CS2100 Computer Organisation
Lab #3: Exploring QtSpim

[This document is available on LumiNUS and module website http://www.comp.nus.edu.sg/~cs2100]

Name: _____   Student No.: _____

Lab Group: _____

Objective

In this lab, you will explore the **QtSpim**, a MIPS simulator available on Windows, Mac OS and Linux. This document and its associated files (**sample1.asm**, **sample2.asm** and **sample3.asm**) can be downloaded from LumiNUS or the module website.

> You are encouraged to bring a thumb-drive to store the assembly programs for your lab session. Alternatively, you may create a directory on desktop in the lab computer for this purpose.

Software and Documentation

The following resources are available on the CS2100 website → "Resources" → "Online", or https://www.comp.nus.edu.sg/~cs2100/2_resources/online.html directly.

1. **QtSpim software**: Download the correct QtSpim for your platform (either Windows or MacOS). Follow the installation instructions to install QtSpim in your system. In the lab, QtSpim for Windows has already been installed on the computers.

2. **Assemblers, Linkers, and the SPIM Simulator documentation**: An overview and reference manual for SPIM and MIPS32 instruction set. The sections that you should definitely read are **Section A.9 (SPIM)** and **Section A.10 (MIPS R2000 Assembly Language).** This document can also be found in the "Computer Organization and Design" reference book as Appendix A (3rd edition and before) or Appendix B (4th Edition). Note that the SPIM discussion is based on the older version of the simulator. Although there is significant change in the looks (UI) of the simulator, the underlying mechanism and information are largely intact.

Introduction

SPIM, which is just **MIPS** spelled backwards, is a software that simulates the working of a MIPS processor. It is capable of running MIPS32 assembly language programs and it allows user to inspect the various internal information of a MIPS processor. SPIM is a self-contained system for running MIPS programs. It contains a debugger and provides a few operating system–like services. SPIM is much slower – 100 or more times! – than a real computer. The most current version of SPIM is known as QtSpim which is basically the simulator SPIM with the Qt GUI framework on top. QtSpim is cross-platform and currently available for Windows, Mac OS and Debian Linux.

My First MIPS Program: `sample1.asm`

Let's give it a try! Before you start, make sure you have the 3 assembly programs `sample1.asm`, `sample2.asm` and `sample3.asm` from the zip file in a working directory of your choice.

Click on the QtSpim shortcut at the desktop to invoke QtSpim. You should see a screen similar to the following:
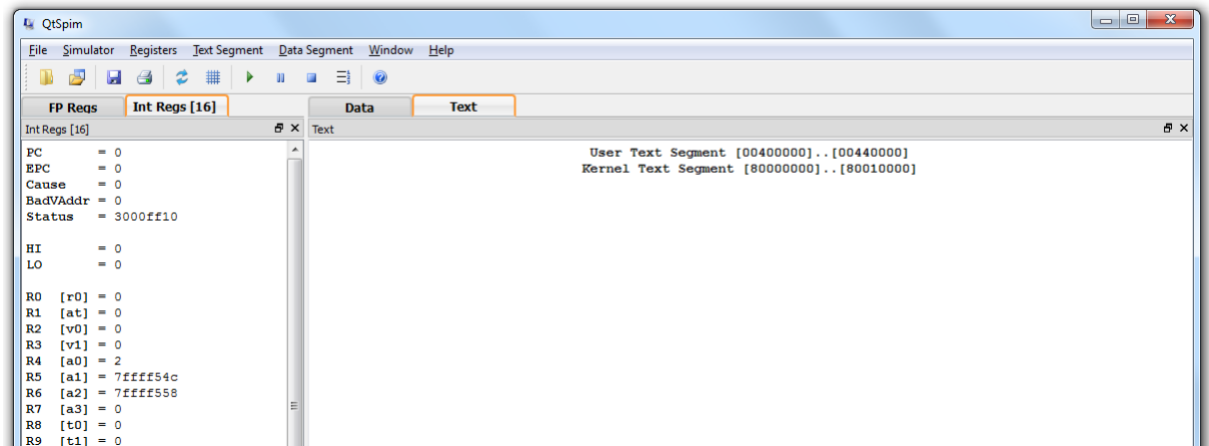

