SYSTEMS I — LAB 5
An introduction to MIPS assembly

In today's lab, we will do some assembly programming. In particular, we're going to use a program that simulates a MIPS CPU, known as **SPIM**. You're going to get set up with SPIM, make sure you can use it to run a program, and then do some assembly coding of your own.

1 Setup to use the MIPS simulator

Logging in: In order to use the MIPS simulator, you must first logon to `romulus.amherst.edu` or `remus.amherst.edu`, which is a UNIX (Linux) machine. To login using *Xming*, software that allows you to login graphically to these servers, follow the Windows *Xming* instructions that describe how to use this software on the Windows machines in SMudd 014. Notice that this page also describes how to install and use *Xming* on your Windows machine. If you have a Mac, follow the Mac *Xming* instructions.

One-time setup: Once you have started *Xming*, you will have a *terminal window*, within which there is a *shell prompt*—that is, a prompt at which you can type commands. At the prompt, type the following two lines, being sure to include the period that begins the filename on the second line, and also being sure **not to type the dollar sign (`$`)**, since that is my representation of the shell prompt itself for these instructions:

```
$ ~sfkaplan/public/cs16/init
$ source .cshrc
```

If you get an error message in response to either of these commands, ask for help. (No news is good news—if you enter the command and the prompt then returns with no intervening messages, then the command worked.) Note that **you should not use these commands again**, or else you might lose some of your work.

Checking your files: If all went well in the previous steps, you should be ready to go. Note that in the future, you simply need to login, bring up a terminal window, and begin work.

When you first login, you will be working in your *home directory*—the UNIX analog of your *My Documents* folder. The following command (using the lower-case `L`, not the numeral `1`) . . .

```
$ ls -l
```

. . . will list the files and subdirectories (subfolders) in your home directory. In should contain two new files:

```
add-two-numbers.s
formula.s
```

2 Try running a simple program

Follow these steps to try running a program with SPIM and ensure that it works:

1. First, run *Emacs*, a programming text editor, to examine the `add-two-numbers.s` file, like so:

   ```
   emacs add-two-numbers.s &
   ```

   We will discuss what you see here. There is a boilerplate preface, and then the label `__start`, which is the marker that will tell the simulator where the instructions for this program begin. There is also a strange pair of instructions that follow the comment, "Exit the program." We will discuss what these are and how they work.

2. Now we're going to use SPIM itself. Start SPIM with the following command, where the ampersand (`&`) is necessary to run the program in the *background*—that is, SPIM continues to run while you are allowed to type more commands in your terminal window:

   ```
   xspim &
   ```

3. You should see a new window appear that will have lots of information about the registers (labeled both by their numbers, `R0` through `R31`, and their symbolic names, e.g., `$s0`). Among other things, there will also be some buttons in the middle of this window that we'll use to control the simulator.

4. Begin by loading a program. Click on the **load** button. It will bring up a window, in which you can type `add-two-numbers.s` and press `Enter`.

5. You should see the instructions for that program appear in the middle of the window, with the first instruction highlighted. **Notice that some of the instructions aren't quite what you saw in the *Emacs* window.** The `li` instruction has been replaced with `ori`. We will discuss that `li` is a *pseudo-instruction*. Also notice that the registers are not given by symbolic names (like `$s0`) but instead by register number (like **$16** for register number 16). Finally note that data appears in *hexidecimal*, also known as base 16. We'll discuss why that representation has been chosen, and how to read it. Specifically, a hexidecimal digit is one of 16 values, from 0 to 9, and then from a to f. Thus, here is a table that shows some values in decimal, binary, and hexidecimal:

decimal	binary	hexidecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	a
11	1011	b
12	1100	c
13	1101	d
14	1110	e
15	1111	f
16	10000	10
17	10001	11
18	10010	12
19	10011	13
20	10100	14
21	10101	15
22	10110	16
23	10111	17
24	11000	18
25	11001	19
26	11010	1a
27	11011	1b
28	11100	1c
29	11101	1d
30	11110	1e
31	11111	1f
32	100000	20

6. Now click on **step**, so that we can run the program one instruction at a time, and see what happens. You will get a little window that asks for some more information—you can just use the default values. Each time you click the **step** button **within that little window**, the current instruction will be executed, and the simulator will move to the next instruction. If the instruction was supposed to modify a register somehow, look at the register values and ensure that the value has changed as you expected.

7. When you have stepped through this whole program, be sure that register `$s0` has exactly the value that you thought it would. You can click the **reload** button, which will present (so long as you hold down the mouse button) a pull-down menu of one item (**assembly**

program). Select that to reload and re-set the program. Note that you can run the whole program, without stepping, by clicking the **run** button.

3 Make a new, modified program

Now that you have seen what a MIPS assembly program looks like, and you've seen how to run one within SPIM, try writing your own small calculation program. Open `formula.s` with *Emacs*, and complete the program that was started there.

Specifically, after the `__start` label, insert code to do what the comments suggest. Specifically, write instructions that will load the given constants into registers, and then write instructions that carry out the given (simple) formula. The special instructions that end your program are already provided.

Once written, **test your program** in SPIM. Move through the program, step by step, and be sure that the registers change value with each step as you expected.

4 How to submit your work

We will be using the `cs16-submit` command to turn in programming work. Specifically, you should submit your completed `formula.s` like so:

```
cs16-submit lab-5 formula.s
```

This assignment is due on Friday, October 10, at 11:00 am