IT5002 Computer Systems and Applications

Assignment 1 – Assembly Programming

1. Introduction

You may complete this assignment singly or in pairs. If you complete in pairs, you should submit only ONE copy of your assignment. Ensure that the names of both partners are shown in the Answer Book.

As this assignment is quite long, you are advised to do it with a partner rather than on your own.

This assignment is worth 40 marks.

In this assignment we will be using SPIM, the MIPS simulator, to explore how high-level language structures can be implemented in assembly. While we will not be looking at the intricacies of how compilers work (they are very complex), we will get a feel on how variables are created and used in programs.

SPIM (which is simply MIPS spelt backwards) takes MIPS assembly code in a file, assembles it to produce the machine code, then runs it in the simulator.

This assignment will help you understand how programs are actually built in assembly, and how they can access operating system features like printing to the screen or reading the keyboard. It serves as a good introduction to the next part of this course.

SUBMISSION DEADLINE

Save your report as AxxxxxxY.pdf where AxxxxxxY is the student ID of the person submitting. You DO NOT need to submit your print.asm file. **If you are working with a partner, only ONE person should submit.**

Upload your PDF to Canvas by FRIDAY 10 OCTOBER 2025 2359. You can submit until 11 October 00:30 hours, after which **NO SUBMISSION WILL BE ACCEPTED REGARDLESS OF REASON.**

PLAGIARISM AND USE OF AI POLICY

Members of the same pair of students should submit only one copy. No student is to copy and submit the work, whether report or code, of another student in any way. **No use of any form artificial intelligence, including but not limited to Large Language Models, is permitted.**

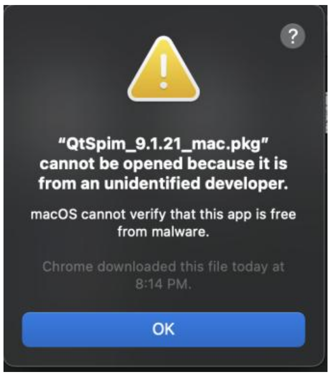
Disciplinary action will be taken against any student who infringes these policies.

2. Obtaining and Installing SPIM

You can download SPIM at https://sourceforge.net/projects/spimsimulator/files/. The current version is 9.1.24, and you can download for MacOS, Windows and Linux. Please choose the version that is most suited to your computer, and follow the installation instructions.

Special note for MacOS users:

You may see this message when attempting to install/run SPIM:



"QtSpim_9.1.21_mac.pkg" cannot be opened because it is from an unidentified developer.

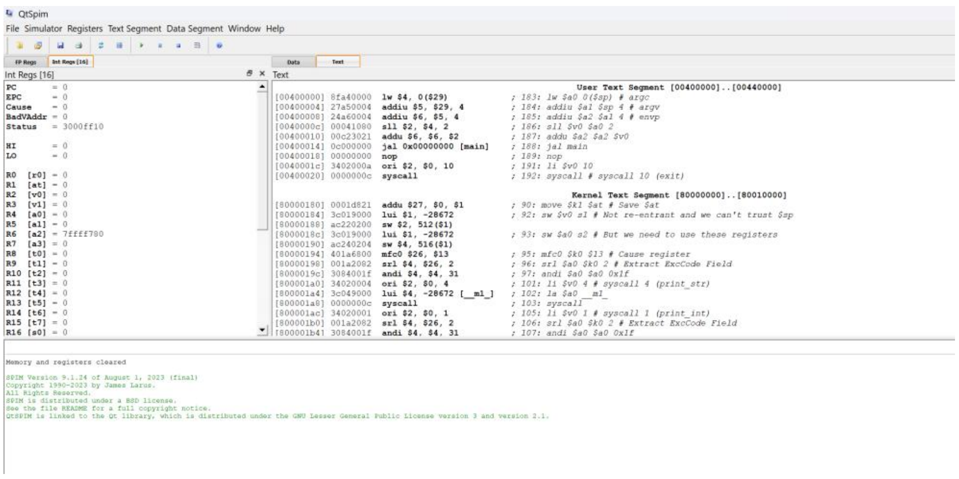
macOS cannot verify that this app is free from malware.

Chrome downloaded this file today at 8:14 PM.

Please see https://support.apple.com/en-sg/guide/mac-help/mh40616/mac for how to deal with this issue.

3. Using QTSpim

Start up QTSpim. You will see a screen similar to this:

