CS2100: Computer Organisation Lab #3: Exploring QtSpim

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) are all available on Canvas and the course 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 \rightarrow "Resources" \rightarrow "Online", or <u>https://www.comp.nus.edu.sg/~cs2100/2_resources/online.html</u> directly.

1. **QtSpim software**: Download the <u>latest version</u> (should be 9.1.24 last time I checked) of 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.

Note: MacOS users may encounter this error:



See <u>https://support.apple.com/en-sg/guide/mac-help/mh40616/mac</u> for how to get around this.

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



BEFORE YOU BEGIN, please read the (very) frequently asked question below.

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.

Let's give it a try! Before you start, make sure you have downloaded the 3 assembly programs **sample1.asm**, **sample2.asm** and **sample3.asm** from Canvas or the course website to 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 \rightarrow Settings" (or **QtSpim->Preferences** on Macs) 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".

MIPS Simulation Set	tings	
📃 Bare Machine		Accept Pseudo Instructions
Enable Delaye	ed Branches	Enable Delayed Loads
Enable Mappe	ed IO	
(Simple Machine	Bare Machine
Exception Handler		
I load Exception	on Handler	File 2014 Exception Handlers >
	of the function	The SPENEXCEPTION Handler >> (

Click "OK" to save your setting. Go to "File \rightarrow 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 \rightarrow Console" to enable it.

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

<pre># sample1.asm .text main: addi \$t1, \$zero, 97 li \$v0 10</pre>	addi \$9, \$0, 97 ; 3: addi \$t1, \$zero, 97 ori \$2, \$0, 10 ; 4: li \$v0, 10 syscall ; 5: syscall
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 th	he "Single Step"	button to	execute	one	assembly	statement.
	\					

