Very brief SPIM tutorial for 3827

Prof. Rubenstein

November 7, 2018

1 SPIM Overview

SPIM is a MIPS32 Emulator that we will use. You can download and see detailed documentation at `http://spimsimulator.sourceforge.net`. SPIM will *emulate execution* your MIPS assembly code, which means that the code is not assembled into machine code, and is not directly run on the microprocessor of your machine (your machine's microprocessor is probably not directly MIPS-compatible anyhow).

A simple sample SPIM program, available at `http://www.cs.uic.edu/~troy/spring04/cs366/ex2.html`, is presented below to familiarize us with SPIM.

In class, we cover the MIPS instruction set architecture. To write a program, there are a few more things we need to cover:

- Data and code "placement" in memory: we need to instruct the SPIM simulator where to put our code and the code's data in (SPIM's simulated) memory. Rather than put the code at a fixed address, we can label the various data items we need placed and then let the simulator choose the addresses automatically (and in real assembly code, the assembler could similarly choose the memory locations).
- Pseudo-instructions: MIPS is a RISC (reduced instruction set) architecture, so there are a limited set of simple instructions made available. More complex operations will be represented as a pseudo-instruction. The SPIM assembler recognizes the pseudo-instructions and knows how to convert it into a (series of) actual MIPS instructions.

A simple example of a pseudo-instruction is:

```
move $a0, $s0
```

which copies the value from register $s0 into register $a0. This is not a "real" MIPS instruction, but the MIPS assembler can convert this instruction to:

```
add $a0, $s0, $zero
```

which takes the value in $s0, adds the zero registers value (which is 0) and stores that result in $a0. That's about the simplest pseudo-instruction there is. Some are more complex, discussed later.

First, look at the entire Sample SPIM program (on the next page so it's all on a single page):

```
# Sample spim program
#
# Written by Pat Troy, 11/8/2002
.data
prompt: .asciiz "Enter in an integer: "
str1: .asciiz "the answer is: "
newline: .asciiz "\n"
bye: .asciiz "Goodbye!\n"
.globl main

.text
main:

# initialize
li $s0, 10

# prompt for input
li $v0, 4
la $a0, prompt
syscall

# read in the value
li $v0, 5
syscall
move  $s0, $v0

loop:
# print str1
li $v0, 4
la $a0, str1
syscall

# print loop value
li $v0, 1
move $a0, $s0
syscall

# print newline
li $v0, 4
la $a0, newline
syscall

# decrement loop value and branch if not negative
sub $s0, $s0, 1
bgez $s0, loop

# print goodbye message
li $v0, 4
la $a0, bye
syscall
li $v0, 10 ## Added by Prof. Rubenstein for proper termination
```

```
syscall  ## Added by Prof. Rubenstein for proper termination
```

2 Breakdown of the code

- **Comments**: On any line of code, anything after a '#' is a comment and is ignored by the assembler (it's there to help understand the code).
- **Labels**: On any line that starts with alpha-numeric text and ends with a ':' is a label. What immediately follows the label (code or program data) will be stored in memory at some address and that label can be used within your assembly code to reference that address. The assembler will determine the address.
- **Directives**: Text preceded by a '.' are directives. They provide the assembler with some special information. In this example we have:
  - **.data**: Tells the assembler that the following information is **program data** that should be stored in memory. For instance, ASCII strings that need to be printed are often stored here, each with its own label for reference by the program.
  - **.asciiz**: An ASCII string that should be stored in memory (often also provided with its own label so that it can be directly accessed by the program), and terminated with a NULL (ASCII value 0) character. Note that the **.ascii** directive doesn't provide the NULL termination. However, when ASCII text is printed, the printing stops when a NULL character is reached, so it's generally important to include the NULL character.
  - **.globl**: Followed by a label, that label is now *global*, meaning that it can be accessed by MIPS assembler code in another file (like declaring a function in a header file in c). We don't need this, and it didn't have to be in the sample code.
  - **.text**: informs the assembler that what follows is code.
- **Pseudo-instructions**: these are instructions that are not actually part of MIPS, but which can easily be converted to (one or more) MIPS instructions by the assembler:
  - **li**: Load-immediate: loads a constant into a register. li REG const maps directly to ORI REG $zero const for a constant no larger than 2^{16}. For larger constants, it maps to the combination LUI (load-upper-immediate) and ORI (to be discussed in class).
  - **la**: Load-address into a register. This pseudo-instruction is useful because a) it can be given a label for some data and the assembler will figure out the corresponding address and b) addresses are 32-bit quantities, and since an instruction needs to use some bits of the 32-bit instruction word to indicate the instruction, the 32 bits (sometimes) cannot be passed in within a single instruction. Generally, **la** will be converted to the pair of instructions **lui** (load-upper-immediate) which loads a 16-bit constant into the 16 **high-order** bits and zeros the 16 low order bits, followed by an **ori** (or-immediate) that fills in the remaining 16 bits. Note that in some cases, the address may have all 0's in either the highest or lowest 16 significant bits, in which case, the pseudo-instruction can map to a single instruction.
  - **syscall**: This is a very powerful pseudo-instruction. Its meaning is to call make a *system call* to the kernel (perhaps implemented as an interrupt or as a procedure call, the details are not spelled out). **syscall** can perform one of several tasks - the task which it is to perform is indicated by the value to which $v0 is set, and the task is performed on the data in (or, depending on the value of $v0, the data at the address) stored in $a0. The tasks are (for the value in $v0):
    1. print integer
    2. print float
    3. print double
    4. print string ($a0 holds address of string)
    5. read integer

