Remember to bring this along to your lab!

CS2100 Computer Organisation
Lab #7: Exploring PCSpim
(22 – 26 March 2010)

[This document is available on course website http://www.comp.nus.edu.sg/~cs2100/3_ca/labs.html]

Name: ___________________________________ Matric. No: ______________________

Lab Group: _____

Objective

In this lab, you will explore the **PCSpim**, a MIPS simulator on the Microsoft Windows systems. This document and its associated files (**sample1.asm** and **sample2.asm**) can be downloaded from the course website given above.

Software and Documentation

The following documents are available on the **Online Resources** page of the CS2100 course website: http://www.comp.nus.edu.sg/~cs2100/2_resources/online.html

1. **PCSpim**: Download PCSpim and follow the installation instructions to install Spim in your system. In the lab, PCSpim is already installed.

   Note that when you first execute PCSpim, it might not locate the file "exceptions.s". Set the location of this file correct, by adding the pathname of the directory where you install PCSpim in.

2. **Getting Started with PCSpim**: This document provides an overview of the Microsoft Windows version of spim (PCSpim). It is also available on the CD that comes with the textbook under Tutorials.

3. **Assemblers, Linkers, and the SPIM Simulator**: An overview and reference manual for SPIM and MIPS32 instruction set. It is also available on the CD that comes with the textbook as Appendix A. The sections that you should definitely read are **Section A.9 (SPIM)** and **Section A.10 (MIPS R2000 Assembly Language).**

Introduction

SPIM is a software simulator that runs assembly language programs written for processors that implement the MIPS32 architecture. SPIM's name is just MIPS spelled backwards. SPIM can read and immediately execute assembly language files. 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 than a real computer (100 or more times!).

One interesting feature of SPIM assembly language is that it provides **pseudo-instructions**, which appear as real instructions in assembly language programs. The hardware, however, knows nothing about pseudo-instructions, so the assembler translates them into equivalent sequences of actual machine instructions.

Here are two examples of pseudo-instruction:

- 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`.

In MIPS assembly language, we will also find names that begin with a period, for example **`.data`** and **`.globl`**. These are **assembler directives** that tell the assembler how to translate a program but do not produce machine instructions. For example:

```
        .data
item:   .word 1
        .text
        .globl main       # Must be global
main:   lw         $t0, item
```

Numbers are decimal (base 10) by default. If they are preceded by **0x**, they are interpreted as hexadecimal (base 16). Hence, 256 and 0x100 denote the same value.

