**Take Home Test 1 CSC 342**

Danish Faruqi

April 5, 2021

Start Time and Date: April 2, 2021 1:53pm

End Time and Date: asa

***“I will neither give nor receive unauthorizes assistance on this TEST. I will use only one computing device to perform this TEST. I will not use cell while performing this TEST” - Danish Faruqi***

**Table of Contents**

1. **[Title Page](#Title_Page)**
2. [**Table of Contents**](#Table_of_Contents)
3. [**Objective**](#Objective)
4. [**Part 1 - MIPS**](#MIPS)
   1. [2-2\_1.asm](#MIPS_2_2_1)
   2. [2-2\_2.asm](#MIPS_2_2_2)
   3. [2-3\_1.asm](#MIPS_2_3_1)
   4. [2-3\_2.asm](#MIPS_2_3_2)
   5. [2-5\_2.asm](#MIPS_2_5_2)
   6. [2-6\_1.asm](#MIPS_2_6_1)
   7. [2-7\_1.asm](#MIPS_2_7_1)
   8. [main\_myadd.asm](#MIPS_myadd)
   9. natural\_generator.asm
5. **Part 2 - x86 Intel on Windows 32-bit**
   1. 2-2\_1.c
   2. 2-2\_2.c
   3. 2-3\_1.c
   4. 2-3\_2.c
   5. 2-5\_1.c
   6. 2-6\_1.c
   7. 2-7\_1.c
   8. main\_myadd.c
   9. natural\_generator.c
6. **Part 3 – LINUX, gcc, gdb 64bit on Intel x86-64 ISA**
   1. 2-2\_1.c
   2. 2-2\_2.c
   3. 2-3\_1.c
   4. 2-3\_2.c
   5. 2-5\_1.c
   6. 2-6\_1.c
   7. 2-7\_1.c
   8. natural\_generator.c
   9. main\_myadd.c
7. **Conclusion**

**Objective**

The objective of this test is to demonstrate the knowledge and compare MIPS, Windows 32bit Intel x86 ISA, and Linux 64bit Intel x86-x64 ISA architectures. Examples from Chapter 2 will help show the different concepts of Mips and C language. Other architectures and features will be present that are compared and highlighted throughout this test.

**Part 1 – MIPS**

**2-2\_1.asm**

2-2\_1.asm focuses on showing the relation of registers on basic level. **Figure 1** shows the code written out where a is being assigned the value b + c and d is assigned the value a – e. Each static variable is assigned a register and the values are present in the corresponding register. In the end the registers are updated again to hold the new values and the values in memory are also updated since they are all static variables.

**Figure 2** and **Figure 3** show that for each variable a through e have a corresponding register $s0 - $s4 respectively. Before each register can be assigned a value, the address of each variable is loaded from memory to be stored in the $at register, from which the value is retrieved from memory and stored into the correct register.

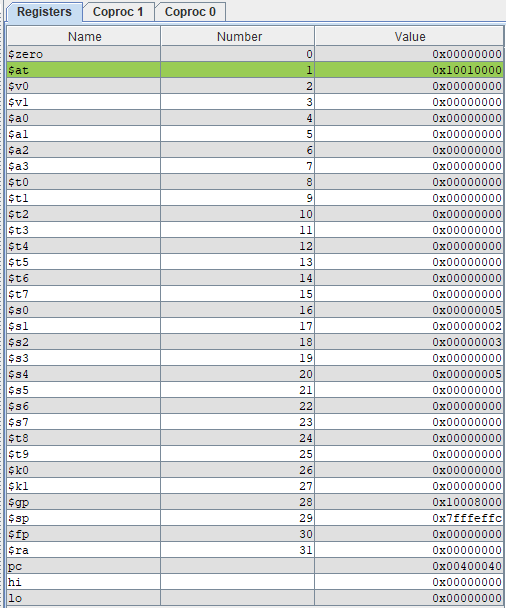
****For example, the address of static variable a, which is 0x10010000 is first stored in the register $at. Since a is the first value stored in memory location 0x10010000 at offset 0 (+0), the value is retrieved from memory and pushed onto register $s0. The value of a is 0x00000001 in big endian. This is done for the rest of the variables, all pulled from memory since they are static variables in big endian. To perform the operation a = b + c, register $s1 is added to $s2 and stored in $s0. The word is then stored back into memory at offset 0 (+0) in big endian as the value 0x00000005. To perform the operation d = a - e, register $s0 is subtracted from $s4 and stored in $s3. The word is then stored back into memory at offset c (+c) in big endian as the value 0x00000000.

Figure 2 - Faruqi\_2-2\_1 Registers

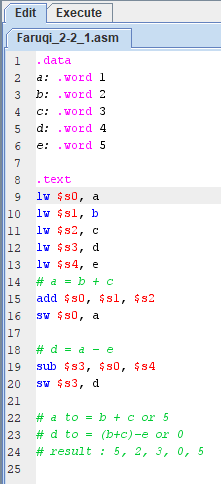


Figure - Faruqi\_2-2\_1 Code

$at stores memory address

$s0 stores value of a

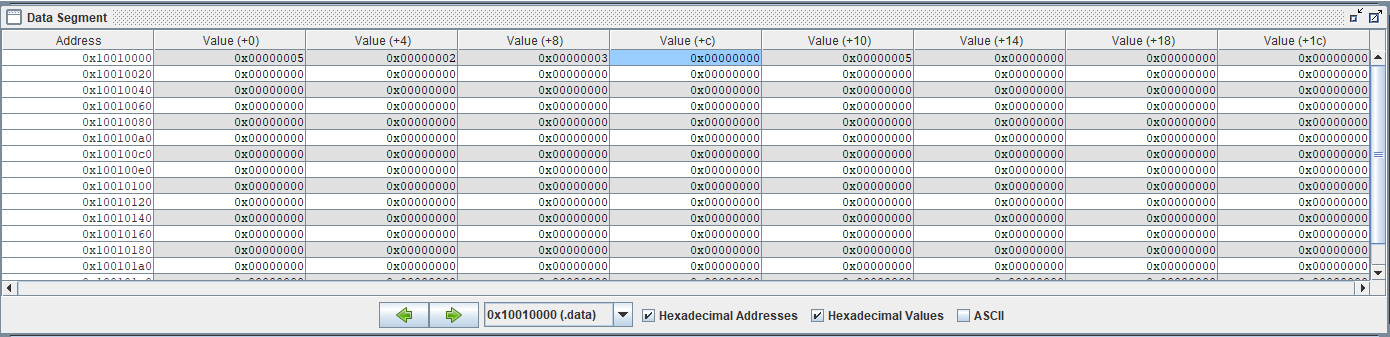
****Variable values are stored in memory and reflect all changes.

