

# CS447 Lab #1

---

## Introduction


This course will use the MIPS Assembler and Runtime Simulator (MARS). See <http://courses.missouristate.edu/KenVollmar/mars/> for information and a link to the download site.

MARS is already installed in the department computer labs. Please ask for help if you have any trouble locating or running the program.

*Note: You do not need to formally submit anything for Lab #1. The work that you do during recitation will be sufficient to receive credit.*

## Part 1: Assemble, run, and step through a MIPS program in MARS.


In this first exercise, you will make use of a MIPS program that has already been created for you. From the course website (<http://pitt.edu/~jfb42/cs0447/>) download the `hello_world.asm` file. If you're working on one of the lab machines, note that any files you save to the hard drive will not remain after you log out, so be sure to use some other storage for any files you wish to keep.

Click on the MARS icon to start the program. From the **File** menu, choose **Open**, or click the  icon in the toolbar. Locate and open the `hello_world.asm` program.

You should now see the MIPS instructions in the main window, under the "Edit" tab. Notice that nothing yet appears under the "Execute" tab. This means we still need to assemble the program.



Click the Assemble icon (or choose **Assemble** from the **Run** menu). Assuming that the program assembled successfully, the screen should automatically switch to the **Execute** tab, and at the bottom of the screen, MARS will report that the program assembled successfully. Under **Text Segment**, you will see the instructions that MARS will actually run ("Basic" column). These instructions may be different from those you see in the editing window, because any pseudoinstructions will be broken down into basic instructions (see the MARS **Help** menu to see lists of the basic and pseudo-instructions).

Now that you've assembled the program, you can run it. Click the run icon  in the toolbar. Assuming that all went well, the Run I/O output should look like this:

**Hello, world!**

-- program is finished running --



If you want to run the program again, you'll need to use the reset icon .

Sometimes you may want to run the program at a slow speed. Try repositioning the pointer on



the **Run Speed Slider** and run the program again. Note that the program not only runs slower, but you can now see that the individual instructions are being highlighted as the program proceeds.

**Reset** the program again, and go back to the **Edit** tab. The following will guide you through the program step by step, with some review of the different instructions involved. You may want to go back and forth between the **Edit** and **Execute** tabs.

Under the Edit tab, on lines 1-2 we see:

**.data**

**msg: .asciiz "Hello, world!"**

If you do not remember what the **.data** and **.asciiz** directives mean, go to the **Help** menu and check out the **Directives** tab. Quickly summarized, these lines indicate that the ASCII representation of "Hello, world!" should be placed in memory, with the end of the string marked by the NUL character. Under the **Execute** tab, in the **Data Segment** section, you will notice that nonzero hexadecimal values appear in the first row of memory locations. This is where the string is stored.

Returning to the **Edit** tab, you'll see the following on lines 3-12:

**.text**

**# Print the message**

**la \$a0, msg # Put the message address in a0**

**li \$v0, 4 # Put the "print string" syscall code in v0**

**syscall # Call it!**

**# Terminate the program**

**li \$v0, 10 # Put the "exit" syscall code in v0**

**syscall # Call it!**

Now switch back to the **Execute** tab, and view the **Text Segment** field. The Source column shows what was present under the **Edit** tab, and the Basic column shows the instructions that MARS will actually run. Note that this program replaces three instructions: an **la** instruction and two **li** instructions. Note in particular the replacement of **la** with **lui** and **ori**. See the 4<sup>th</sup> set of course slides for more information on why this must be done.

Now let's try running **just the lui and ori instructions** that replaced **la**. Recall that the load address instruction is being used to place in register **\$a0** the memory address corresponding to the contents associated with **msg**. To run the first two instructions only, use the single step



icon in the toolbar. Click this twice, and you'll see that the end result is a change in the value of register **\$a0**: The value now reflects the memory address where the Hello, World! string is located. (Note that MARS also provides a similar icon that allows you to go backwards in single steps).

Under **Text Segment**, you will now see a yellow highlight over the first **li** instruction (and the **addiu** instruction that replaced it). This is the next instruction to be run. Its purpose is to load the immediate value **4** into register **\$v0**. As stated in the comments, we are preparing to invoke a system call. If you open the MARS **Help**, and view the **Syscall** tab, you will see a table explaining how the various **Syscalls** work. By the Service "print string," you'll see that the code **4** must be placed in **\$v0**, and **\$a0** must contain the address of the string you wish to print. The **la** instruction took care of the **\$a0** part, and the first **li** instruction will take care of the **\$v0** part.

Use the single step icon to run the **li** instruction (technically, **addiu** when executed), and note the change in the value of **\$v0**.

Now, run the **syscall** instruction. "Hello, world!" is printed to the I/O window.

Finally, run the next **li** and **syscall** instructions. As stated in the comments, the result is a syscall that gracefully exits the program.

## Part 2: Modify the hello\_world program to subtract one user-supplied integer from another.

In this second part, you will change the original hello\_world program, so you may wish to save the current file to a different name.

In this program, we would like to request that the user input two integers, and then we will output the difference between the first and second values. First, let's replace the text in the **.data** section. Change line 2 so that the stored string is no longer "Hello, World!", but now "Please enter an integer:". If you assemble and run the program, you'll see that the I/O output has changed.

Next, we'll need to add a syscall to read the first integer. Look up the "read integer" syscall in the **Help** section. Note that this syscall *does not* require that any value be placed into **\$a0**.

However, we will need to change the contents of **\$v0**. So, below the first syscall (the one that prints the string), add two new instructions:

- The first instruction should update the value of **\$v0** to the syscall code specified in the table for “read integer.”
- The second instruction is a **syscall**.

If you try running the program at this point, you’ll see that the user is prompted for an integer (in the I/O window).

Reading through the syscall table, you’ll see that the integer that the user enters will be stored in **\$v0**. Because we want to read a second value from the user, we will need to either copy or move **\$v0** to a temporary register. Therefore, after the two new lines that implemented your new syscall, write a new line that places the value of **\$v0** into temporary register **\$s0**. There are multiple ways you can achieve this; feel free to use any way that works.

If you run the program at this point, you’ll see that the user’s integer ends up in **\$s0**.

Now we can add another syscall, to get the second integer. Here, you can just repeat the first two syscalls, for printing a string and reading the integer.

Now that we have our two integers, we’d like to subtract the second value from the first value. Note that our first integer will be stored in **\$s0**, and the second integer will be stored in **\$v0**. Use a **sub** instruction to perform the subtraction. The **sub** instruction will require a destination register, and a particularly good choice for this destination would be the register that stores the integer required for the “print integer” syscall (see **Help**).

Now that you’ve performed the subtraction, and stored the result in the appropriate register, you can prepare for the print integer syscall. Update **\$v0** to the appropriate value, and then add a syscall instruction.

At this point, you should have a functional program. The output might look something like this:

**Please enter an integer:20**

**Please enter an integer:5**

**15**

**-- program is finished running --**