

QtSpim Tutorial

Contents

1. [Installation](#)
2. [Usage](#)
3. [Overview](#)
4. [First Program](#)

Installation

We will be using [SPIM](#), a MIPS simulator, in order to learn assembly programming. The current version of SPIM is "QtSPIM", aka SPIM using the [Qt](#) cross-platform application GUI framework:

Install the [Qt](#) framework that SPIM requires:

```
unix> sudo apt-get install qt4-dev-tools qt4-doc libqt4-help
```

Download a pre-packaged version of SPIM for Linux that uses the Qt GUI framework: *(If you are running an older 32-bit Linux installation, change 64 to 32 below)*

```
unix> wget
http://sourceforge.net/projects/spimsimulator/files/qtspim_9.1.9_linux64.deb
```

Install the QtSPIM package:

```
unix> sudo dpkg -i qtspim_9.1.9_linux64.deb
```

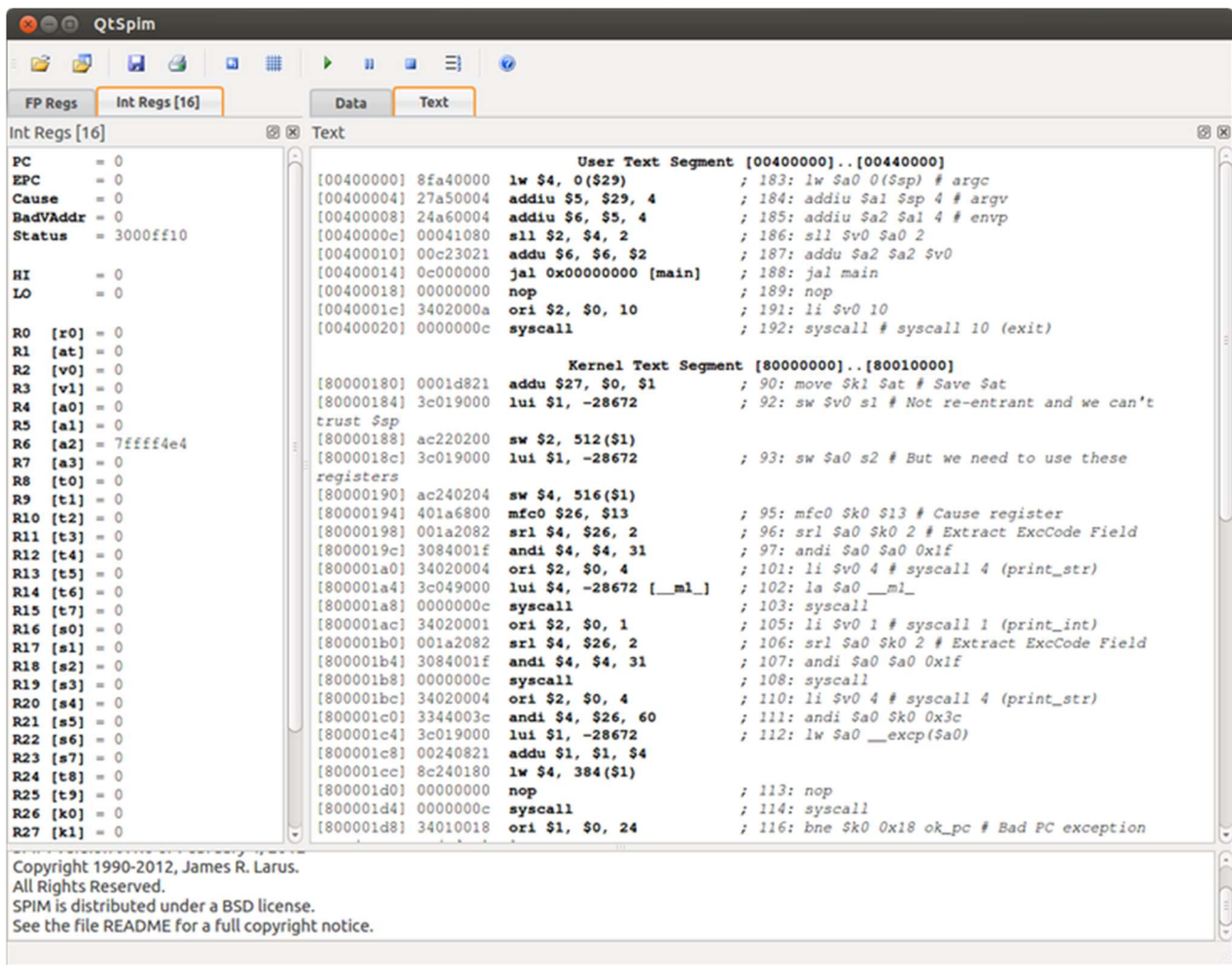
Usage

To launch QtSPIM:

```
unix> qtspim &
```

Overview

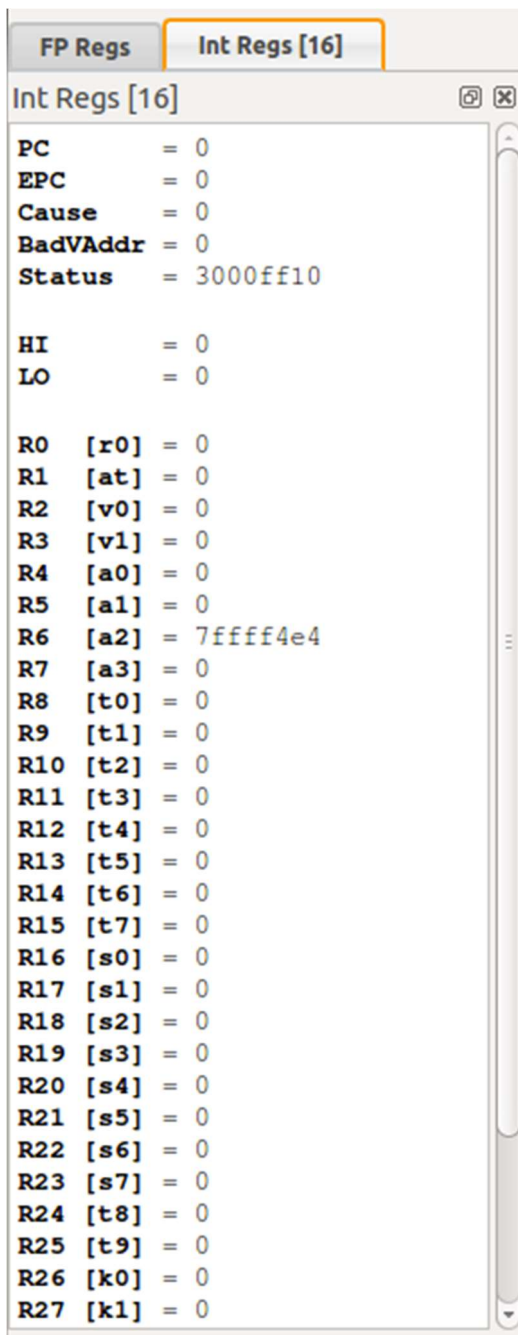
After launching QtSPIM, the main window should appear as shown below.



There are three primary sections contained within this window: The *Register* panel, *Memory* panel, and *Messages* panel.

Register Panel

The Register panel (shown below) shows the contents of all the MIPS registers. There are two tabs in this panel: one for the *floating point* registers and one for the *integer* registers. The integer registers include general purpose registers (R1-R31), along with special purpose registers such as the Program Counter (PC).