Figure 2 - Faruqi\_2-2\_1 MIPS Register

Figure - Faruqi\_2-2\_1 Code

Figure 3 - Faruqi\_2-2\_1 Data Segment

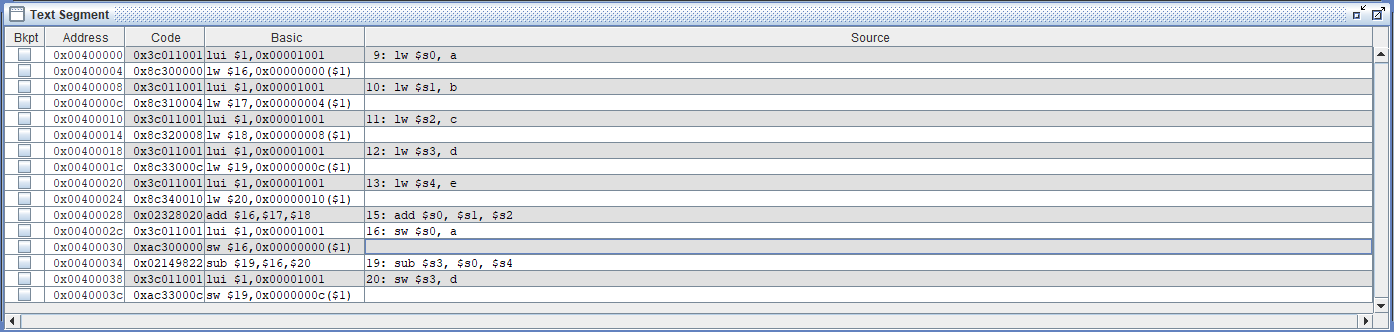
****

Figure 4 - Faruqi\_2-2\_1 Instructions

**2-2\_2.asm**

2-2\_2.asm also focuses on showing the relation of registers on basic level. **Figure 5** shows the code written out where f is being assigned the value (g + h) – (I + j). Each static variable is assigned a register and the values are present in the corresponding register. In the end the registers are updated again to hold the new values and the values in memory are also updated since they are all static variables. There is also a use of two other registers to store the temporary values of (g + h) and (I + j).

**Figure 6** and **Figure 8** show that for each variable f through j have a corresponding register $s0 - $s4 respectively. And to store the temporary values of (g + h) and (I + j) registers $t0 and $t1 are used respectively. Before each register can be assigned a value, the address of each value is loaded from memory to be stored in the $at register, from which the value is retrieved from memory and stored into the correct register as big endian.

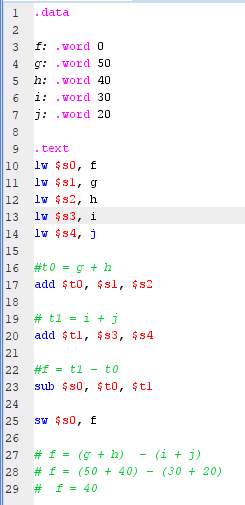
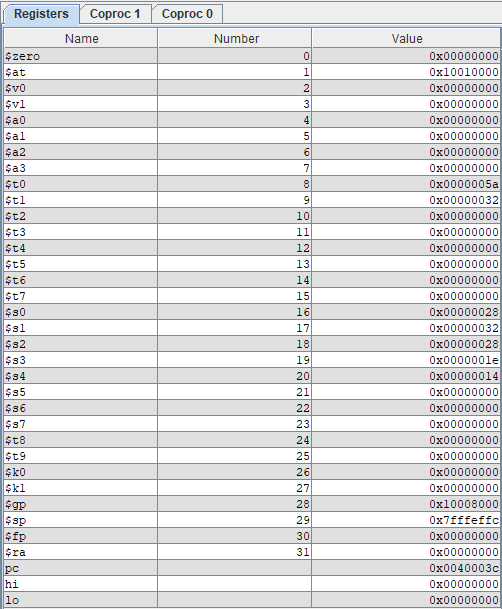
For example, to get the value of g, the address of g is stored into register $at, which is 0x10010000. Then the word is loaded into $s1 by adding 4 (+4) to the memory address since it is the second value. That value of g, 0x00000032 is then stored in $s1. After this is done for every value, the value of registers $s1 + $s2 is stored in a new register called $t0. Register $t1 stores the value of $s3 + $s4. From here, register $s0 stores the value of $t0 - $t1 which is 0x00000028. And finally, the value of $s0 is stored back in memory since it is a static variable in big endian. It’s stored at the address 0x10010000 by adding 0 (+ 0) since f is the first address on memory.

Figure 6 - Faruqi\_2-2\_2.asm Register

Figure 5 - Faruqi\_2-2\_2.asm Code

$s0 is initially 0 but then stores f = (g + h) - (i + j) or f = 0x00000028

$t0 stores g + h and $t1 stores i + j

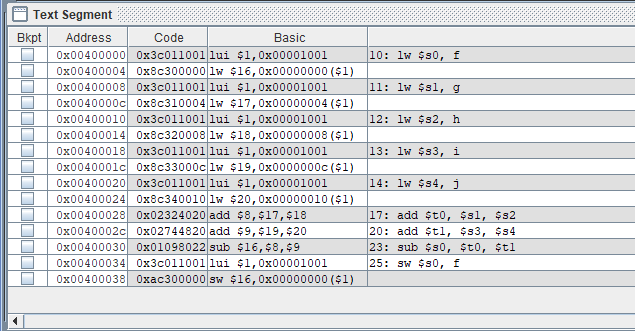
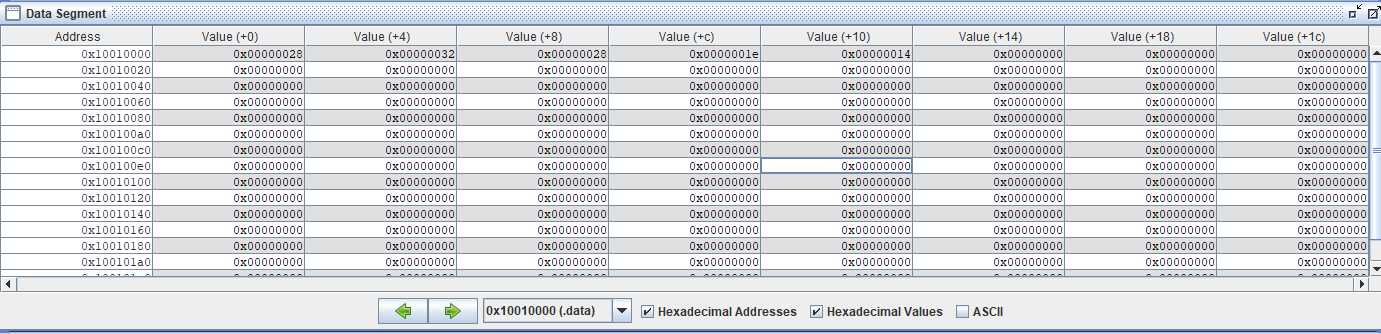


Figure 7 - Faruqi\_2-2\_2.asm Instructions



No other value changes on value since no other value was changed.

F is stored at + 0 and so when its value is changes, the value is changed in memory as well.

