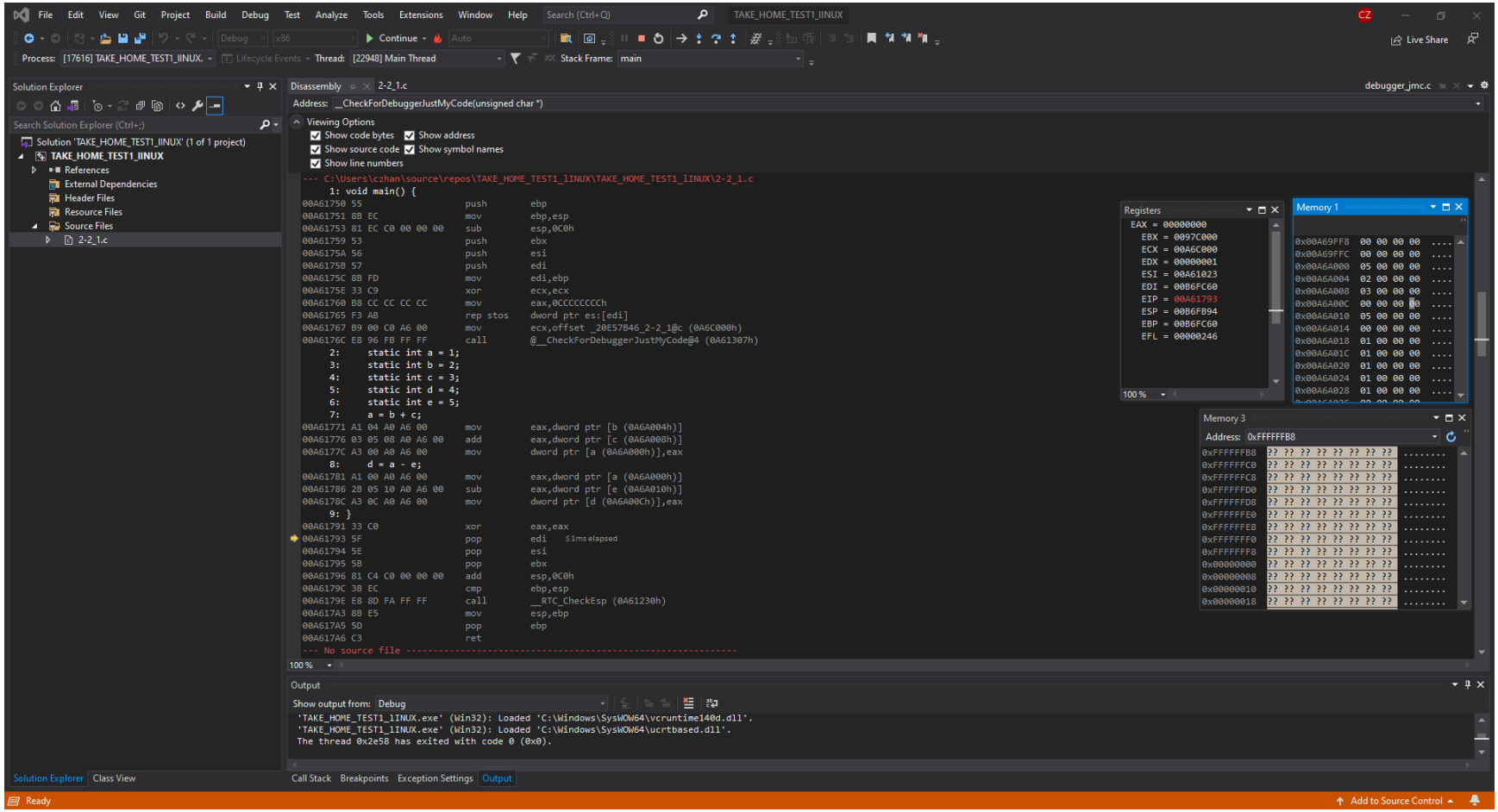
In the screenshot above, it showcases the disassembly window, the registers and memory window of file 2-2_1.c that was presented in the exam contents. The disassembly window shows the instructions of what occurs and at what addresses and the registers show the instructions such as the base pointer, instruction pointer and such and finally the memory which shows what is stored. If you notice at the register window, at EIP, it points to 0x00A61750 which indicates what instruction it will be conducting at the next step



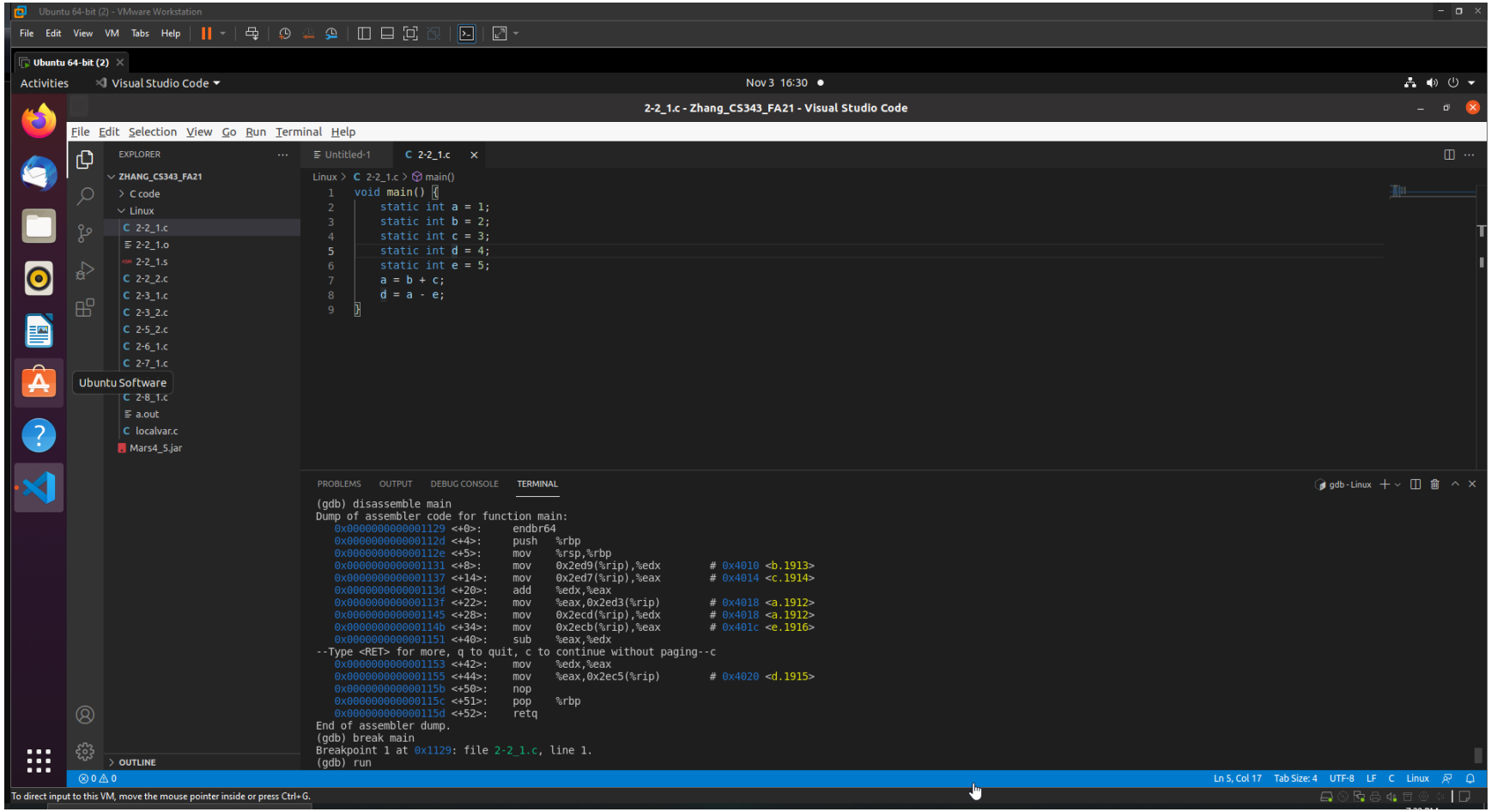
Please notice the Register window where tie current RIP is now 00A61771 and if you look at the disassembly window with the yellow arrow, it does show which instruction it is about to run. Furthermore if you notice the address 0x00A6A004 = 2, and look at the memory window, it shows in detail that 0x00A6A004 = 2. Furthermore, if you continue to observe the memory window 1, you can see 0x00A6A000 = 1 and every offset +4 shows the value stored for static integers a,b,c,d,e.



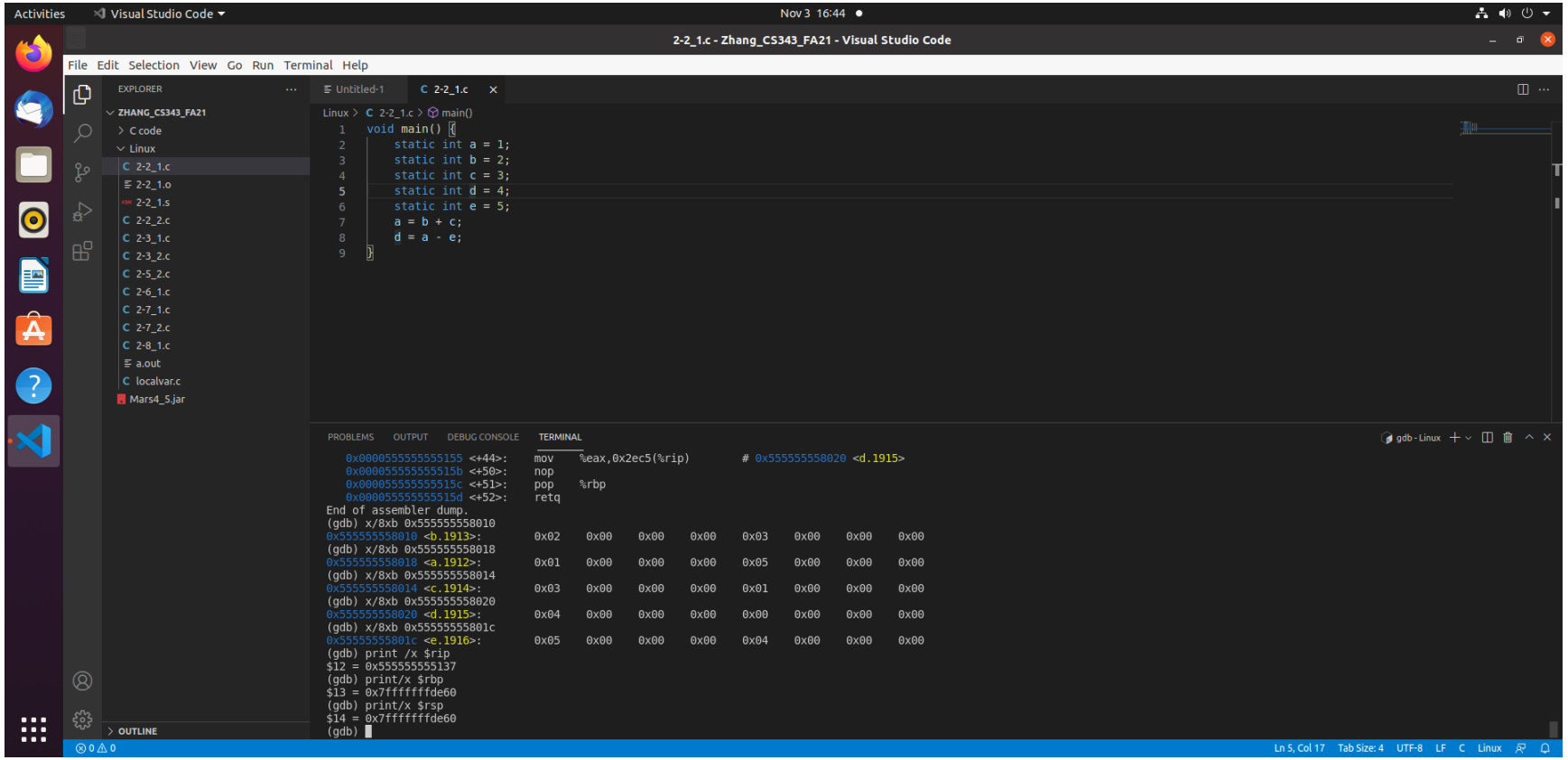
Now if you look at where the instruction is and look at the register window, it now says 0x00A6A000 = 5 which means that a = b + c has just been completed and now a = 5 which is stored at the address 0x00A6A000. You can also observe this in the disassembly window as it performs mov on a to store the value of 5.



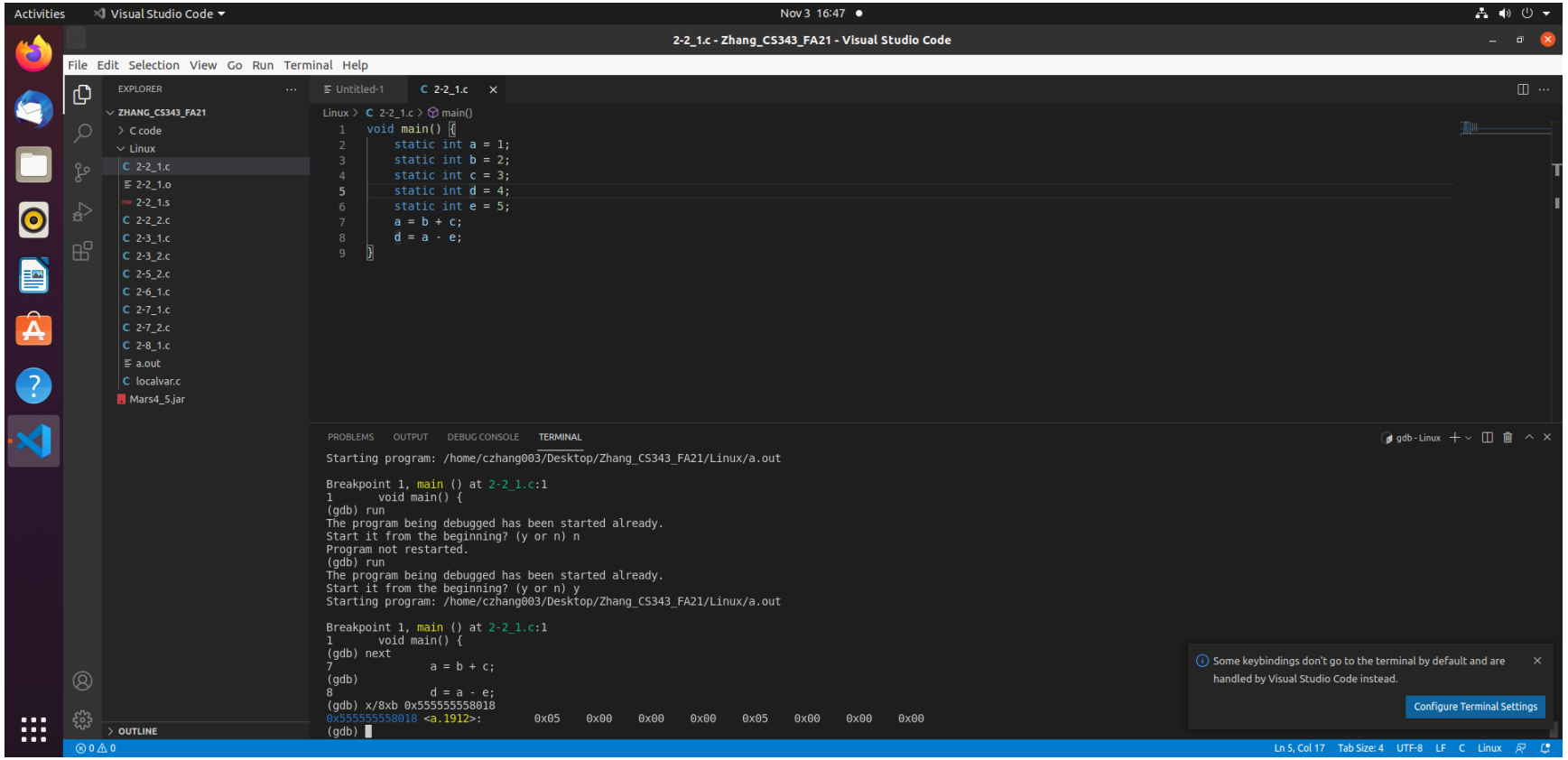
Please observe this screenshot carefully as if we look at the address 0x00A6A00C in the Memory window. We can notice that the value is now 0 which means that the instruction d = a – e has been performed successfully as 5 – 5 = 0 and it is now stored in the address of d.



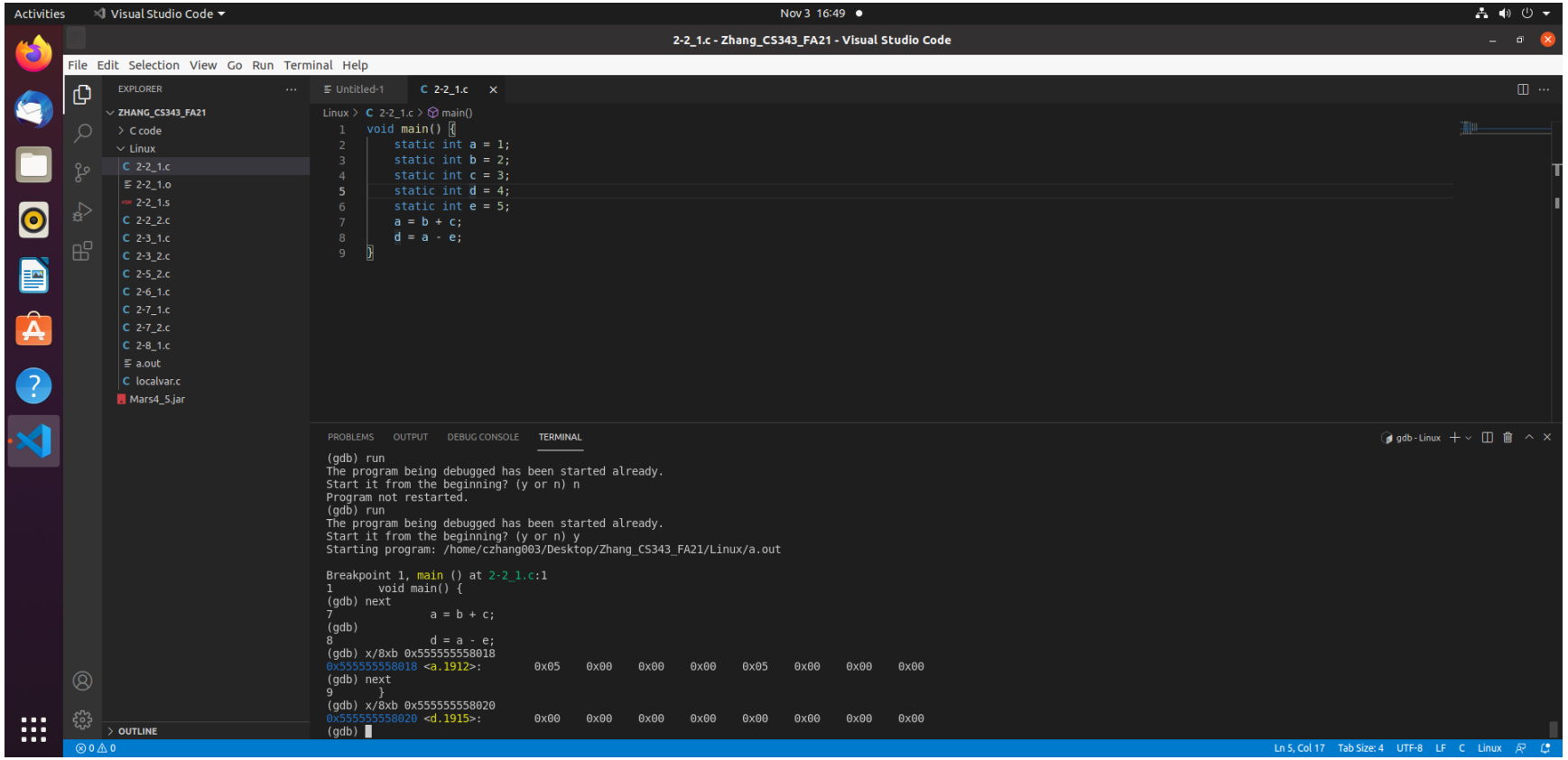
In the screenshot above, it showcases the code as well as the dump of the assembler code which we will be referencing often. %rbp is the base pointer, %rsp is the stack pointer, %rip is the instruction pointer



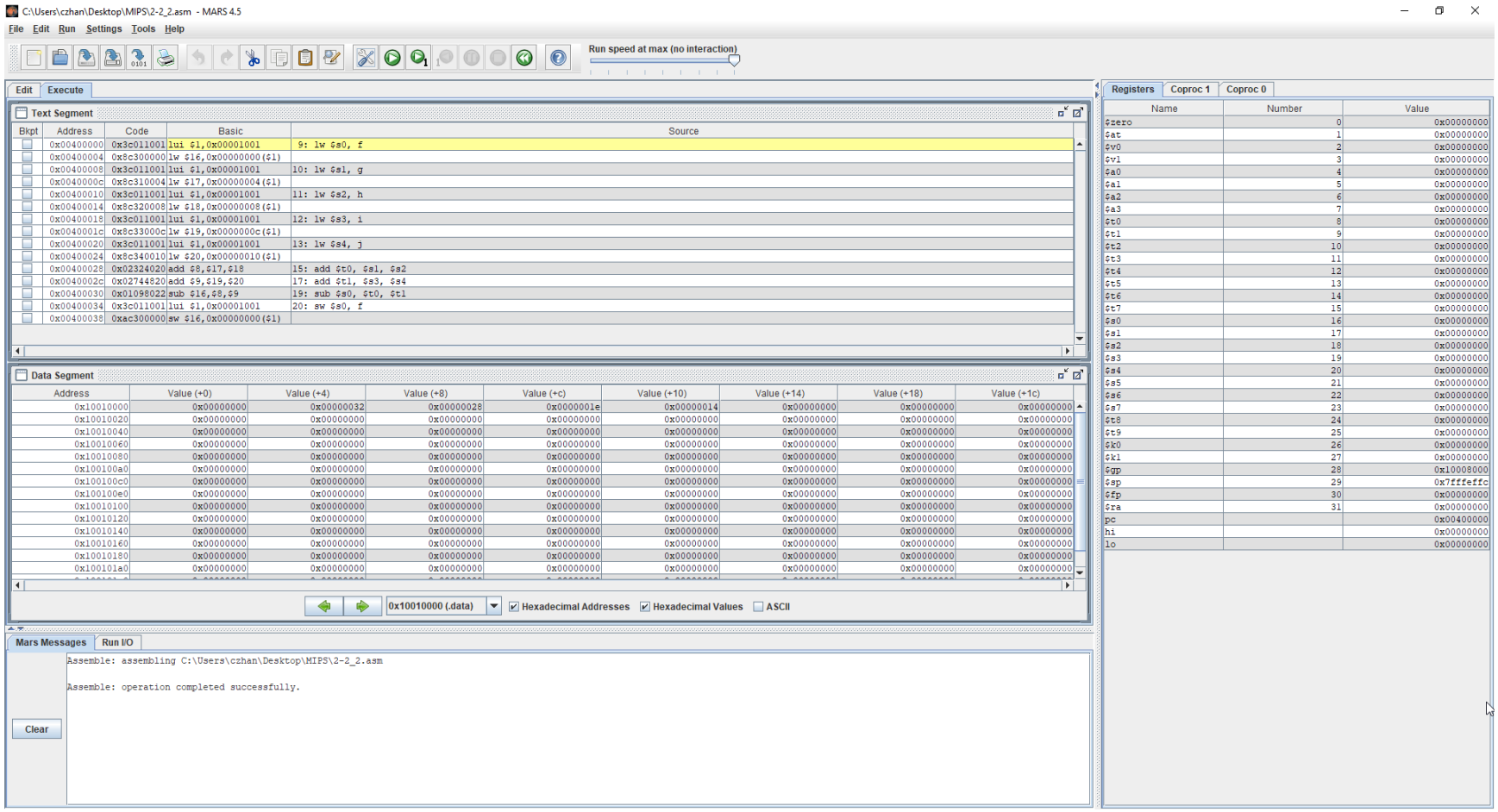
Look carefully at the terminal in this screenshot and you will notice that I have printed out the values of a,b,c,d,e which you can notice that each has their own corresponding values of 1-5. I have also printed out the instruction pointer which points to the next instruction at $12 = 0x5555555555137 and we can also notice the base pointer and the stack pointer being the same thing. Stack pointer will NOT change throughout the program life time as there are NO local variables and only static variables.



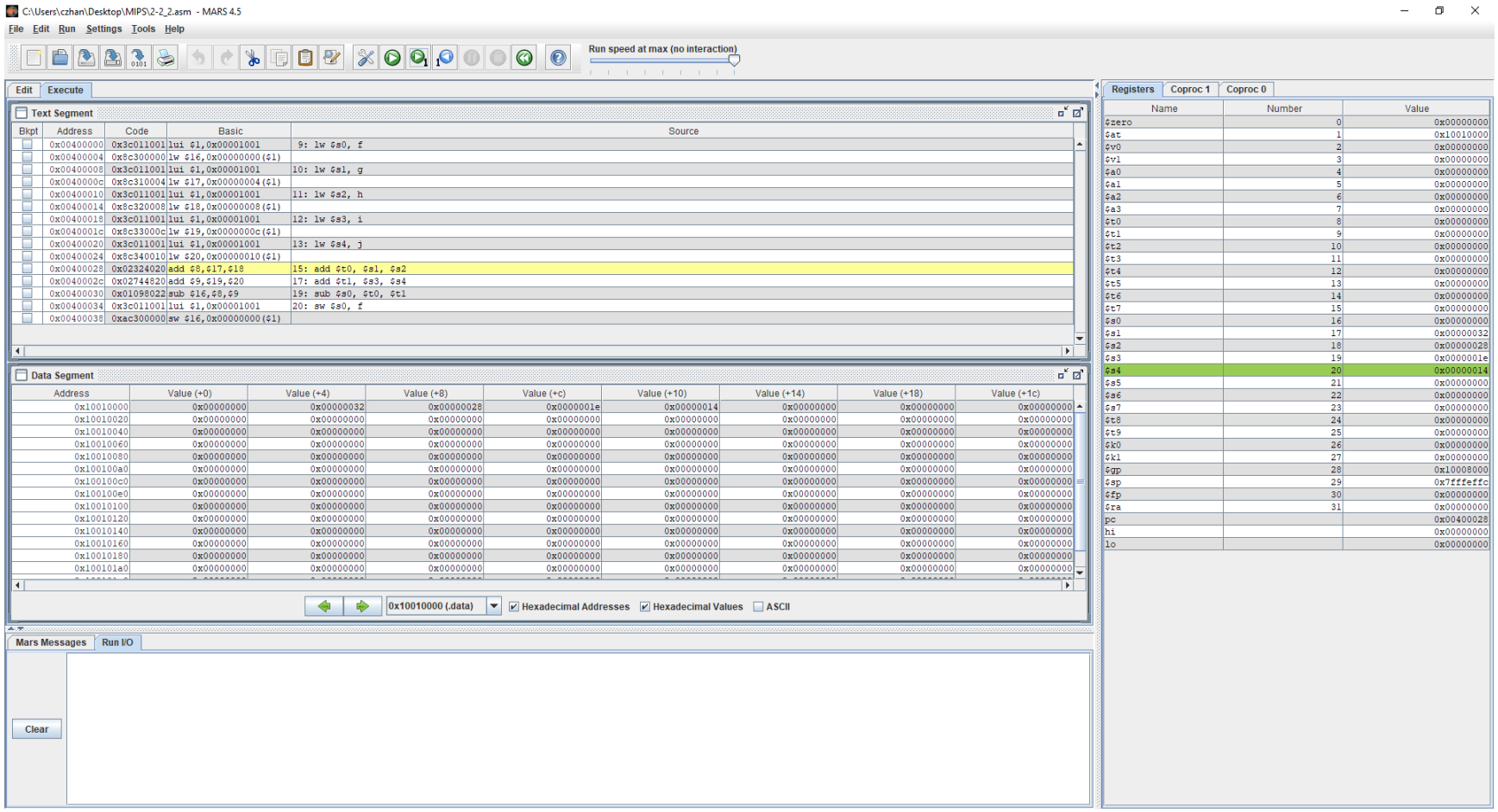
In this screenshot, notice that value of A is now at 0x05 whereas previously it was 0



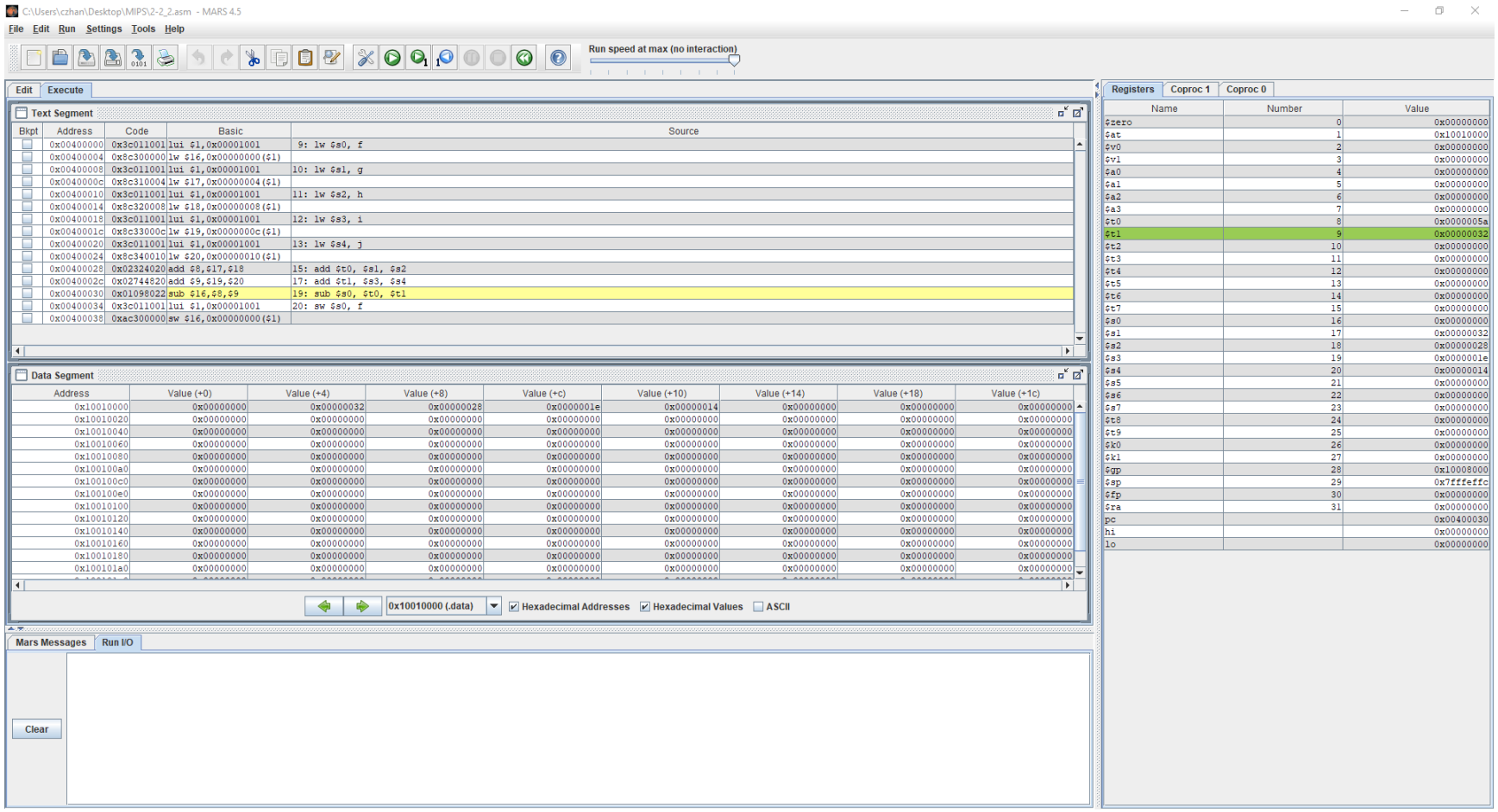
In the next instruction step, what was performed was d = a – e which should be 5 – 5 = 0 and if we look at the value of d, it is now 0x00 which means that the program has run successfully without any error and 0 is now stored into the address of d.



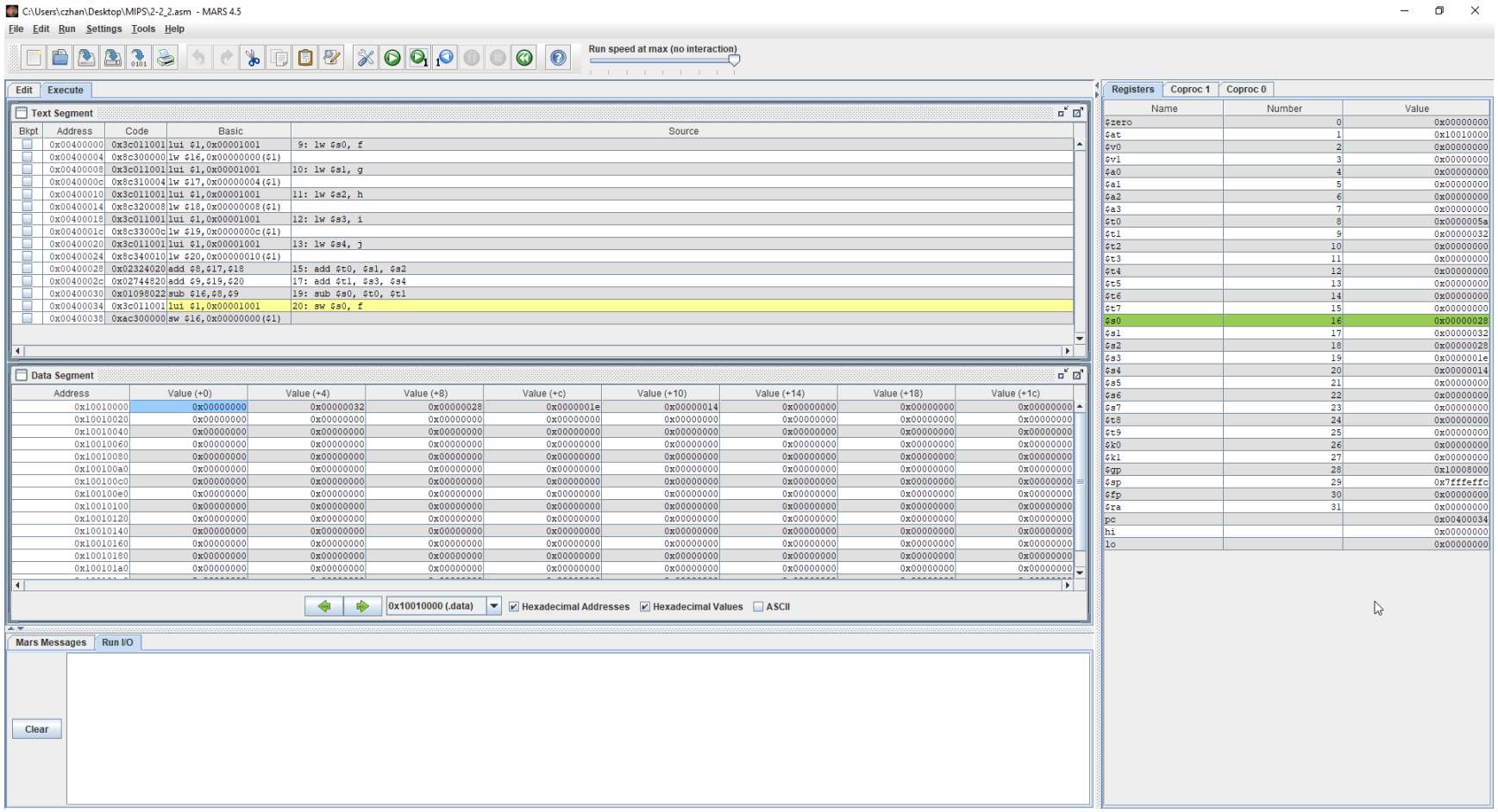
In the screenshot above, what is showcased is the machine instruction that is to be done, the register window and the Data segment. If we look at PC, we can see the instruction to be run which is 0x004000000 and we can also see $sp. Since all variables are strictly STATIC and there are no local variables, $sp will not be changing. You will also notice this throughout the other screenshots.



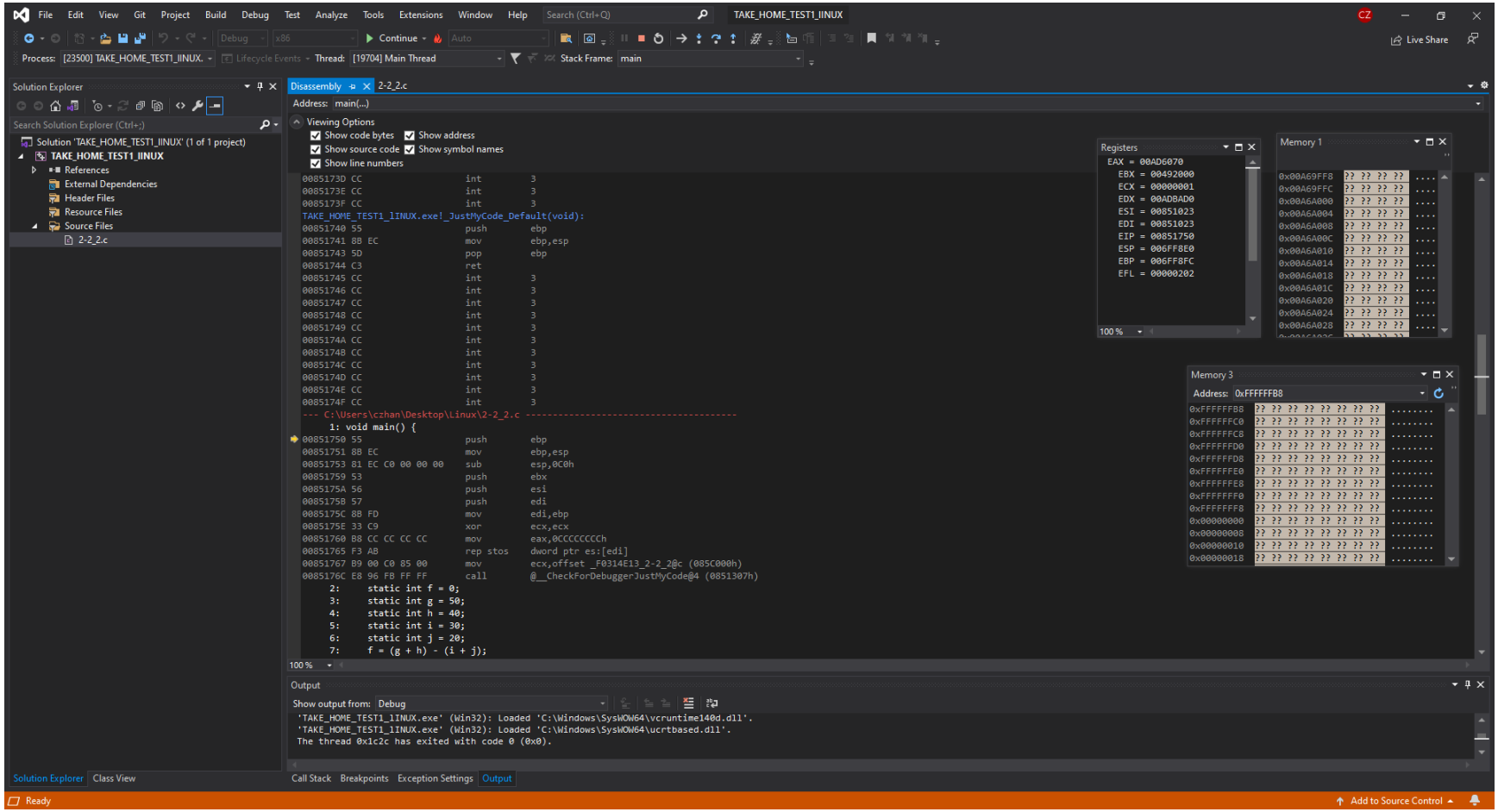
In the screenshot above is several instruction steps after the initial one as you can tell by the PC which is now pointing at 0x00400028. You can also confirm that this instruction is going to be done if you look over at the Text segment. Furthermore, please look at $s0 - $s4 which now have values stored in them as the instructions that were run were to store the values into f,g,h,I,j. Please also look at the Data segments offset values to confirm that these values are being stored properly as at every offset interval, there is a value stored in it.



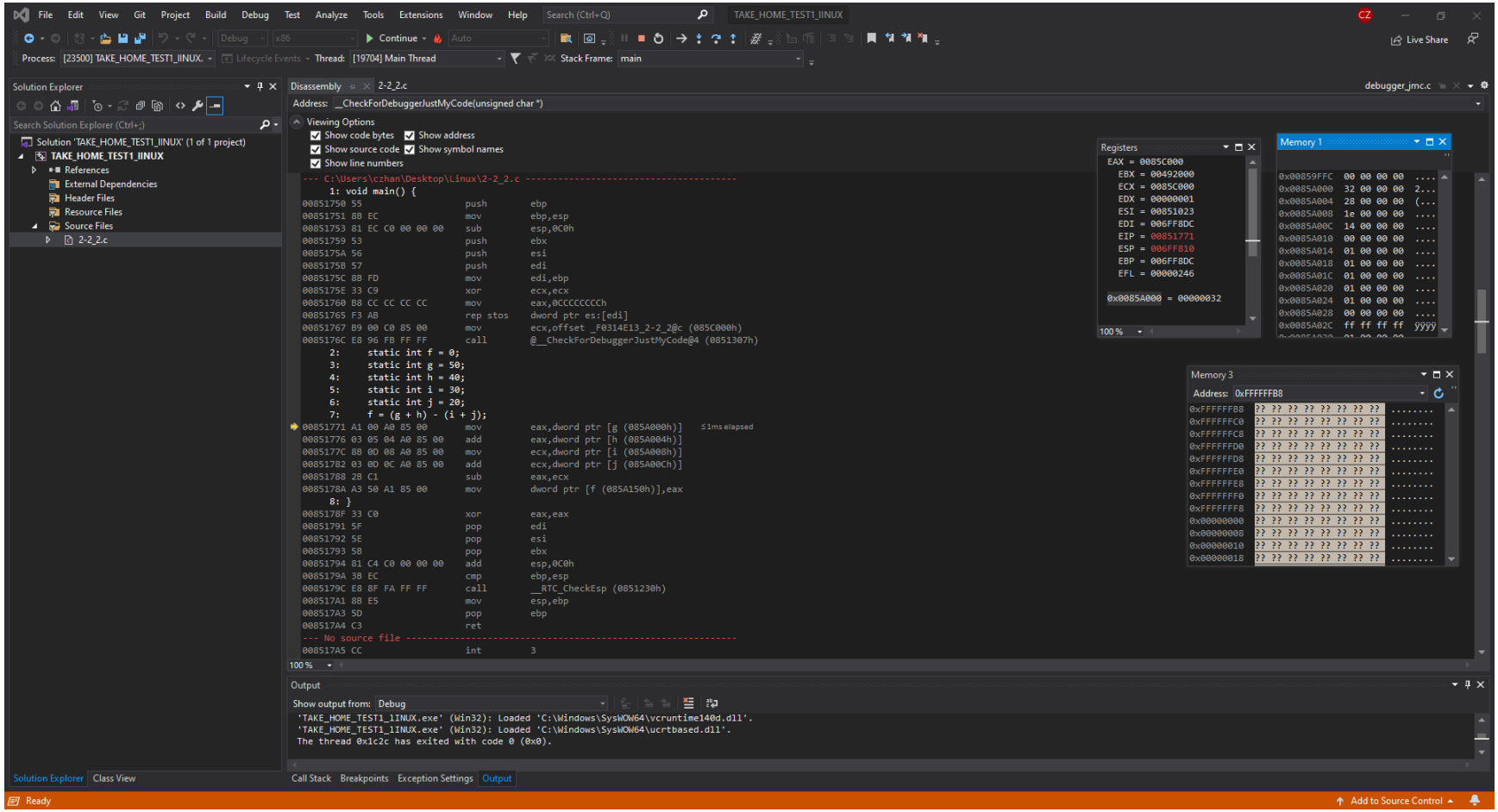
In the above screenshot, we move forward by two instruction steps so that we calculate t0 = g + h and t1 = i + j. G has a value of 50 and h has a value of 40 so 40 + 50 = 90 = 5a and I = 30 and j = 20 so 30 + 20 = 50 = 32 which indicates that the program has run correctly so far. You can also confirm these values over in the Register window at $t0 and $t1. As mentioned previously as well, please look at $sp and notice that the value has not changed because there are no local variables.



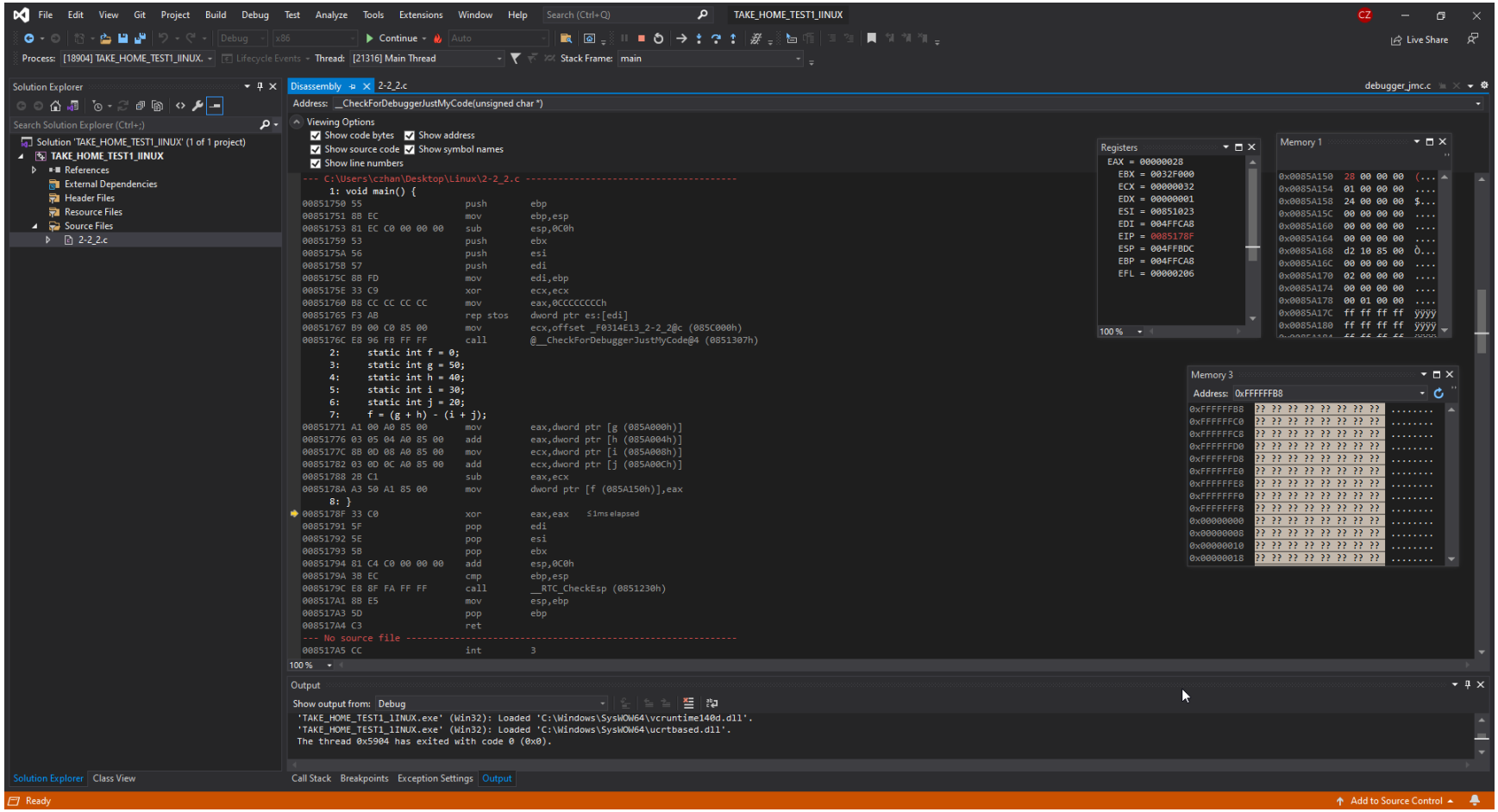
Lastly, please look at this screenshot and notice that the what was performed was f = t0 – t1 which is 90 – 50 = 40 = 28. You can observe the value at $s0 as that is what the machine instruction indicates that is where f is to be stored. $s0's previous value was 0 and now it has been replaced with the value of 28(hex). You can also look at the Data segment with offset + 0 to notice that the value has changed



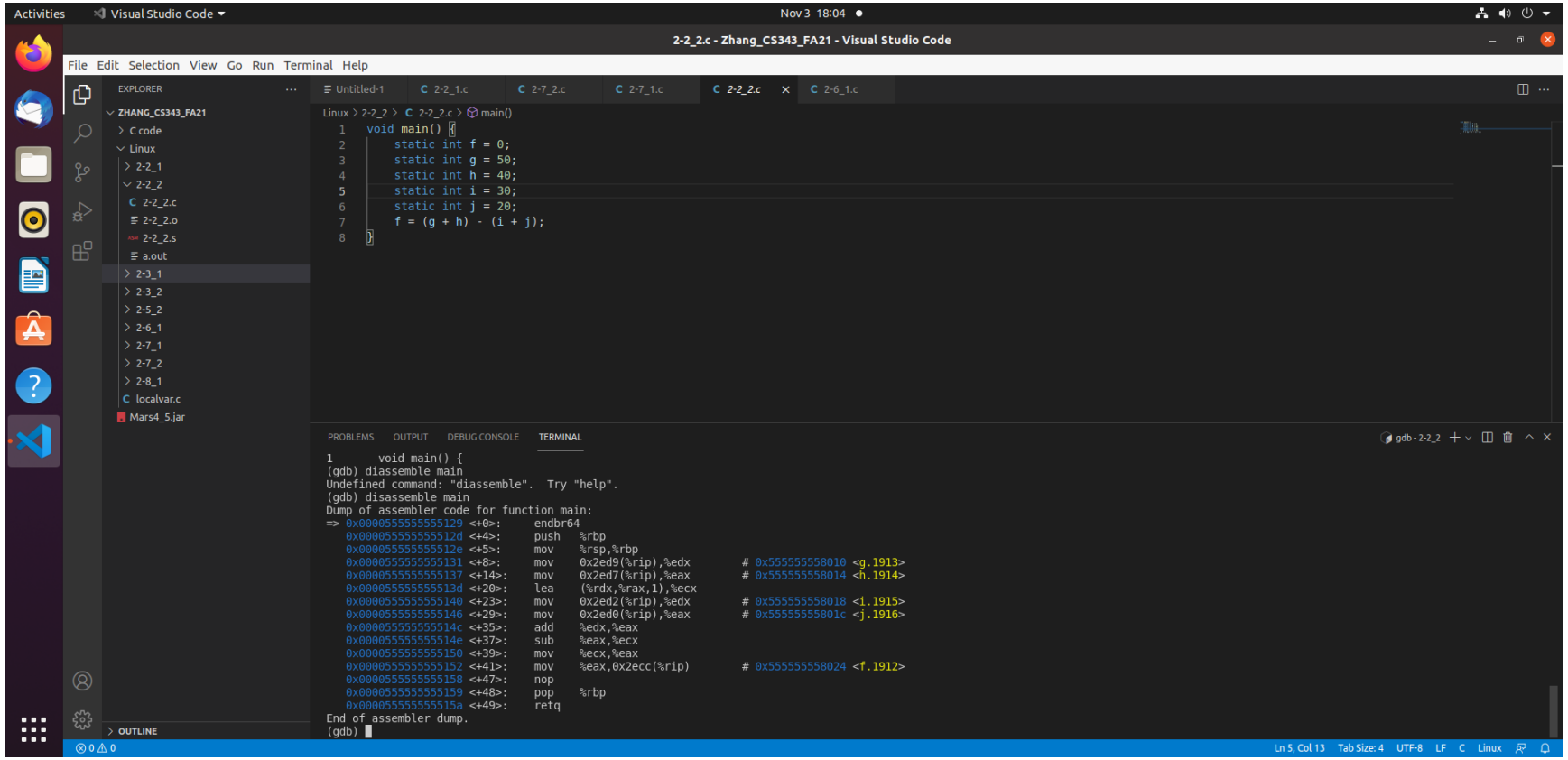
In the screenshot above, what is showcased is the disassembly page, the Memory windows, and the register window. The register window showcases important registers such as the EIP which is the instruction pointer which points towards the next instruction that it will perform; in this case it is 000851750. You can confirm this by looking at the disassembly window for confirmation. You can also see the EBP or the base pointer and the ESP which is the stack pointer.



