

Lab 03 – Worksheet

Name: Shaheer Qureshi, Areeba Izhar

ID: sq09647,
ai09625

Section: T1

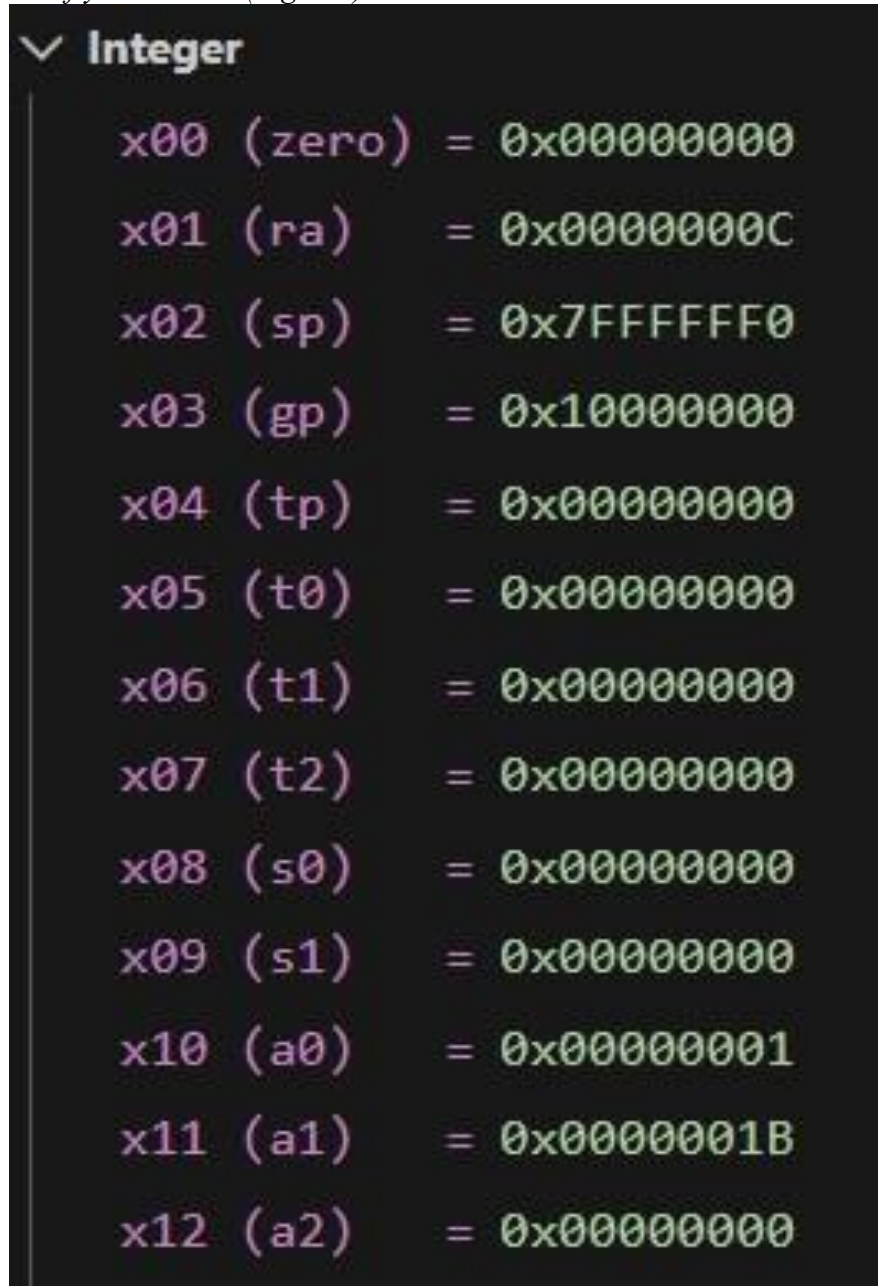
Note: Assumptions and logics should be explained separately in tasks after the task results.

