

Lab Sheet 1 for CS G524 Advanced Computer Architecture

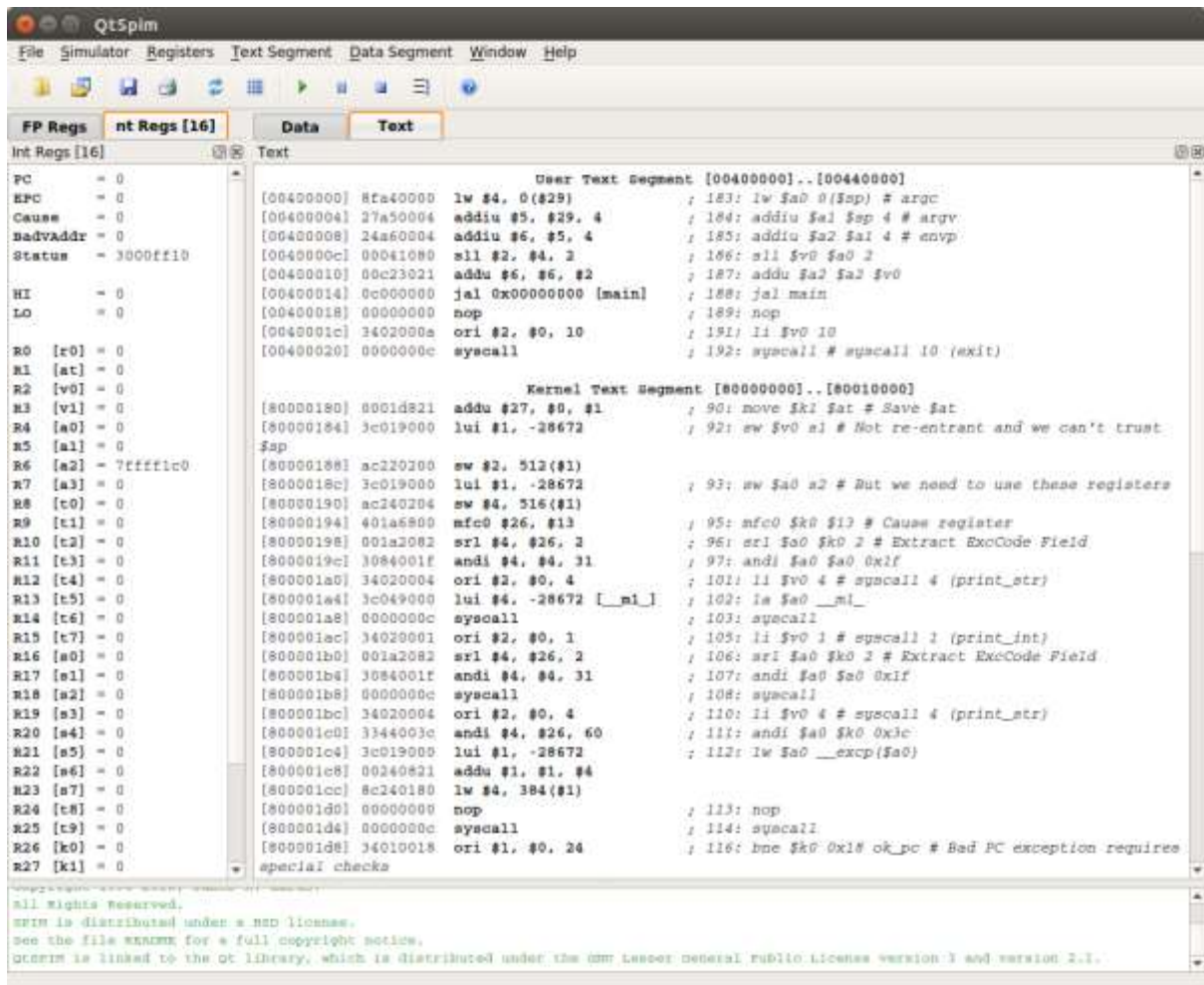
Semester 2 – 2019-20

Goals for Lab 1: Installing and launching QTSPIM – this tool will be used for Lab Sheet 1

Installing and launching QTSPIM

Open a terminal to run the following commands.

- Check the OS and the processor support on your Linux machine – verify whether it is 32 bit or 64 bit. [Try commands like `uname -p` ; `uname -a` ; `man uname`]
- Download the appropriate QTSPIM binary from Internet (<https://sourceforge.net/projects/spimsimulator/files/>) or from <http://172.16.103.147/CA>
- Install using `dpkg` tool. Do not forget to get “super user privilege” using `sudo`.
`$ sudo dpkg -i <qtspim package>`
- Launch QTSPIM



The screenshot shows the QtSpim MIPS simulator interface. The 'Registers' panel on the left lists 32 registers (R0-R31) with their current values. The 'Text' panel on the right displays the assembly code for the 'User Text Segment' and 'Kernel Text Segment'. The 'User Text Segment' code includes instructions for loading arguments, setting up the environment, and calling the main function. The 'Kernel Text Segment' code includes instructions for saving registers, setting up the environment, and calling the main function. The 'Data' panel is also visible, showing the memory layout.