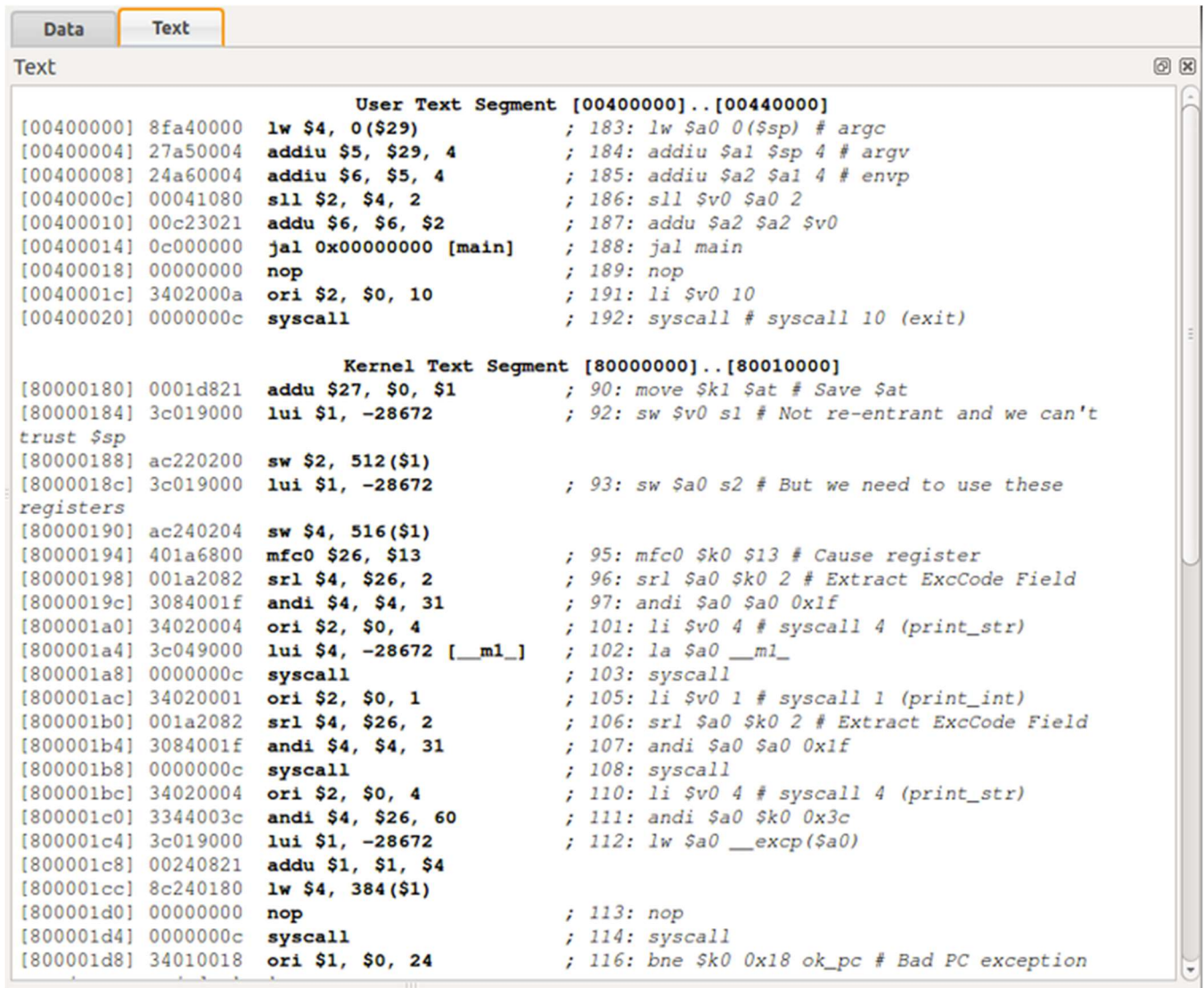


Memory Panel

The Memory panel has two tabs: Data and Text. The Text tab shows the contents of the *Program* memory space. From left to right, this includes:

1. The memory address of an instruction in hexadecimal (shown in brackets)
2. The contents of that memory address in hexadecimal. **In binary form, this is the actual MIPS instruction that the processor runs!**
3. The "human-readable" assembly language instruction using the hardware register numbers (shown in **bold**).

4. The assembly language program you wrote using symbolic register names and memory address symbols (shown in *italics*)



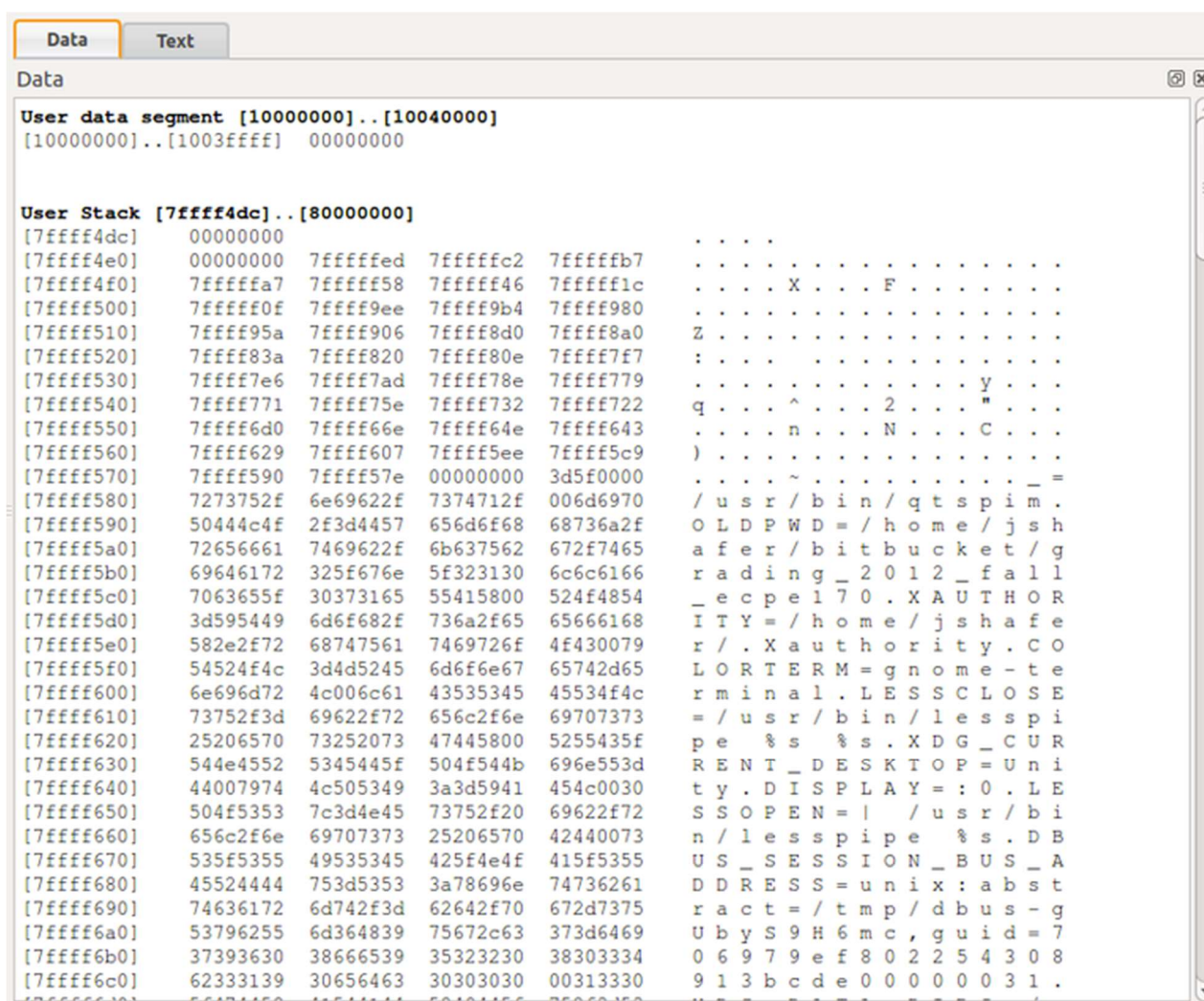
The screenshot shows a debugger window with the 'Text' tab selected. It displays two segments of assembly code: 'User Text Segment' and 'Kernel Text Segment'. Each line of code consists of a memory address in brackets, a hexadecimal value, and the assembly instruction. Comments are provided for many instructions, often starting with a semicolon and a line number. The 'User Text Segment' starts at address [00400000] and ends at [00440000]. The 'Kernel Text Segment' starts at [80000000] and ends at [80010000].

```

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 $a0 # Save $a0
[80000184] 3c019000 lui $1, -28672 ; 92: sw $v0 $1 # Not re-entrant and we can't
trust $sp
[80000188] ac220200 sw $2, 512($1)
[8000018c] 3c019000 lui $1, -28672 ; 93: sw $a0 $2 # 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
[800001b8] 0000000c syscall ; 108: syscall
[800001bc] 34020004 ori $2, $0, 4 ; 110: li $v0 4 # syscall 4 (print_str)
[800001c0] 3344003c andi $4, $26, 60 ; 111: andi $a0 $k0 0x3c
[800001c4] 3c019000 lui $1, -28672 ; 112: lw $a0 __excp($a0)
[800001c8] 00240821 addu $1, $1, $4
[800001cc] 8c240180 lw $4, 384($1)
[800001d0] 00000000 nop ; 113: nop
[800001d4] 0000000c syscall ; 114: syscall
[800001d8] 34010018 ori $1, $0, 24 ; 116: bne $k0 0x18 ok_pc # Bad PC exception
```

The Data tab shows the contents of the Data memory space. This includes the variables and array data you create, along with the stack content.



Messages Panel

The Messages panel displays messages from QtSPIM to the user.

Memory and registers cleared
Loaded: /tmp/qt_temp.MT3159
SPIM Version 9.1.6 of February 4, 2012
Copyright 1990-2012, James R. Larus.
All Rights Reserved.

First Program

A variety of [MIPS example programs](#) are available to you. Start with the first program - [example1.asm](#) - by downloading it to your computer.

Load your first program by selecting **File -> Reinitialize and Load File**. (Recall that the new Ubuntu "Unity" GUI doesn't show the top-of-screen menubar until you mouse over it). This clears the register space and resets the simulator. (Otherwise, you could load and run several programs in a row on the same machine state.)

You can scroll down in the Text pane to see that the assembly code has been loaded into Program memory space. In this case, the first instruction is at memory location 0x00400024. (Why doesn't it start at memory address zero? The program starts with the function main(), but there is some code that runs before main).

Now that the program has been loaded, you can run a simulation of the assembly instructions. You have three choices:

1. Run the program from beginning to end (via the "play" Run/Continue button or F5). This is useful for seeing the final output of the program.
2. Step through the program one line at a time (via the "123" Single Step button or F10). This is useful to see how each assembly instruction affects the machine state (i.e. memory and register values).
3. Run the program until you reach a breakpoint (which can be set by right-clicking on any line in the Memory panel).