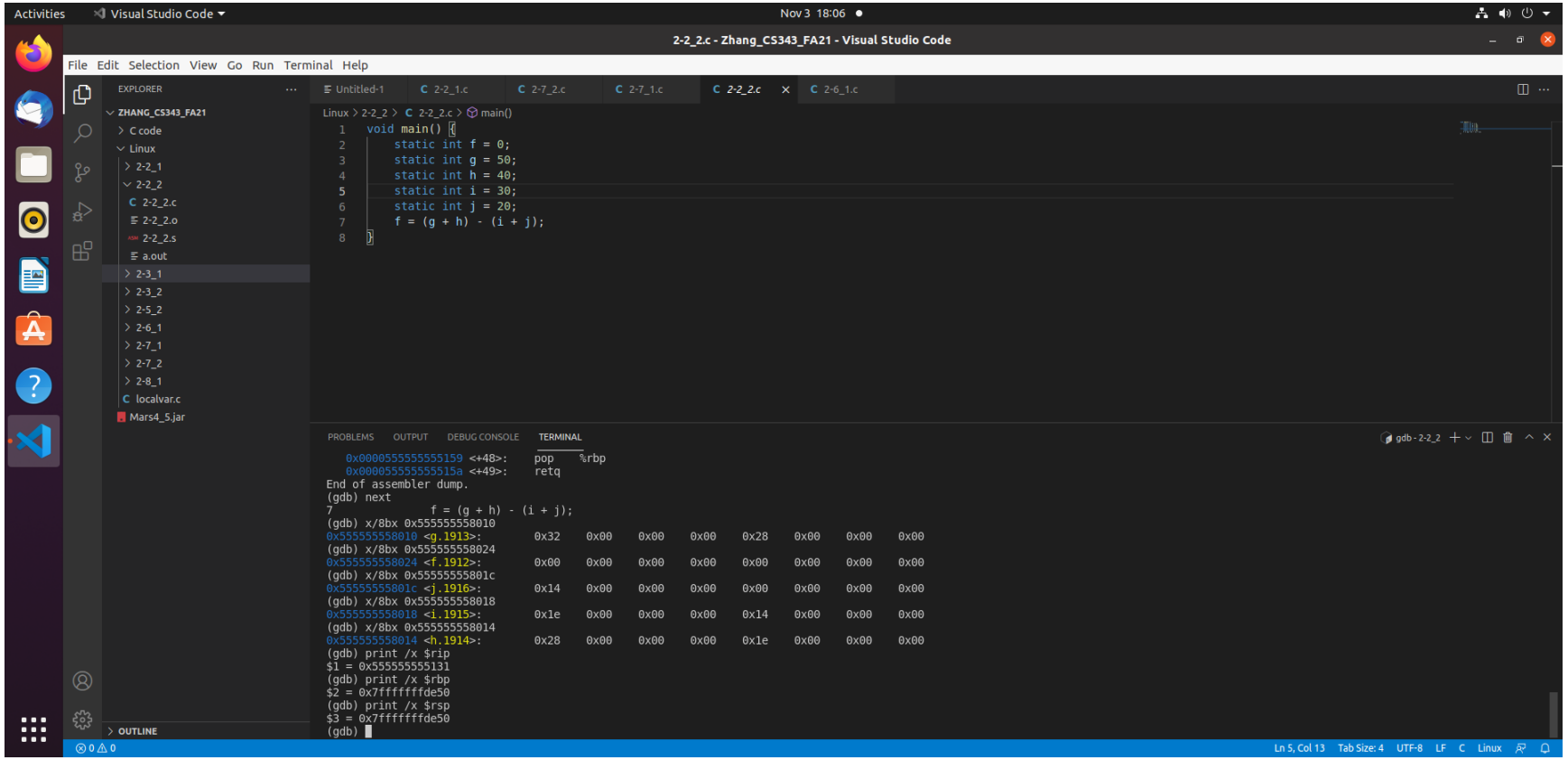
In the screenshot above, what is showcased over in the Memory window is the values of f,g,h,I and j being allocated to addresses, If you look at Memory window 1, you can see the values 00, 32, 28, 1e and 14 which are hex values for 0, 50, 40, 30, 20.



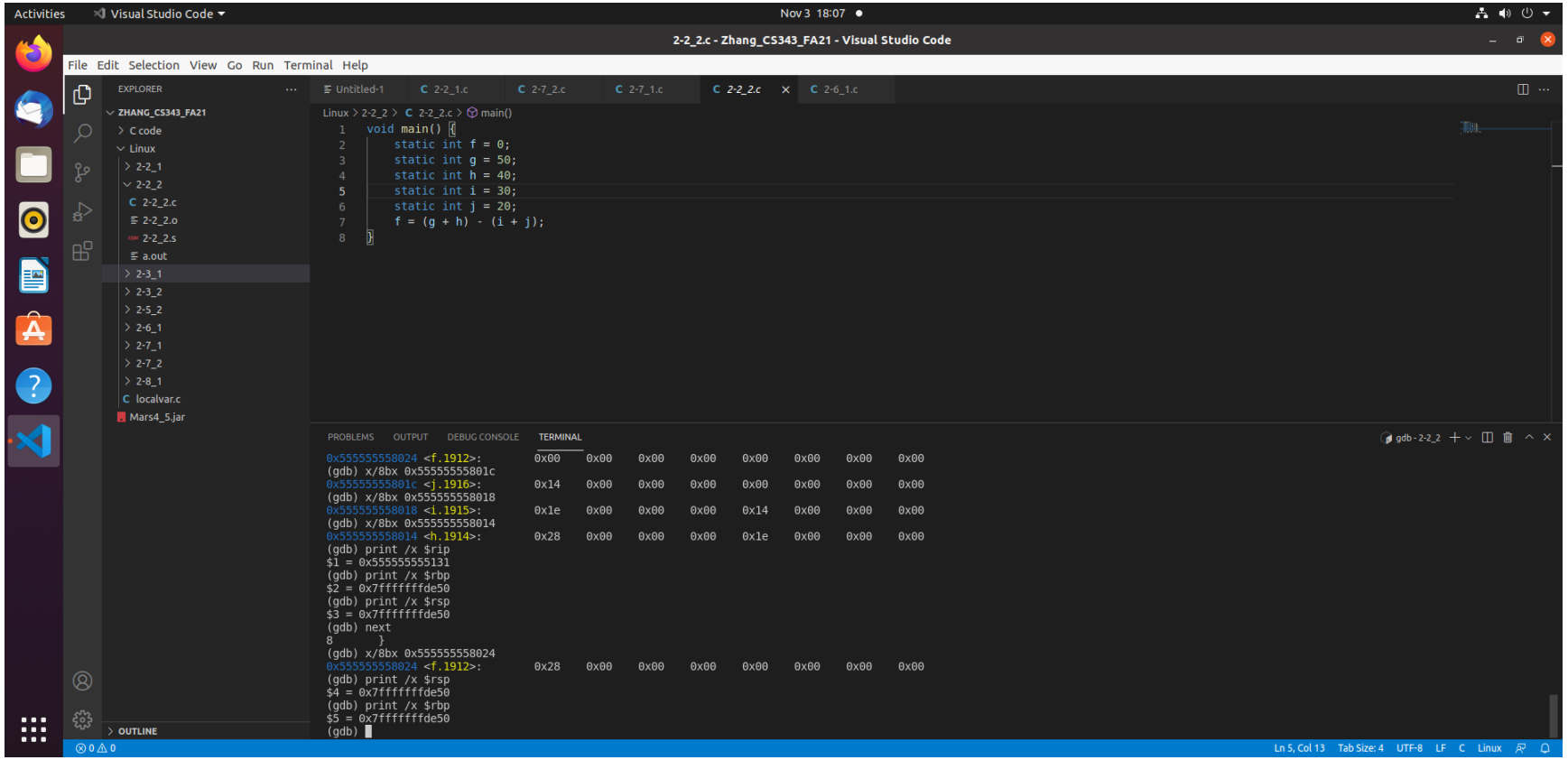
Lastly, in this screenshot, please notice that the function f = (g+h) – (I + j) has been performed as the first address at the top in Memory 1 has been replaced by 28. F = (50 + 40) – (30 + 20) = 40 = 28 in hex so this confirms that the test has been successful and there has been no errors running the program. You can also look at EAX which stores the arithmetic value that has just been calculated. EAX = 28



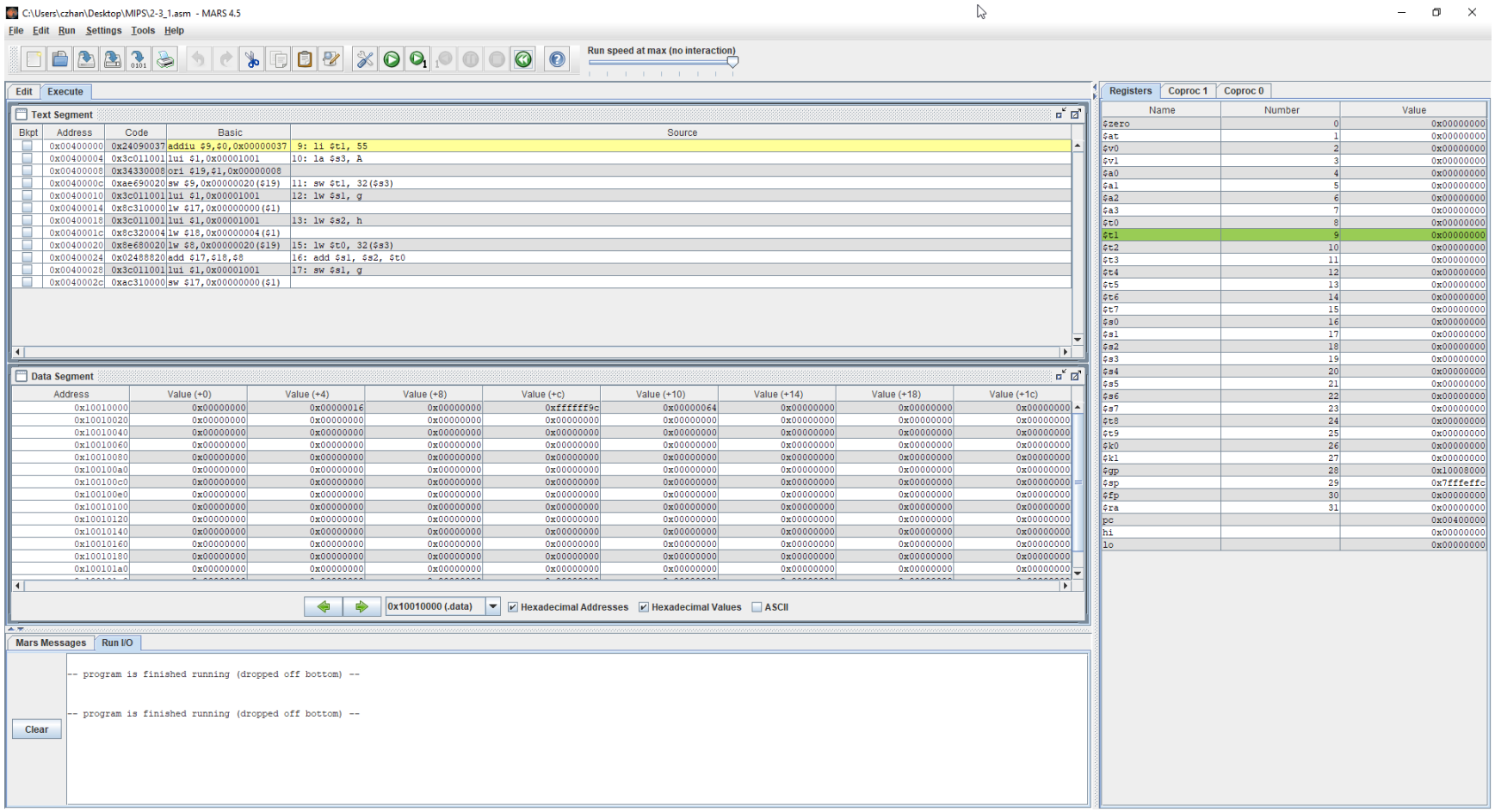
In the screenshot above, it showcases running the in GDB after compiling in GCC. In the terminal, please notice the dump of the code as it will be relevant throughout the other showcases. What we can notice in the dump file is the address of f,g,h,I and J and that is how we are going to be assuring ourselves that the program is running correctly and efficiently



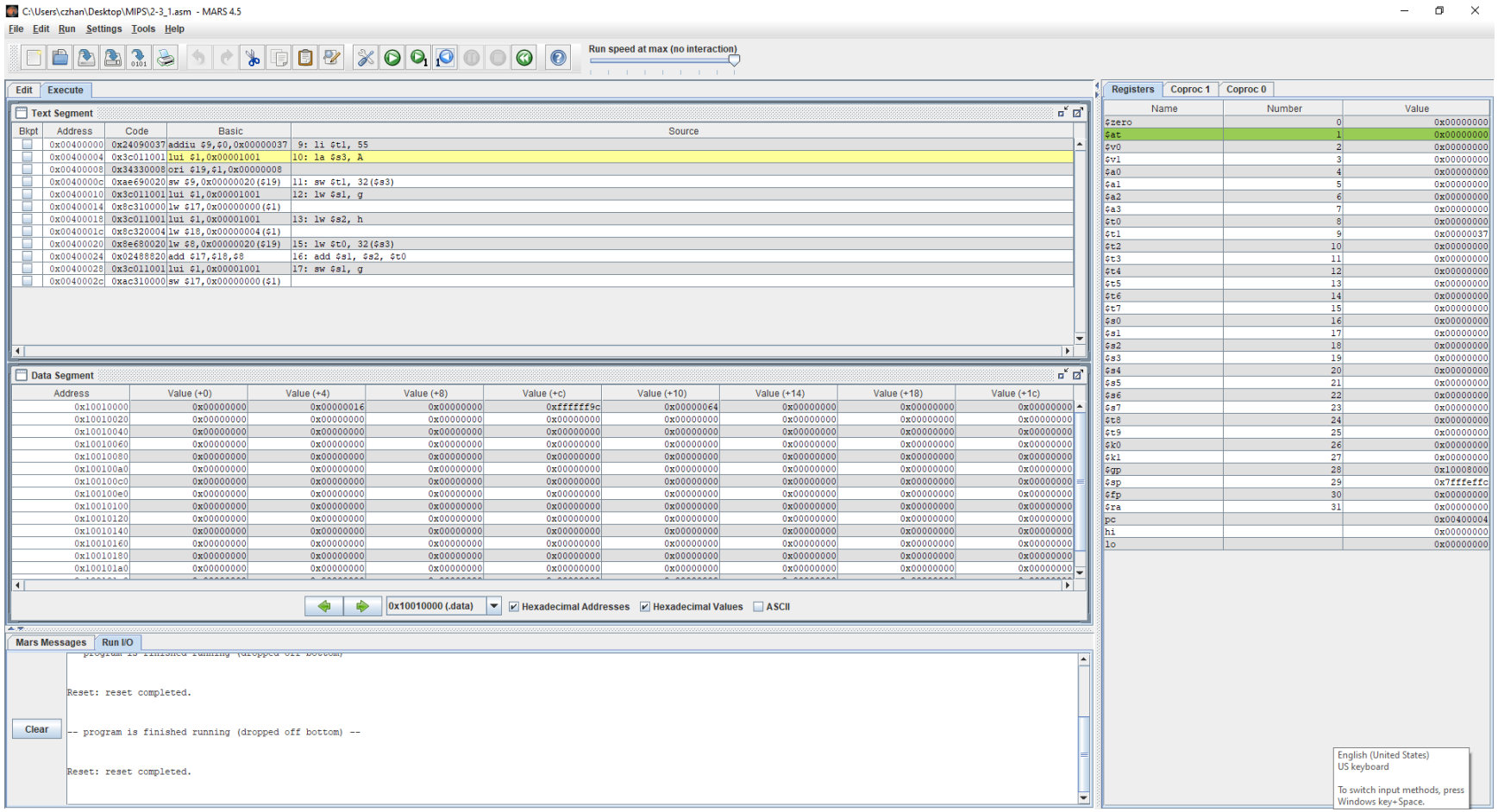
In the screenshot above, all integers have been assigned to their addresses and as we can see above, we have the values of f,g,h,I,j at 00,32,28,1e,14 respectively. Furthermore, I have also printed out the base pointer, stack pointer and the instruction pointer. The instruction pointer is showing which instruction it will run next which is f = (g + h) – (I + j) and we can notice that ebp and esp are the same and they should stay the same throughout the programs run time as all variables are static, none local



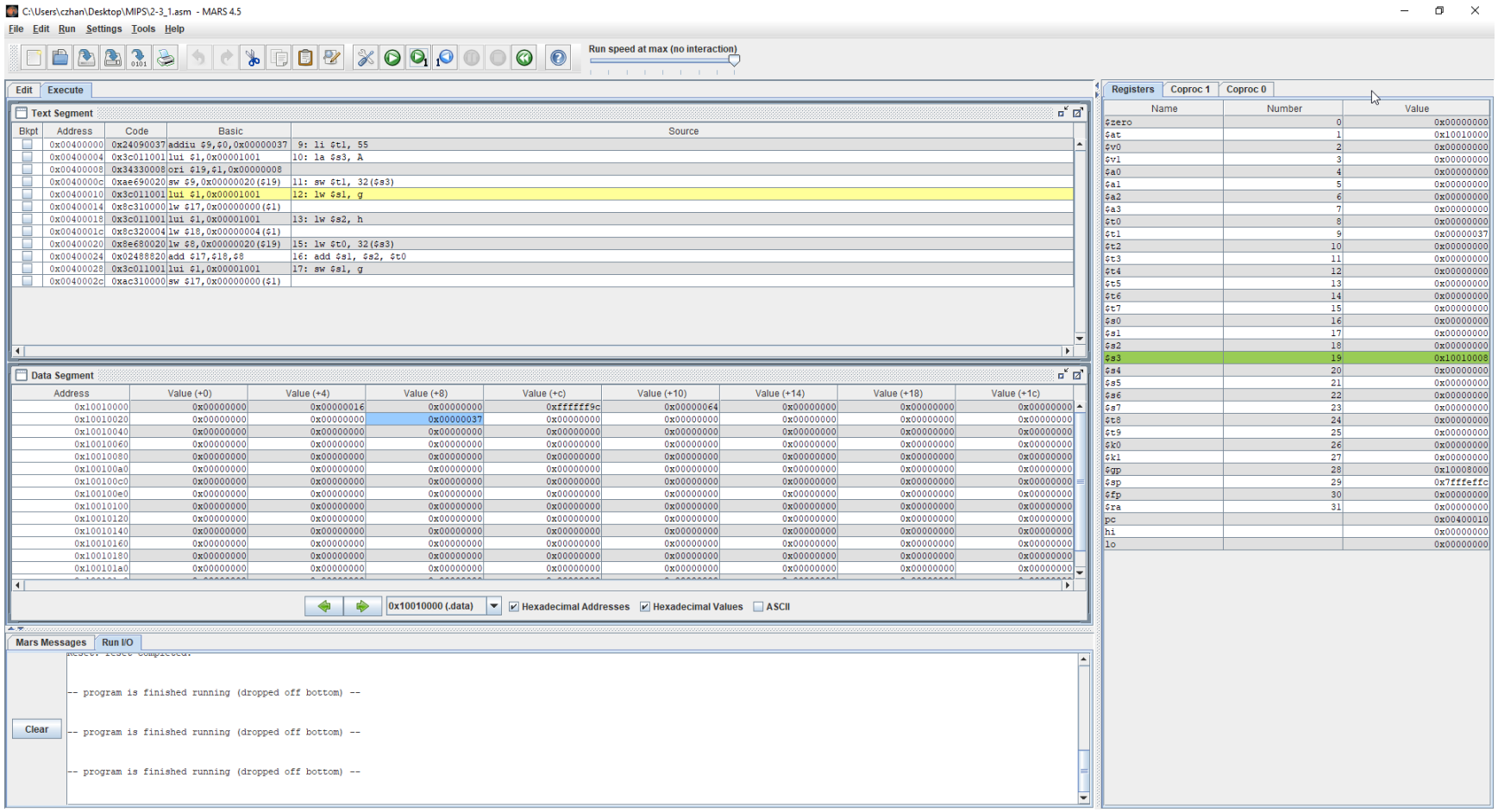
In the screenshot above, what is showcased is the function f = (g + h) – (I + j) being completed as the value of f has now been replaced with the hex value of 28. f = (50 + 40) – (30 + 20) = 40 = 28 which is correct. Therefore, we can confirm that the program has run successfully and we also notice that $rbp and $rsp has not changed due to the fact that there are no local variables and only static so it should not change.



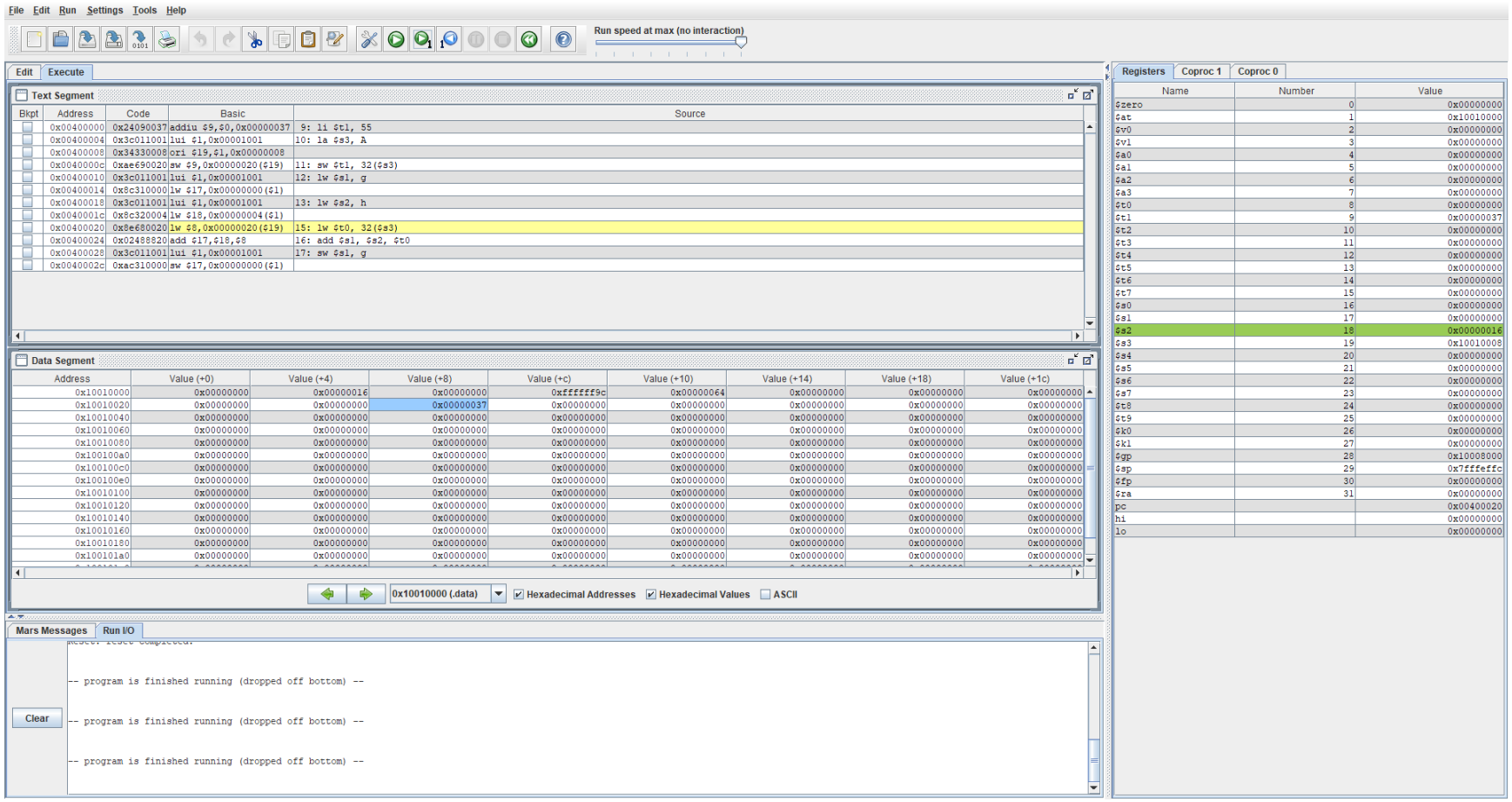
In the figure above, what is showcased is the instruction to perform a[8] + h and in this bit of code, there are two static variables g and h and you will notice once again that $sp, the stack pointer will never change in values as we are only working with static variables as the array is also static.



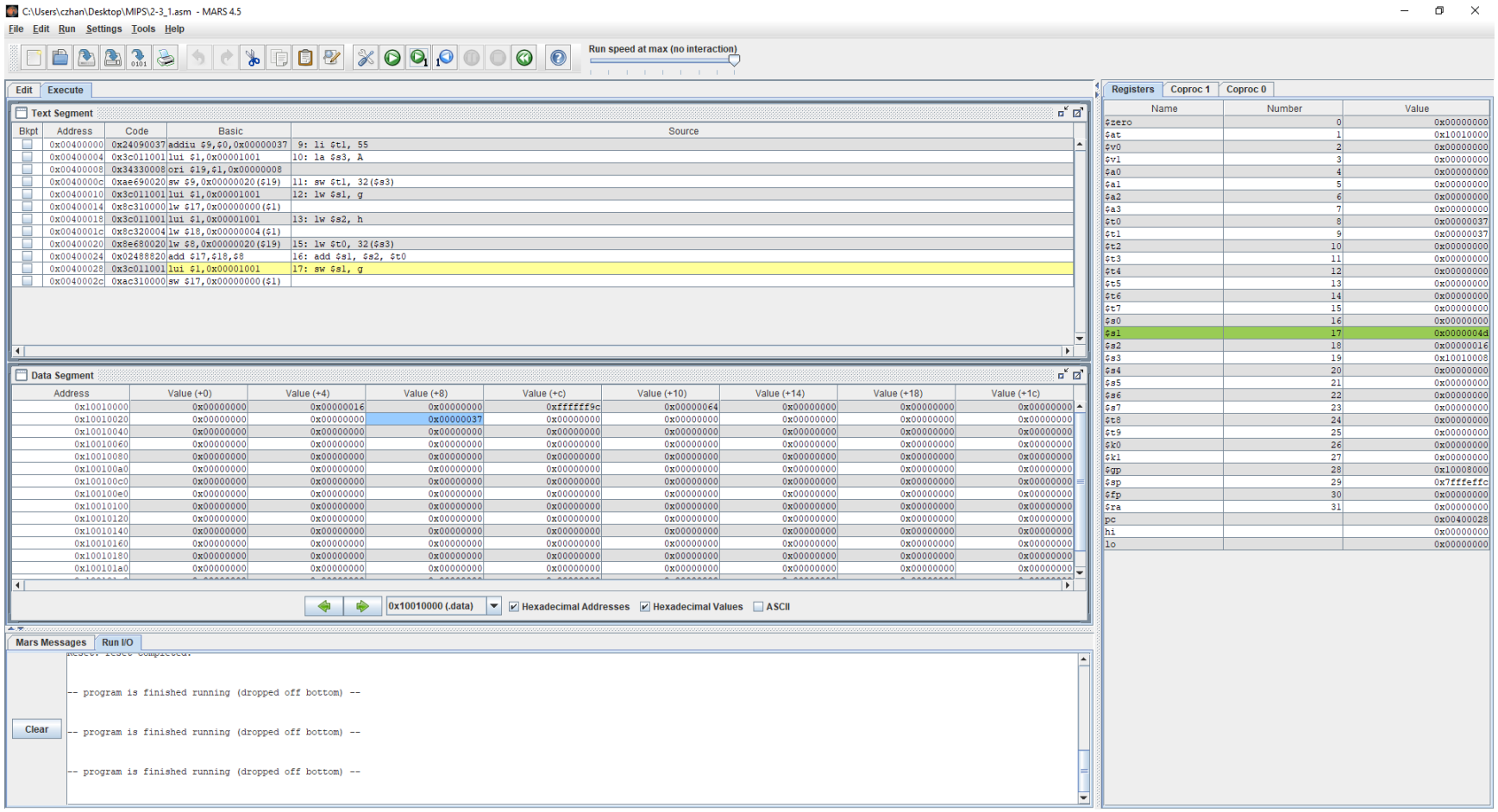
In the figure above, what just happened is $t1 now has a value written into it which is 0x0000037 which in decimal form is 55. If you look at the text segment, you will notice that the instruction that was run was li $t1, 55 which means that 55 was loaded into t1.



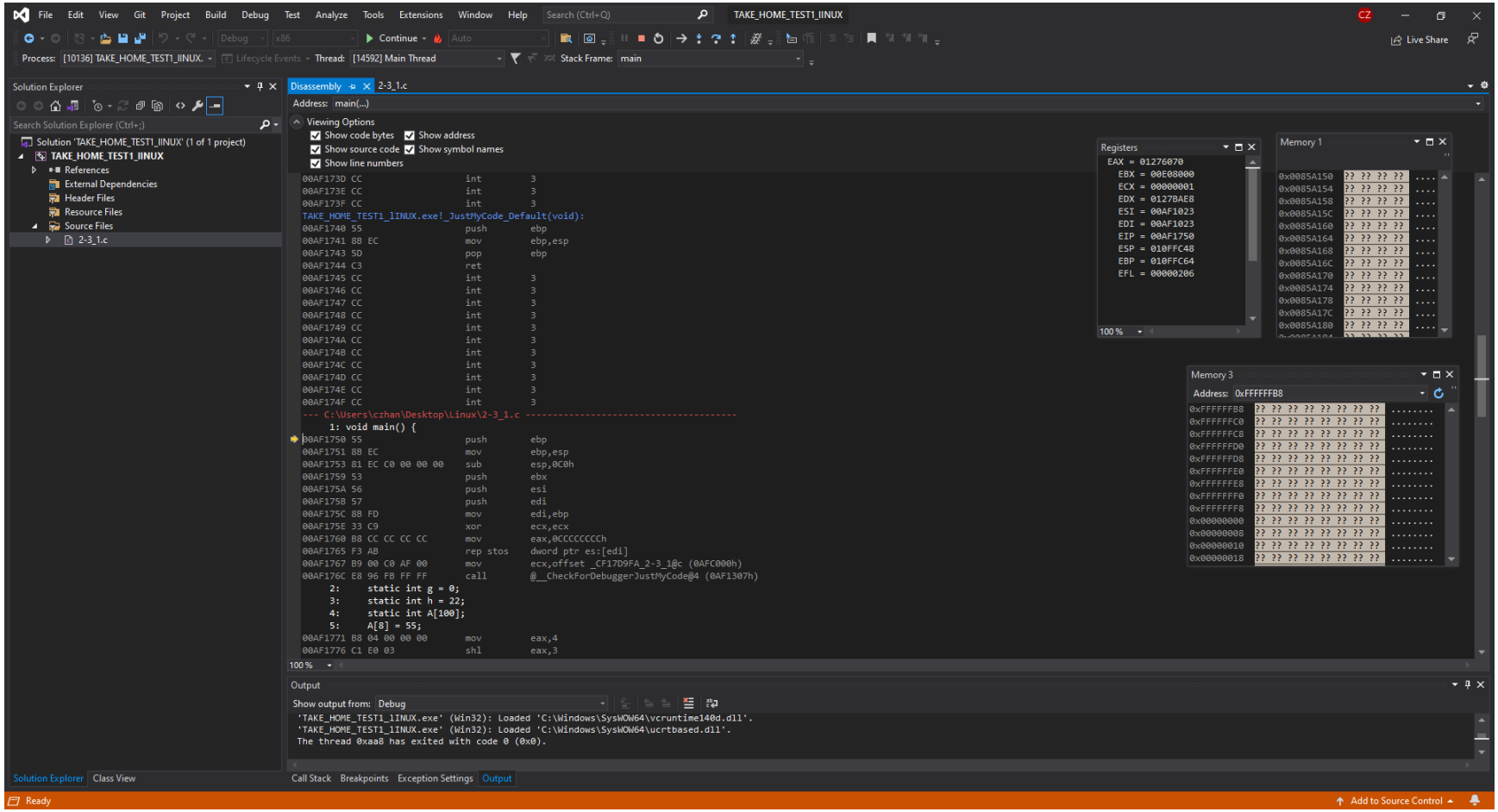
In the figure above, what has happened is ori was performed with 0008 and $at to find the first element of the array A which we can see is in $s3 with the value 10010008. Then it is loaded onto the Data segment as we can see in the Data segment window 0x10010020 with offset + 8



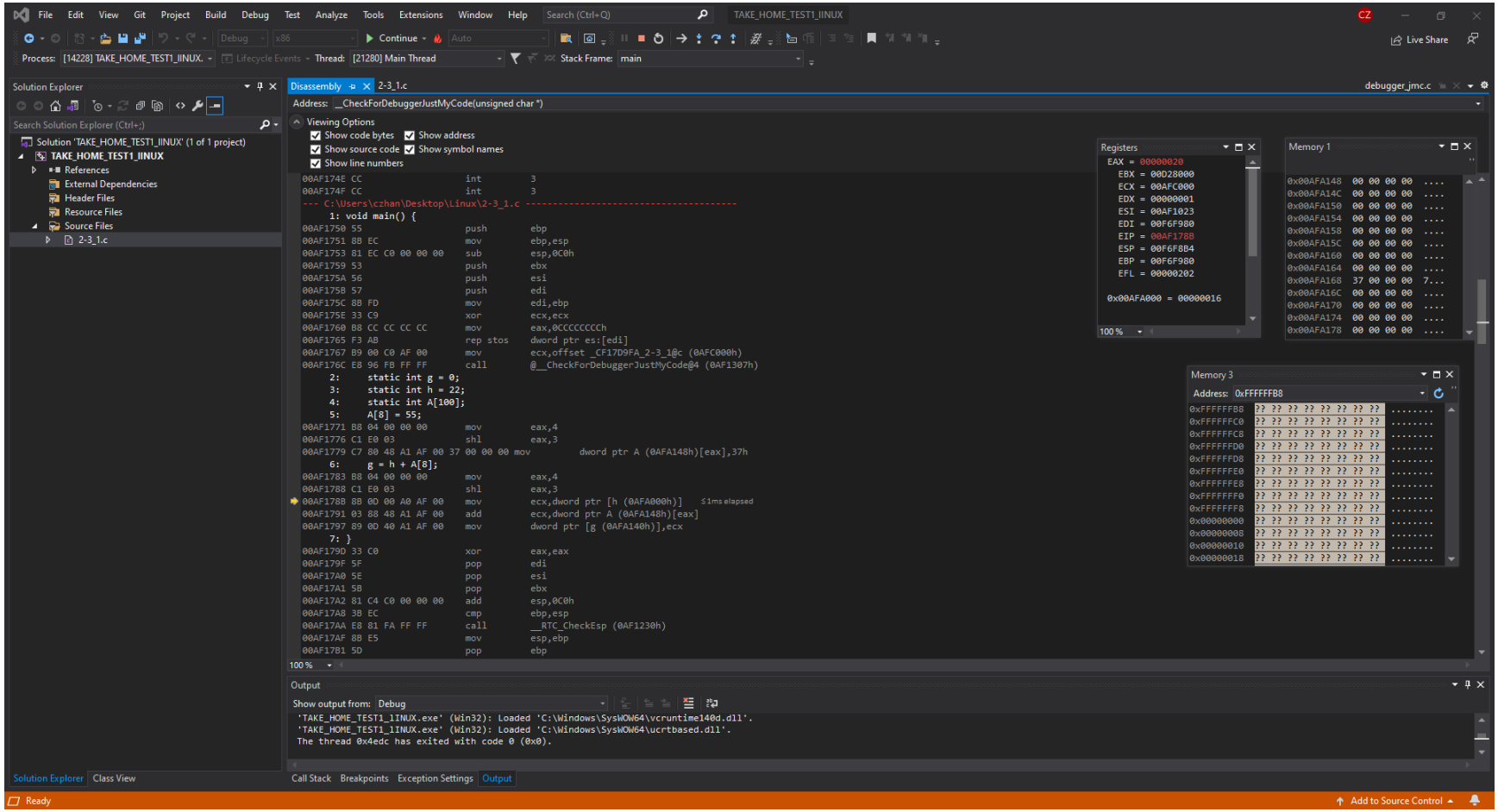
In the figure above, what is shown now is variable h being written into $s2 with the value 0x00000016 which when converted to decimal is 22 which is indeed the value of h in the assembly code. We can also see that in the text Segment that g was also written into $s1 but it has a value of 0.



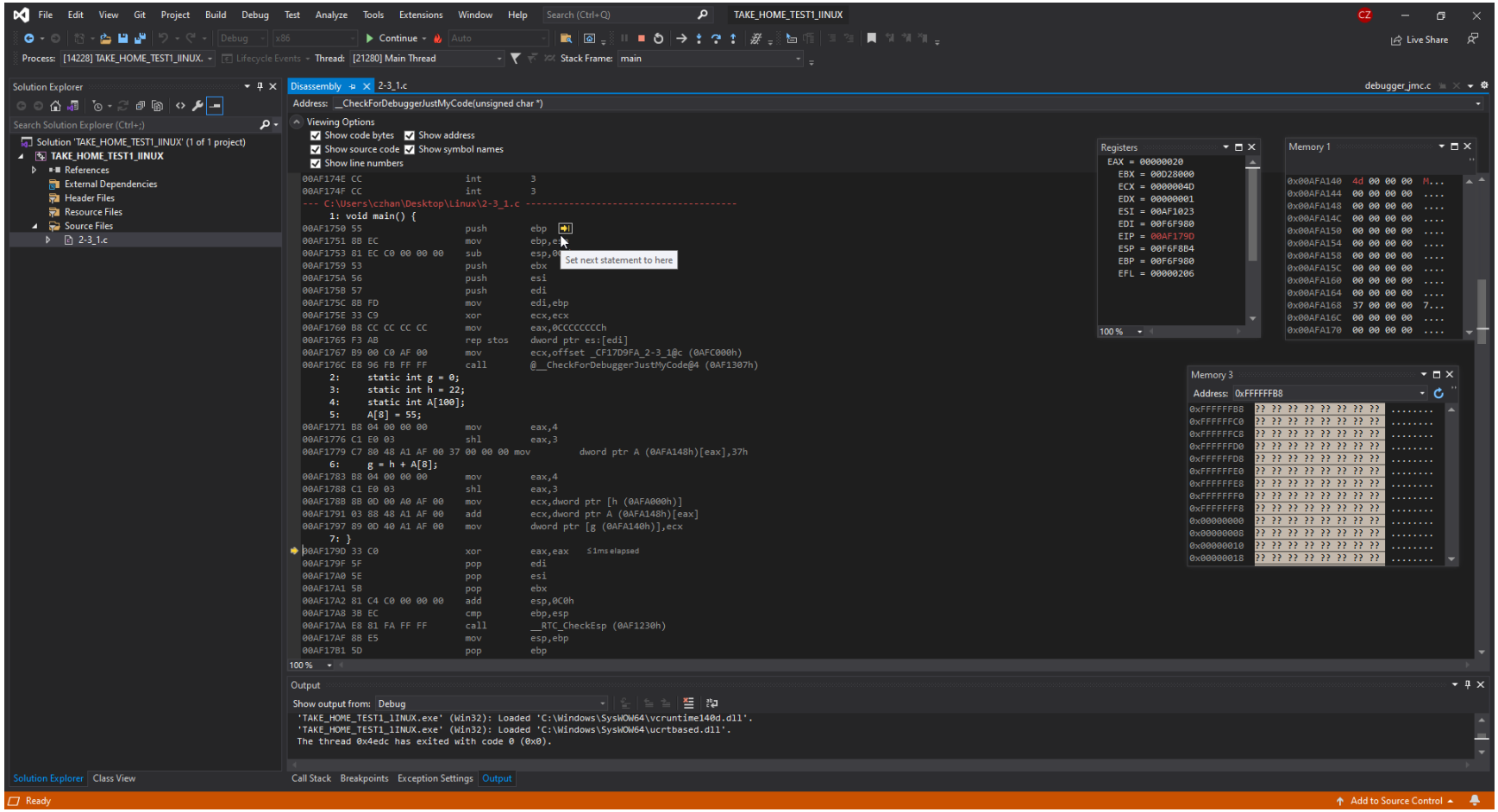
Lastly, what is shown in the figure above is the operation of A[8] + h which is 55 + 22 = 77 which in decimal is 4d. If we look over at the register window, we will notice that $s1 is now written with the value 4d, which indicates correctness in the code. You can also look over at the text segment and see the assembly code written as add $s1, $s2, $t0 which means write into $s1, $s2 + $t0 which as we can see, was performed.



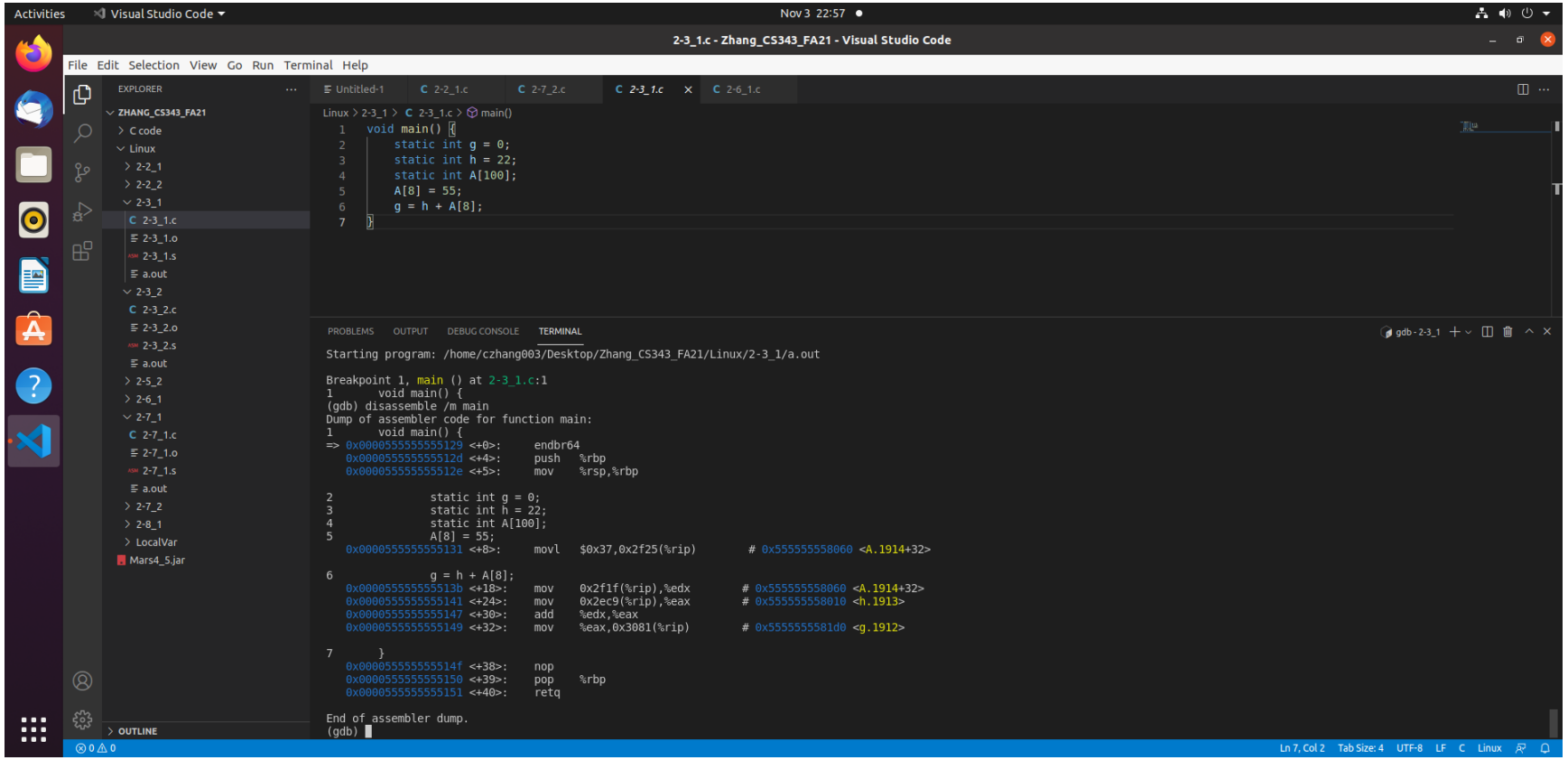
What is showcased in the figure above is the initial debug window which is included with the disassembly tab, the register windows and the Memory windows. EIP is pointing towards the next instruction 00AF1788 and will continue pointing towards the next instruction throughout the programs run time.



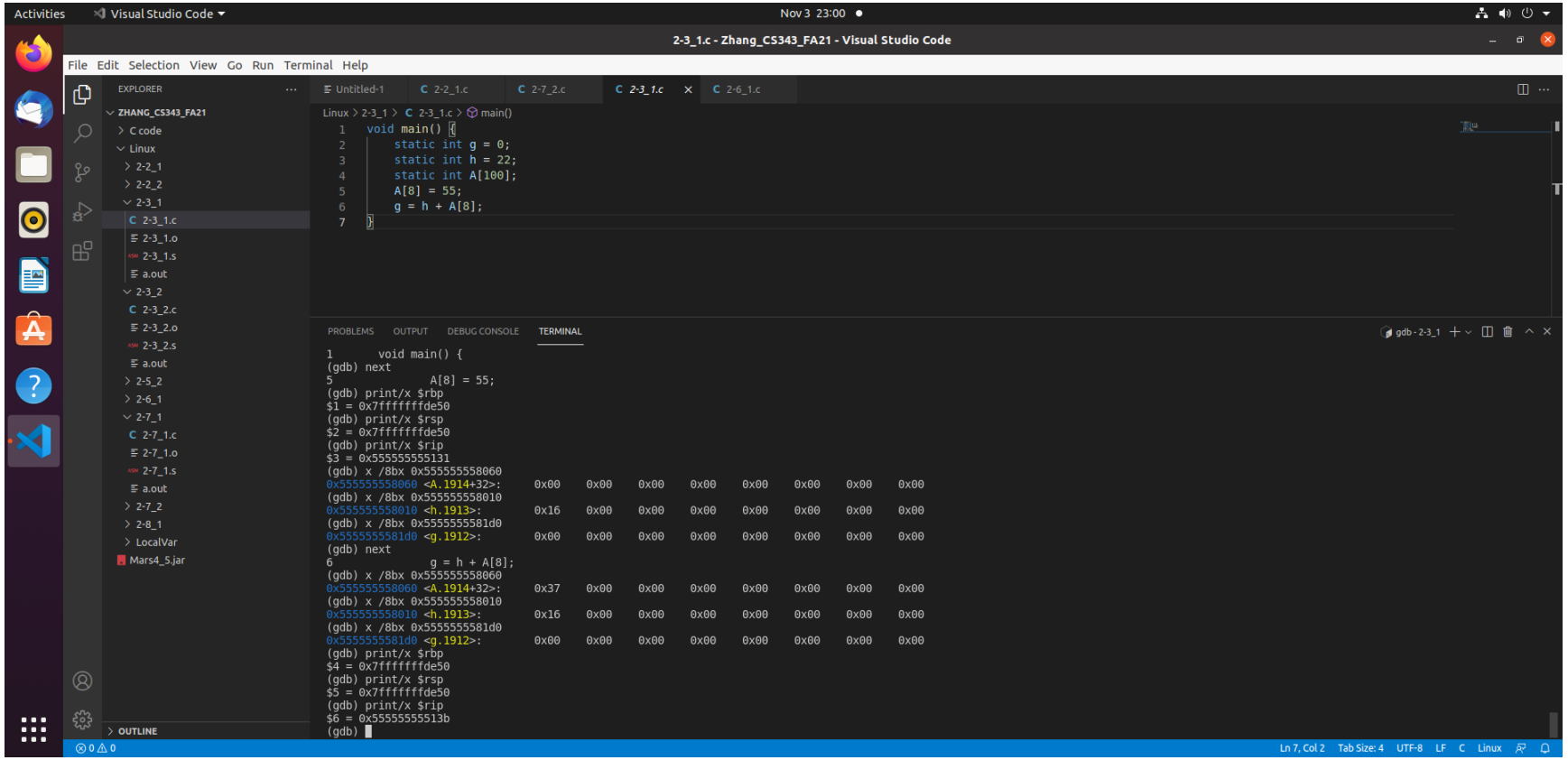
In the figure above, what is now showcased is h's value being allocated to a address as we can see in the register window that 0x00AFA000 now has the value 0x00000016 which is 22 in decimal form. Next if we look at the memory 1 window, we can notice that 00AFA168 has the value of 37 now when converted to decimal is 55 which is should be.