Let's check that we have the right simulator configuration for the labs. Click on "Simulator → Settings" on the menu bar. Go to the "MIPS" tab then:

1. Click the "Simple Machine" button to use the simplest setting for the processor simulation.
2. Uncheck the "Load Exception Handler".
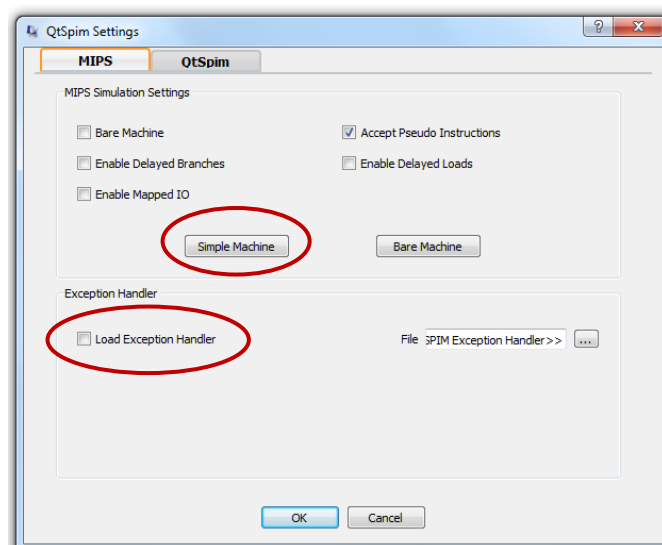
Click "OK" to save your setting. Go to "File → Reinitialize and Load File" and select `sample1.asm` from your working directory.

With the settings mentioned, your QtSpim window should look exactly the same as follows:
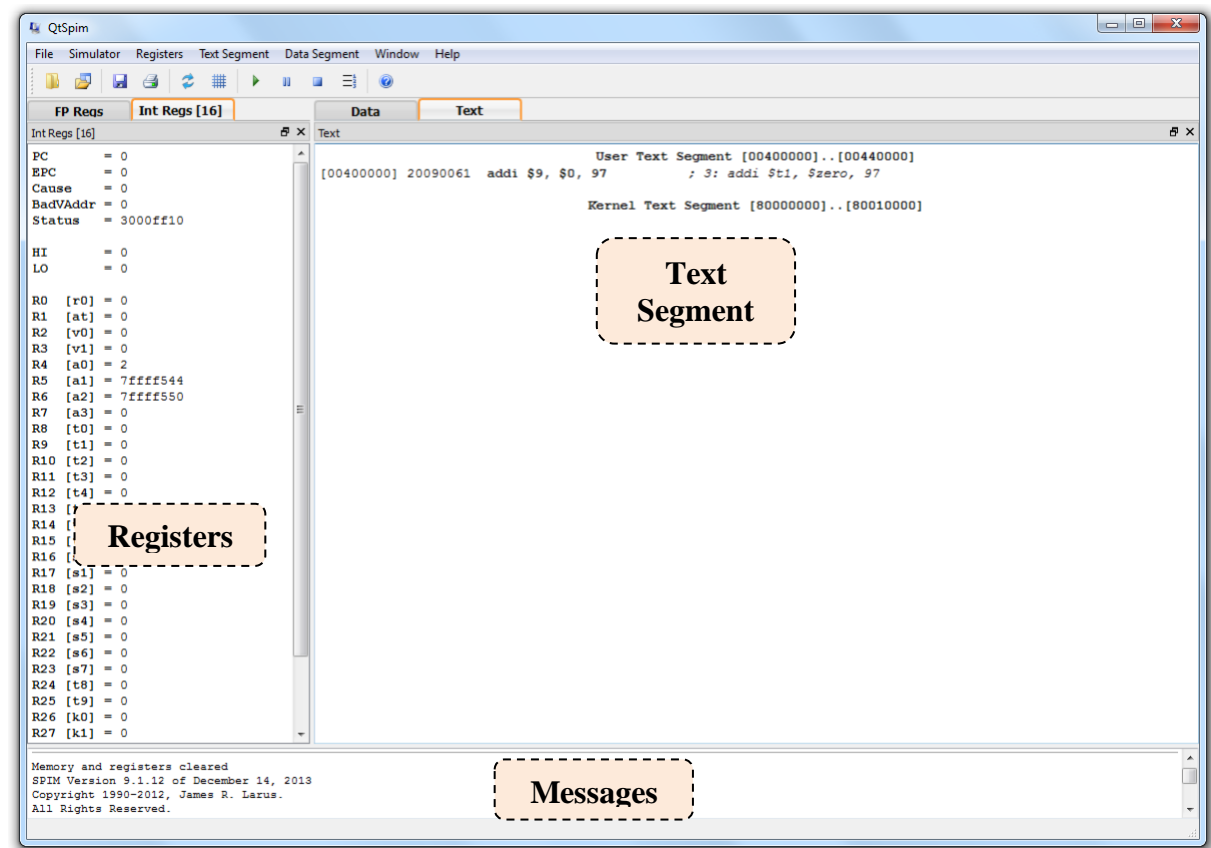
The display is divided into three frames: *Registers*, *Text Segment* and *Messages*.