Step through the complete program, figure out what it does, and make sure you understand how the QtSPIM environment works.

example1.asm

 example1.asm — Assembly Source Code, 1 KB

File contents

```
# A demonstration of some simple MIPS instructions
# used to test QtSPIM

    # Declare main as a global function
    .globl main

    # All program code is placed after the
    # .text assembler directive
    .text

# The label 'main' represents the starting point
main:
    li $t2, 25           # Load immediate value (25)
    lw $t3, value        # Load the word stored at label 'value'
    add $t4, $t2, $t3     # Add
    sub $t5, $t2, $t3     # Subtract

    # Exit the program by means of a syscall.
    # There are many syscalls - pick the desired one
    # by placing its code in $v0. The code for exit is "10"
```

```

li $v0, 10 # Sets $v0 to "10" to select exit syscall
syscall # Exit

# All memory structures are placed after the
# .data assembler directive
.data

# The .word assembler directive reserves space
# in memory for a single 4-byte word (or multiple 4-byte words)
# and assigns that memory location an initial value
# (or a comma separated list of initial values)
value: .word 12

```

example2_hello_world.asm

 example2_hello_world.asm — Assembly Source Code, 1 KB

File contents

```

# "Hello World" in MIPS assembly
# From: http://labs.cs.upt.ro/labs/so2/html/resources/nachos-doc/mipsf.html

# All program code is placed after the
# .text assembler directive
.text

# Declare main as a global function
.globl main

# The label 'main' represents the starting point
main:
    # Run the print_string syscall which has code 4
    li      $v0, 4          # Code for syscall: print_string
    la      $a0, msg        # Pointer to string (load the address of msg)
    syscall

    li      $v0, 10         # Code for syscall: exit
    syscall

# All memory structures are placed after the
# .data assembler directive
.data

# The .asciiz assembler directive creates
# an ASCII string in memory terminated by
# the null character. Note that strings are
# surrounded by double-quotes
msg: .asciiz "Hello World!\n"

```

example3_io.asm

 example3_io.asm — Assembly Source Code, 1 KB (1146 bytes)

File contents

```
# Simple input/output in MIIPS assembly
# From: http://labs.cs.upt.ro/labs/so2/html/resources/nachos-doc/mipsf.html
```

```
    # Start .text segment (program code)
    .text

    .globl main
main:
    # Print string msg1
    li      $v0,4          # print_string syscall code = 4
    la      $a0, msg1      # load the address of msg
    syscall

    # Get input A from user and save
    li      $v0,5          # read_int syscall code = 5
    syscall
    move    $t0,$v0        # syscall results returned in $v0

    # Print string msg2
    li      $v0,4          # print_string syscall code = 4
    la      $a0, msg2      # load the address of msg2
    syscall

    # Get input B from user and save
    li      $v0,5          # read_int syscall code = 5
    syscall
    move    $t1,$v0        # syscall results returned in $v0

    # Math!
    add     $t0, $t0, $t1   # A = A + B

    # Print string msg3
    li      $v0, 4
    la      $a0, msg3
    syscall

    # Print sum
    li      $v0,1          # print_int syscall code = 1
    move    $a0, $t0        # int to print must be loaded into $a0
    syscall

    # Print \n
    li      $v0,4          # print_string syscall code = 4
    la      $a0, newline
    syscall

    li      $v0,10         # exit
    syscall

    # Start .data segment (data!)
    .data
msg1:    .asciiz "Enter A:  "
```




```

msg2:    .ascii "Enter B:  "
msg3:    .ascii "A + B = "
newline: .ascii "\n"

```

example4_loop.asm

 example4_loop.asm — Assembly Source Code, 1 KB (1044 bytes)

File contents

```

# Simple routine to demo a loop
# Compute the sum of N integers: 1 + 2 + 3 + ... + N
# From: http://labs.cs.upt.ro/labs/so2/html/resources/nachos-doc/mipsf.html

```

```

        .text

        .globl main
main:
    # Print msg1
    li    $v0,4          # print_string syscall code = 4
    la    $a0, msg1
    syscall

    # Get N from user and save
    li    $v0,5          # read_int syscall code = 5
    syscall
    move   $t0,$v0       # syscall results returned in $v0

    # Initialize registers
    li    $t1, 0         # initialize counter (i)
    li    $t2, 0         # initialize sum

    # Main loop body
loop:   addi   $t1, $t1, 1    # i = i + 1
        add    $t2, $t2, $t1 # sum = sum + i
        beq    $t0, $t1, exit # if i = N, continue
        j      loop

    # Exit routine - print msg2
exit:   li    $v0, 4       # print_string syscall code = 4
        la    $a0, msg2
        syscall

    # Print sum
    li    $v0,1          # print_string syscall code = 4
    move   $a0, $t2
    syscall

    # Print newline
    li    $v0,4          # print_string syscall code = 4
    la    $a0, lf
    syscall
    li    $v0,10         # exit
    syscall

    # Start .data segment (data!)
    .data


```

```

msg1:  .asciiz "Number of integers (N)?  "
msg2:  .asciiz "Sum = "
lf:    .asciiz "\n"

```

example5_function_without_stack.asm

 example5_function_without_stack.asm — Assembly Source Code, 1 KB (1504 bytes)

File contents

```

# Simple routine to demo functions
# NOT using a stack in this example.
# Thus, the function does not preserve values
# of calling function!

# -----

        .text

        .globl main
main:
    # Register assignments
    # $s0 = x
    # $s1 = y

    # Initialize registers
    lw     $s0, x           # Reg $s0 = x
    lw     $s1, y           # Reg $s1 = y

    # Call function
    move   $a0, $s0         # Argument 1: x ($s0)
    jal    fun              # Save current PC in $ra, and jump to fun
    move   $s1, $v0         # Return value saved in $v0. This is y ($s1)

    # Print msg1
    li     $v0, 4           # print_string syscall code = 4
    la     $a0, msg1
    syscall

    # Print result (y)
    li     $v0, 1           # print_int syscall code = 1
    move   $a0, $s1         # Load integer to print in $a0
    syscall

    # Print newline
    li     $v0, 4           # print_string syscall code = 4
    la     $a0, lf
    syscall

    # Exit
    li     $v0, 10          # exit
    syscall

# -----

# FUNCTION: int fun(int a)
# Arguments are stored in $a0

```

```

        # Return value is stored in $v0
        # Return address is stored in $ra (put there by jal instruction)
        # Typical function operation is:

fun:    # Do the function math
        li $s0, 3
        mul $s1,$s0,$a0      # s1 = 3*$a0   (i.e. 3*a)
        addi $s1,$s1,5       # 3*a+5

        # Save the return value in $v0
        move $v0,$s1


        # Return from function
        jr $ra               # Jump to addr stored in $ra

# -----

        # Start .data segment (data!)
        .data
x:      .word 5
y:      .word 0
msg1:   .asciiz "y="
lf:     .asciiz "\n"

```

example5_function_with_stack.asm

 example5_function_with_stack.asm — Assembly Source Code, 1 KB (1958 bytes)

File contents

```

# Simple routine to demo functions
# USING a stack in this example to preserve
# values of calling function

# -----

        .text

        .globl main
main:
        # Register assignments
        # $s0 = x
        # $s1 = y

        # Initialize registers
        lw     $s0, x         # Reg $s0 = x
        lw     $s1, y         # Reg $s1 = y

        # Call function
        move   $a0, $s0       # Argument 1: x ($s0)
        jal    fun            # Save current PC in $ra, and jump to fun
        move   $s1,$v0        # Return value saved in $v0. This is y ($s1)

        # Print msg1
        li     $v0, 4         # print_string syscall code = 4

```

```
la      $a0, msg1
syscall
```

```
# Print result (y)
li      $v0,1          # print_int syscall code = 1
move    $a0, $s1       # Load integer to print in $a0
syscall
```

```
# Print newline
li      $v0,4          # print_string syscall code = 4
la      $a0, lf
syscall
```

```
# Exit
li      $v0,10         # exit
syscall
```

```
# -----
```

```
# FUNCTION: int fun(int a)
# Arguments are stored in $a0
# Return value is stored in $v0
# Return address is stored in $ra (put there by jal instruction)
# Typical function operation is:
```

```
fun:    # This function overwrites $s0 and $s1
        # We should save those on the stack
        # This is PUSH'ing onto the stack
        addi $sp,$sp,-4      # Adjust stack pointer
        sw   $s0,0($sp)     # Save $s0
        addi $sp,$sp,-4      # Adjust stack pointer
        sw   $s1,0($sp)     # Save $s1
```

```
# Do the function math
li      $s0, 3
mul     $s1,$s0,$a0         # s1 = 3*$a0 (i.e. 3*a)
addi    $s1,$s1,5           # 3*a+5
```

```
# Save the return value in $v0
move    $v0,$s1
```

```
# Restore saved register values from stack in opposite order
# This is POP'ing from the stack
lw      $s1,0($sp)         # Restore $s1
addi    $sp,$sp,4          # Adjust stack pointer
lw      $s0,0($sp)         # Restore $s0
addi    $sp,$sp,4          # Adjust stack pointer
```

```
# Return from function
jr      $ra                # Jump to addr stored in $ra
```

```
# -----
```

```
# Start .data segment (data!)
```

```
.data
```

```
x:      .word 5
```

```
y:      .word 0
```

```
msg1:   .asciiz "y="
```

```
lf:     .asciiz "\n"
```

example6_loop_with_function.asm

 example6_loop_with_function.asm — Assembly Source Code, 2 KB (2498 bytes)