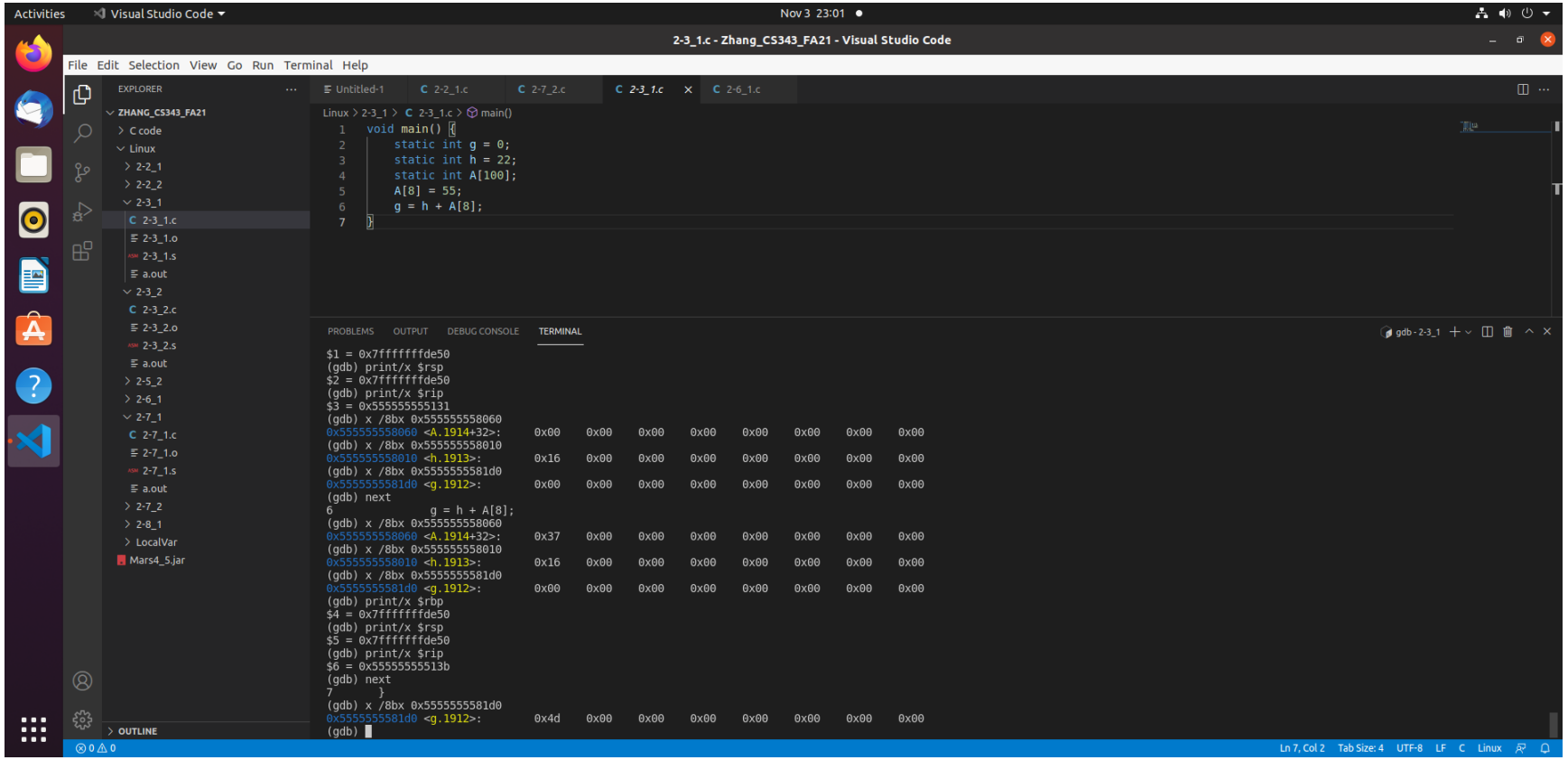
Lastly, in the figure above, what can be observed first is where the disassembly windows arrow is pointing at and it is shown that the arrow is outside of the written code so we know that instructions for that bit of code is done. Next, please look at ECX which now has the value of 4D which is the sum of a[8] and h. You can also see this in the Memory window 1 where the address 00AFA140 has the value of 4d which when translated over to decimal, is 77.



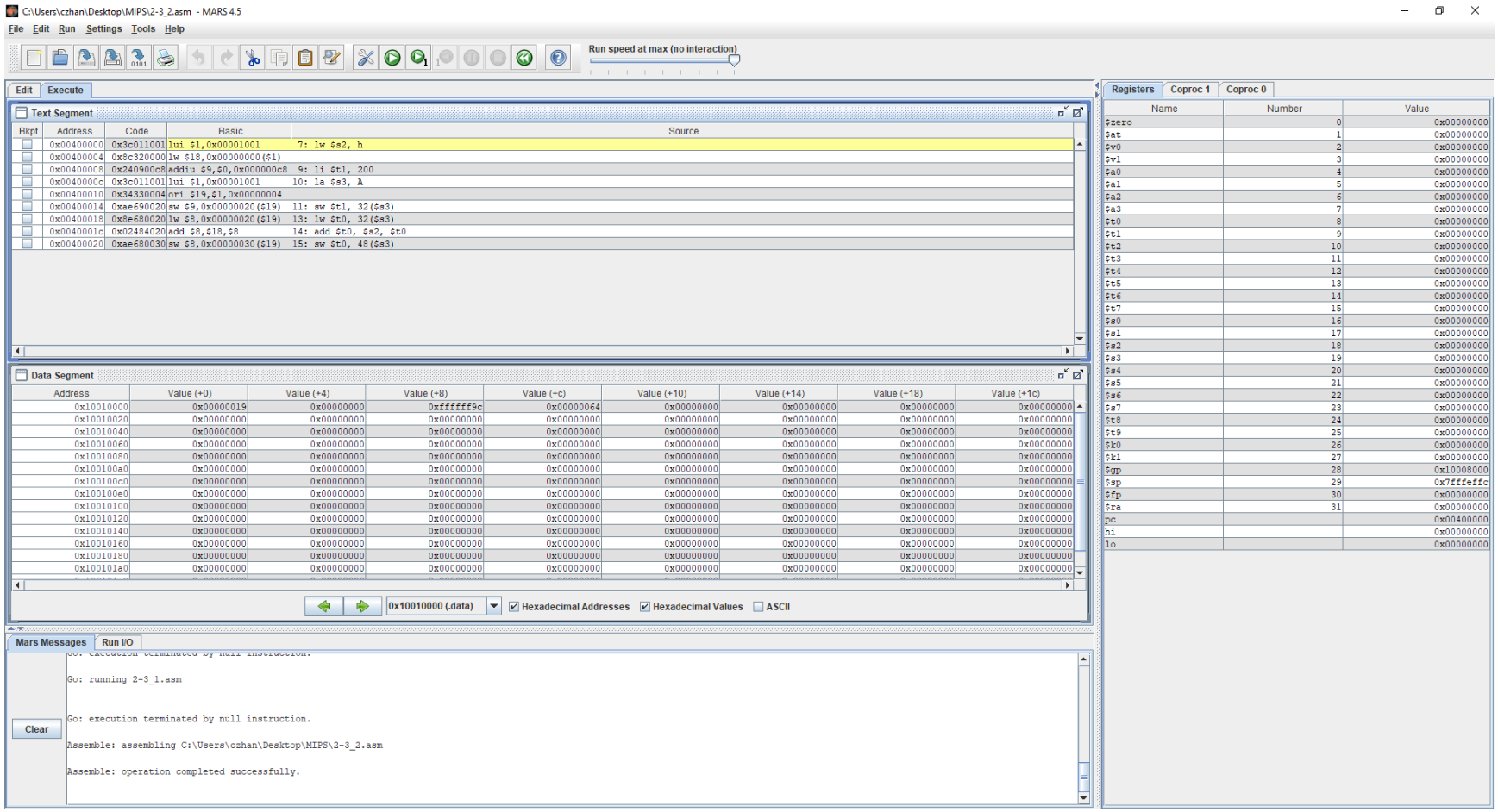
In the figure above is the program being run in the LINUX environment using GCC and a windows compiler. What is showcased in the debugger is the address of A,H and g which is extremely important for understanding what is going to be shown as values will be written into those variables.



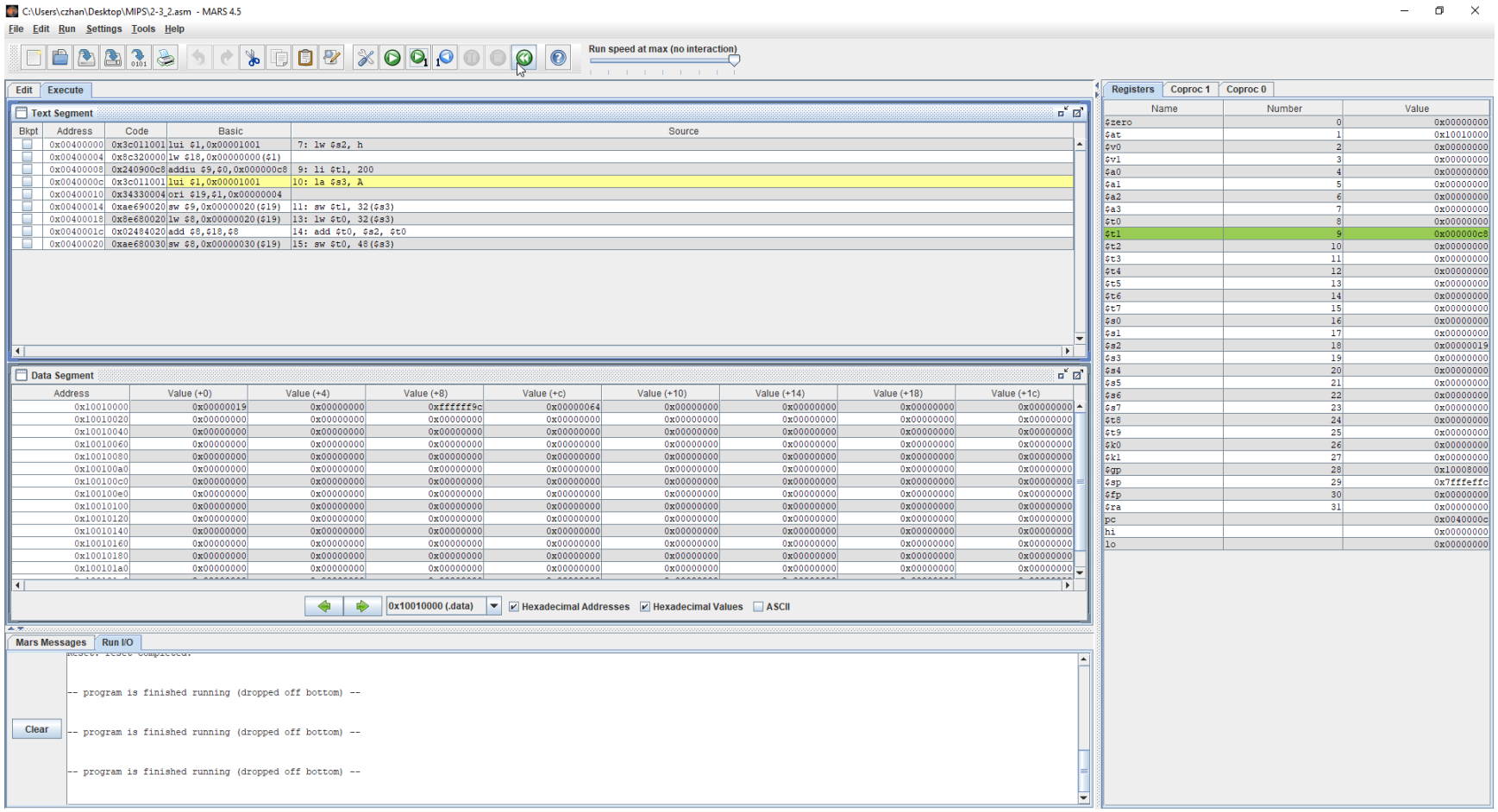
In this figure above, what is showcased is the allocation of value into the addresses and as you can see, in instruction 5 (to confirm that it is indeed instruction 5, look at the top where it says 5 …. A[8] = 55;. We can see that A has nothing written into it yet, and h has the hex value of 16 at the moment with g also having 0 which is correct. A[8] currently has no value written into it yet as but after running the instructions in 5, we can see that A now has a value of 37 in hex. What this shows is that a[8] now has a value written into it on offset +32.



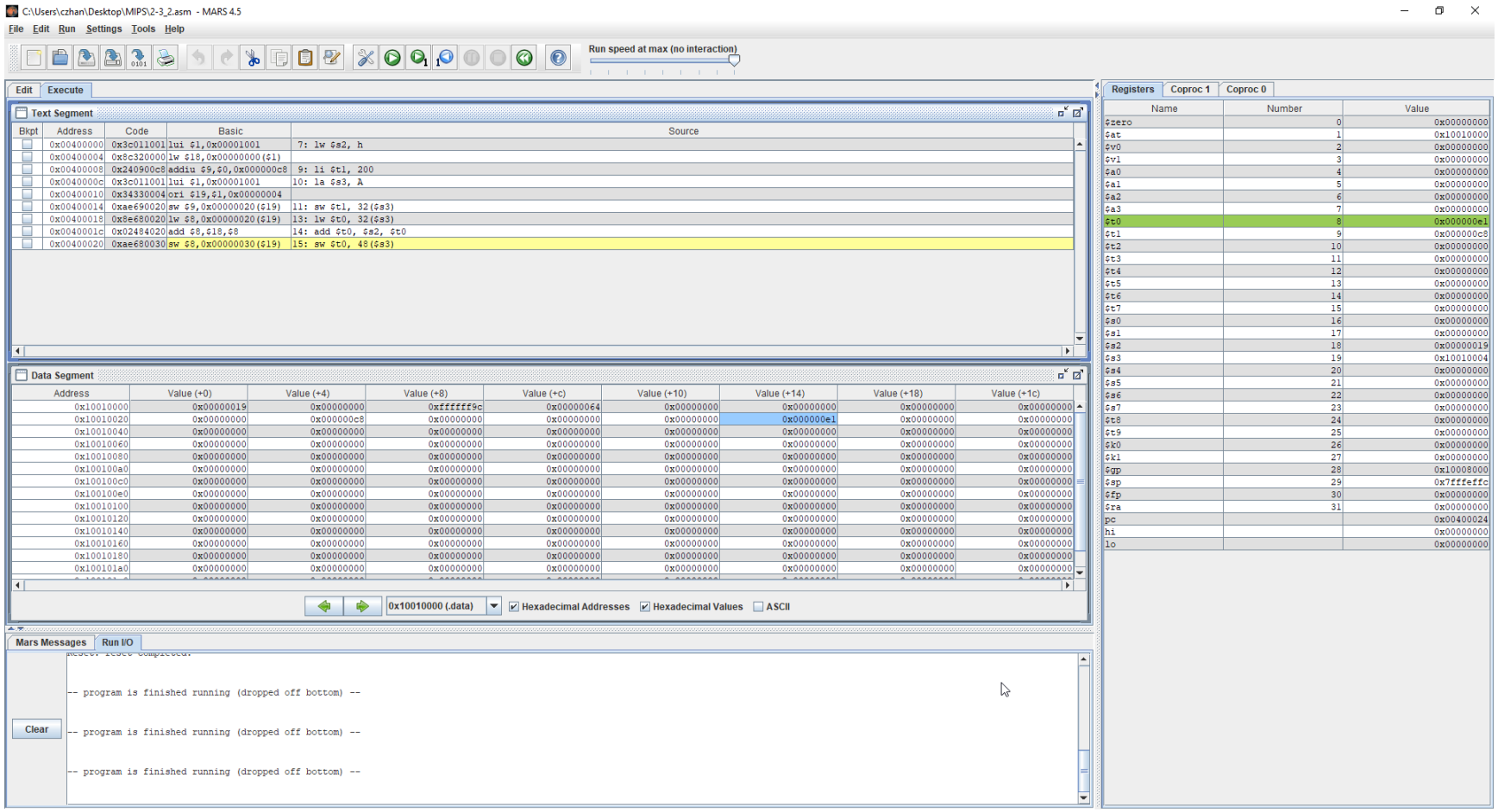
In the figure above, we have gone to the next step after allocating value 37 into 9th index of Array A which is to add the 9th index of array 8 with h which gives us 0x4d. 0x4d when converted to decimal is 77 (55 + 22 = 77) and what is printed in the bottom of the screenshot is indeed 77 so correctness has been confirmed.



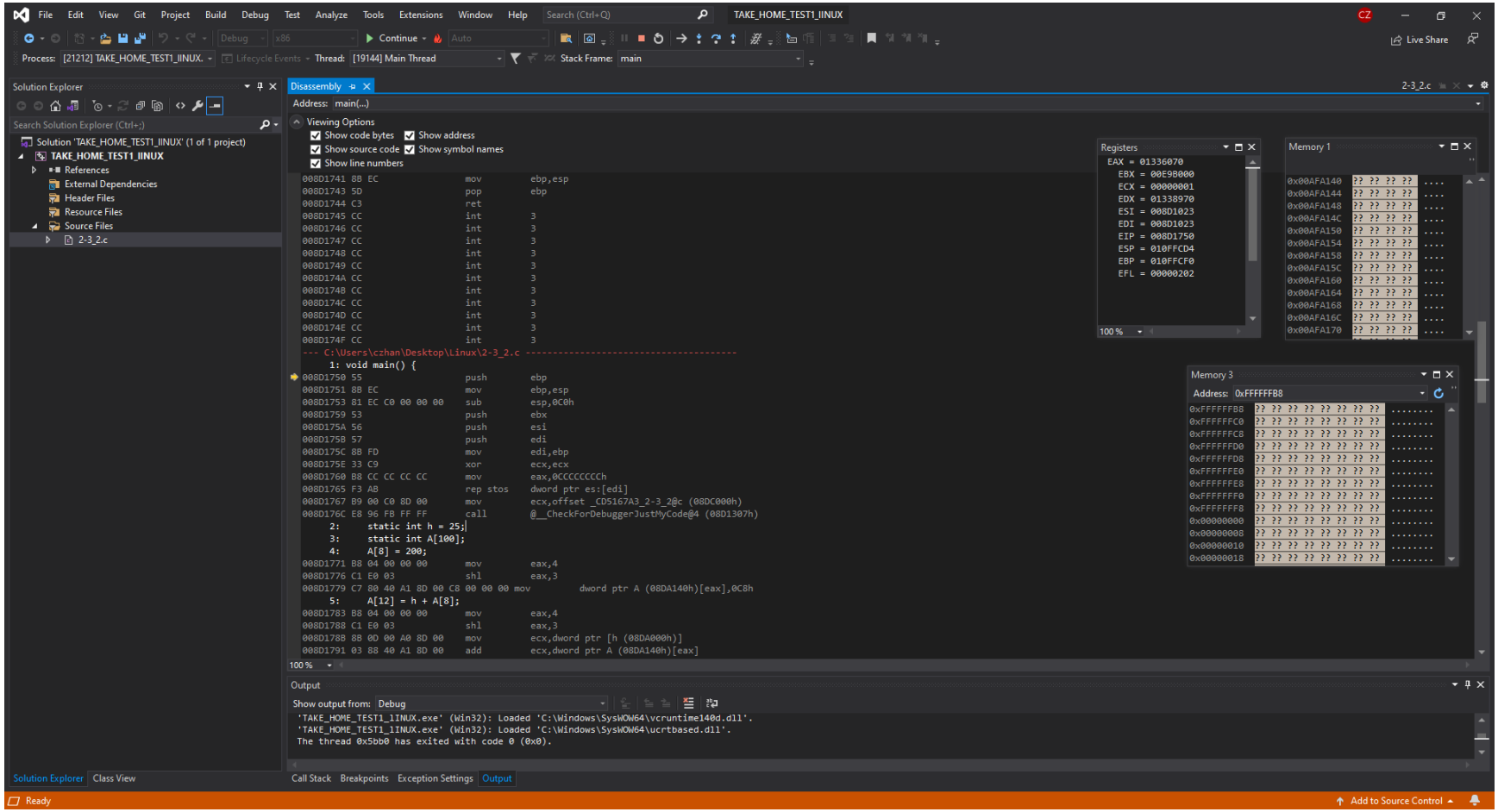
In the figure above, it showcases the MIPS instruction for the code in question 2-3_2 and what is provided to us is a Text segment, Data Segment and Register window. If we look at $PC we can tell what the next instruction to be done will be. In my $pc, it shows that 0x00400000 will be the next instruction and we h and the array A are static therefore we will not see $sp changing in value since there are no local variables.



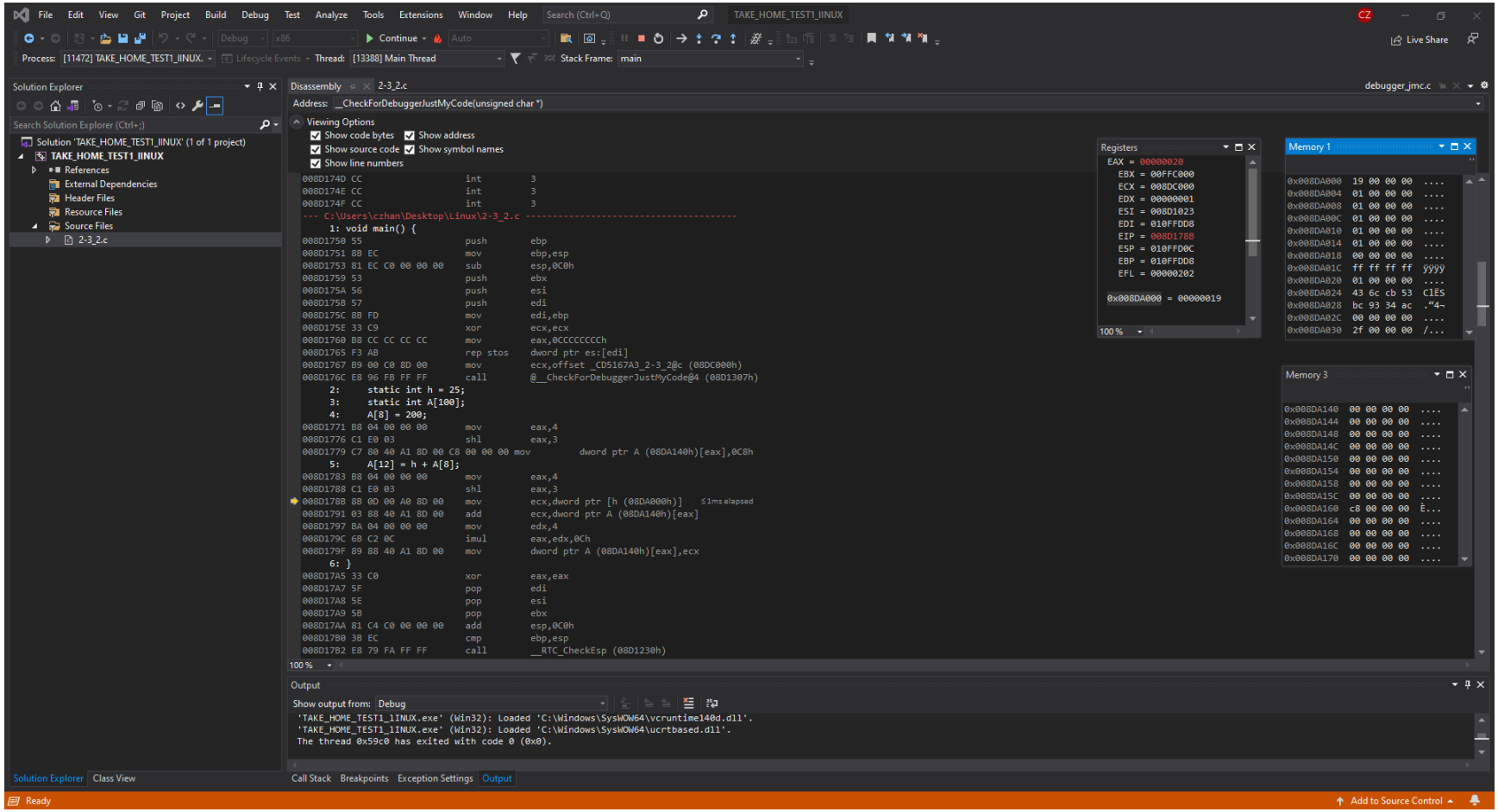
In the figure above, what we can observe is the address of h is being loaded into $s2 as it is the instruction listed on the address 0x0040000. The instruction is loadword of h into $s2 which we can see did happen if we look over at the register window at $s2. Furthermore, we also can notice that $t1 has been loaded 200 as 200 in hexadecimal c8 which is true as we can pin point where it is in the register window.



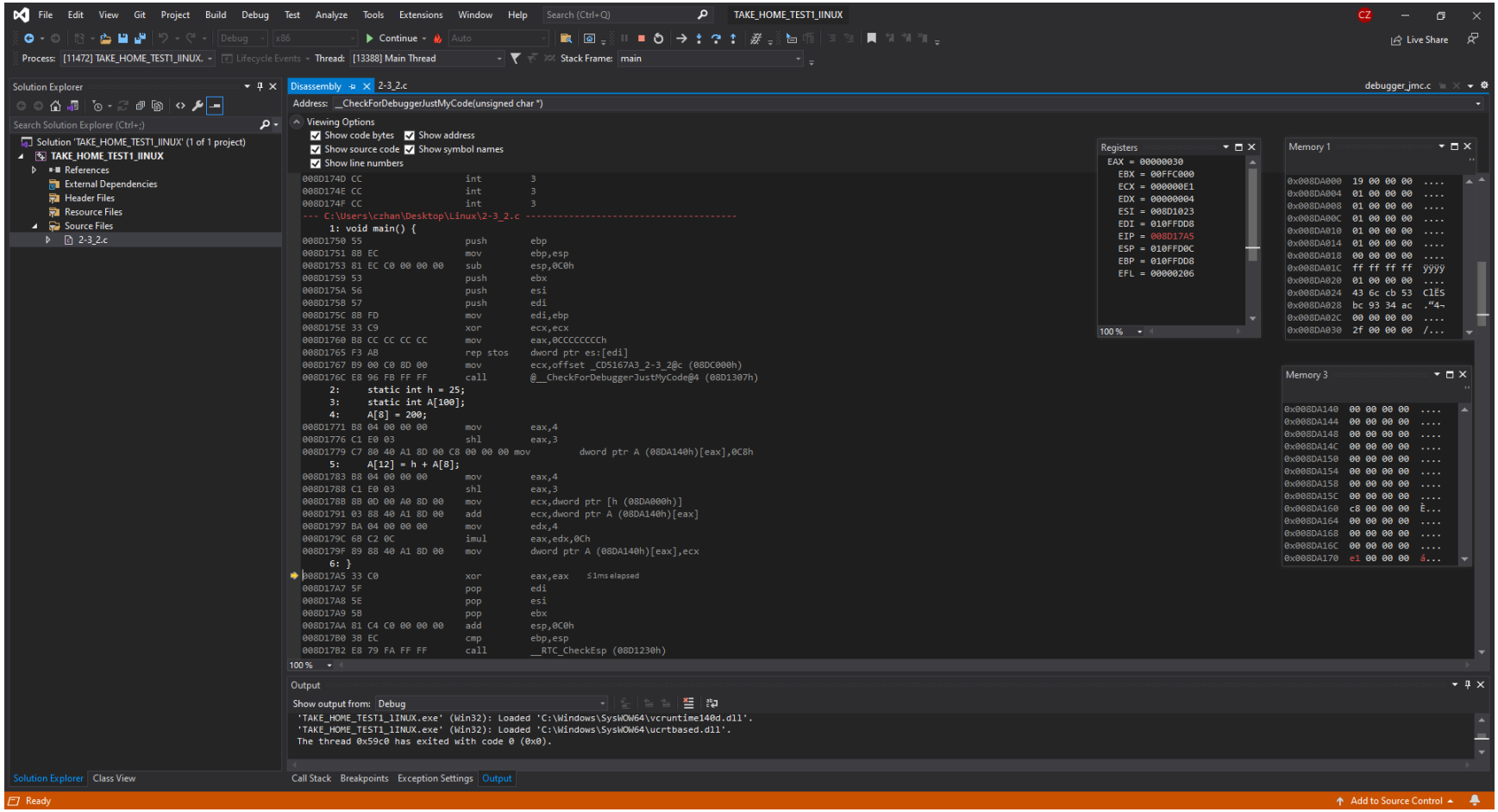
In the final figure for the MIPS instruction above, we can see all the instructions that were done. First we can notice that the word in $t0 was saved into then loaded onto $t0. Please look at the instructions at 0x00400014 and 0x00400018. What is done next is $s2 and $t0 were added into together then saved into the register $t0 and as we can see, there is a value that is not 0 inside there. The math that occurred there was 200 + 25 where 200 is A[8] and 25 is h which gives us 225 and in hex, the value is 1e. We can confirm this by looking at the register window and looking for $t0. Finally, we need to confirm that the value was stored in the 12th index of the array and we can do that by looking at the text segment and the data segment. Please observe that on +48 offset, the value 1e was stored within and the instruction for storage is in the text segment where it performs the operation saveword into $t0 with +48 offset of $s3 which is the 12th index of the array



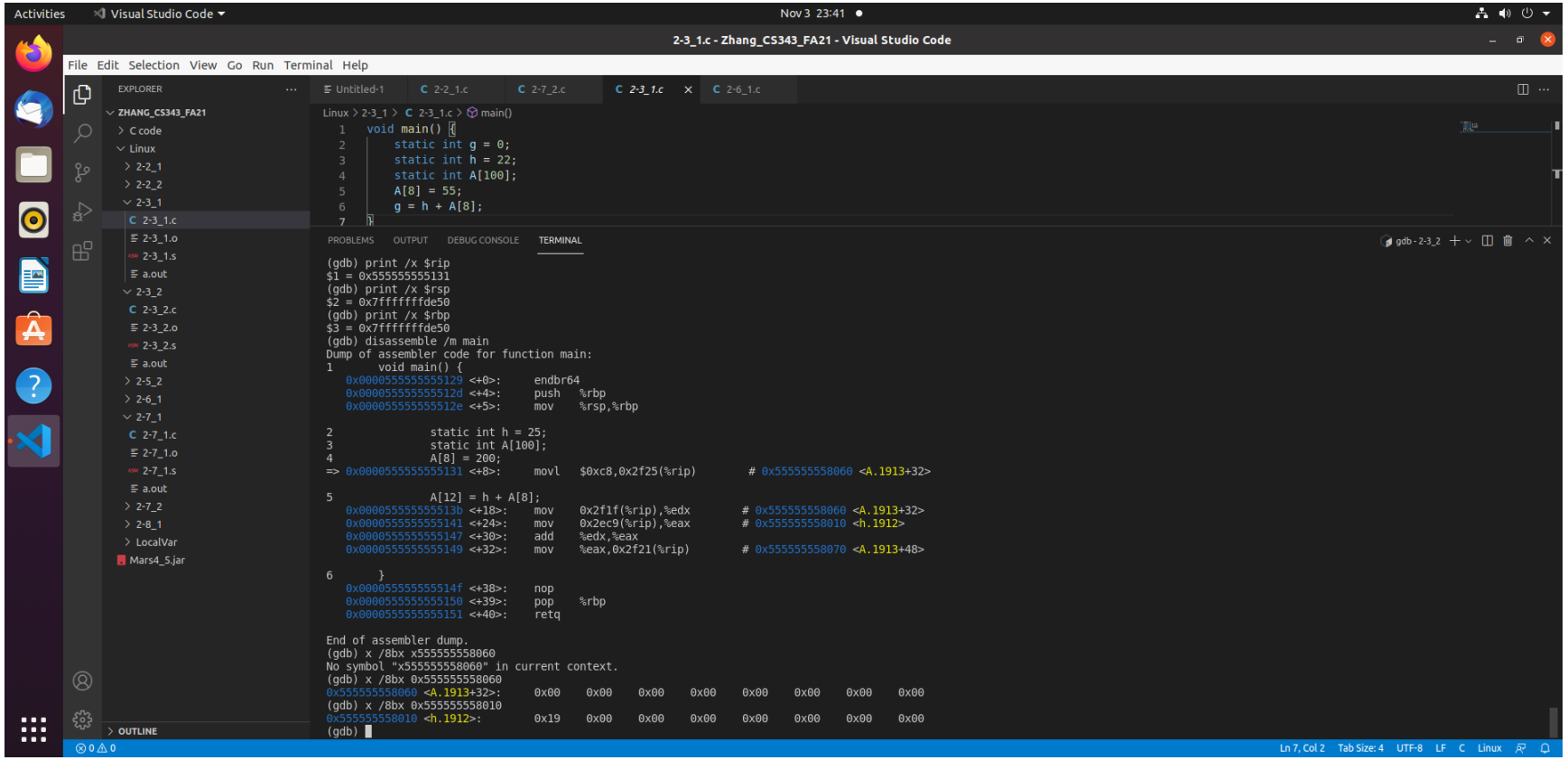
In the figure above, what is showcased is the debugger for question 2-3_2 using an intelx86 processor where what we can observe is the register windows and the two memory windows.



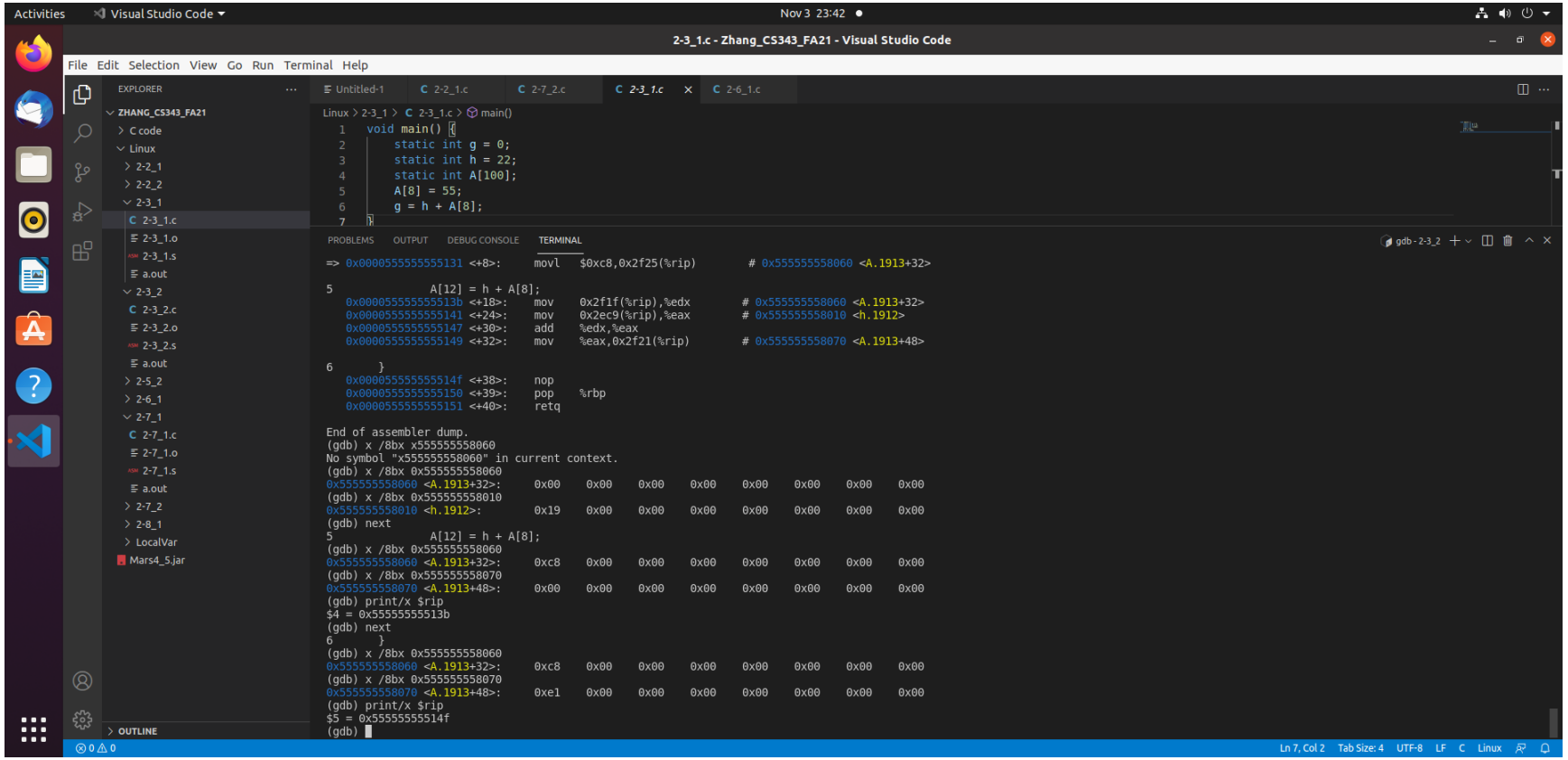
In this figure above, what is showcased is the static int h, the static array A and that 8th index value of the array A has the stored value of 200. If we look over at Memory window 1, what we can see is that the address 0x008DA800 now has the value of 19 and in decimal that value is 25. Next, look at Memory window 3 where at the address 0x008DA140 has the value 0. That is the 1st element of the Array A and if we look at 0x008DA160, we can notice that there is the value c8 written into it, or 200 in decimal form. The offset is +32 and we can see that clearly in memory window 3 as we can count down 8 to see the allocated memory in the 8th index value of the array A



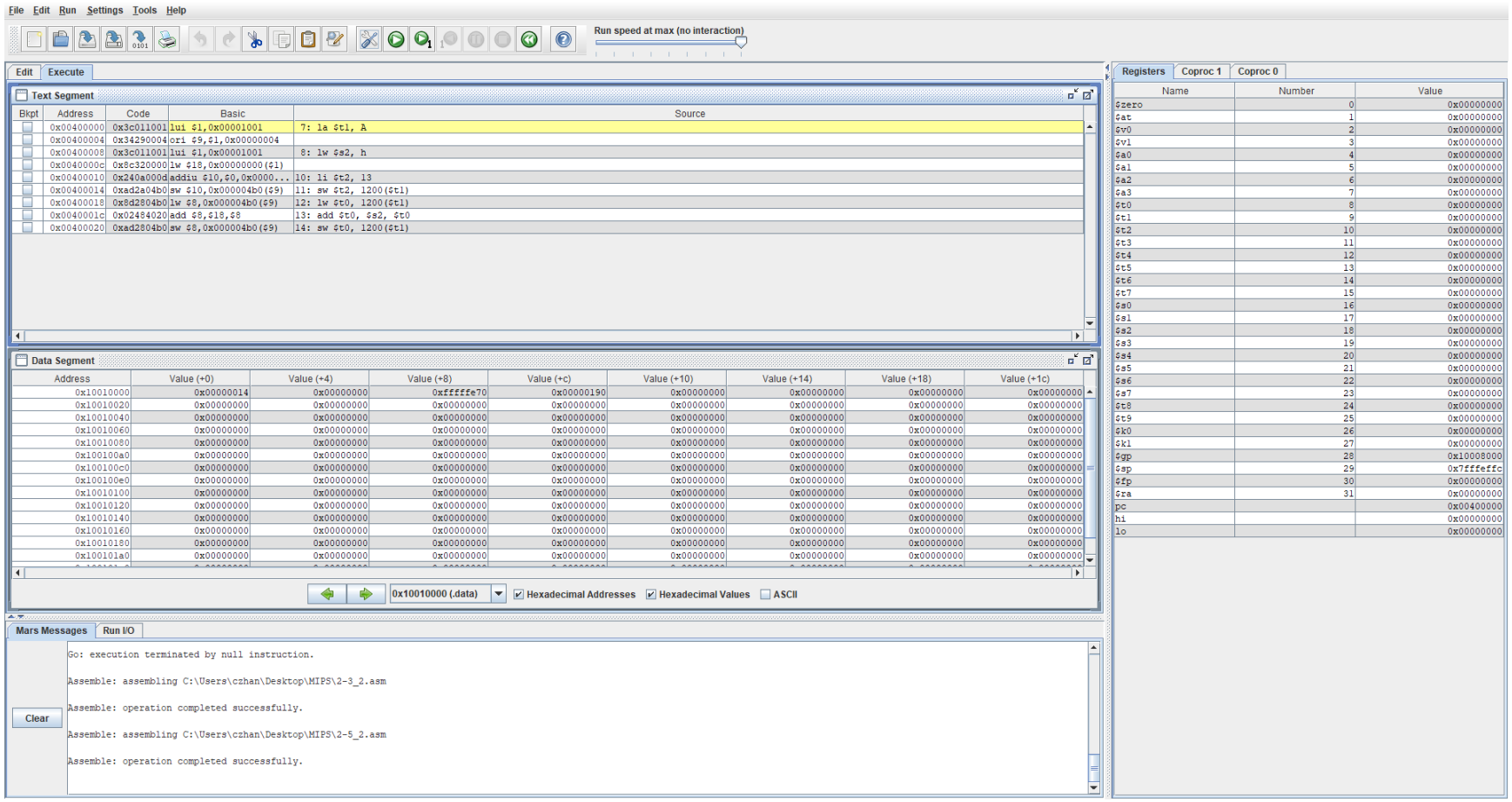
In the figure above, what is shown is the code being run completely through and that this programmed has run successfully. If we look over at memory window 3, we can see that at the address 0x008DA170, the value e1 is written into it and that is because that is the 12th index of the Array A or Array A with offset +48. 200 + 25 = 225 which when converted into decimal, returns 1e.



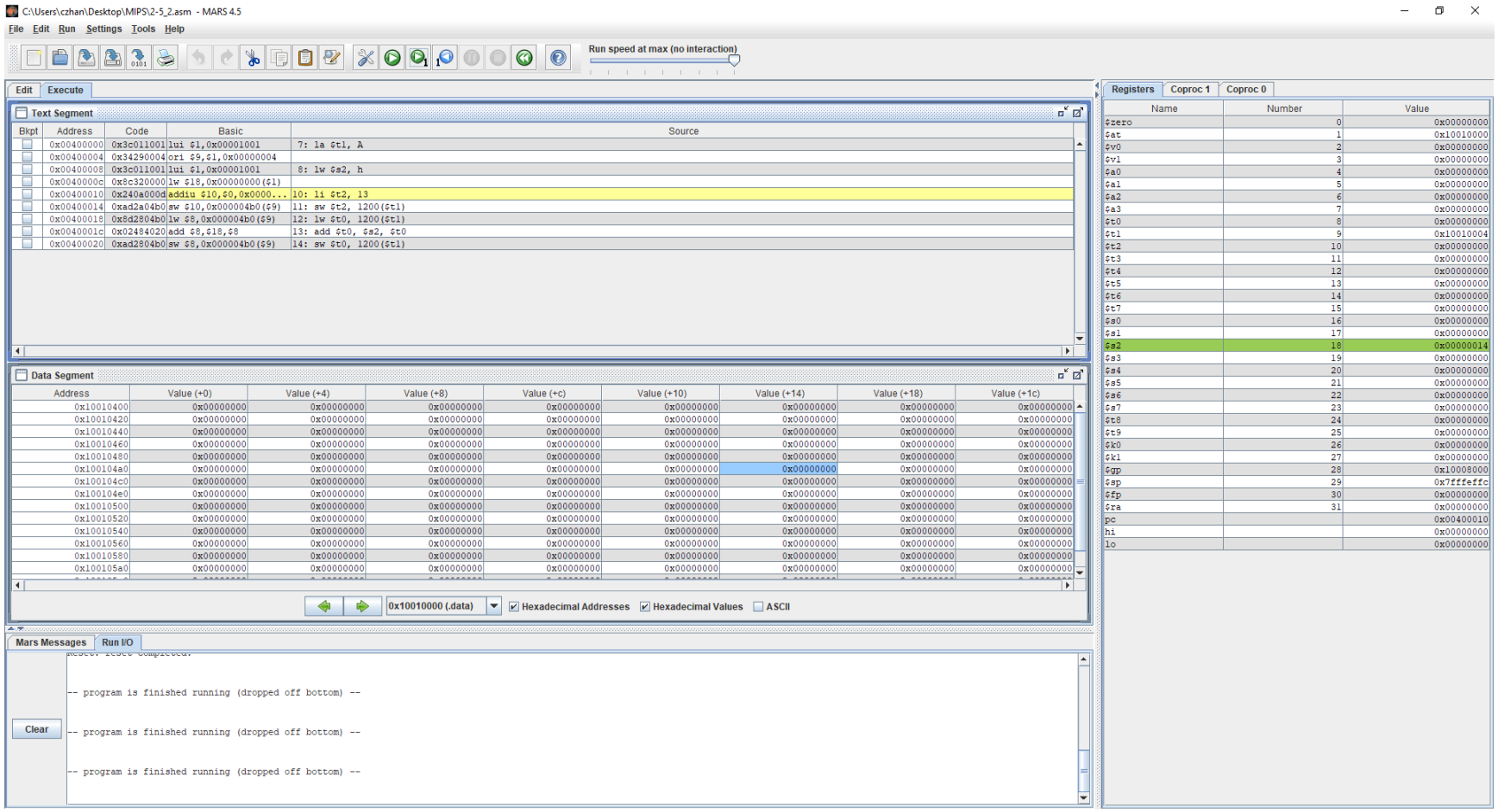
In the figure above, what is shown is the code being compiled in a linux environment using a windows 10 compiler. In the Terminal, what is displayed is the dump file of the code which shows all the instructions that will be done, as well as the addresses of the array A and h, which we will be using to confirm that values are being processed properly and being stored properly. If you look at the bottom of the screenshot, what is printed is the value of A and value of h. h has the value of 0x19 or 25 stored and A currently has nothing stored in it. You can also look at the printed values of $rip to see the next instructions that will be done, then looking into the dump file to confirm if $rip is correct.



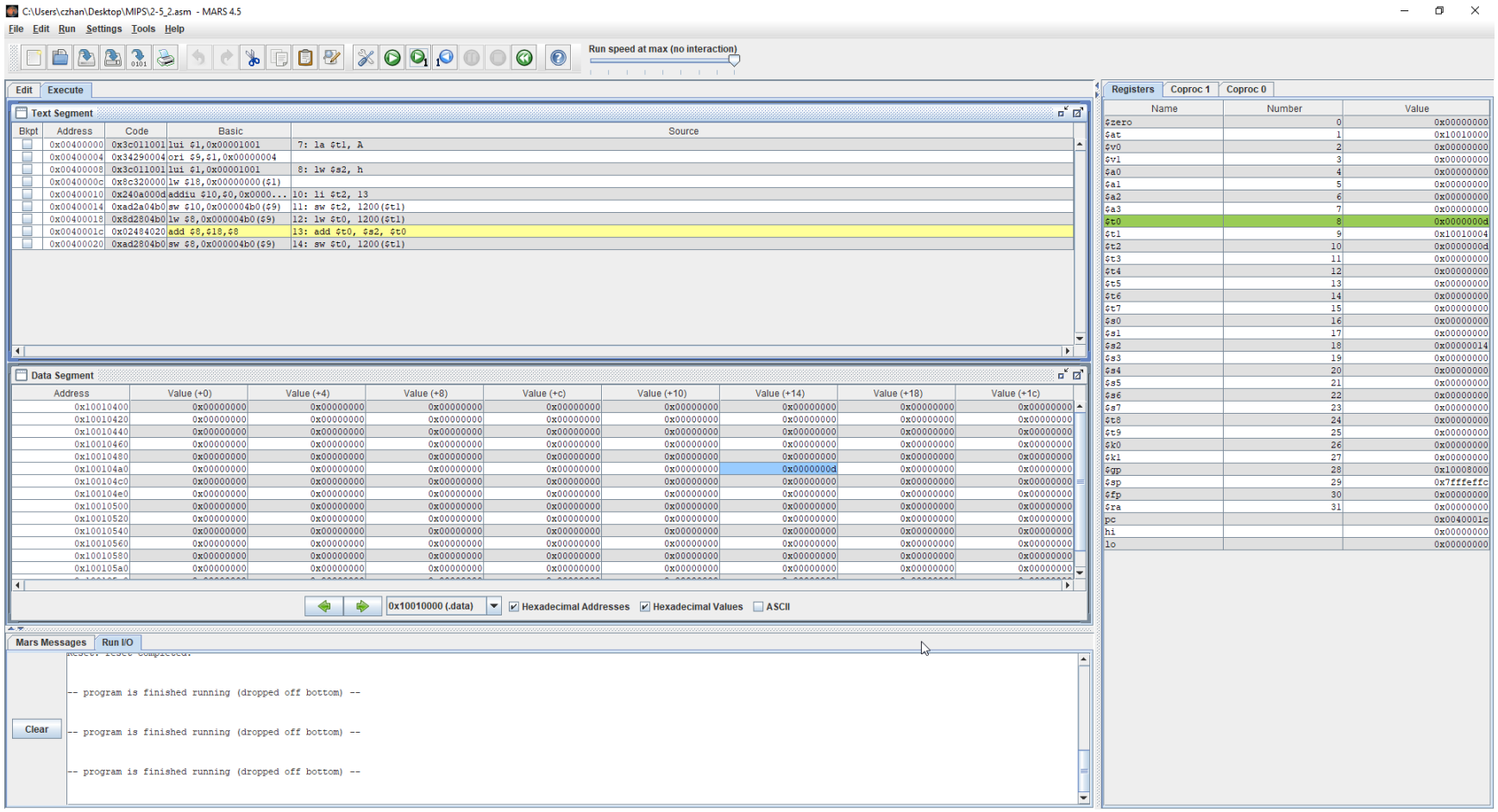
In this figure above, what is showcased is the progression of the code through the steps A[8] = 200 and A[12] = h + A[8]. At step 5, we can notice that A[8] now has the value of 200 stored inside it as c8 is 200 in hex form. Furthermore, what I printed to check correctness is A[12] as well which has nothing written in it, which is correct. Next, we go to the next step where A[12] = A[8] + h which is 225 = 200 + 25 which should output e1. If we look at the bottom of the screenshot, we now see that A[12] now has a value of 0xe1 written into it which shows that A[12] = A[8] + h was performed successfully with no errors. Please also notice the $rip instructions and that each instruction was properly executed according to the dump file.