- *Registers* frame: This window shows the values of special and integer registers in the MIPS CPU. There is a tab for FPU (floating point unit) registers, which you can safely ignore.

- *Text Segment* frame: This window displays instructions from your assembly program. As **sample1.asm** contains three lines of assembly code, you will see these lines. However, the lines in your code may not appear verbatim in QtSpim (to be explained later).

- *Messages* frame: This is where QtSpim writes messages and where error messages appear.

Note that the *Text Segment* Frame can change to display the *Data Segment* by clicking on the "Data" tab. The data segment represents the data your program loaded into the "main memory", we'll see more of this later.

Besides the above, there is also a separate *Console* window for input and output. If the Console window is hidden, you can click "Windows → Console" to enable it.

Let us now take a look at the **`sample1.asm`**:

```
# sample1.asm
        .text
main: addi $t1, $zero, 97
        li $v0, 10
        syscall
```

```
addi $9, $0, 97 ; 3: addi $t1, $zero, 97
ori $2, $0, 10  ; 4: li $v0, 10
syscall         ; 5: syscall
```

The first line is a comment line, which begins with "#". The second line **`.text`** is the *assembler directive* that specifies the starting address of the source code. If no starting address is specified, the default value **0x00400000** will be assumed.
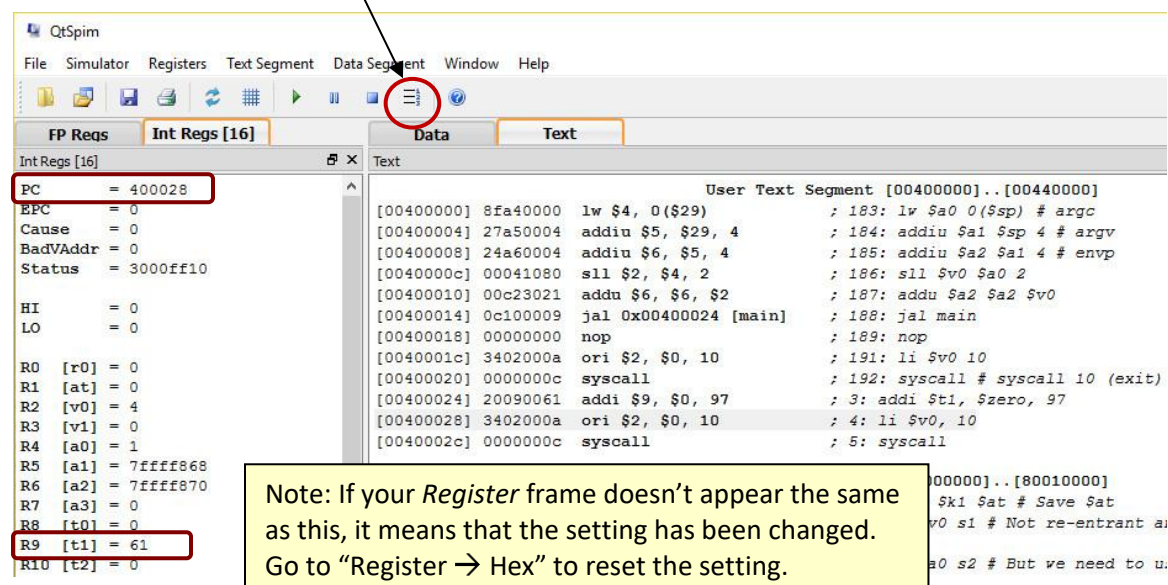
The next line contains a MIPS instruction: the *add immediate* (**`addi`**) instruction will place the addition result (0 + 97), i.e. decimal value 97, into register **`$t1`**. Numbers are decimal (base 10) by default. If they are preceded by **0x**, they are interpreted as hexadecimal (base 16). For example, 256 and 0x100 denote the same value. In QtSpim, the registers **`$t1`** and **`$zero`** are translated into their respective numbered registers **`$9`** and **`$0`**.