File contents

```
# Simple routine to demo a loop
# Compute the sum of N integers: 1 + 2 + 3 + ... + N
# Same result as example4, but here a function performs the
# addition operation:  int add(int num1, int num2)

# -----

        .text

        .globl main
main:
    # Register assignments
    # $s0 = N
    # $s1 = counter (i)
    # $s2 = sum

    # Print msg1
    li      $v0,4          # print_string syscall code = 4
    la      $a0, msg1
    syscall

    # Get N from user and save
    li      $v0,5          # read_int syscall code = 5
    syscall
    move    $s0,$v0        # syscall results returned in $v0

    # Initialize registers
    li      $s1, 0         # Reg $s1 = counter (i)
    li      $s2, 0         # Reg $s2 = sum

    # Main loop body
loop:   addi    $s1, $s1, 1   # i = i + 1

    # Call add function
    move    $a0, $s2        # Argument 1: sum ($s2)
    move    $a1, $s1        # Argument 2: i ($s1)
    jal     add2            # Save current PC in $ra, and jump to add2
    move    $s2,$v0         # Return value saved in $v0. This is sum ($s2)
    beq     $s0, $s1, exit  # if i = N, continue
    j       loop

    # Exit routine - print msg2
exit:   li      $v0, 4       # print_string syscall code = 4
    la      $a0, msg2
    syscall

    # Print sum
    li      $v0,1           # print_string syscall code = 4
    move    $a0, $s2
    syscall

    # Print newline
```

```

li      $v0,4          # print_string syscall code = 4
la      $a0, lf
syscall
li      $v0,10         # exit
syscall

# -----

# FUNCTION: int add(int num1, int num2)
# Arguments are stored in $a0 and $a1
# Return value is stored in $v0
# Return address is stored in $ra (put there by jal instruction)
# Typical function operation is:
# 1.) Store registers on the stack that we will overwrite
# 2.) Run the function
# 3.) Save the return value
# 4.) Restore registers from the stack
# 5.) Return (jump) to previous location
# Note: This function is longer than it needs to be,
# in order to demonstrate the usual 5 step function process...

add2:   # Store registers on the stack that we will overwrite (just $s0)
        addi $sp,$sp, -4      # Adjust stack pointer
        sw $s0,0($sp)        # Save $s0 on the stack

        # Run the function
        add $s0,$a0,$a1      # Sum = sum + i

        # Save the return value in $v0
        move $v0,$s0

        # Restore overwritten registers from the stack
        lw $s0,0($sp)
        addi $sp,$sp,4       # Adjust stack pointer

        # Return from function
        jr $ra               # Jump to addr stored in $ra

# -----

# Start .data segment (data!)
.data
msg1:   .asciiz "Number of integers (N)?  "
msg2:   .asciiz "Sum = "
lf:     .asciiz "\n"

```


Lab: Introduction to PCSpim

Important Information

1. In the lab room pls installed PCSpim version 9.1.9. in your PC
2. To open the PCSpim, you can double click on the icon on your desk top, or open `C:\Program Files\PCSpim\pcspim.exe`
3. Once you get it opened, click on the "Simulator" and then select "Settings" . When you see the table, make sure that "Load exception file" is not checked. Only need to be done once, not every time.

Then, close the PCSpim, and then, reopen it.
By then, all the examples in the lab notes will work fine.

4. If you wish to install PCSpim 9.1.9 or the other versions for your home computer, you can refer to the following website:
<http://sourceforge.net/projects/spimsimulator/files/>

SPIM is a simulator for the MIPS architecture.
In the lab, we are going to use PCSpim version 9.1.9.
For more information about SPIM, follow the link:
<http://www.cs.wisc.edu/~larus/spim.html>

MIPS Architecture

SPIM is a simulator for the MIPS. MIPS processor contains 32 general purpose registers that are numbered from 0 to 31. The registers are used in the SPIM instructions, particularly, in arithmetic operations.

Here is a summary of the MIPS register file for your reference.

Register Name	Number	Usage
Zero	\$0	Constant 0
\$at	\$1	Reserved for assembler
\$v0	\$2	Used for the expression evaluation and results of a function
\$v1	\$3	
\$a0	\$4	Used to pass arguments to functions
\$a1	\$5	
\$a2	\$6	
\$a3	\$7	
\$t0 - \$t7	\$8-\$15	Temporary (not preserved across call)
\$s0 - \$s7	\$16-\$23	Saved temporary (preserved across call)
\$t8 - \$t9	\$24-\$25	Temporary (not preserved across call)
\$k0 - \$k1	\$26-\$27	Reserved for OS kernel

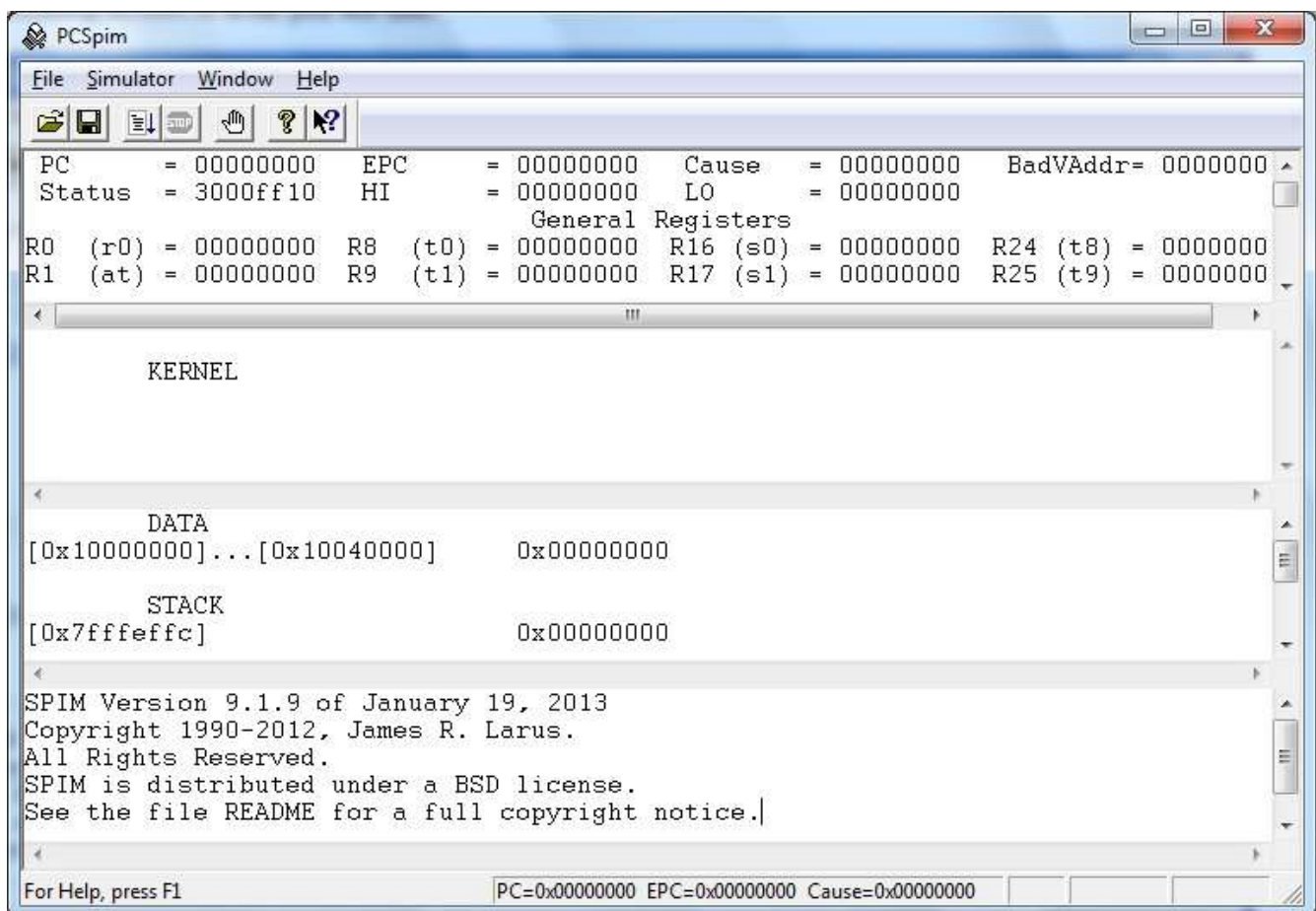
\$gp	\$28	Pointer to global area
\$sp	\$29	Stack Pointer
\$fp	\$30	Frame Pointer
\$ra	\$31	Return address (used by function call)

NOTE: The MIPS processor also has 16 floating point registers **\$f0, \$f1, ..., \$f15** to hold numbers in floating point form, such as 3.1415.

Start up PCSpim



You can start up **PCSpim** by clicking on the icon from the "Start" menu or "All Programs" on a lab PC. The following screen is what you will see.



