CENG3420 - Computer Organization & Design
Lab1 Report

1155193237 - Yu Ching Hei
email: chyu2@cse.cuhk.edu.hk

Date: February 22, 2024

Question 1. Write a RISC-V assembly program step by step as shown below:

1. Define three variables var1, var2 and var3 which will be loaded from terminal using syscall.
2. Increase var1 by 3, multiply var2 by 2.
3. increase var3 by var1 + var2.
4. print var1, var2 and var3 to terminal using syscall.

Answer:

First, I declared a variable of datatype ".ascii" and named it as endl, which stored the constant string literal '\n'.

Then the program will require the user to input 3 integers by loading immediate 5 to change the system call mode into readInt mode. Then issue a system call to request for input. After the system call, the integer would be stored at a0 as return value. Then I perform the data copy from a0 to t0 by mv a0, t0 and the same for t1 and t2 as well. Then I perform the arithmetic to the data. By running addi t0, t0, 3, I can add a hard code constant 3 to t0, then store back to t0 (t0 = t0 + 3).

Since mul instruction does not accept immediate, so I loaded a register t3 with the multiplier "2" using the li t3, 2.

Then I perform the multiplication by multiplying t1 with t3 and then stored back to t3 (t1 = t1 * t3) by doing mul t1, t1, t3. Followed by the last modification to the numbers, since we cannot sum a value with 2 other value at the same time, to perform var3 = var3 + var1 + var2, I have to do the addition twice.

First I added t0 to t2 then stored it back to t2, then I add another t1 to t2 and then store it back to t2. For a cleaner code, I wrote a function for printing prtNL for printing a new line character after printing each variable.

In the prtNL function, I first changed the mode of system call to 4 which is printString from address. Then I load the address from memory by la a0, endl. Then I issue a system call and return to the return address.

Now I start to print the variable to the console one by one. I change the system call mode to 1 which is printInt, then copy the data from the registers that store the data representing var1, var2 and var3 respectively to the a0 register which is the function argument register.

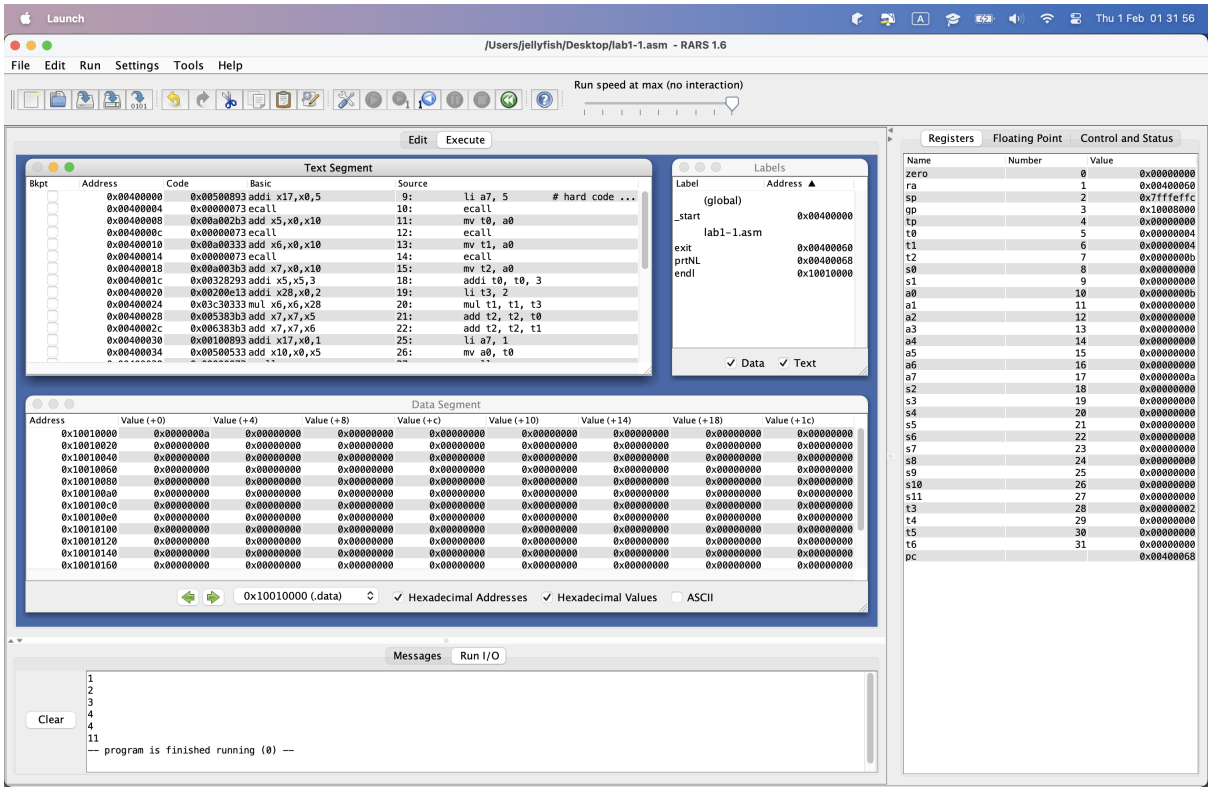
Finally, I loaded register a7 with 10, which changes the system call mode to exit the program with code 0.