Figure 8 - Faruqi\_2-2\_2.asm Data Segment

**2-3\_1.asm**

2-3\_1.asm shows memory operands along with normal operands. It focuses on showing the relationship of memory with operands and how to access memory. **Figure 12** shows the code written out where static variables g and h are declared along with the static array A. A[8] is set to a value and then used to compute g = h + A[8].

At first, the register $t1 stores 0x00000037 or 55 in decimal as shown in **Figure 10**. Then array A’s address is loaded int register $at and pushed into register $s3, which is 0x10010008 in big endian. Then the value of $t1, which is 0x00000037 is stored in memory at location 0x10010008 + 32. 0x10010008 is the address of A[0] is memory, to get to A[8], 8 is multiplied by 4 to get to the correct address. From here the value of g is loaded from memory into register $s1 since g is a static variable in big endian. Next, in the same way, h’s value of 0x00000016 is loaded from memory since it’s a static variable in big endian in $s2. Now 0x00000037 is loaded from A[8] into $t0, again to access it from memory you need to access 0x10010008 + 32. $s1 stores the addition of 0x00000016 + 0x00000037, which is $s2 + $t0. In the end since g is a static variable it’s new value needs to be saved into memory. At memory address 0x10010000 + 0 the value 0x0000004d is stored. In the end we have

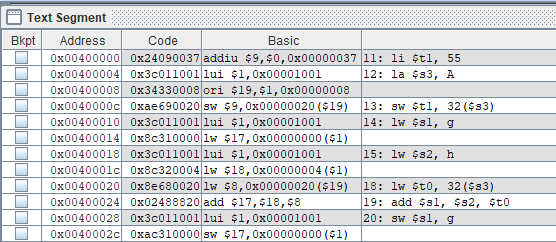
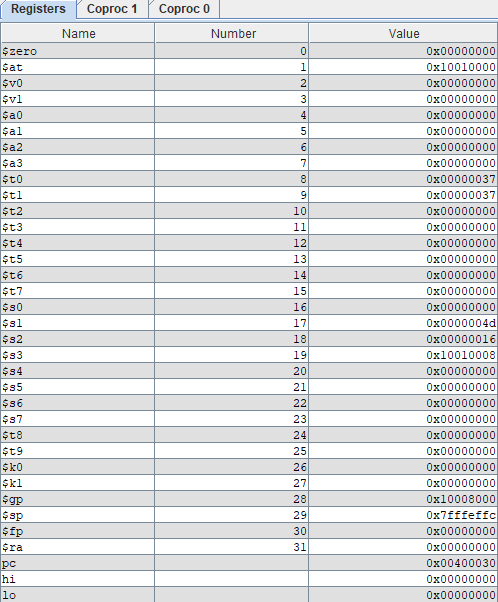
g = h + A[8] 🡺 g = 0x00000016 + 0x00000037 🡺 g = 0x0000004d or g = 77

Figure 9 - Faruqi\_2-3\_1 Instructions

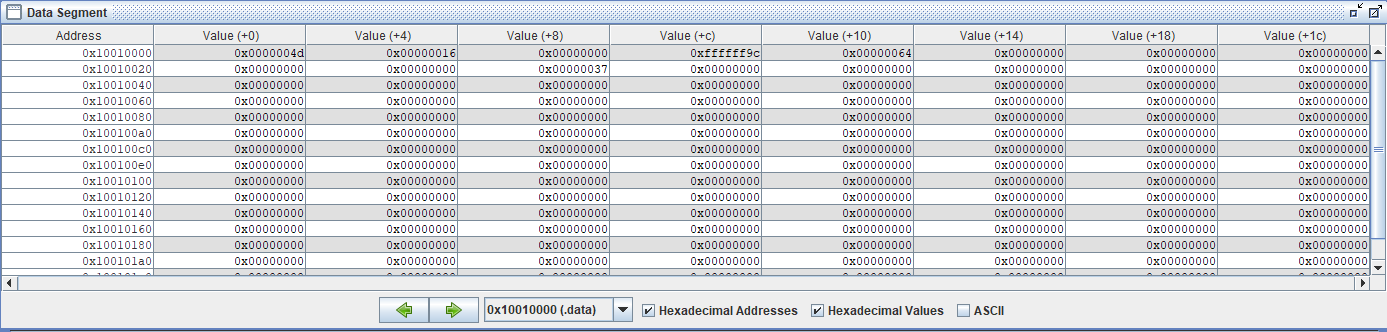


The Array start at 0x10010008 but A[8] is at 010010028

$s1 stores the value of g

$s3 stores the address of the array

Figure 10 - Faruqi\_2-3\_1 Register



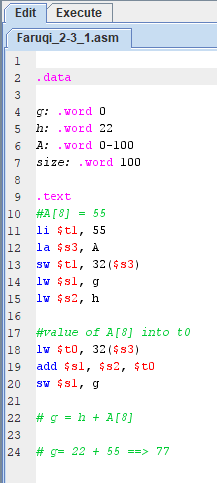
Figure 11 - Faruqi\_2-3\_1 Data Segment

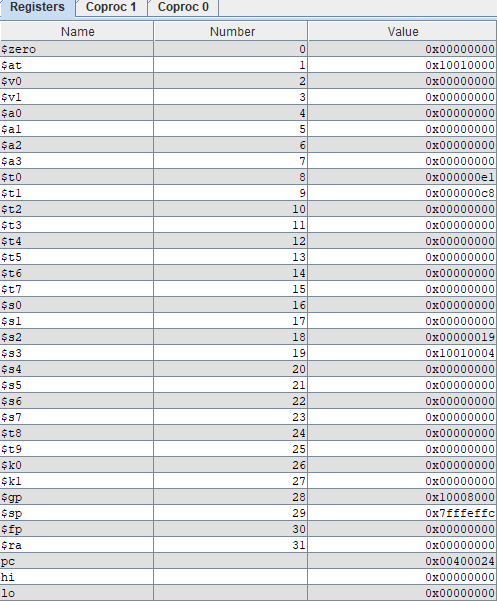
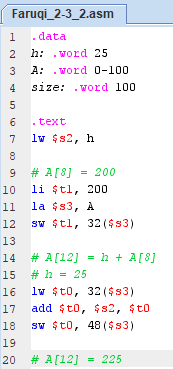
Figure 12 - Faruqi\_2-3\_1 Code

**2-3\_2.asm**

2-3\_2.asm also shows memory operands along with normal operands. It focuses on showing the relationship of memory with operands and how to access memory. **Figure 13** shows the code written out where static variables h = 25 is declared along with the static array A. A[8] is set to 200 a value and then used to compute A[12] = h + A[8].

