

CENG3420 - Computer Organization & Design

Lab1 Report

1155193237 - Yu Ching Hei
email: chy2@cse.cuhk.edu.hk

Date: February 20, 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.

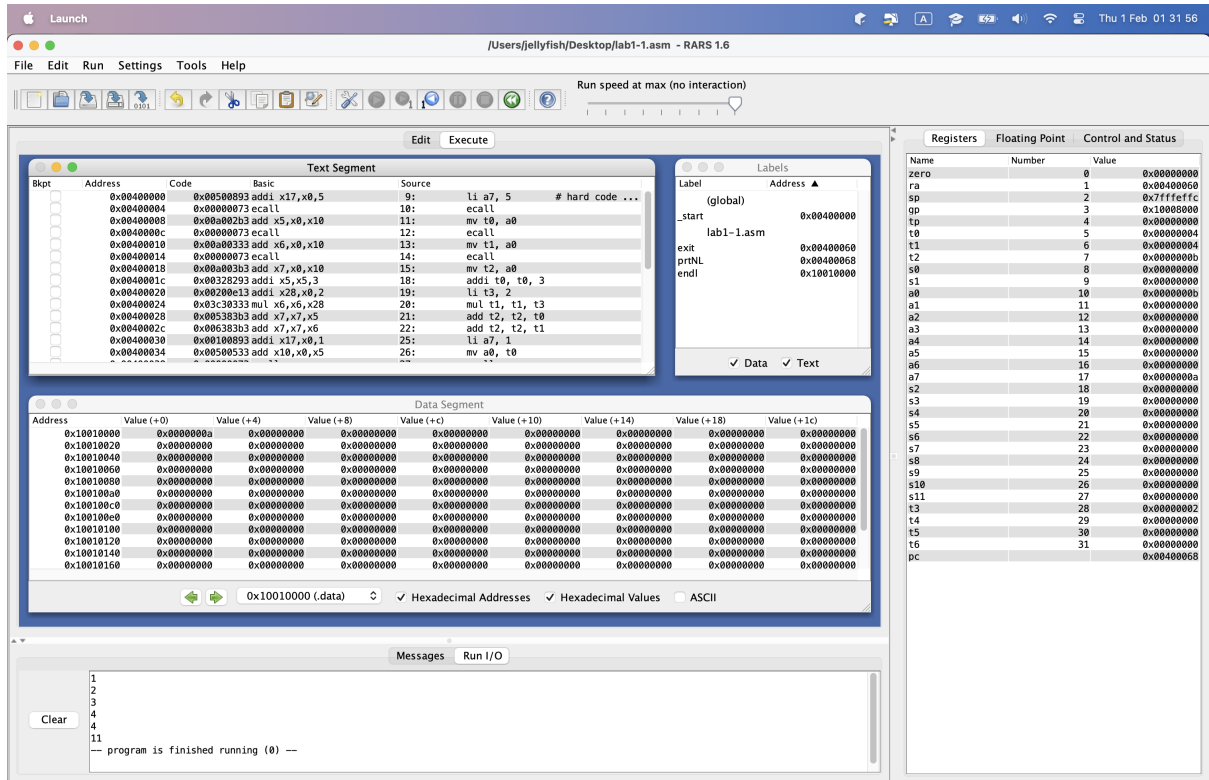
Now I start to print the variable to the console one by one. I change the system call mode to 1 which is printInt, then copy the data from the registers that store the data representing var1, var2 and var3 respectively to the a0 register which is the function argument register.

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
<pre># read int li a7, 5 ecall mv t0, a0 ecall mv t1, a0 ecall mv t2, a0</pre>	<pre># modify integer addi t0, t0, 3 li t3, 2 mul t1, t1, t3 add t2, t2, t0 add t2, t2, t1</pre>	<pre># 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</pre>	<pre>prtNL: li a7, 4 la a0, endl ecall ret</pre>

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

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

Figure 2: Console Results

The screenshot displays the RARS 1.6 emulator interface. The main window shows the assembly code for a program named 'lab1-2.asm'. The code is organized into a table with columns for Address, Code, Basic, and Source. The assembly instructions include various MIPS instructions like 'auipc', 'addi', 'li', 'slli', 'add', 'lw', 'sw', and 'space'. The 'Data Segment' table below the code shows memory addresses and their corresponding values. The 'Labels' table on the right lists labels and their addresses. The 'Registers' table on the far right shows the current values of MIPS registers. The console at the bottom displays the output of the program, which is a sequence of numbers: -1 5 4 2 1 8 11 35 22 78. The console also indicates that the program is finished running.