Task 1

Provide appropriately commented codes (Mention question part before each part)

```
ASM Task1.s U X ASM Task1a.s
LabsCA-Git-Shaheer_Areeba > Lab03 > ASM Task
1  .text
2  .globl main
3  main:
4      addi x10, x0, 12
5      addi x11, x0, 15
6      jal x1, sum
7      addi x11, x10, 0
8      li x10, 1
9      ecall
10     j exit
11     sum:
12         add x10, x11, x10
13         jalr x0, 0(x1)
14     exit:
15         j end
16
17     end:
18         j end
```

Add screenshot of your results (register)



Function Logic:

In our main function, we first loaded value 12 in register x10 (a). Then we loaded value 15 in register x11 (b). the jal x1, sum statement jumps to sum function. It is saving the very next line address. When the line jumps to the sum function, it performs the sum operations by adding a and b and storing it in x10. This is useful since we need to RETURN the output value.

The jalr x0, 0(x1) is our return instruction that jumps to address stored in x1 (in the main function)

Then in main, the next statement to go to is addi x11, x10, 0 so it stores return output in x11. Then the ecall statement prints the output in terminal.

Task 2

Provide appropriately commented codes (Mention question part before each part)

```
absCA-Git-Shaheer_Areeba > Lab03 > ASM Task2.s
1  .text
2  .globl main
3  main:
4      li x10, 5          #g
5      li x11, 4          #h
6      li x12, 3          #i
7      li x13, 2
8
9      jal x1, leaf_example
10     addi x11, x10, 0 #ecall always prints x11
11     li x10, 1
12     ecall
13     j exit
14
15     leaf_example:
16         addi x2,x2, -12 #assign stack
17         sw x18, 0(x2)
18         sw x19, 4(x2)
19         sw x20, 8(x2)
20
21         add x18, x10, x11    #temp1 = g + h
22         add x19, x12, x13    #temp2 = i + j
23         sub x20, x18, x19    #f = temp1 - temp2
24
25         addi x10, x20, 0 #return f
26
27
28         lw x18, 0(x2)
29         lw x19, 4(x2)
30         lw x20, 8(x2)
31         addi x2, x2, 12 #free space for stack
32         jalr x0, 0(x1)
33
34     exit:
35         j end
36     end:
37         j end
```

note: Since our systems weren't able to process long long integer, instead of taking 8 offset, we choose offset 4 (which is long int)

Add screenshot of your results

Integer		
x00	(zero)	= 0x00000000
x01	(ra)	= 0x00000014
x02	(sp)	= 0x7FFFFFF0
x03	(gp)	= 0x10000000
x04	(tp)	= 0x00000000
x05	(t0)	= 0x00000000
x06	(t1)	= 0x00000000
x07	(t2)	= 0x00000000
x08	(s0)	= 0x00000000
x09	(s1)	= 0x00000000
x10	(a0)	= 0x00000001
x11	(a1)	= 0x00000004
x12	(a2)	= 0x00000003
x13	(a3)	= 0x00000002
x14	(a4)	= 0x00000000
x15	(a5)	= 0x00000000
x16	(a6)	= 0x00000000
x17	(a7)	= 0x00000000
x18	(s2)	= 0x00000000
x19	(s3)	= 0x00000000
x20	(s4)	= 0x00000000
x21	(s5)	= 0x00000000
x22	(s6)	= 0x00000000

Code Logic:

For this code, we first declared our g,h,i,j with sample values. The next statement, jal x1, leaf_example allows us to jump the the leaf_example function we made, and this later will save the return address in x1. In the leaf_example function, we declared a stack with space - 12 and stack pointer pointing to it, (sp -> x2). Each stack space (4 byte offset) is storing space for the operations to perform. The output is stored in x20 and this final result is stored in x10. Since the address is safe, we then deallocate the stack to 'free space' and then the jalr x0, 0(x1) is used, to return to main

- Our assumption was using 32-bit RISC-V as we used sw, lw (4 bytes) and adjusted our stack by 12 bytes for 3 registers.