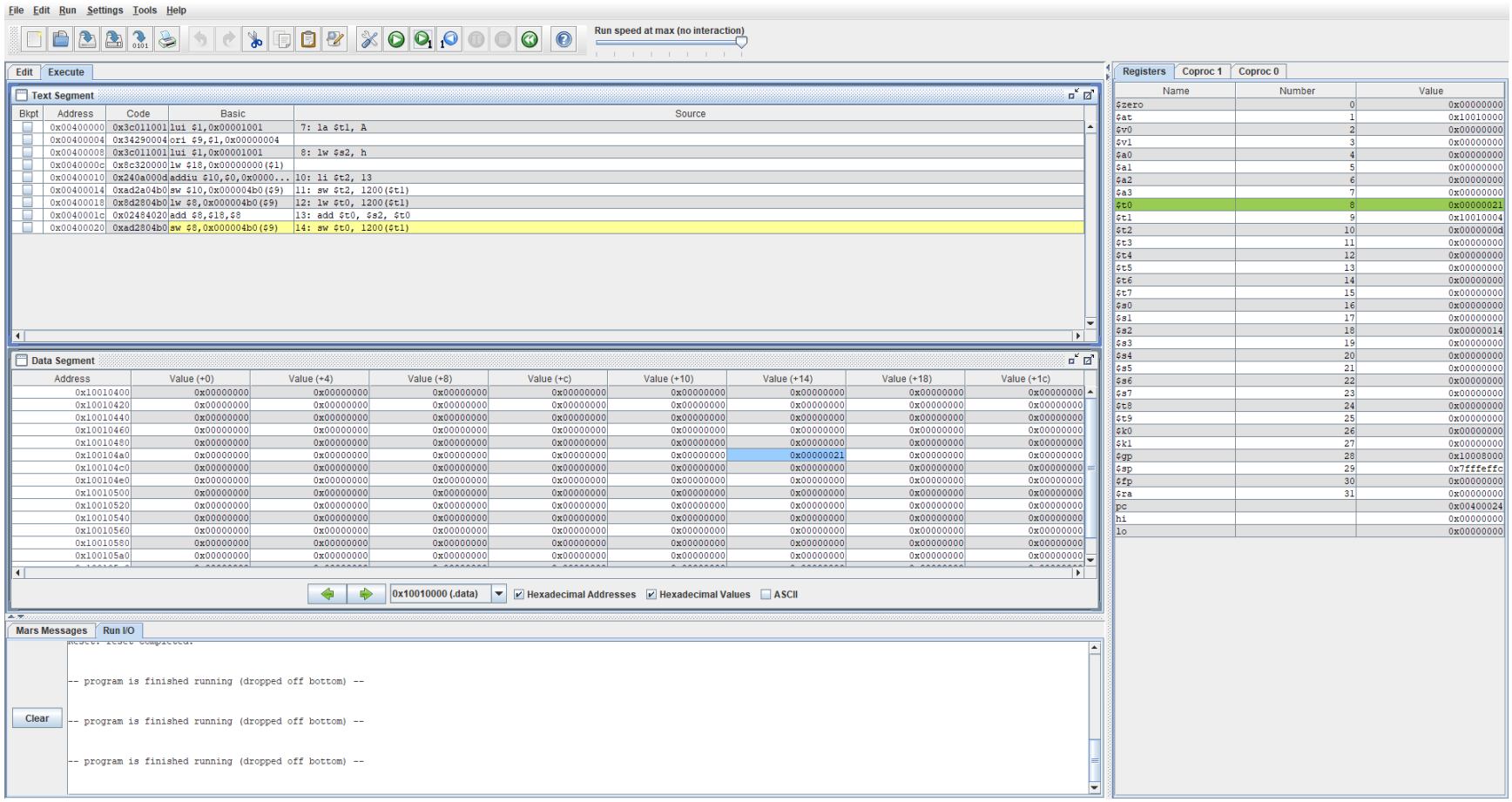
In the figure above, what is shown is the code for question 2-5_2 being compiled and ran using MARS simulator. Above what we can see is $pc which points towards the next instruction to run, $sp which is the stack pointer but will not change because we are only working with static variables. Int h and array A are both static in this code therefore $sp will not be changing in value.



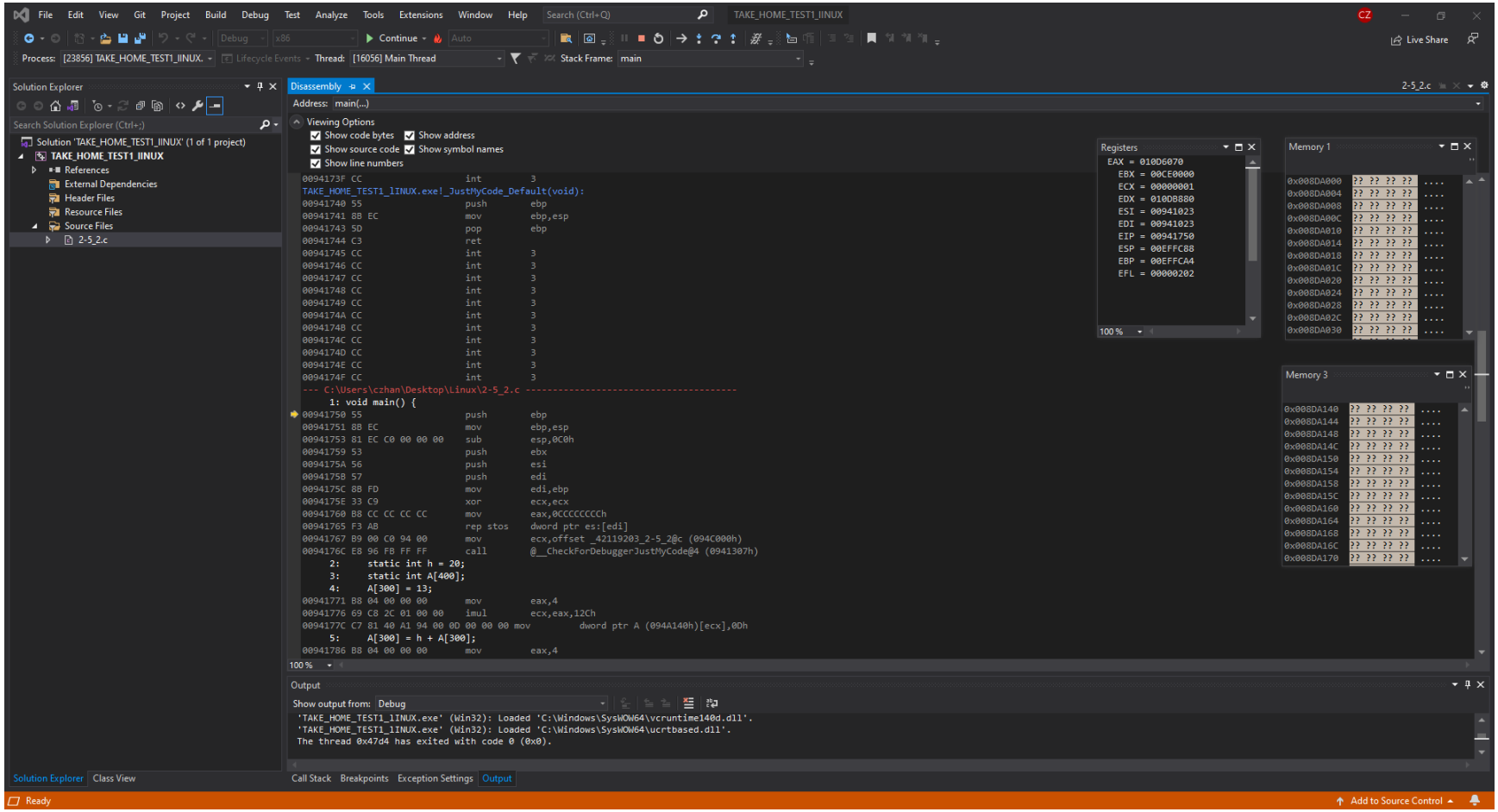
In the figure above, what we can observe is that the instructions la $t1, A and lw $s2, h were run and we can confirm that this has occurred by looking at $t1 and $s2 in the register window. La $t1, A basically means that we are loading the address of A so that we can work with the indexes of it and so we loaded it into $t1. As we can see, $t1 has the address 10010000, which is the address of Array A. Next we look at lw $2, h which basically means that we are loading the word of h into $s2. If we look at $s2, we can see that it has the hex value of 14 which when converted to decimal is 20.



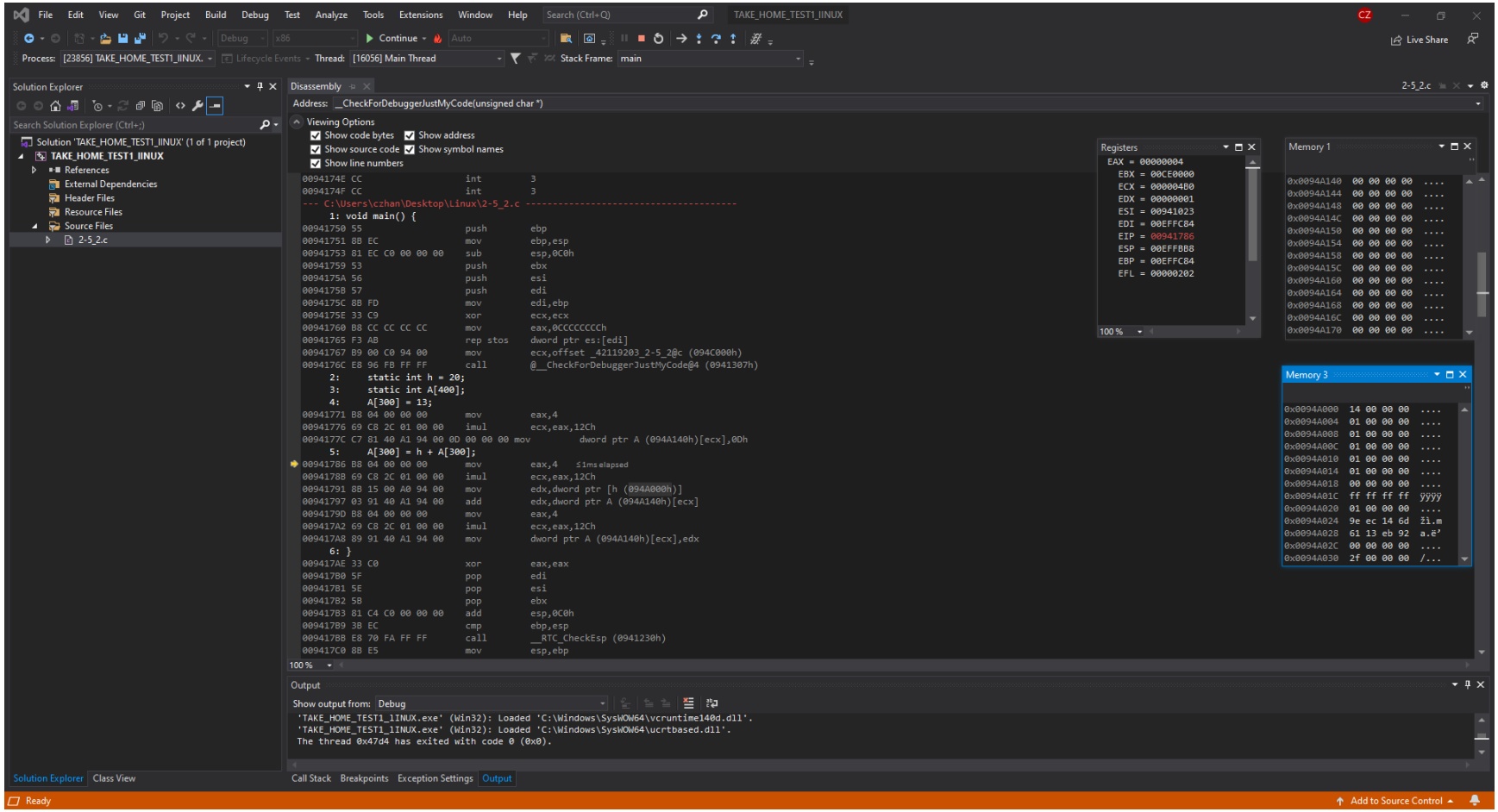
In the figure above, we have taken 3 more steps into the program where we perform the instructions li $t2, 13 which means we load $t2 with 13, sw $t2, 1200($t1) which means that we store the value of $t2 which is d, into the A[300] as indicated by 1200($t1) where t1 is the address of the array. The 3rd step that we take is lw $t0, 1200($t1) which means that we are loading the value of A[300] into $t0 and if we look at $t0, we can notice that it has the value d now, which is correct. Lastly, we can see in the data segment that value d has been stored into the 300th index of the array therefore proving correctness in code.



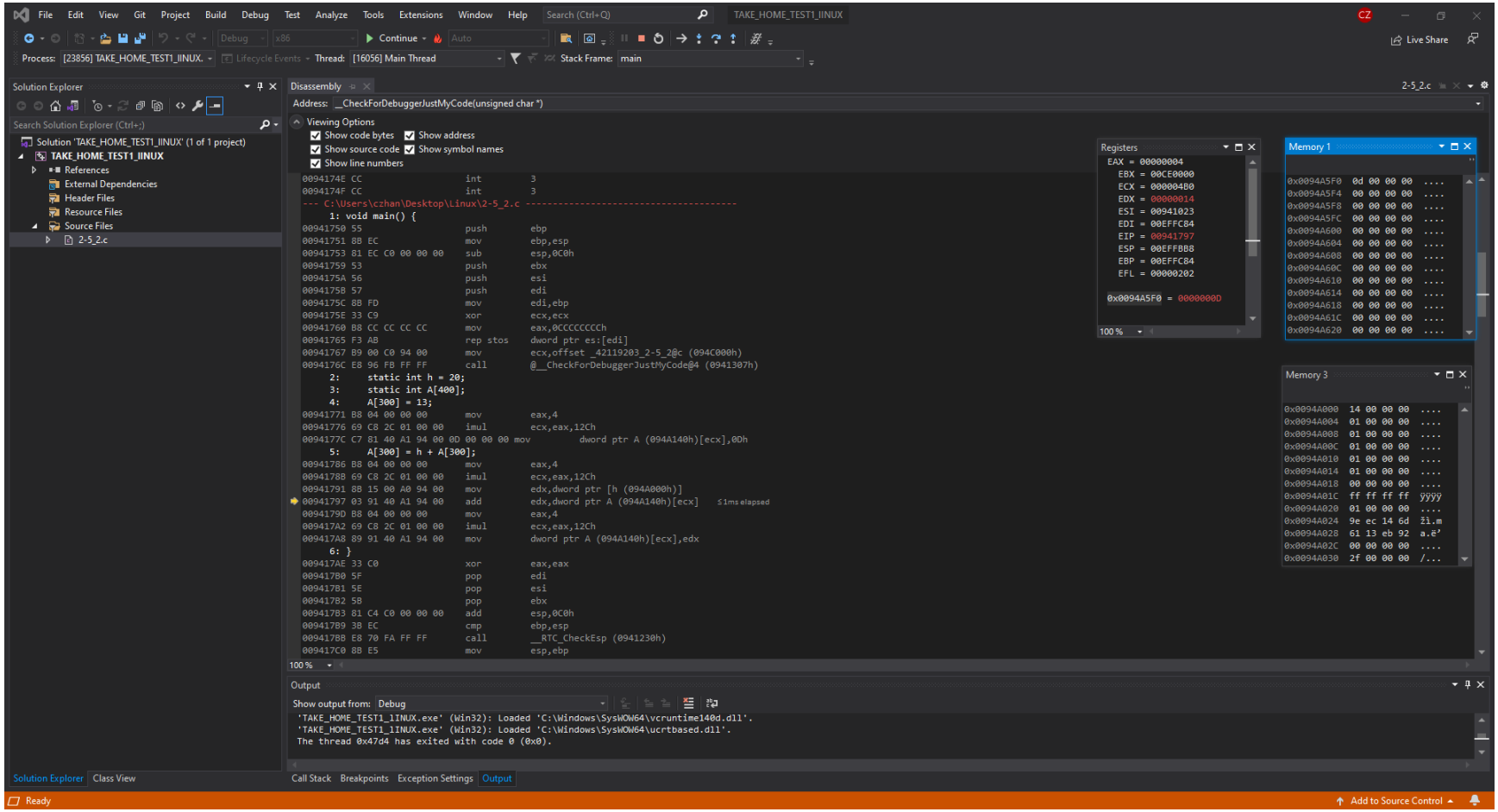
Finally, we perform the last 2 instructions which is add $t0, $s2, $t0 and sw $t0, 1200($t1). Add $t0 $s2, $t0 means that we are adding the value in $s2 with $t0, which is 20 + 13 which gives us the hex value of 21. We can look over at the register window to confirm that 21 is now the value in $t1 and from that, we can determine that the instruction has worked properly. Lastly, sw $t0, 1200($t1) saves the word into the array and we can confirm this by looking at the data segment and confirming that 21 has been stored into the 300th index.



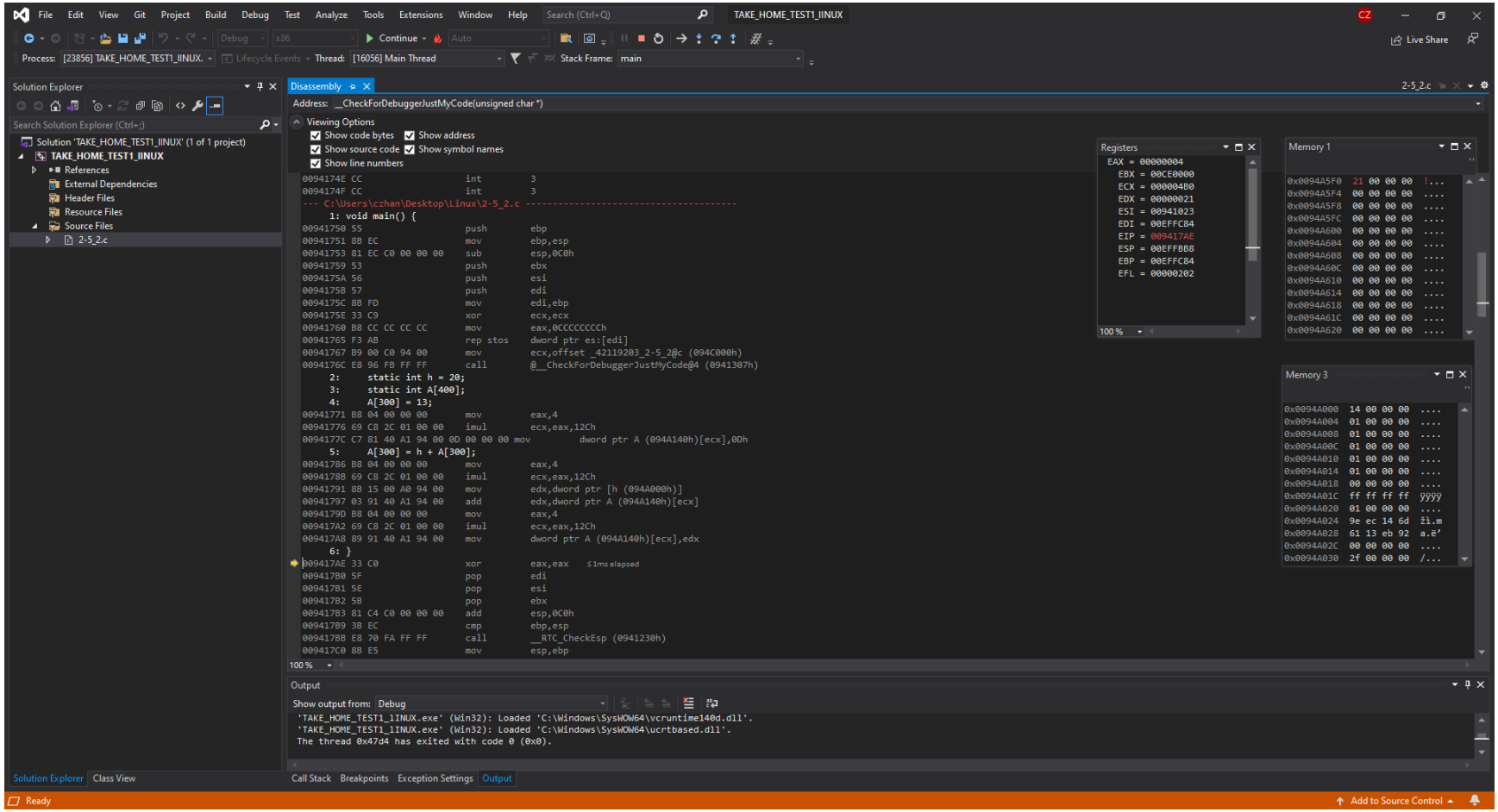
In the figure above, question 2-5_2 was compiled using the intelx86 processor in which we can observe the Registers and memory window of the program to prove correctness in code and to compare with the other compile methods. From the figure above, we can see the EIP or the instruction register which points towards the next instruction to be done, the ESP or the stack pointer which will not be changing after we load in the h and A. Furthermore, based on the C code, we have initialized both the array A and integer h as static.



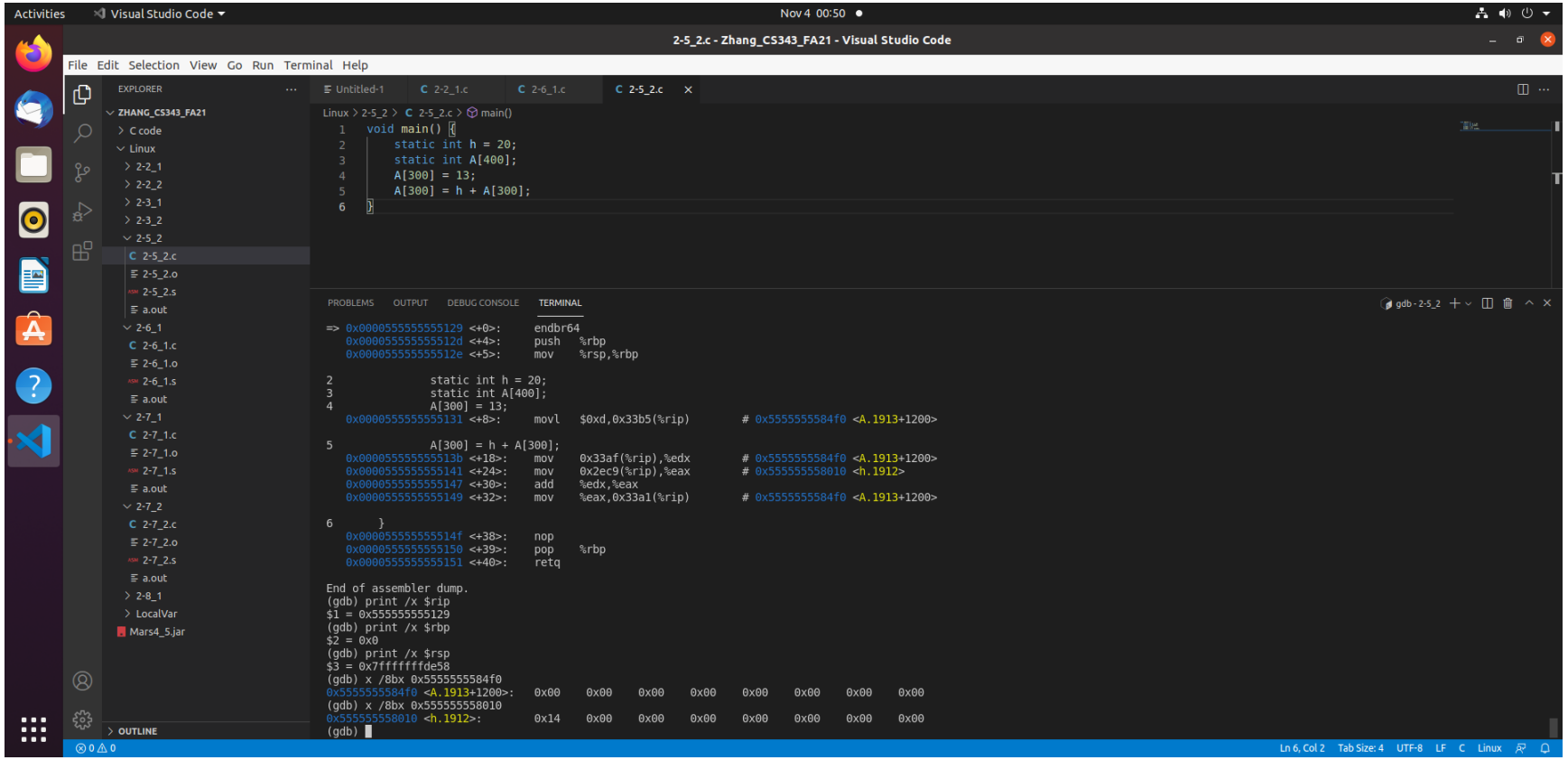
In the figure above, I have stopped the instructions after loading in values of h into its address and initializing the array. in memory window 3, we can see that the address 0x0094A000 has a hex value of 14 which is correct as 14 in hex is 20 in decimal. Furthermore, the array indexes have no values therefore all values are at 0. Lastly please remember that ESP now will not be changing as there are no static variables.



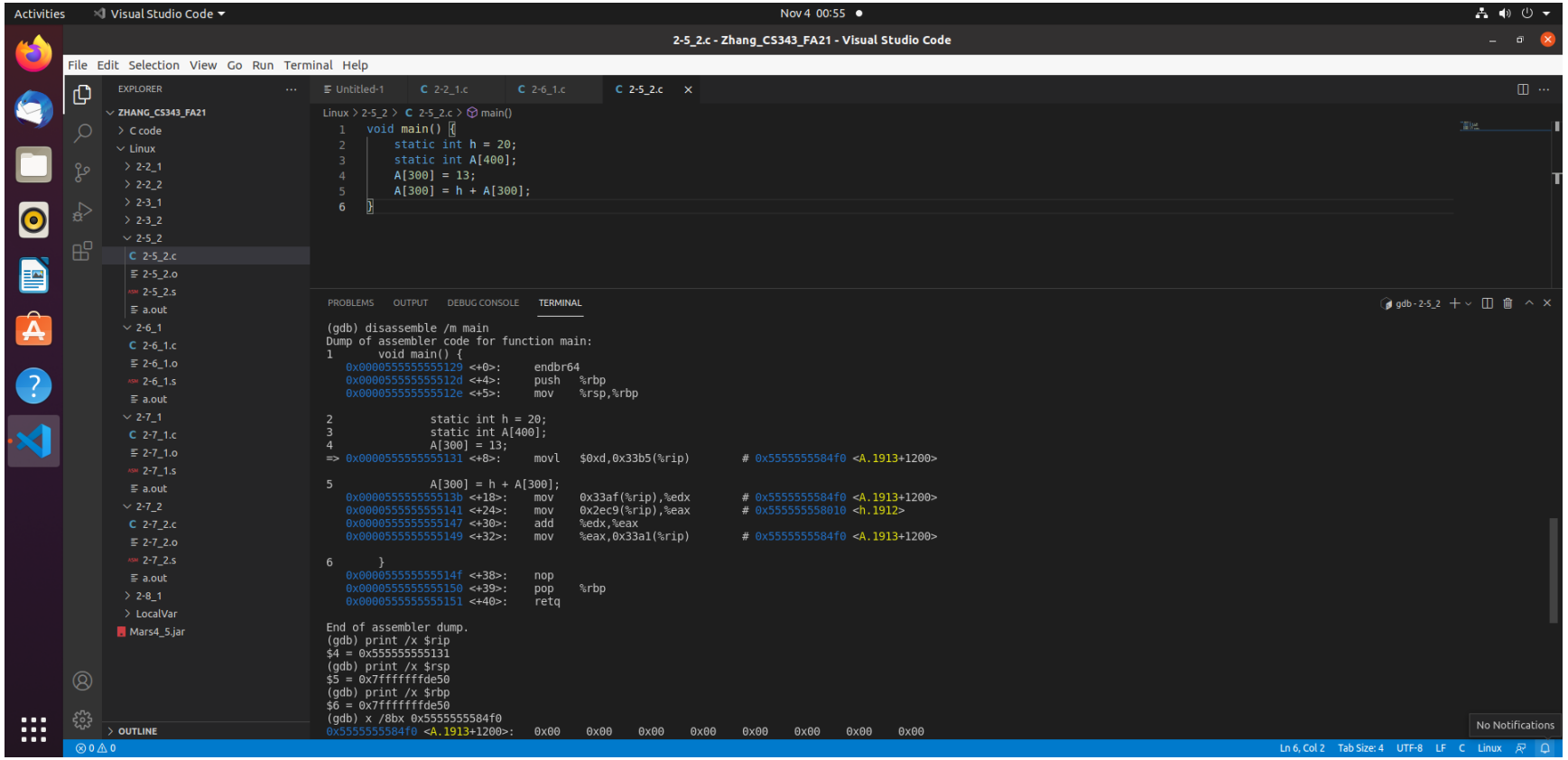
In the figure above, what is now shown is the allocation of a value into the 300th index of the Array A. If we look over at Memory window 1, we can notice that the address at the top now holds the value 0d in which 0d = 13 when converted to decimal form. Next, look at ESP and we can see that ESP has not changed therefore my statement above holds true still.



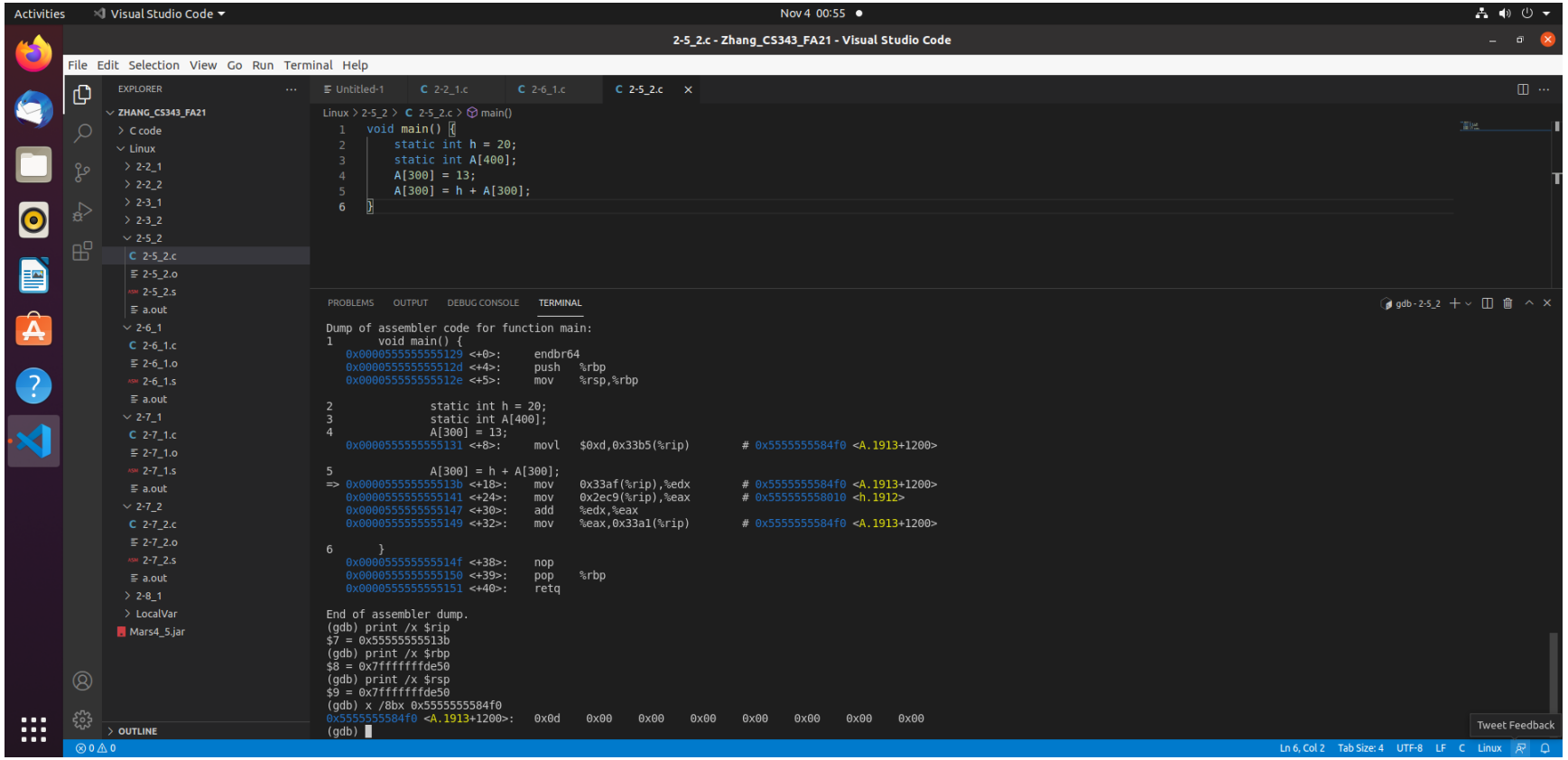
In this final figure, the instruction A[300] = h + A[300] has been performed and we can confirm this by looking at memory window 1 where the address at the top, now has the value 21. The address at top is the 300th index of the array A and it previously had the value d and now it has the value 21 which proves that instructions have ran properly. Lastly look over at ESP and we can see that it has not changed therefore what I have previously stated is true.



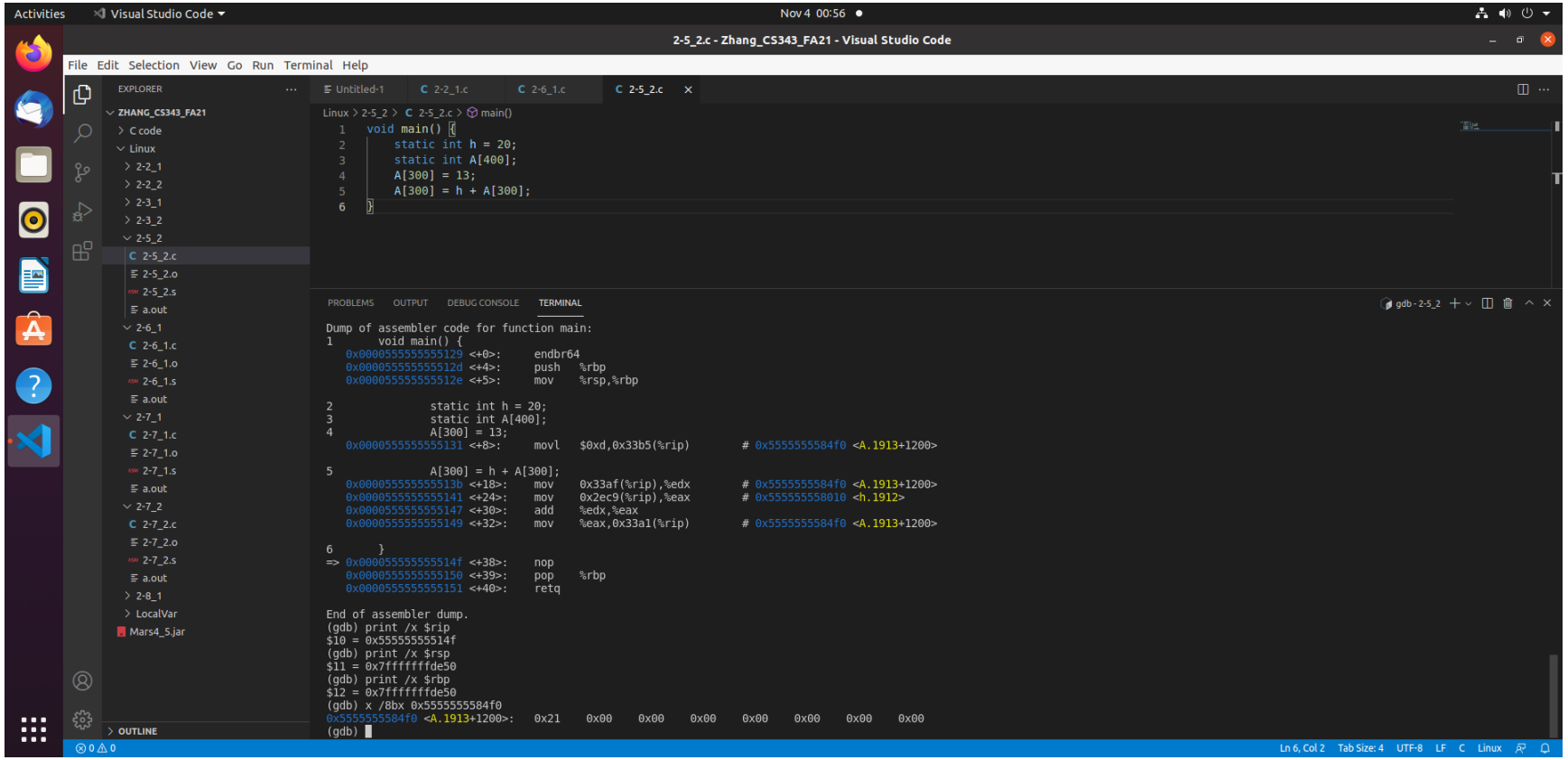
In the figure above, what is showcased is the question 2-5_2 being compiled in a LINUX environment using a windows compiler. If we look over at the terminal, what can be seen is the dump file, the $rip, $rsp, $rbp, h and array A. $rsp is the stack pointer and that value will not change as all variables are static in this program and $rip is the instruction pointer. $rbp is the base pointer which will be the same as $rsp. If we look at the bottom of the screenshot, we can see the values stored in the 300th index of array A and the value stored in the address of h, 0x00 and 0x14 respectively.



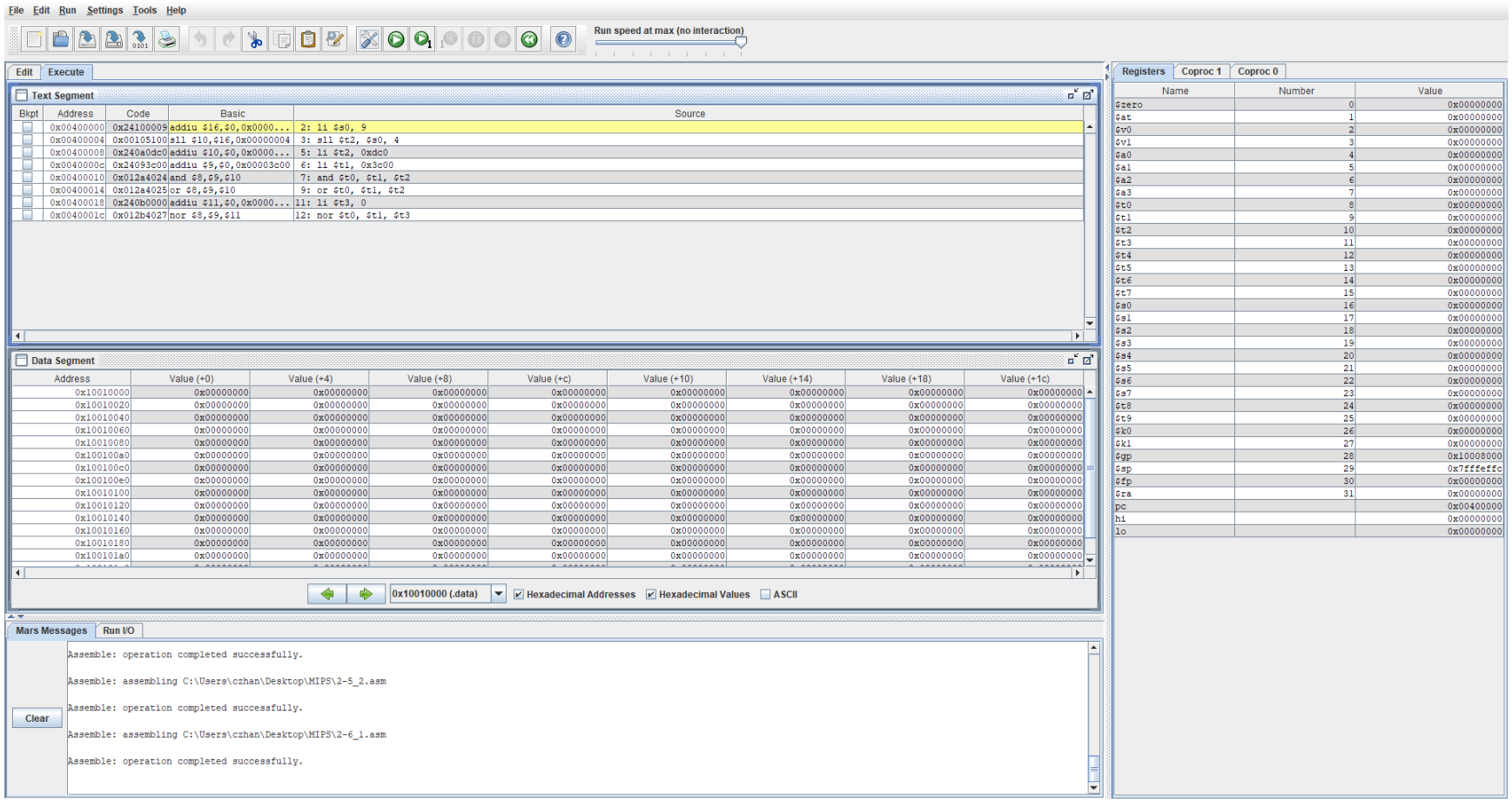
Next, we go to the next instruction and we can notice that the $rip has a new instruction it is pointing to which it should and that the array A at 300th index still has no value. $rsp still has the same value because all variables are static, none local. We can also confirm that $rip is correct by looking at the dump file



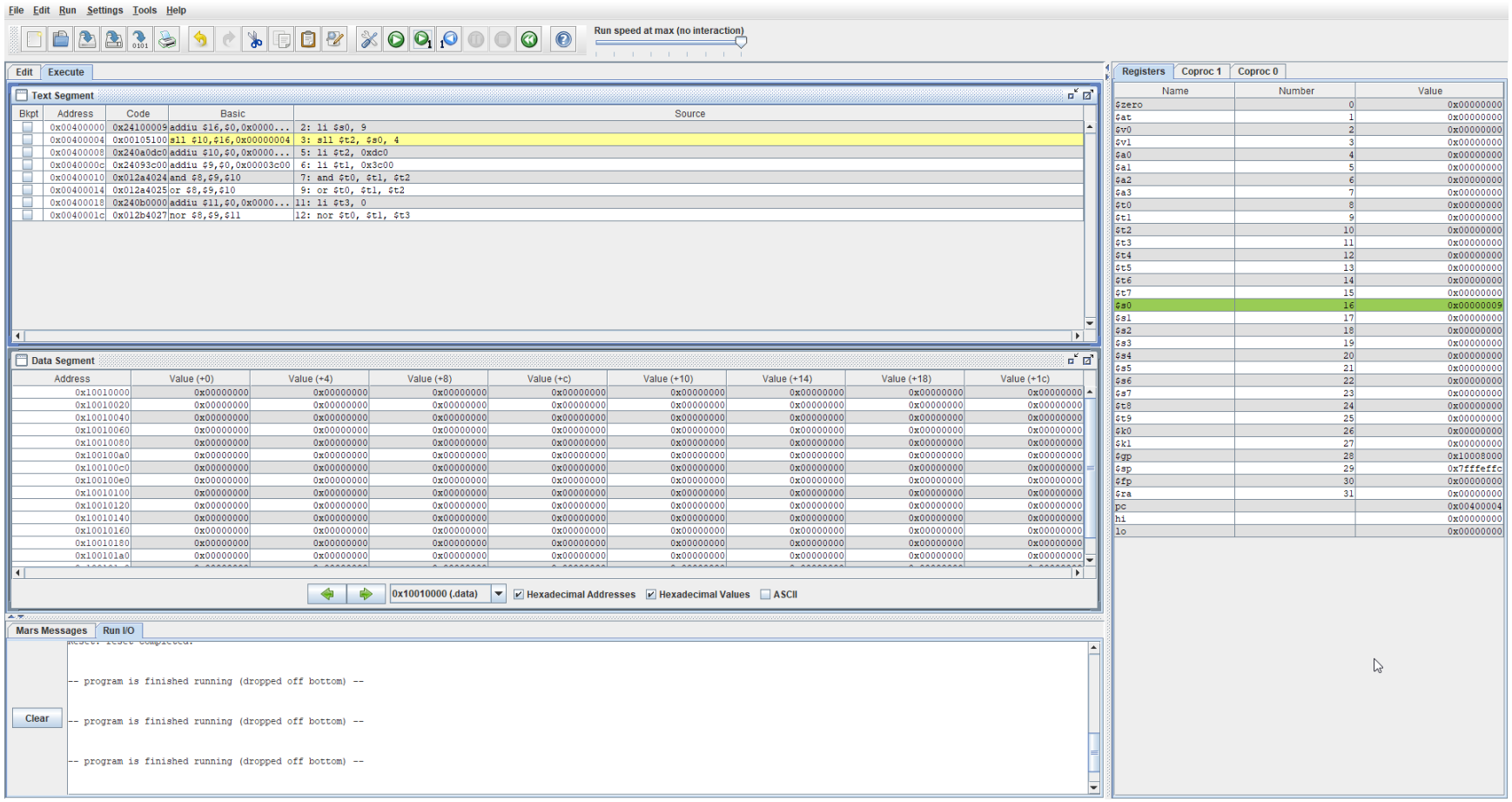
In the figure above, what is showcased is the value 13 being stored into the 300th index of array A. We can now see that the value stored in the 300th index is now d whereas before it was 0. Furthermore $rsp remains unchanged and to confirm the instruction pointers correctness, we can look at the dump file.



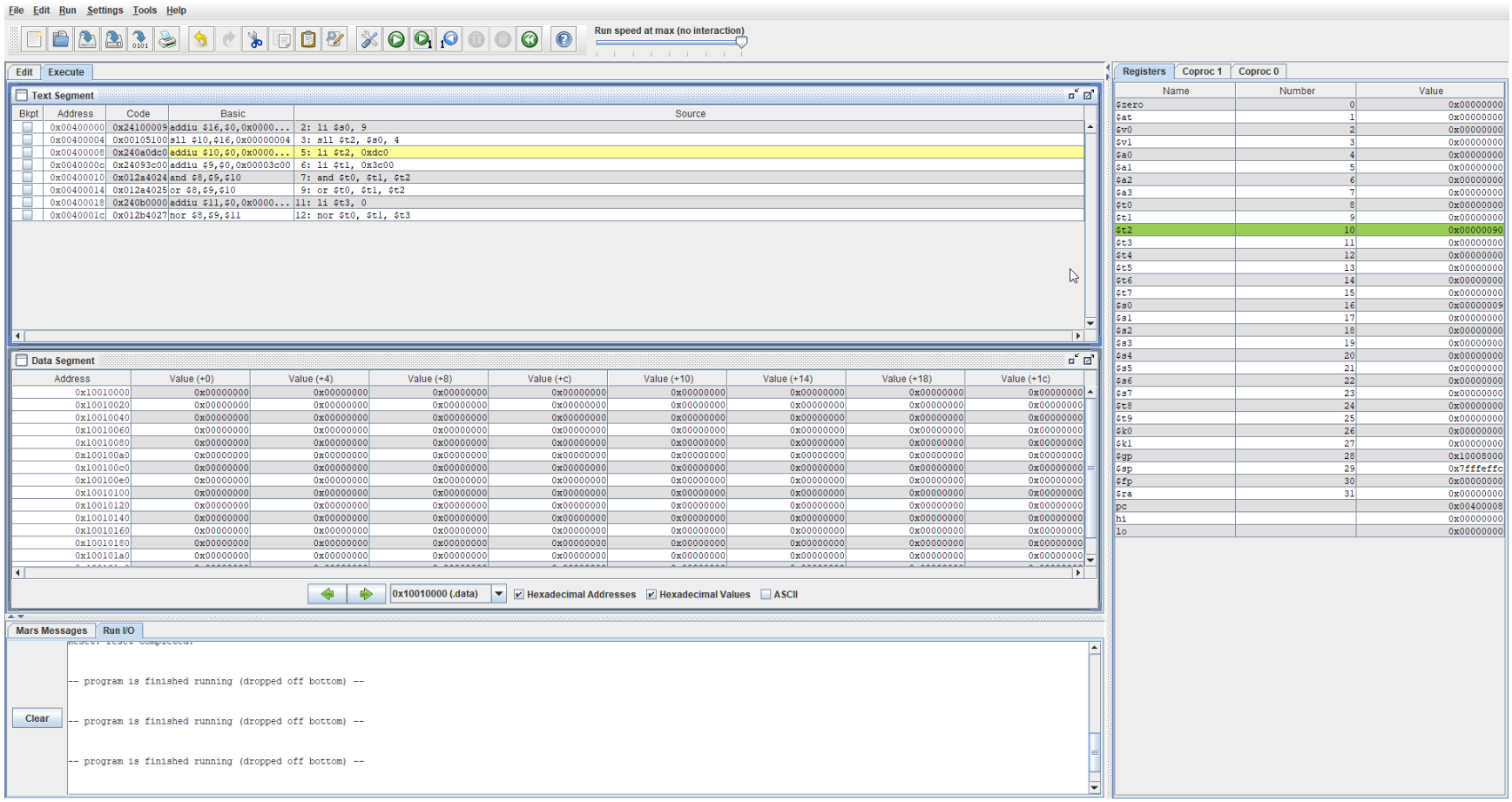
Lastly, in this figure we can see the A[300] + h = A[300] being performed as the value stored in A[300] is now 0x21 as opposed to 0x0d which was stored in it before. What has happened is 20 + 13 = 33 which is 0x21 in hex value. Notice $rsp also has not changed and that $rip is in also correct as we can confirm this by looking in the dump file.



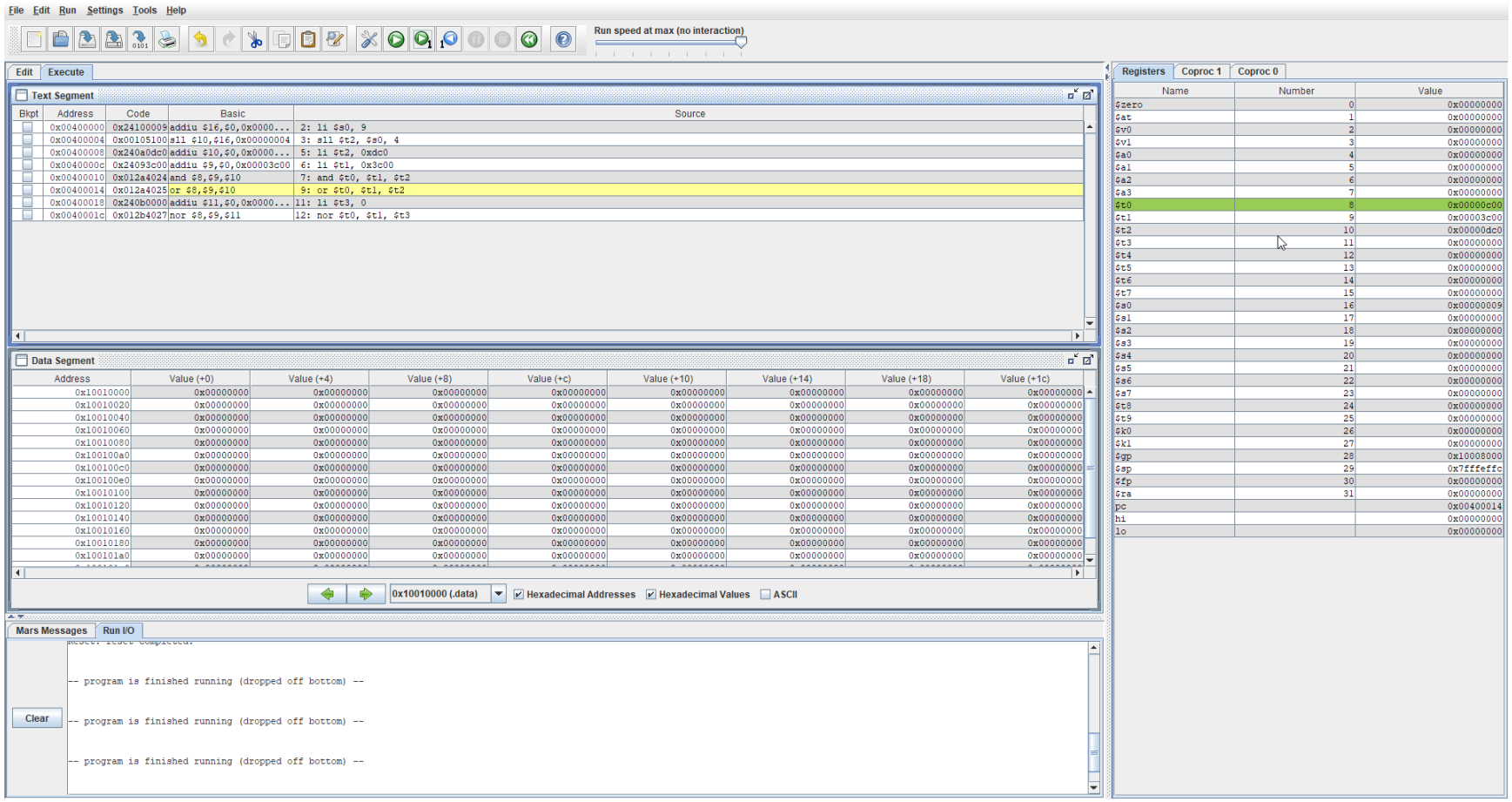
In the figure above, what will be performed is shift left, and, or and nor in the MARS simulator. What is provided is a register window, a text segment and a data segment. $pc tells us what instruction is next, $sp tells us the stack pointer, however the value of $sp will not change as we are not working with any local variables.