This works the same way as the previous example worked as shown in **Figure 14**. Starting with h, the value of h is retrieved from memory at address 0x10010000 + 0 since it’s the first value stored at that address. It’s a static variable so it’s value is stored in memory in little endian. The value of h is 0x00000019 or 25 in decimal. Similar to before 200 or 0x000000c8 is stored in the register $t1. Address of A is loaded into $s3, which is 0x10010004 and will be sued from here on to edit the values in memory. First 200 which is stored at $t1 is loaded into memory address 0x10010004 + 0x00000020 which is A[8]. 8\*4 = 32, so we need to add 32 or 0x00000020 to base pointer to get to the 8th index. This is all stores in memory since the array is a static variable and being MIPS its in big endian. 200 or 0x000000c8 is then loaded from address 0x10010004 + 0x00000020 or A[8] into $t0. $t0 then stores the addition of itself and $s2 which is 25 + 200 = 225 or 0x00000019 + 0x000000c8 = 0x000000e1. Now the only thing left is to store this new value into A[12] which is 0x10010004 + 0x00000030 or + 32 from base pointer. The new value of A[12] is 0x000000e1.

A[12] = h + A[8] 🡺 A[12] = 25 + 200 = 225 🡺 A[12] = 0x00000019 + 0x000000c8 = 0x000000e1



Value of h

Value of A[12]

Value of A[8]

Figure 14 - Faruqi\_2-3\_2.asm Register

Figure 13 - Faruqi\_2-3\_2.asm Code

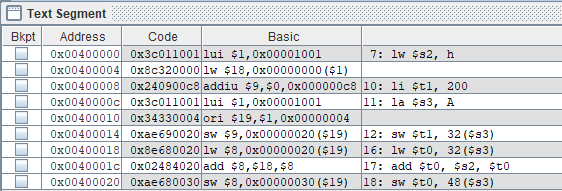


Figure 15 - Faruqi\_2-3\_2.asm Instructions

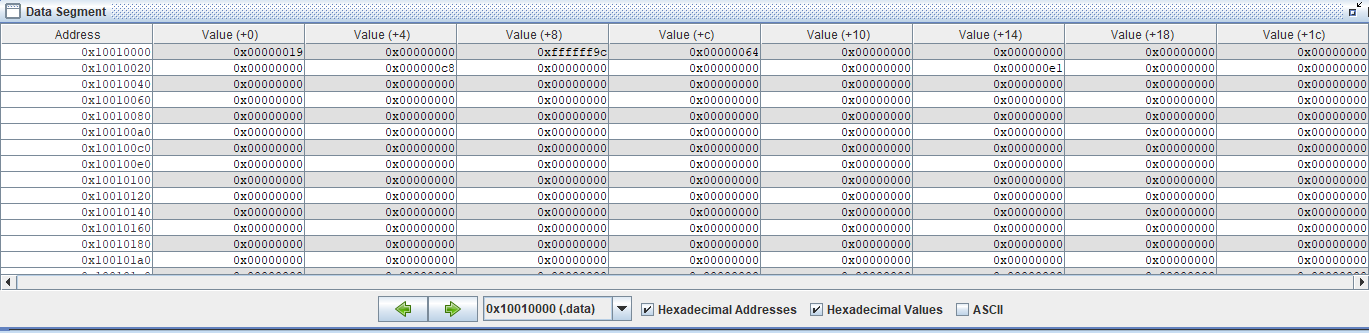


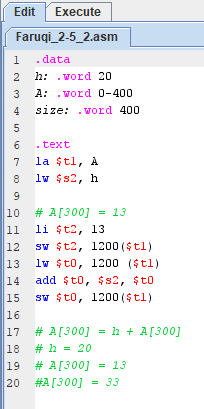
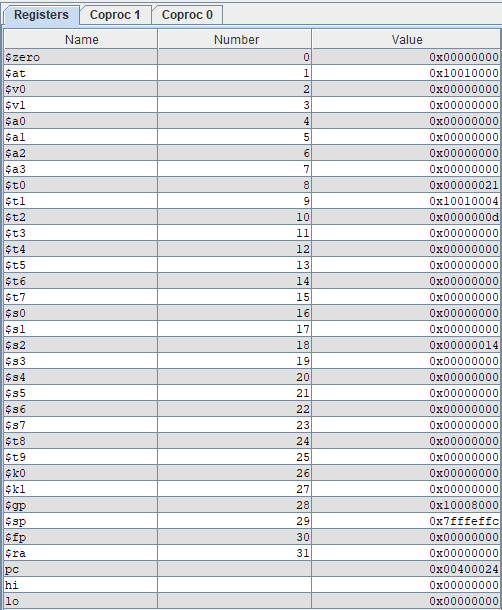
Figure 16 - Faruqi\_2-3\_2.asm Data Segment

The final value of A[12] gets store base pointer + 48 away which is A[0] + 12 indices

**2-5\_2.asm**

2-2\_2.asm focuses on showing the difference between how humans instruct computers and how computers see instructions. Here the essential instruction being performed is A[300] = A[300] + h. Humans would instruct computers this way, but this is not what the computer sees. Computer sees several instructions in between one to call things from memory, locate them and to assign them value.

In this example, the as seen in **Figure** **17**, the static variable h with the value 20 is created and an array A is created. Then A[300] is set to 13, and then A[300] is set to h + A[300]. To go step by step we can see that initially, address of static array A is loaded from memory and put into register $t1. Then, the value of static variable h is loaded from memory (which is in big endian) to $s2. The value of h is 20 or 0x00000014, then A[300] is assigned a value. At first $t2 holds the value 13 or 0x0000000d. Then the value in $t2 is stored in memory at address 0x10010004 + 1200. Since we want to store at A[300], we multiply 300\*4 = 1200. Then 1200 is added to the base pointer of array A. which is 0x10010004 + 0x000004B0 in hex. It is loaded back into $t0 from the same place (0x10010004 + 0x000004B0 or A[300]). Then $t0 holds the value of A[300] which is 13 + value of h at register $s2 which is 20. Now $t0 is 33 or 0x00000021 as shown in **Figure 18**. Then finally, as shown in **Figure 20,** the value of $t0 is stored back into memory since array A is static, at the address A[300] or 0x10010004 + 0x000004B0. It is possible to see the loading, saving, and retrieval of data that happens by computers behind every instruction that a human may command.