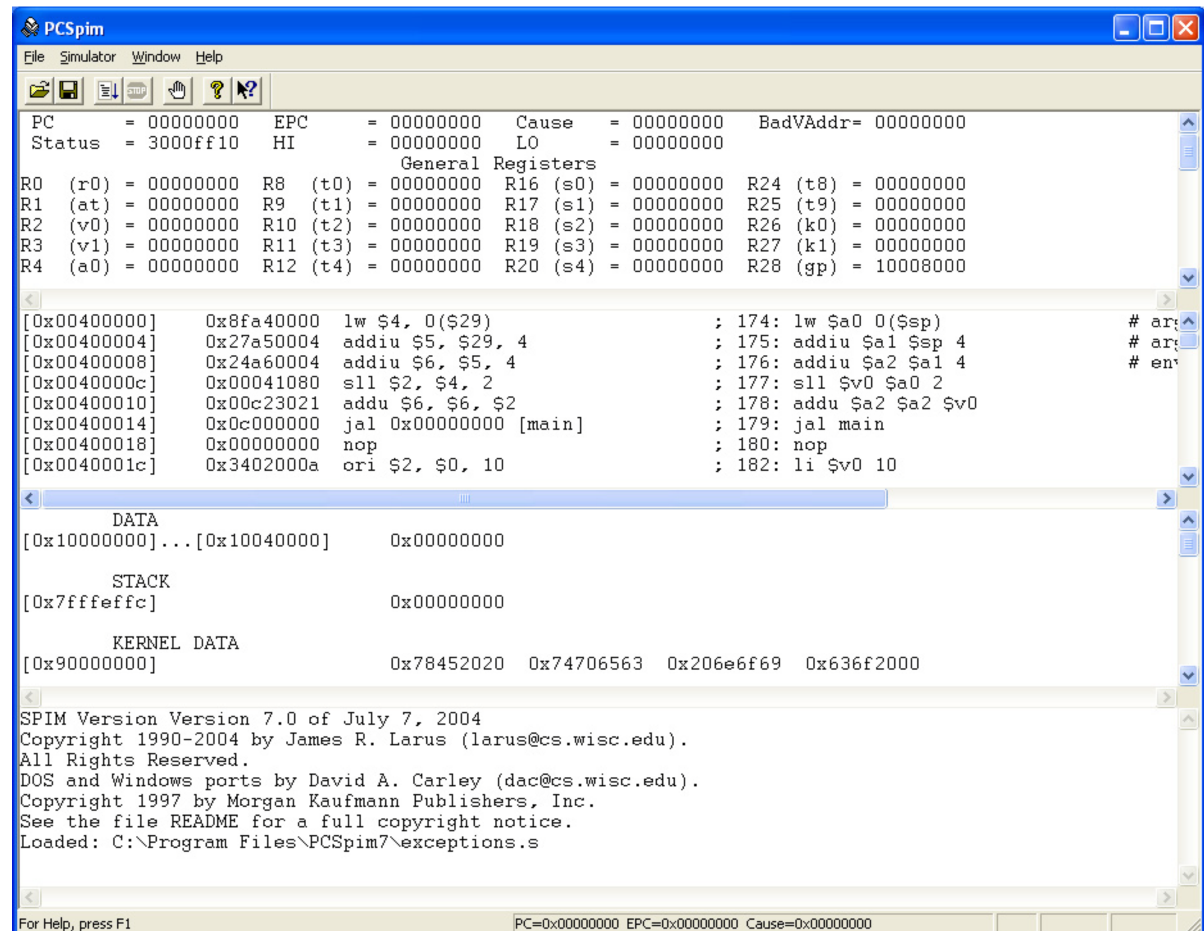
SPIM supports a subset of the MIPS assembler directives. The set of directives that will be used in the lab exercise are given below:

`.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.

Text segment and **Data segment** are two distinct sections in the object file produced by the assembler. The text segment contains the machine language code for routines in the source file. The data segment contains a binary representation of the data in the source file.

My First MIPS Program: sample1.asm

In the lab, you may click on the PCSpim shortcut at the desktop to invoke PCSpim. You should see the following screen.



The display is divided into four windows: *Registers*, *Text Segment*, *Data Segment*, and *Messages*.

- *Registers* window: This window shows the values of all registers in the MIPS CPU and FPU (floating point unit, which is not of interest to us). This display is updated whenever your program stops running.
- *Text Segment* window: This window displays instructions from both your program and the system code that is loaded automatically when PCSpim starts running.
- *Data Segment* window: This window displays the data loaded into your program's memory and the data on the program's stack.
- *Messages* window: This is where PCSpim writes messages. This is where error messages appear.

Besides the above, there is also a separate *Console* window for input and output.

A directory **C:\Program Files\PCSpim\cs2100** has been created for you. Download the file `sample1.asm` into this directory. Invoke PCSpim and click on "File" → "Open" to open this file. If you want to make any modification to `sample1.asm`, you can use your favourite text editor such as WordPad. The content of `sample1.asm` is shown below:

```
# sample1.asm
      .text
main: addi $t1, $zero, 97
```

