CS2100: Computer Organisation
Lab #4: Writing MIPS code using QtSpim

Name: ______________________________________ Student No.: ___________________

Lab Group: ______

Remember to bring this along to your lab!

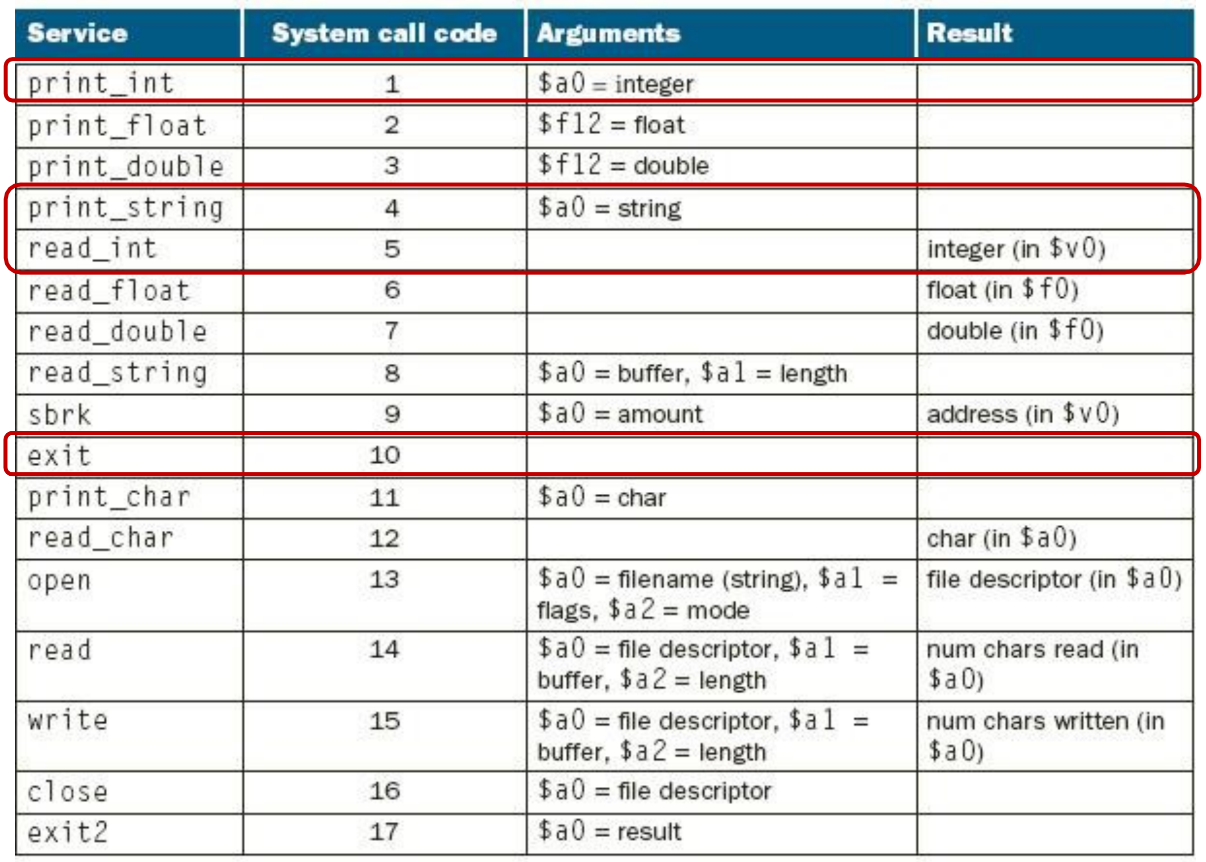
Objective

In this lab, you will use the **QtSpim** to understand how typical programs are written. This document and its associated files (`messages.asm` and `arrayCount.asm`) can be downloaded from Canvas or the CS2100 course website.

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. QtSpim provides a small set of operating-system-like services through the system call (`syscall`) instructions (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` – `$a2` (see Figure A.9.1 below). System calls that return values put their results in register `$v0`. For this lab, we are interested in only the following system calls: `print_int` (code 1), `print_string` (code 4), `read_int` (code 5) and `exit` (code 10).



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 via register **`$a0`** which it prints on the console. The **`print_int`** system call (system call code 1) is passed an integer via register **`$a0`** which it prints on the console. The **`exit`** system call (system call code 10) terminates the program.

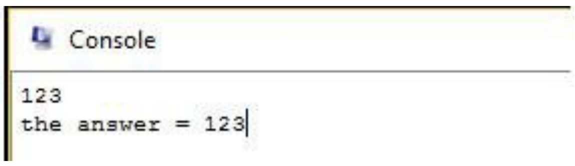
The **`li`** (load immediate) and **`la`** (load address) are pseudo-instructions (refer to Lab #3).

Run the above program to verify your understanding.

Task 1: Modify messages.asm [2 marks]

Modify **`messages.asm`** and call the new program **`task1.asm`**. The modified program should read the value to be printed from the console before printing the value. Recall that a string is simply an array of character. So we pass the address of the first element of the null-terminated string. The system call **`read_int`** 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`.