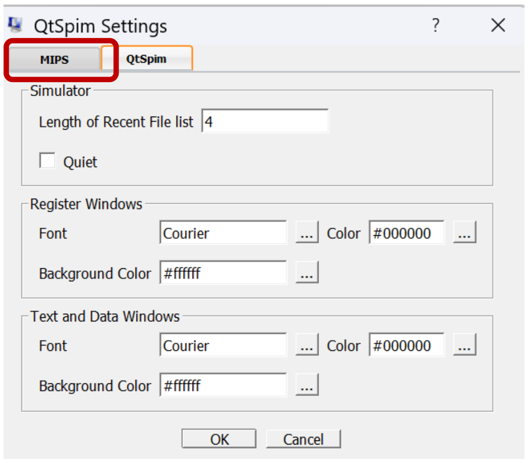

QtSpim
File Simulator Registers Text Segment Data Segment Window Help
FP Regs
Int Regs [16]
Data
Text

```
Int Regs [16]
PC      = 0
EPC     = 0
Cause   = 0
BadVAddr = 0
Status  = 3000ff10

HI      = 0
LO      = 0

R0  [r0] = 0
R1  [at] = 0
R2  [v0] = 0
R3  [v1] = 0
R4  [a0] = 0
R5  [a1] = 0
R6  [a2] = 7ffff780
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
```

```
Text
                                                User Text Segment [00400000]..[00440000]
[00400000] 8fa40000  lw $4, 0($29)              ; 183: lw $a0 0($sp) # argc
[00400004] 27a50004  addiu $5, $29, 4           ; 184: addiu $a1 $sp 4 # argv
[00400008] 24a60004  addiu $6, $5, 4            ; 185: addiu $a2 $a1 4 # envp
[0040000c] 00041080  sll $2, $4, 2              ; 186: sll $v0 $a0 2
[00400010] 00c23021  addu $6, $6, $2            ; 187: addu $a2 $a2 $v0
[00400014] 0c000000  jal 0x00000000 [main]      ; 188: jal main
[00400018] 00000000  nop                        ; 189: nop
[0040001c] 3402000a  ori $2, $0, 10             ; 191: li $v0 10
[00400020] 0000000c  syscall                    ; 192: syscall # syscall 10 (exit)

                                               Kernel Text Segment [80000000]..[80010000]
[80000180] 0001d821  addu $27, $0, $1           ; 90: move $k1 $at # Save $at
[80000184] 3c019000  lui $1, -28672             ; 92: sw $v0 s1 # Not re-entrant and we can't trust $sp
[80000188] ac220200  sw $2, 512($1)
[8000018c] 3c019000  lui $1, -28672             ; 93: sw $a0 s2 # But we need to use these registers
[80000190] ac240204  sw $4, 516($1)
[80000194] 401a6800  mfc0 $26, $13              ; 95: mfc0 $k0 $13 # Cause register
[80000198] 001a2082  srl $4, $26, 2             ; 96: srl $a0 $k0 2 # Extract ExcCode Field
[8000019c] 3084001f  andi $4, $4, 31            ; 97: andi $a0 $a0 0x1f
[800001a0] 34020004  ori $2, $0, 4              ; 101: li $v0 4 # syscall 4 (print_str)
[800001a4] 3c049000  lui $4, -28672 [__m1_]     ; 102: la $a0 __m1_
[800001a8] 0000000c  syscall                    ; 103: syscall
[800001ac] 34020001  ori $2, $0, 1              ; 105: li $v0 1 # syscall 1 (print_int)
[800001b0] 001a2082  srl $4, $26, 2             ; 106: srl $a0 $k0 2 # Extract ExcCode Field
[800001b4] 3084001f  andi $4, $4, 31            ; 107: andi $a0 $a0 0x1f
```

```
Memory and registers cleared

SPIM Version 9.1.24 of August 1, 2023 (final)
Copyright 1990-2023 by James Larus.
All Rights Reserved.
SPIM is distributed under a BSD license.
See the file README for a full copyright notice.
QtSPIM is linked to the Qt library, which is distributed under the GNU Lesser General Public License version 3 and version 2.1.
```

You need to set up QTSpim to model the simpler MIPS machine that we are using in our course. To do so, click Simulator->Settings, which will bring up this dialog box. Click on the "MIPS tab":

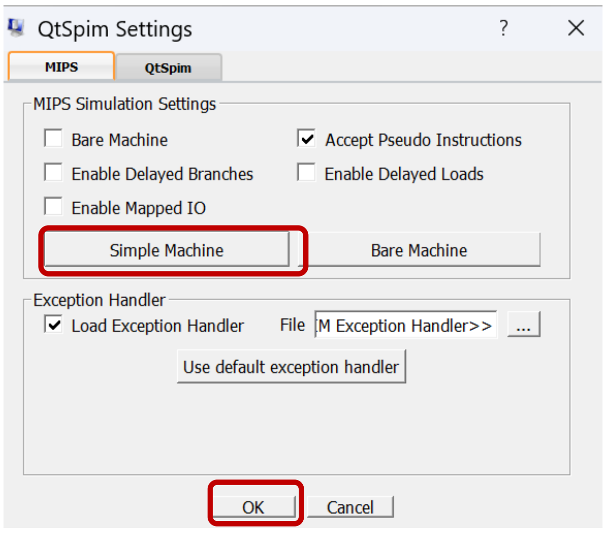


Simulator
Length of Recent File list: 4
Quiet: ☐

Register Windows
Font: Courier | Color: #000000
Background Color: #ffffff

Text and Data Windows
Font: Courier | Color: #000000
Background Color: #ffffff

On the MIPS tab, click on the Simple Machine button, then click OK:



MIPS Simulation Settings
☐ Bare Machine | ☑ Accept Pseudo Instructions
☐ Enable Delayed Branches | ☐ Enable Delayed Loads
☐ Enable Mapped IO

Exception Handler
☑ Load Exception Handler | File: M Exception Handler>>

There are two sample files sample1.asm and sample2.asm provided to you. The code for sample1.asm is shown below:

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

In the later part of IT5002 you will learn that when a program is loaded into memory, the Operating System (OS), like Windows 11 or MacOS, it uses memory in several different ways:

Memory Type	**What it is used for**
Text	Program code
Data	Global variables
Stack	Local variables and function arguments
Heap	Dynamically allocated variables (Not used here)

Here the .text line in sample1.asm tells the QTSpim assembler that the instructions that follow should be put into the "text segment" of the program's memory.