Table 1: Main Code for Question 1

Input Integer	Add and Multiply	Print Integer & Exit	New Line Function

`# read int`
`li a7, 5`
`ecall`
`mv t0, a0`
`ecall`
`mv t1, a0`
`ecall`
`mv t2, a0`
	`# modify integer`
	`addi t0, t0, 3`
	`li t3, 2`
	`mul t1, t1, t3`
	`add t2, t2, t0`
	`add t2, t2, t1`
		`# out int`
		`li a7, 1`
		`mv a0, t0`
		`ecall`
		`jal prtNL`
		
		`li a7, 1`
		`mv a0, t1`
		`ecall`
		`jal prtNL`
		
		`li a7, 1`
		`mv a0, t2`
		`ecall`
		
		`jal exit`
			`prtNL:`
			`li a7, 4`
			`la a0, endl`
			`ecall`
			`ret`

Figure 1: Console Results



Question 2. An array array1 contains the sequence -1 22 8 35 5 4 11 2 1 78, each element of which is .word. Rearrange the element order in this array such that,

- All the elements smaller than the 3rd element (i.e. 8) are on the left of it
- All the elements bigger than the 3rd element (i.e. 8) are on the right of it

Answer:

Fisrt I have created a two variables in the memory, namely:

- array1: a .word array that contains 10 elements given in the question
- space: a .ascii variable that store the delimiter character ‘ ’ for printing

8 is the pivot, -1, 5, 4, 2, 1 are the element smaller than 8 in the array, and we have 22, 35, 11, 78 larger than 8. Detail are shown in the following parts.

1. I swapped the pivot(8) with the final element in the array
2. Then I call the PARTITION function to do the partition by passing in 3 parameter to the function namely array address(s0), lowest index(s1), highest index(s2)
3. In the PARTITION function, it takes in 3 parameter, they are s0(Array1 location), s1(permanent storage of lowest position) and s2(permanent storage of the highest position). It scans for any element that is smaller than the pivot, then swap with first element that is greater than the pivot
4. After finished one round of searching and swapping, the loop end and swap the pivot with the first element that is larger than the pivot
5. Print the partitioned array
6. Exit the program with exit code 0

