Lab 1: Introduction to MARS

Submission timestamps will be checked and enforced strictly by the CourseWeb; **late submissions will not be accepted**. Check the due date of this lab on the CourseWeb. Remember that, per the course syllabus, if you are not marked by your recitation instructor as having attended a recitation, your score will be cut in half.

Launching MARS

This semester we will be working with a MIPS simulator called the MIPS Assembler and Runtime Simulator, or "MARS" for short. MARS runs on Java, so be sure you have the Java Runtime Environment (JRE) properly installed on your system before attempting to run MARS. **Download a copy of MARS version 4.5 (`mars4_5.zip`) from the CourseWeb under "Labs/Recitations" → "Lab 1"**.

The zip file contains a directory called `mars4_5`. To run the basic simulator, follow these steps:

1. Extract the zip to produce the `mars4_5` directory.
2. Open a command prompt.
3. Using the command prompt, enter the `mars4_5` directory you just extracted.
4. Run `java Mars`. (If you get a `NoClassDefFoundError`, make sure you used a capital `M`.)

Once MARS is running you will see many buttons, menus, and symbols. Do not worry if you do not understand what they do or mean. We will learn many of them for this course; others will not need.

Writing a simple MIPS assembly program

To begin a new MIPS assembly program:

1. Click on the "Edit" tab at the top left. In this pane, you will see your assembly language program.
2. Go to "File" → "New" to start a new program.

Let's start with a simple program. Begin the program by typing:

```
.text
```

This says that the lines to follow are program instructions (and not, e.g., data). Let's do the following calculation:

```
    # $t9 = 179 + (-293) + 561
```

The `#` symbol begins a comment in MIPS assembly language, so you can go ahead and type this into your program if you would like. First, let's start by ensuring that the register `$t9` starts with the value 0:

```
    addi $t9, $zero, 0
```

The instruction above says to take what is stored in the register `$zero`, add 0 to it, and put the result into the register `$t9`. Note that the register `$zero` always contains 0; so this put $0 + 0$ into the register `$t9`, which is 0. A similar instruction that gives the same result is

```
    add  $t9, $zero, $zero
```

The different between these two instructions is that for `addi` instruction, the third item must be an immediate value but for the `add` instruction, it must be a register.
Next, let's add 179 to our total, by taking the value in the register `$t9`, adding 179, and storing the result back into the register `$t9`:

```
    addi $t9, $t9, 179
```

Let's try running what we have so far!

3. Before you can go further, MARS needs you to save the file. Let's name the file `lab01.asm`.
4. Now, go to the "Run" menu and select "Assemble" (alternatively, click the wrench icon or use the keyboard shortcut F3)

On the "Execute" tab, the "Text Segment" window shows you:

- Address: The location where this instruction is stored in memory.
- Code: The machine code of the instruction, which is 32 bits wide.
- Basic: Not to helpful at this point; we will come back to this later.
- Source: The original instruction you typed in.

The "Data Segment" window in the middle shows you the contents of the part of memory where data is stored. We have not put any data in memory, so the values are all 0.

The bottom window gives messages from the simulator. Any error messages will be displayed here.

Let's continue:

5. Now, click on teh "Go" symbol (the green forward arrow by itself, or keyboard shortcut F5) to "execute" the machine code (technically, "simulate" it).
6. Look at the value of register `$t9` in the "Register" panel on the right.

The value of the register `$t9` should be 0x000000b3 which is 179 in hexadecimal. T0 see values of registers in decimal, uncheck the "Hexadecimal Values". Now we want to add the value -293 to our running sum stored in the register `$t9`, so we write another instruction:

```
    addi $t9, $t9, -293
```

And lastly, we want to add 561:

```
    addi $t9, $t9, 561
```

So, your program should look like the following:

```
.text
    addi $t9, $t9, 0
    addi $t9, $t9, 179
    addi $t9, $t9, -293
    addi $t9, $t9, 561
```

Assembly the program again, and run it. You should see that the value stored in the register `$t9` is 0x000001bf which is 447 in decimal.

Installing and using a MARS tool

You may have noticed a "Tool" menu when you were looking at MARS earlier. In addition to simulating the standard features of a MIPS processor, MARS supports the addition of tools which can be used to help us interact with our data and computations in cool and interesting ways. Some of these are already built into MARS, but we will be adding a simple tool to help us further explore how numbers are stored in registers.

First **download `PositiveZeroNegative.zip` from the Lab 1 section of the CourseWeb**. This zip file contains three Java class files. Extract these file to the `mars4_5/mars/tools` directory inside your MARS installation, then launch MARS again with the command `java Mars`. If you have extracted the class files into the correct directory, you should see "Smiley Face (Register) V0.1" as an option under the "Tools" menu. Selecting this tool opens a small window containing a yellow cartoon face.