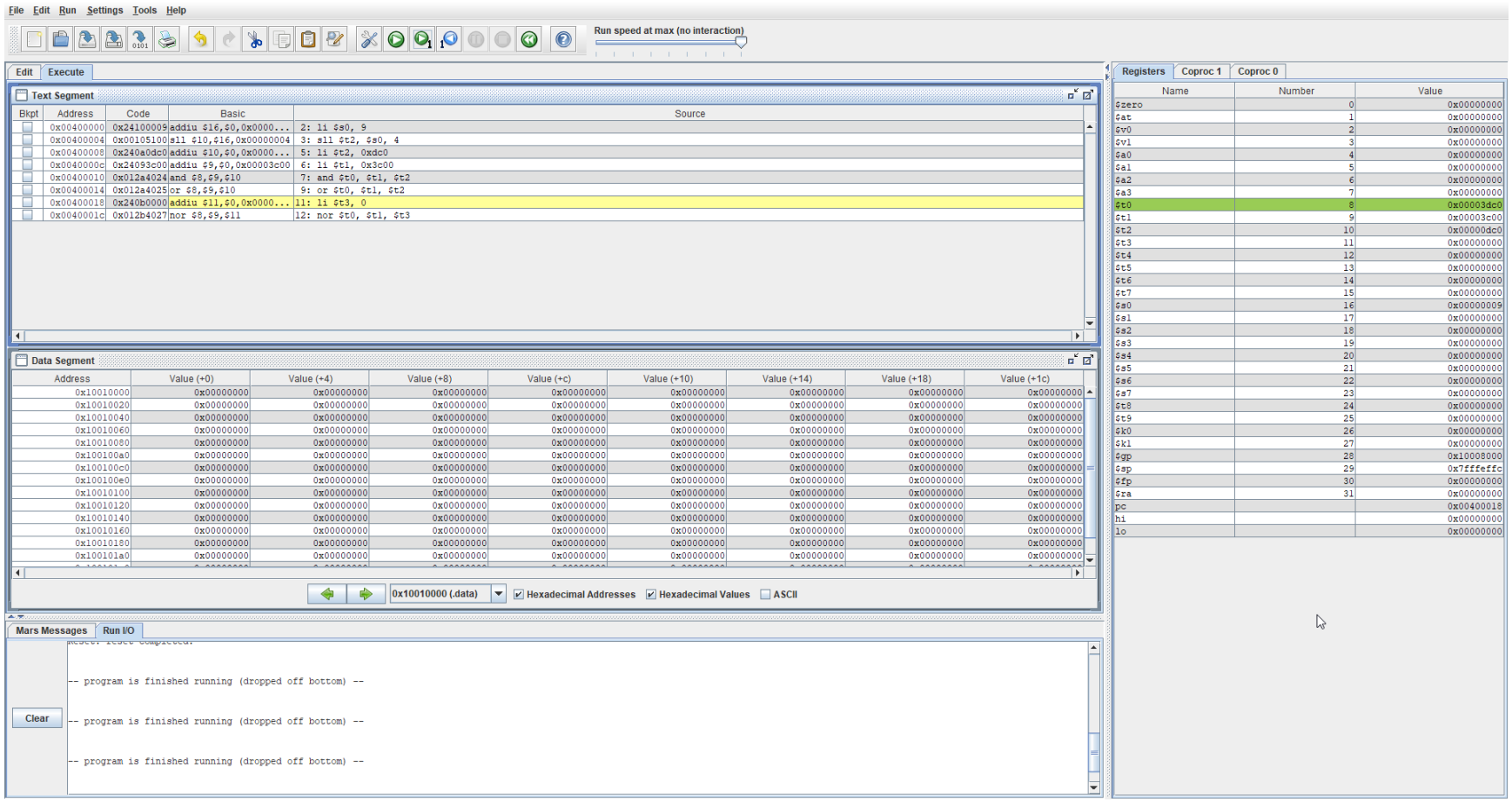
In the figure above, what was performed was li $s0, 9 which means that we are writing the value 9 into $s0 so that we can perform left shift on it. Notice the $pc that points towards the next instruction to be done and that $sp is still the same.



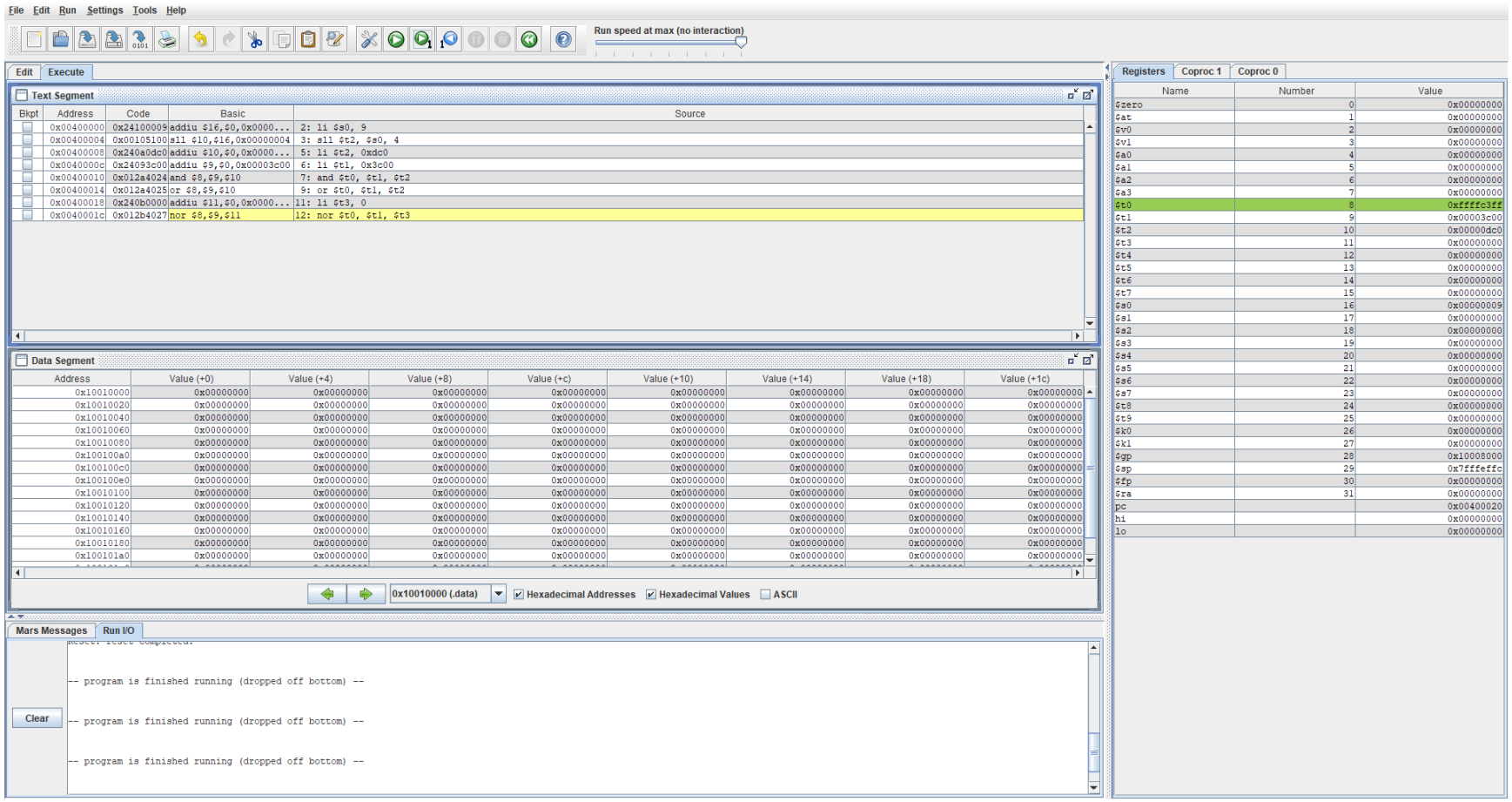
In the figure above, shift left was performed and as we can see in $t2, 9 has been shifted 1 bit leftwards. If we look at the Text segment, we can notice that the shift value is 4 and that is because with each offset, we add 4 therefore we use 4 as the amount we shift. Suppose, we use 1 instead of 4 as the shift amount, what happens then is that instead of having 9 being shifted left, what is written in $t2 is 12 which is incorrect.



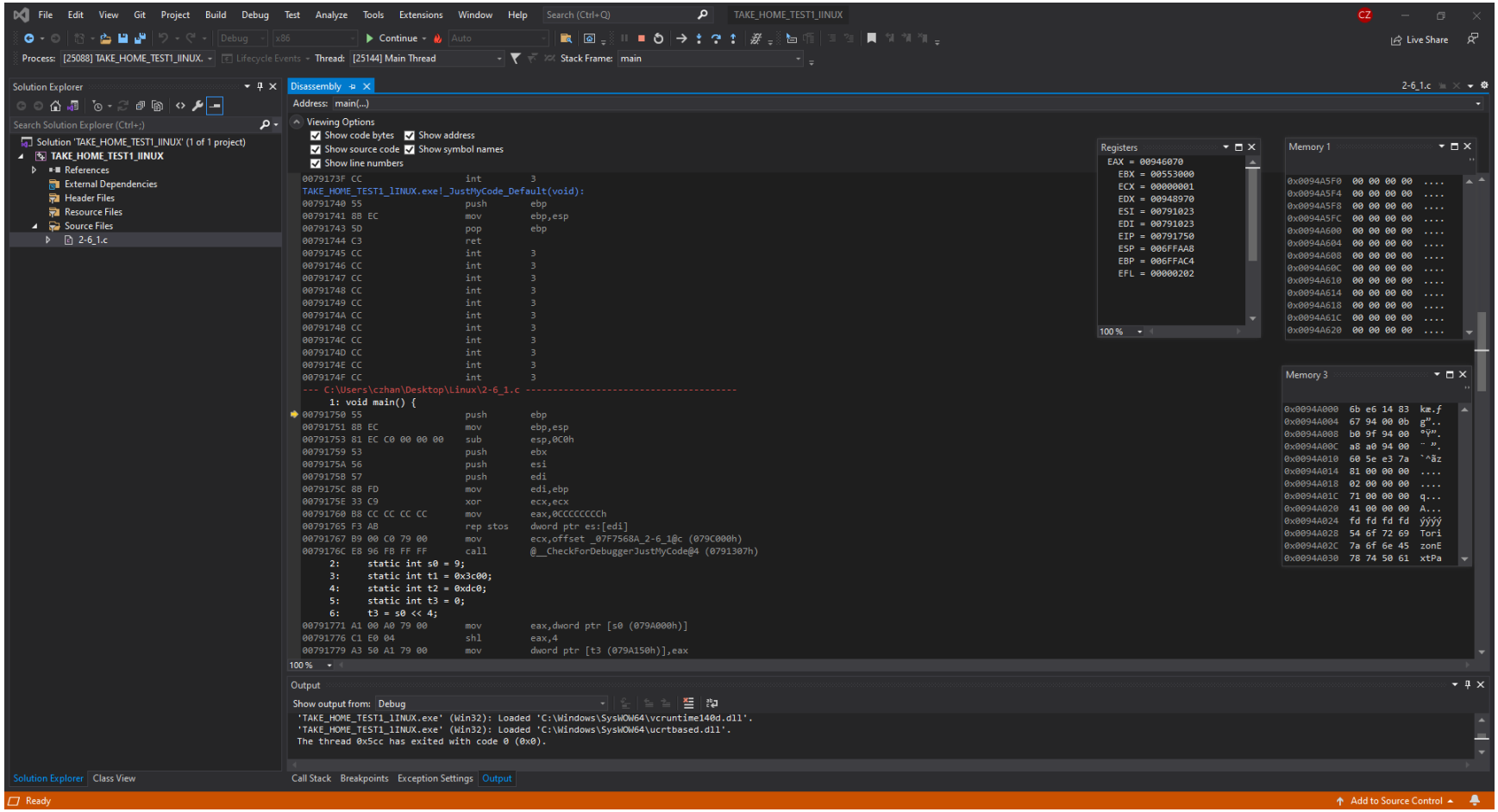
In the figure above, what was performed was the and operation where we loaded in the values 0xdc0 and 0x3c00 into $t2 and $t1 respectively. This instruction can be seen in the instruction address 0x00400008 and 0x0040000c. Next, we perform the and operation on $t2 and $t2 which gives us 0x00000c00 which is correct and therefore proves that the instruction for and has been conducted perfectly.



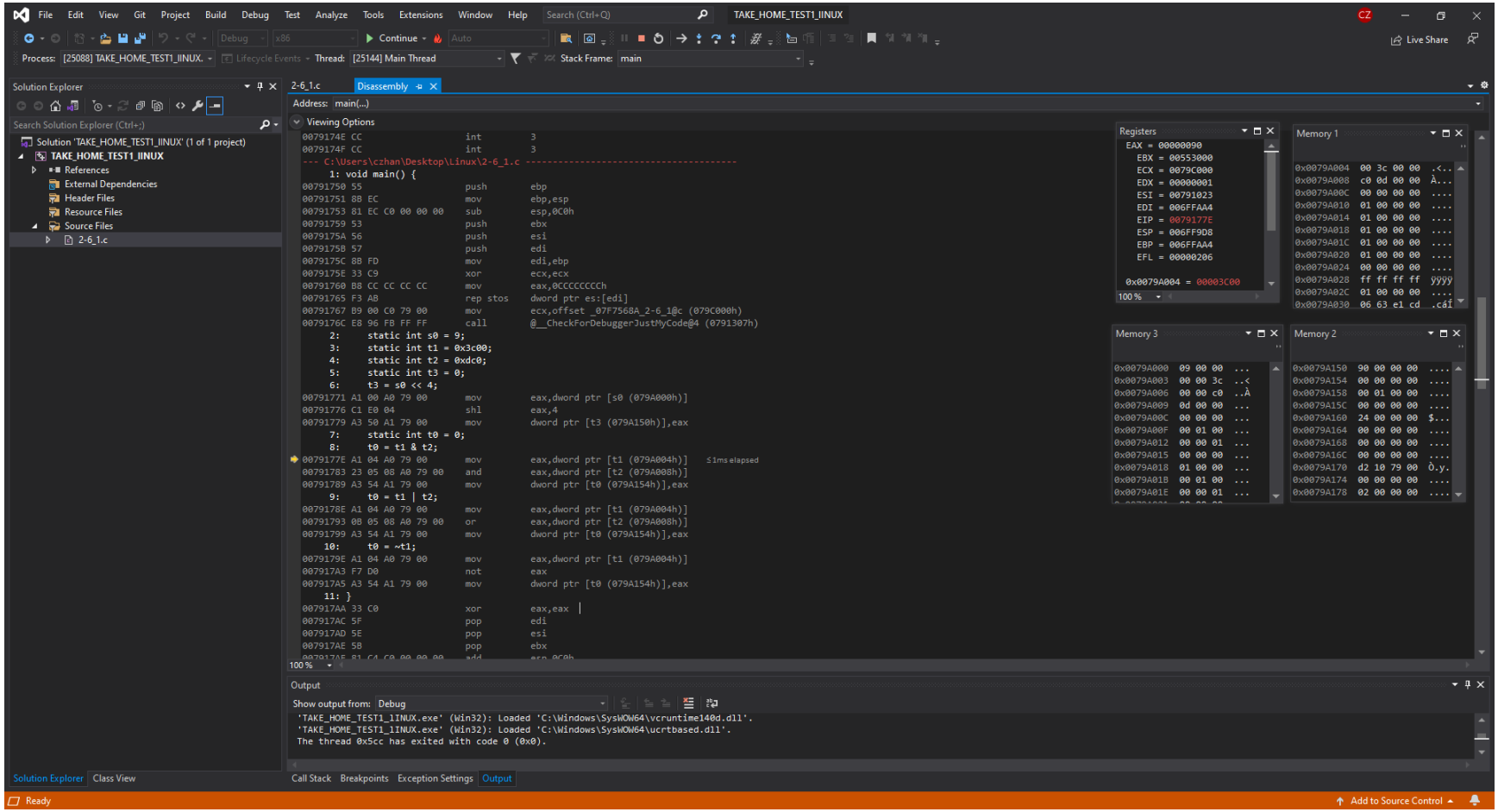
In the figure above, what is performed is the or operation and we can notice in $t0, a new value of 0x00003dc0. What occurred was $t1 and $t2 had the or operation done of it, then the result was written over to $t0 which is correct. Finally, look at $pc to confirm what the next instruction address and compare it with the addresses in the text segment address tab.



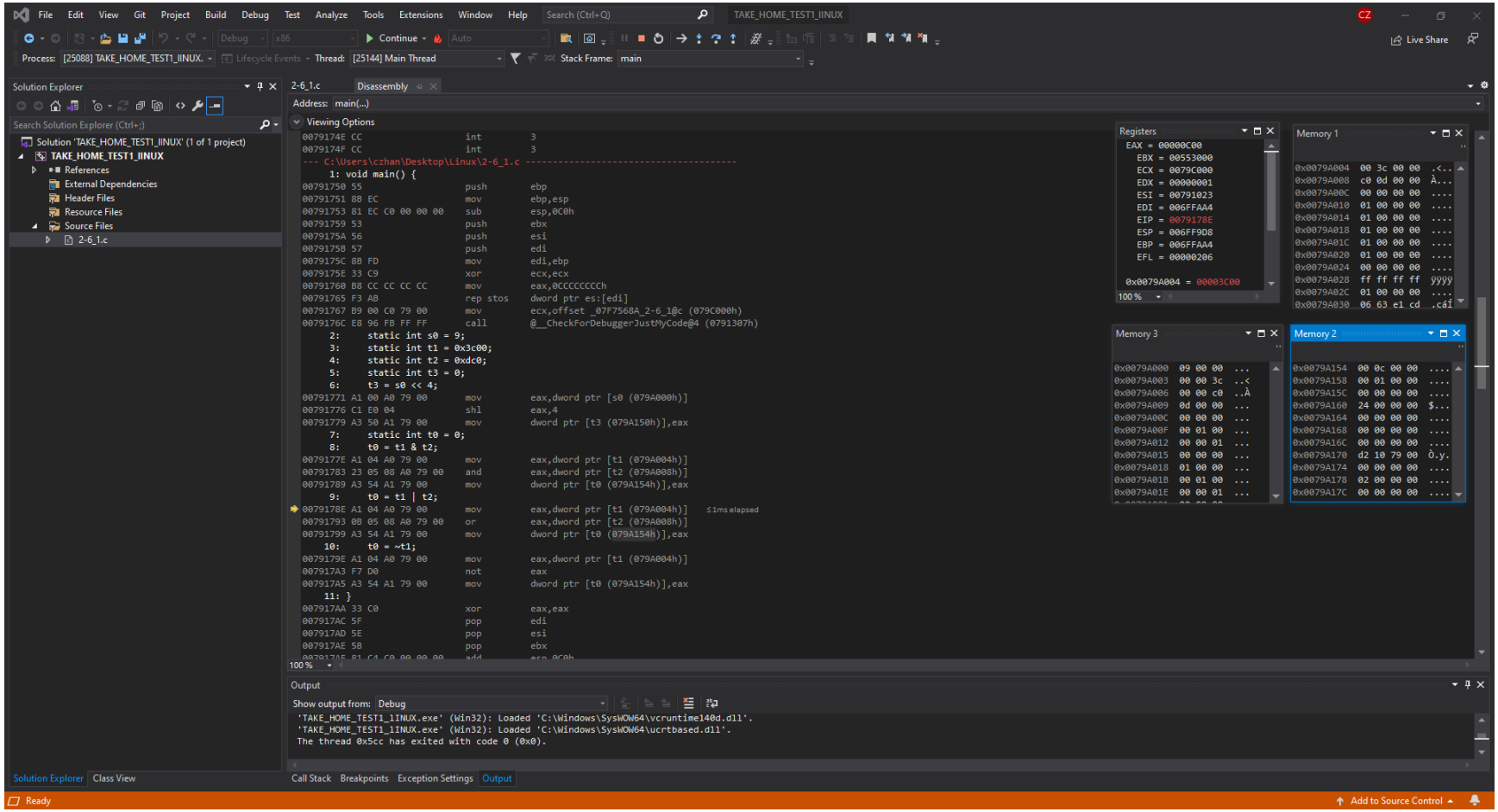
Lastly, in this figure, the nor instruction was done and 0 was also loaded into the register $t3. To confirm the correctness of the operation, we look at $t0 as the output was written there. In $t0 what was written is 0xffffc3ff which is correct.



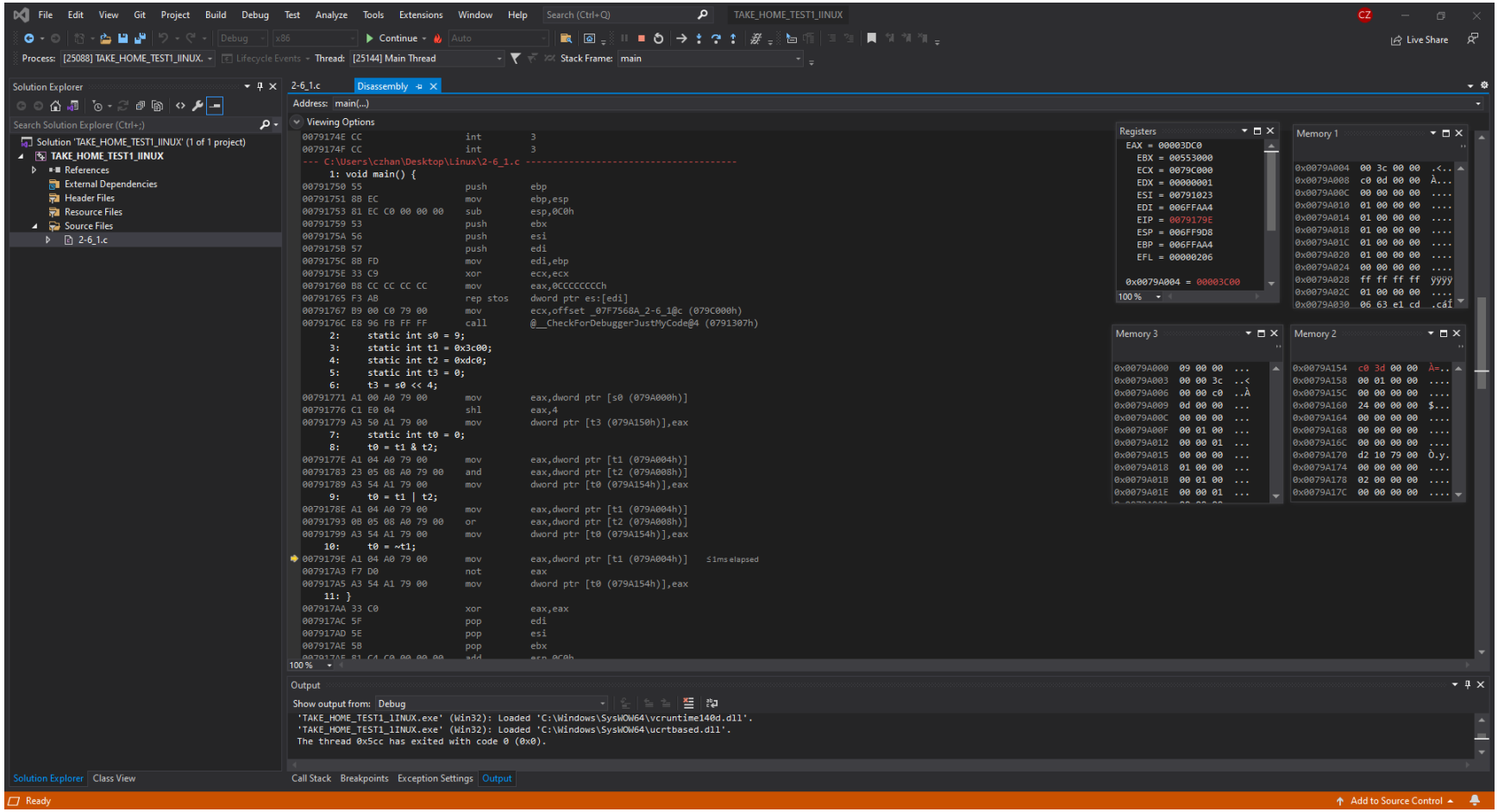
In the figure above, the question 2-6_1 has been compiled using the intelx86 processor and we can see the register window which shows the stack pointer (ESP), the instruction pointer (EIP) and others more. Furthermore, we also have memory windows which will help us confirm that values are being written into addresses properly and to compare with the other compilation methods.



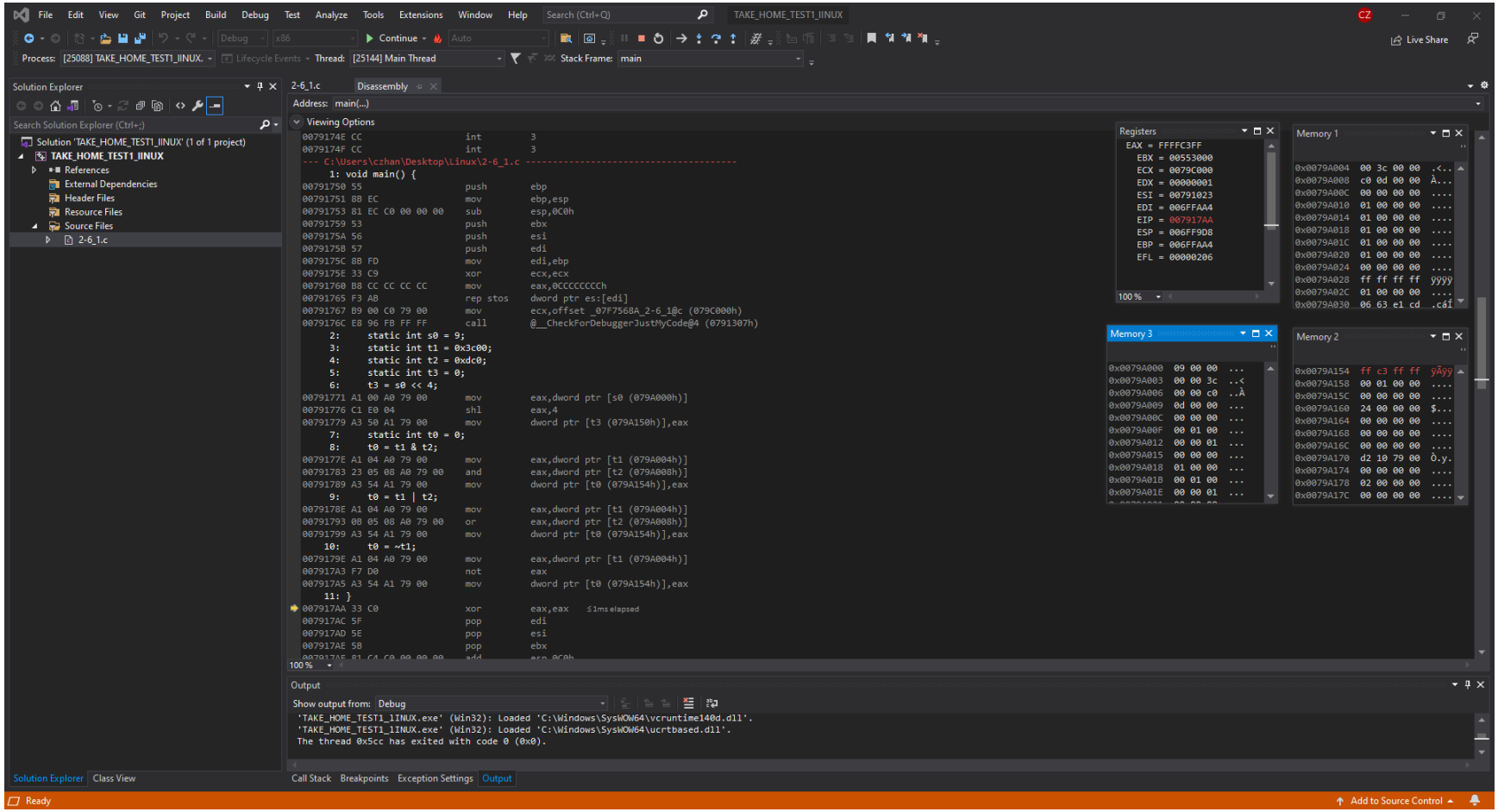
In the figure above, what is showcased is the shift left operation and the initialization of s0, t1, t2 and t3. If we look over at memory window 1, we can see t1, t2 and t3 being initialized as 3c00, dc0 and 0 respectively. Next we look over at memory window 3 which has the value 9 written into it and in memory 2, what was performed was the shift left operation. At the top most address, we can see that the value stored is 90 which proves that the shift left operation has been conducted properly. Furthermore, please pay attention to the ESP as it will not change as there are no local variables to work with.



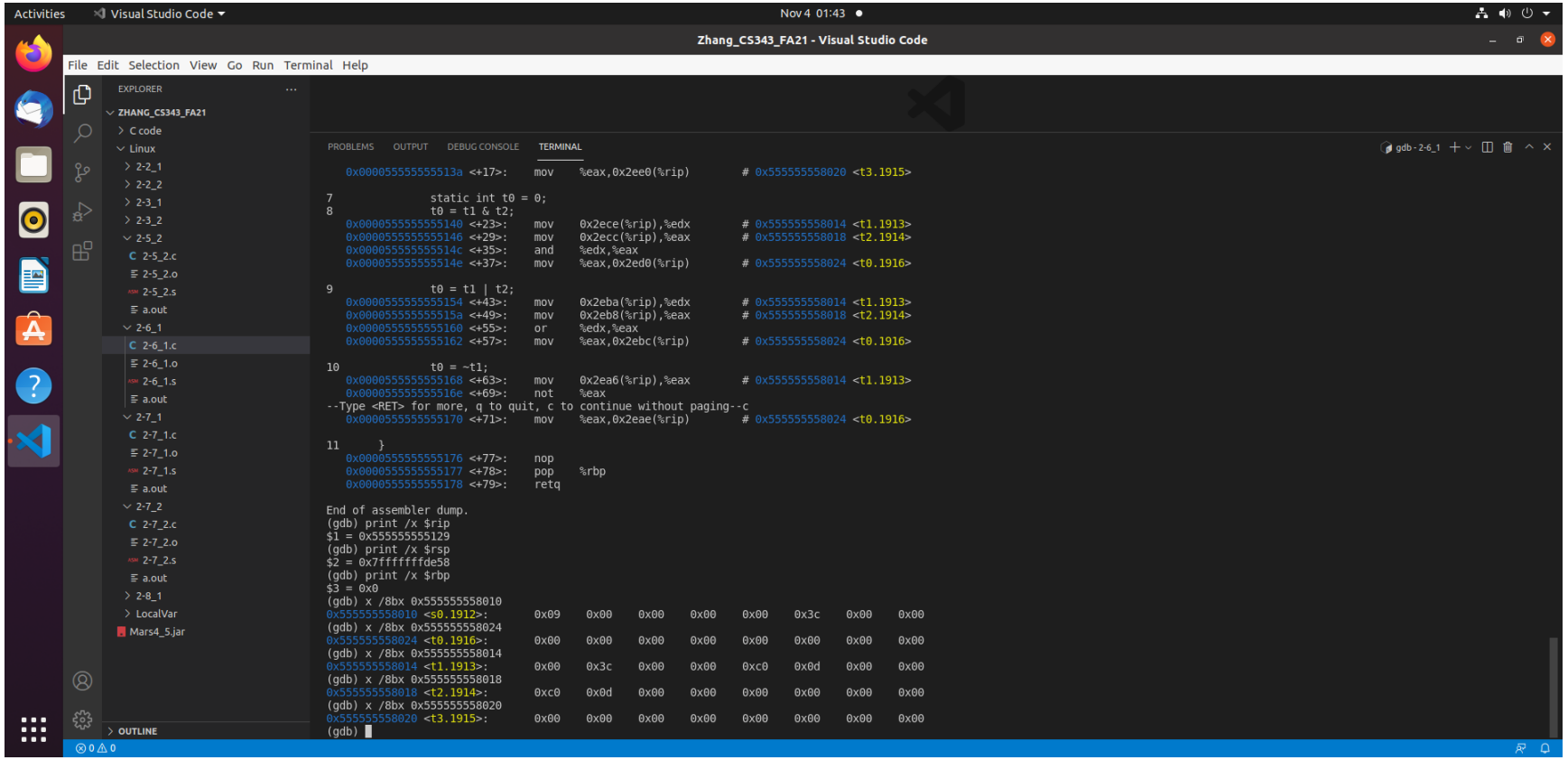
In the figure above, we are performing the and operation onto t1 and t2, then storing it into t0. If we look over at Memory window 1, we can see t1 and t2 being stored at the top 2 most addresses. Next if we look at memory window 2, we can see that the operation t1 and t2 has been conducted and the value outputted is 0c00 which is correct. Lastly, we can look at ESP and notice that it has not changed and EIP's next instruction is 007817BE. If we look for that address in the disassembly file, we can see which instruction is next.



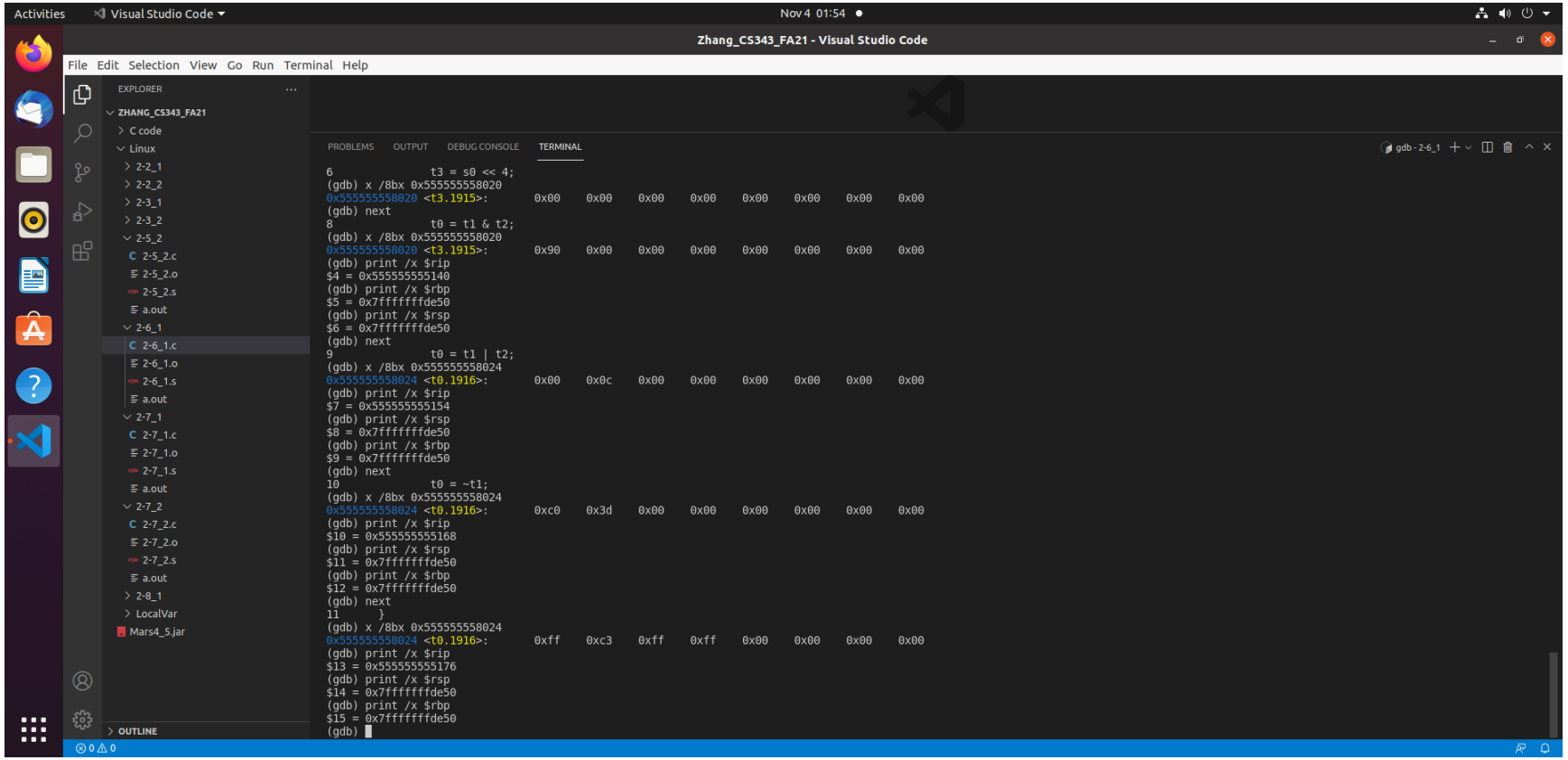
In the figure above, what is performed is the or operation on t1 and t2 then outputted onto t0. Like the and operation, we look at memory window 1 and see that the top 2 most addresses hold the values of t1 and t2. Then at Memory window 2, we can see that the value 3dc0 is written in the top most address which is the output value of t1 or t2. Furthermore, we can notice that ESP has not changed because once again, there are no local variables to work.



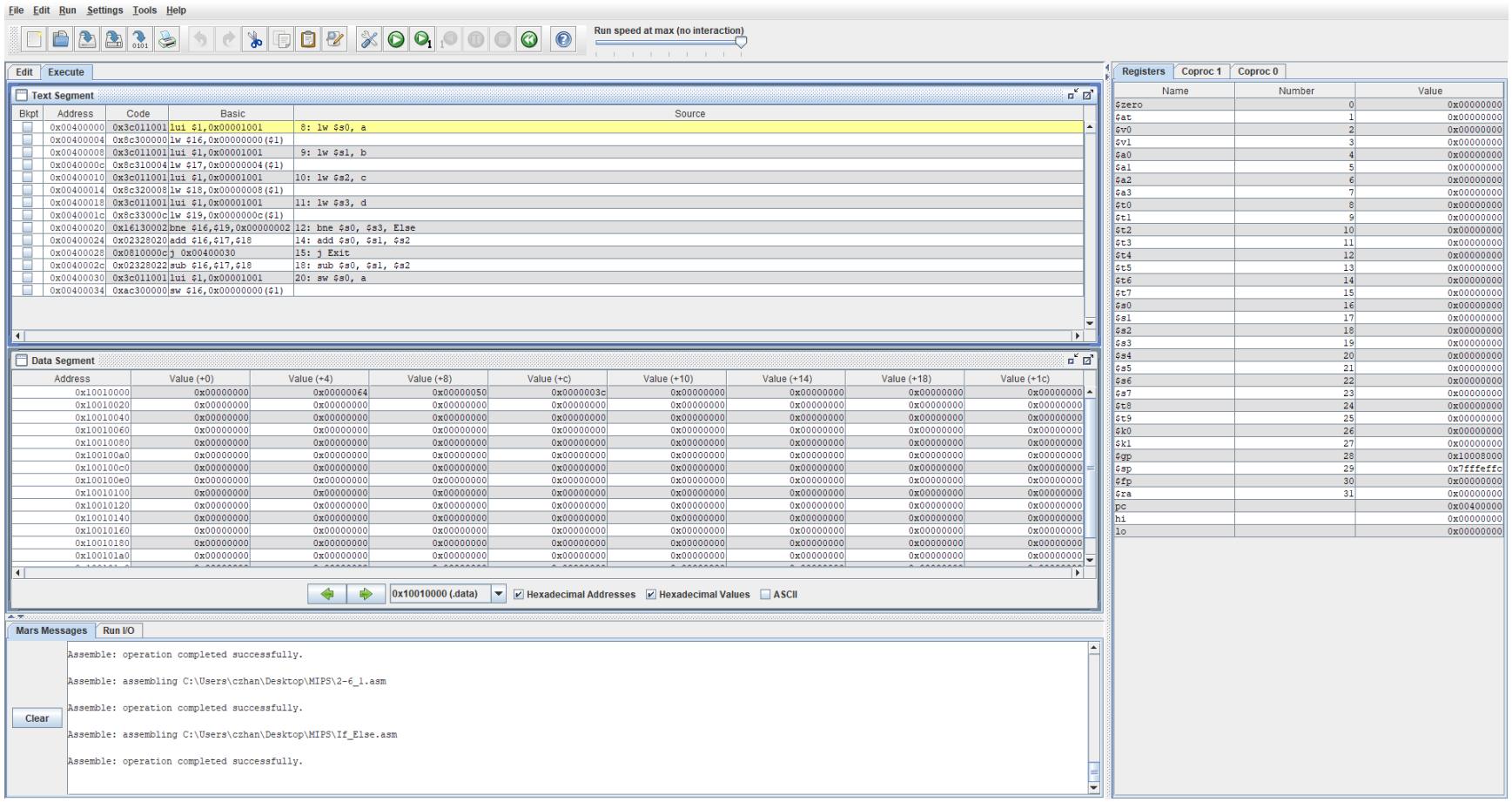
In this final figure, we perform the nor operation onto t1, t2 and t3. In Memory window 1, what is held in the top 3 addresses are the values of t1, t2 and t3 and in Memory window 2, the output of the nor operation on t1, t2 and t3 is in the top most address. The difference between MIPS instruction and the intelx86 instruction is that we are doing not t1 for the nor operation, however, the same value is outputted, which is, ffffc3ff.



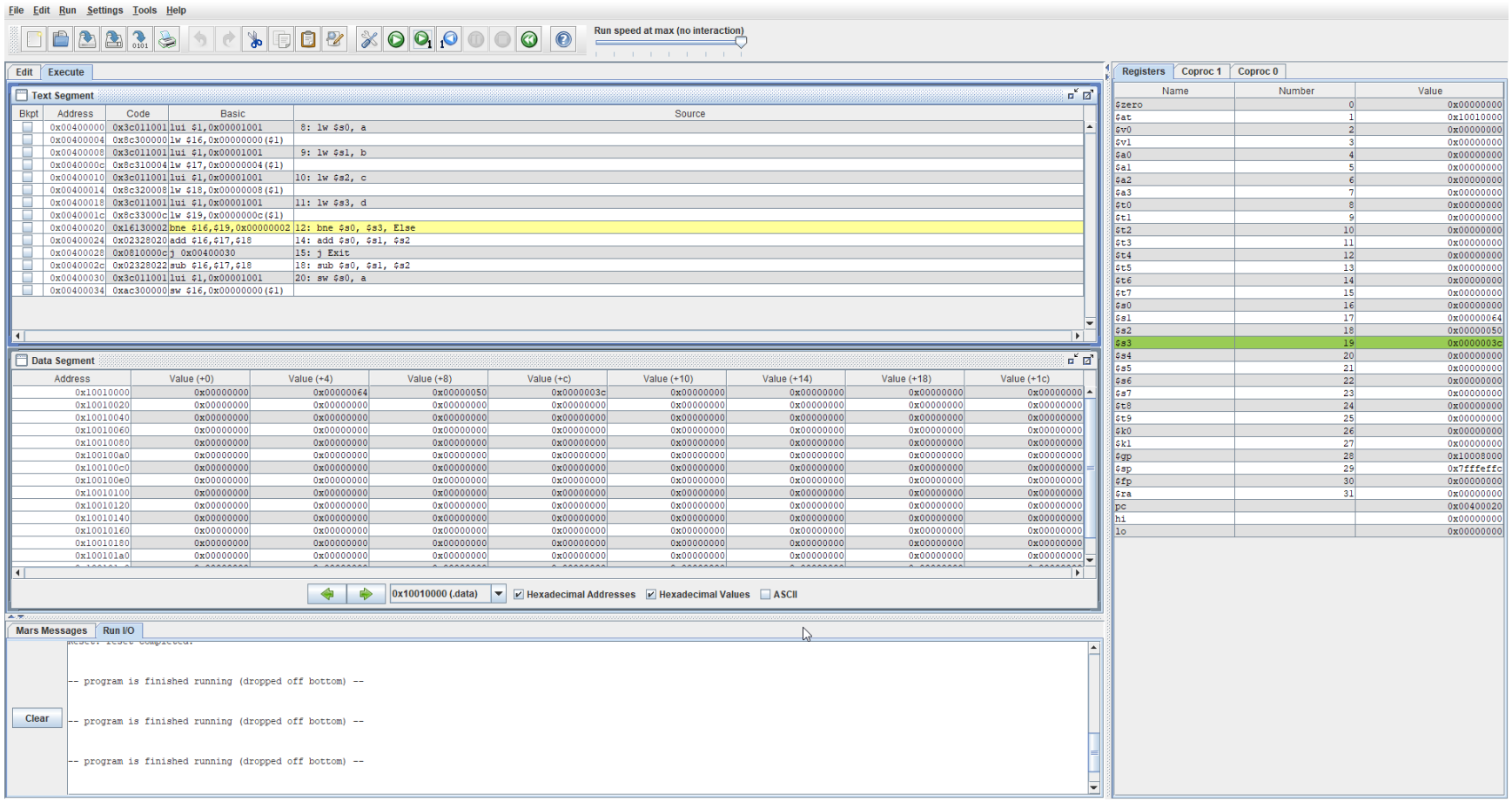
In the figure above, what is shown is question 2-6_1 being compiled with gcc windows compiler. Using GDB, we obtained the dump file of 2-6_1 and I have also printed out $rip, $rsp and $rbp as well as s0, t0, t1, t2, t3. We can refer back to this figure to ensure that $rip is following the instructions provided in the dump file. Furthermore $rsp won't change as there are no local variables and $rbp shares the same value as $rsp. In the above figure we can see that s0 = 0x09, t0 = 0x00, t1 =0x3c00 , t2 = 0dc0, t3 = 0x00.



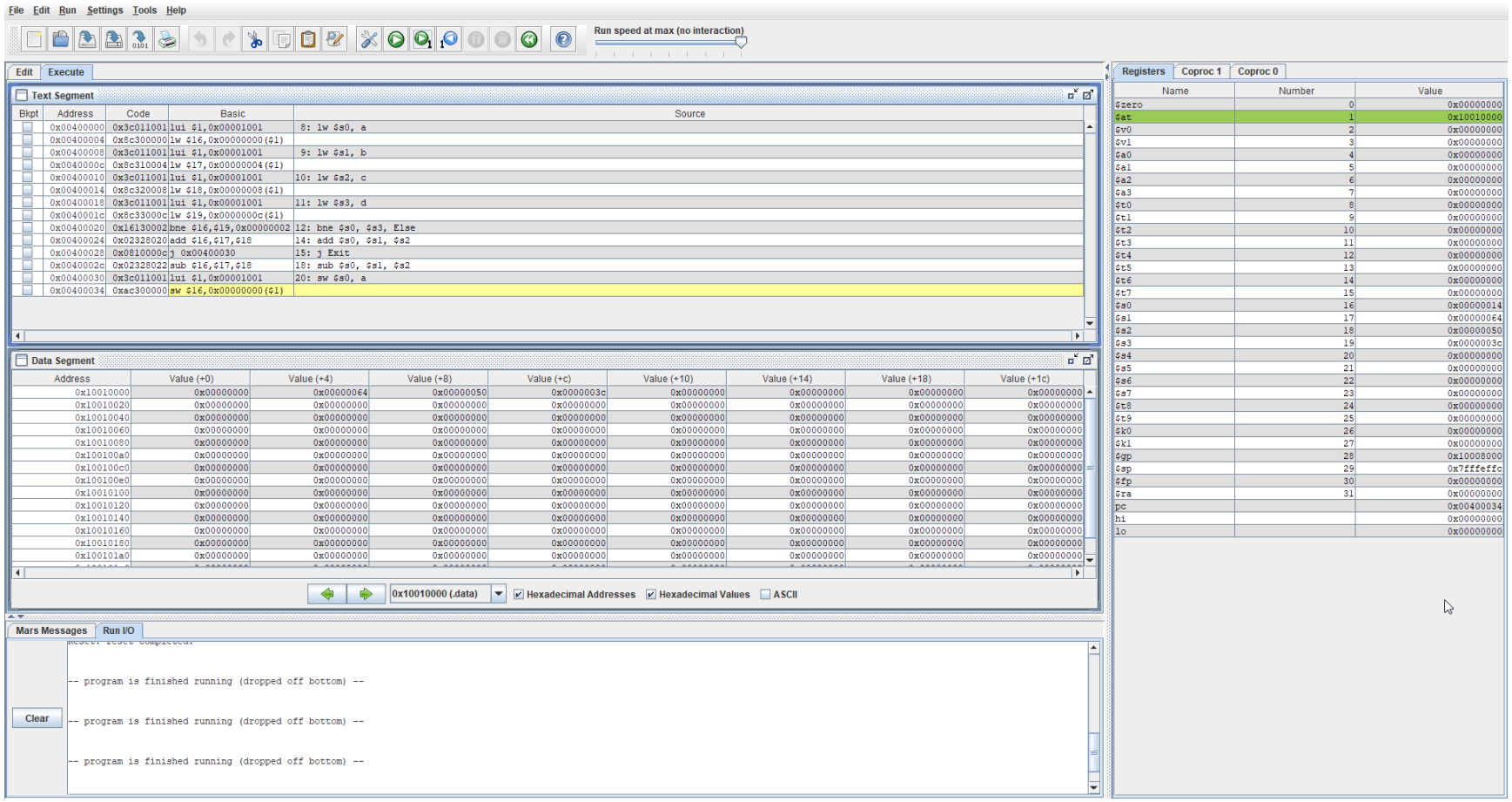
In the figure above, what is shown are the operations shift left, and, or and nor being conducted. If we look at the line under t0 = t1 & t2, we can see that the shift left operation has been conducted properly as 0x09 when shifted left by 4 is 0x90 and t3 is outputted 0x90. Next is the and operation which we can observe under the line t0 = t1 | t2. What we can see is that t0 = 0c00 which when compared to the other compilation methods, proves that this is correctly done. Next is the or operation which we can observe under the line t0 = ~t1. What we can see is that t1 or t2 returns the value of 0x3dco which is correct when comparing to the other outputs of the intelx86 and MARS compilers. Last is nor operation which we can observe under the line }. What we notice is that the output is 0x7777c377 and when comparing to the other compilers output, they are the exact same. As a closing statement to question 2-6_1 being compiled with gcc is that we can look at each $rip printed and look for them in the dump file in the figure previously mentioned to confirm correctness in instruction pointer and $rsp never changed.