Table 2: Main Code for Question 2

<table>
<tr><th>Function</th><th>Code</th></tr>
<tr><td>PARTITION function</td><td>

```
PARTITION:      #partition(a(s0), lo(s1), hi(s2))
        addi    sp, sp, -4
        sw      ra, 0(sp)

        slli    t0, s2, 2
        add     t0, t0, s0
        lw      t0, 0(t0)       # pivot = A[hi]
        addi    t1, s1, -1      # i = lo - 1
        mv      t2, s1          # j = lo

LOOP1:
        bgt     t2, s2, LOOPDONE
        slli    t3, t2, 2
        add     t3, t3, s0
        lw      t4, 0(t3)       # temp = A[j]
        bge     t4, t0, ELEMENT_GTE_PIVOT

        addi    t1, t1, 1       # i++
        addi    sp, sp, -4
        sw      t1, 0(sp)

        slli    t1, t1, 2
        add     t1, t1, s0
        lw      t5, 0(t1)
        sw      t5, 0(t3)       # A[i] = A[j]
        sw      t4, 0(t1)       # A[j] = temp

        lw      t1, 0(sp)
        addi    sp, sp, 4
ELEMENT_GTE_PIVOT:
        addi    t2, t2, 1       # j++
        j       LOOP1
LOOPDONE:
        addi    t3, t1, 1       # i++
        mv      a0, t3          # save i + 1 for return
        slli    t3, t3, 2
        add     t3, t3, s0
        lw      t4, 0(t3)       # temp = A[i + 1]

        slli    t5, s2, 2
        add     t5, t5, s0
        lw      t6, 0(t5)       # store A[hi]

        sw      t6, 0(t3)       # A[i+1] = A[hi]
        sw      t4, 0(t5)       # A[hi] = A[i+1]

        lw      ra, 0(sp)
        addi    sp, sp, 4
        jr ra
```

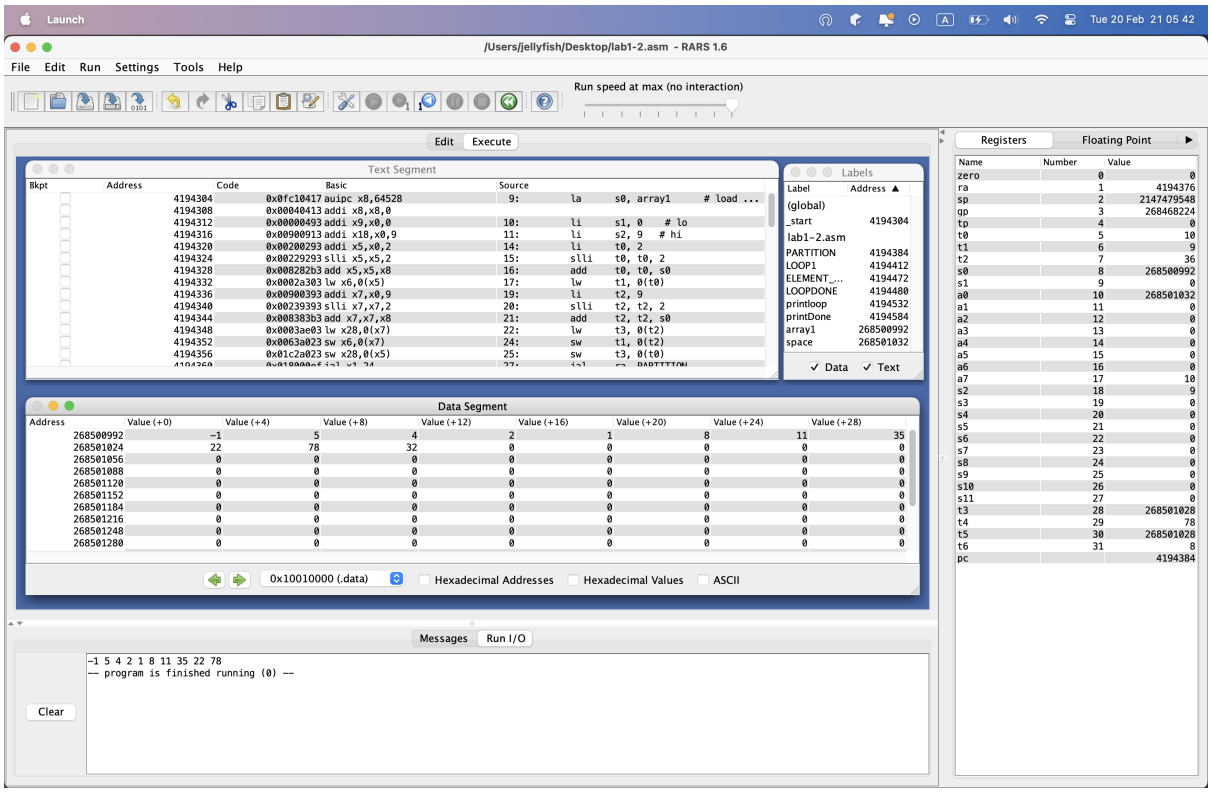
</td></tr>
<tr><td>printLoop (A looping function to store the elements in the temp array into the original array and print them out one by one)</td><td>

```
printloop:
        bgt     t0, t1, printDone
        slli    t2, t0, 2
        add     t3, t2, s0
        lw      t4, 0(t3)
        mv      a0, t4
        li      a7, 1
        ecall
        li      a7, 4
        la      a0, space
        ecall
        addi    t0, t0, 1
        j       printloop

printDone:
        ret
```