The next instruction is "li $v0, 10", which is a pseudo-instruction that loads the immediate value 10 into $v0, and is equivalent to addi $v0, $zero, 10 or ori $v0, $zero, 10.

The last line "syscall" is called a "trap instruction", and for now it is sufficient to understand that its job is to call the OS. QTSpim provides the following OS calls:

Service	System call code	Arguments	Result
`print_int`	1	`$a0` = integer	
`print_float`	2	`$f12` = float	
`print_double`	3	`$f12` = double	
`print_string`	4	`$a0` = string	
`read_int`	5		integer (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	address (in `$v0`)
`exit`	10		
`print_char`	11	`$a0` = char	
`read_char`	12		char (in `$a0`)
`open`	13	`$a0` = filename (string), `$a1` = flags, `$a2` = mode	file descriptor (in `$a0`)
`read`	14	`$a0` = file descriptor, `$a1` = buffer, `$a2` = length	num chars read (in `$a0`)
`write`	15	`$a0` = file descriptor, `$a1` = buffer, `$a2` = length	num chars written (in `$a0`)
`close`	16	`$a0` = file descriptor	
`exit2`	17	`$a0` = result	

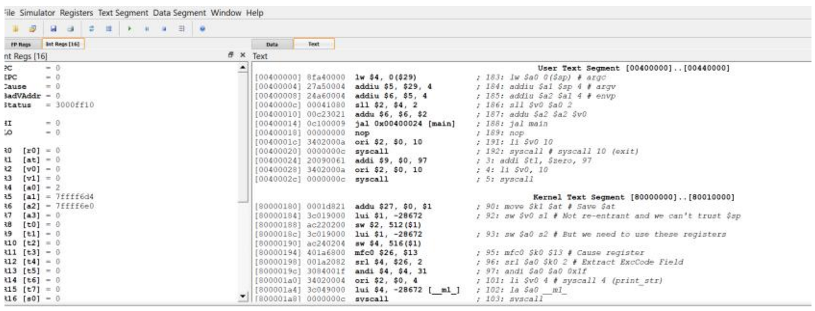
Table 1. OS Calls in QTSpim.

Here, the two statements:

```
li $v0, 10
syscall
```

Simply terminates the program.

To load and run sample1.asm, click File->Reinitialize and Load File, then navigate to your sample1.asm, and double click on the filename. Your screen will now look like this:


ile Simulator Registers Text Segment Data Segment Window Help
FP Regs
Int Regs [16]
Data
Text

```
nt Regs [16]
PC       = 0
EPC      = 0
Cause    = 0
BadVAddr = 0
Status   = 3000ff10

HI       = 0
LO       = 0

R0  [r0] = 0
R1  [at] = 0
R2  [v0] = 0
R3  [v1] = 0
R4  [a0] = 2
R5  [a1] = 7ffff6d4
R6  [a2] = 7ffff6e0
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
```

```
Text
                                User Text Segment [00400000]..[00440000]
[00400000] 8fa40000  lw $4, 0($29)             ; 183: lw $a0 0($sp) # argc
[00400004] 27a50004  addiu $5, $29, 4          ; 184: addiu $a1 $sp 4 # argv
[00400008] 24a60004  addiu $6, $5, 4           ; 185: addiu $a2 $a1 4 # envp
[0040000c] 00041080  sll $2, $4, 2             ; 186: sll $v0 $a0 2
[00400010] 00c23021  addu $6, $6, $2           ; 187: addu $a2 $a2 $v0
[00400014] 0c100009  jal 0x00400024 [main]     ; 188: jal main
[00400018] 00000000  nop                       ; 189: nop
[0040001c] 3402000a  ori $2, $0, 10            ; 191: li $v0 10
[00400020] 0000000c  syscall                   ; 192: syscall # syscall 10 (exit)
[00400024] 20090061  addi $9, $0, 97           ; 3: addi $t1, $zero, 97
[00400028] 3402000a  ori $2, $0, 10            ; 4: li $v0, 10
[0040002c] 0000000c  syscall                   ; 5: syscall

                                Kernel Text Segment [80000000]..[80010000]
[80000180] 0001d821  addu $27, $0, $1          ; 90: move $k1 $at # Save $at
[80000184] 3c019000  lui $1, -28672            ; 92: sw $v0 s1 # Not re-entrant and we can't trust $sp
[80000188] ac220200  sw $2, 512($1)
[8000018c] 3c019000  lui $1, -28672            ; 93: sw $a0 s2 # But we need to use these registers
[80000190] ac240204  sw $4, 516($1)
[80000194] 401a6800  mfc0 $26, $13             ; 95: mfc0 $k0 $13 # Cause register
[80000198] 001a2082  srl $4, $26, 2            ; 96: srl $a0 $k0 2 # Extract ExcCode Field
[8000019c] 3084001f  andi $4, $4, 31           ; 97: andi $a0 $a0 0x1f
[800001a0] 34020004  ori $2, $0, 4             ; 101: li $v0 4 # syscall 4 (print_str)
[800001a4] 3c049000  lui $4, -28672 [__m1_]    ; 102: la $a0 __m1_
[800001a8] 0000000c  syscall                   ; 103: syscall
```