6. read float
7. read double
8. read string ($a0 holds address to write to)
9. sbrk (allocate memory, like malloc, in bytes)
10. exit (terminate the program)
11. print character
12. read character
13. open (file)
14. read (from file)
15. write (to file)
16. close
17. exit2 (exit and return a value)

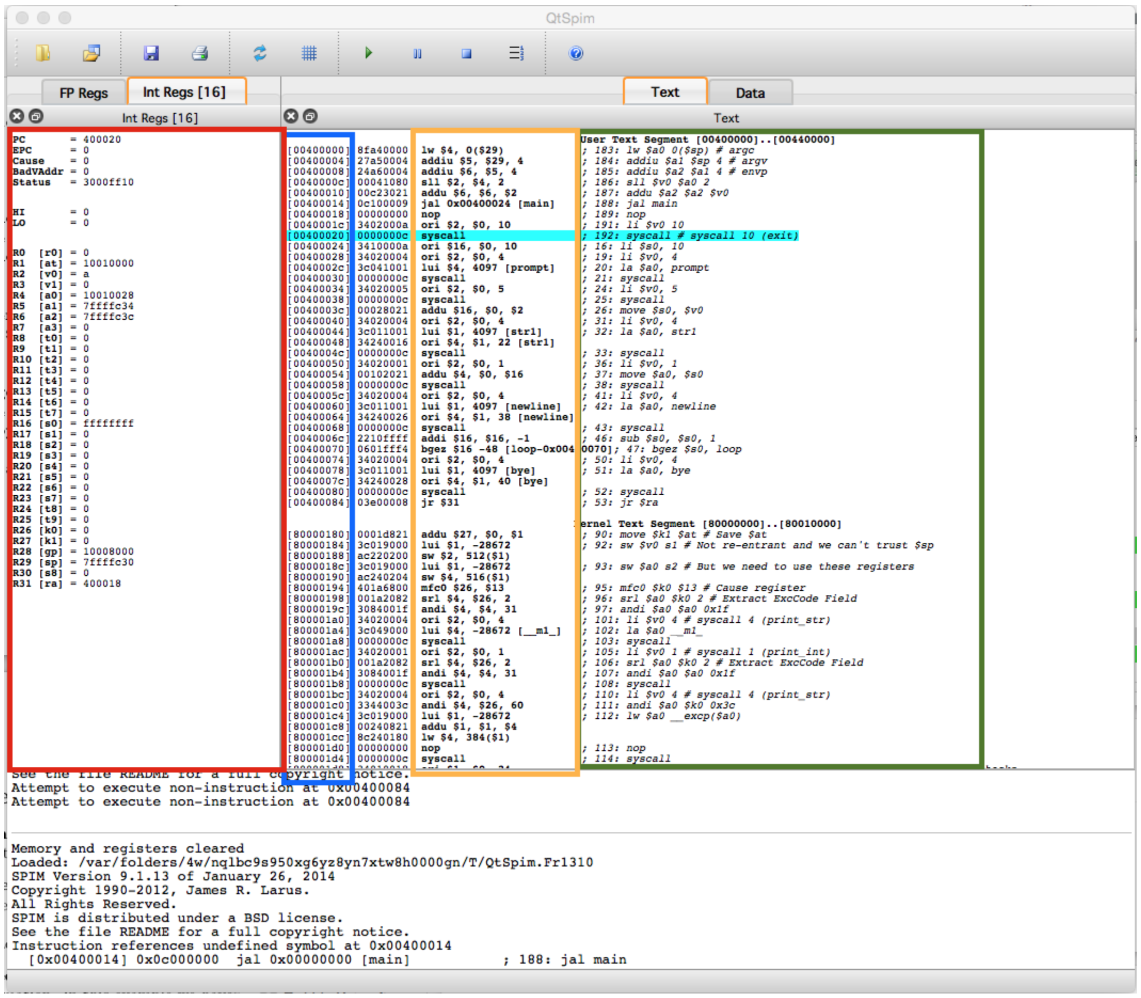
– **move** and **sub**: discussed in class. **move** is simply **add** with the $zero register, and **sub** is **addi** with the constant negated.