Observe the value of register **`$t1`** (register 9) and the value of **`PC`** (program counter) in the *Registers* window. They are both zero at the moment.

The next two lines **`li $v0, 10`** and **`syscall`** constitute a system call to exit. The line **`li $v0, 10`** is a pseudo-instruction (li = load immediate) that is translated into **`ori $2, $0, 10`**. This will be explained in the next section and we will ignore them here.

If you want to make any modification to **`sample1.asm`**, you can use any text editor such as NotePad, etc. After modification, you need to reload it with "File → Load File".

Now click on the "Single Step" button to execute one assembly statement.

You should see the changes to the **`PC`** and **`$t1`** register in the register frame.

What number base is the value in register $t1? Answer: _____base 16_____.

Text Segment **Frame**

Now, take a closer look at the text segment frame:
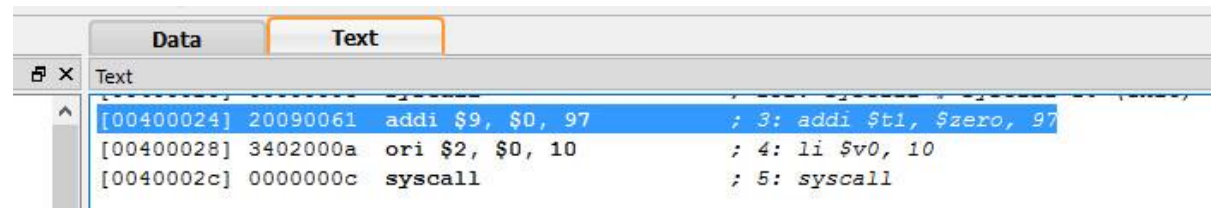
For each line,

- the first column – **[0x00400024]** – is the memory location (address) of the instruction; (the default starting address is 0x00400000; some lines of code are inserted before your code by the system, hence your code starts at 0x00400024)
- the second column – **0x20090061** – is the encoded instruction in hexadecimal;
- the third column – **addi $9, $0, 97** – is the native code; and
- the fourth column – **addi $t1, $zero, 97** – is your source code.

Everything after the semicolon is the actual line from your source code. The number "3:" refers to the line number of the corresponding MIPS instruction in the source code.
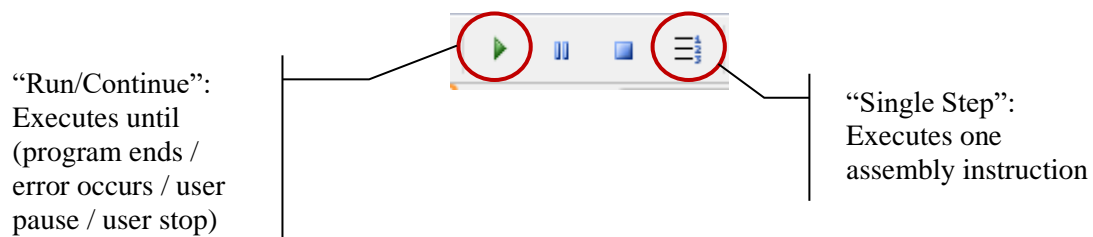
Sometimes there is nothing after the semicolon. This means that the instruction was produced by QtSpim as part of translating a pseudo-instruction into more than one real MIPS instruction.