Create an Assembly Language Program

To create an assembly language program, you need to use a text editor such as **NotePad** in Microsoft Windows environment. The file name must have a **.s** at the end. Let's assume that you have created the

following program called **hello.s** by using **NotePad** on a PC. The file **hello.s** contains the source code of the program to print out a character string "hello world". We will use PCSpim to open and execute this program so that you can get a whole picture of the process to see the program output. We will analyze this program in more details afterwards.

```
##
##      hello.s - prints out "hello world"
##
##      a0 - points to the string
##

#####
#
#      text segment
#
#####

        .text
        .globl __start
__start:      # execution starts here

        la $a0,str      # put string address into a0
        li $v0,4         # system call to print
        syscall          # out a string

        li $v0,10        # Exit
        syscall          # Bye!

#####
#
#      data segment
#
#####

        .data
str:      .asciiz "hello world\n"

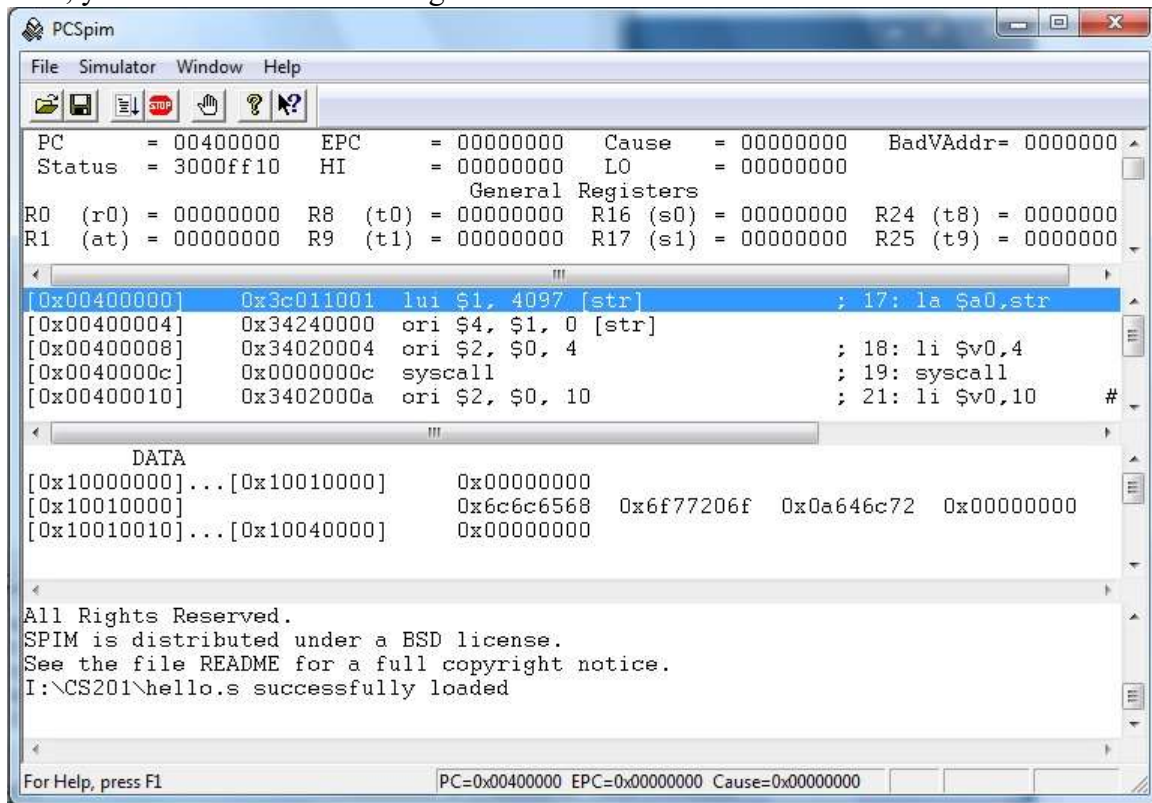
##
## end of file hello.s
```

Open and Run the Program

To open the program, click on the "File" menu from PCSpim screen and find the program that you want to open.



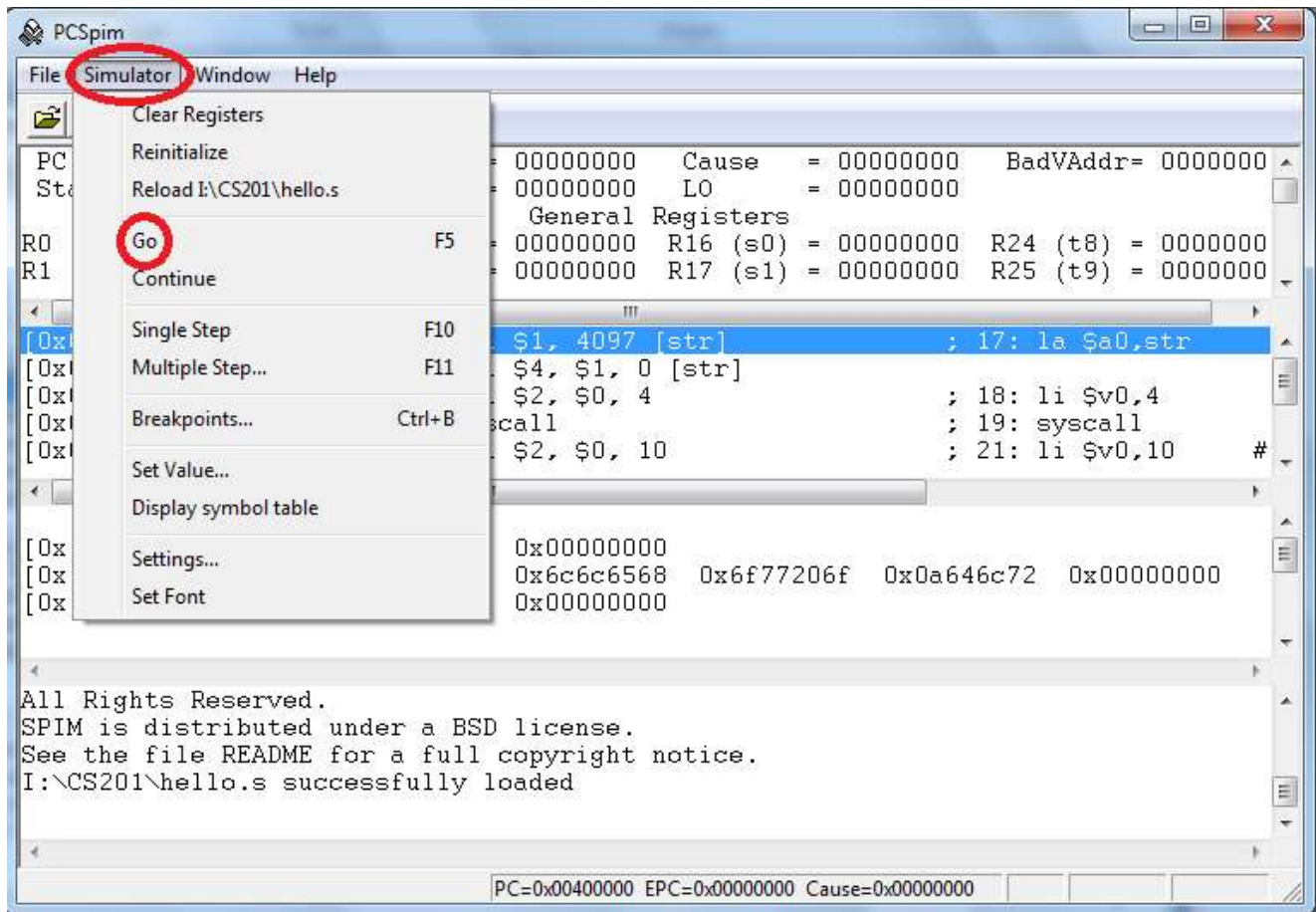
Then, you would see the following screen:



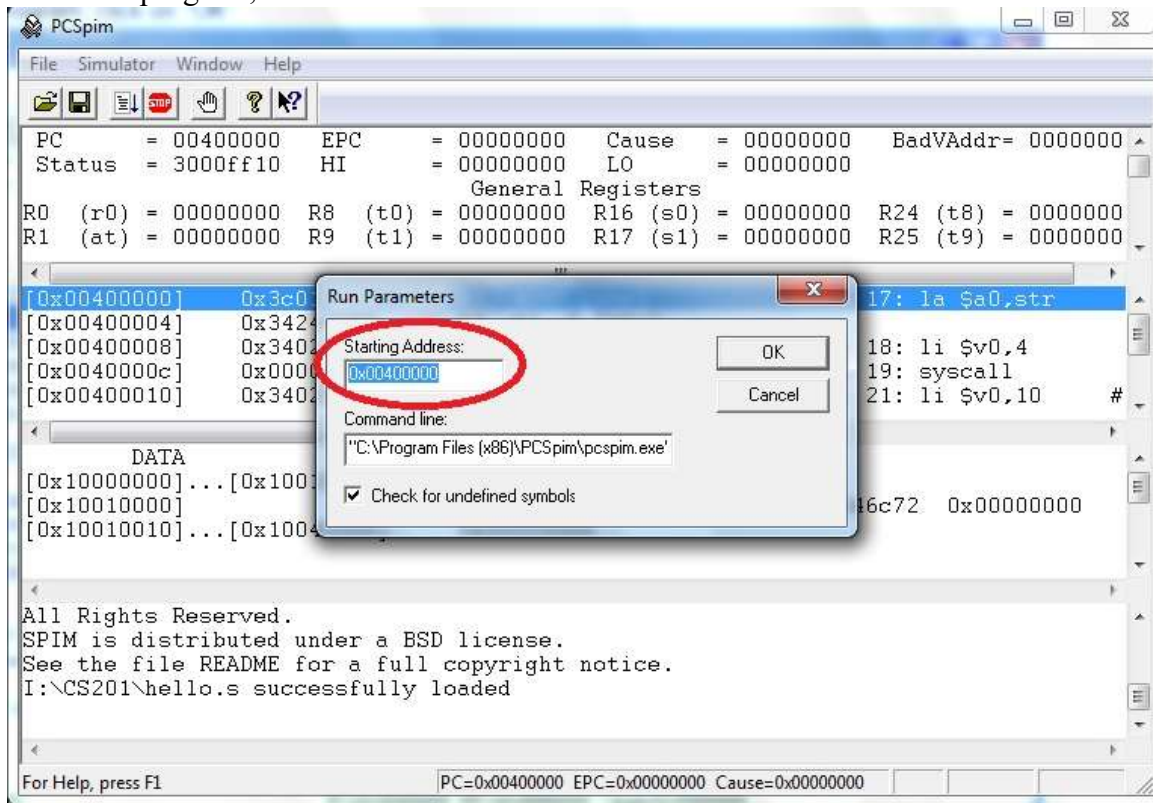
The image shows the PCSpim MIPS simulator window. At the top, the title bar reads 'PCSpim'. Below it is a menu bar with 'File', 'Simulator', 'Window', and 'Help'. A toolbar contains icons for file operations and simulation control. The main display area is divided into several sections:

- Registers:** A table showing the current state of MIPS registers. PC is 00400000, Status is 3000ff10, EPC is 00000000, HI is 00000000, Cause is 00000000, and BadVAddr is 00000000. General registers R0 through R25 are all shown as 00000000.
- Instructions:** A list of instructions with their addresses and assembly code. The first instruction at address 0x00400000 is highlighted in blue: `lui $1, 4097 [str] ; 17: la $a0, str`. Other instructions include `ori $4, $1, 0 [str]`, `ori $2, $0, 4 ; 18: li $v0, 4`, `syscall ; 19: syscall`, and `ori $2, $0, 10 ; 21: li $v0, 10`.
- DATA:** A section showing memory data at addresses 0x10000000 through 0x10040000. The data values are mostly 0x00000000, with some non-zero values at 0x10010000 and 0x10010010.
- Footer:** A status bar at the bottom shows 'For Help, press F1' and the current register values: 'PC=0x00400000 EPC=0x00000000 Cause=0x00000000'.