</td></tr>
</table>

Figure 2: Console Results



Question 3. Implement Quicksort *w.r.t.* the following array in ascending order with RISC-V assembly programming:
In the first line, you will be given an integer n ($2 < n < 100$), representing the array size. In the second line, a sequence of n integers will be provided.

Answer:

In this question, I have created 2 variables in memory, namely:

- array: a .word array for storing the integer input and sorting
- space: a .ascii variable that store the delimiter character ‘ ’ for printing

I solve this question with the follwoing stpes:

1. For the input part, the first input will be the number of array elements, let's call it n. (n - 1) is the index of the last element of the array, at the same time it could also be used as a looping bound for both input and output.
2. For input and output, I have used the same looping structure, but by passing in a0 = -1 (a0 ¡ 0) and a0 = 0, the function of the loop will be changed to input and output respectively since I have set a0 as a parameter and determine whether it should branch to input action or output action. This could help reduce the redundancy of similar code and simplify the code.
3. For the quicksort part, I have done the typical way of quicksort which is basically if (low < high) then do the partition and then do the quicksort for the left and right part of the pivot respectively. It takes in 3 parameters, namely s0(array location), a1(low w.r.t each partition), a2(hi w.r.t each partition), they are further passed into the PARTITION function under the quicksort function.
4. I have reused the same PARTITION function as done in Lab1-2, but in this question, I have changed the parameter from s0(array location), s1(permanent storage of low), s2(permanent storage of hi) into s0(array location), a1(low w.r.t each partition), a2(hi w.r.t each partition). The main difference here is that in question 2, since we are not going to do multiple times of partitioning, so it is good enough to just use the low and high position of the whole array. But for quicksort, each recursion will have a new partition action. If I still use s1(lo) and s2(hi) as the parameter, there would be no changes. Therefore, I have change the lo and hi to be a1 and a2, which are changed depends on the subarray that the quicksort is currently processing. It is not necessary to change s0 as well since s0 is just for the address of the array and it is never changed in each recursive call of quicksort. It scans for any element that is smaller than the pivot, then swap with first element that is greater than the pivot, finally swap the pivot with the first element that is larger than the pivot.
5. After recursive call of quicksort functions, the array got partitioned into 10 subarrays that are of length of 1 element and they are sorted since the smaller element will go to the left side of the pivot.
6. Finally, print the sorted array separated by “ ” and exit the program with exit code 0.

*The console input is separated by ‘\n’ while the output is the final line separated by ‘ ’ character.