Running and Stepping through Code

You can control the execution by choosing different modes.

"Run/Continue": Executes until (program ends / error occurs / user pause / user stop)

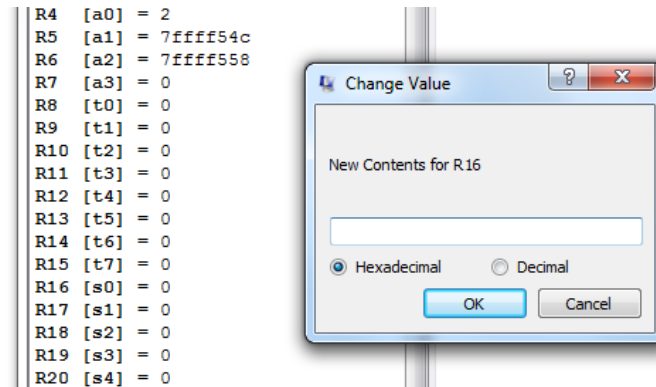"Single Step": Executes one assembly instruction

---

FAQ: I ran the code and encountered this "**Exception occurred at PC = 0x00000000**" error, what should I do?

Answer: Your QtSpim is not using address 0x00400000 as the default start address of your code. Click on "Simulator → Run Parameters" and change the value under "Address or label to start running program" from 0x00000000 to 0x00400000.

Setting Value into a Register

You may right click on a register in the register frame and select "Change Register Contents" to assign value directly into that register. Try to change the value of **R16** (which is **$16** or **$s0**) to decimal value **100** (or hexadecimal **64**).

```
R4  [a0] = 2
R5  [a1] = 7ffff54c
R6  [a2] = 7ffff558
R7  [a3] = 0
R8  [t0] = 0
R9  [t1] = 0
R10 [t2] = 0
R11 [t3] = 0
R12 [t4] = 0
R13 [t5] = 0
R14 [t6] = 0
R15 [t7] = 0
R16 [s0] = 0
R17 [s1] = 0
R18 [s2] = 0
R19 [s3] = 0
R20 [s4] = 0
```

Change Value

New Contents for R16

○ Hexadecimal   ○ Decimal

[OK]   [Cancel]

Writing a message: `sample2.asm`

Download **sample2.asm** into your working directory. Click on "File" → "Reinitialize and Load File" to open this file.

The content of **sample2.asm** is shown below:

```
# sample2.asm
      .data 0x10000100
msg:  .asciiz "Hello"
      .text
main: li $v0, 4
      la $a0, msg
      syscall
      li $v0, 10
      syscall
```

You will notice that there are a couple of special names starting with a period, e.g. ".data", ".text" etc. These are **assembler directives** that tell the assembler how to translate a program but do not produce machine instructions. The second line **.data** directive specifies the starting address (0x10000100) of the data (in this case, the string "Hello"). The **.asciiz** directive stores a null-terminated string (a string that ends with a null character) in memory.

For your reference, the set of directives that will be used in the lab exercises is given in the Appendix.

The first two statements are both pseudo-instructions:

- Load immediate (A-57 in Appendix A):
  li rdest, imm   # load the immediate **imm** into register **rdest**.
- Load address (A-66 in Appendix A):
  la rdest, address   # load computed address – not the contents of the location
   – into register **rdest**.

As discussed in lecture, these pseudo-instructions will be converted into an equivalent sequence of actual machine instructions. <u>Note that the use of pseudo-instruction is usually not allowed in test/exam</u>, so please don't get overly reliant on them ☺.

The **syscall** instruction makes a system call, and the value in register **$v0** indicates the type of call. The value **4** in **$v0** indicates a *print_string* call (A-43 to A-43 in Appendix A: System Calls), and the string to be printed is indicated by **$a0**.