Value of h

Value of A[300] on register (final value)

Figure 18 - Faruqi\_2-5\_2.asm Register

Figure 17 - Faruqi\_2-5\_2.asm Code

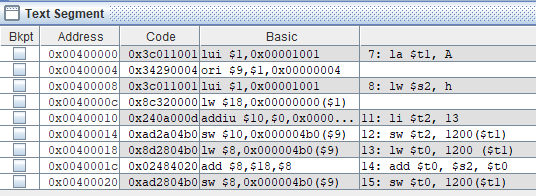


Figure - Faruqi\_2-5\_2.asm Instructions

Value of A[300] stored in memory

Value of h stored in memory

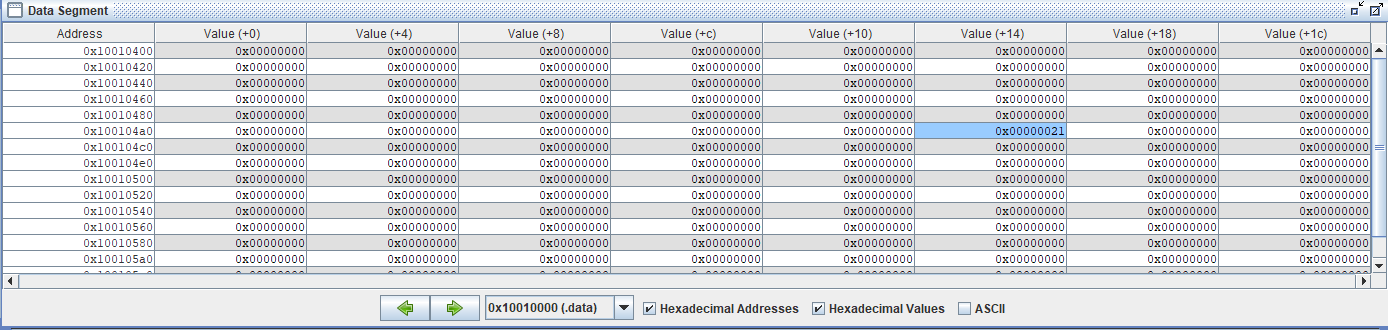
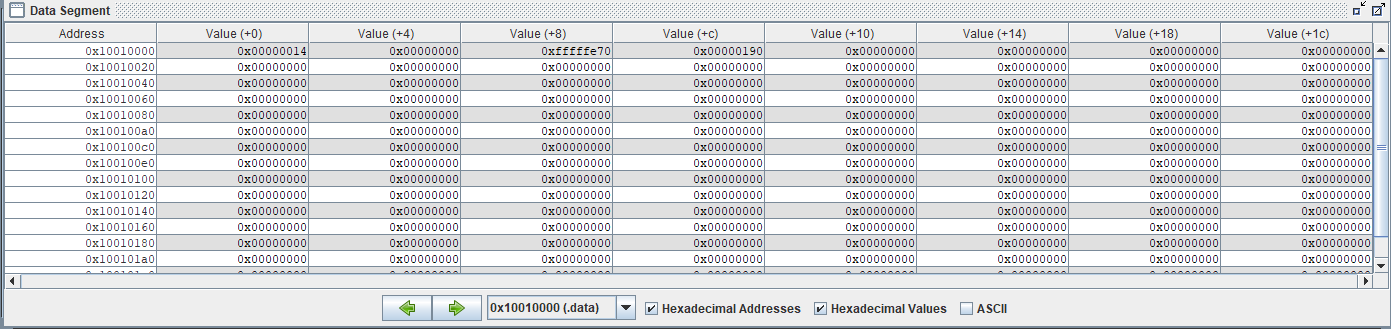


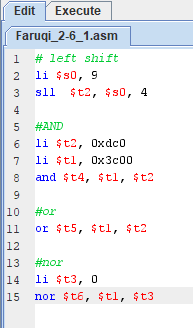
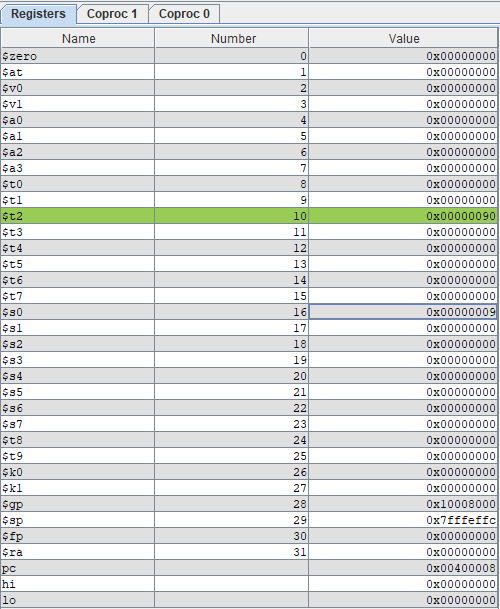
Figure 20 - Faruqi\_2-5\_2.asm Data Segment

Figure 19 - Faruqi\_2-5\_2.asm Data Segment

**2-6\_1.asm**

2-6\_1.asm demonstrates operands present in MIPS and how a computer would deal with them. Here left shifting, and, or, and nor are demonstrated as showing in **Figure 21**. One thing to note is that no memory was used in this program since there were no static variables or anything stored in memory that was to be loaded. As such everything was present in registers and was controlled from registers.

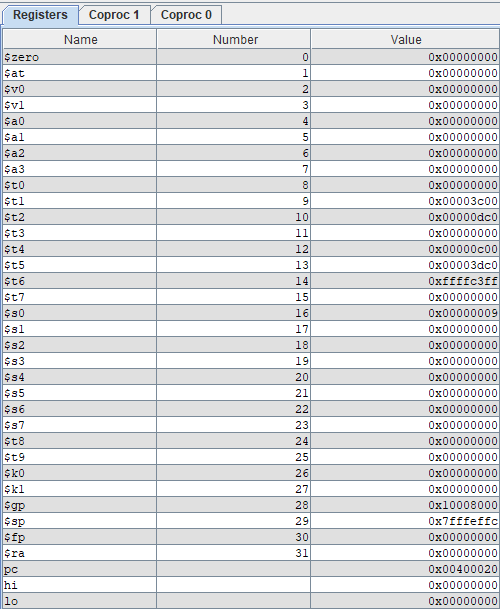
The first action as seen in **Figure 21** is that left shift is performed. They move all the bits in a word to the left or right, filling the emptied bits with 0s. Since shifts are dealt in binary, so left shift a word in hex it must be shifted by 4. As seen in **Figure 22**, shifting the word 4 or one place in hex turns 0x00000009 is $s0 to 0x00000090 in $t2. Next to perform AND and OR operations, values are loaded into registers $t1 AND $t2. As seen in **Figure 23** AND is a bit by-bit operation that leaves a 1 in the result only if both bits of the operands are 1. This this leaves the result of AND in register $t4 to be 0x00000c00. OR is a bit-by-bit operation that places a 1 in the result if either operand bit is a 1. So the operation of $t1 OR $t2 stored in $t5 is 0x00003dc0 as shown in **Figure 23**. NOR is different in MIPS since there is no NOT present in MIPS, which means to perform a NOT you need to NOR against a 0. NOT takes one operand and places a 1 in the result if one operand bit is a 0, and vice versa. So here with 0 in register $t3, it performs the operation 0x00003c00 NOR 0x00000000 giving the result 0xffffc3ff in $t6 as seen in **Figure 23.**