Table 3: Main Code for Question 3

Function	Code

Loop handeling input and output	`loop_begin:`
	`bgt t0, s2, loop_end`
	`lw a0, 4(sp)`
	`blt a0, x0, input_action`
	`beqz a0, output_action`
	
	`input_action:`
	`li a7, 5`
	`ecall`
	`slli t2, t0, 2`
	`add t2, t2, s0`
	`sw a0, 0(t2)`
	`j loop_iterate`
	
	`output_action:`
	`slli t2, t0, 2`
	`add t2, t2, s0`
	`lw a0, 0(t2)`
	`li a7, 1`
	`ecall`
	`li a7, 4`
	`la a0, space`
	`ecall`
	`j loop_iterate`
	
	`loop_iterate:`
	`addi t0, t0, 1`
	`j loop_begin`
	
	`loop_end:`
	`li t0, 0`
	`lw ra, 0(sp)`
	`addi sp, sp, 4`
	`jr ra`

QUICKSORT	`QUICKSORT:  # QUICKSORT(A(s0), lo(a1), hi(a2))`
	`addi sp, sp, -4`
	`sw ra, 0(sp)`
	
	`bgt a1, a2, ENDSORT`
	`jal ra, PARTITION  # return pivot(a0)`
	
	`addi sp, sp, -12`
	`sw a2, 8(sp)`
	`sw a1, 4(sp)`
	`sw a0, 0(sp)`
	`addi a2, a0, -1`
	`jal ra, QUICKSORT`
	`lw a0, 0(sp)`
	`lw a1, 4(sp)`
	`lw a2, 8(sp)`
	`addi sp, sp, 12`
	
	`addi sp, sp, -12`
	`sw a2, 8(sp)`
	`sw a1, 4(sp)`
	`sw a0, 0(sp)`
	`addi a1, a0, 1`
	`jal ra, QUICKSORT`
	`lw a0, 0(sp)`
	`lw a1, 4(sp)`
	`lw a2, 8(sp)`
	`addi sp, sp, 12`
	
	`ENDSORT:`
	`lw ra, 0(sp)`
	`addi sp, sp, 4`
	`jr ra`

Table 4: Main Code for Question 3(continue)

Function	Code

PARTITION	`PARTITION:  #partition(a(s0), lo(a1), hi(a2))`
	`addi sp, sp, -4`
	`sw ra, 0(sp)`
	`slli t0, a2, 2`
	`add t0, t0, s0`
	`lw t0, 0(t0)  # pivot = A[hi]`
	`addi t1, a1, -1  # i = lo - 1`
	`mv t2, a1  # j = lo`
	
	`LOOP1:`
	`bgt t2, a2, LOOPDONE`
	`slli t3, t2, 2`
	`add t3, t3, s0`
	`lw t4, 0(t3)  # temp = A[j]`
	`bge t4, t0, ELEMENT_GTE_PIVOT`
	
	`addi t1, t1, 1  # i++`
	`addi sp, sp, -4`
	`sw t1, 0(sp)`
	
	`slli t1, t1, 2`
	`add t1, t1, s0`
	`lw t5, 0(t1)`
	`sw t5, 0(t3)  # A[i] = A[j]`
	`sw t4, 0(t1)  # A[j] = temp`
	
	`lw t1, 0(sp)`
	`addi sp, sp, 4`
	`ELEMENT_GTE_PIVOT:`
	`addi t2, t2, 1  # j++`
	`j LOOP1`
	`LOOPDONE:`
	`addi t3, t1, 1  # i++`
	`mv a0, t3  # save i + 1 for return`
	`slli t3, t3, 2`
	`add t3, t3, s0`
	`lw t4, 0(t3)  # temp = A[i + 1]`
	
	`slli t5, a2, 2`
	`add t5, t5, s0`
	`lw t6, 0(t5)  # store A[hi]`

Figure 3: Console Results(Output: Last line)