Now select "Go" from the "Simulator" menu:



To run the program, click on "OK".



Here is the program output on the "Console".



Now let's analyze the program and see how it works.

Analyze the Program

The MIPS assembly language program using PCSpim is usually held in a file with `.s` at the end of the file name.

For example **hello.s**. This file is processed line by line by the SPIM program. In this file,

A sharp sign `#` is used to begin inline comments.
i.e. Everything from the `#` to the end of the line is
ignored by the assembler.

An identifier is a sequence of alphanumeric characters,
underbars/underscores (`_`), and dots (`.`) that do not begin with a number.

A label is declared by putting identifiers at the beginning of a line followed by a colon. For example: **str:**

An operation field contains either a machine instruction or an assembler directive.

One or more operands are required by some machine instructions. Assembler directives may also require operands.

A constant is a value that does not change during program assembly or execution. A string constant is delimited by double quotes(""). For example: "Hello World\n"

The program **hello.s** prints out the message **hello world**.

It sets up the string in the data segment and makes a system call to print out this string in the text segment, followed by a system call to exit the program.

```
.text

    - a directive which tells the assembler to place
      what follows in the text segment.

.globl __start
__start:

    - These two lines attach the label __start
      to the first instruction so that the assembler
      can identify where the execution should begin.
      All the exercises in the lab will have these two
      lines unchanged.
```

I/O Manipulation - MIPS System Calls (syscall)

SPIM provides a set of operating-system-like services through the system call (syscall) instructions. Basic input and output can be managed and implemented by system calls.

To request a service, a program loads the system call code into register **\$v0** and arguments into registers **\$a0, \$a1, \$a2, and \$a3**.

Here is a summary of the system calls for your reference.

Service	Call Code in \$v0	Arguments	Results
Print Integer	1	\$a0=number (to be printed)	
		Example Code: li \$a0, 89 li \$v0, 1 syscall	
Print Float	2	\$f12=number	

```
Example Code:
l.s $f12, flabel
li $v0, 2
syscall
```

```
Print double      3          $f12=number
                      (to be printed)
```

```
Example Code:
1.d $f12, dlabel
li $v0, 3
syscall
```

```
Print String      4      $a0=address
                      (of string in memory)
```

```
Example Code:
    la $a0, str1
    li $v0, 4
    syscall
```

```
Read Integer      5                                number in $v0
```

```
Example Code:
li $v0, 5
syscall
```

Read Float	6	number in \$f0
------------	---	----------------

```
Example Code:
li $v0, 6
syscall
```

```
Read double      7                                number in $f0
```

```
Example Code:
li $v0, 7
syscall
```

```
Read String      8      $a0=address
                   (of input string in memory)
                   $a1=length of buffer(n bytes)
```

```
Example Code:
    la $a0, str1
li $a1, 80
    li $v0, 8
    syscall
```

```

        .data
str1:    .space 80

```

Sbrk	9	\$a0=n-byte	address in \$v0
(Dynamically		(to allocate)	
allocate n-byte			
of memory)			

Example Code:

```
li $a0, 80
li $v0, 9
syscall # Get memory

move $a0, $v0
li $a1, 80
li $v0, 8
syscall # Read String
li $v0, 4
syscall # Print String
```

Exit	10
------	----

Example Code:

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

A Few Special Characters

Character	Escape
=====	=====
Newline	\n
Tab	\t
Double Quote	\"

[MIPS Reference Card](#)

In all examples, \$1, \$2, \$3 represent registers. For class, you should use the register names, not the corresponding register numbers.

Arithmetic Instructions

Instruction	Example	Meaning	Comments
add	add \$1,\$2,\$3	$\$1 = \$2 + \$3$	
subtract	sub \$1,\$2,\$3	$\$1 = \$2 - \$3$	
add immediate	addi \$1,\$2,100	$\$1 = \$2 + 100$	"Immediate" means a constant number
add unsigned	addu \$1,\$2,\$3	$\$1 = \$2 + \$3$	Values are treated as unsigned integers, not two's complement integers
subtract unsigned	subu \$1,\$2,\$3	$\$1 = \$2 - \$3$	Values are treated as unsigned integers, not two's complement integers
add immediate unsigned	addiu \$1,\$2,100	$\$1 = \$2 + 100$	Values are treated as unsigned integers, not two's complement integers
Multiply (without overflow)	mul \$1,\$2,\$3	$\$1 = \$2 * \$3$	Result is only 32 bits!
Multiply	mult \$2,\$3	$\$hi, \$low = \$2 * \3	Upper 32 bits stored in special register hi Lower 32 bits stored in special register lo
Divide	div \$2,\$3	$\$hi, \$low = \$2 / \3	Remainder stored in special register hi Quotient stored in special register lo

Logical

Instruction	Example	Meaning	Comments
and	and \$1,\$2,\$3	$\$1 = \$2 \& \$3$	Bitwise AND
or	or \$1,\$2,\$3	$\$1 = \$2 \$3$	Bitwise OR
and immediate	andi \$1,\$2,100	$\$1 = \$2 \& 100$	Bitwise AND with immediate value
or immediate	ori \$1,\$2,100	$\$1 = \$2 100$	Bitwise OR with immediate value
shift left logical	sll \$1,\$2,10	$\$1 = \$2 \ll 10$	Shift left by constant number of bits
shift right logical	srl \$1,\$2,10	$\$1 = \$2 \gg 10$	Shift right by constant number of bits

Data Transfer

Instruction	Example	Meaning	Comments
load word	lw \$1,100(\$2)	$\$1 = \text{Memory}[\$2 + 100]$	Copy from memory to register
store word	sw \$1,100(\$2)	$\text{Memory}[\$2 + 100] = \1	Copy from register to memory

load upper immediate	lui \$1,100	\$1=100x2 ¹⁶	Load constant into upper 16 bits. Lower 16 bits are set to zero. <i>Pseudo-instruction</i> (provided by assembler, not processor!)
load address	la \$1,label	\$1=Address of label	Loads computed address of label (not its contents) into register <i>Pseudo-instruction</i> (provided by assembler, not processor!)
load immediate	li \$1,100	\$1=100	Loads immediate value into register
move from hi	mfhi \$2	\$2=hi	Copy from special register hi to general register
move from lo	mflo \$2	\$2=lo	Copy from special register lo to general register
move	move \$1,\$2	\$1=\$2	<i>Pseudo-instruction</i> (provided by assembler, not processor!) Copy from register to register.

Variations on load and store also exist for smaller data sizes:

- 16-bit halfword: lh and sh
- 8-bit byte: lb and sb

Conditional Branch

All conditional branch instructions compare the values in two registers together. If the comparison test is true, the branch is taken (i.e. the processor jumps to the new location). Otherwise, the processor continues on to the next instruction.

Instruction	Example	Meaning	Comments
branch on equal	beq \$1,\$2,100	if(\$1==\$2) go to PC+4+100	Test if registers are equal
branch on not equal	bne \$1,\$2,100	if(\$1!=\$2) go to PC+4+100	Test if registers are not equal
branch on greater than	bgt \$1,\$2,100	if(\$1>\$2) go to PC+4+100	<i>Pseudo-instruction</i>
branch on greater than or equal	bge \$1,\$2,100	if(\$1>=\$2) go to PC+4+100	<i>Pseudo-instruction</i>
branch on less than	blt \$1,\$2,100	if(\$1<\$2) go to PC+4+100	<i>Pseudo-instruction</i>
branch on less than or equal	ble \$1,\$2,100	if(\$1<=\$2) go to PC+4+100	<i>Pseudo-instruction</i>

Note 1: It is much easier to use a label for the branch instructions instead of an absolute number. For example: beq \$t0, \$t1, equal. The label "equal" should be defined somewhere else in the code.

Note 2: There are **many variations** of the above instructions that will **simplify writing programs!** Consult the [Resources](#) for further instructions, particularly H&P Appendix A.

Comparison

Instruction	Example	Meaning	Comments
set on less than	slt \$1,\$2,\$3	if(\$2<\$3)\$1=1; else \$1=0	Test if less than. If true, set \$1 to 1. Otherwise, set \$1 to 0.
set on less than immediate	slti \$1,\$2,100	if(\$2<100)\$1=1; else \$1=0	Test if less than. If true, set \$1 to 1. Otherwise, set \$1 to 0.