The value **10** in **$v0** indicates an *exit* call.

We will explore more about system calls in the next lab.

Click on the "Data" tab on the *Text Segment* frame to display the *Data Segment.* Take a close look at the content of memory address **0x10000100**:
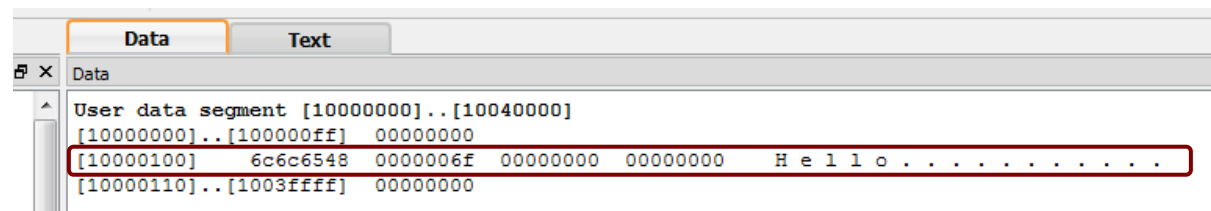
Can you figure out where and how is the string "Hello" stored? Write out the ASCII values, in hexadecimal form, of the characters 'H', 'e', 'l' and 'o' below:

'H': 48   'e': 65   'l': 6C   'o': 6F

Now, run the program (press the "Run/Continue" button). What do you see on the *Console* window? As mentioned before, you can ignore the "Attempt to execute non-instruction …." error message for now.

Answer: Hello

Modifying a message: `sample3.asm`

Download `sample3.asm` into your working directory. Click on "File" → "Reinitialize and Load File" to open this file. The file is the same as `sample2.asm` at this point except a few comments added near the end of the main routine. We are going to edit the program to perform some simple tasks.

First, take some time to check your understanding of the `li` (load immediate) and `la` (load address) pseudo-instructions. Use the "Single Step" button to step through the program. Stop at the line "`syscall`". What do you see in the register `$v0` and `$a0` at this point?

Answer: `$v0` = ___4___   `$a0` = ___10000100___

Let us now read a single character (i.e. single byte) from the string into a register. Edit the given program `sample3.asm` as described below:

Add an instruction (`lb` – load byte) to load the value of 'o' (the last character in "Hello") into register `$t0`. Add the instruction after the "`syscall`" instruction. What is the correct memory address to use? (Remember to save your edits, then reload the assembly file in QtSpim).

Answer: `lb $t0,` ___4($a0)___

Now, we want to change the 'o' into 'O' (capital Oh). Let us first change the value in register `$t0` by simple arithmetic. What is the difference in ASCII value between 'o' and 'O'?

Answer: ___32___

Use the (`addi` – add immediate) instruction to change the value in `$t0` to the ASCII value of 'O':

Answer: `addi` ___$t0, $t0, -32___

Finally, let us put this back into the memory. Use the (`sb` – store byte) instruction to place the changed value of `$t0` into the position 'o' (i.e. overwriting the value in memory). Take note of the change in data segment when you execute this instruction.

Answer: `sb` ___$t0, 4($a0)___

Now, just make another **syscall** to print the string again. Add the instruction **syscall** at the end of the program as indicated by the comment. What do you see in the output when you run the program?

Answer: _____HellO_____

Marking Scheme: Report/Demonstration (20 marks).

Appendix

QtSPIM assembler directives:

`.data <addr>`	Subsequent items are stored in the data segment. If the optional argument *addr* is present, subsequent items are stored starting at address *addr*.
`.globl sym`	Declare that label *sym* is global and can be referenced from other files.
`.ascii str`	Store the string *str* in memory, but do not null-terminate it.
`.asciiz str`	Store the string *str* in memory and null-terminate it.
`.text <addr>`	Subsequent items are put in the user text segment. In SPIM, these items may only be instructions or words (see the `.word` directive below). If the optional argument *addr* is present, subsequent items are stored starting at address *addr*.
`.word w1,..., wn`	Store the *n* 32-bit quantities in successive memory words.