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.

Task 2: Getting Real (`arrayCount.asm`) [18 marks]

When we discuss MIPS code in the lecture, it is common to see the “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 QtSpim, the separation and mapping between memory and register may seem strange to you. In a real processor, the difference in access speed of register versus memory can be multiple of 10 times!

Let us modify the “count zero element” example from Lecture #8 (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 **multiples of X** in a given array of **8** non-negative numbers, where **X** is a user chosen power-of-two value, e.g. 1, 2, 4, 8, ….*

Download **arrayCount.asm** from Canvas or the course website. The initial content of the file is:

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

        .text
main:
     # code to set up the variable mappings
     add $zero, $zero, $zero  # dummy instructions
                              # can be removed
     ............
     # code for reading in the user value X

     # code for counting multiples of X in arrayA

     # code for printing result

     # code for terminating the program
     li   $v0, 10
     system
```

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.

Where is the array **arrayA** located in the data segment? Give the base address (starting address) of the array:

arrayA is at 0x______________________

Where is the program variable **count** in the data segment?

count is at 0x_______________________

(Hint: Don't forget that 999 is in decimal.)

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. Fill in the assembler directive below:

arrayA: ___________________________________

Now, let us perform the following mappings:

> Base address of **arrayA** ➔ **$t0** (similar to notation used in lectures)
> **count** ➔ **$t8**

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

To map **arrayA**: ____________________________

____

To map **count**: ____________________________

____

We are almost ready to tackle the task. One last obstacle is to figure out how to check for "multiples of X, where X is a power-of-two". Recall that **andi** instruction can be used to find the remainder of division by a power of two. For the following questions, give the correct **mask** for the **andi** instruction to compute "**$t4 = $t3 % X**".

If X is **32**: **andi $t4, $t3, 0x**__________

If X is **8**: **andi $t4, $t3, 0x**__________

Observe that we can easily generate the mask from X. If X is stored in register **$t8**, complete the following instruction to generate the mask in register **$t5**. (*Hint:* look at the mask as a **number**).

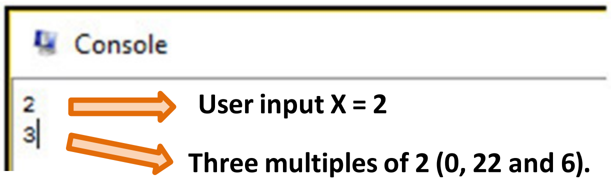
_________ **$t5, $t8,** _________ (fill in the operation and the last operand)

We are now ready to finish off the task. Write the necessary code to:

a. Read user input value, `X`. You can assume `X` is always a power-of-two integer, i.e. there is no need to check for invalid user input.

b. Count the number of multiples of `X` in `arrayA` **and print the result on the console**.

You should use loop wherever appropriate, or full credit will not be given. Sample code can be found in Lecture #8 MIPS Part 2 (slides 32-34). Here's a sample output screenshot for a predefined array `{11, 0, 31, 22, 9, 17, 6, 9}` and user input of `X = 2`. The output is 3 as there are three multiples of 2 in the array: 0, 22 and 6.



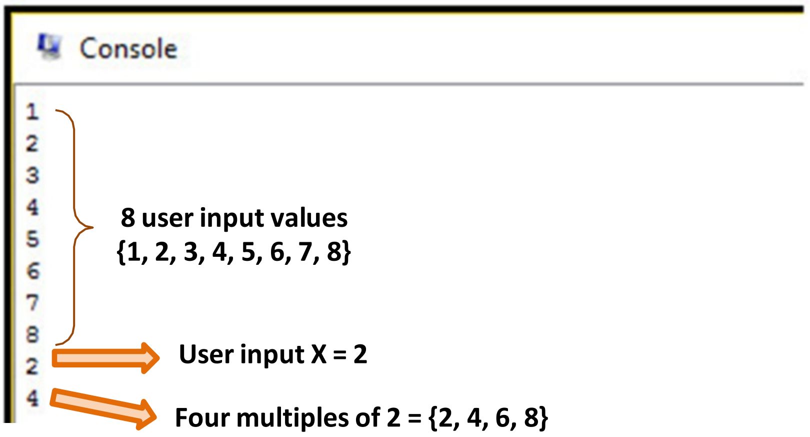
```
2
3
```

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

Task 3: Making it "real-er" (`inputArrayCount.asm`) [5 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 multiples of X found (where X is a power-of-two also entered by the user). By reusing your code in task 2, you only need to add a couple of new instructions. Below is a sample run:



```
1
2
3
4
5
6
7
8
2
4
```

Please note that we read the array values before the user enters the value X. Your labTA will test your program with some test cases.

Notes: You should prepare your programs **before the lab**. Your labTA will mark your work in your presence. <u>You do not need to submit any program. Please do not send your programs to your labTA after the lab; they will **NOT** be accepted.</u>

Marking Scheme: Report/Demonstration (25 marks)