Word gets left shifted 4 bits in binary and one space in hex

Figure 22 - Faruqi\_2-6\_1.asm Register

Figure 21 - Faruqi\_2-6\_1.asm Code



AND Operation

OR Operation

NOR Operation

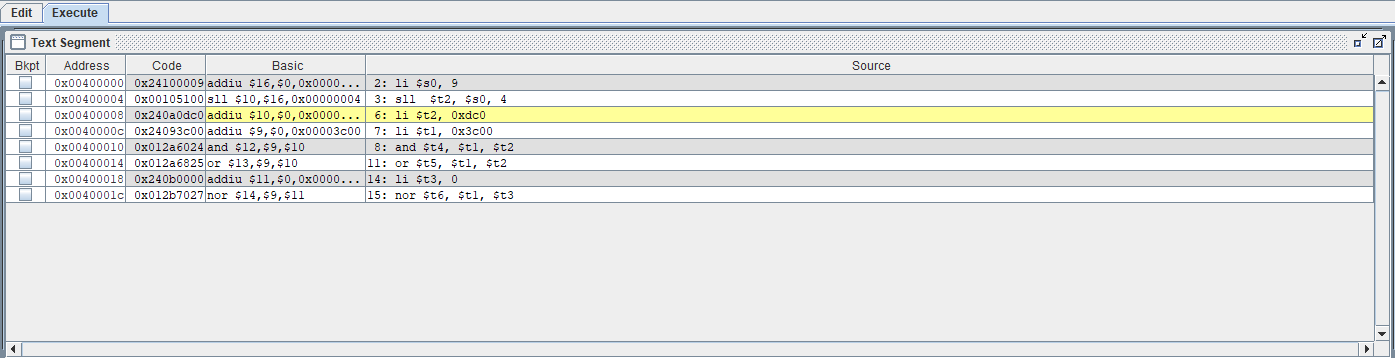
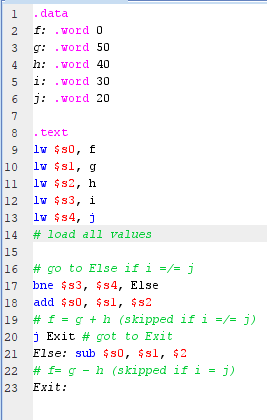
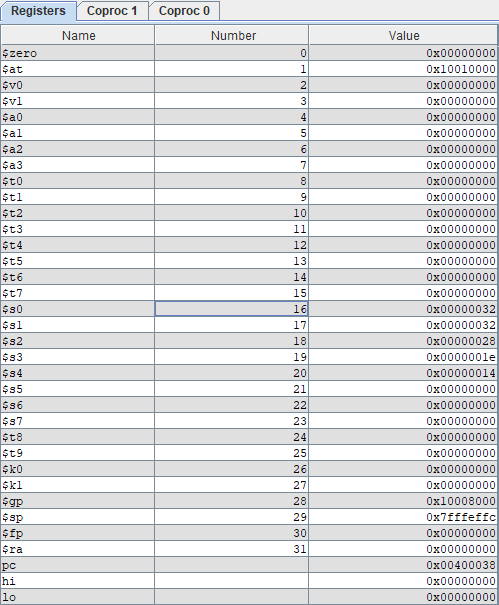
Figure 23 - Faruqi\_2-6\_1.asm Register

Figure 24 - Faruqi\_2-6\_1.asm Instructions

**2-7\_1.asm**

2-7\_1.asm deals with how a computer uses loops in a program. A computer as seen in **Figure 25**, for comparison a computer would use equals (beq) and not equals (bne), it is more efficient to check for bne first so that it doesn’t have to later on, saving it one step. And so, in **Figure 28** bne is performed before be to reduce the amount of comparisons.

To start off, first every word is loaded into the register from memory since they are static. They are loaded in by getting the address from $a1 register and loading f-j into registers $s0 - $s4 accordingly. After all values are loaded into register as seen in **Figure 26**,bne is performed on $s3 and $s4, its comparing if 0x0000001e == 0x00000014. Since they are not equal, it skips the rest of the instructions until it reaches Else. Else subtractions $s1 and $s2 ($s1 - $s2) and puts the value in $s0 which is 0x00000032. The addition is not performed since $s3 and $s4 are not equal, if they were then $s1 + $s2 would have been performed.



Value of comparison stored

Figure 25 - Faruqi\_2-7\_1.asm Code

Other values stored for comparison.

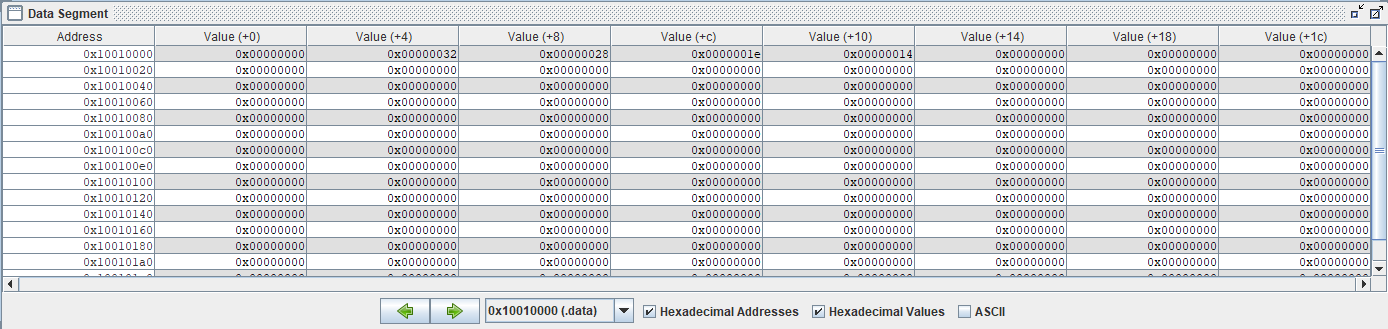
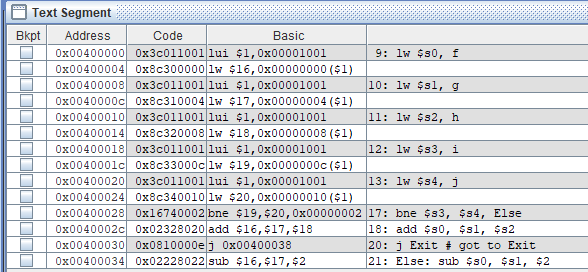


Figure 27 - Faruqi\_2-7\_1.asm Data Segment

Figure 26 - Faruqi\_2-7\_1.asm Register

BNE with else skip instruction

Figure 28 - Faruqi\_2-7\_1.asm Instructions

**main\_myadd.asm**

main\_myadd.asm is a representation of topics covered in chapter 2.8 which deal with procedures and nested procedures. This code represents the leaf\_eaxmple function which involves procedure calling and addition. For this function as seen in **Figure 29** there is a jump statement to go back to the calling routine.

In this example, 3 items need to be stored on the stack and so the stack pointer is subtracted by 12. Each value has an offset of 4 and so to get 3 it’s 3\*4 = 12. Then registers $t0, $t1, and $s0 are saved to be used later. As seen in **Figure 31** the next thing to save are the instructions themselves. Since this is a function, the instructions need to be saved so they can be used later. And each variable is assigned a register, and registers $t0, $t1, and $s0 store the operations that are performed on the values. Since there are only 3 instructions in the function, each register will store an instruction. Th return value is returned to a special return value register. Before the function ends the registers that were used are freed up and the space on the stack is added back to remove and values that were put on them.