IntRegs[16] F × Text PC = 400028 User Text Segment [0040000][00440000] EPC = 0 [00400001] & \$	FP Req	s Int Regs [16]		Data	Tex	t		
PC = 400028 User Text Segment [0040000][00440000] EPC = 0 [0040000] 8fa40000 !w \$4, 0(\$29) ; 183: lw \$a0 0(\$sp) # arg Cause = 0 [0040000] 2735004 addiu \$5, \$29, 4 ; 184: addiu \$a1 \$sp 4 # arg BadVAdar = 0 [0040000] 2735004 addiu \$5, \$29, 4 ; 184: addiu \$a1 \$sp 4 # arg Status = 3000ff10 HI = 0 [0040000] 00c23021 addu \$6, \$6, \$2 ; 187: addu \$a2 \$a2 \$x0 L0 = 0 [0040001] 00c23021 addu \$6, \$6, \$2 ; 187: addu \$a2 \$a2 \$x0 [00400013] 0000000 is 00000000 nop ; 188: jal main [00400014] 0c100009 jal 0x00400024 [main] ; 188: jal main [00400015] 3402000a ori \$2, \$0, 10 ; 191: li \$v0 10 [00400020] 000000c syscall ; 192: syscall \$ syscall 1 [00400021] 2090061 addi \$9, \$0, 97 ; 3: addi \$1, \$zero, 97 [00400022] 000000c syscall ; 5: syscall \$0, 10 ; 1: \$v0, 10 [00400022] 1 3402000a ori \$2, \$0, 10 ; 4: li \$v0, 10 [00400022] 34020000 ori \$2, \$0, 10 <t< th=""><th>t Regs [16]</th><th></th><th>₽×</th><th>Text</th><th></th><th></th><th></th><th></th></t<>	t Regs [16]		₽×	Text				
EPC = 0 Cause = 0 BadVAddr = 0 Status = 3000ff10 HI = 0 L0 = 0 BadVAddr = 0 Status = 3000ff10 HI = 0 L0 = 0 R0 [r0] = 0 R0 [r0] = 0 R1 [at] = 0 R2 [v0] = 4 [v1] = 0 [0040002] 3402000a R3 [v1] = 0 R4 (status) = 10 R3 [v1] = 0 R4 [0040002] 3402000a ori \$2, \$0, 10 R4 [v1] = 0 R4 [0040002] 3402000a ori \$2, \$0, 10 R4 [v1] = 0 R4 [0040002] 3402000a ori \$2, \$0, 10 R4 [v1] = 0 R4 [v1] = 0 R4 [0040002] 3402000a ori \$2, \$0, 10 [0040002] 3402000a ori \$2, \$0, 10 \$4 11 \$v0, 10	С	= 400028	^	(F)		User Text	Segment [004	100000][00440000]
Cause = 0 BadVAddr = 0 BadVAddr = 0 Status = 3000ff10 HI = 0 LO = 0 R0 [r0] R0 [r0] R1 [at] ati [at] ati [at] BadVAddr = 0 ID = 0 R0 [r0] R0 [r0] R1 [at] B2 [v0] R1 [at] R2 [v0] R3 [v1] R4 [v1] R3 [v1] R4 [004002c] 3402000a cri §2, \$0, 10 ; 4: li \$v0, 10 [0040002c] 000000c syscall ; 192: syscall \$zero, 97 [0040002c] 3402000a cri §2, \$0, 10 ; 4: li \$v0, 10 [0040002c] 000000cc syscall ; 5 syscall	PC	= 0		[00400000]	8fa40000	lw \$4, 0(\$29)	; 183: lv	\$a0 0(\$sp) # argc
BadVAddr = 0 [00400008] 24a60004 addiu \$6, \$5, 4 : 185: addiu \$a2 \$a1 4 # e Status = 3000ff10 [00400002] 00041080 sll \$2, \$4, 2 : 185: addiu \$a2 \$a2 4 # e HI = 0 [00400002] 00041080 sll \$2, \$4, 2 : 185: addiu \$a2 \$a2 \$a2 \$v0 LO = 0 [00400012] 00023021 addu \$6, \$6, \$2 : 187: addu \$a2 \$a2 \$a2 \$v0 R0 [r0] = 0 [00400018] 0000000 nop : 188: nop R1 [at] = 0 [00400012] 3402000a ori \$2, \$0, 10 : 191: li \$v0 10 R2 [v0] = 4 [00400024] 2009061 addi \$9, \$0, 97 : 3: addi \$t1, \$zero, 97 R3 [v1] = 0 [0040002c] 000000c syscall : 5: syscall	ause	= 0		[00400004]	27a50004	addiu \$5, \$29, 4	; 184: ad	diu \$a1 \$sp 4 # argv
Status = 3000ff10 HI = 0 LO = 0 R0 [r0] = 0 R1 [at] = 0 R2 [v0] = 4 [v1] = 0 [0040002] 000000 R3 [v1] = 0 R4 [v1] = 0 R5 [v1] = 0 R4 [v2] syscall R5 [v1] = 0 R6 [v1] = 0 R4 [v2] syscall R5 [v1] = 0 R5 [v1] = 0 R5 [v2] syscall R5 [v3] support R5 [v4] support R5 [v5] support R5 [v5] support R5 [v2] support R5 [v3] support R5 [v4] support R5 [v5] support R5 [v4] support R5 [v4] support R5 [v5] support R5 [v5] support R5 [v4] support R5 [v5] support R5 [v5] support	adVAddr	= 0		[00400008]	24a60004	addiu \$6, \$5, 4	; 185: ad	diu \$a2 \$a1 4 # envp
HI = 0 [00400010] 00c23021 addu \$6, \$6, \$2 ; 187; addu \$a2 \$a2 \$v0 LO = 0 [00400010] 00c23021 addu \$6, \$6, \$2 ; 187; addu \$a2 \$a2 \$v0 R0 = 0 [00400014] 0c10000 jal 0x00400024 [main] ; 188; jal main [00400012] 3000000 nop ; 189; nop [00400012] 3402000a ori \$2, \$0, 10 ; 191: li \$v0 10 [00400020] 000000c syscall ; 192; syscall # syscall 1 [00400021] 2009061 addi \$9, \$0, 97 ; 3: addi \$11, \$zero, 97 [00400022] 000000c syscall ; 15v0, 10 [0040002c] 000000c syscall ; 5vyscall	tatus	= 3000ff10		[0040000c]	00041080	sll \$2, \$4, 2	; 186: sl	1 <mark>\$v0 \$a0 2</mark>
HI = 0 LO = 0 R0 [r0] = 0 R1 [at] = 0 R2 [v0] = 4 R3 [v1] = 0 R3 [v1] = 0 R4 [v1] = 0 R5	vite.	CONT. NO.		[00400010]	00c23021	addu \$6, \$6, \$2	; 187: ad	du \$a2 \$a2 \$v0
LO = 0 [[00400018] 0000000 nop ; 189: nop [[0040001c] 3402000a ori \$2, \$0, 10 ; 191: li \$v0 10 [[00400020] 000000c syscall ; 192: syscall # syscall 1 [[00400024] 20090061 addi \$9, \$0, 97 ; 3: addi \$t1, \$zero, 97 [[00400028] 3402000a ori \$2, \$0, 10 ; 4: li \$v0, 10 [[0040002c] 000000c syscall ; 5: syscall	I	= 0		[00400014]	0c100009	jal 0x00400024 [main]	; 188: ja	l main
R0 [r0] = 0 [0040001c] 3402000a ori \$2, \$0, 10 ; 191: li \$v0 10 R1 [at] = 0 [00400020] 000000c syscall ; 192: syscall # syscall 1 R2 [v0] = 4 [00400024] 20090061 addi \$9, \$0, 97 ; 3: addi \$t1, \$zero, 97 R3 [v1] = 0 [0040002c] 000000c syscall ; 4: li \$v0, 10 rest [v0] = 4 [0040002c] 000000c syscall ; 5: syscall	0	= 0		[00400018]	00000000	nop	; 189: no	p
RU [r0] = 0 [00400020] 000000c syscall ; 192: syscall # syscall 1 R1 [at] = 0 [00400024] 20090061 addi \$9, \$0, 97 ; 3: addi \$11, \$zero, 97 R2 [v0] = 4 [00400024] 20090061 addi \$9, \$0, 10 ; 4: li \$v0, 10 R3 [v1] = 0 [0040002c] 000000c syscall : 5: syscall	0 1-01			[0040001c]	3402000a	ori \$2, \$0, 10	; 191: li	\$v0 10
R1 [at] = 0 [00400024] 20090061 addi \$9, \$0, 97 ; 3: addi \$t1, \$zero, 97 R2 [v0] = 4 [00400028] 3402000a ori \$2, \$0, 10 ; 4: li \$v0, 10 R3 [v1] = 0 [0040002c] 000000c syscall ; 5: syscall	1 [ru]	= 0		[00400020]	000000c	syscall	; 192: sy	scall # syscall 10 (exit
[v0] - 1 [00400028] 3402000a ori \$2, \$0, 10 ; 4: li \$v0, 10 [R3 [v1] = 0 [00400028] 0000000c syscall ; 5: syscall	2 [m0]	= 0		[00400024]	20090061	addi \$9, \$0, 97	; 3: addi	\$t1, \$zero, 97
[0040002c] 000000c syscall : 5: syscall	2 [V0] 3 [v1]	= 0		[00400028]	3402000a	ori \$2, \$0, 10	; 4: li \$	v0, 10
	4 [a0]	= 1		[0040002c]	0000000c	syscall	; 5: sysc	all
R5 [a1] = 7fff868	5 [a1]	= 7ffff868		41 6954		92702	999 (BB)	7
Re [a2] = 7ffff870	6 [a2]	= 7ffff870	stor if y	our Doai	ctor from	aa daaca't annaar ti		000000][80010000]
R7 [a3] = 0 [Note: If your <i>Register</i> frame doesn't appear the same ski sat # save	7 [a3]	= 0	ote: ir y	our Regis	ster fram	ne doesn't appear ti	ie same	\$k1 \$at # Save \$at