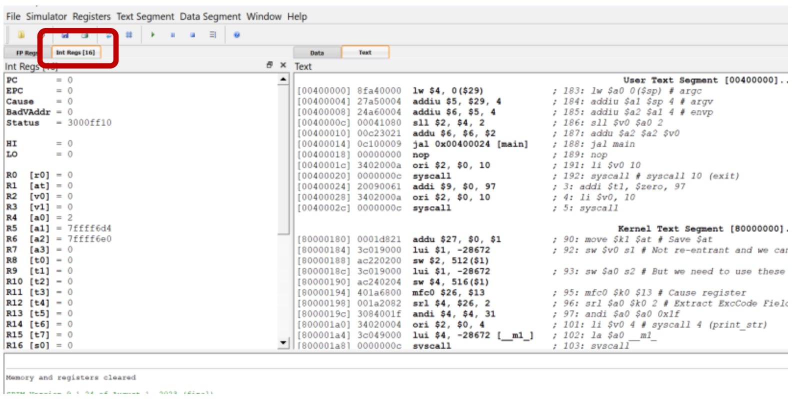
Execution will begin at address 0x00400000, which is code that a C compiler would introduce to extract command-line arguments, which is irrelevant to us here. At line 0x00400014, we see a "jal 0x00400024".
The "jal" instruction is a "jump-and-link" instruction which is the MIPS equivalent of a function all.

When “jal” is executed, the return address 0x00400018 will be placed into a register called $ra. We will look more at this later on. The main point here is that at the end of the function, it is possible to do a “jr $ra” to jump back to the return address, thus performing a function return.
Address 0x00400024 is the entry point “main” for our program, and the actual assembly looks like this:

```
[00400024] 20090061  addi $9, $0, 97          ; 3: addi $t1, $zero, 97
[00400028] 3402000a  ori $2, $0, 10           ; 4: li $v0, 10
[0040002c] 0000000c  syscall                  ; 5: syscall
```

The original source code is shown next to the actual assembly instructions. Note how li $v0, 10 has been translated to ori $2, $0, 10 (i.e. ori $v0, $zero, 10)

On the left pane you will see the register contents. There are two sets of registers, each on a different tab. The FP Regs tab shows the contents of the Floating Point registers, and the Int Regs tabe shows the contents of the Integer Registers. Since we did not study the Floating Point datapath, we will concern ourselves only with the Integer Registers:


File Simulator Registers Text Segment Data Segment Window Help
FP Regs
Int Regs [16]
Data
Text
Int Regs [16]
Text

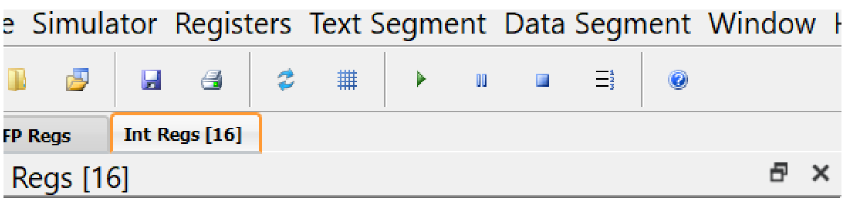
```
PC       = 0
EPC      = 0
Cause    = 0
BadVAddr = 0
Status   = 3000ff10

HI       = 0
LO       = 0

R0  [r0] = 0
R1  [at] = 0
R2  [v0] = 0
R3  [v1] = 0
R4  [a0] = 2
R5  [a1] = 7ffff6d4
R6  [a2] = 7ffff6e0
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
```

```
                                                        User Text Segment [00400000]..
[00400000] 8fa40000  lw $4, 0($29)            ; 183: lw $a0 0($sp) # argc
[00400004] 27a50004  addiu $5, $29, 4         ; 184: addiu $a1 $sp 4 # argv
[00400008] 24a60004  addiu $6, $5, 4          ; 185: addiu $a2 $a1 4 # envp
[0040000c] 00041080  sll $2, $4, 2            ; 186: sll $v0 $a0 2
[00400010] 00c23021  addu $6, $6, $2          ; 187: addu $a2 $a2 $v0
[00400014] 0c100009  jal 0x00400024 [main]    ; 188: jal main
[00400018] 00000000  nop                      ; 189: nop
[0040001c] 3402000a  ori $2, $0, 10           ; 191: li $v0 10
[00400020] 0000000c  syscall                  ; 192: syscall # syscall 10 (exit)
[00400024] 20090061  addi $9, $0, 97          ; 3: addi $t1, $zero, 97
[00400028] 3402000a  ori $2, $0, 10           ; 4: li $v0, 10
[0040002c] 0000000c  syscall                  ; 5: syscall

                                                      Kernel Text Segment [80000000].
[80000180] 0001d821  addu $27, $0, $1         ; 90: move $k1 $at # Save $at
[80000184] 3c019000  lui $1, -28672           ; 92: sw $v0 s1 # Not re-entrant and we ca
[80000188] ac220200  sw $2, 512($1)
[8000018c] 3c019000  lui $1, -28672           ; 93: sw $a0 s2 # But we need to use these
[80000190] ac240204  sw $4, 516($1)
[80000194] 401a6800  mfc0 $26, $13            ; 95: mfc0 $k0 $13 # Cause register
[80000198] 001a2082  srl $4, $26, 2           ; 96: srl $a0 $k0 2 # Extract ExcCode Fiel
[8000019c] 3084001f  andi $4, $4, 31          ; 97: andi $a0 $a0 0x1f
[800001a0] 34020004  ori $2, $0, 4            ; 101: li $v0 4 # syscall 4 (print_str)
[800001a4] 3c049000  lui $4, -28672 [__m1_]   ; 102: la $a0 __m1_
[800001a8] 0000000c  syscall                  ; 103: syscall
```

```
Memory and registers cleared
```

Here you will see the contents of $0 to $31.