Task 3 Provide appropriately commented codes (Mention question part before each part)

```
ASM Task2.s  ASM Task4.s  ASM Task3.s  X
Users > sheeru > Downloads > ASM Task3.s
1  .text
2  .globl main
3  main:
4      li x5, 0x100
5      li x6, 10
6      sw x6, 0(x5) # 0 at address 0x100
7      li x6, 20
8      sw x6, 4(x5)
9      li x6, 30
10     sw x6, 8(x5)
11     li x6, 40
12     sw x6, 12(x5)
13     li x6, 50
14     sw x6, 16(x5) # 50 at address 0x10C
15     li x10, 0x100
16     li x11, 2 #k parameter 2
17
18     jal x1, swap
19     li x10, 0x100
20     lw x11, 12(x10)
21     li x10, 1
22     ecall
23     j exit
24
25 swap:
26     li x28, 4
27     mul x6, x11, x28 #calculate offset k*4 [slli x6, x11, 2]
28     add x6, x10, x6
29     #temp var
30     lw x5, 0(x6)
31     lw x7, 4(x6)
32     sw x7, 0(x6)
33     sw x5, 4(x6)
34     jalr x0, 0(x1)
35
36 exit:
37     j end
38 end:
39     j end
```

Add screenshots of your results

Integer		
x00 (zero)	=	0x00000000
x01 (ra)	=	0x00000038
x02 (sp)	=	0x7FFFFFF0
x03 (gp)	=	0x10000000
x04 (tp)	=	0x00000000
x05 (t0)	=	0x0000001E
x06 (t1)	=	0x00000108
x07 (t2)	=	0x00000028
x08 (s0)	=	0x00000000
x09 (s1)	=	0x00000000
x10 (a0)	=	0x00000001
x11 (a1)	=	0x0000001E
x12 (a2)	=	0x00000000
x13 (a3)	=	0x00000000
x14 (a4)	=	0x00000000
x15 (a5)	=	0x00000000
x16 (a6)	=	0x00000000
x17 (a7)	=	0x00000000
x18 (s2)	=	0x00000000
x19 (s3)	=	0x00000000
x20 (s4)	=	0x00000000
x21 (s5)	=	0x00000000
x22 (s6)	=	0x00000000
x23 (s7)	=	0x00000000
x24 (s8)	=	0x00000000
x25 (s9)	=	0x00000000
x26 (s10)	=	0x00000000
x27 (s11)	=	0x00000000
x28 (t3)	=	0x00000004
x29 (t4)	=	0x00000000
x30 (t5)	=	0x00000000
x31 (t6)	=	0x00000000

BLACKBOX				
Address	+0	+1	+2	+3
0x00000118	0	0	0	0
0x00000114	0	0	0	0
0x00000110	50	0	0	0
0x0000010C	30	0	0	0
0x00000108	40	0	0	0
0x00000104	20	0	0	0
0x00000100	10	0	0	0
0x000000FC	0	0	0	0
0x000000F8	0	0	0	0
0x000000F4	0	0	0	0
0x000000F0	0	0	0	0
0x000000EC	0	0	0	0
0x000000E8	0	0	0	0

Address: Up Down
 Jump to: -- choose --
 Display Format: Decimal
 Bytes per Row: 4

Code Logic:

First our code populates an array with a base address of 0x100. Then each value with an offset of 4 bytes is stored in the array.

Then we load the base address of the array as the first argument for the swap function (x10 respectively)

The second argument is k (we stored a dummy value of 2) in x11.

Since its an array and to get the values, we need to set its offset and for that we stored a value of 4 (4 bytes) in x28 and multiplied it by index k. ($2 * 4 = 8$ bytes offset)

Then the add statement (add x6, x10, x6) adds this offset to base address and now the x6 register will point at kth index of v[]

Then we used the simple swapping logic to load the value of v[k] in temp reg (x5)

Then then value of v[k+1] in temp reg (x7). And then changed the valued stored in these registers (interchanged with each other)

The jalr x0, 0(x1) statement helps to return to the main function, and lw x11, 12(x10) is what loads value at 0x10C (val 40 in our array) into x11. And ecall prints it.

- Our assumption was using 32-bit RISC-V as we used sw, lw (4 bytes) and adjusted our stack by 12 bytes for 3 registers.

