SYSTEMS I — LAB 6
Conditional statements and loops in assembly

We're going to continue with MIPS assembly and our `spim` simulator. Specifically, we want to write assembly programs that use *if-then-else* and loop structures. Moreover, we'll add a few new tools to our arsenal: the ability to load and store values from main memory, and the ability to produce some text output in a window.

1 Printing values to a window

Recall that to end a program, we use the following instructions:

```
li $v0, 10
syscall
```

We will discuss how the `syscall` instruction is handled later, when we get to operating systems material. In the meantime, what you need to know is that when the `syscall` instruction is performed, the CPU jumps into the mini-operating-system that is part of the `spim` simulator; that is, it performs a *system call*. The mini-OS examines the integer in `$v0`, treating it as a *request code*—a number that indicates which operation the program is requesting the OS to perform. `10` is the request code for terminating (ending) the program.

In order to display output in a window, we are going to use this same system-call mechanism.

1.1 Integers

Assume that you have an integer value in `$t5` that you want to print in a window. To do so, use the following instructions:

```
li $v0, 1
move $a0, $t5
syscall
```

First, notice that `1` is the request code for printing an integer value. Second, the `move` instruction is new, and it is used for copying values from one register to another—here, we copy the value from `$t5` to `$a0`. Third, note that the operating system will assume that the integer you want to print will be stored in `$a0`, which is why we needed to copy the value from `$t5`.

1.2 Text

Printing text is a little bit different, since we don't know how to store text in registers (and in fact, we will not do so). Assume that you want to print the message, "The mid-term was too hard!" To do so, you must first add some lines to the end of your assembly program like so:

```
          .data
TestMsg:  .asciiz "The mid-term was too hard!\n"
```

First, the `.data` marker needs to appear **only once** in your program, no matter how many messages follow. It is a marker for the assembler so that it knows that constant values are about to follow, and not instructions.

Second, the marker `.asciiz` tells the assembler that it is about to see a text string, enclosed in quotes. Note that the trailing `\n` is the *newline character*, which is added to advance the text cursor to the next line in the printed window. Finally, the label `TestMsg:` will be converted, by the assembler, into a main memory location wherever that label appears in the instructions.

The result of this added code is that this string will be loaded into main memory along with your program at some pre-defined address. That way, the program can use the string by referring to its label, which becomes its main memory address. How do you use the string? Like this:

```
li $v0, 4
la $a0, TestMsg
syscall
```

The request code for printing text is `4`. The `la` instruction **l**oads an **a**ddress—here, the main memory address at which the string labeled `TestMsg` will reside is loaded into register `$a0`, which is where the mini-OS will look for it. Finally, `syscall` again initiates the system call, causing the printing to occur.

2 Using main memory for data

So far, we have assumed that main memory holds only the instructions that compose a program. However, main memory can also store data values—the same values that we've been storing in registers. Since the total memory space of the register file is small, and the main memory is quite large, we want to be able to move values from the register file into main memory and back again. It is much like going to the library, selecting the books you need from the stacks, and putting them on a desk at which you will work. The books on the desk are ones you can immediately use, much like values in the register file are ones that can be immediately operated on by instructions. The books in the stacks are ones that you must bring to the desk before you use them, just like values in main memory must be copied into the register file before being operated upon. If the desk gets full, you must put a book back in the stacks before taking another one; if your register file is full of meaningful values, you must copy some into main memory for safe-keeping before using those registers to hold different values.

2.1 The stack

Our use of main memory for data will be simple. Specifically, for each program, there are (at least) two regions of memory: the *text*, which stores the machine code, and the *stack*. We will need to examine the stack in more detail next week, but for today's purposes, we need only know that it is a separate region of memory in which we can store some data.

In particular, once your program is running, the *stack pointer*, which indicates the address at which top of the stack resides in main memory, will be stored in the register `$sp`. Notice that the stack `grows downward`—that is, if we want to make more space on the stack, we need to **subtract** some number of bytes from `$sp`, providing us space to put data values.

For example, consider that when your program starts, `$sp = 1000` (in decimal, for simplicity). Also assume that you want to store four different register values in main memory. To do so, we first reduce `$sp` by 16, to `984`. That leaves us 16 bytes, or 4 words of space, for our 4 values, where the first value will reside at address 984, the second at 988, the third at 992, and the last at 996. All we need now are the instructions that allow us to copy values from a register to the addresses and back again.

2.2 Storing

First assume that you have two values in `$s3` and `$a1` that you want to store on the stack. To do so, first we need to make space on the stack:

```
addi $sp, $sp, -8
```

Whatever the stack pointer's value was, it is now decreased by 8, giving us space for two words, where that space starts at the address contained in `$sp`. Now, we can *store* the values of our registers at those locations:

```
sw $s3, 0($sp)
sw $a1, 4($sp)
```

The `sw` (**s**tore **w**ord) instruction takes the value from a given register and stores it at some main memory location. Here, the first `sw` takes the value from `$s3` and copies it to main memory at the address `$sp + 0`. The second `sw` copies the value in `$a1` to main memory address `$sp + 4`. That is, the number before the opening parenthesis is the *offset*—the number of bytes added to address contained in the register specified within the parentheses.

2.3 Loading

The copy a value back into a register from main memory, we *load* it with the `lw` instruction. For example, let's say we want to the load the value at `$sp + 12` into register `$t2`:

```
lw $t2, 12($sp)
```

3 Your assignment

There are three programs that you should write (or modify). We'll address each in turn.

3.1 Conditional statements

For the first, you will a simple program that places two values into registers, compares those values, and then prints one of two messages depending on the result of the comparison. In Java-like pseudocode, here's what we want:

```
$t0 = 10
$t1 = 5
if ($t0 < $t1) {
  print "The then branch!\n"
} else {
  print "The else branch!\n"
}
```

The program should be written in a file named `conditional.s`, and it should begin like this:

```
         .text
         .globl __start

__start:
```

After the `__start:` label is where your first instruction should go. Don't forget to end the program with a system call using request code 10. Test your code using `spim`. Vary the values that are loaded into `$t0` and `$t1` to be sure that the program works.

3.2 Loops

Start by copying the following file from my directory, like so:

```
$ cp ~sfkaplan/public/cs16/find-max.s .
```

This program begins by creating main memory space in the stack region for an array of integers (five of them). It then assigns values into each of the array's entries by loading a constant into `$t0` and then storing that value from `$t0` into the array. The length of the array is stored in `$s1`.

Your goal is to then write a loop that finds the maximum value of the entries in the array. When the loop is done, `$s0` should contain a maximum value from the array. It is a contrived problem in that you must pretend that you don't know the values of the array, nor its length, in writing this loop; you should depend only on `$sp` and `$s1` for the location and length of this array of integers. In other words, I should be able to change the length of the array and its contents, and your loop should still work.

3.3 Triangles

Start a new program from scratch in a file named `triangle.s`. This program prints a pattern. Specifically, if the program is set to print a pattern of size 6, it should print the follow:

```
Size = 6

*
**
***
****
*****
******
```

This program must begin by assigning the *size* value into register `$t0`—a value that I should be able to change to any non-negative integer when I use your code. It should then print the pattern described above and exit.

4 How to submit your work

We will be using the `cs16-submit` command to turn in programming work. Specifically:

```
cs16-submit lab-6 conditional.s find-max.s triangle.s
```

This assignment is due on Friday, October 31, at 11:00 am