2.1 Small error in code on webpage

Note that there is a small error/oversight in the code on the webpage. The main function is itself a procedure, and should **return** control when done. Without the **jr** instruction at the end, the emulator will register an error as it calls "goodbye" but then fails to return control - control just keeps moving forward and runs into garbage.

3 Using SPIM

What is described here is based on the QTSpim emulator, as run on a Mac. The main window looks like this (colored boxes added by Prof. Rubenstein):



```
PC       = 400020
EPC      = 0
Cause    = 0
BadVAddr = 0
Status   = 3000ff10


HI       = 0
LO       = 0


R0  [r0] = 0
R1  [at] = 10010000
R2  [v0] = a
R3  [v1] = 0
R4  [a0] = 10010028
R5  [a1] = 7ffffc34
R6  [a2] = 7ffffc3c
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] = ffffffff
R17 [s1] = 0
R18 [s2] = 0
R19 [s3] = 0
R20 [s4] = 0
R21 [s5] = 0
R22 [s6] = 0
R23 [s7] = 0
R24 [t8] = 0
R25 [t9] = 0
R26 [k0] = 0
R27 [k1] = 0
R28 [gp] = 10008000
R29 [sp] = 7ffffc30
R30 [s8] = 0
R31 [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
[0040001c] 3402000a  ori $2, $0, 10                  ; 191: li $v0 10
[00400020] 0000000c  syscall                         ; 192: syscall # syscall 10 (exit)
[00400024] 3410000a  ori $16, $0, 10                 ; 16: li $s0, 10
[00400028] 34020004  ori $2, $0, 4                   ; 19: li $v0, 4
[0040002c] 3c041001  lui $4, 4097 [prompt]           ; 20: la $a0, prompt
[00400030] 0000000c  syscall                         ; 21: syscall
[00400034] 34020005  ori $2, $0, 5                   ; 24: li $v0, 5
[00400038] 0000000c  syscall                         ; 25: syscall
[0040003c] 00028021  addu $16, $0, $2                ; 26: move $s0, $v0
[00400040] 34020004  ori $2, $0, 4                   ; 31: li $v0, 4
[00400044] 3c011001  lui $1, 4097 [str1]             ; 32: la $a0, str1
[00400048] 34240016  ori $4, $1, 22 [str1]
[0040004c] 0000000c  syscall                         ; 33: syscall
[00400050] 34020001  ori $2, $0, 1                   ; 36: li $v0, 1
[00400054] 00102021  addu $4, $0, $16                ; 37: move $a0, $s0
[00400058] 0000000c  syscall                         ; 38: syscall
[0040005c] 34020004  ori $2, $0, 4                   ; 41: li $v0, 4
[00400060] 3c011001  lui $1, 4097 [newline]          ; 42: la $a0, newline
[00400064] 34240026  ori $4, $1, 38 [newline]
[00400068] 0000000c  syscall                         ; 43: syscall
[0040006c] 2210ffff  addi $16, $16, -1               ; 46: sub $s0, $s0, 1
[00400070] 0601fff4  bgez $16 -48 [loop-0x004 0070]; 47: bgez $s0, loop
[00400074] 34020004  ori $2, $0, 4                   ; 50: li $v0, 4
[00400078] 3c011001  lui $1, 4097 [bye]              ; 51: la $a0, bye
[0040007c] 34240028  ori $4, $1, 40 [bye]
[00400080] 0000000c  syscall                         ; 52: syscall
[00400084] 03e00008  jr $31                          ; 53: jr $ra

                                                     ernel 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
[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
```

```
See the file README for a full copyright notice.
Attempt to execute non-instruction at 0x00400084
Attempt to execute non-instruction at 0x00400084

Memory and registers cleared
Loaded: /var/folders/4w/nqlbc9s950xg6yz8yn7xtw8h0000gn/T/QtSpim.Fr1310
SPIM Version 9.1.13 of January 26, 2014
Copyright 1990-2012, James R. Larus.
All Rights Reserved.
SPIM is distributed under a BSD license.
See the file README for a full copyright notice.
Instruction references undefined symbol at 0x00400014
  [0x00400014] 0x0c000000  jal 0x00000000 [main]          ; 188: jal main
```

