CS2100 Computer Organisation

Lab 04: Coding in MIPS (Week 14th September)

Instruction

Short and clean
We have separated the lab information into i) **instruction** and ii) **report**. Whenever there is a question in the instruction (easily identified as they have **[X pts]** tagged to the end), write / type your answer in the corresponding location in the **report** document. Please take note of the submission specification at the end of this document.

Objective

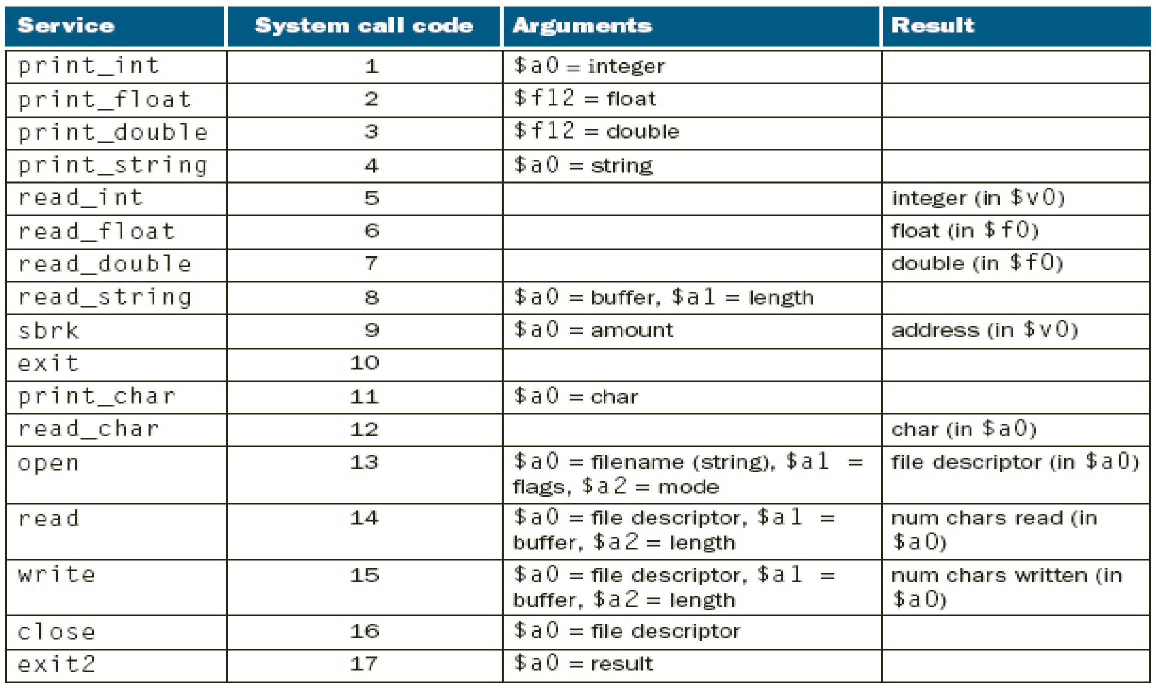
In this lab, you will use **QtSpim** to understand how typical programs are written. Make sure you have access to all the files included in the `Lab04.zip` archive: `messages.asm` and `arracyCount.asm`.

Please note that this lab is rather challenging. Please complete as much as you can before attending the lab session, otherwise, it is very likely that you would not be able to finish in time.

Reading and Writing Message to Console Window: `messages.asm`

Recall that in Lab #3 `sample2.asm`, you made use of the system call (`syscall`) to print some text. SPIM provides a small set of operating-system-like services through the system call (`syscall`) instruction (see Appendix A, pages A-43 to A-45).

To request a service, a program loads the system call code into register `$v0` and arguments into registers `$a0` – `$a3` (see Figure A.9.1 below). System calls that return values put their results in register `$v0`. For this lab, we are interested in only the following system calls: `print_string`, `print_int`, `read_int` and `exit`.



Service	System call code	Arguments	Result
print_int	1	$a0 = integer	
print_float	2	$f12 = float	
print_double	3	$f12 = double	
print_string	4	$a0 = string	
read_int	5		integer (in $v0)
read_float	6		float (in $f0)
read_double	7		double (in $f0)
read_string	8	$a0 = buffer, $a1 = length	
sbrk	9	$a0 = amount	address (in $v0)
exit	10		
print_char	11	$a0 = char	
read_char	12		char (in $a0)
open	13	$a0 = filename (string), $a1 = flags, $a2 = mode	file descriptor (in $a0)
read	14	$a0 = file descriptor, $a1 = buffer, $a2 = length	num chars read (in $a0)
write	15	$a0 = file descriptor, $a1 = buffer, $a2 = length	num chars written (in $a0)
close	16	$a0 = file descriptor	
exit2	17	$a0 = result	

FIGURE A.9.1 System services.