Note: There are **many variations** of the above instructions that will **simplify writing programs!** Consult the [Resources](#) for further instructions, particularly H&P Appendix A.

Unconditional Jump

Instruction	Example	Meaning	Comments
jump	j 1000	go to address 1000	Jump to target address
jump register	jr \$1	go to address stored in \$1	For switch, procedure return
jump and link	jal 1000	\$ra=PC+4; go to address 1000	Use when making procedure call. This saves the return address in \$ra

Note: It is much easier to use a label for the jump instructions instead of an absolute number. For example: j loop. That label should be defined somewhere else in the code.

System Calls

The SPIM simulator provides a number of useful system calls. These are **simulated**, and **do not represent MIPS processor instructions**. In a real computer, they would be implemented by the operating system and/or standard library.

System calls are used for input and output, and to exit the program. They are initiated by the syscall instruction. In order to use this instruction, you must first supply the appropriate arguments in registers \$v0, \$a0-\$a1, or \$f12, depending on the specific call desired. (In other words, not all registers are used by all system calls). The syscall will return the result value (if any) in register \$v0 (integers) or \$f0 (floating-point).

Available syscall services in SPIM:

Service	Operation	Code (in \$v0)	Arguments	Results
print_int	Print integer number (32 bit)	1	\$a0 = integer to be printed	None

print_float	Print floating-point number (32 bit)	2	\$f12 = float to be printed	None
print_double	Print floating-point number (64 bit)	3	\$f12 = double to be printed	None
print_string	Print null-terminated character string	4	\$a0 = address of string in memory	None
read_int	Read integer number from user	5	None	Integer returned in \$v0
read_float	Read floating-point number from user	6	None	Float returned in \$f0
read_double	Read double floating-point number from user	7	None	Double returned in \$f0
read_string	Works the same as Standard C Library fgets() function.	8	\$a0 = memory address of string input buffer \$a1 = length of string buffer (n)	None
sbrk	Returns the address to a block of memory containing n additional bytes. (Useful for dynamic memory allocation)	9	\$a0 = amount	address in \$v0
exit	Stop program from running	10	None	None
print_char	Print character	11	\$a0 = character to be printed	None
read_char	Read character from user	12	None	Char returned in \$v0
exit2	Stops program from running and returns an integer	17	\$a0 = result (integer number)	None

Notes:

- The **print_string** service expects the address to start a null-terminated character string. The directive **.asciiz** creates a null-terminated character string.
- The **read_int**, **read_float** and **read_double** services read an entire line of input up to and including the newline character.
- The **read_string** service has the same semantics as the C Standard Library routine fgets().
 - The programmer must first allocate a buffer to receive the string
 - The read_string service reads up to *n-1* characters into a buffer and terminates the string with a null character.
 - If fewer than *n-1* characters are in the current line, the service reads up to and including the newline and terminates the string with a null character.
- There are a few additional system calls not shown above for file I/O: **open**, **read**, **write**, **close** (with codes 13-16)

Assembler Directives

An assembler directive allows you to request the assembler to do something when converting your source code to binary code.

Directive	Result
.word w1, ..., wn	Store n 32-bit values in successive memory words
.half h1, ..., hn	Store n 16-bit values in successive memory words
.byte b1, ..., bn	Store n 8-bit values in successive memory words
.ascii str	Store the ASCII string str in memory. Strings are in double-quotes, i.e. "Computer Science"
.asciiz str	Store the ASCII string str in memory and null-terminate it Strings are in double-quotes, i.e. "Computer Science"
.space n	Leave an empty n -byte region of memory for later use
.align n	Align the next datum on a 2^n byte boundary. For example, .align 2 aligns the next value on a word boundary

Registers

MIPS has 32 general-purpose registers that could, technically, be used in any manner the programmer desires. However, by convention, registers have been divided into groups and used for different purposes. Registers have both a *number* (used by the hardware) and a *name* (used by the assembly programmer).

This table **omits special-purpose registers** that will not be used in ECPE 170.

Register Number	Register Name	Description
0	\$zero	The value 0
2-3	\$v0 - \$v1	(values) from expression evaluation and function results
4-7	\$a0 - \$a3	(arguments) First four parameters for subroutine
8-15, 24-25	\$t0 - \$t9	Temporary variables
16-23	\$s0 - \$s7	Saved values representing final computed results
31	\$ra	Return address

LAB

MIPS Assembly Programming (Basic)

(Individual) (10 Marks)

Overview

In this lab, you will gain experience with the SPIM simulator and MIPS assembly programming by writing a sequence of programs with increasing complexity.

Lab - Getting Started

We will be using **SPIM**, a MIPS simulator, in order to learn assembly programming. Before lab, you should **complete the QtSPIM tutorial**, including installing the simulator and running your first program.

Familiarize yourself with the various MIPS reference materials on the Resources page, particularly the **MIPS Instruction Set** overview that are already given.

Review the **MIPS Example Programs** to see basic arithmetic, looping, I/O, and function calls, plus good assembly programming style that are already given.

If you are using the GEdit text editor, don't forget to install the syntax highlighting extension, which makes your code easier to read.

Programming Requirements (for all exercises):

(1) **File names:** Names your assembly source code something which makes it obvious what the file contains, like: part1.asm

(2) **Commenting:** ****Each line** of assembly code must be documented with a comment!**

This comment can be terse. For example, "i++" is a perfectly reasonable comment for the line `addi $t1,$t1,1` where register \$t1 holds the loop counter i.

(3) **Commenting:** Each region of assembly code must clearly have a ****comment block**** documenting the overall purpose of that region, along with what values each register holds. You can use your own judgement for a reasonable region size.

Lab Submission:

(1) All source code and lab report PDF must be submitted.

Lab Part 1 - Arithmetic

Write a complete MIPS program that calculates the equation shown below (in C):

```
int main()
{
    int A=15;
    int B=10;
    int C=5;
    int D=2;
    int E=18;
    int F=-3;
    int Z=0;

    Z = (A-B) * (C+D) + (E-F) - (A/C);
}
```

A-F can be stored in temporary registers. **However, the final result Z must be an integer word stored in memory when your program finishes executing. It is also a good idea to write this C/C++ program and run it to be sure your assembly code uses the same operator precedence rules.**

Lab Report:

(1) Take two screenshots of the MIPS register panel: one before your program runs, and one after your program finishes. Put the register panel in Decimal mode (right-click) so it is easy to see register values.
(2) Take two screenshots of the MIPS memory panel (data tab): one before your program runs, and one after your program finishes. Put the memory panel in Decimal mode (right-click), so it is easy to see memory values. In the after-execution capture, **circle the memory location (not register) that contains the final calculated value of Z.**

Lab Part 2 - Branches

Write a complete MIPS program that implements the same algorithm shown below (in C):

```
int main()
{
    // Note: I should be able to change
    // the values of A, B, and C when testing
    // your code, and get correct output each time!
    // (i.e. don't just hardwire your output)
    int A=10;
    int B=15;
    int C=6;
    int Z=0;

    if(A > B || ((C+1) == 7))
        Z = 1;
    else if(A < B && C > 5)
        Z = 2;
    else
```

```

    Z = 3;

switch(Z) {
    case 1:
        Z = -1;
        break;
    case 2:
        Z = -2;
        break;
    case 3:
        Z = -3;
        break;
    default:
        Z = 0;
        break;
}
}

```

A-C and Z must be integer words in memory, both when the program begins and when the program ends. In between, they can be stored in registers.

Tip: Consult the Resources page for extra MIPS instructions (or pseduo-instructions) to make the task of comparing values and branching easier.

Lab Report:

- (3) Take two screenshots of the MIPS register panel: one before your program runs, and one after your program finishes. Put the register panel in Decimal mode (right-click) so it is easy to see register values.
- (4) Take two screenshots of the MIPS memory panel (data tab): one before your program runs, and one after your program finishes. Put the memory panel in Decimal mode (right-click), so it is easy to see memory values. In the after-execution capture, **circle the memory location (not register) that contains the final calculated value of Z.**

Lab Part 3 - Loops

Write a complete MIPS program that implements the same algorithm shown below (in C):

```

int main()
{
    int Z=4;
    int i;

    for(i=0; i<=21; i=i+3) {
        Z++;
    }

    do {
        Z++;
    } while (Z<100);

    while(i > 0) {
        Z--;
        i--;
    }
}

```



```
}  
}
```

I and Z must be integer words in memory.

Lab Report:

(5) Take a screenshot of the MIPS register panel after your program finishes. Put the register panel in Decimal mode (right-click) so it is easy to see register values.

(6) Take a screenshot of the MIPS memory panel (data tab) after your program finishes. Put the memory panel in Decimal mode (right-click), so it is easy to see memory values. **Circle the memory location (not register) that contains the final calculated value of Z.**

Lab Part 4 - Arrays

Write a complete MIPS program that implements the same algorithm shown below (in C):

```
int main()  
{  
    int A[5]; // Empty memory region for 5 elements  
    int B[5] = {1,2,4,8,16};  
    int i;  
  
    for(i=0; i<5; i++) {  
        A[i] = B[i] - 1;  
    }  
  
    i--;  
    while(i >= 0) {  
        A[i]=(A[i]+B[i]) * 2;  
        i--;  
    }  
}
```

A and B must be arrays of integer words in memory, but i can be a register.

Lab Report:

(7) Take a screenshot of the MIPS register panel after your program finishes. Put the register panel in Decimal mode (right-click) so it is easy to see register values.