Note: Please refer to the following: (i) HP_AppA.pdf (ii) Chapter 2 Hennessey & Patterson Book (iii) **MIPS Reference Data Card (“Green sheet”)**

Goal 2: To get introduced to QTSPIM and implement some code related to - System Calls and User Input. Further, we will do basic integer Add/Sub/And/Or and their immediate flavours (e.g. ori).

Reference for MIPS assembly – refer to the **MIPS Reference Data Card (“Green sheet”)** uploaded in CMS. Further, some of the QTSPIM assembly instructions are beyond this data card (e.g. the pseudo instruction *la*).

Additionally use Appendix A (HP_AppA.pdf) from Patterson and Hennessey “Assemblers, Linkers and the SPIM Simulator” for gaining background knowledge of SPIM.

In this lab we focus on reversing only integer based instructions (add, or, subi etc.).

Reference for Registers:

0 zero constant 0	16 s0 callee saves
1 at reserved for assembler	...
2 v0 results from callee	23 s7
3 v1 returned to caller	24 t8 temporary (cont'd)
4 a0 arguments to callee	25 t9
5 a1 from caller: caller saves	26 k0 reserved for OS kernel
6 a2	27 k1
7 a3	28 gp pointer to global area
8 t0 temporary	29 sp stack pointer
...	30 fp frame pointer
15 t7	31 ra return Address
	caller saves

System calls as well as functions (in later part of the semester) should take care of using the registers in proper sequence. Especially take note of V0, V1 [R2, R3 in QTSPIM] and a0-a3 [R4-R7 in QTSPIM] registers.

Reference for System Calls:

Service	Code (put in \$v0)	Arguments	Result
print_int	1	\$a0=integer	
print_float	2	\$f12=float	
print_double	3	\$f12=double	
print_string	4	\$a0=addr. of string	
read_int	5		int 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	addr in \$v0
exit	10		

Reference for Data directives:

.word w1, ..., wn

-store n 32-bit quantities in successive memory words

.half h1, ..., hn

-store n 16-bit quantities in successive memory half words

.byte b1, ..., bn

-store n 8-bit quantities in successive memory bytes

.ascii str

-store the string in memory but do not null-terminate it

-strings are represented in double-quotes "str"

-special characters, eg. \n, \t, follow C convention

.asciiz str

-store the string in memory and null-terminate it

.float f1, ..., fn

-store n floating point single precision numbers in successive memory locations

.double d1, ..., dn

-store n floating point double precision numbers in successive memory locations

.space n

-reserves n successive bytes of space

Layout of Code in QTSPIM: Typical code layout (*.asm file edited externally)

objective of the program

.data #variable declaration follows this line

.text #instructions follow this line

main: # the starting block label

...

xxx

yyy

zzz

.....

li \$v0,10 #System call- 10 => Exit;

syscall # Tells QTSPIM to properly terminate the run

#end of program

Exercise 0: Understanding Pseudo instruction.

Not all instructions used in the lab will directly map to MIPS assembly instructions. Pseudo-instructions are instructions not implemented in hardware. E.g. using \$0 or \$r0 we can load constants or move values across registers using add instruction.

E.g. li \$v0, 10 actually gets implemented by assembler as ori \$v0, \$r0, 10


In subsequent exercises, identify the pseudo instruction by looking at the actual code used by QTSPIM.

Exercise 1: Integer input and output and stepping through the code.

Invoking system calls to output (print) strings and and input (read) integers.

The following code snippet prints the number 10 on console. Modify it to read any number and print it back.

Hint: To copy it from \$v0 to \$a0, you can use add or addi with 0 or similar options.

Edit the code in your editor of choice and then load it in QtSpim. Single Step [] through the code and look at the register values as you execute various instructions.

The demo code is given below:

```

# demo code to print the integer value 10

.data #variable declaration follow this line
# sample string variable declaration - not used in first exercise.
myMsg: .asciiz "Hello Enter a number." # string declaration
      # .asciiz directive makes string null terminated

.text #instructions follow this line
main:
li $a0,10
li $v0,1
syscall

li $v0,10 #System call - Exit - QTSPIM to properly terminate the run
syscall
#end of program

```

Exercise2: Modify the above code to output “myMsg” along with the input integer. You will use load address MIPS instruction (la \$a0, myMsg)

Exercise 3: Take 2 integers as input, perform addition and subtraction between them and display the outputs. The result of addition is to be displayed as "The sum is =" and that of subtraction is to be displayed as "The difference is =". Check if negative integers can be handled.