CS2100 Computer Organisation

Lab 03: SPIM? You mean MIPS? (Week 7th September)

Instruction

Short and clean
We have separated the lab information into i) **instruction** and ii) **report**. You **ONLY** need to submit your **report** into Luminus Folder no longer than 2359 on the same day you have the online lab. 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.

Objective

In this lab, you will explore the **QtSpim**, a MIPS simulator available on Windows, Mac OS and Linux. Make sure you have access to all the files included in the `Lab03.zip` archive: `sample1.asm`, `sample2.asm` and `sample3.asm`.

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 [Prepare before Lab]

1. **QtSpim software**: Take a look at http://spimsimulator.sourceforge.net/ main page to learn more about SPIM. **Download and install the correct QtSpim for your platform** at https://sourceforge.net/projects/spimsimulator/files/.

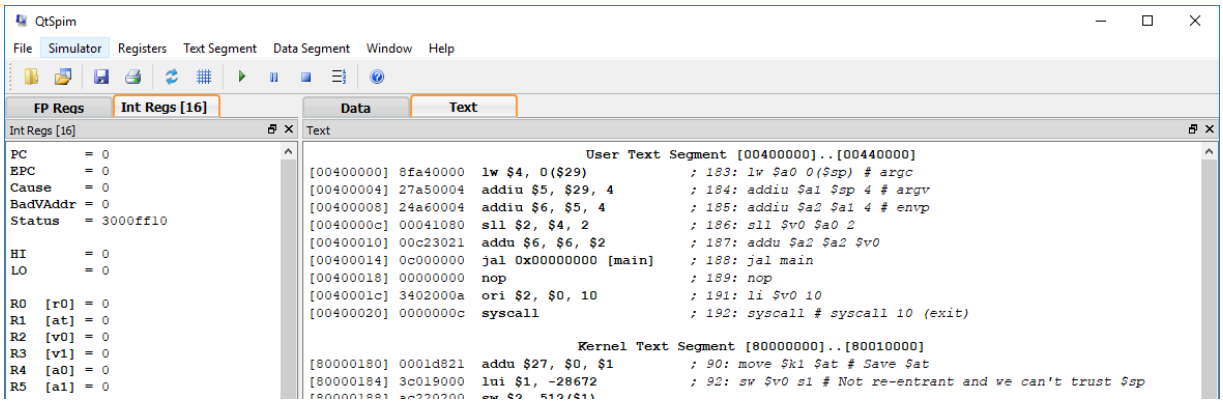
2. **Assemblers, Linkers, and the SPIM Simulator documentation (attached document)**: An overview and reference manual for SPIM and MIPS32 instruction set. The sections that you should definitely read are **Section A.9 (SPIM)** and **Section A.10 (MIPS R2000 Assembly Language)**. This document can also 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 (probably at a factor of 100x or more!) than a real computer, but it gives you insider look on the working of a processor. The most current version of SPIM is known as **QtSpim** which is essentially the simulator SPIM tacked 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`

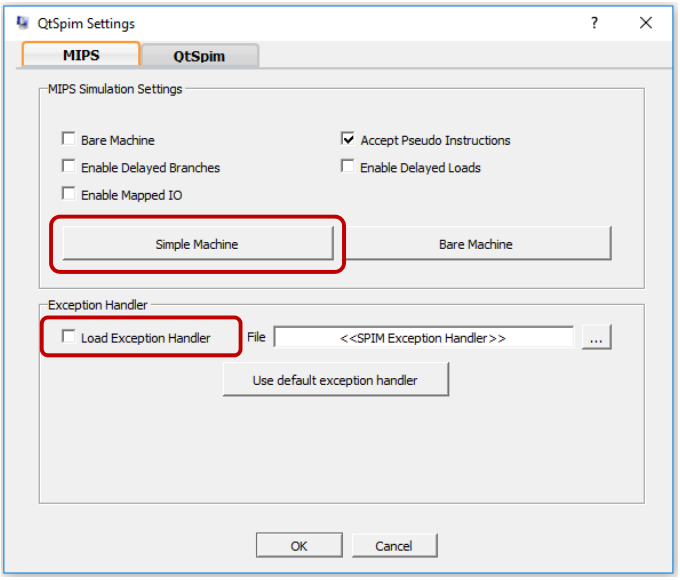
Start **QtSpim**, you should see a screen similar to the following:



Let's configure the simulator for our labs. Click on "`Simulator`→`Settings`" on the menu bar. Go to the "MIPS" tab then:

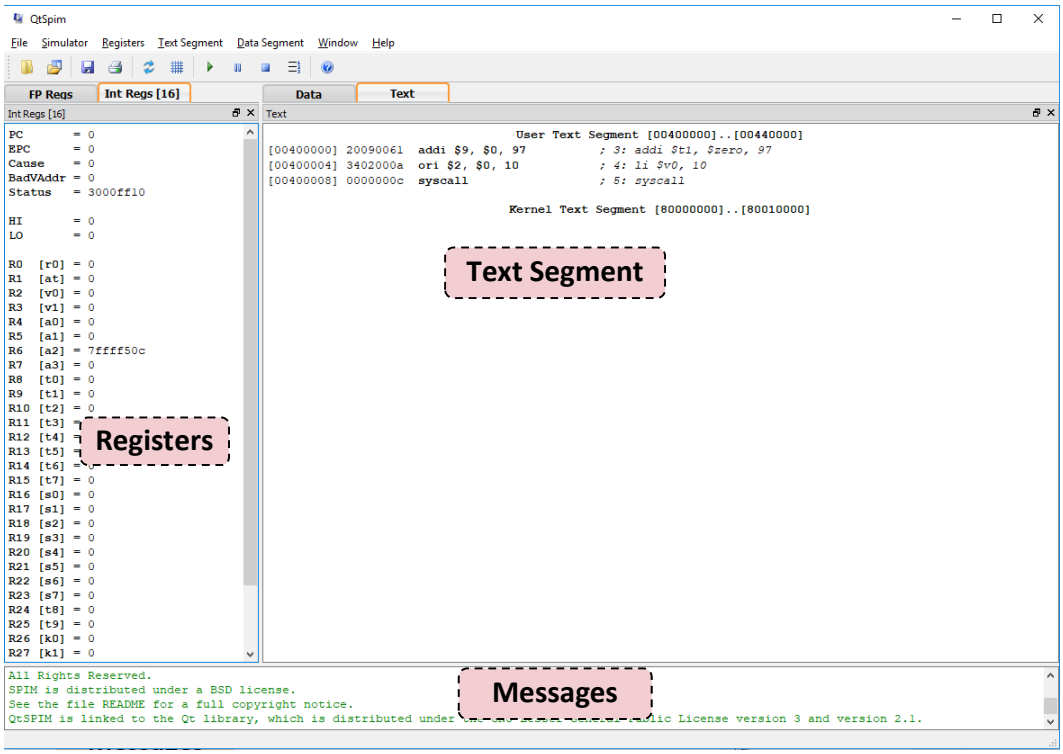
1. Click the "Simple Machine" button to use the simplest setting for the processor simulation.
2. Uncheck the "Load Exception Handler".

See the screenshot next page to check that you have selected the right options.



3. Click "OK" to save your setting.
4. Click on "Simulator→Run Parameter", then type in "0x00400000" in the "Address or label to start running program" text box.
5. Go to "File → Reinitialize and Load File" and select **`sample1.asm`** from your working directory.

With the new settings, your **QtSpim** window should look exactly as follows:



The display contains three frames:

1. ***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.
2. ***Text Segment* frame**: This window displays instructions from your assembly program. In this case, you see the three assembly instructions in **`sample1.asm`**. Note: some instructions in your assembly program may not appear verbatim in **QtSpim** (more later).
3. ***Messages* frame**: **QtSpim** output / error messages will appear here.