Figure 1: QTSpim as it appears on a Mac

Here is what you are looking at. This window contains several sub-windows - the views of the sub-windows can be modified.

- Red box: the register file (and some additional registers outside the register file). All 32 registers in the register file are listed, along with their values (in hex, though there is an option to view in decimal or binary). Some “special” registers are listed above the register file. Of specific importance to us is **PC**: the program counter, the register that “points” to the address containing the current instruction being executed. In the snapshot, the PC equals 400020 (hex) which means that the instruction at 400020 is currently (next) to be executed. Note that the “Int Regs” are currently showing. There are also floating point registers in SPIM - we won’t be using those.

- Blue box: Addresses. We are looking at the “Text” window, which means we are looking at the code. Note that the address held by the PC is highlighted (light blue) to indicate that this address holds the “current” instruction.

- Orange box: The “actual” instructions, any pseudo-code has been converted (note that **syscall** is still there, so it should be considered a special instruction with no substitute.
- Green box: comments, which describe the (pseudo)-code that led to the given instruction. Note, for instance, the commented line numbered 51 (la $a0, bye). This pseudo-instruction is converted into two real instructions. Note also that the actual code has inserted values for labels (for some reason, the decimal value is shown, which is particularly annoying when it corresponds to the 16 higher-order bits of an address, e.g., line 51: 4097 (which is 1001 in hex) is loaded into the high order bits of $a0, and the “bye” data in fact resides at (hex) address 10010028 (i.e. in decimal, $4097 \cdot 2^{16} + 40$.

Note that the combination of information in the green box (your code) and orange box (actual instructions) is helpful to see how labels are mapped to addresses, and to see the mapping of register names (e.g., $s0) to its actual register number (e.g., $16 for $s0).

4 Running SPIM

This documentation is specific to the version on a Mac, although other versions are probably very similar, specific key clicks are likely to be different.

4.1 Loading and Running Code

Since your code should be in a single file, use the “File → Load and Reinitialize” option when loading in your code.

You can run your code by choosing “Simulator → Run/Continue” or F5. The code will run until normal termination, error, or hitting a breakpoint (see below).

Alternatively, you can then execute your code one instruction at a time by choosing “Simulator → Single Step” or F10.

You can set a breakpoint at any instruction, where the execution will be paused when control (i.e., the address in the program counter (PC)) reaches that point. To set a breakpoint, hold down ctrl and click on the instruction for which you wish to set a breakpoint. This process can also be used to remove the breakpoint.

4.2 Restarting

Assuming your code is not affected by initial values in registers in the register file (i.e., it begins by loading into the registers the values it needs), you re-run your code by holding down ctrl and clicking on the PC (top left of the Int Regs screen) and selecting “Change Register Contents”. My code was also assigned a starting address of 0x400000 (hex), so to restart, I set PC to 40000, and then run again.

You can instead select to clear all registers in the menu. If you then click on “Run Parmeters” (sic), a box appears that states “Address or label to start program” - in which you can enter your desired starting address (it seems to generally pick the right starting location).

4.3 Debugging

Debugging MIPS can be very challenging. However, you can have the simulator step through the code one instruction at a time. Also, you can set a “breakpoint” at an instruction where the simulator will pause execution upon reaching that instruction. To set a breakpoint, place your cursor over the instruction and hold down “ctrl” while clicking on the mouse button. You will see an option to set (or clear) a breakpoint.