Text Segment

Bkpt	Address	Code	Basic	Source
	4194304	0x0fc10417	auipc x8,64528	9: la s0, array1 # load ...
	4194308	0x00040413	addi x8,x8,0	
	4194312	0x00000493	addi x9,x8,0	10: li s1, 0 # lo
	4194316	0x00000913	addi x18,x8,9	11: li s2, 9 # h1
	4194320	0x00200293	addi x5,x8,2	14: li t0, 2
	4194324	0x00229293	slli x5,x5,2	15: slli t0, t0, 2
	4194328	0x00022023	add x5,x5,x8	16: add t0, t0, s0
	4194332	0x0002a303	lw x6,0(x5)	17: lw t1, 0(t0)
	4194336	0x00000393	addi x7,x8,9	19: li t2, 9
	4194340	0x00239393	slli x7,x7,2	20: slli t2, t2, 2
	4194344	0x00033033	add x7,x7,x8	21: add t2, t2, s0
	4194348	0x0003a033	lw x28,0(x7)	22: lw t3, 0(t2)
	4194352	0x0063a023	sw x6,0(x7)	24: sw t1, 0(t2)
	4194356	0x01c2a023	sw x28,0(x5)	25: sw t3, 0(t0)
	4194360	0x00000000	li x1,0	27: li t4, 0

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+12)	Value (+16)	Value (+20)	Value (+24)	Value (+28)
268500992	-1	5	4	2	1	8	11	35
268501024	22	78	32	0	0	0	0	0
268501056	0	0	0	0	0	0	0	0
268501088	0	0	0	0	0	0	0	0
268501120	0	0	0	0	0	0	0	0
268501152	0	0	0	0	0	0	0	0
268501184	0	0	0	0	0	0	0	0
268501216	0	0	0	0	0	0	0	0
268501248	0	0	0	0	0	0	0	0
268501280	0	0	0	0	0	0	0	0

Labels

Label	Address
(global)	
_start	4194304
lab1-2.asm	
PARTITION	4194384
LOOP1	4194412
ELEMENT_...	4194472
LOOPDONE	4194488
printloop	4194532
printDone	4194584
array1	268500992
space	268501032

Registers

Name	Number	Value
zero	0	0
ra	1	4194376
sp	2	2147479548
gp	3	268468224
tp	4	0
t0	5	10
t1	6	9
t2	7	36
s0	8	268500992
s1	9	0
a0	10	268501032
a1	11	0
a2	12	0
a3	13	0
a4	14	0
a5	15	0
a6	16	0
a7	17	10
s2	18	9
s3	19	0
s4	20	0
s5	21	0
s6	22	0
s7	23	0
s8	24	0
s9	25	0
s10	26	0
s11	27	0
t3	28	268501028
t4	29	78
t5	30	268501028
t6	31	8
pc		4194384

Console

```

-1 5 4 2 1 8 11 35 22 78
-- program is finished running (0) --
  
```

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 following steps:

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 $a0 = -1$ ($a0 \neq 0$) 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 handling input and output	<pre> 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 </pre>
QUICKSORT	<pre> 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 </pre>

Table 4: Main Code for Question 3(continue)

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

Figure 3: Console Results(Output: Last line)

The screenshot displays the RARS 1.6 emulator interface. The main window is titled "/Users/jellyfish/Desktop/lab1-3.asm - RARS 1.6". The interface includes a menu bar (File, Edit, Run, Settings, Tools, Help), a toolbar, and a status bar indicating "Run speed at max (no interaction)".

The central area is divided into several panels:

- Text Segment:** Displays assembly code with columns for Address, Code, Basic, and Source. The code includes instructions like `addi x17,x0,5`, `ecall`, `la s0,array`, `li s1,0`, `addi a0,a0,-1`, `mv s2,a0`, `li a0,-1`, `jal ra,preLoop`, `mv a1,s1`, `mv a2,s2`, `jal ra,QUICKSORT`, `li a0,0`, and `jal ra,preLoop`.
- Data Segment:** Displays memory values at various addresses. The values are mostly 0, with some non-zero values at addresses 268500992 (32), 268501024 (22), 268501056 (0), 268501088 (0), 268501120 (0), 268501152 (0), 268501184 (0), 268501216 (0), 268501248 (0), and 268501280 (0).
- Registers:** A table showing the state of various registers. The registers include `zero`, `ra`, `sp`, `gp`, `tp`, `t0`, `t1`, `t2`, `s0`, `s1`, `a0`, `a1`, `a2`, `a3`, `a4`, `a5`, `a6`, `a7`, `s2`, `s3`, `s4`, `s5`, `s6`, `s7`, `s8`, `s9`, `s10`, `s11`, `t3`, `t4`, `t5`, `t6`, and `pc`. The values are mostly 0, with some non-zero values for `ra` (4194360), `sp` (2147479540), `gp` (268468224), `t2` (268501832), `s0` (268500996), `a0` (268500992), `a1` (11), `a2` (12), `a3` (13), `a4` (14), `a5` (15), `a6` (16), `a7` (17), `s2` (18), `s3` (19), `s4` (20), `s5` (21), `s6` (22), `s7` (23), `s8` (24), `s9` (25), `s10` (26), `s11` (27), `t3` (28), `t4` (29), `t5` (30), `t6` (31), and `pc` (4194368).
- Console:** Displays the output of the program. The output is:


```

5
4
11
2
1
78
-1 1 2 4 5 8 11 22 35 78
-- program is finished running (0) --
      
```