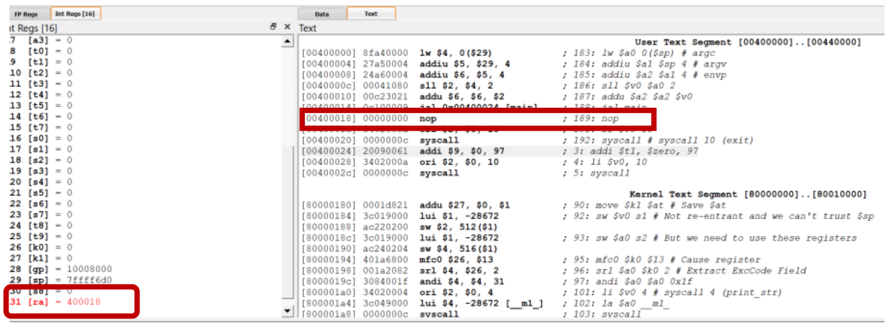
The button bar at the top gives you some options:



The buttons we are most interested in are:

Button	Name	Function
☰	Single Step	Executes the instruction at the cursor, updating the registers and machine states
▶	Run	Runs the program
⏸	Pause	Pauses the program
■	Stop	Stops the program
↻	Clear Registers	Sets all registers to 0

Click the single-step buttons until the cursor reaches the "jal" instruction, observing how the registers change.
Before stepping through the jal instruction, scroll the Int Registers window down until you can see register $31 ($ra). Press the single step button again, and you will notice that register $31 now contains 0x400018:



```
7  [a3] = 0
8  [t0] = 0
9  [t1] = 0
10 [t2] = 0
11 [t3] = 0
12 [t4] = 0
13 [t5] = 0
14 [t6] = 0
15 [t7] = 0
16 [s0] = 0
17 [s1] = 0
18 [s2] = 0
19 [s3] = 0
20 [s4] = 0
21 [s5] = 0
22 [s6] = 0
23 [s7] = 0
24 [t8] = 0
25 [t9] = 0
26 [k0] = 0
27 [k1] = 0
28 [gp] = 10008000
29 [sp] = 7ffff6d0
30 [s8] = 0
31 [ra] = 400018
```

```
                                                       User Text Segment [00400000]..[00440000]
[00400000] 8fa40000  lw $4, 0($29)            ; 183: lw $a0 0($sp) # argc
[00400004] 27a50004  addiu $5, $29, 4         ; 184: addiu $a1 $sp 4 # argv
[00400008] 24a60004  addiu $6, $5, 4          ; 185: addiu $a2 $a1 4 # envp
[0040000c] 00041080  sll $2, $4, 2            ; 186: sll $v0 $a0 2
[00400010] 00c23021  addu $6, $6, $2          ; 187: addu $a2 $a2 $v0
[00400014] 0c100009  jal 0x00400024 [main]    ; 188: jal main
[00400018] 00000000  nop                      ; 189: nop
[illegible]
[00400020] 0000000c  syscall                  ; 192: syscall # syscall 10 (exit)
[00400024] 20090061  addi $9, $0, 97          ; 3: addi $t1, $zero, 97
[00400028] 3402000a  ori $2, $0, 10           ; 4: li $v0, 10
[0040002c] 0000000c  syscall                  ; 5: syscall

                                                       Kernel Text Segment [80000000]..[80010000]
[80000180] 0001d821  addu $27, $0, $1         ; 90: move $k1 $at # Save $at
[80000184] 3c019000  lui $1, -28672           ; 92: sw $v0 s1 # Not re-entrant and we can't trust $sp
[80000188] ac220200  sw $2, 512($1)
[8000018c] 3c019000  lui $1, -28672           ; 93: sw $a0 s2 # But we need to use these registers
[80000190] ac240204  sw $4, 516($1)
[80000194] 401a6800  mfc0 $26, $13            ; 95: mfc0 $k0 $13 # Cause register
[80000198] 001a2082  srl $4, $26, 2           ; 96: srl $a0 $k0 2 # Extract ExcCode Field
[8000019c] 3084001f  andi $4, $4, 31          ; 97: andi $a0 $a0 0x1f
[800001a0] 34020004  ori $2, $0, 4            ; 101: li $v0 4 # syscall 4 (print_str)
[800001a4] 3c049000  lui $4, -28672 [__m1_]   ; 102: la $a0 __m1_
[800001a8] 0000000c  syscall                  ; 103: syscall
```

This is the instruction immediately after the jal instruction. Note now the cursor is at address 0x400024, which is the first instruction of our program.
We can scroll up the Int Registers window till we see $9 and $2, then single step through our program to see how the registers change (note that the Int Registers window shows values in hexadecimal by default. You can right-click to change to Decimal or Binary). Once we execute the syscall, the program ends. If we single-step again, QTSpim will start executing the program again.

This program isn't very exciting as it just adds 97 to register $9 and terminates. Let's now load sample2.asm, which is more fun:

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

```
        syscall
        li $v0, 10
   syscall
```

Let’s examine this code. The lines:

```
        .data 0x10000100
msg:    .asciiz "Hello"
```

Firstly declare a “data segment” starting at address 0x10000100. As explained earlier the “data segment” holds all global variables.