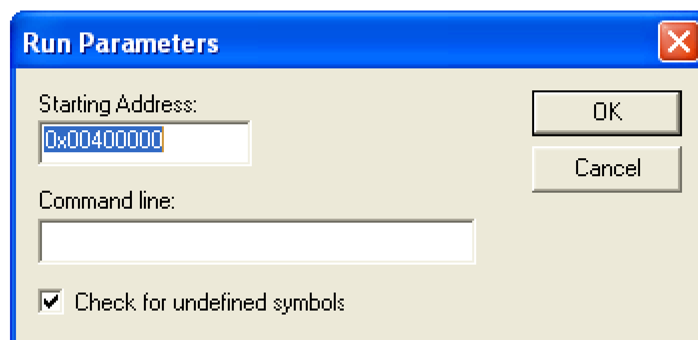
The first line is a comment line. The second line `.text` is the assembler directive that specifies the starting address of the source code. Since no starting address is specified, the default value 0x00400000 will be assumed. (The prefix 0x indicates hexadecimal number.)

The next line contains the single-line source code. The decimal value 97 is to be assigned to register `$t1`.

Observe the value of the register `$t1` (register 9) and the value of the PC (program counter) in the *Registers* window.

```
 PC     = 00400000     EPC     = 00000000     Cause    = 00000000     BadVAddr= 00000000
 Status = 00000000     HI      = 00000000     LO       = 00000000
                                  General Registers
R0  (r0) = 00000000  R8  (t0) = 00000000  R16 (s0) = 00000000  R24 (t8) = 00000000
R1  (at) = 00000000  R9  (t1) = 00000000  R17 (s1) = 00000000  R25 (t9) = 00000000
R2  (v0) = 00000000  R10 (t2) = 00000000  R18 (s2) = 00000000  R26 (k0) = 00000000
R3  (v1) = 00000000  R11 (t3) = 00000000  R19 (s3) = 00000000  R27 (k1) = 00000000
R4  (a0) = 00000000  R12 (t4) = 00000000  R20 (s4) = 00000000  R28 (gp) = 10008000
```

Now click on "Simulator" → "Go" (or press F5) to run your code. You will see this pop-up window:



Click "Ok" to accept the default starting address 0x00400000.

Now check the value of register `$t1` (register 9) in the *Registers* window again. What do you see? What number system is the value in? Answer: ________________________.

```
 PC     = 00000000     EPC     = 00000000     Cause    = 00000000     BadVAddr= 00000000
 Status = 00000000     HI      = 00000000     LO       = 00000000
                                  General Registers
R0  (r0) = 00000000  R8  (t0) = 00000000  R16 (s0) = 00000000  R24 (t8) = 00000000
R1  (at) = 00000000  R9  (t1) = 00000061  R17 (s1) = 00000000  R25 (t9) = 00000000
R2  (v0) = 00000000  R10 (t2) = 00000000  R18 (s2) = 00000000  R26 (k0) = 00000000
R3  (v1) = 00000000  R11 (t3) = 00000000  R19 (s3) = 00000000  R27 (k1) = 00000000
R4  (a0) = 00000000  R12 (t4) = 00000000  R20 (s4) = 00000000  R28 (gp) = 10008000
```

Note: If your *Register* window doesn't appear the same as above, that is, the values are shown in decimal, it means that the setting has been changed. Go to "Simulator" → "Setting" to make the necessary change so that the *Register* window resembles the above.

Text Segment Window

Click inside the *Text Segment* window and use the <Page Down> key or move the scroll bar downward until you see your one-line code amidst the lines:

```
[0x00400008]    0x24a60004  addiu $6, $5, 4            ; 142: addiu $a2, $a1, 4    # envp
[0x0040000c]    0x00041080  sll $2, $4, 2              ; 143: sll $v0, $a0, 2
[0x00400010]    0x00c23021  addu $6, $6, $2            ; 144: addu $a2, $a2, $v0
[0x00400014]    0x0c100009  jal 0x00400024 [main]      ; 145: jal main
[0x00400018]    0x00000000  nop                        ; 146: nop
[0x0040001c]    0x3402000a  ori $2, $0, 10             ; 148: li $v0 10
[0x00400020]    0x0000000c  syscall                    ; 149: syscall                  # sy
[0x00400024]    0x20090061  addi $9, $0, 97            ; 3: addi $t1, $zero, 97
```

For each line,

- the first column – `[0x00400024]` – is the memory location (address);
- the second column – `0x20090061` – is the hexadecimal version of instruction;
- the third column – `addi $9, $0, 97` – is the native code; and
- the fourth column – `addi $t1, $zero, 97` – is your source code.