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`**:

Lines from **`sample1.asm`**	**Explanation**
`# sample1.asm`	Comment
`.text`	Assembler directive to specify starting address of the 1st instruction. If nothing is specified (as in this case), the default value **`0x00400000`** will be assumed.
`main: addi $t1, $zero, 97`	*add immediate* (**`addi`**) instruction will place the addition result $(0 + 97)_{10}$ 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.
`li $v0, 10`	This line and the next constitute a *system call* to **exit()**. The line **`li $v0, 10`** is a pseudo-instruction, which will be explained later.
`syscall`	Make a predefined system call, i.e. invoke system functionality. In this case, **exit()** is invoked to stop the processor.

Note that a few **pseudo-instructions** in the program are translated into **actual** MIPS instruction when loaded into **QtSpim**:

Lines from `sample1.asm`	Loaded in QtSPIM
`main: addi $t1, $zero, 97`	`addi $9, $0, 97` The registers `$t1` and `$zero` are translated into their respective numbered registers `$9` and `$0`.
`li $v0, 10`	`ori $2, $0, 10` The `li` is a pseudo-instruction.

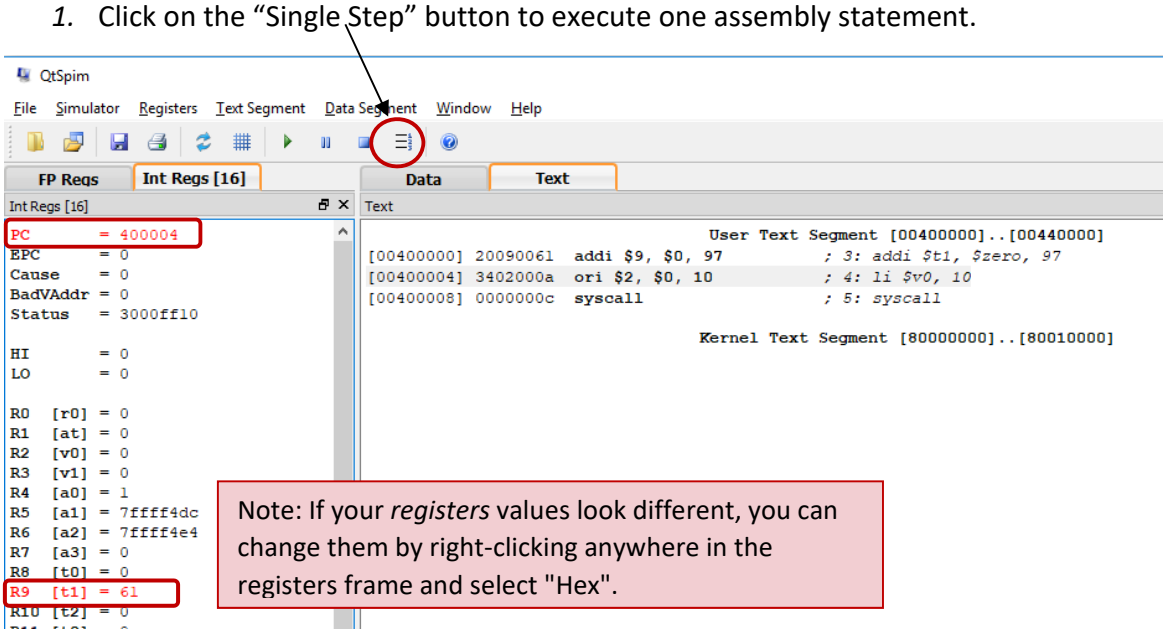
Observe that the original assembly instructions are shown as the comment in the ***Text Segment*** if the translation differs from the original.

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".

Executing the Assembly Program

Observe the value of register `$t1` (register 9) and the value of `PC` (program counter) in the *Registers* window. The values for these registers can be found in the screenshot below.