Task 4 Provide appropriately commented codes

```
ASM Task1.s    ASM Task2.s    ASM Task3.s    ⋮ ||
LabsCA-Git-Shaheer_Areeba > Lab03 > ASM Task4.s
1  .text
2  .globl main
3  main:
4      li x5, 0x200
5      li x6, 's'
6      sb x6, 0(x5) # 's' at address 0x200
7      li x6, 'h'
8      sb x6, 1(x5)
9      li x6, 'a'
10     sb x6, 2(x5)
11     li x6, 'h'
12     sb x6, 3(x5)
13     li x6, 'e'
14     sb x6, 4(x5)
15     li x6, 'e'
16     sb x6, 5(x5)
17     li x6, 'r'
18     sb x6, 6(x5)
19     li x6, 0
20     sb x6, 7(x5) #null
21
22     li x10, 0x100
23     li x11, 0x200
24
25     jal x1, strcpy
26     j exit
27
28     strcpy:
29         addi sp, sp, -4 #create stack space
30         sw x19, 0(x2) #i save i
31         li x19, 0
```

```
32 |
33 | loop1:
34 |     add x5, x19, x11 #get y[i]
35 |     lb x6, 0(x5)      #load y[i]
36 |     add x7, x19, x10 #get x[i]
37 |     sb x6, 0(x7)      #store in x[i]
38 |
39 |     beq x6, x0, endloop #if y[i] == null, end loop
40 |
41 |     addi x19, x19, 1 #i++
42 |     j loop1
43 | endloop:
44 |     lw x19, 0(x2)
45 |     addi x2, x2, 4 #empty stack
46 |     jalr x0, 0(x1) #return to caller
47 |
48 | exit:
49 |     j end
50 |
51 | end:
52 |     j end
53 |
```

Add screenshots of your results

The image shows two screenshots of a memory viewer tool. The top screenshot shows a memory dump starting at address 0x00000218, with a search for 'ersh' at address 0x200. The bottom screenshot shows a memory dump starting at address 0x00000118, with a search for 'ersh' at address 0x100. Both screenshots show a table of memory addresses and their corresponding byte values, with a search bar and navigation buttons.

Memory Viewer Screenshot 1 (Top):

Address	+0	+1	+2	+3
0x00000218	0x00	0x00	0x00	0x00
0x00000214	0x00	0x00	0x00	0x00
0x00000210	0x00	0x00	0x00	0x00
0x0000020C	0x00	0x00	0x00	0x00
0x00000208	0x00	0x00	0x00	0x00
0x00000204	e	e	r	0x00
0x00000200	s	h	a	h
0x000001FC	0x00	0x00	0x00	0x00
0x000001F8	0x00	0x00	0x00	0x00
0x000001F4	0x00	0x00	0x00	0x00
0x000001F0	0x00	0x00	0x00	0x00
0x000001EC	0x00	0x00	0x00	0x00
0x000001E8	0x00	0x00	0x00	0x00

Address: 0x200 Up Down
Jump to: -- choose --
Display Format: ASCII
Bytes per Row: 4

Memory Viewer Screenshot 2 (Bottom):

Address	+0	+1	+2	+3
0x00000118	0x00	0x00	0x00	0x00
0x00000114	0x00	0x00	0x00	0x00
0x00000110	0x00	0x00	0x00	0x00
0x0000010C	0x00	0x00	0x00	0x00
0x00000108	0x00	0x00	0x00	0x00
0x00000104	e	e	r	0x00
0x00000100	s	h	a	h
0x000000FC	0x00	0x00	0x00	0x00
0x000000F8	0x00	0x00	0x00	0x00
0x000000F4	0x00	0x00	0x00	0x00
0x000000F0	0x00	0x00	0x00	0x00
0x000000EC	0x00	0x00	0x00	0x00
0x000000E8	0x00	0x00	0x00	0x00

Address: 0x100 Up Down
Jump to: -- choose --
Display Format: ASCII
Bytes per Row: 4