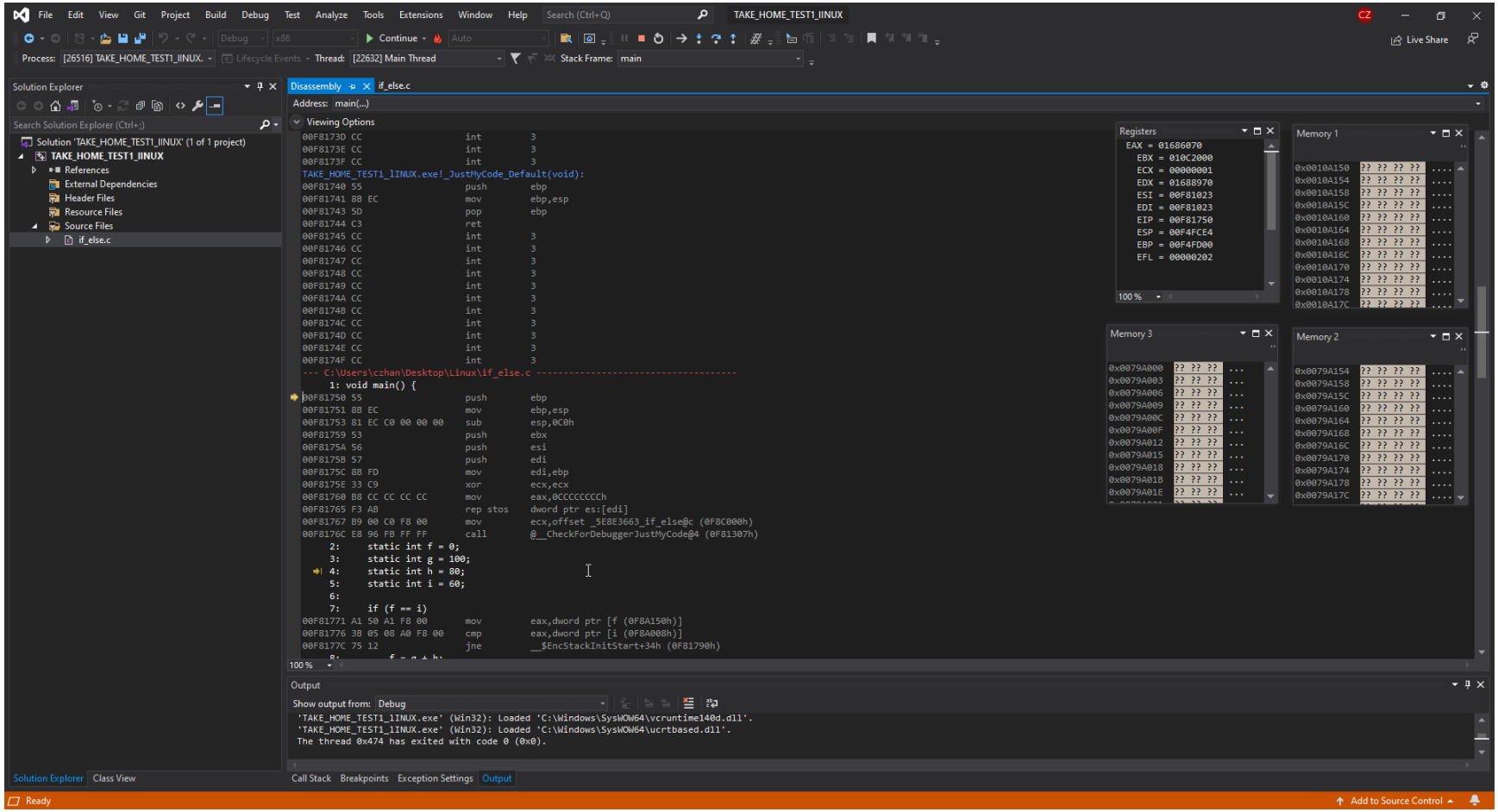
In the figure above, what is showcased is the question 2-7_1, a modified version of 2-2_1 that includes an if-else statement. We can look at $pc to see what next instruction and $sp to look at the stack pointer. We also have a data segment to see saved words and the text segment to see instructions that are being run.



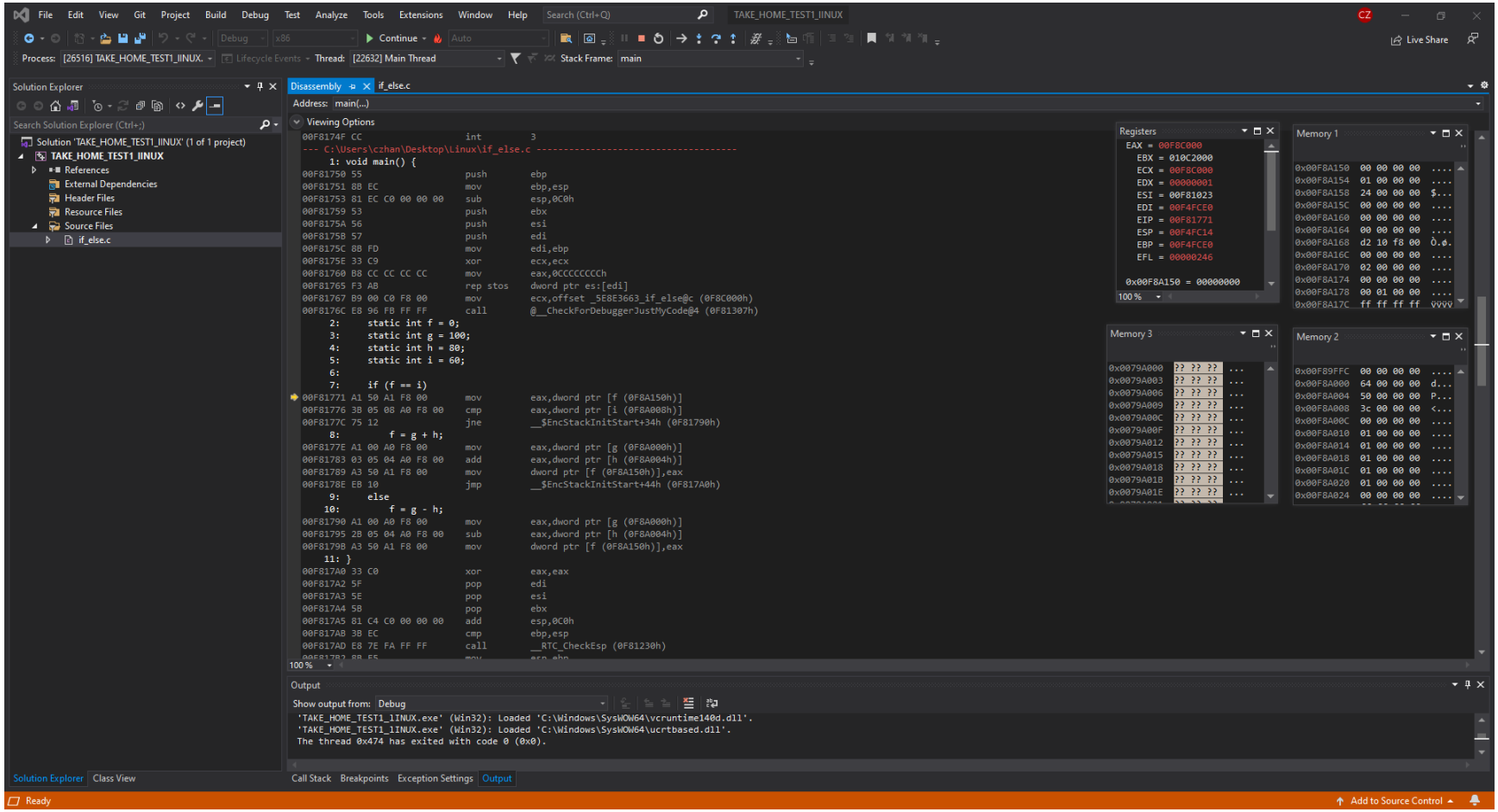
In this image, the instructions that were done were loading the values of a,b c and d into $s0, $s1, $s2 and $s3. If we look at the register window to the right, we can see that those registers now have hex values 0, 64, 50, 3c. We can also look at the data segment and see the offset values of +0, +4, +8 and +c to confirm this. Lastly we look at $sp and we can see that it is not changing because we are not working with local variables, only static variables.



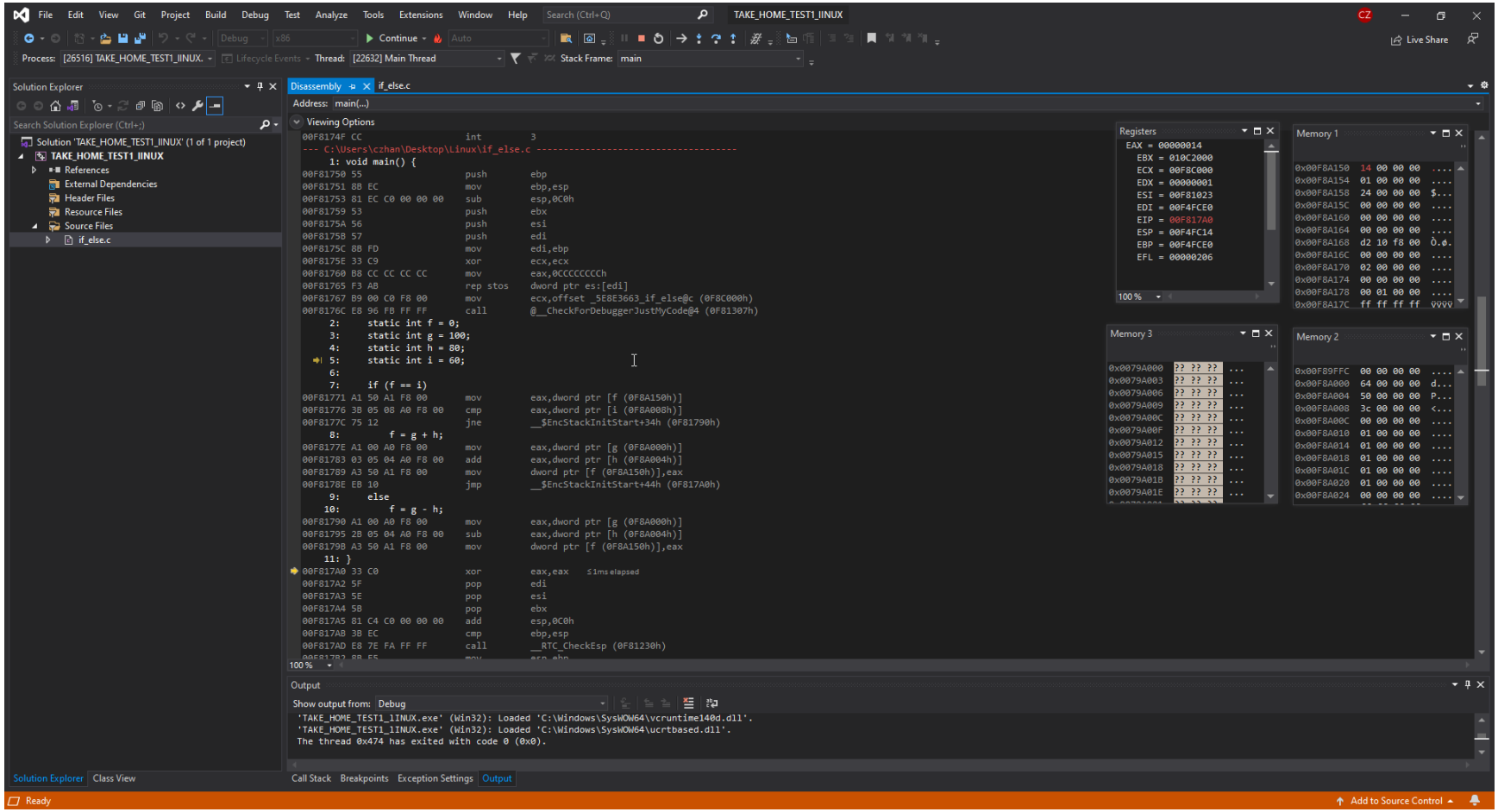
Lastly, in this figure, what was done is instruction 0x00400020 checks to see if $s0 = $ s3 or if value a = d and in this case, it is not. A has a value of 0 and d has a value of 3c so we move to the instruction 0x00400002c where we subtract $s1 and $s2 and store it in $s0. We can look at the register window on the right to confirm that s0 is indeed $s1 - $s2 as $s1 - $s2 is 50 – 3c in which 14 and we can see that value in $s0.



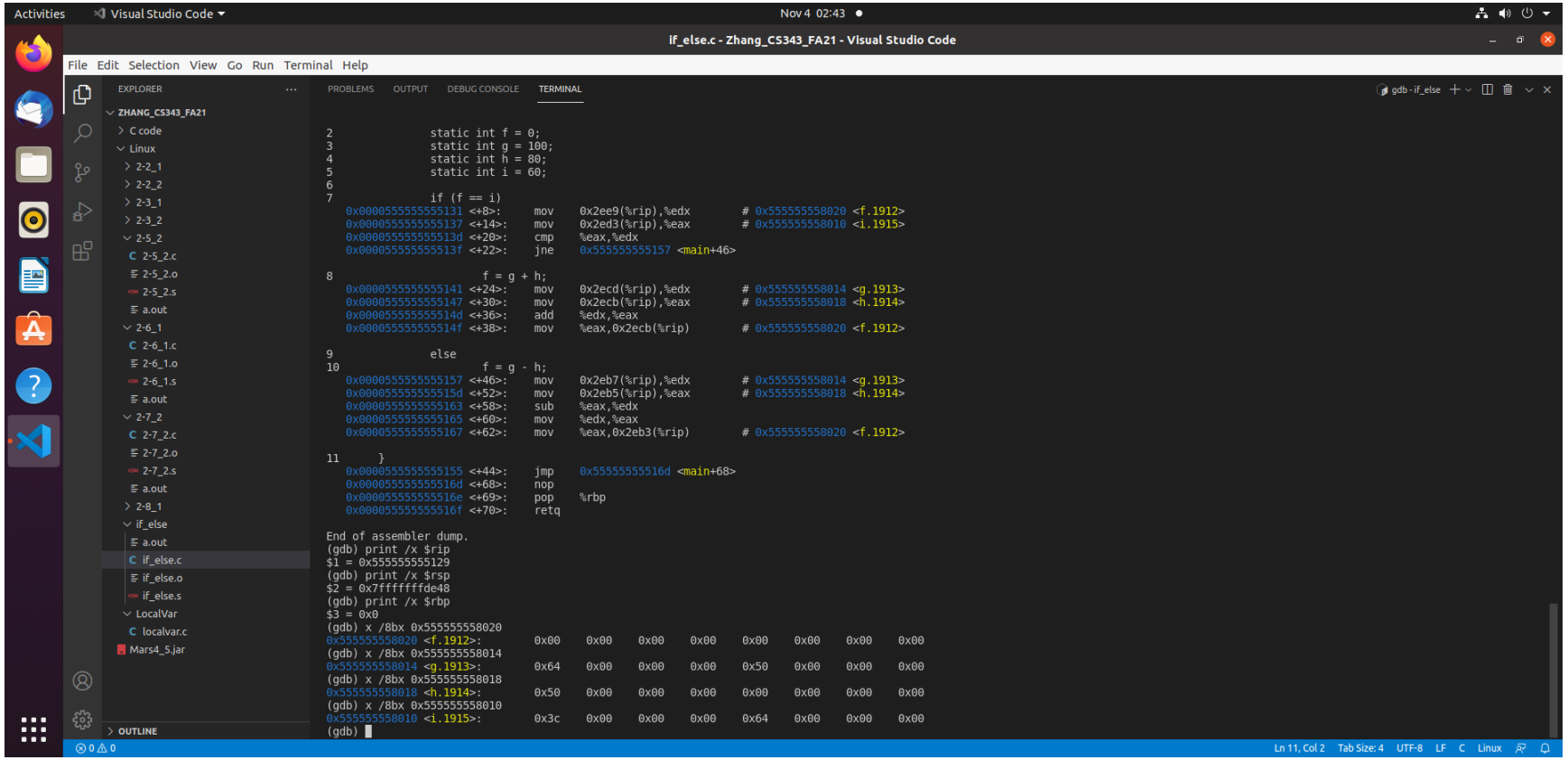
In the figure above, we are shown the basic disassembly page, register window and memory windows given when compiling using an intelx86 processor. In this problem ,we will be observing the if-else statements and how it works



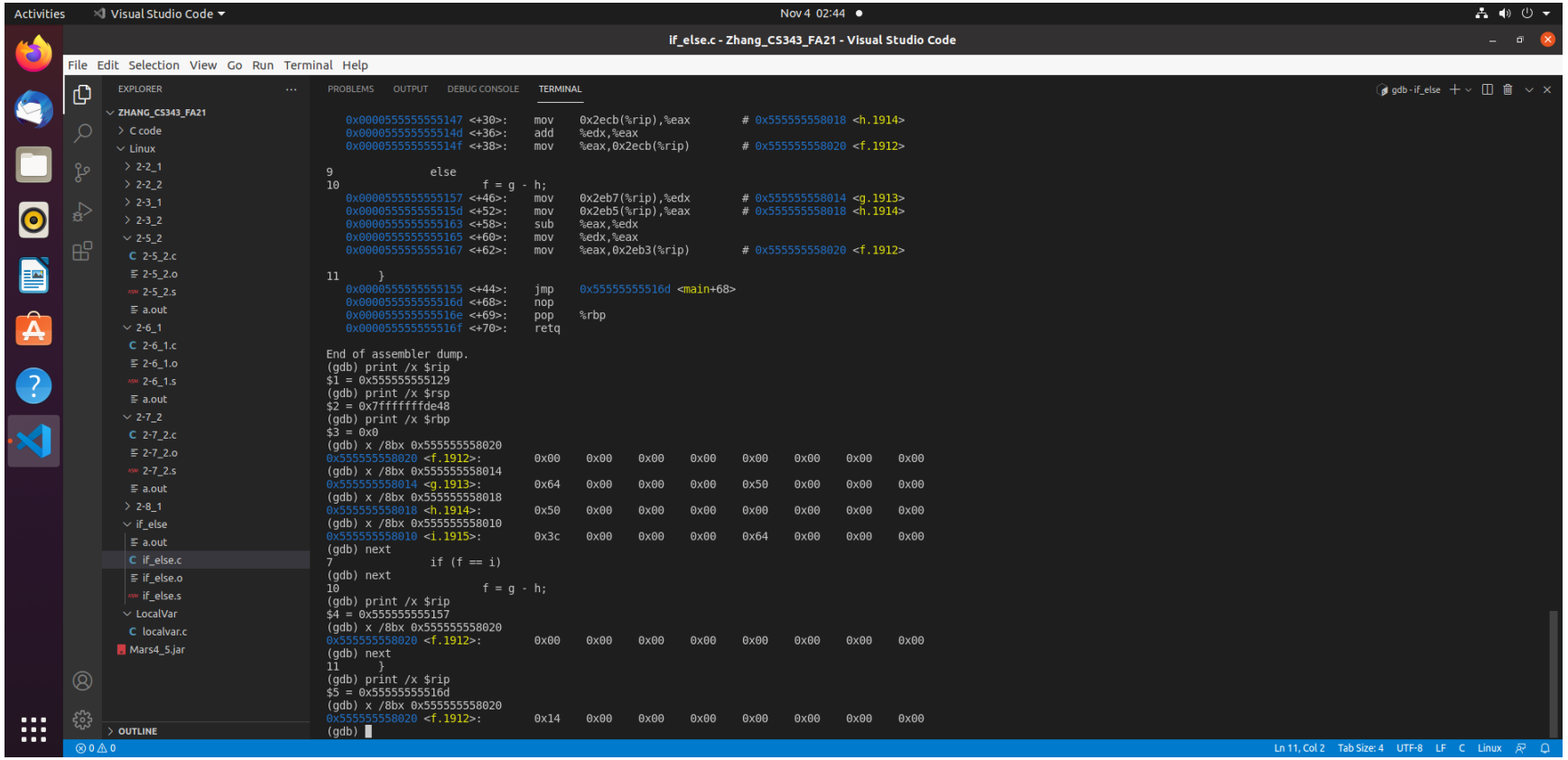
In this figure, we can see that in memory window 1, the address of f is 0x00F8A150 and it has a value of 0 which is correct. If we look over at memory window 2, we can see the address of g, h and I as well as their corresponding values of 64, 50 and 3c (Respective).



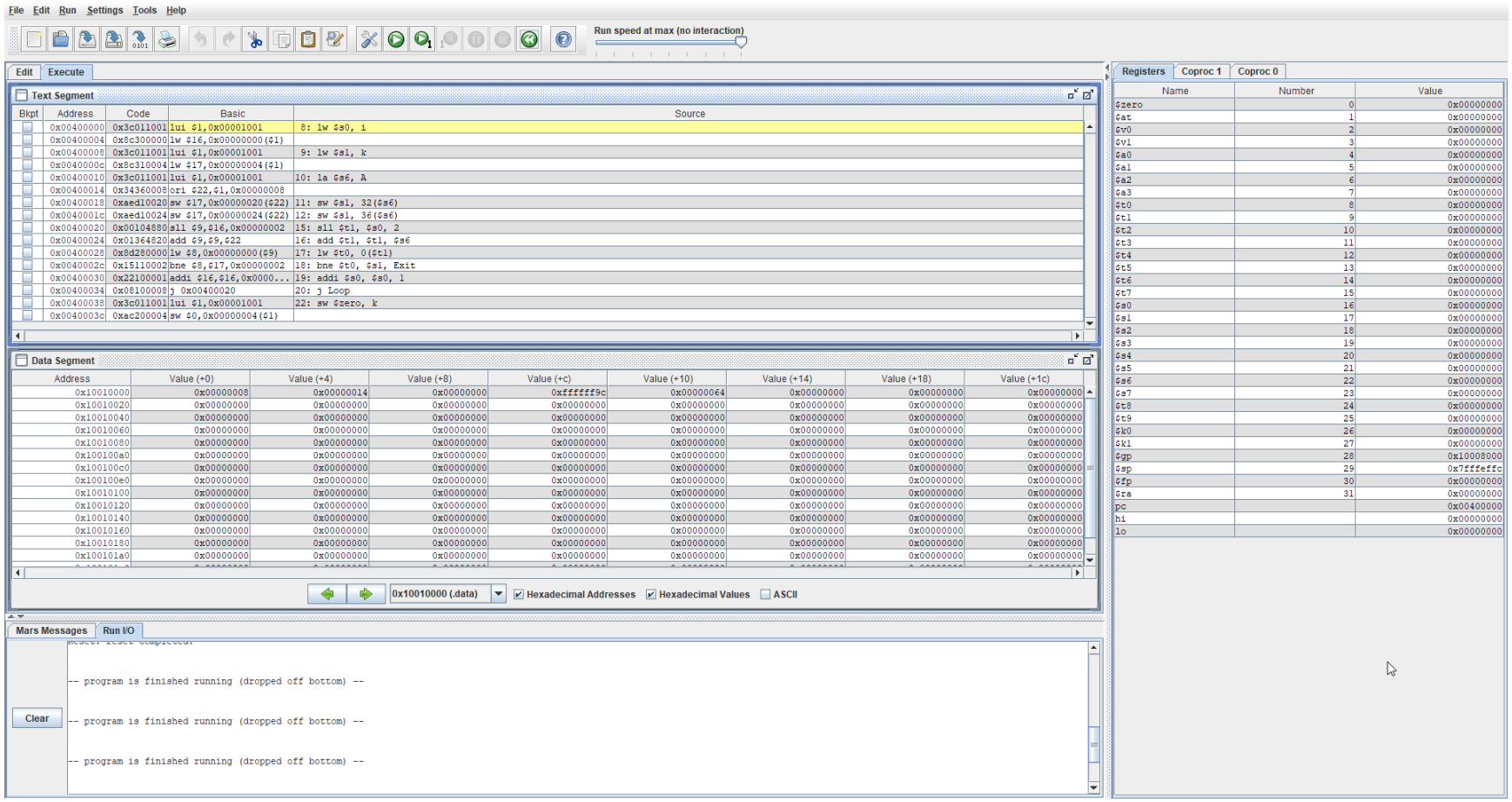
In this figure, what has happened is that the statement if(f == 1) was not fulfilled, therefore, all instructions up until the else statement are skipped and the next instruction that is performed is f = g-h . g – h is 64-50 therefore f= 4 and if we look at memory window 1, we can see that f now has a value of 14 stored in it. Furthermore, we can look at ESP and see that it has not changed since the 2nd screenshot of the intelx86 compilation and that is because we are only working with static variables, not local variables.



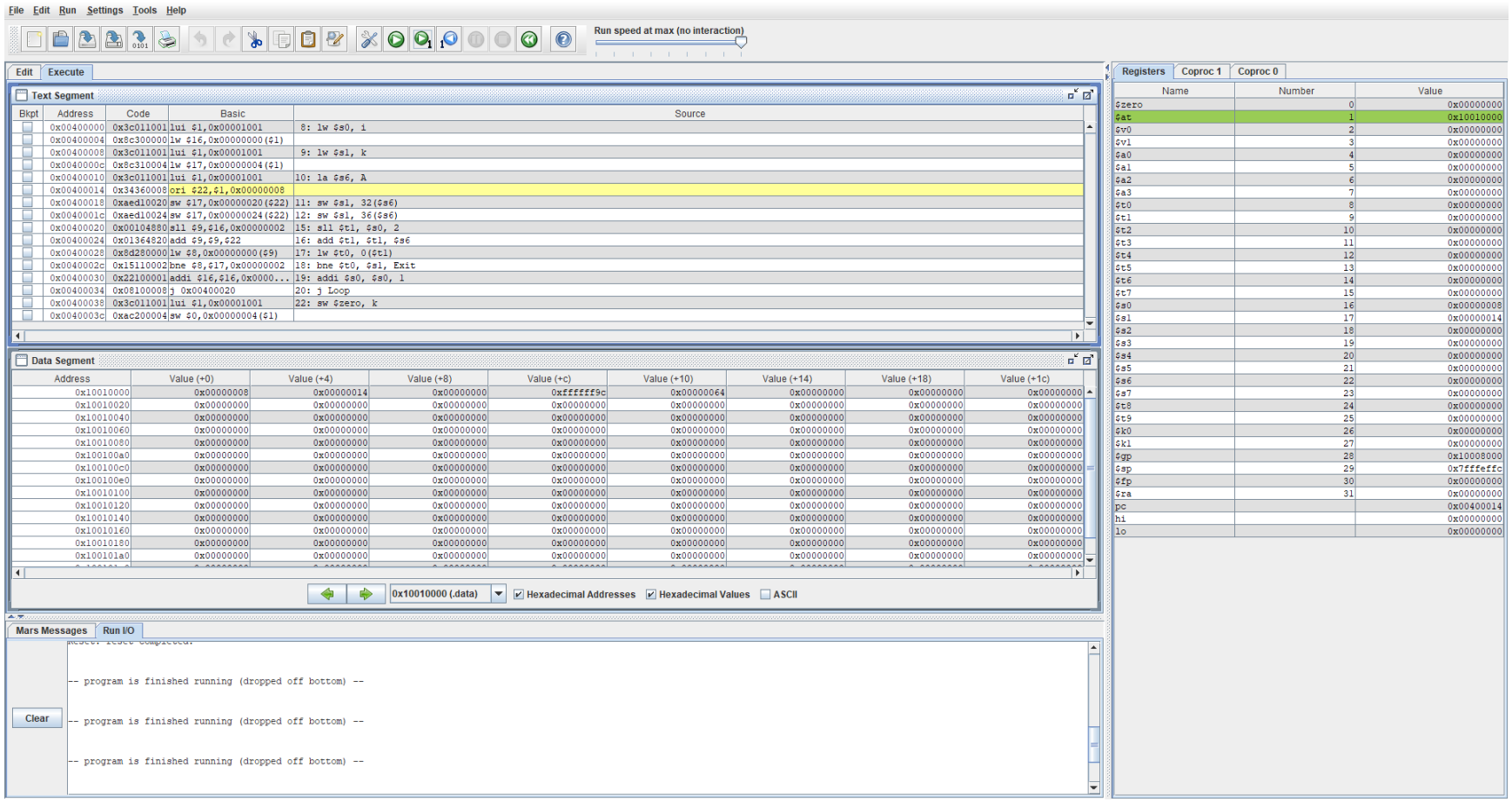
In the figure above, what is showcased is question 2-7_1 being compiled with the GCC windows compiler and looked into using the gdb command. Here we will observe how the if else statement is being conducted. First, we look at the instruction addresses that is the dump of the code and below the dump are the current values stored in f,g,h and i. What's also printed is $rip, $rsp and $rbp which are the instruction pointer, stack pointer and base pointer respectively. $rsp will not change and $rbp will have the same value as $rbp because we are not working with any local variables, only static variables.



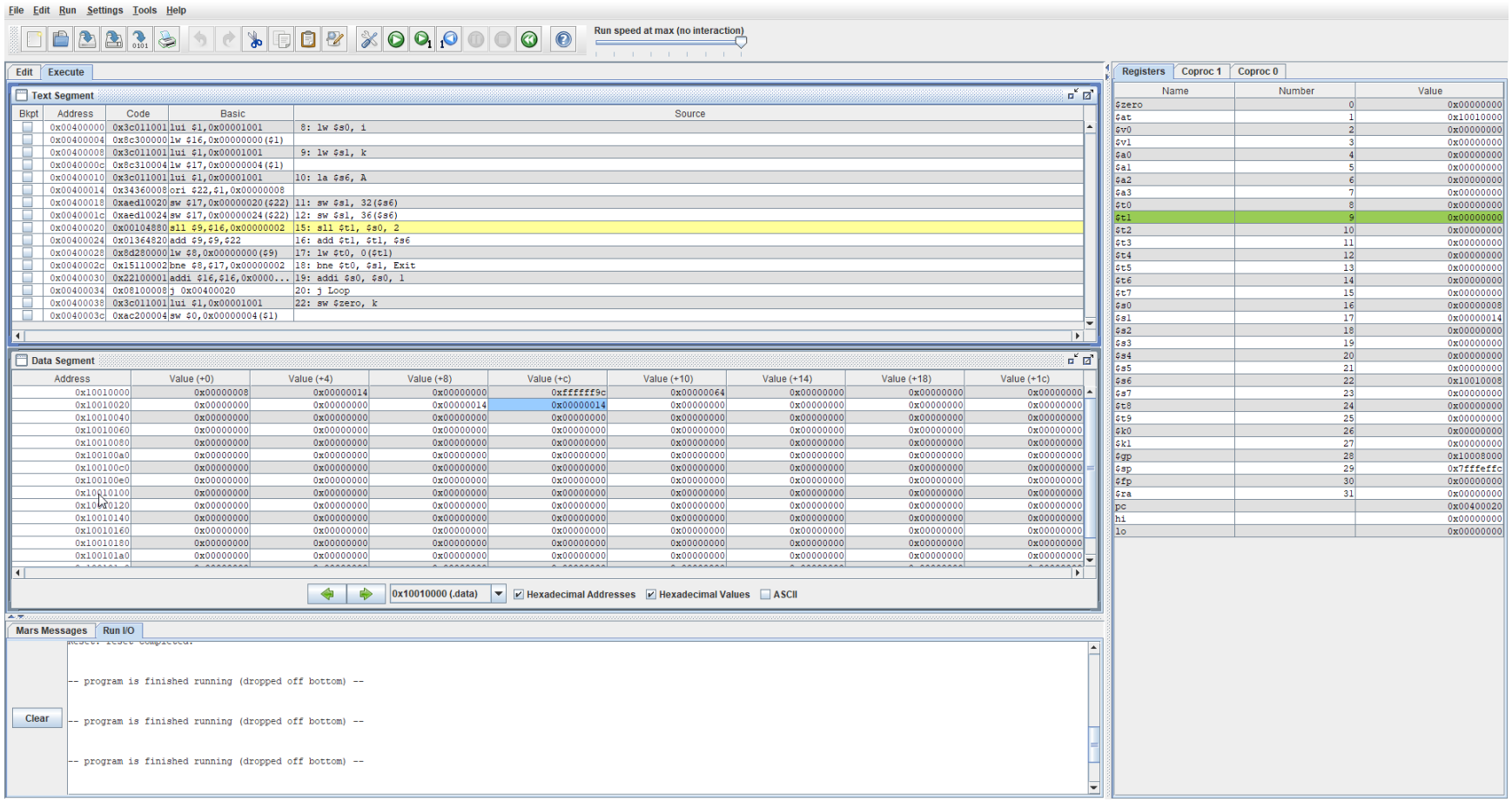
In the figure above, what is shown is how the if-else statement functioned. What I did above is continued through the if(f == 1) statement to see what would happen. What occurred was it skipped instructions 8 and 9 and went straight to 10 since it could not fulfill the if statements condition, then subtracted g and h to get a value of 14 that is stored in f. We can also notice by looking at the $rip and looking up the addresses in the dump file to confirm instructions being skipped.



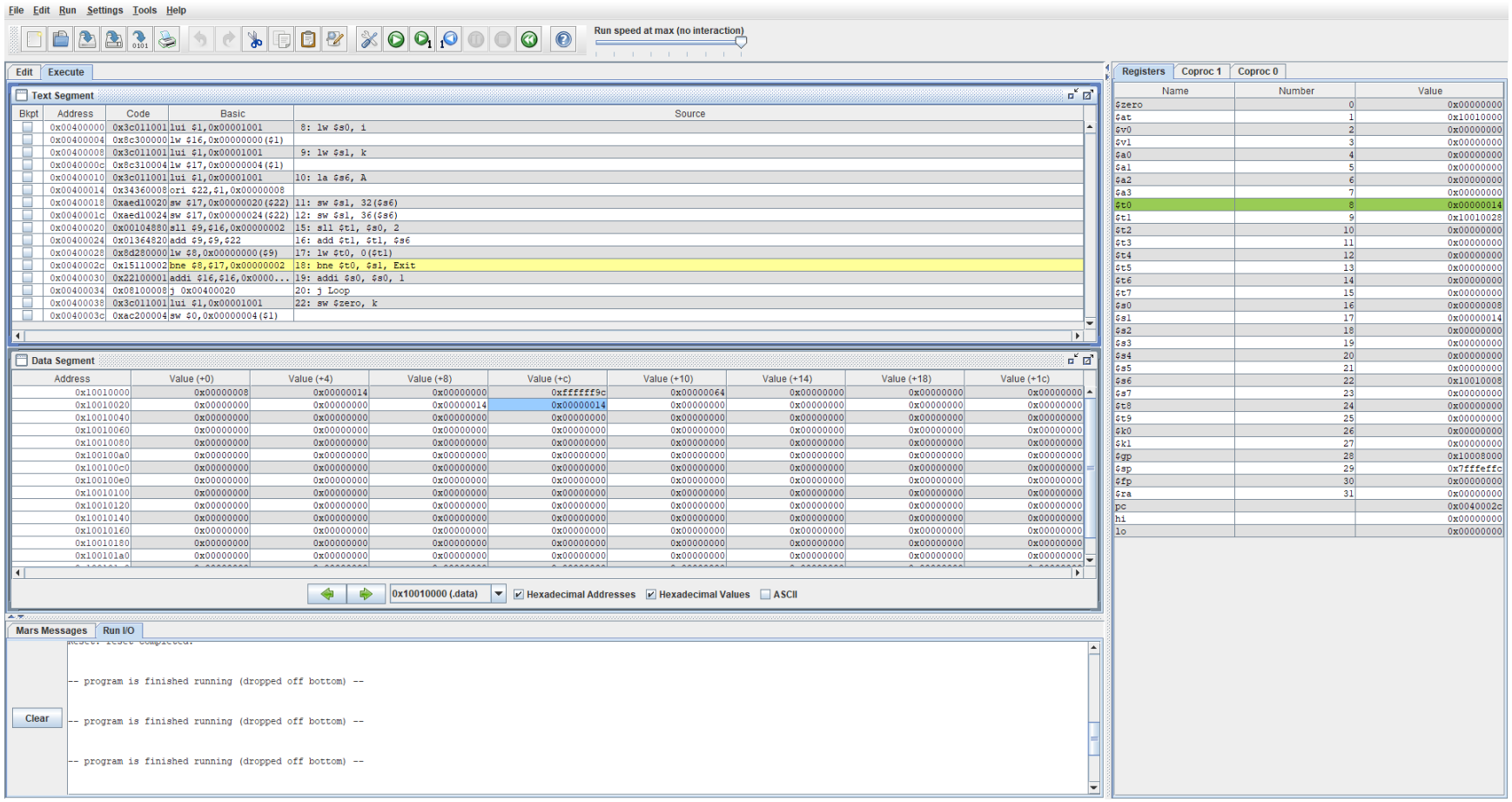
In the imagine above, I compiled 2-7_2 using the MARS simulator. $pc tells us what instruction is next and $sp tells us what the stack pointer is but it doesn't change because we are only using static variables and not any local variables. In this program, what is to be simulated is a while loop that checks if the array A[8] and A[9] are = 20 and if so, then we increment the index values by 1 and continue comparing.



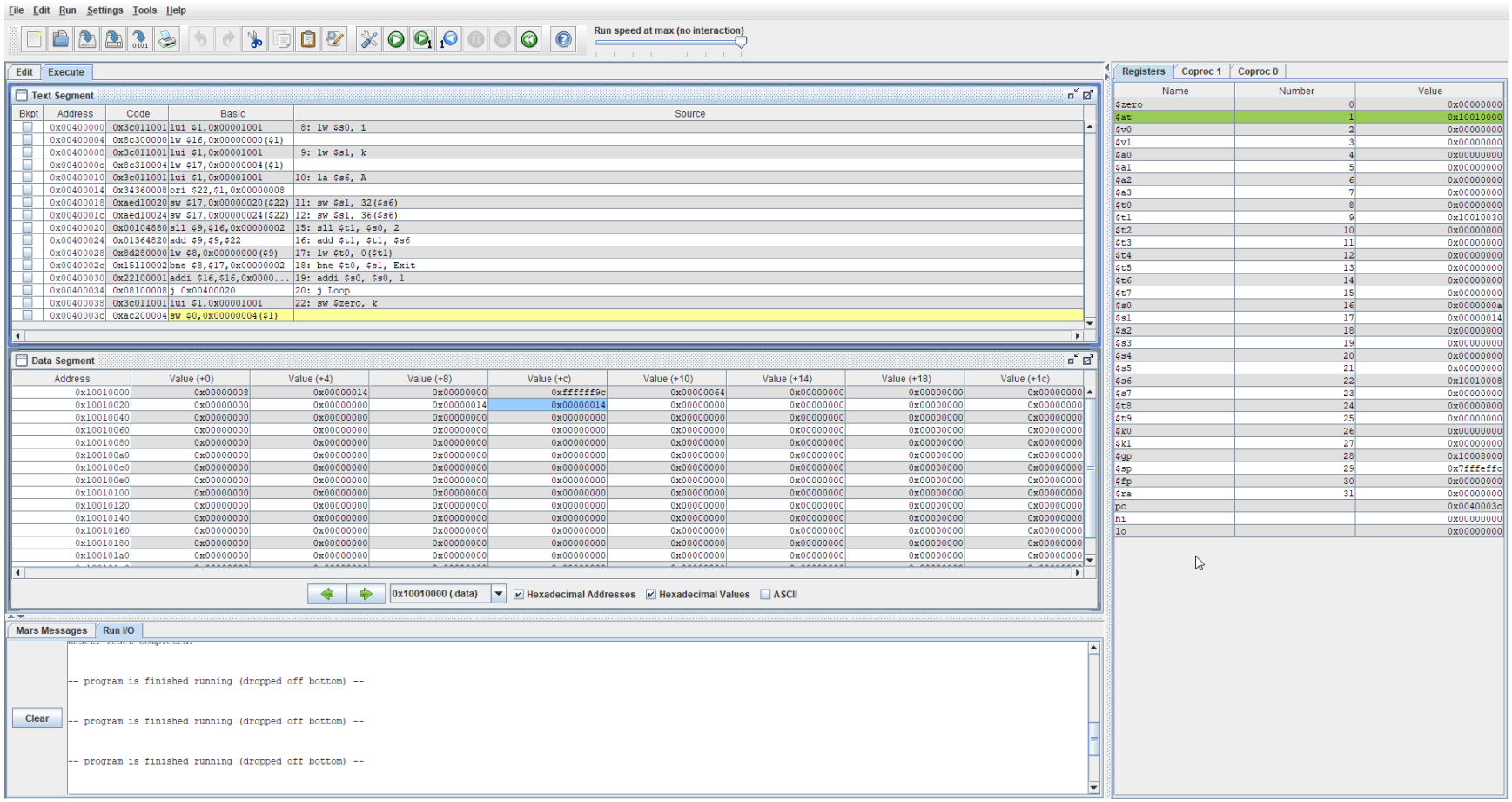
In this figure above, what was done is words of I was loaded into $s0 and k was loaded into $s1 and the address of the array was loaded into $s6. You can observe this in the register window as $s0 now has a value of 8, $s1 now has a value of 14 and $s6 has an address 10010008. You can also observe these values that were being loaded in from the data segment.



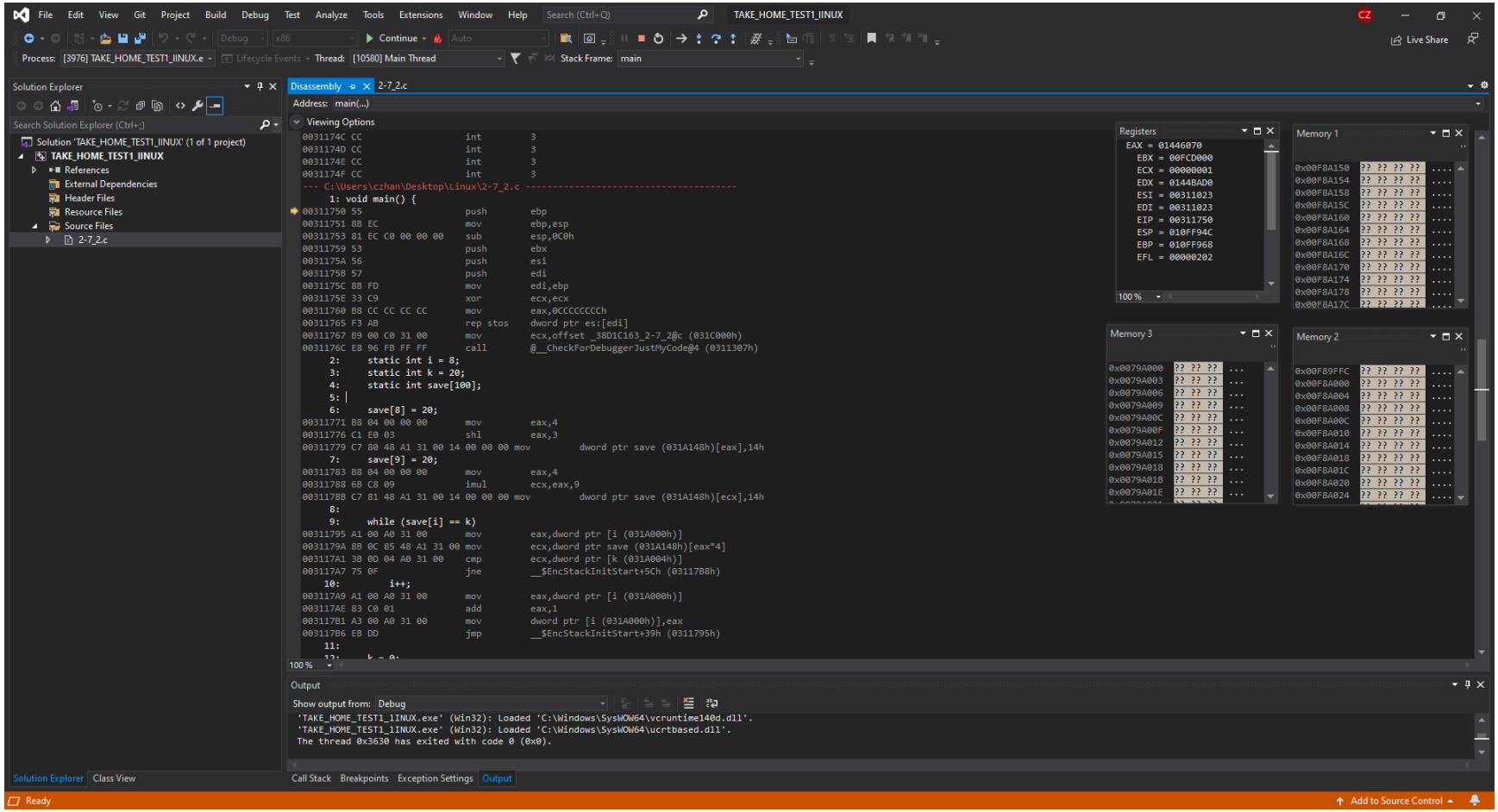
In this figure above, I proceeded with several extra instructions where I allocated values $s0 into the 8th and 9th index of array A. As we can see in the text segment, the instructions that were done are sw $s1, 32($s6) and sw $ s1, 36($s7). This can be confirmed in the data segment window at the offset +32 and +36.



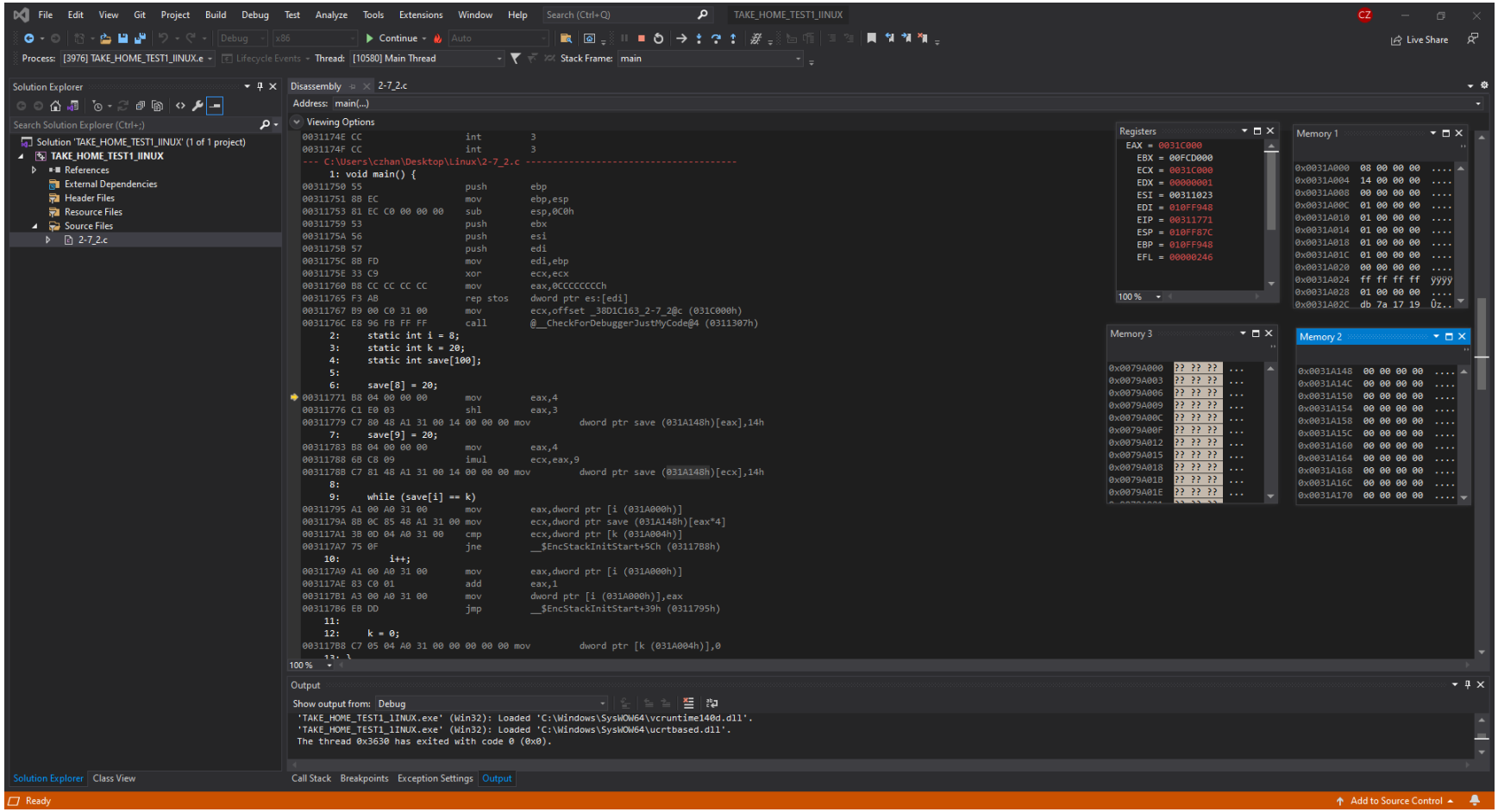
In the figure above, we add $s0 with 1 and continue a loop if $t0 and $s1 are equal to each other. In this case, they are so we continue the loop and we can observe these values in the register window. $t0 now has the value 14 and is the value of the 8th index of the array A. $s1 is the variable that stored the value 14 inside it so we use it continuously for comparison to check the while loop condition.