Everything following the semicolon is the actual line from your assembly file that produced the instruction. The number "3:" is the line number in that file. Sometimes nothing is on the line after the semicolon. This means that the instruction was produced by SPIM as part of translating a pseudo-instruction into more than one real MIPS instruction.

Note that the first line of your source code begins at address 0x00400024. The lines above it are for system set-up.

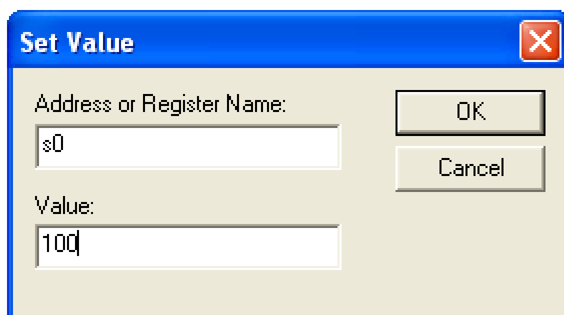
Stepping Through

Now, click "Simulator" → "Reload …\sample1.asm" to reload `sample1.asm`. Click "Simulator" → "Single Step" (or F10) to step through the code one line at a time. Observe the change in the content of PC as you step through the code, and the change in `$t1` (R9) as you step over the `addi` instruction.

To step through the code multiple lines at a time, use "Simulator" → "Multiple Step" (or F11) and enter the number of steps desired.

Setting Value into a Register

You may click "Simulator" → "Set Value" to assign value into a register of your choice. For example, entering `R16` or `s0` in the "Address or Register Name" field and `100` in the "Value" field of the pop-up window, as shown below, will assign decimal value 100 (or hexadecimal 64) into register $s0. You may enter the value in hexadecimal form, for example, using `0x64` instead of 100.



Writing a message: sample2.asm

Download `sample2.asm` into your working directory. Click on “File” → “Open” 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
```

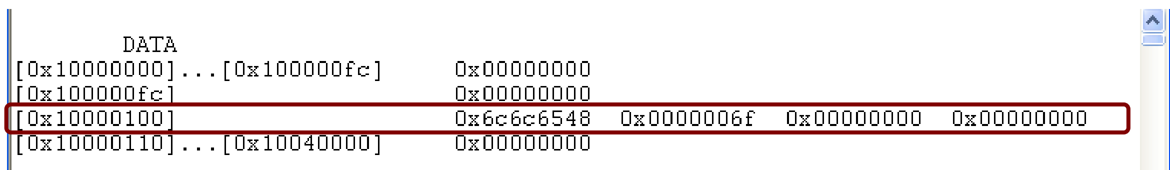
The second line `.data` is an assembler directive that specifies the starting address (0x10000100) of the data (in this case, the string “Hello”). The `.asciiz` directive stores a null-terminated string (a string that ends with a null character) in memory.

There are two pseudo-instructions in this code:

- 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`.

The `syscall` instruction is 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 concerned is retrieved from `$a0`.

Look at the *Data Segment* window and read the information on the line concerning memory address 0x10000100:



Can you figure out where is the string “Hello” stored? Write out the ASCII values, in hexadecimal form, of the characters ‘H’, ‘e’, ‘l’ and ‘o’ below:

‘H’: ____ ‘e’: ____ ‘l’: ____ ‘o’: ____

The PCSpim simulator assumes the byte order of the underlying architecture, which is Pentium in our case. Are the data stored in *big-endian order* or *little-endian order*?

Answer: ______________________

Now, run the program (“Simulator” → “Go” or press F5). What do you see on the *Console* window?

Answer: ______________________

Marking Scheme: Attendance (5 marks), Report/Demonstration (10 marks); Total: 15 marks.