For example, the following code in messages.asm prints "the answer = 5".

```
# messages.asm
  .data
str: .asciiz "the answer = "
  .text
main:
    li    $v0, 4     # system call code for print_string
    la    $a0, str   # address of string to print
    syscall          # print the string

    li    $v0, 1     # system call code for print_int
    li    $a0, 5     # integer to print
    syscall          # print the integer

    li    $v0, 10    # system call code for exit
    syscall          # terminate program
```

The print_string system call (system call code 4) is passed a pointer (memory address) to a null-terminated string, which it writes to the console. The print_int system call (system call code 1) is passed an integer and it prints the integer on the console. The exit system call (system call code 10) indicates the end of the program.

The `li` (**l**oad **i**mmediate) and `la` (**l**oad **a**ddress) are pseudo-instructions:

- `li` : place the indicated constant value in a register. Depending on the size of the constant, either an `addi` or a pair of `lui`+`ori` instructions are used.
- `la` : ask the assembler to place the address of the variable in a register (i.e. effect is similar to the "&" address-of operator in C).

Run the program to verify your understanding.

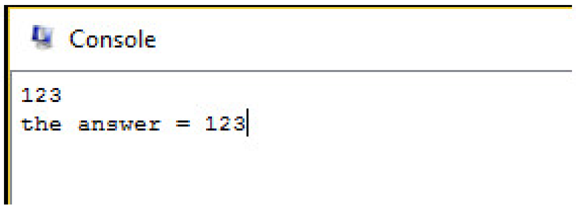
Task 1: Modify `messages.asm` [2 marks]

Copy `messages.asm` over to a new program `task1.asm`. The new program should:

i. Read the value **V** from the console using the **`read_int`** system call. This system call reads an entire line of input up to and including the newline. Characters following the number are ignored. Note that **`read_int`** modifies the register `$v0` (where you put the code for system call) as it returns the integer value in register `$v0`.

ii. Print the value **V**. The system call **`read_int`** reads an entire line of input up to and including the newline. Characters following the number are ignored.

The following screen capture shows a run of the program. The first line is your input, and the second line is the output of your program.



```
123
the answer = 123
```

Demonstrate your new program `task1.asm` to your lab TA. **[3pts]**

Task 2: Getting Real (`arrayCount.asm`) [12 marks]

The MIPS code in the lecture usually have a “`variable mappings`” list. The list indicates how certain program variables are “mapped” to their respective registers. In this task, we are going to actually perform these mappings.

First, let us learn about allocating memory space for variables in a program. The assembler directive "**`.data`**" allows us to reserve memory space in the data segment. These reserved locations are used to store the values of various program variables during program execution.

> **Key idea:** Values of program variables are stored in the memory. We load them into registers (perform a mapping) only when we want to manipulate or access them during execution.

This is because register is a fast storage **in the processor**, while memory is a much slower storage **outside the processor**. As the access speed is not simulated in the QtSpim, the separation and mapping between memory and register may seem strange to you. In real processor, the access speed difference between register and memory can be more than a factor of 10!

Let us modify the “count zero element” example from Lecture #10 (Section: Array and Loop) for this task. For simplicity, let us reduce the array size to **8**. The problem statement now reads:

> *Count the number of elements **that are smaller or equal to X,** in a given array of **8** non-negative numbers. **X*** is a user chosen non-negative value, e.g. 0, 7, 23, etc

Take a look at the initial content of the file `arrayCount.asm`:

```
# arrayCount.asm
        .data
arrayA: .word 1, 0, 2, 0, 3       # arrayA has 5 values
count:  .word 999                 # dummy value

        .text
main:
     # code to setup the variable mappings
     add $zero, $zero, $zero      # dummy instructions, can be removed
     ............

# read in user input, X

# counting elements <= to X in arrayA

# printing result

# terminating program
```

```
    li  $v0, 10
    syscall
```

The main routine contains several ***dummy instructions*** (instructions with no real effect) so that you can step through the program to observe the content in the data segment.