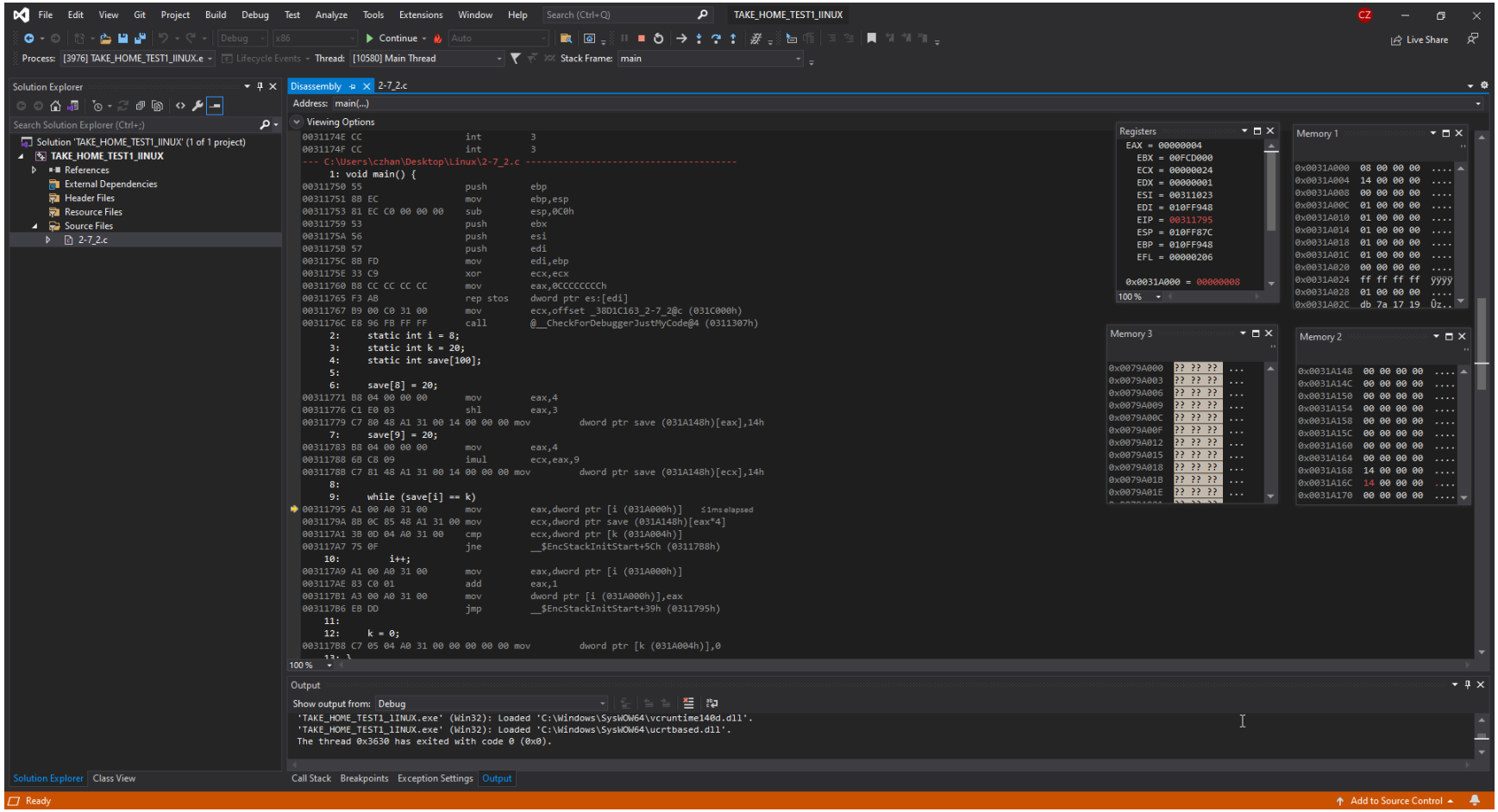
In this figure above, it shows that the loop has been completed and if we look over at $s0, the value is now “a” whereas, the value of $s0 was previously 8. This shows that this programmed looped twice to mee the exit conditions of the while loop.



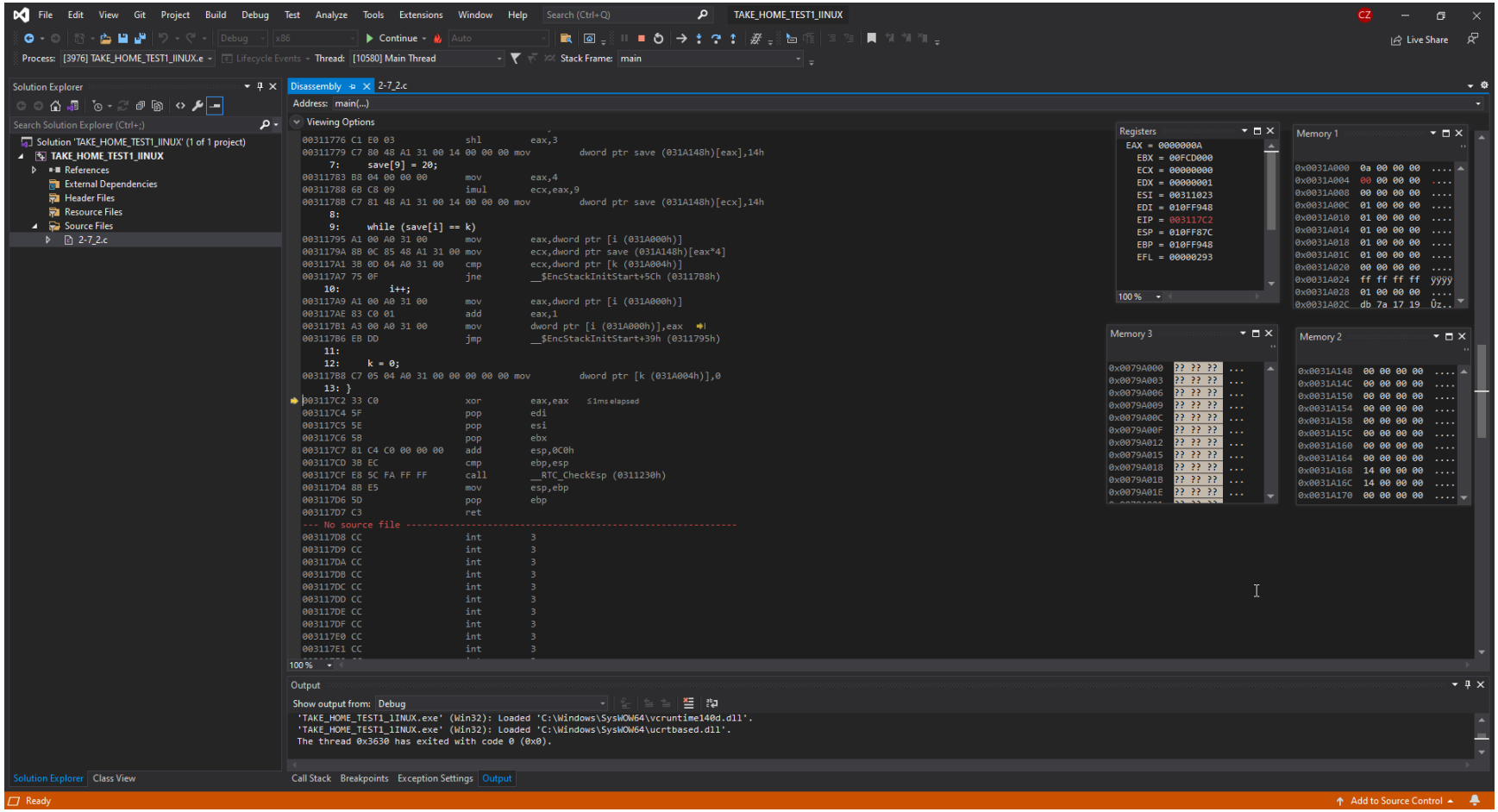
In the figure above, we have compiled question 2-7_2 using the intelx86 processor and can now observe the disassembly window, the registers and memory windows to compare with other compilation methods. Since we are working with only static variables, ESP will not be changing. For each figure below this one, we can look at the EIP and compare it to the ones we see in the disassembly window to confirm correctness



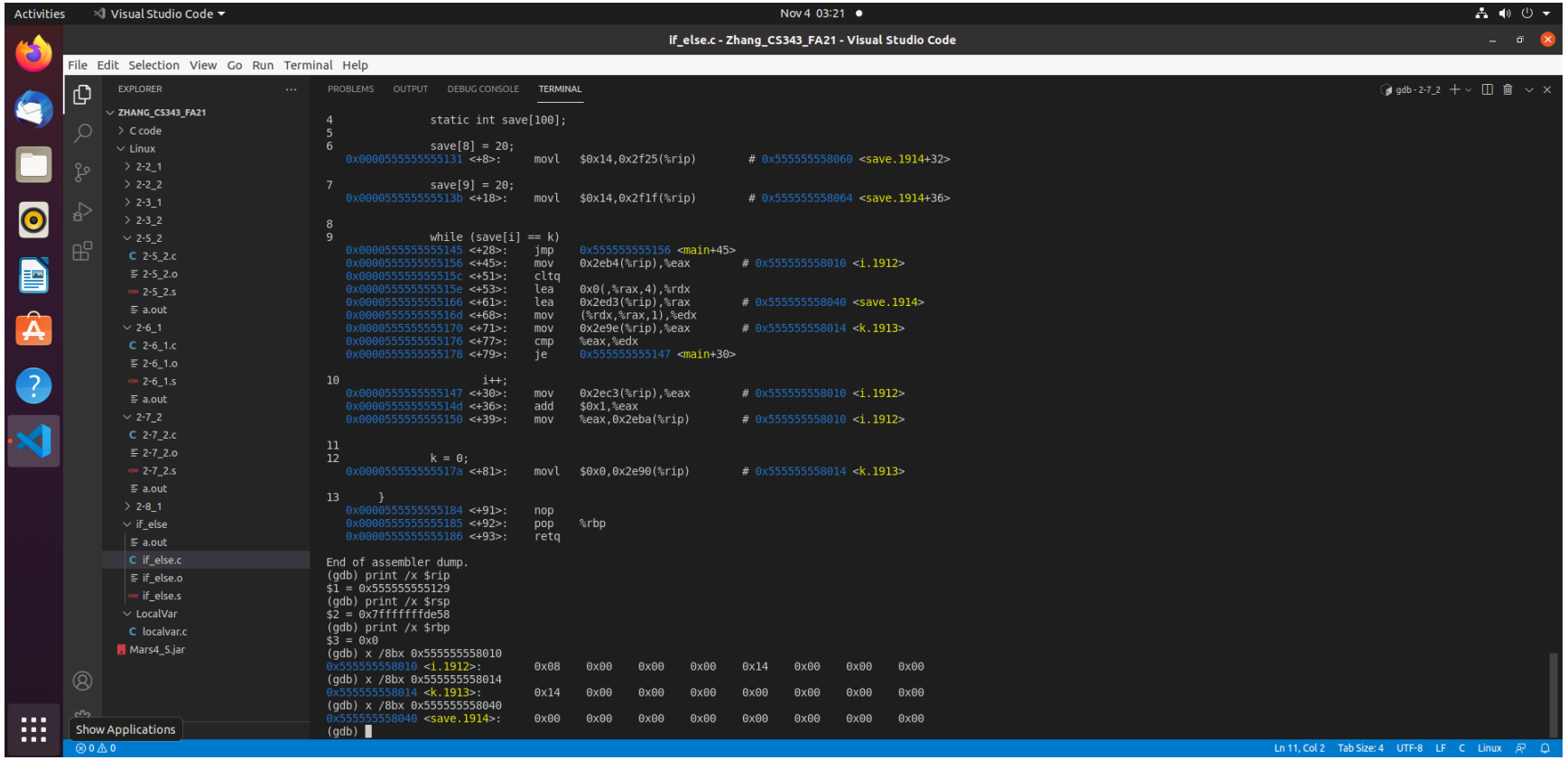
In the figure above, we have loaded the values of I, K and array save. If you look over to memory window 1, we can see the values of I and K being stored at the top most 2 addresses as 08 and 14, respectively. Furthermore, in memory window 2, we have loaded the array save but currently it has no value saved in it therefore, we only have zeroes.



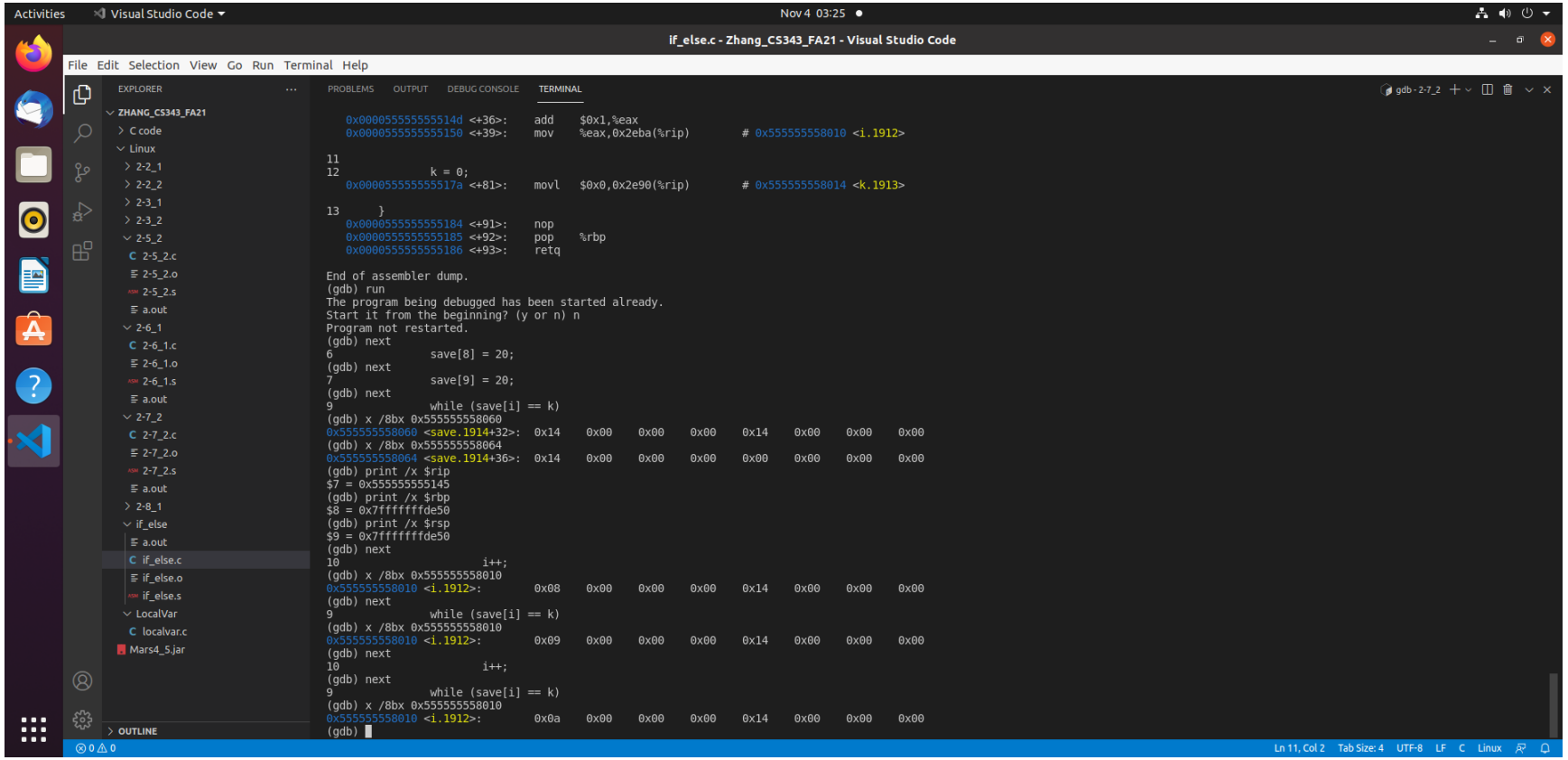
In the figure above, the next steps that were taken were to allocate the value 20 into the 8th and 9th index of the array save. We can observe that this is true in memory window 2 at the addresses 0x0031A168 and 0X0031A16C as they now have the hex value of 20.



In this figure, we can observe that the while loop has occurred twice as if we look over in the memory window 1, what is stored in the address 0x0031A000 is now 0a instead of 08 that it previously had. That is because we increment I by 1 every time the exit condition of the while loop is not met therefore 8 + 1 + 1 is 10 which is “a” in hex.

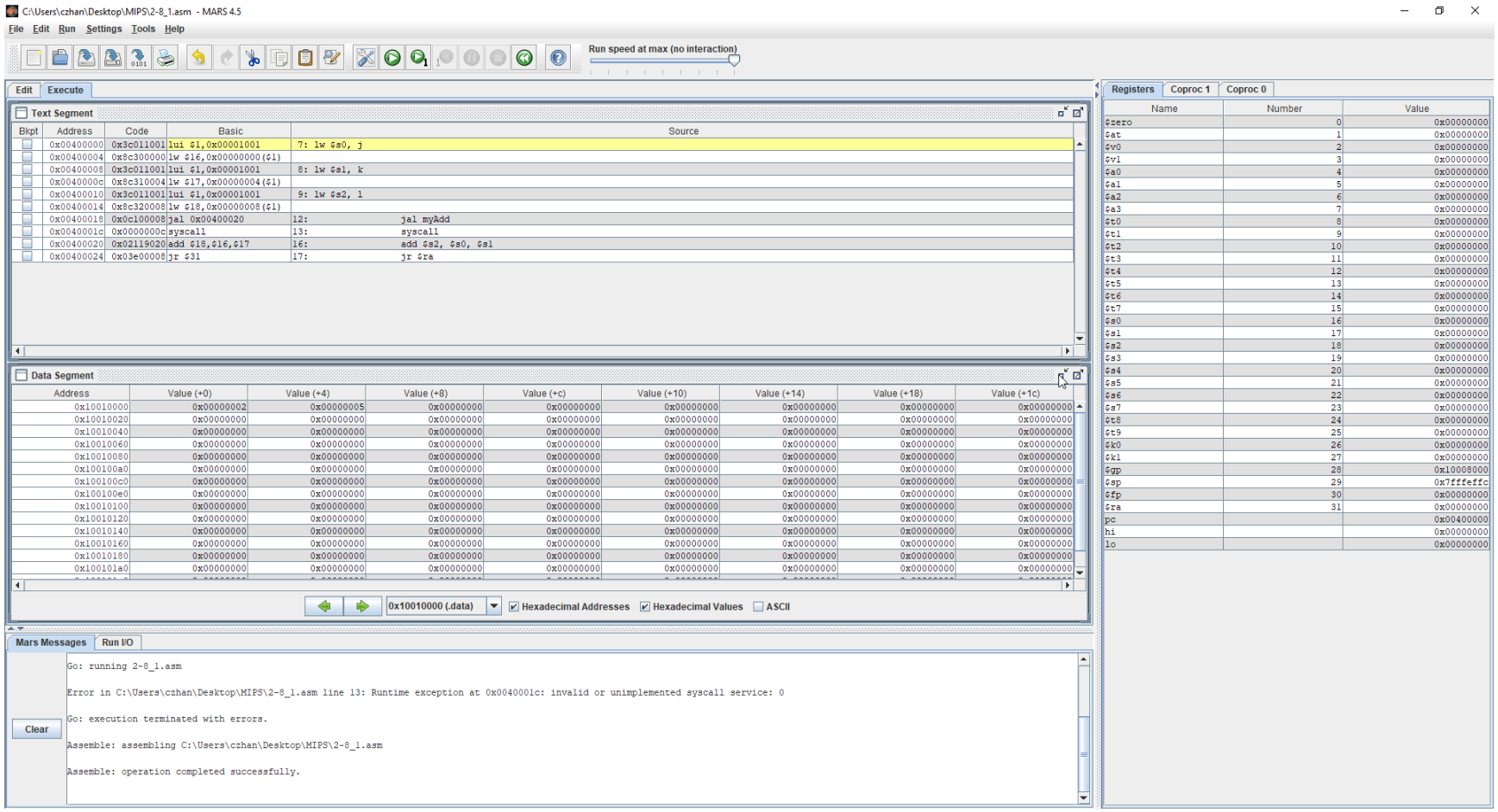


In the figure above, we have compiled question 2-7_2 using the gcc windows compiler. What I am showing in the terminal tab is the dump file of the code as well as the $rip, $rsp, $rbp which is the instruction pointer which points towards the next instruction, the stack pointer which is going to be the same since we aren't using local variables and the base pointer which points towards the top of the stack which is basically going to be the same as the stack pointer. I have also printed out the values stored in I, k and the save array. I has the value of 08, k has the value 14 and save has all of its indexes as 0.

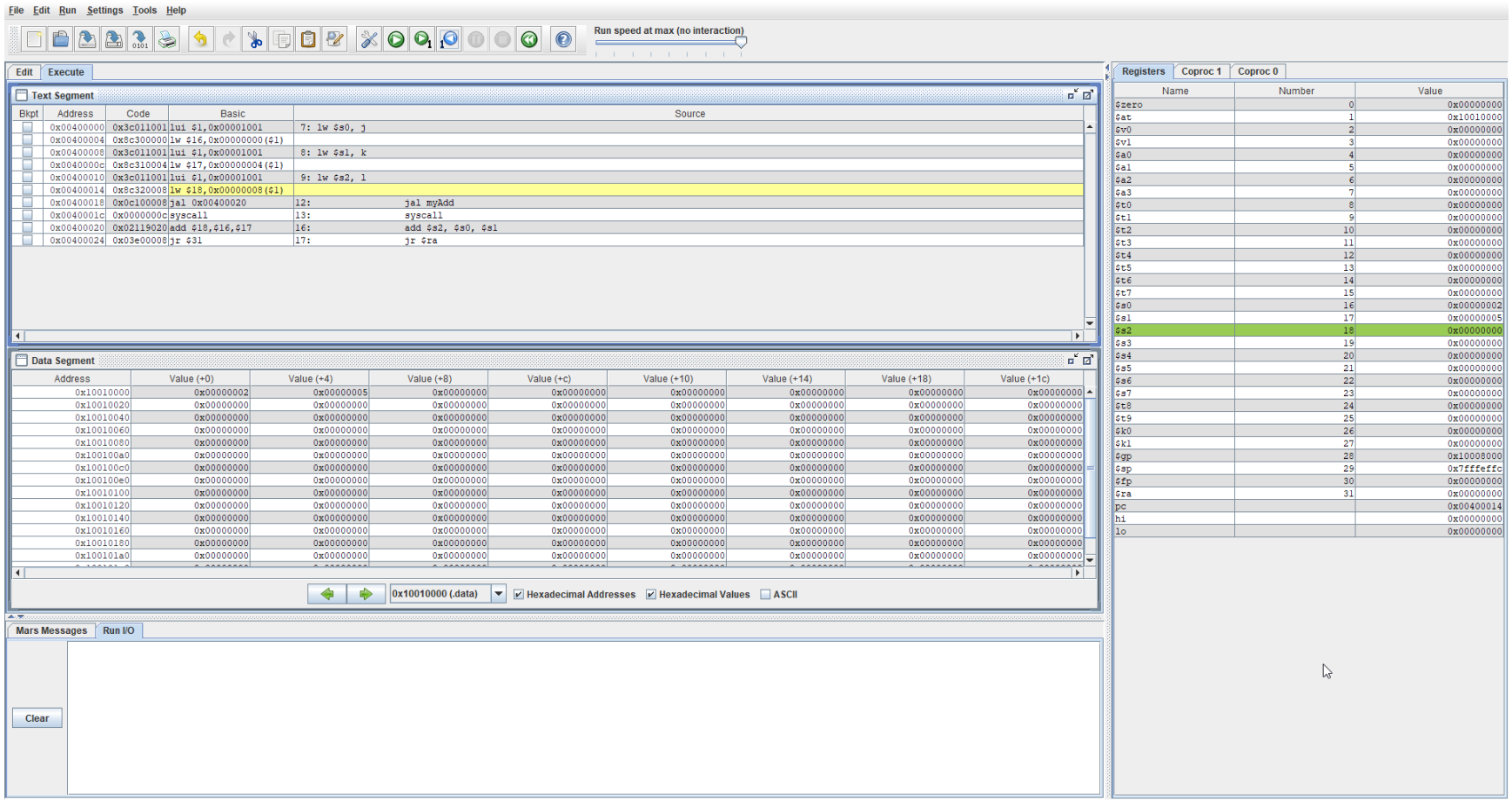


In the figure above, I have performed the entire program and we start by looking at instruction 6 and 7 which is save[8] = 20 and save[9] = 20. After those instructions were done, I printed out the array as you can see under the line “while (save[i] == k)”. In the +32 offset array printout, we can +32 has the value 0x14 and 4 more to the right is +36 which has a value of 0x14 as well. I also printed out +36 offset array as well which showcases that +36 has the value 0x14.

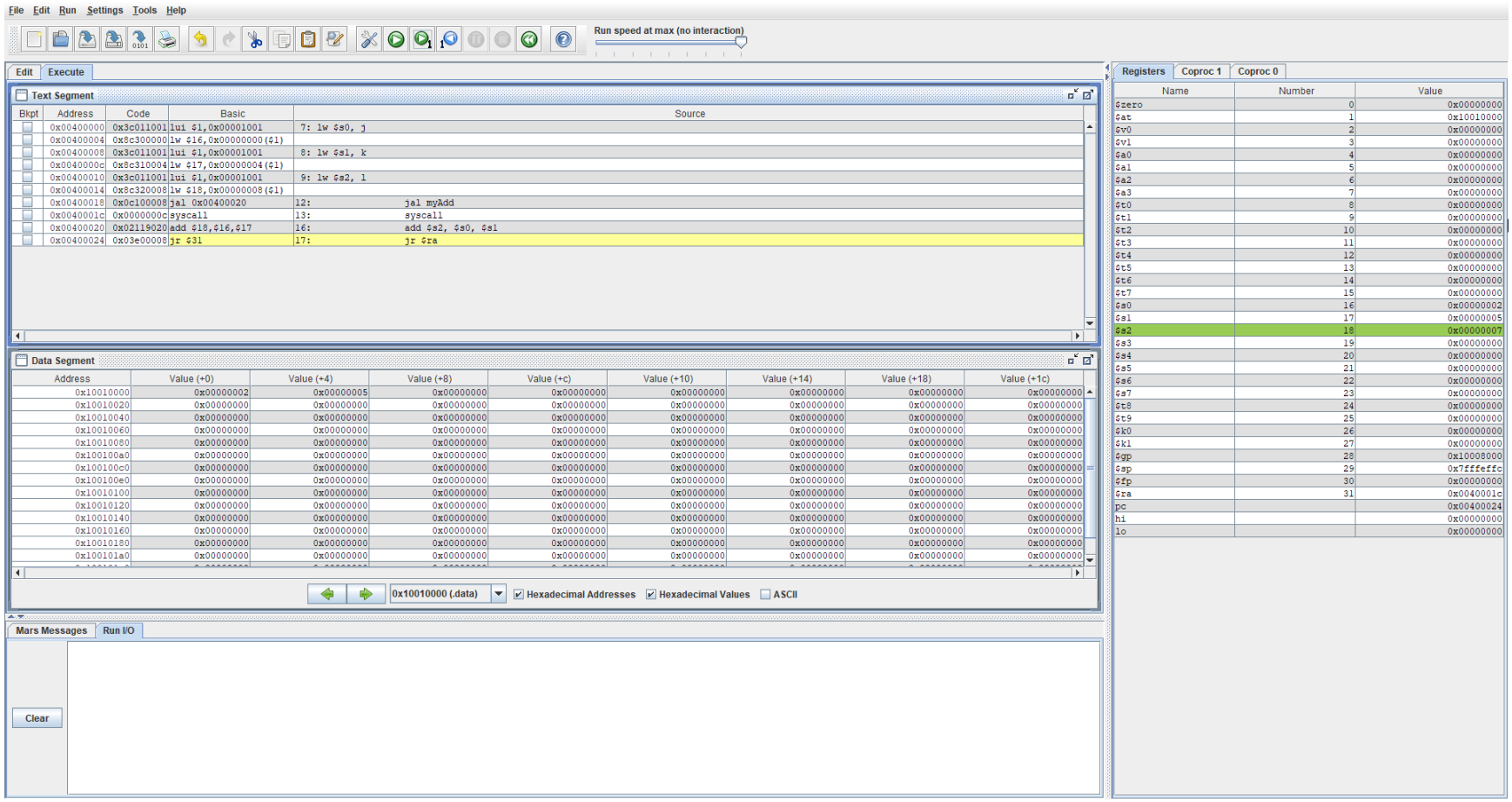
Next, what was performed is the while loop where integer I would be incremented until exit conditions are met. In this case, exit conditions aren’t met for 2 loop cycles therefore integer I was incremented twice and holds the value of “a” in bottom most line in the figure above. This shows that the while loop was successful and when comparing it to the other compiler methods, there is almost no differences between them.



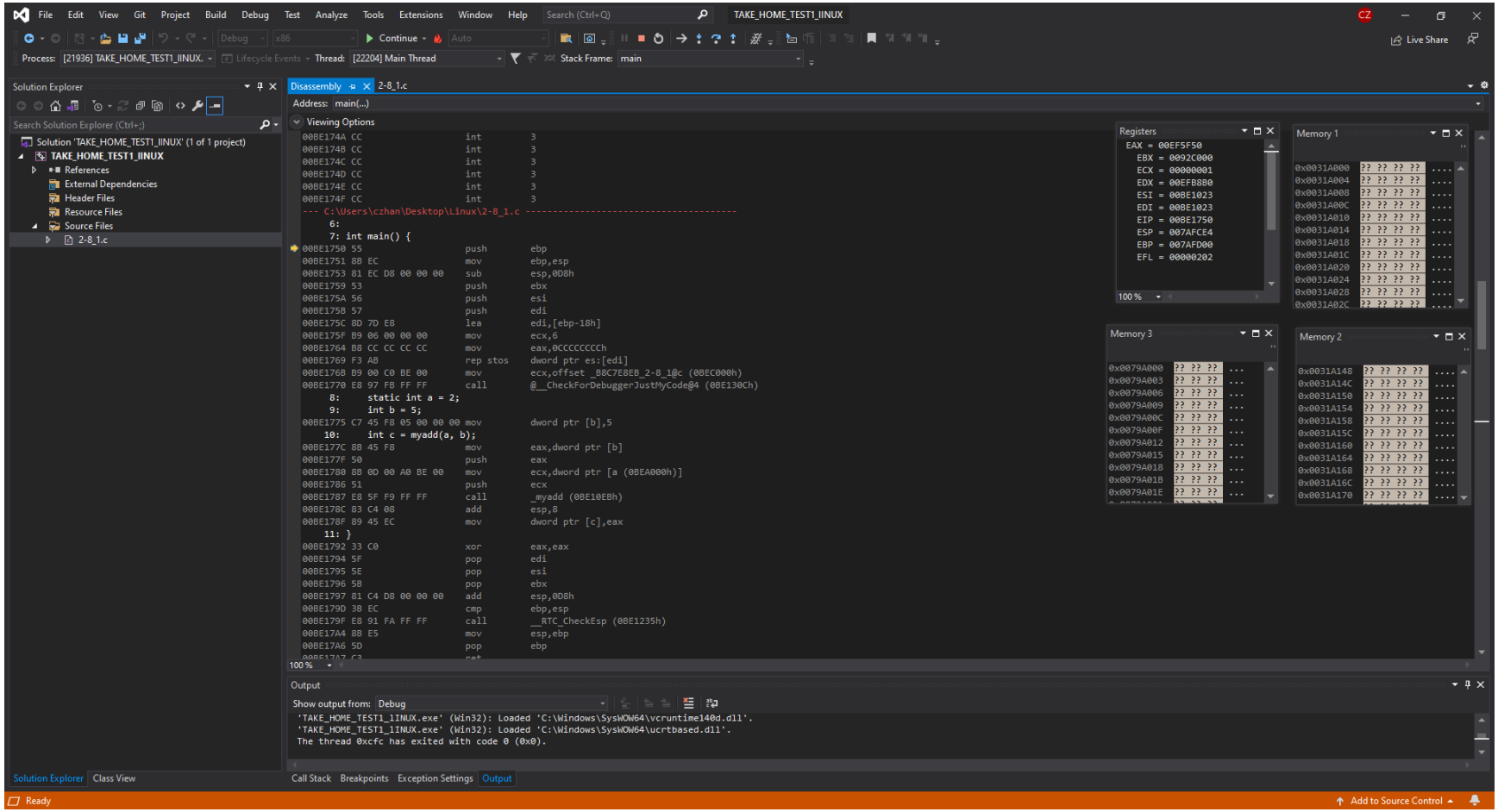
In the figure above, I am showing how the myAdd function works, we have $pc which tells us what the next instruction will be and in the MIPS file, we are using static variables that will be inputted into our myAdd function. The functionality that should be performed is 2 + 5 which should equal to 7. Furthermore, since we are only working with static variables, $sp is not subject to any changes. The reason for static variable in question 2-8_1 I was unable to figure out how to get local variables.



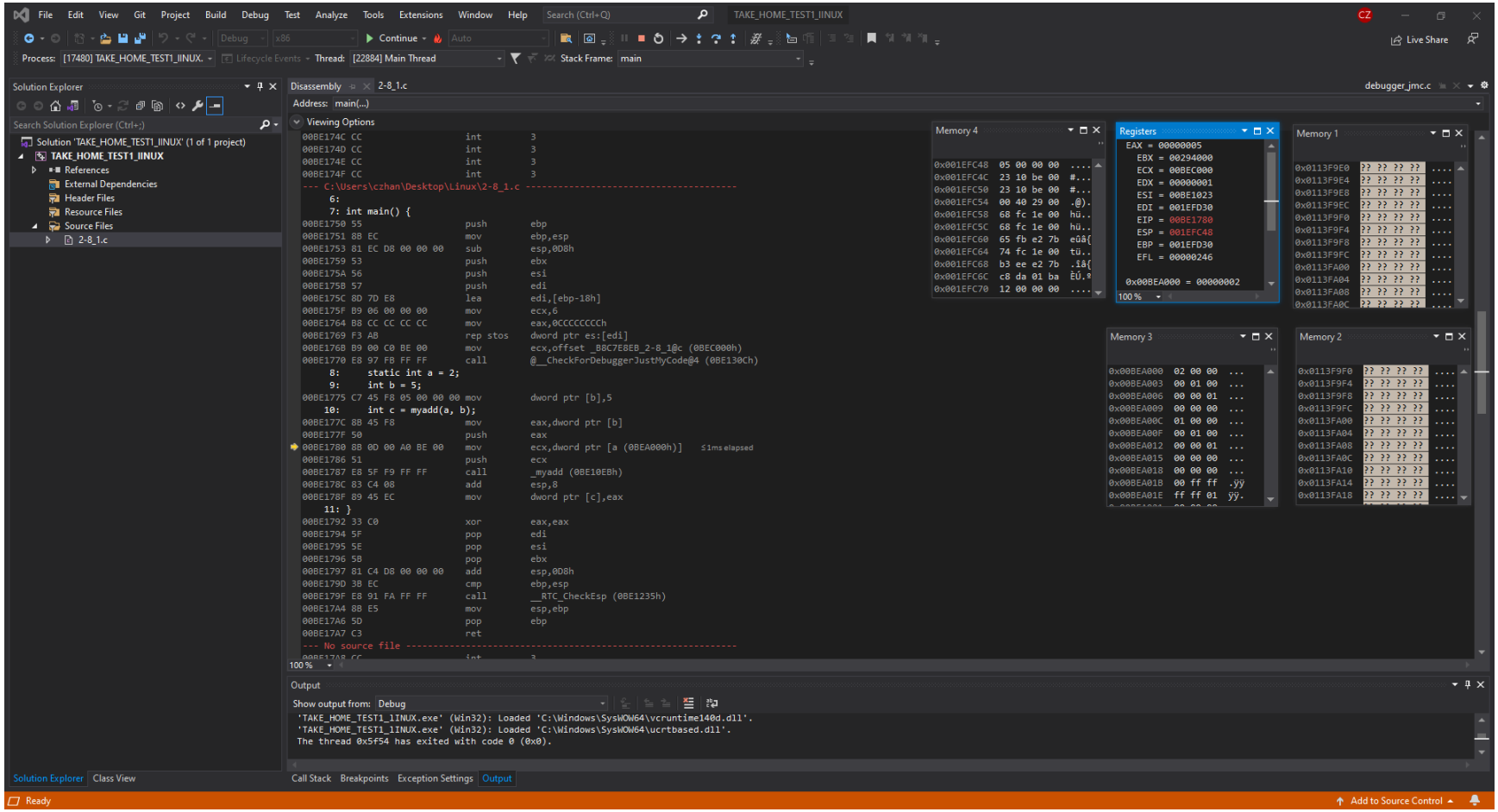
In the figure above, several instructions were run where I loaded in registers $s0, $s1 and $s2 with values stored in j k and l. You can confirm that these values are in the register if you look at the register window. We can now see that the values 2,5 and 0 are now written into those registers.



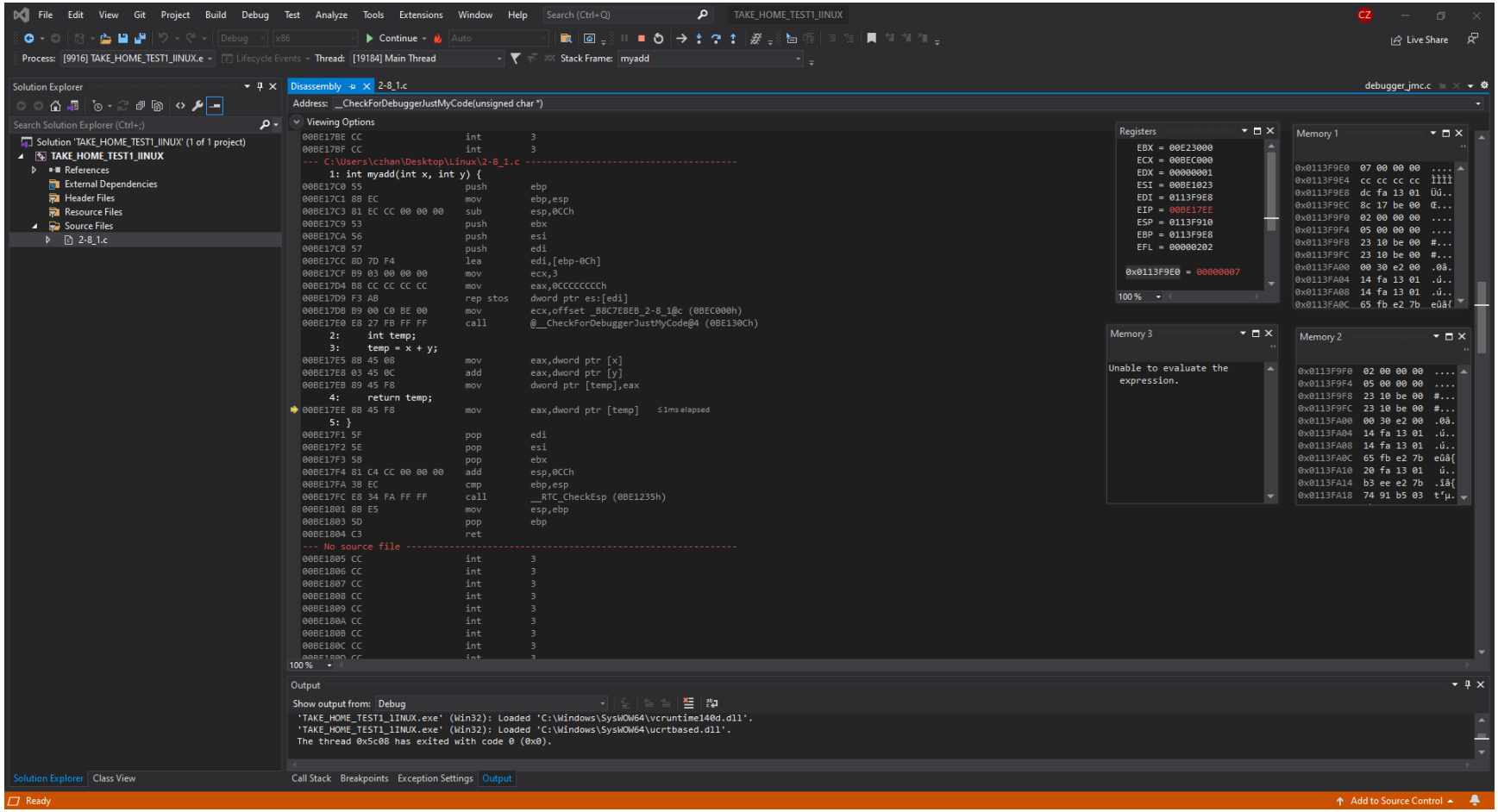
Finally, in this last figure, the myAdd function was called where it performs $s2 = $s1 + $s2 and if we look at the instruction register, we can see that $s2 now has the value 7 written into it.



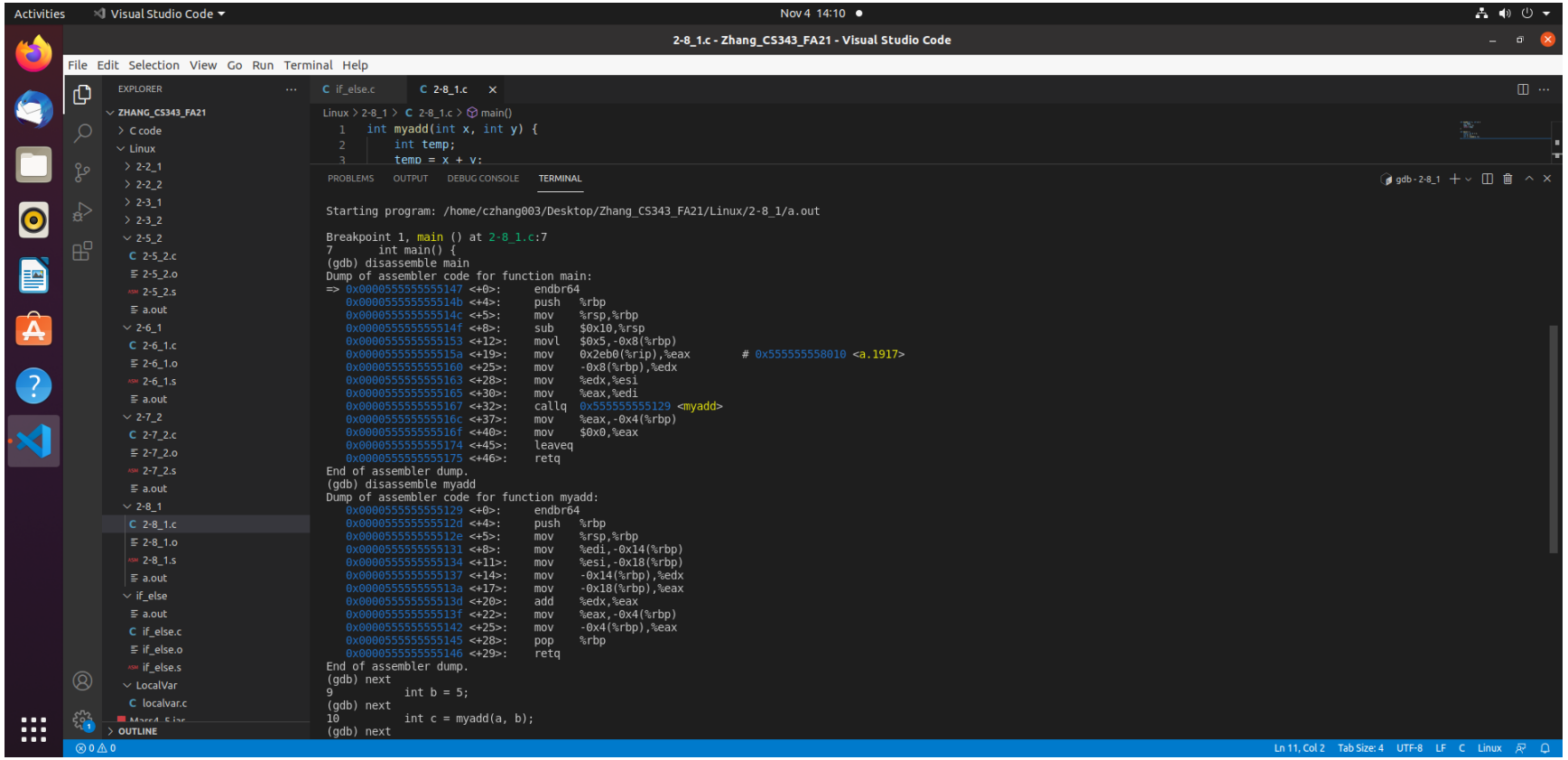
In the figure above, I have compiled the question 2-8_1 using an intelx86 processor and what is provided is the register window, disassembly file and the memory windows.



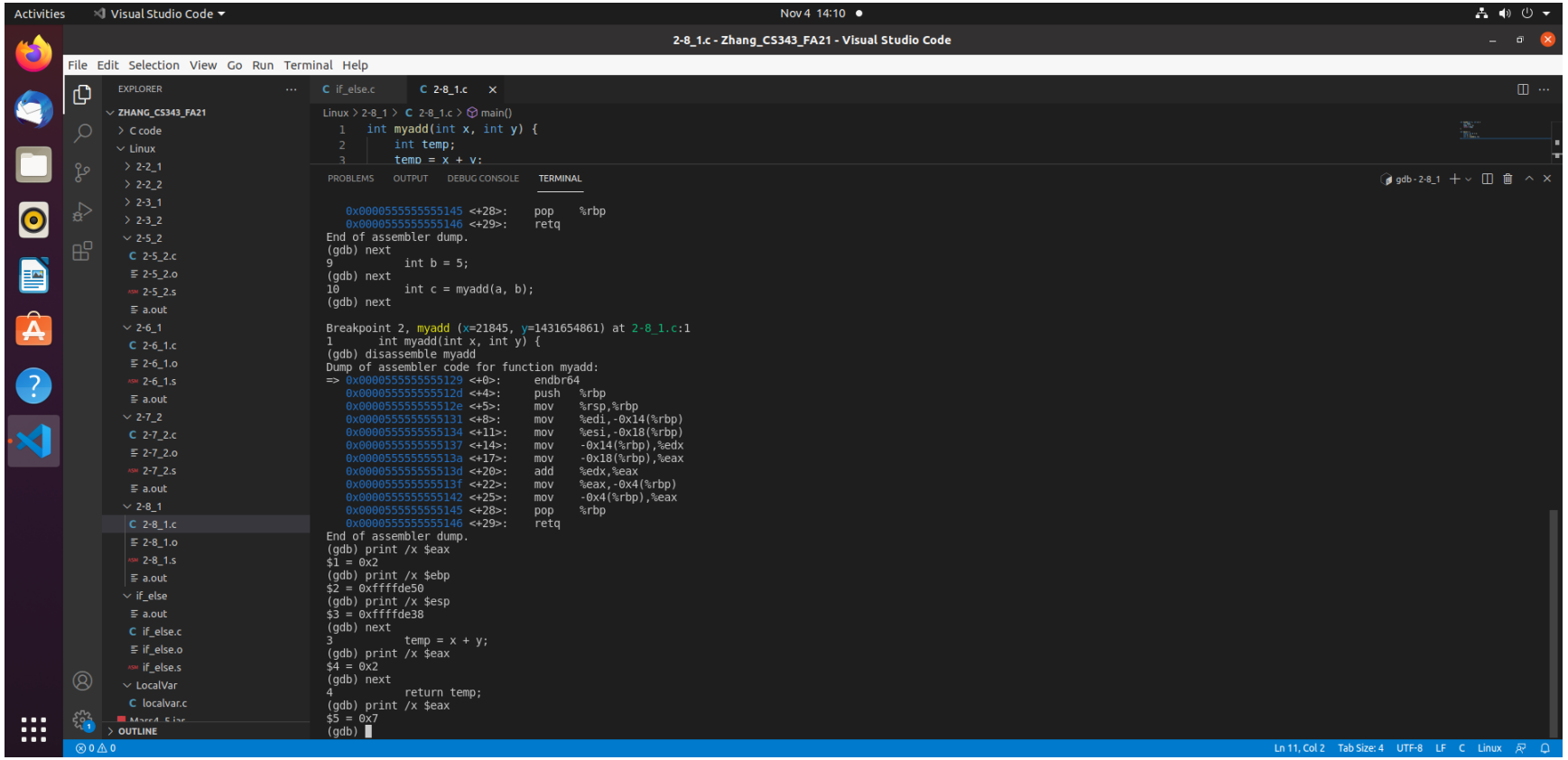
In the figure above, we have moved several instructions forward such that we can see that in this bit of code, we have local variables therefore the stack pointer has something to point to. If we look in the register window, ESP has the address 001EFC48, which has a value of 5 (look in memory window 4). Furthermore, at the bottom. We also see that 0x00BEA000 holds the value of 2 therefore we know that the values have been stored properly.



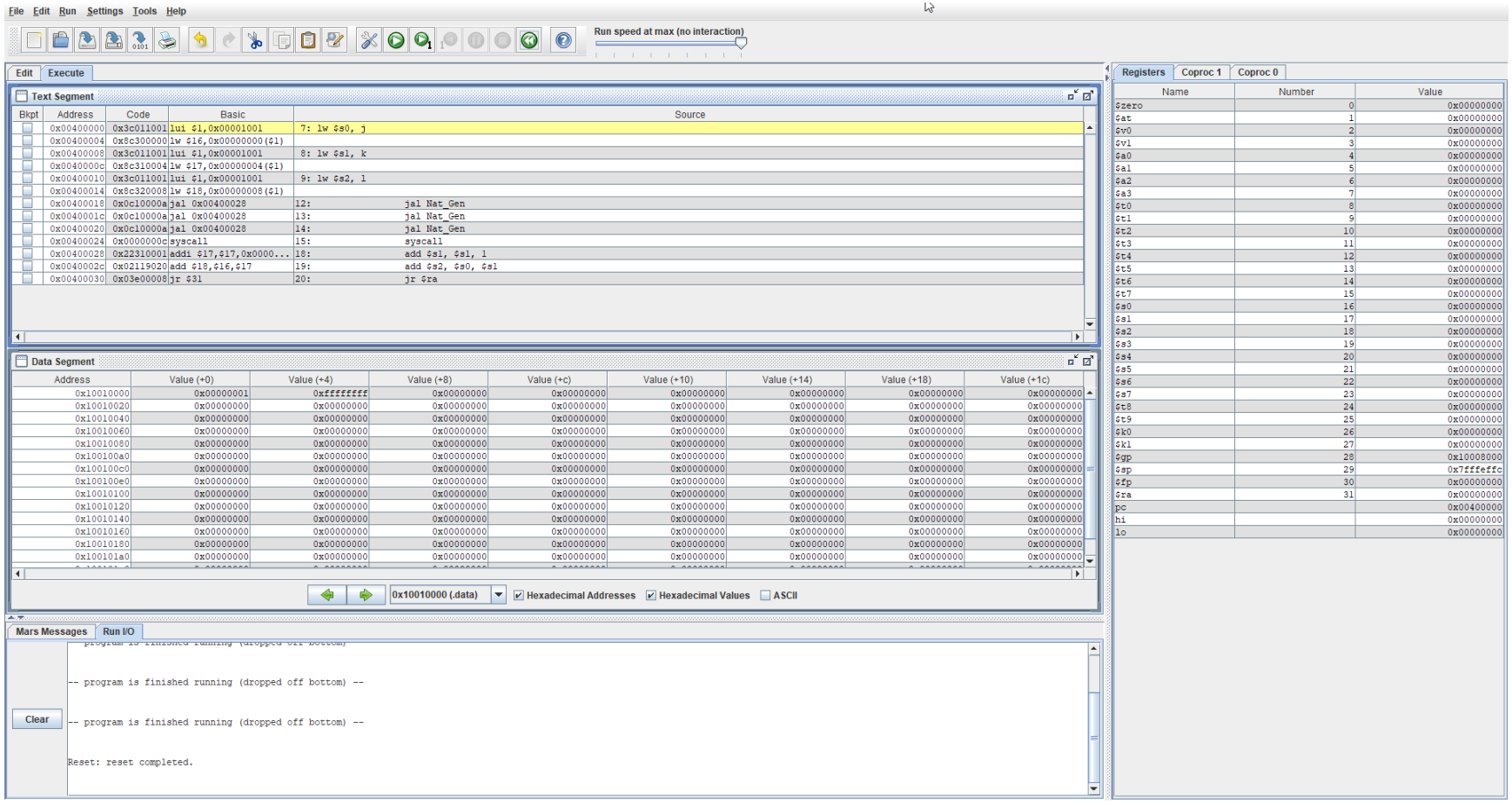
In this figure above, we are now looking at the myAdd function disassembly file and we are past the return value and as we can see the address 0x00113F9E0 has the value of 7 which means that the values have been added successfully. We can also look in memory window 2 to see where the values 2 and 5 have been allocated and if you look at the top 2 most addresses, values 2 and 5 are stored.