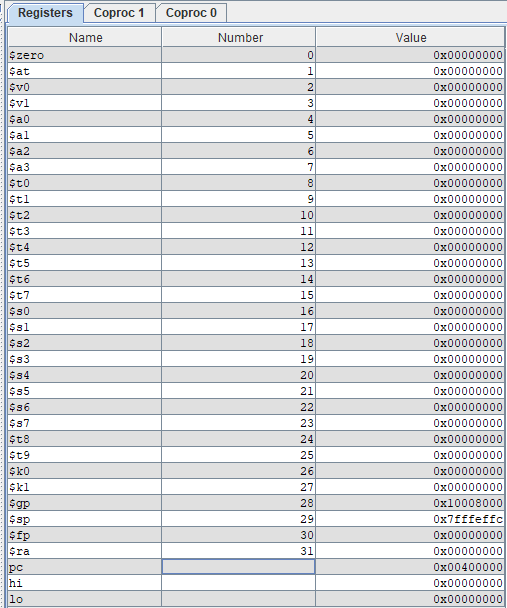
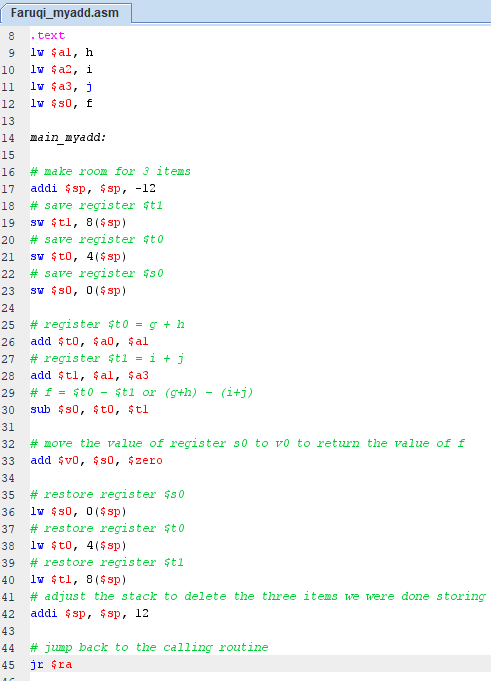
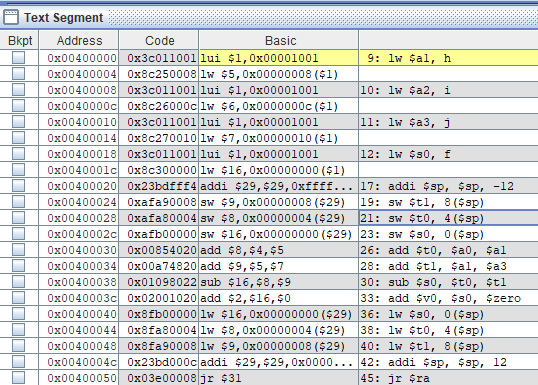
In the end a last jump statement is called to return to the returning address where the function was called. When the function is called now, it receives it’s values which get stored on the stack and the function is executed. It follows all the steps and goes through the instruction now that it has values to place in the registers. In the end it will again pass the value to the return value register and remove the values from all other registers and the stack.

Figure 30 - main\_myadd.asm Register



Stack pointer assigned

Figure 29 - main\_myadd.asm Code



Jump instruction to return back to call

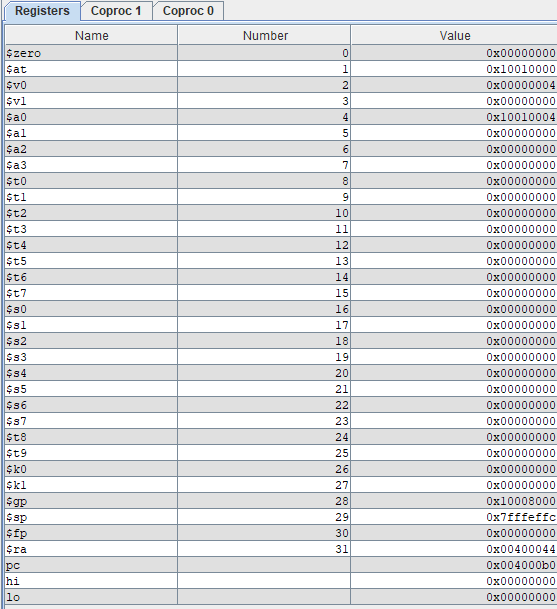
Figure 31 - main\_myadd.asm Instructions

**natural\_generator.asm**

natural\_generator takes the idea of procedures to another level by providing a main function which runs the function inside of itself. As seen in **Figure 32**, since the main function calls natural\_generator as it’s first instruction.

The computer jumps to the end and starts executing the instructions of natural\_generator. It jumps to line 50 where it first stored 4 words onto the stack, thus -16. Two of those are for the variable values themselves and the other two are the for calculations performed on them. First the stack is filled with the values of 4 registers $t0 -$t2 and $s0. Since they are all 0 there is no affect made. Now the registers must represent the values that are stored into the stack and later used. Since b is a static variable it will be stored onto memory later, until then it will be stored on the stack. $t1 stores the value of a = 1 (0x00000001), $t2 stores the value of the static variable b = -1which is gotten from memory (0xffffffff). Then according to the function statement, b = b +1 and since b = 0xffffffff, 0xffffffff + 0x00000001 = 0x00000000. And then the value is stored in memory since b is a static variable in big endian. $t0 now stores the value of b + a or $t1 + $t2 = 1. Later we need to return this value so $t0’s value is pushed onto $v0. Since the function is finished, the space is deallocated from every register that was used in the calculation and the space is returned to the stack using +12. As seen in **Figure 34**, the function ends with a jump statement to return to the area which called it.

