ECE/CS 250 – Prof. Bletsch
Recitation #3
Assembly Programming with SPIM

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.

PART 1

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For the MIPS programming you do in this class, you will use the QtSpim simulator (a newer version of the venerable SPIM simulator) to run and test your assembly programs.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. Computing options:

- **Docker container**: Already has QtSpim pre-installed and ready to go.
- **Local tools**: You can download QtSpim and find documentation for it here: http://sourceforge.net/projects/spimsimulator/files/
- **Login.oit.duke.edu**: No graphical QtSpim, but has the software necessary to run Homework 2's automated test tool.

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.

NOTE: If you'd like a guided intro to SPIM, you can consult the video "SPIM intro" on the course site.

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. Be Your Own Compiler

Consider the following C program:

```
#include<stdio.h>
int main() {
    int i=1;
    while (i<=10) {
        int sq = i*i;
        printf("%d %d\n",i,sq);
        i++;
    }
    return 0;
}
```

Going line-by-line, convert it to MIPS assembly. Below is a skeleton for your code – fill in code for the boxes. You can copy/paste this into a text editor to get started. **NOTE: *After* you've gotten this (asking for help as needed), you can consult the video "C to MIPS 1" on the course site for a narrated solution.**

```
# code section
.text

# ======================== start of function: main
# int main() {
main:

# int i=1;
li $t0, 1

# while (i<=10) {
_loop:
```

```
                                # if t0 is above 10, branch out of loop
```

```
# int sq = i*i;
```

```
                                # compute t0*t0 and put result in t1
```

```
# printf("%d %d\n",i,sq); # we'll do this in four stages
#   (print i)
```

```
#   (print space)
```

```
# (print sq)



#  (print newline)



# i++;
                                                    # increment t0

# } // end of loop
                                  # branch to top of loop unconditionally
_endloop:

# return 0;



# ======================== end of function: main

# data section
.data
# a literal space string for printing
str_space: .asciiz " "
# a literal newline string for printing
str_newline: .asciiz "\n"
```

Get your code working in QtSpim. **Homework 2 will ask you to write the same programs as Homework 1, except in MIPS. Starting with your C code and mentally "compiling" it to MIPS is a good strategy to get started!**

5. Again with the strlen

In Recitation 2, you wrote your own version of `strlen()` called `my_strlen()`. Following a methodology similar to the above, convert this function to MIPS. A test harness has been written for you here – add your code to it and test it out. *NOTE: Because the newline character is included in the input string, the number printed will be 1 higher than you might expect.*

END OF PART 1

If you're in recitation now, please continue and dig into PART 2 until recitation ends.

PART 2

NOTE: If you're getting stuck or lost with regard to stack discipline, you can check out the videos "C to MIPS" 2 and 3 on the course site for a narrated exercise in doing Task 4 with function calls.

6. 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.

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

Copy your work from Task 6 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?

8. Saving the Callee-Saved Registers

Copy your work from Task 7 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 1 and 2. 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. 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?

9. 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?

10. A Little Bit of Recursion

Copy your work from Task 8 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), and returns the result of this recursion plus $t2. For this to work, $t2 must be properly saved before the call and restored after, else you get the wrong answer.

Formally, this means that foo is defined as:

foo(a0,a1) =	a0+a1	when a0+a1>10,
	foo(a0+1,a1+1)+a0+a1	otherwise

A correct solution should yield foo(1,2)=35.

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.

ALL DONE?

Nice! Don't head out, though. Work on the current homework and talk to the TAs for help. You can leave if your homework tester shows all passing and you've turned in all the written & code materials.