Answer the following in the report.

a. Give the base address (starting address) of the array **arrayA** located in the data segment. **[1pt]**

b. Give the base address of the program variable **count** in the data segment. (Hint: the initial value "999" is in decimal……). **[1 pt]**

c. The given code only allocates 5 elements for **arrayA**. **Enlarge the array to size 8**. You can place any valid integer values for the new locations. Give the assembler directive in the report. **[1 pt]**

Now, let us perform the following mappings (similar to notation used in lectures):

```
Base address of  arrayA  ➔ $t0
                 count  ➔ $t8
```

You may use the "**la**" (load address) instruction here to help. Give the instruction sequence (which may consist of 1 or 2 instructions) to map the following variables:

d. Map base address of **arrayA** into register **$t0**. **[2 pts]**
e. Map program variable **count** into register **$t8**. **[2 pts]**

We are almost ready to tackle the task. One last obstacle is to figure out how to check for "*Elements **that are smaller or equal to X***". Translate the following C-like code fragment into MIPS **using <u>real MIPS instructions</u> only** (i.e. you need to translate any pseudo instruction):

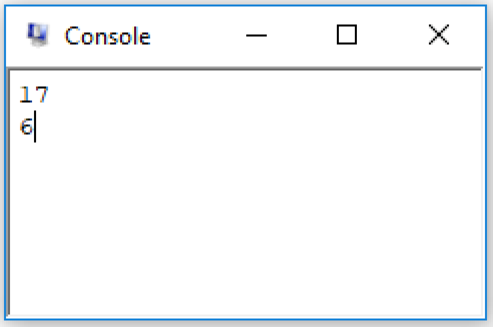
```
if ( $t4 <= $t5 )       //array element in $t4, X in $t5
    $t8 = $t8+1        // "count" in $t8
```

For simplicity, you can assume the variables / values are already setup according to the comments above. **Give your MIPS translation in the report. [2 pts]**

We are now ready to finish off this task. Add in additional code to:

i. Read user input value, **X**. You can assume **X** is always a non-negative integer, i.e. there is no need to check for invalid user input.
ii. Count the number of elements of **arrayA** that are **smaller or equal to X**.
iii. Print the result from (b) on screen.

You should use loop wherever appropriate, or full credit will not be given. Sample code can be found in Lecture #08 MIPS Part 2. Here's a sample output screenshot for a sample array **{11, 0, 31, 22, 9, 17, 6, 9}** and user input of **X = 17**. The output is **6** as there are three values in the array that are smaller than or equal to **17**: **11, 0, 9, 16, 6 and 9**



```
17
6
```

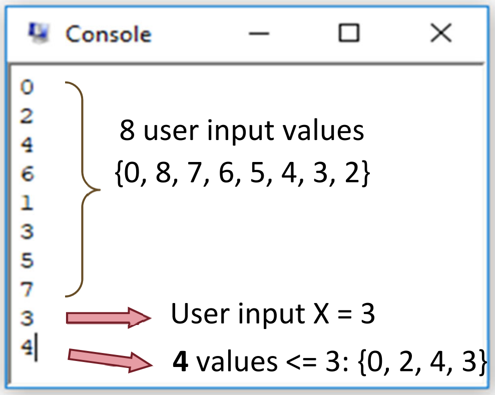
Try to use different values in your code to test. Also, please make sure the "**count**" value is reflected properly **in the data segment at the end of execution**.

[You should self-verified that your output is correct once you put in all the code pieces.]

Task 3: Making it "real-er" (`inputArrayCount.asm`) [3 marks]

This is a follow-up task on task 2. First, make a copy of your solution in task 2 and name it "`inputArrayCount.asm`".

Your task is very "simple" – add code to read **8 values** from the user and store them in the array `arrayA`. Then print the number of ***that are smaller or equal to X***. By reusing your code in task 2, you only need to add a couple of new instructions. Below is a sample run:



```
0
2
4
6
1
3
5
7
3
4
```

Please note that we read the array values before the user enters the value **X**. Submit your completed `inputArrayCount.asm`. Your labTA will test your program with some test cases. **[3 pts]**

Submission You need to submit into Luminus Folder no longer than **2359 on the same day** you have the online lab. Please zip the **report document** and the completed **inputArrayCount.asm** into **[student_id]_lab04.zip**, e.g. **A1234567X_lab04.zip**. Submit into the correct folder on Luminus before 2359 of your lab day.
Marking Scheme: Report & Code – 12 marks; Demonstration – 3 marks; Total: 15 marks.