1. Click on the "Single Step" button to execute one assembly statement.

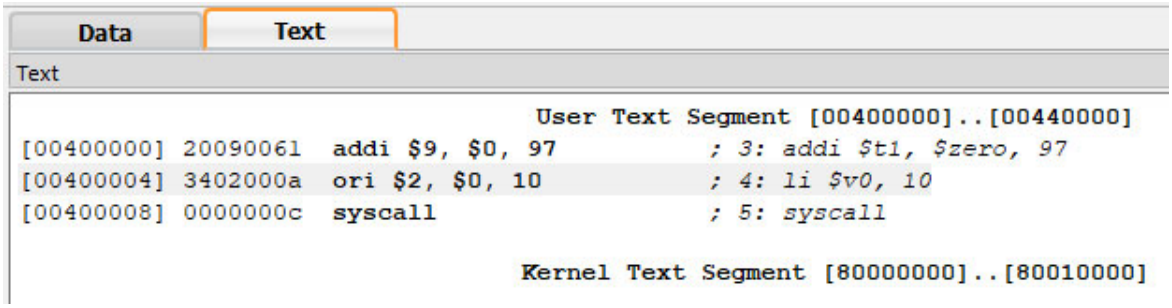


Note: If your *registers* values look different, you can change them by right-clicking anywhere in the registers frame and select "Hex".

What number base is the value in register **`$t1`**? **[1 pt]** is in hexadecimal

Text Segment **Frame**

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



For each line:

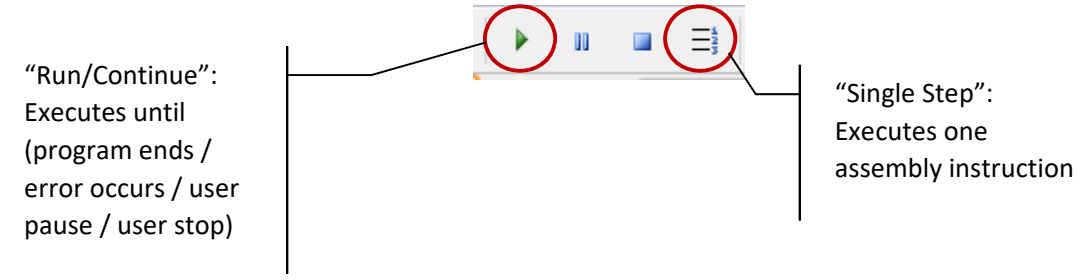
Column number	Example	Explanation
1	`[0x00400000]`	Memory address of the instruction. The default starting address is **`0x00400000`** for the first instruction.
2	`0x20090061`	The instruction encoding in hexadecimal, i.e. the machine code equivalent of this assembly instruction.
3	`addi $9, $0, 97`	Actual native assembly code.
4	`addi $t1, $zero`	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(s) was added 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.

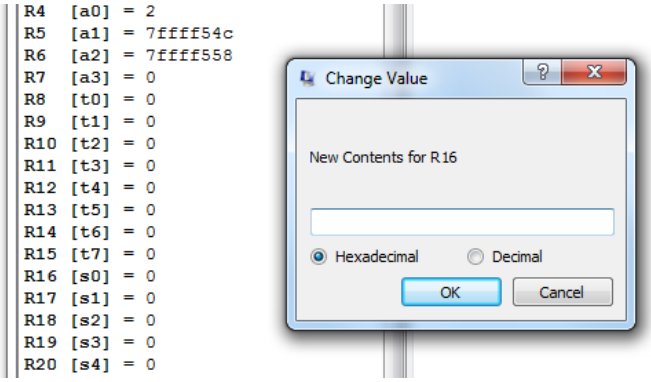


FAQ: I ran the code and encountered this "**Exception occurred at PC = 0x00000000**" error, what should I do?

Answer: Your QtSpim is not using address **0x00400000** as the default start address. 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**).