Integer		
x00 (zero)	=	0x00000000
x01 (ra)	=	0x00000050
x02 (sp)	=	0x7FFFFFF0
x03 (gp)	=	0x10000000
x04 (tp)	=	0x00000000
x05 (t0)	=	0x00000207
x06 (t1)	=	0x00000000
x07 (t2)	=	0x00000107
x08 (s0)	=	0x00000000
x09 (s1)	=	0x00000000
x10 (a0)	=	0x00000100
x11 (a1)	=	0x00000200
x12 (a2)	=	0x00000000
x13 (a3)	=	0x00000000
x14 (a4)	=	0x00000000
x15 (a5)	=	0x00000000
x16 (a6)	=	0x00000000
x17 (a7)	=	0x00000000
x18 (s2)	=	0x00000000
x19 (s3)	=	0x00000000
x20 (s4)	=	0x00000000
x21 (s5)	=	0x00000000

Code logic:

The 'main' part populates the character array char by char at 0x200 using sb (store byte)
It includes a null terminator at the end.

Reg x11 stores the address of the array (0x200) and x10 loads the destination array (0x100)
[our parameters for the strcpy function]

jal x1, strcpy jumps to the function and stores address in x1 (return address)

Since x19 used in the strcpy is a saved register, the stack pointer (sp) is decremented to create
space (sw x19, 0(x2)) to save old value of x19.

It then sets x19 to 0.

The loop1 (copy loop) adds current index x19 to base address of x11 and x10 to find the byte
locations and loads a single byte into x6 and stores it in destination through [sb x6, 0(x7)]

Beq statement checks for null terminator to end loop and if its not, the index I keeps
incrementing (addi x19, x19, 1) by 1 and loop continues.

At the end, the original value of x19 is loaded from stack and sp is returned to its original
position.

- Our assumption was using 32-bit RISC-V as we used sw, lw (4 bytes) and adjusted
our stack by 12 bytes for 3 registers.

Lab 03 – Introduction to RISC V Assembly (Jumps and Returns)

Assessment Rubrics

Points Distribution

Task No.	LR2 Code	LR5 Results	AR7 Report Submission and Git Upload
Task 1	/10	/05	/10 & /10
Task 2	/10	/10	
Task 3	/10	/10	
Task 4	/15	/10	
Total Points	/100 Points		
CLO Mapped	CLO 2		

For description of different levels of the mapped rubrics, please refer the provided Lab Evaluation Assessment Rubrics and Affective Domain Assessment Rubrics.

#	Assessment Elements	Level 1: Unsatisfactory Points 0-1	Level 2: Developing Points 2	Level 3: Good Points 3	Level 4: Exemplary Points 4
LR2	Program/Code / Simulation Model/ Network Model	Program/code/simulation model does not implement the required functionality and has several errors. The student is not able to utilize even the basic tools of the software.	Program/code/simulation model/network model has some errors and does not produce completely accurate results. Student has limited command on the basic tools of the software.	Program/code/simulation model/network model gives correct output but not efficiently implemented or implemented by computationally complex routine.	Program/code/simulation /network model is efficiently implemented and gives correct output. Student has full command on the basic tools of the software.
LR5	Results & Plots	Figures/ graphs / tables are not developed or are poorly constructed with erroneous results. Titles, captions, units are not mentioned. Data is presented in an obscure manner.	Figures, graphs and tables are drawn but contain errors. Titles, captions, units are not accurate. Data presentation is not too clear.	All figures, graphs, tables are correctly drawn but contain minor errors or some of the details are missing.	Figures / graphs / tables are correctly drawn and appropriate titles/captions and proper units are mentioned. Data presentation is systematic.
AR9	Report	All the in-lab tasks are not included in report and / or the report is submitted too late.	Most of the tasks are included in report but are not well explained. All the necessary figures / plots are not included. Report is submitted after due date.	Good summary of most the in-lab tasks is included in report. The work is supported by figures and plots with explanations. The report is submitted timely.	Detailed summary of the in-lab tasks is provided. All tasks are included and explained well. Data is presented clearly including all the necessary figures, plots and tables.