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:	
---	--

Text Segment Frame

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

	Data	Text		
₽×	Text			
^	[00400024] 2	20090061	addi \$9, \$0, 97	; 3: addi \$t1, \$zero, 97
	[00400028] 3	3402000a	ori \$2, \$0, 10	; 4: li \$v0, 10
	[0040002c] (000000c	syscall	; 5: syscall

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

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]	=	7fff558	
R7	[a3]	=	0	🐐 Change Value
R8	[t0]	=	0	
R9	[t1]	=	0	
R10	[t2]	=	0	
R11	[t3]	=	0	New Contents for R16
R12	[t4]	=	0	
R13	[t5]	=	0	
R14	[t6]	=	0	
R15	[t7]	=	0	Hexadecimal Decimal
R16	[s0]	=	0	
R17	[s1]	=	0	OK Cancel
R18	[s2]	=	0	
R19	[s3]	=	0	
R20	[s4]	=	0	

Writing a message: sample2.asm

Download sample2.asm into your working directory. Click on "File" \rightarrow "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 <u>assembler directives</u> 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. Note that the use of pseudo-instruction is usually not allowed in test/exam, so please don't get overly reliant on them O.

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**:

		Data	Text														
8	x	Data															
	*	User data se	gment [1000	0000][10	040000]												
		[10000000]	[100000ff]	00000000													
		[10000100]	6c6c6548	0000006f	00000000	00000000	H	e	1	1	0	•	 •	•	•	•	
		[10000110]	[1003ffff]	00000000													_
		[10000110]	[1003ffff]	00000000													

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':_____ 'e':_____ 'l':_____ 'o':_____

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:

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 v_0 and a at this point?

Answer: **\$v0** = **\$a0** =

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 (1b – 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: 1b \$t0,

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:

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

Answer: **addi**

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**

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:

Marking Scheme: Report/Demonstration (20 marks).

Appendix

QtSPIM assembler directives:

.data ≺addr>	Subsequent items are stored in the data segment. If the optional argument <i>addr</i> is present, subse- quent items are stored starting at address <i>addr</i> .
.globl sym	Declare that label <i>sym</i> is global and can be refer- enced from other files.
.ascii str	Store the string <i>str</i> in memory, but do not null- terminate it.
.asciiz str	Store the string <i>str</i> in memory and null-termi- nate it.
.text ≺addr>	Subsequent items are put in the user text seg- ment. In SPIM, these items may only be instruc- tions or words (see the .word directive below). If the optional argument <i>addr</i> is present, subse- quent items are stored starting at address <i>addr</i> .
.word w1,, wn	Store the n 32-bit quantities in successive memory words.