The line `msg:  .asciiz  “Hello”` creates a new global variable called “msg”, which is an “asciiz” or “null-terminated string” containing the word “Hello”. A “null-terminated string” is a sequence of ASCII characters terminated with the NULL character, which is ASCII 0. This is C’s standard way of representing strings (which is completely different from a String in Python, which is a class).

The next few lines creates our program.

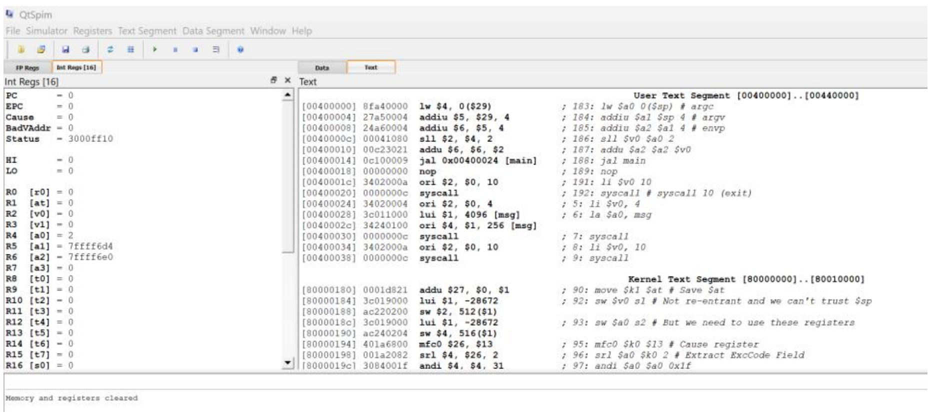
```
       .text
main:  li $v0, 4
       la $a0, msg
       syscall
       li $v0, 10
   syscall
```

As before, “.text” declares a “text” segment that holds our program code. Our code loads 4 into $v0, which, as per Table 1 above, lets us print the contents of a string. The code then does “la $a0, msg”, which is a pseudo-instruction to load the address of our variable msg into register $a0, as required by the OS call. We then perform a syscall to actually print the string.

(Yes, the print statement in Python works in a similar way to this)

The next two lines load 10 into $v0 and does a syscall, which terminates our program.

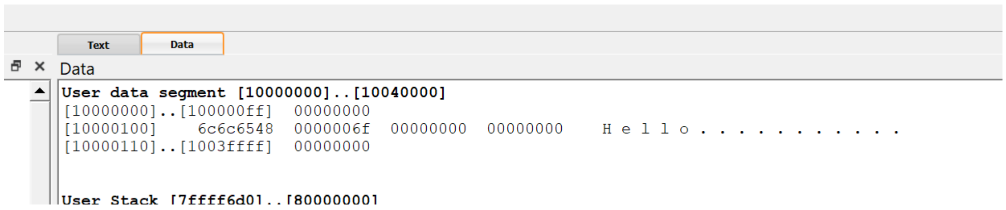
As before, click File->Reinitialize and Load File, and load in sample2.asm. Your screen will look like this:


QtSpim
File Simulator Registers Text Segment Data Segment Window Help
FP Regs
Int Regs [16]
Data
Text

```
Int Regs [16]
PC       = 0
EPC      = 0
Cause    = 0
BadVAddr = 0
Status   = 3000ff10

HI       = 0
LO       = 0

R0  [r0] = 0
R1  [at] = 0
R2  [v0] = 0
R3  [v1] = 0
R4  [a0] = 2
R5  [a1] = 7ffff6d4
R6  [a2] = 7ffff6e0
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
```

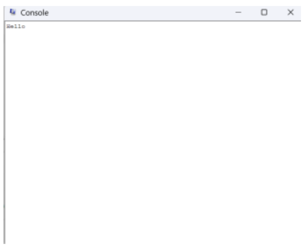
```
Text
                                                          User Text Segment [00400000]..[00440000]
[00400000] 8fa40000  lw $4, 0($29)             ; 183: lw $a0 0($sp) # argc
[00400004] 27a50004  addiu $5, $29, 4          ; 184: addiu $a1 $sp 4 # argv
[00400008] 24a60004  addiu $6, $5, 4           ; 185: addiu $a2 $a1 4 # envp
[0040000c] 00041080  sll $2, $4, 2             ; 186: sll $v0 $a0 2
[00400010] 00c23021  addu $6, $6, $2           ; 187: addu $a2 $a2 $v0
[00400014] 0c100009  jal 0x00400024 [main]     ; 188: jal main
[00400018] 00000000  nop                       ; 189: nop
[0040001c] 3402000a  ori $2, $0, 10            ; 191: li $v0 10
[00400020] 0000000c  syscall                   ; 192: syscall # syscall 10 (exit)
[00400024] 34020004  ori $2, $0, 4             ; 5: li $v0, 4
[00400028] 3c011000  lui $1, 4096 [msg]        ; 6: la $a0, msg
[0040002c] 34240100  ori $4, $1, 256 [msg]
[00400030] 0000000c  syscall                   ; 7: syscall
[00400034] 3402000a  ori $2, $0, 10            ; 8: li $v0, 10
[00400038] 0000000c  syscall                   ; 9: syscall

                                                          Kernel Text Segment [80000000]..[80010000]
[80000180] 0001d821  addu $27, $0, $1          ; 90: move $k1 $at # Save $at
[80000184] 3c019000  lui $1, -28672            ; 92: sw $v0 s1 # Not re-entrant and we can't trust $sp
[80000188] ac220200  sw $2, 512($1)
[8000018c] 3c019000  lui $1, -28672            ; 93: sw $a0 s2 # But we need to use these registers
[80000190] ac240204  sw $4, 516($1)
[80000194] 401a6800  mfc0 $26, $13             ; 95: mfc0 $k0 $13 # Cause register
[80000198] 001a2082  srl $4, $26, 2            ; 96: srl $a0 $k0 2 # Extract ExcCode Field
[8000019c] 3084001f  andi $4, $4, 31           ; 97: andi $a0 $a0 0x1f

Memory and registers cleared
```