Writing a message: **sample2.asm**

Click on "File" → "Reinitialize and Load File" to open the file "**sample2.asm**"

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

```
# sample2.asm
      .data 0x10000100   specifies starting address of msg to be stored in memory
msg:  .asciiz "Hello"    stores a null terminated string in memory at the specified address
      .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 data is a string "Hello". The **.asciiz** directive stores a null-terminated string in memory. The string stored is just like a C string, i.e. a string that ends with a null character.

For your reference, the set of directives used in the lab exercises can be found 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** into register **rdest**.

These pseudo-instructions are converted into an equivalent sequence of actual machine instructions.

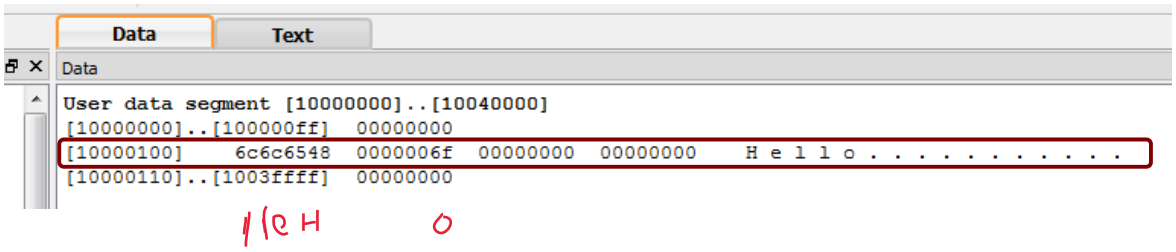
The **syscall** instruction makes a system call, and the value in register **$v0** indicates the type of call. For example:

- 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.

Data Segment

Data Segment is used to store constant values, variables content. 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**:



Try to figure out where and how is the string "Hello" stored. Give the ASCII values, in hexadecimal form, of the characters 'H', 'e', 'l' and 'o' in the report. [**2pts**]

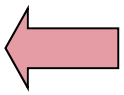
lui the first 4 bytes, then ori the last byte

Now, run the program (press the "Run/Continue" button) and report what do you see on the *Console* window? [**1 pt**]

Modifying a message: `sample3.asm`

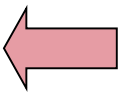
Click on "File" → "Reinitialize and Load File" to open **`sample3.asm`**. The file is the same as **`sample2.asm`** except a few comments added near the end of the main routine. We are going to edit the program to perform some simple tasks.

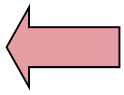
First, take some time to check your understanding of the **`li`** (load immediate) and **`la`** (load address) pseudo-instructions. Use the "Single Step" button to step through the program. Stop at the line of the first "**`syscall`**". Report the content of register **`$v0`** and **`$a0`** at this point. [**2pts**]

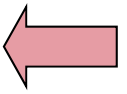


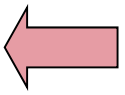
Let us now read a single character (i.e. single byte) from the string into a register. Edit the given program **`sample3.asm`** using your favorite editor as described below:

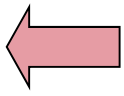
1. Add an instruction (**`lb`** – load byte) to load the value of 'o' (the last character in "Hello") into register **`$t0`**. Add the instruction after the first "**`syscall`**" instruction. Give the instruction in report.**[2pts]** If you need to re-run the new program, remember to save your edits, then reload the assembly file in QtSpim.
2. Now, we want to change the 'o' into 'O' (capital Oh). Let us first change the value in register **`$t0`** by simple arithmetic. What is the difference in ASCII value between 'o' and 'O'? [**1pt**]
3. Give the (**`addi`** – add immediate) instruction to change the value in **`$t0`** to the ASCII value of 'O'. [**1pt**]
4. Let us put the modified value back into the memory. Give the (**`sb`** – store byte) instruction to place the changed value of **`$t0`** into the position 'o' (i.e. overwriting the value in memory). [**2pts**] Take note of the change in data segment when you execute this instruction.
5. Now, just make another **`syscall`** to print the string again. Add the instruction **`syscall`** near the end of the program as indicated by the comment. What do you see in the ***Console Window*** when you run the program? [**2 pts**]











Demonstrate the completed program to your lab TA. [**2 pts**]

Submission Please zip the report document and the completed **sample3.asm** into **[student_id]_lab03.zip**, e.g. **A1234567X_lab03.zip**. Submit into the correct folder on Luminus before 2359 of your lab day.
Marking Scheme: Report – 14 marks; Demonstration – 2 marks; Total: 16 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.