The smiley Face tool is a very rudimentary tool: The face smiles when the value stored in the register `$t9` is a positive value, frowns when `$t9` is negative, and maintains a neutral expression when `$t9` is equal to zero.

Open the program you created in previous section. Let's step through the program one instruction at a time using the "Step" tool (the green forward arrow with the "1", or F7 on your keyboard). while keeping the Smiley Face tool open.

Since the first instruction simply ensures that the register `$t9` starts out as 0, the face keeps its neutral expression when we execute it. If we step through the second instruction (adding 179), we see that the value in `$t9` becomes positive, so the face smiles. Executing the third instruction (adding -293) causes our total in `$t9` to become negative, so the face frowns. Lastly, adding 561 brings us back to a positive number, and the face is pleased once more.

But even though we want to see our program execute more slowly, manually stepping through our program like this is tiresome! MARS has a feature for that, too! There is a slider at the top which says "Run speed at max (no interaction)". Drag this slider to the left to change the speed at which instructions are run. If you go slow enough (1 or 2 instructions per second), you can see the instructions execute individually. This can become vary helpful in larger programs.

This is pretty cool, but we would really like to improve upon this program in two ways:

- We want our program to loop through the various faces it encounters, rather than just going through them once.
- We want our program to automatically advance (with a short delay!) between each face, so we do not have to manually step through the program one instruction at a time or resort to changing the executing speed.

Let's handle each of these in turn.

To loop our program, we will use another one of the many instructions from the MIPS instruction set. You can actually see a full list of instructions by going to "Help" → "Help" and looking at the "Basic Instructions" tab. We will only cover the essentials in this course, but here is your proof that there is a whole lot more MIPS can do.

If you scroll down to the "j"s, you will see several instruction related to jumping. Since we want our program to infinitely loop, we want the unconditional jump, or the `j` instruction. We will cover some of the others later.

So, let's add a `j` instruction to the end of our program. You instruction should look something like:

```
    j   top_of_loop
```

You may have noticed, as you were typing, that a helpful tip appeared giving you information about the format of this instruction. The `j` instruction requires a target, which is what `top_of_loop` means above. The `j` instruction says to jump unconditionally (this is, always), and the `top_of_loop` says that the jump should take the program to the instruction labeled `top_of_loop`.

But we do not have an instruction labeled `top_of_loop`. Let's fix that. A label is simply an arbitrary name followed by a colon. Label names should describe their purpose. So, let's add that label to the first instruction of our program by changing it to read:

```
top_of_loop: addi $t9, $zero, 0
```

Now, if we reassemble our program and step thorough it, we can see that executing the fifth instruction (`j`) brings the program back up to our first instruction. Note that an instruction at a label does not have to be on the same line as the label. Sometimes labels may be long and you want to indent your code neatly as shown below:

```
.text

top_of_loop:
    addi $t9, $t9, 0
    addi $t9, $t9, 179
    addi $t9, $t9, -293
    addi $t9, $t9, 561
    j    top_of_loop
```

Now, let's handle getting the program to pause automatically between faces. MIPS assembly does not have the ability to do this directly. For example, there is no "sleep" instruction for you to use. Fortunately, MARS supports system calls (or `syscall`). Syscalls let you program perform more complex tasks (or "services") for us, such as sleeping, asking for user input, and generating random numbers. In real computers, these tasks are carried out by the operating system. In our case, the MARS simulator will perform the task for us.

The "Help" box in MARS also has a tab that is all about syscalls. It is very long and detailed, so do not read all of it. Instead, just read the first few short paragraphs, and stop before you get to the "Table of Available Services". **Pay particular attention to the order of the steps for issuing a syscall, as well as the fact that the service number must be loaded into the register `$v0` before issuing the syscall**. We will be using a lot of syscalls as we continue in this course, so this is important.

Now take a quick look through the table. It is quite big. MARS supports dozens of different syscalls. For now, we are only interested in syscall 32 (sleep).

What would a 500-millisecond sleep look like in MIPS? First we have to load the appropriate service number into `$v0` (that is, 32). Then, the syscall table tells us that we have to load the argument to this syscall, which is the length of the sleep in milliseconds, into `$a0`. Then we issue the `syscall` instruction. Altogether, this looks something like this:

```
addi $v0, $zero, 32        # Syscall 32: Sleep
addi $a0, $zero, 500       # Time to sleep (500 ms)
syscall                    # Perform the syscall (sleep for 500 ms)
```

Together, these three instructions will cause our MIPS program to hand off to MARS (which acts as our operating system). MARS will then wait 500 milliseconds before returning control to our program for it to resume normal execution.

Since we want to pause after each of our four original instructions, we need to add these three sleep related instructions after each one. **Go ahead and do so now**. Now, when we reassemble our program, we can simply run it with the "Run" tool (F5) at max speed and we see the face animate through its whole range of emotions over and over again.

Submission

Submit your `lab01.asm` file via CourseWeb before the due date stated on the CourseWeb.