ECE/CS 250 – Summer 2017 – Prof. Bletsch
Recitation #3 *abbreviated*
Assembly Programming with SPIM

I've consolidated tasks 5+ of recitation 3 to shorten the work while preserving the lesson. If you haven't mastered those concepts, this may help.

Objective: In this recitation, you will learn how to write MIPS assembly programs that run on the SPIM emulator of a MIPS system. Complete as much of this as you can during recitation. If you run out of time, please complete the rest at home.

1. Download SPIM

For the MIPS programming you do in this class, you will use the QtSpim simulator (a newer version of the venerable SPIM simulator) to run and test your assembly programs. QtSpim is a program that simulates the behavior of MIPS32 computers and can run MIPS32 assembly language programs. Download QtSpim and find documentation for it at:

http://sourceforge.net/projects/spimsimulator/files/

2. Run a Short Sample Program on SPIM

A helpful reference is a simple program that I've provided for you on the course site (simple.s). This simple program sums the entries in a list of 9 integers. Download and run this program on SPIM. Try running it to completion first and then run it again using the single-step feature to walk through each instruction one at a time. Look at how the PC, register values, and memory values change as a result of each instruction. You're going to want to get good at stepping through programs, because this is largely how you'll debug your own programs.

3. Write a Very Simple MIPS Program

Write a MIPS program that prints out the integers from 0 to 10. Write this program as a loop (i.e., don't just declare a string "0, 1, 2, etc." and print that string).

4. Write a Somewhat Less Simple MIPS Program

Write a MIPS program that reads in the user's name and the user's age (from the console) and then prints out the year in which that person will turn 50 years old. For example, if the user types in "Dan 40", then the program should print out "Dan will turn 50 years old in 2026." You may assume that the user has already celebrated their birthday this year.

Note: More advanced MIPS material is below. It is unclear at this time if we'll cover all the stuff necessary to tackle it by this recitation, so we may revisit this content in lecture or possibly out of class.

5. Calling a Procedure, Passing Args and Return Values

Write a short MIPS program with a main function that calls (using "jal") another function called foo. The foo function takes two arguments (both are ints) and returns one value (also an int). You must follow conventions for arguments and return values: you must pass the arguments through $a0 and $a1 registers and you must return the value from foo in $v0. For now, let foo simply compute the sum of the two arguments and return that result. In main, please set $a0=1 and $a1=2.

6. Saving the Caller-Saved Registers, Using the Stack

Copy your work from Task 5 into a new file. Now modify main so that it saves the caller-saved $t registers before calling foo and then restores them after foo returns. You must modify main to use two $t registers ($t0 and $t1) to initially hold the values it's going to pass to foo (but main still must pass them through the $a registers, so they must be copied from $t to $a). (Instead of setting $a0=1 and $a1=2, set $t0=1 and $t1=2, then copy from the $t regs to the $a regs.) After foo returns to main, main should then compute the result from foo plus the sum of these two $t registers into $t5. (main may not use the $a registers for this purpose!) To make room for main to save these $t registers, main must create space on the stack. You will move $sp to make room for these two $t regs, copy them there before calling foo, and then copy them back into the $t registers after foo returns. In summary, your code should look like the following (where the parentheticals show the instructions needed):

```
main:
    # reserve a stack frame for 2 words (t0,t1) (subu or similar)
    #  (note: we're ignoring the need to save $ra for this exercise)
    # set $t0=1 and $t1=2 (li)
    # set $a0=$t0 and $a1=$t1 (move)
    # save t registers to stack (sw)
    # call foo (jal)
    # restore t registers from stack (lw)
    # compute v0+t0+t1 into t5 (add)

foo:
    # add a0+a1, store and store result in v0 (add)
    # return (jr)
```

Once the program works and you verify you got the right value in t5 at the end of main, let's break it. Modify foo such that it sets all of the $t registers to zero right before you compute v0. Comment out the lines of code in main that save and restore the $t registers (the blue steps above). What happens? Do you still get a correct program result in t5?

7. Saving the Callee-Saved Registers

Copy your work from Task 6 into a new file. Modify main such that, before it calls foo, it sets two of the callee-saved $s registers ($s0 and $s1) to the values 5 and 6. After foo returns to main, main should take the result from foo and add it to $s0 and $s1.

Now modify foo such that it saves the callee-saved $s registers when it begins and restores them just before returning. You will modify foo to move $sp, etc., just like you did with the caller-saved registers in main in Task 3. In summary, your code should look like the following (where the parentheticals show the instructions needed):

```
main:
    # set $s0=1 and $s1=2 (li)
    # set $a0=$s0 and $a1=$s1 (move)
    # call foo (jal)
    # compute v0+s0+s1 into t5 (add)

foo:
    # reserve a stack frame for 2 words (s0,s1) (subu or similar)
    # save s registers to stack (sw)
    # add a0+a1, store and store result in v0 (add)
    # restore s registers from stack (lw)
    # return (jr)
```

As before, once the program works and you verify you got the right value in t5 at the end of main, let's break it. Modify foo such that it sets all of the $s registers to zero right before you compute v0. Comment out the lines of code in foo that save and restore the $s registers (the blue steps above). What happens? Do you still get a correct program result in t5?

8. Thought Exercise

Why do we need these register usage conventions? Couldn't the programmer just manage all of the registers on his/her own without these conventions? If we know that foo won't modify any $t registers, can't we skip saving/restoring the $t registers in main?

9. A Little Bit of Recursion

Copy your work from Task 7 into a new file. Modify foo such that it uses $t2 to hold the sum of its arguments ($a0 and $a1). If $t2 is greater than 10, then it simply returns that sum (in $v0). Else, it calls itself with its arguments each incremented by 1 (i.e., $a0 +1, $a1+1).

Now foo is both a callee AND a caller. You'll have to modify foo to save its caller-saved register ($t2) and $ra on the stack.