(8) Take a screenshot of the MIPS memory panel (data tab) after your program finishes. Put the memory panel in Decimal mode (right-click), so it is easy to see memory values. **Circle the final values of array A.**

Lab Part 5 - I/O, Loops, and Arrays

Write a complete MIPS program that implements the same algorithm shown below (in C):

```
int main()  
{
```

```

char string[256];
int i=0;
char *result = NULL; // NULL pointer is binary zero

// Obtain string from user, e.g. "Constantinople"
scanf("%255s", string);

// Search string for letter 'm'.
// Result is pointer to first m (if it exists)
// or NULL pointer if it does not exist
while(string[i] != '\0') {
    if(string[i] == 'm') {
        result = &string[i];
        break; // exit from while loop early
    }
    i++;
}

if(result != NULL)
    printf("First match at address %d\n", result);
else
    printf("No match found\n");
}

```

The array of characters is an array of **bytes**, not words! The result pointer must be stored in memory when the program finishes.

10 points extra credit: write code to convert the value of '**result**' into a hexadecimal string and print it out.

Lab Report:

(9) Take a screenshot of the MIPS memory panel (data tab) after your program finishes. Put the memory panel in Hex mode (right-click), since Decimal mode will not allow us to distinguish between bytes. **Circle two things: the final value of the pointer 'result' in memory, and the corresponding location that result points to.** Does that location in memory contain the ASCII code for the character 'm'? *(If not, you had better check your work!)*

Lab Report - Wrapup:

- (1) What was the best aspect of this lab?
- (2) What was the worst aspect of this lab?
- (3) How would you suggest improving this lab in future semesters?

MIPS Assembly Programming (Advanced)

(Group wise) (10 Marks)

Overview

In this lab, you are going to write a program (in both C and MIPS assembly) that allows a human to play against a computer in "Dots and Boxes".

Lab

In this game, you have a 4 by 4 grid of dots. At the beginning, the computer will randomly pick which player (the human or the computer) moves first. That player draws a line between two adjacent dots (in a row or column). The next player then does the same thing. At any point, if a player adds a line which creates a box then that player "wins" the box and the box is filled in with their identifier (in this case, either a "H" or "C"). The first player to fill in 5 or more boxes wins the game.

You should not allow the player to make an illegal move, or to enter invalid input. For example, a player can't redraw an existing line or attempt to draw two lines at once.

The computer's "strategy" for taking a turn should be very simple.

1. Randomly select two adjacent coordinates.
2. Check that no line already exists between those coordinates; if one does, go back to Step 1.
3. Draw the line and check to see if any new squares are filled in, and if so has it won.

At the end of a game, you should print out a message indicating who won. An example of the game output is shown here. Your output does not have to match this exactly.

```
Welcome to Dots and Boxes!
Version 1.0
Implemented by Ken Hughes
```

```
Enter two positive numbers to initialize the random number generator.
Number 1: 123
Number 2: 456
```

```
I will make the first move.
```

```
My turn: I draw between b1 and c1
```

```
  1 2 3 4
a  . . . .
b  . . . .
c  | . . .
d  . . . .
```

```
Your turn:
```

```
Enter coordinate of the first dot: b1
Enter coordinate of the second dot: c1
```

There is already a line there.

Enter coordinate of the first dot: b1

Enter coordinate of the second dot: c2

Coordinates must be adjacent

Enter coordinate of the first dot: a1

Enter coordinate of the second dot: a2

My turn: I draw between a3 and a4

```
  1 2 3 4
a  . _ . _ .
b  . . . .
c  | . . .
d  . . . .
```

Your turn:

Enter coordinate of the first dot: a2

Enter coordinate of the second dot: a3

My turn: I draw between c1 and d1

```
  1 2 3 4
a  . _ . _ .
b  . . . .
c  | . . .
d  | . . .
```

Your turn:

Enter coordinate of the first dot: a1

Enter coordinate of the second dot: b1

My turn: I draw between c2 and c3

```
  1 2 3 4
a  . _ . _ .
b  | . . .
c  | . _ .
d  | . . .
```

Your turn:

Enter coordinate of the first dot: c1

Enter coordinate of the second dot: c2

My turn: I draw between b3 and b4

```
  1 2 3 4
a  . _ . _ .
b  | . . _
c  | _ . .
d  | . . .
```

Your turn:

Enter coordinate of the first dot: c3

Enter coordinate of the second dot: c3

Coordinates must be adjacent

Enter coordinate of the first dot: b3

Enter coordinate of the second dot: c3

My turn: I draw between d1 and d2

```
  1 2 3 4
a  . _ . _ .
b  | . . _
c  | _ _ | .
d  | _ _ | .
```

d |_. . .

Your turn:

Enter coordinate of the first dot: c4

Enter coordinate of the second dot: d4

My turn: I draw between a4 and b4

```
  1 2 3 4
a  .-.-.-
b  | | . _|
c  | _ _| .
d  | _ . . |
```

Your turn:

Enter coordinate of the first dot: b1

Enter coordinate of the second dot: b2

My turn: I draw between d2 and d3

```
  1 2 3 4
a  .-.-.-
b  | _ . _|
c  | _ _| .
d  | _ . . |
```

Your turn:

Enter coordinate of the first dot: a2

Enter coordinate of the second dot: b2

My turn: I draw between c2 and d2

```
  1 2 3 4
a  .-.-.-
b  |H| . _|
c  | _ _| .
d  |C| _ . |
```

Your turn:

Enter coordinate of the first dot: d3

Enter coordinate of the second dot: d4

My turn: I draw between c3 and d3

```
  1 2 3 4
a  .-.-.-
b  |H| . _|
c  | _ _|C|
d  |C|C| _|
```

Your turn:

Enter coordinate of the first dot: c3

Enter coordinate of the second dot: c4

My turn: I draw between b4 and c4

```
  1 2 3 4
a  .-.-.-
b  |H| . _|
c  | _ _|C|
d  |C|C|H|
```

Your turn:

Enter coordinate of the first dot: a3

Enter coordinate of the second dot: b3

My turn: I draw between b2 and c2

```
  1 2 3 4
a  . _ . _ .
b  |H| _ |H|
c  |C| _ |C|
d  |C|C|H|
```

Your turn:

```
Enter coordinate of the first dot: b2
Enter coordinate of the second dot: b3
You win!
```

```
  1 2 3 4
a  . _ . _ .
b  |H|H|H|
c  |C|H|C|
d  |C|C|H|
```

Part 1: Implement the Dots and Boxes game in C

- (1) You **cannot** use library functions other than `printf()` and `scanf()`. Specifically, you ***cannot*** use `rand()`, because there is no analogous function in the MIPS simulator or assembly language.
- (2) You must provide a **Makefile**.
- (3) **Your game must use at least 3 subroutines -- feel free to use more.** At least one of the subroutines must have parameters and at least one must return a value. For example, you could use: `check_board(player_id)`, `print_board()`, `user_input()`.
- (4) You must *fully check* for valid inputs. Coordinates must always be a letter followed by a number, and the values must be in range.

There is no "C Standard Library" in MIPS. The simulator provides analogous functions for `printf()` and `scanf()`, but nothing to generate random numbers. This snippet code of C code from [Wikipedia](https://en.cppreference.com/w/cpp/algorithm/random) provides an acceptable random number generator for the project. Note that it produces a random 32-bit number, so if you want a random number in a smaller range, use modular division afterwards. Prompt the user for two random numbers to use as seeds (for `m_w` and `m_z`), so your program doesn't produce the same random number sequence each time. The `m_w` and `m_z` variables should be global, or at least persist after the function finishes. (Otherwise, your random number generator will always produce the same output).

```
m_w = <choose-initializer>;    /* must not be zero */
m_z = <choose-initializer>;    /* must not be zero */

uint32_t get_random()
{
    m_z = 36969 * (m_z & 65535) + (m_z >> 16);
    m_w = 18000 * (m_w & 65535) + (m_w >> 16);
    return (m_z << 16) + m_w; /* 32-bit result */
}
```

You should **completely test and debug your C implementation** before moving on to Part 2. Assembly programming is slow and tedious. You DO NOT want to have to fix bugs later that were caused by a sloppy C implementation!

Part 2: Implement the Dots and Boxes game in MIPS assembly

(1) ****Each line** of assembly code must be documented with a comment!**

(2) Each region of assembly code must clearly have a ****comment block**** documenting the overall purpose of that region, along with what values each register holds. You can use your own judgement for a reasonable region size.

(3) **Implement the same subroutines as in your C program.** They must take the same parameters and return the same values, although it is up to you to determine the exact implementation. For example, if you have a C subroutine which returns an array of two values for a coordinate, you may decide to return those values in \$v0 and \$v1.

(4) You can assume that all user inputs have the proper format (a valid letter followed by a valid number); you still must check that coordinates are adjacent and have not already been played.

To reduce the work involved in the MIPS implementation (part 2), you are permitted to replace the computer player with a second human player. This simplification is only for the MIPS part of the assignment. The C implementation must provide all of the features of the full game. However, you can earn 10 points of extra credit for a full MIPS implementation.

Tip: In-class discussion during the time period allotted to Lab 11 will discuss generating a random number in assembly. Sample code will be generated on the whiteboard.

Lab Submission:

(1) There is no lab report for this lab.

(2) All source code must be submitted via Mercurial. Place the source files inside the lab11 folder that was previously created.

Optional Feedback:

(Feel free to include a text file with comments/feedback on the lab)

(1) What was the best aspect of this lab?

(2) What was the worst aspect of this lab?

(3) How would you suggest improving this lab in future semesters?

Submission Date:

1st April, 2017 (Important)