Notes on SPIM

11/5/2011

Table of Contents

SPIM (now in Courier New font :>)

QtSPIM vs pcSPIM

- QtSPIM is newer and can run across all platforms.
  - http://sourceforge.net/projects/spimsimulator/files/

- pcSPIM is older but it is also up-to-date.
  - For Windows users: http://sourceforge.net/projects/spimsimulator/files/
  - For Mac or Linux users: http://spimsimulator.svn.sourceforge.net/viewvc/spimsimulator/
  -

Using pcSPIM on Macor Linux requires compiling the source code. Please refers to the official homepage:

- http://pages.cs.wisc.edu/~larus/spim.html
- http://spimsimulator.sourceforge.net/

You want to step through each line in the simulator, rather than Continue. You need either [dot]asm or [dot]s files.

I recommend using both because I do not know which one Professor Gertner uses. I believe he uses pcSPIM.

Download Source Code

See thread.

<You might want to read and run them in this order>

+ print_string.asm	:	Display a string in the console.
+ sum_demo.asm	:	Adds a number internally and cout the sum.
+ load_byte_demo.asm	:	Demonstrate loading a byte.
+ input_demo.asm	:	Prompt users to enter a string.
+ input_detail_demo.asm	*:*	*Detail comments included.*
++ main_myadd.asm	**:**	**A + 1 = sum and display the result.**
		I rewrote this several time...

SPIM Memory Map

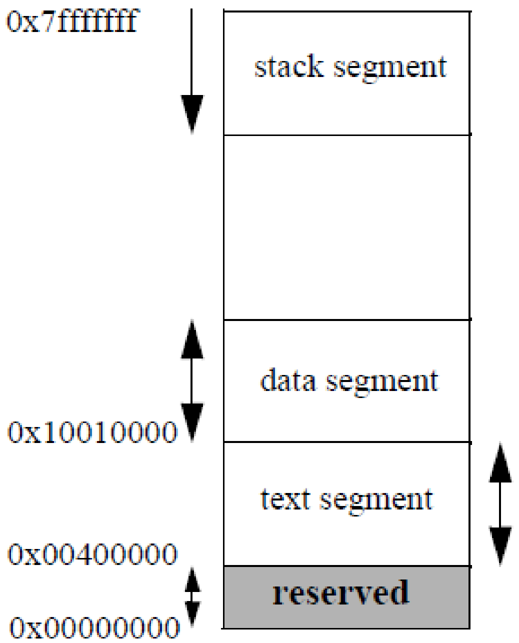

FIGURE 1. SPIM memory map

If you remember from lecture, we discuss the strange **decimal** number 4097:

lui $1, 4097

The load-upper-immediate instruction replaces the upper 16 bits using the second operand (it could be any number or constant) and makes the lower 16 bits all zeros.

For example:

```
lui $8, 0xabcd          # puts 0xabcd 0000 into reg 8
ori $8, $0, 0x123       # sets ls bits; (reg 8 = 0xabcd 0123)
```

More on lui please read this:

http://stackoverflow.com/questions/8018791/mips-load-address-la-doesnt-always-use-register-1

https://groups.google.com/forum/#!topic/csc343fall2011/qEueteCGObc

Pseudoinstructions

We code using pseudoinstructions. For example, jal is just a marco of three jump instructions. When we load word (lw), it is broken down into the following (sample code):

```
lui $1, 4097          ;    === lw $t0, myInt      # load myInt into register $t0
lw $8, 4($1)
```

Keep this in mind. The links above illustrate this. You can observe this by running code on the simulator.

Stack frame Implementation

Caller and callee do different things, and the layout of each stack frame looks like this:

```
[    Parameter n       ]
               ...
[    Parameter 2       ]
[    Parameter 1       ]
[    Return Address    ]
[    Previous EBP      ]
[    Saved Registers   ]
[    Local Variables   ]
```

I steal this from my report.

Caller

- push all the arguments to register $a0 - $a3, and if there are more arguments they will pushed on the stack
- jump using jal (which will link and jump in two clock cycles)

Callee

At the beginning of callee, I think this happens:

- Establish stack frame by subtracting $sp - [frame size].
- Save registers $s0 - $s7 (whichever one is to be used), $fp, and $ra.
- Set frame pointer by adding the stack frame size to $sp. Hence, $fp will point to the first word of the current stack frame.

Before returning, callee will restore, pop, and return control back to caller.

Note on main_myadd.asm

- Create stack frame

I setup a frame for **MAIN** and **MYADD** function calls. Main has this feature because in real life other library functions would be calling **MAIN**. I did this for the sake of illustration.

- Stack frame for MYADD():

I preserved $s0 and $s1 because I want to use them. Compilers assume certain registers will be used (even if they are not) across function calls, so they save them on the stack, and restore the old values later. I

- JAL is a macro.

```
#           $ra <= PC + 4

#           $PC <= address of MYADD

#           nop    # this is a delay-slot
```

$PC is hidden from users so the alternative solution to jal is a bit complicated. If you can just go to my stackoverflow profile. I actually thought of another way, but I don't have time to verify my claim.

The take-away is that jal increment and save address of next instruction (the instruction right below the caller calling function) to $ra, and finally jumps to the entry of the function.

- For Visual Studio:

Please step through the code WITH and WITHOUT optimization. Without optimization you will lose information.... but first run without optimization if you are not comfortable reading disassemble code.

- Reverse Engineering:

Basically you are expected to turn instruction code (in hex) back into human readable instruction code.

I think we should finish chapter two by now. Also visit his website for more exercises.