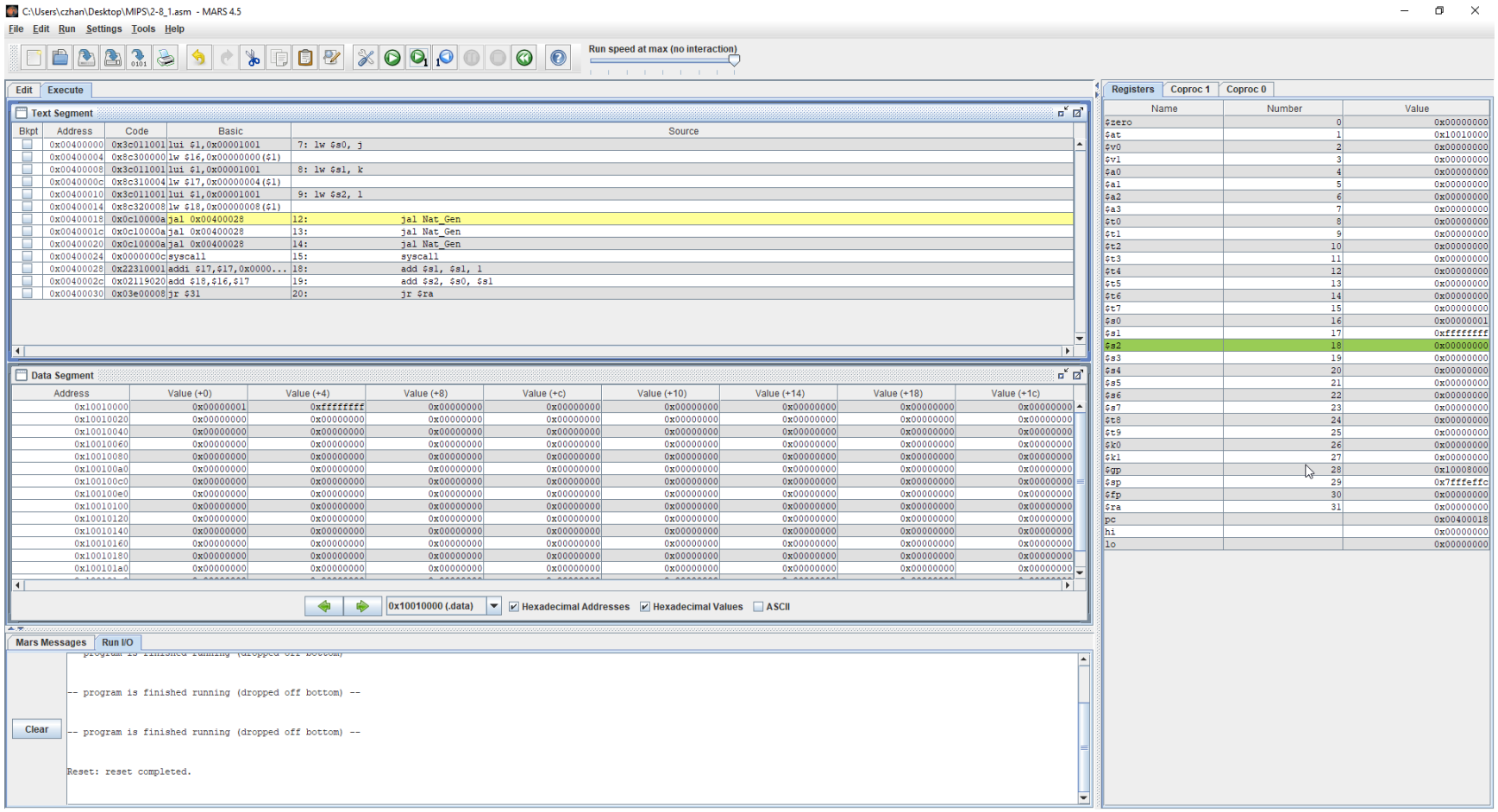
In the figure above, I have compiled the question 2-8_1 using GCC compiler. What is shown above is the assembly dump of both the myAdd function and the main. The dump file shows all the instructions that will be done as we progress through the program.



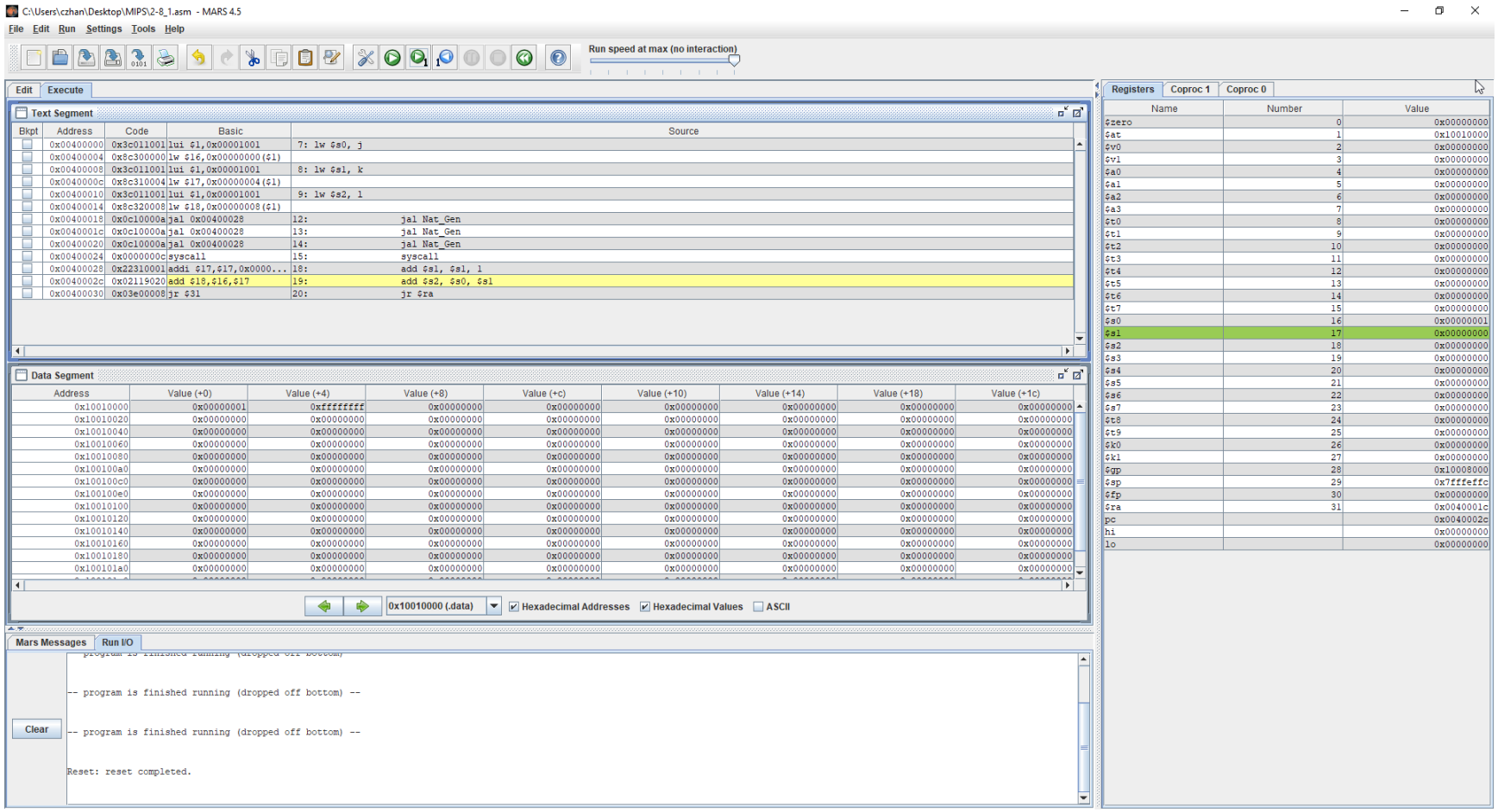
In the figure above, what I have done is I have continued on through many steps in the debugging process and have set two break points. One in the Main file and one in the myAdd function and that is because, we can see the math being done and stored into $eax as we progress. If you look at the figure under the assembly dump, you will see that in the myAdd function, $eax holds the value 0x02 and when we continue past the return temp; line, we see that $eax has the value of 0x7 which means that the myAdd function has done the arithmetic successfully and has returned the value



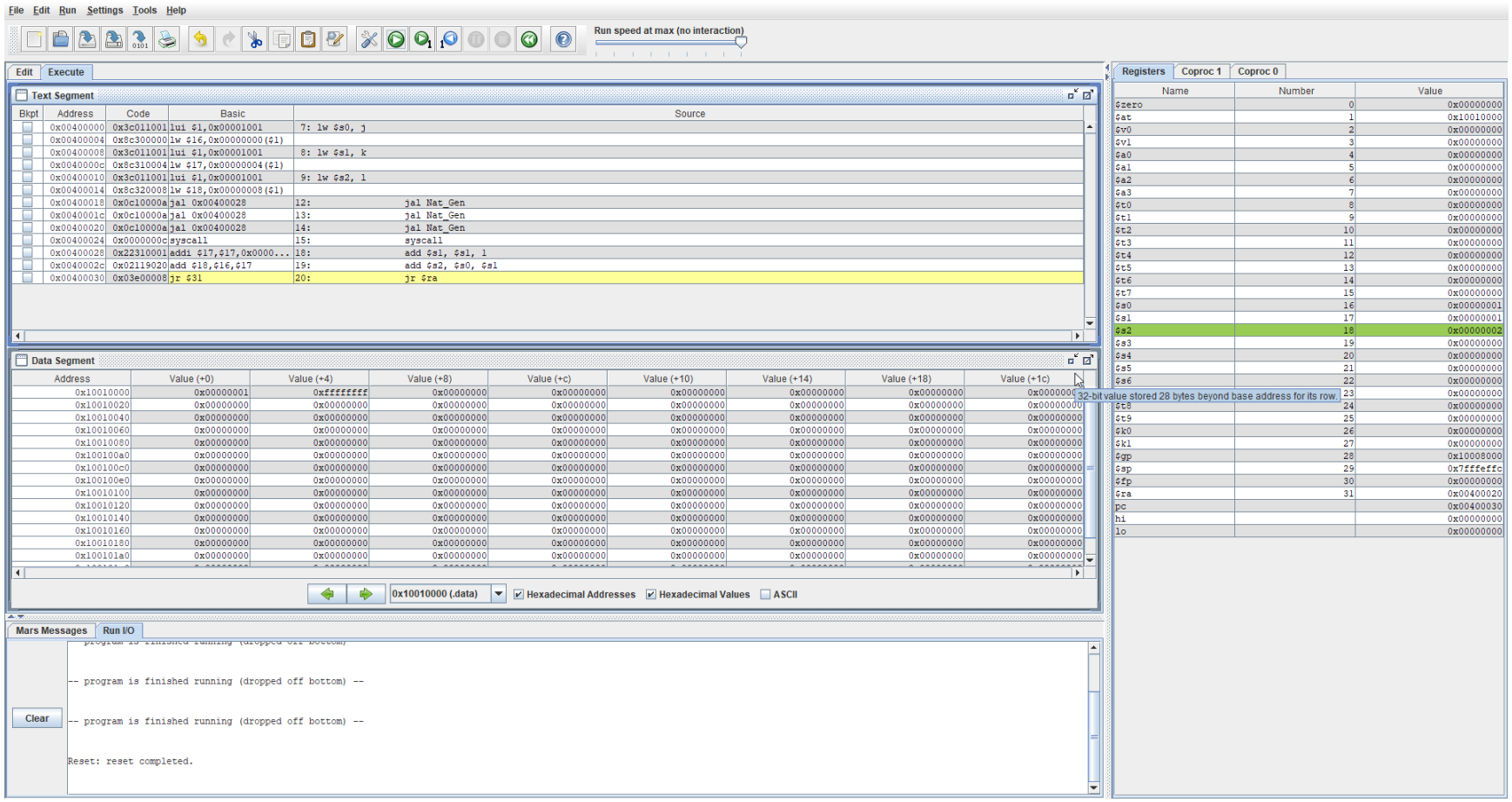
In the figure above, I have written a natural generator assembly file and have compiled it on MARS simulator. You can look in the Text segment to see what instructions are being run and the data segment for saving and loading words as well as the register window to see what is being stored in what register.



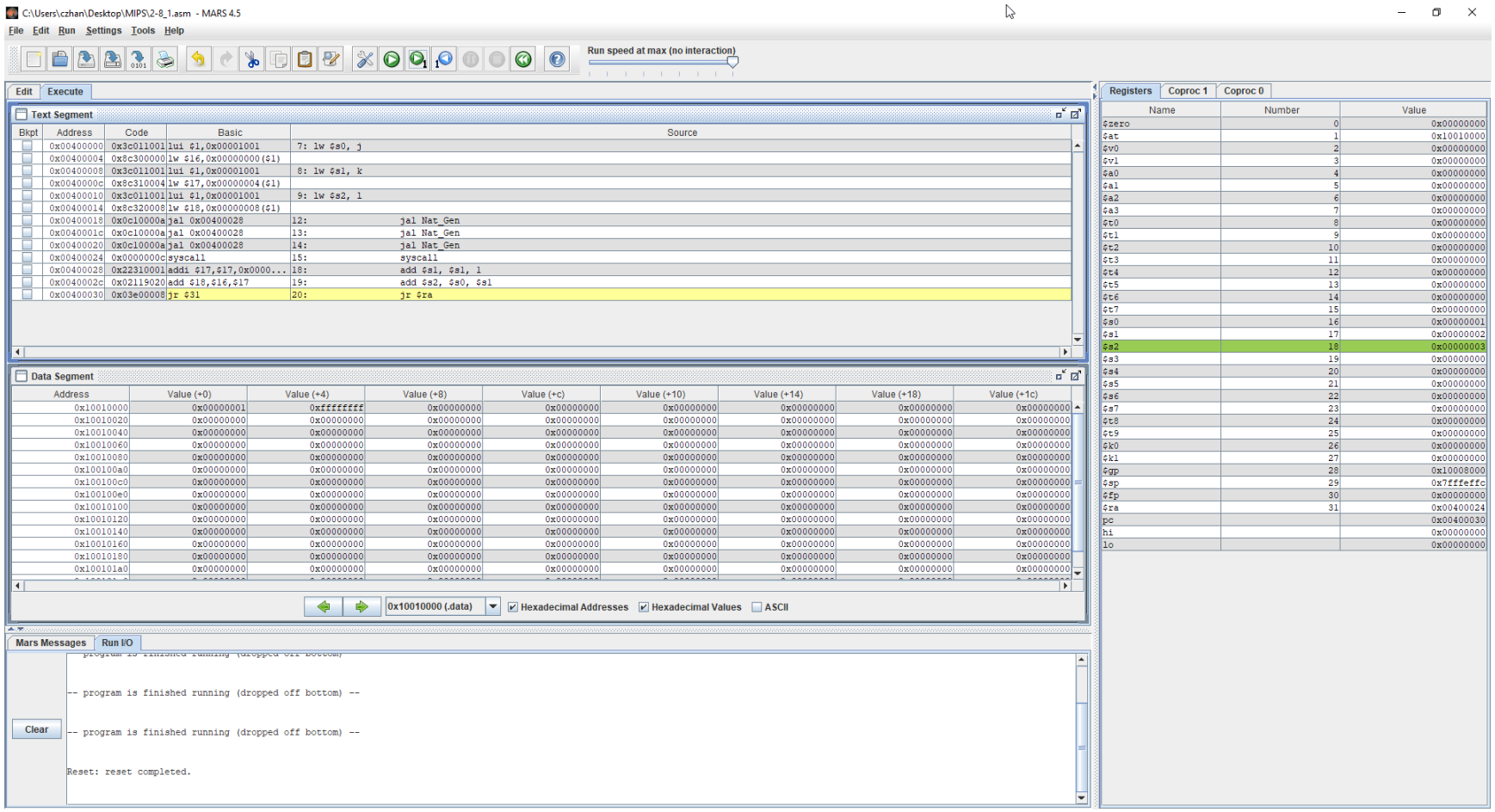
In the figure above, I have ran several instructions where J, K and L were loaded onto $s0, $s1 and $s2 respectively. If we look at the register window in the right, we can see that the values 0, 0xffffffff and 1 are written into $s0, $s1 and $s2. $sp does not change because we are working with static variables and in the nat_gen function, we are using a local static variable.



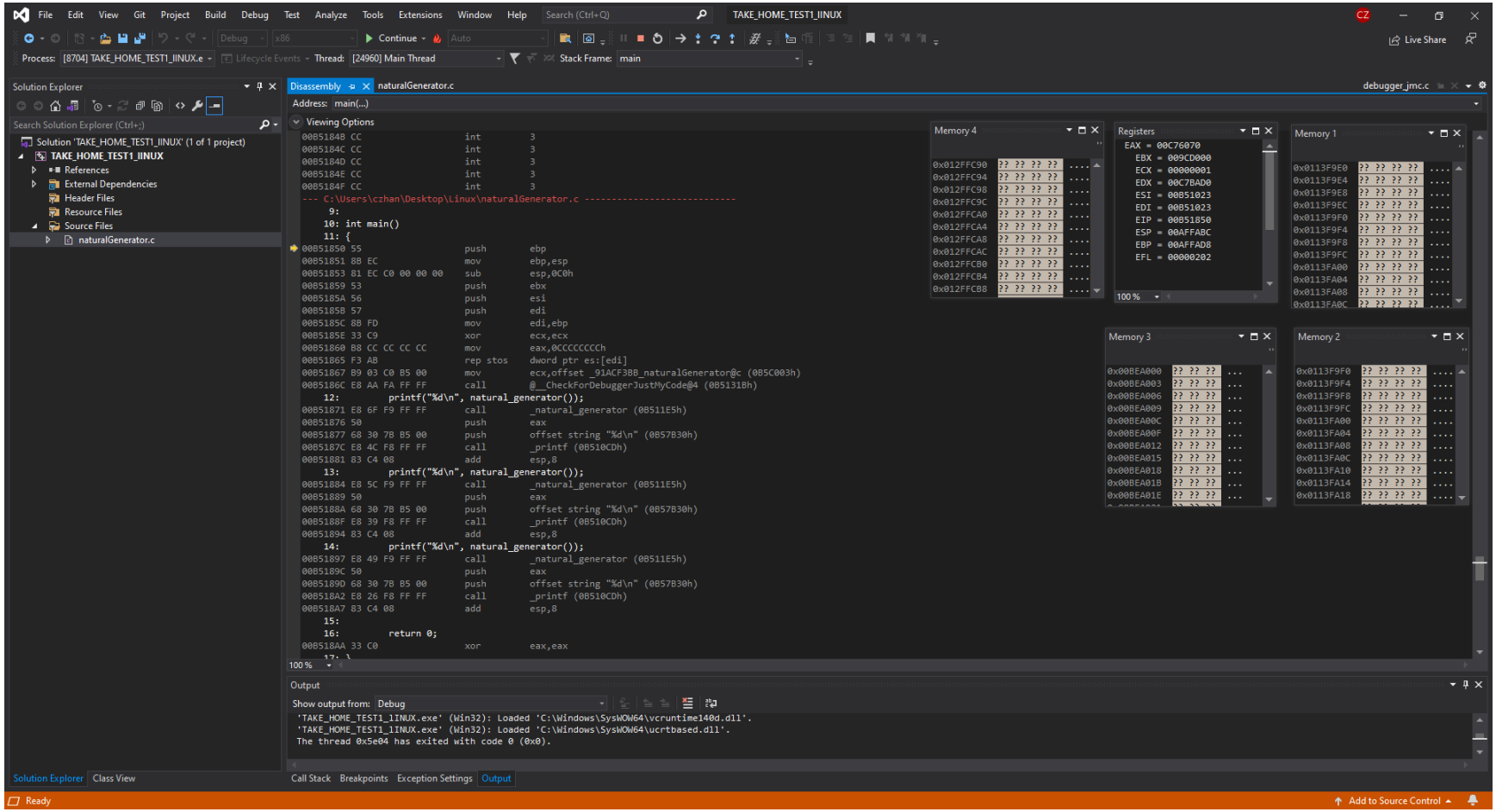
In the figure above, what is shown is the first run of the function call nat_gen and we can see that $s1 has been added 1 and returned value of $s1 + $s2 which is 0. The logic is that everytime the function is called, $s1 is increment by 1 then returned to add with $s2.



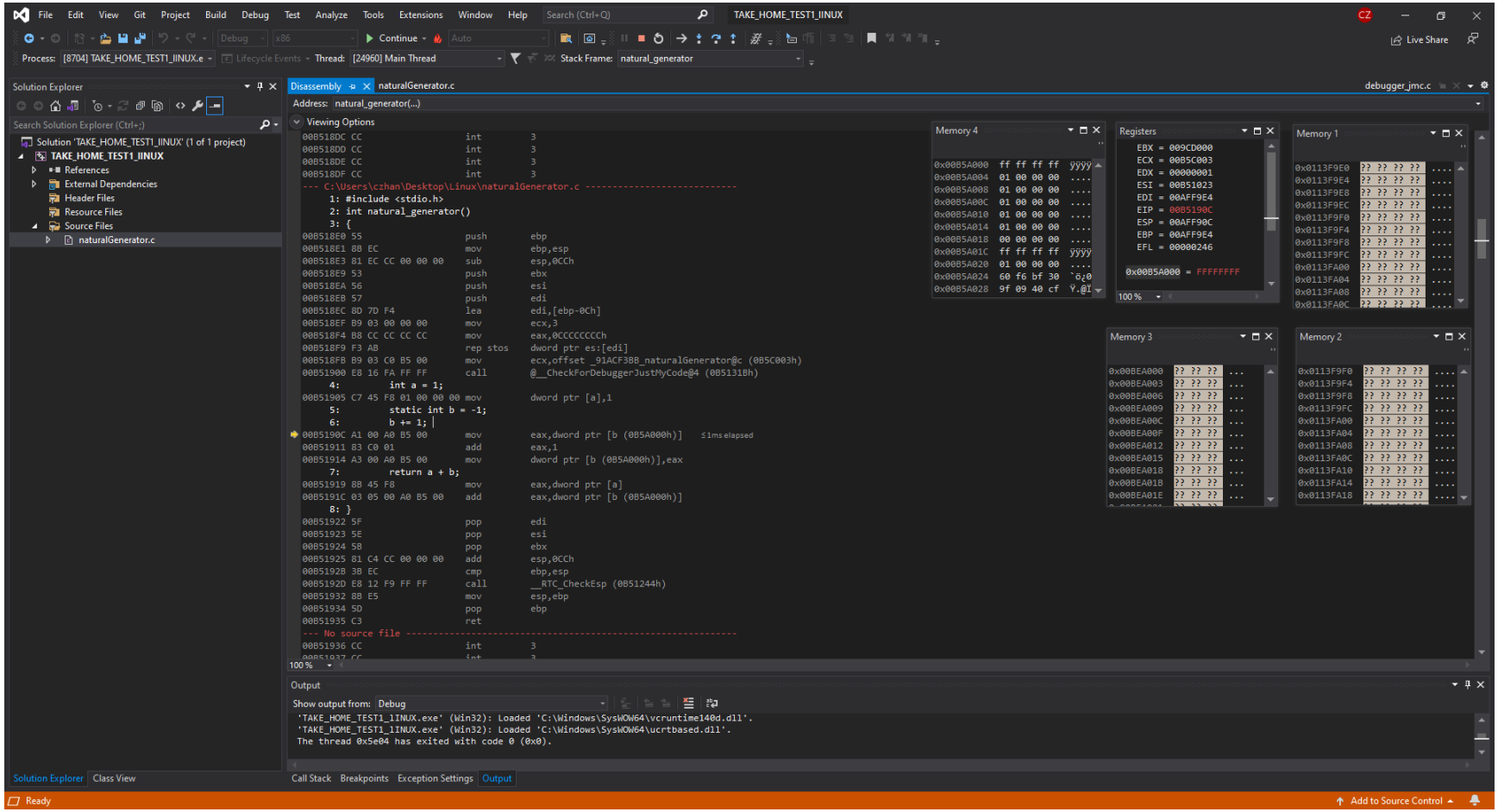
In this figure above, we have run the function call a second time now and if we look at $s2, we can see that the value is now 2 whereas it was 1 before and that is because $s0 + $s1 = 2 because we increment $s1 by 1 every time we run the function then add it to $s0.



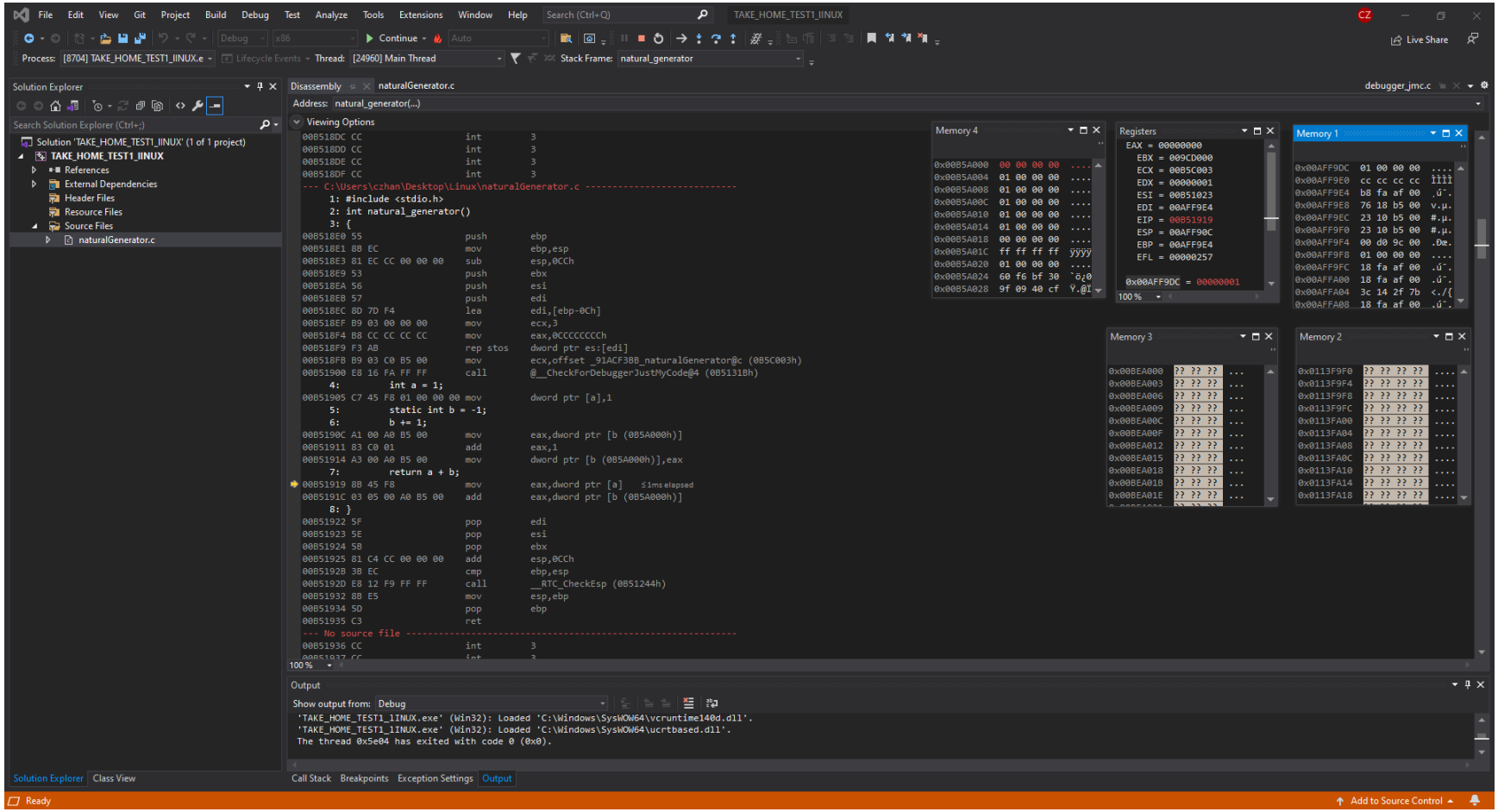
Lastly, the final function call where we increment $s1 then add it to $s0 and this gives us a final value of 3 before termination. This is correct as the static variable that we used does not change in value once it is incremented but if it were to be a local variable, -1 would always be the value of $s1.



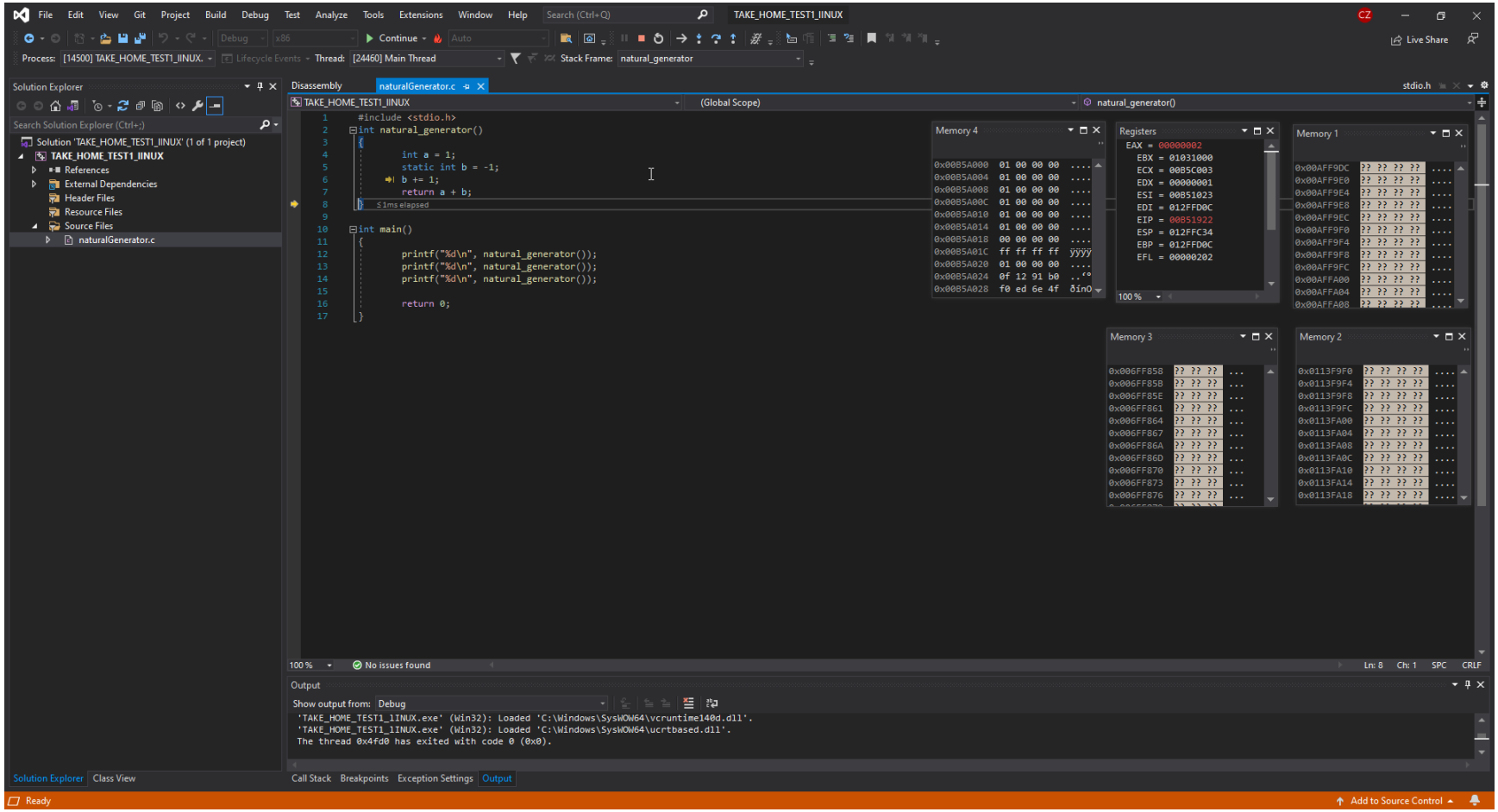
Above figure showcases the natural generator being compiled using an intelx86 compiler with a disassembly window, several memory windows and a register window which we will use to compare the differences and similarities between each different test cases.



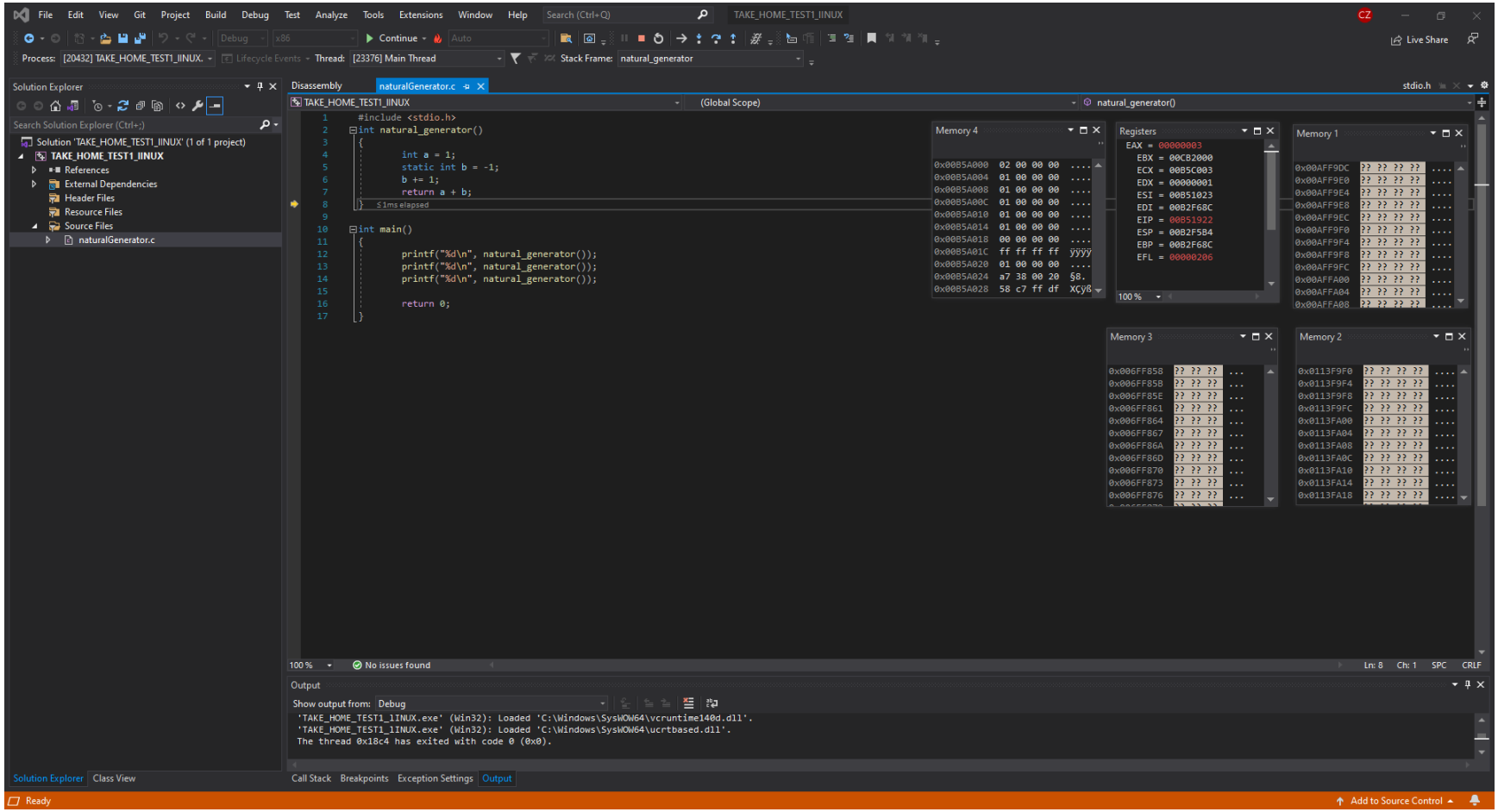
In this figure above, what is shown is the main file being run and going through the first iteration of the natural generator function. We can see the value of b as it is set to FFFFFFFF at the address 0x00B5A00. FFFFFFFF = -1 therefore it is correct and the stack pointer points towards the address 00AFF90C because we are working with a local variable a.



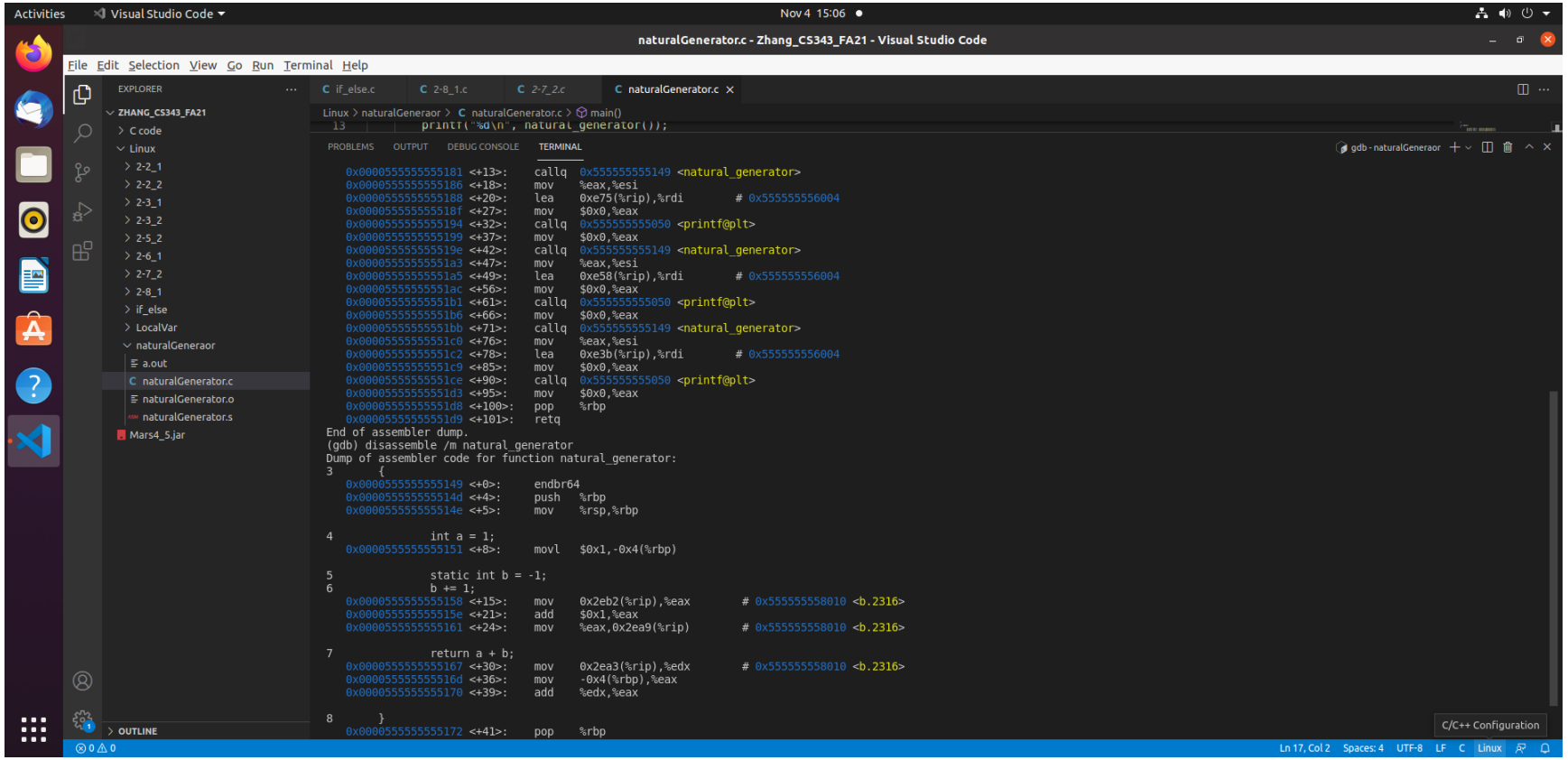
In this third figure, B has been incremented by 1 and then added with a which returns 1 in EAX. We can see the value being incremented if we look at memory window 4 as the value is now 00. Furthermore, we can look over at memory window 1 which shows where the stack pointer is currently pointing at, and it is 1 because that is the value of a.



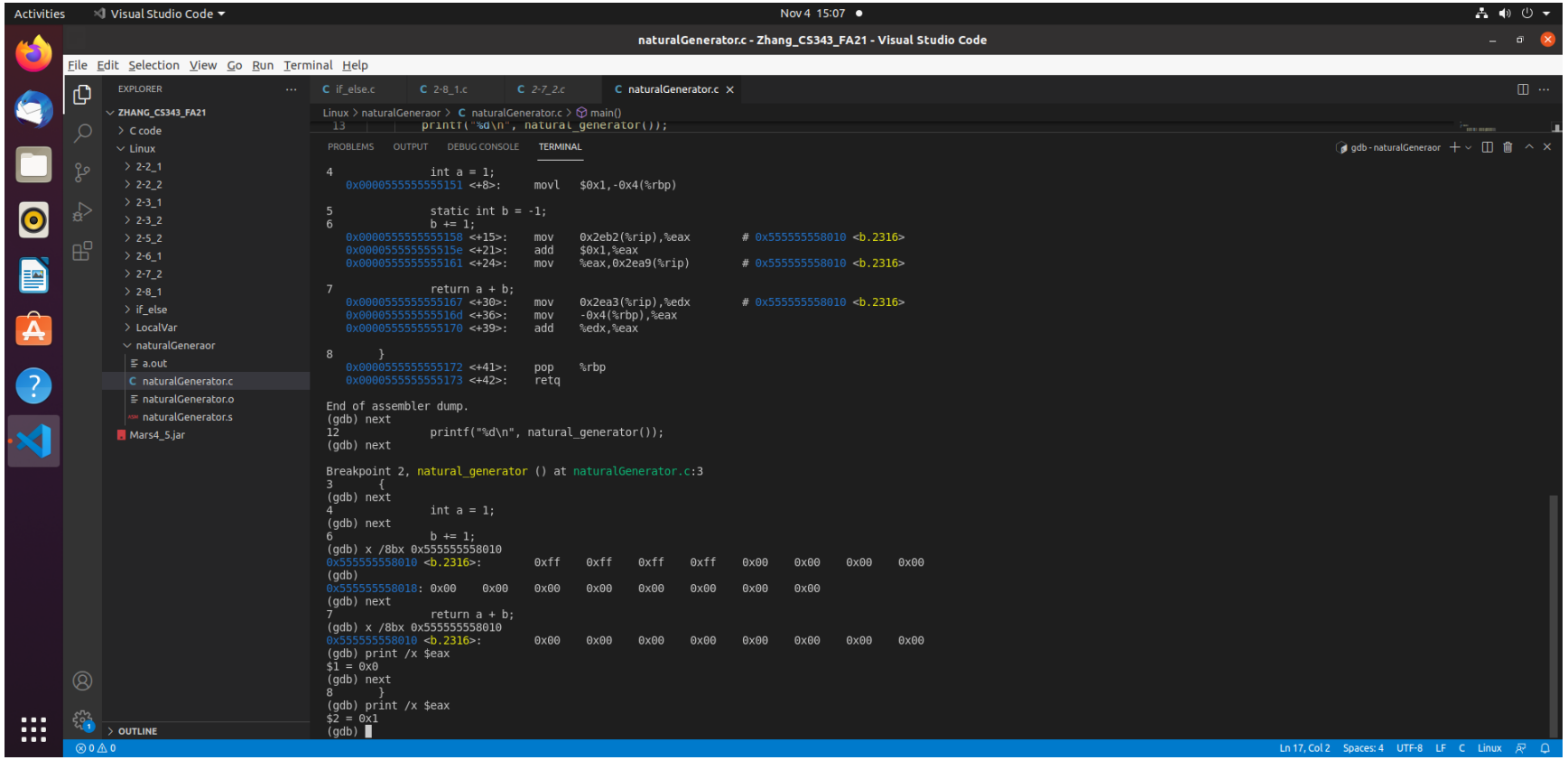
In this figure above, the 2nd iteration of the function call has been ran and we can see there is now a new ESP therefore the old address that is in memory window 1 will not show the local variable value of a and only ??. Furthermore, if we look at EAX, we can see that the computation has been performed successfully as in memory window 1, B has been incremented by 1 once again and now has a value of 1. A + b = 2 now because and b both share a value of 1.



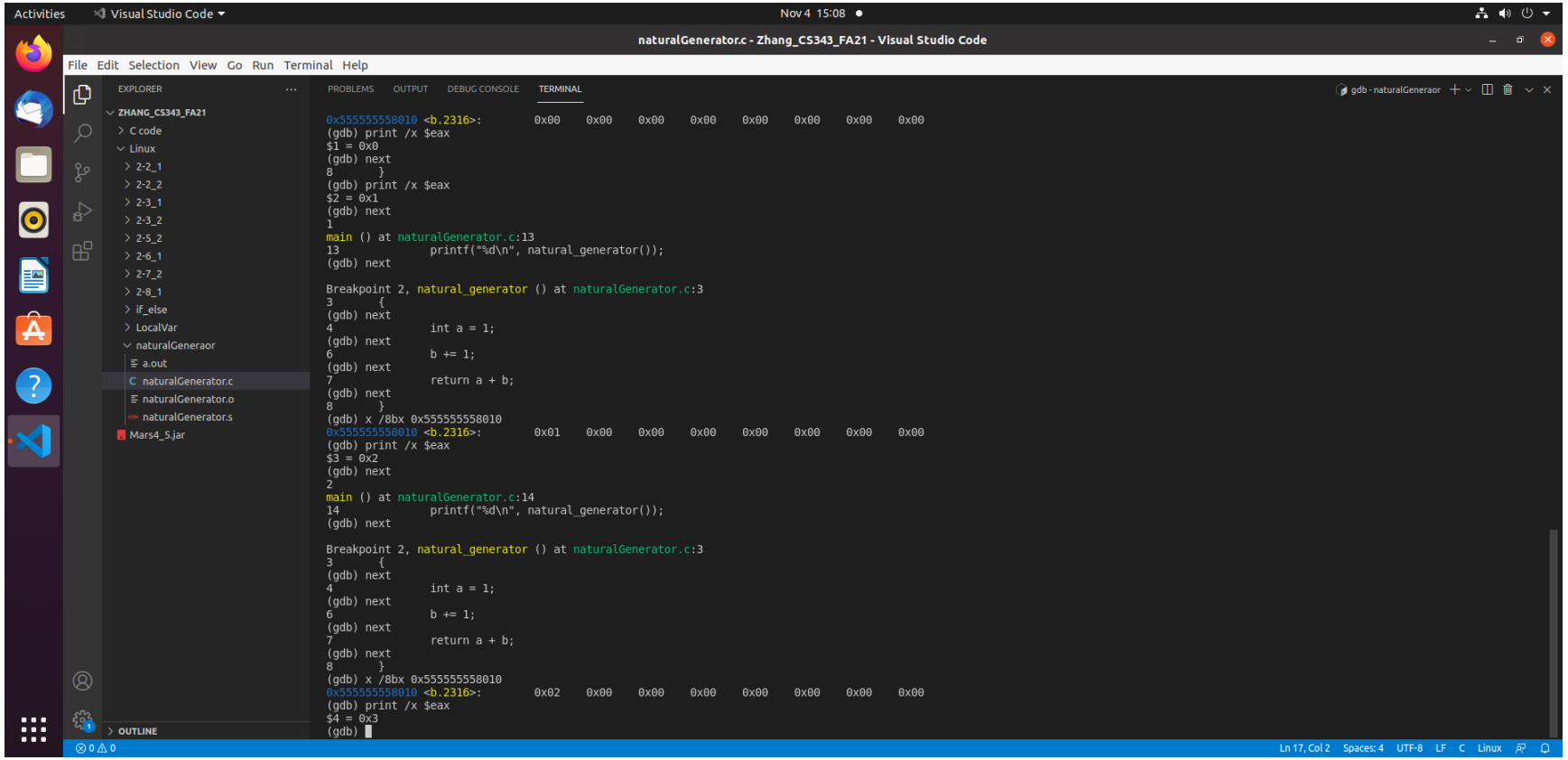
In the figure above, the final iteration of the natural generator function has been run and similarly if we look at memory window 4, we can notice that B has been incremented once more and now has a value of 2. Following the logic of the natural generator, 2 + 1 = 3 so EAX should have 3 as its value and if we look in the register window, EAX does have the value 3 stored inside. Furthermore, we can notice that the ESP has changed once again to point towards the new A value that is in the stack.



The figure above showcases the Natural Generator being compiled using a GCC compiler and what is seen in the figure above is the dump file for both the main and the function natural_generator. What is given to us is the address of b that we can access globally unlike the local variable a.



In this figure, the first iteration of the natural_generator function has been done and if we look at the line under b += 1, we can see the original value of static int b = -1 being represented as hex. We move to the next step and print out b again and we get 0 instead of -1 so we know that it has incremented. Then we continue to print the $eax to see if the sum has been calculated and at the bottom of the figure, we see that $eax now has a value of 1.



In this figure above, all iterations of the function natural_generator has been completed and we can see the value of b being incremented per function call as in the second iteration, b now has a value of 0x01 and then in the last iteration, it has a value of 0x02. Furthermore, $eax also has a value of 0x2 in the 2nd iteration and a value of 0x3 in the last iteration which shows that the function has performed successfully.

In this Test, what was conducted were 3 different cases of compilation. One in MIPS, one in Intelx86 processor and one in GCC compiler. All of the compilers when debugged, showed that the values were computed successfully however, the main difference is the storage. In MIPS, we can notice that the address of the most significant bit was stored first by the looks of the register window and the data segment given to us. In both intelx86 and gcc, the least significant bit was stored first, and we can see this by looking into their memory windows and we can see that we have to read the bits from right to left as opposed to right to left therefore MIPS processor is big endian and Intelx86 & GCC are little endian. I have learned much more about the storage of value in addresses after doing this Take home Test.