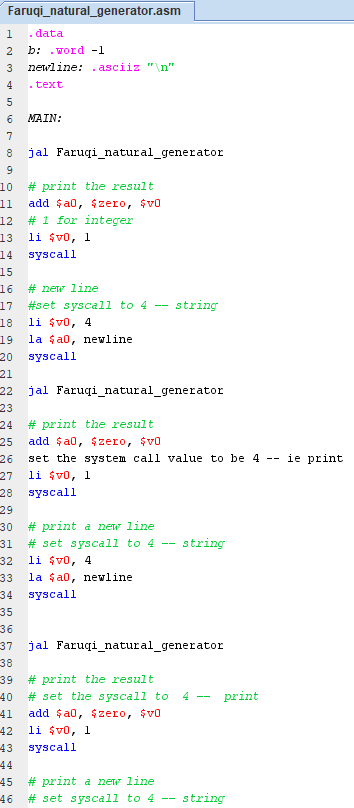
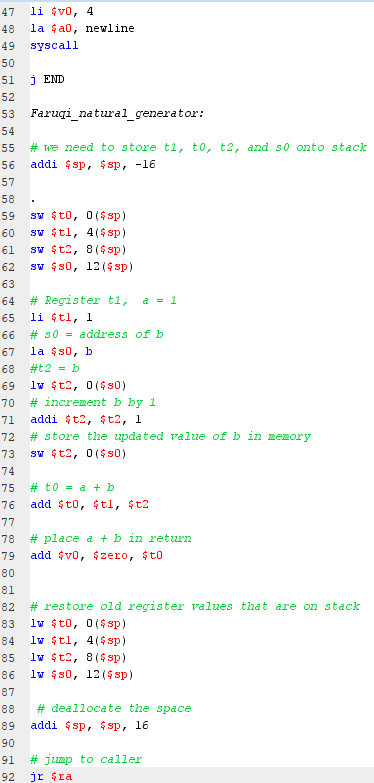
To bring back the return value from $v0, we add it to 0 and store it in $a0. Syscall is called to print the integer value, and then $v0 is set to 4 to indicate a new string needs to be printed. Since we are using $a0 to hold the value of print, it is set to newline, and then syscall run again to print the newline. After which natural\_generator is called again to run it’s instructions 2 more times. The only difference is that when it jumps down to function instructions the static variable b’s value is now 0, and throughout the function it changes to 1. The print sequence then becomes 1, 2, 3 as seen in **Figure 35**.



Return register and syscall value

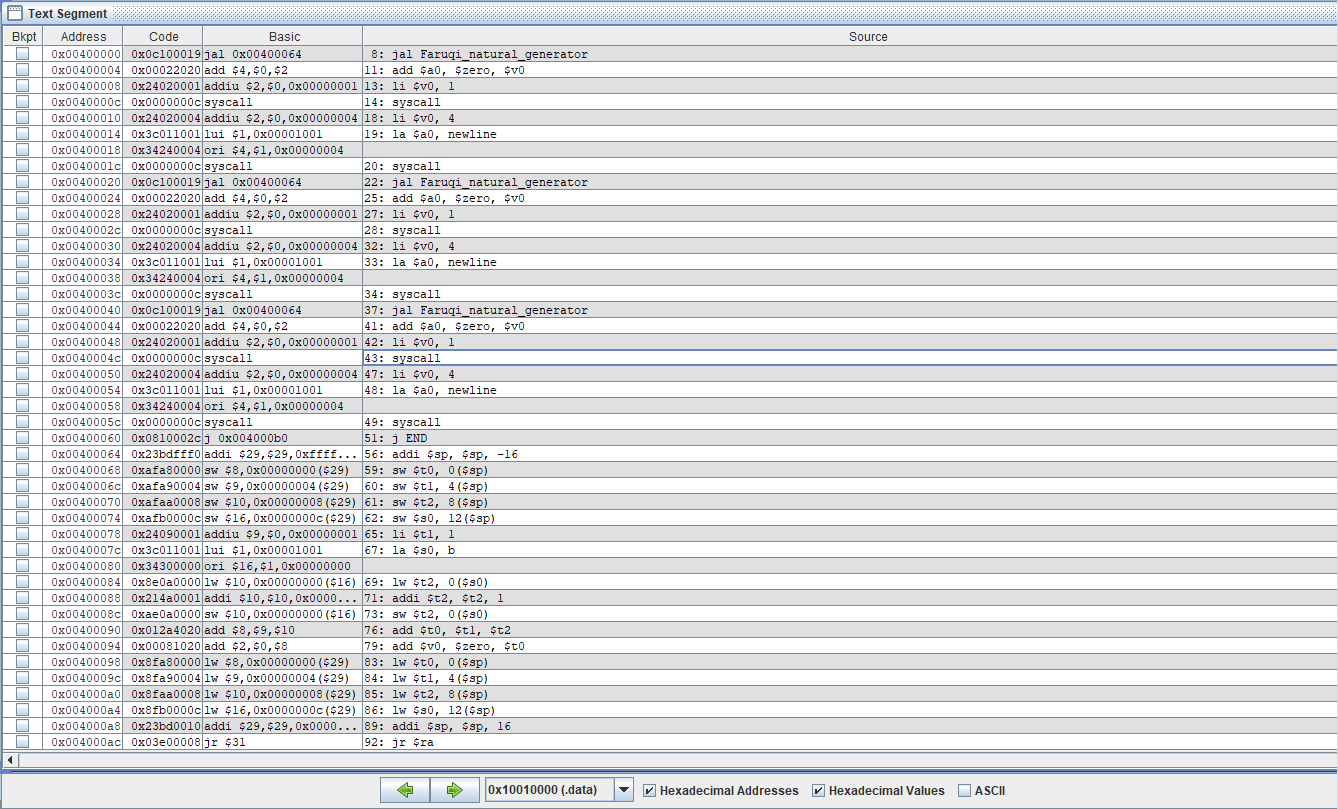
Figure 33 - natural\_generator.asm Register

Figure 32 - natural\_generator.asm Code



Values and calculations stored here

Stack pointer



Jump statement for function

Figure 3 - natural\_generator.asm Instructions

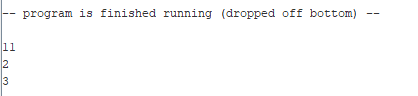
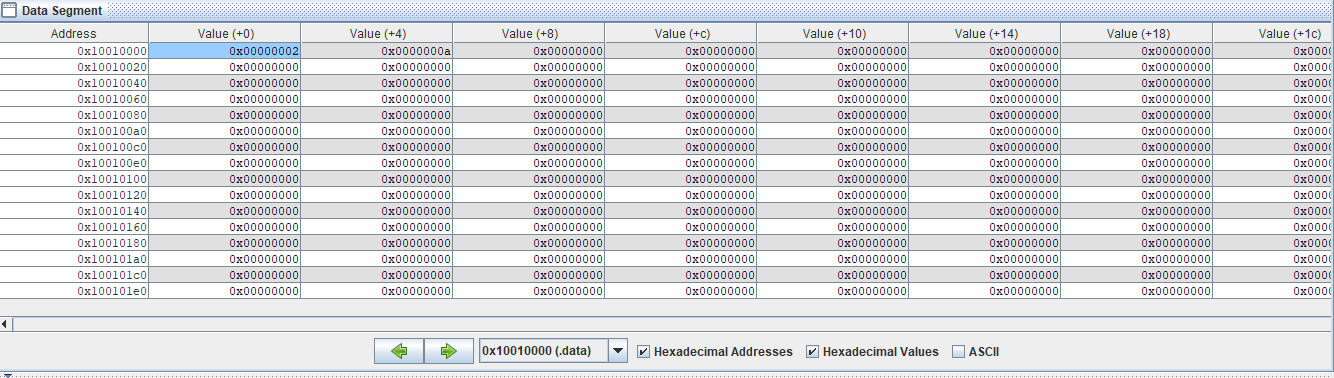


Figure 3 - natural\_generator.asm Output



Static variable b value stored in memory

Figure - natural\_generator.asm Data Entry