As before the program starts at 0x400000 with code to load arguments from the command line. This code was not written by us but inserted by the QTSpim assembler. Our actual code starts at 0x400024. Additionally, click on the “Data” tab to see the data segment:



```
Data
User data segment [10000000]..[10040000]
[10000000]..[100000ff]  00000000
[10000100]     6c6c6548  0000006f  00000000  00000000    H e l l o . . . . . . . . . . .
[10000110]..[1003ffff]  00000000


User Stack [7ffff6d0]..[80000000]
```

You can see our “Hello” message at address 0x10000100. Click on the Text tab again, and step through the program, observing how the registers change. You will also see a Console pop-up showing the word Hello:



```
Hello
```

Question 1. (5 MARKS)
In the data tab we see that msg is at address 0x10000100. The assembler converts the pseudo instruction la $a0, msg to two instructions. State what these instructions are and explain how they load the address of msg into $a0.

4. <u>Assignment Questions</u>

Now that you are familiar with QTSpim, let’s attempt some programming. You will be given snippets of code in Python, and you will need to convert the code to MIPS Assembly.

You need to use a PLAIN TEXT editor like Notepad to write your assembly programs, then load them into QTSpim to test them. You cannot use any kind of richtext, HTML or Markdown editor to do this as they will insert undesirable characters and markups into your program.

a. Functions

We will now look at how to create functions in MIPS.

i. Create a file called ex1.asm, using Notepad or some other suitable PLAIN TEXT editor (Microsoft Word and Wordpad are not suitable as they insert markups to the file.)

ii. Create the data segment to store our Hello World string:

```
        .data 0x10000100
msg:    .asciiz "Hello World"
```

iii. Create text (code) segment, and create the helloworld function. Notice how we create it by using a label. The function loads 4 into $v0, which is the code to print a string, then loads the address of our message into $a0 and performs the syscall:

```
helloworld:   li $v0, 4
              la $a0, msg
              syscall
              jr $ra
```

In the last line we have a jump register to $ra, which will contain the return address to the instruction after the function call.

We now create our “main”, which does a jump-and-link (jal), the MIPS way of doing a function call:

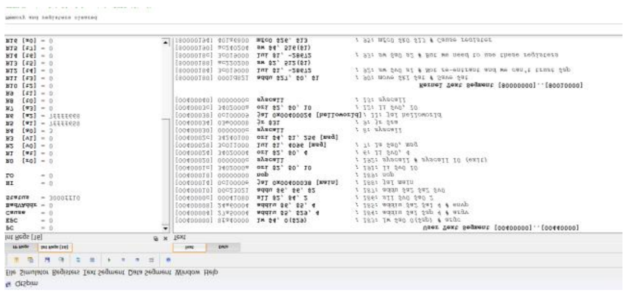
```
main:   jal helloworld
        li $v0, 10
        syscall
```

Main then loads 10 into $v0 and does a syscall to terminate the program.

Your full ex1.asm will look like this:

```
                .data 0x10000100
msg:            .asciiz "Hello World"

                .text
helloworld:     li $v0, 4
                la $a0, msg
                syscall
                jr $ra

main:           jal helloworld
                li $v0, 10
                     syscall
```

Now we click File->Reinitialize and Load File, and load ex.asm. Your screen should look like this:



Question 2. (3 MARKS)

In our code the helloworld function is defined before main. However main is executed first, and without the jal, helloworld will not be executed. Explain why main gets executed first.

Step through the program, carefully observing $ra as the program runs. Then answer the following question.

Question 3. (3 MARKS)

Explain what the jr isntruction does.

Using what you know about fetch stage in the MIPS datapath, explain how functions are implemented with jal and jr work together to implement function calls.

b. Limitations of Function Calls

We will now look at the limitations of function calls in MIPS. You are provided with a program called jal.asm. We begin by creating the data segment, and three strings:

```
    .data 0x10000100
str1:    .asciiz    "This is function 1"
str2:    .asciiz    "This is function 2"
str3:    .asciiz    "This is main"
```

We now declare two functions fun1 and fun2. The function fun1 prints "This is function 1", then calls fun2 which prints "This is function 2".

```
         .text

#fun1: Prints "This is function 1", then calls fun2
fun1:    li $v0, 4
         la $a0, str1
         syscall
         jal fun2
         jr $ra

#fun2: Prints "This is function 2"
fun2:    li $v0, 4
         la $a0, str2
         syscall
         jr $ra
```

Finally we create main, which calls fun1, prints "This is main" then exits:

```
#main: Calls fun1 then prints "This is main"
main:      jal fun1
           li $v0, 4
           la $a0, str3
           syscall
           li $v0, 10
           syscall
```

Click File->Reinitialize and Load, and load in jal.asm. Click run.

Question 4. (2 MARKS)

What output is produced on the Console? (Click Window->Console if you cannot see it.) Do you see "This is main"?

Click File->Reinitialize and Load File again, and reload jal.asm. This time use the single-step button to understand why main never printed “This is main”.

Hint: Examine the contents of $ra after fun2 executes jr $ra and control returns to fun1.

Question 5. (5 MARKS)

Explain, with reference to $ra, why jal.asm does not work as expected.

c. Supporting Nested Function Calls

When one function calls another function, this is called a “nested function call”. We see nexted function calls all the fime when we are programming, yet our MIPS architecture cannot support it.

Or can it? In this part we see how nested function calls are done in assembly.

The Stack Segment

We have already seen that when a program runs in memory, there is a data segment that stores global variables, and a text segment that stores program code. We now introduce a third segment called the “stack segment”, whose job is to facilitate function calls.

The special register $sp (stack pointer) always points to the next location in the stack to be written to, called the “top of stack”.

Every function call is given a “stack frame” within the stack segment. The function uses the stack frame for:

- Passsing parameters to functions and getting back the results. For our purpose we will continue to use the $a and $v registers instead.
- Creating local variables to use within functions. We will not consider this here .
- Saving registers that a function needs to use. We will not consider this here.
- Saving the $ra register.

Whilst $sp points to the “top of stack”, $fp will point to the start of the stack frame.

Question 6. (4 MARKS)

Before a function can use any registers, it is generally required to save those registers first by writing them to the stack, then restore them by reading them from the stack before executing jr $ra to return to the caller. Explain why this is needed. (Hint: Does the function being called (the “callee”) know what registers the caller is using?)

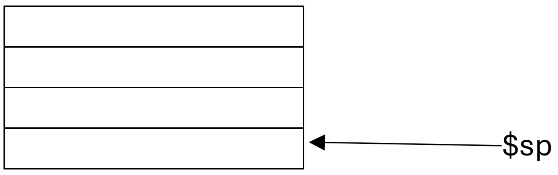
We will now see how to use the Stack Frame to support nested function calls:

i. Saving $fp to the stack.

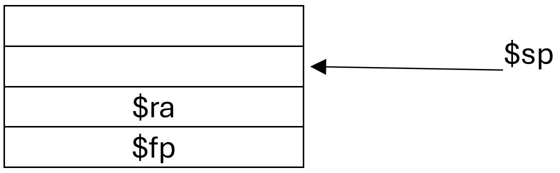
The area of a stack that a function uses to store its information is called the "stack frame", and the frame pointer $fp is a register that points to the start of the stack frame.

Our stack frame is very simple and will only be used to save $fp and $ra. The diagram below illustrates this idea:

Initial state of stack



State of stack just before calling function



We copy $sp to $fp so $fp will points to the start of the stack frame. However before that we must save $fp to the stack. This is how we do it:

```
sw    $fp, 0($sp)      # Save $fp
addi  $fp, $sp, 0      # Copy $sp to $fp
sw    $ra, 4($fp)      # Save $ra
# Increment $sp to point to new top of stack
addi $sp, $sp, 8
```

Writing to the top of the stack and incrementing $sp is called "pushing to the stack"

Question 7. (3 MARKS)

Why do we save $ra at 4($fp) and not at 0($fp) or 0($sp)?

ii. Making our function call.

iii. Restore $ra and $fp:

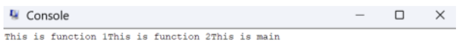
```
lw $ra, 4($fp)         # Restore $ra
addi $sp, $fp, 0       # Copy $fp to $sp
lw $fp, 0($fp)         # Restore fp
```

Question 8. (2 MARKS)

We restore $sp by copying $fp to $sp using addi $sp, $fp, 0. Rewrite this in one other way using MIPS assembly. You are not permitted to use mov.

iv. Return to caller.

Copy jal.asm to a file called jal_fixed.asm, then fix the code so that the nested call now works properly. When you run the code, the console should look like this:



```
This is function 1This is function 2This is main
```

Question 9. (3 MARKS)

Cut and paste the parts of jal_fixed.asm that you fixed, and explain your fixes.

Now that we understand how to build nested functions, let's create a program that does the equivalent of the following in C:

```
X = [20, 110, 17, 5, 3, 12, 18, 8, 99, 25];
count = 10;
for(i=0; i<count; i++) {
      printf("%d", i);
}
```

Follow the steps below (TOTAL MARKS: 10)

i. Create a program called print.asm.

ii. (1 mark) Create a data segment at address 0x10000110, and create the X array and count variable. Note: To create an array of 32-bit values in the data segment, use:

```
label: .word a b c d
```

E.g. to create an array called myarray with element 1, 3, 5:

```
myarray: .word 1 3 5
```

This can also be used to create a variable with a single 32-bit value.

iii. (2 marks) Create a function called printint which prints an integer in $a0 followed by a space character. Cut and paste your code into the answer book.

iv. (5 marks) Create a function called printarray which calls printint repeatedly to print all the elements in X. Cut and paste your code into the answer book.

v. (2 marks) Write a main to call printarray, and then properly exit to the OS. Cut and paste your code to the answer book.

Do not include your completed print.asm in your submission.

5. Conclusion

This lab has given you some experience in writing assembly language programs, and in particular, how program memory is structured into data (global variables), text (program code) and stack (function call support) segments. There is also a heap segment but this was not covered in this assignment. Additionally we also saw how programs can call the Operating System for various services.