

Foothill College
Computer Science Department
CS10 Computer Architecture and Organization

Lab # 2

MARS MIPS Simulator

1 Introduction to MIPS

In this lab, we will examine a representative computer architecture, that of the MIPS CPU.

2 The MIPS CPU: a Programmer's Perspective

The first [MIPS CPU](#) was released in 1981 and has since grown into a family of 32-bit and 64-bit CPUs, each one improving performance and adding functionality over the previous releases. The MIPS architecture is a very clean and elegant RISC architecture, which is one of the main reasons why we are using it as our representative hardware architecture. In its heyday (the 1990s), the MIPS family was one of the most popular [RISC](#) chips to be used. Today, MIPS implementations are primarily used in embedded systems such as Windows CE devices, home routers and video game consoles such as the Sony PlayStation 2 and PlayStation Portable.



MIPS R3000 CPU

In the lectures, we will look at the internal design of a RISC CPU, and the MIPS architecture. In the labs, we will concentrate on how to program the MIPS CPU. To do that, we need to see what functions and features it exposes to the programmer.

2.1 Data Types

MIPS instructions are all 32 bits. There are four basic data sizes supported by the CPU:

- a byte is 8 bits in size. Data items such as ASCII characters are stored in a byte.
- a halfword is 16 bits (i.e. 2 bytes) in size. Data items such as Java *short* are stored in halfwords.
- a word is 32 bits (i.e. 4 bytes) in size. Data items such as Java *int* are stored in words.
- a doubleword is 64 bits (i.e. 8 bytes) in size.

More complex data structures, such as strings, C structs, Java objects etc. are stored in groups of bytes, halfwords, words and doublewords. There are other issues such as [endianness](#) and floating-point storage, which we will cover later.

2.2 Registers

As with most [von Neumann architecture](#) CPUs, the MIPS CPU cannot hold all of the data and instructions required to run a program. Instead, data and instructions are copied into a small set of [registers](#) inside the CPU. Operations are performed on the registers and the results are stored in the registers.

Because the number of registers is limited, we often need to copy values back out to memory to free up registers so that more data can be brought into the CPU: this is known as [spilling the registers](#).

The MIPS CPU provides 32 registers. Most of them are general-purpose in that the CPU hardware does not impose specific functions on each register. That said, it turns out that the software [toolchain](#) (i.e. the assembler, compilers and libraries) do set aside certain registers for certain purposes.

Each register is numbered 0 to 31, and in the MIPS assembly language, each register is identified by its number preceded by a '\$' sign, so \$12 is register 12. Most of the registers also have a *mnemonic* name to remind you of their purpose.

Register	Mnemonic Name	Purpose
0	zero	the value 0
1	\$at	(assembler temporary) Reserved by the assembler
2-3	\$v0 - \$v1	(values) from expression evaluation and function results
4-7	\$a0 - \$a3	(arguments) First four parameters for subroutine. Not preserved across procedure calls
8-15	\$t0 - \$t7	(temporaries) Caller saved if needed. Subroutines can use w/out saving. Not preserved across procedure calls
16-23	\$s0 - \$s7	(saved values) Callee saved. A subroutine using one of these must save original and restore it before exiting. Preserved across procedure calls
24-25	\$t8 - \$t9	(temporaries) Caller saved if needed. Subroutines can use w/out saving. Not preserved across procedure calls
26-27	\$k0 - \$k1	Reserved for use by the operating system kernel
28	\$gp	global pointer . Points to the middle of the 64K block of memory in the static data segment.

29	\$sp	stack p ointer. Points to last location on the stack.
30	\$s8 or \$fp	saved value / frame p ointer. Preserved across procedure calls
31	\$ra	r eturn a ddress

2.3 Instruction Set

The MIPS family provides instructions to perform the following operations:

- load registers with values, either from RAM or with literal values
- store register values (i.e. copy them) out to RAM locations
- basic integer arithmetic: add, subtract, multiply, divide with remainder
- basic floating-point arithmetic: add, subtract, multiply, divide
- logical operations: AND, OR, NOT, exclusive OR (XOR)
- shift operations: shift left, shift right
- comparison operations: ==, !=, <, >, <=, >=
- instructions to change the flow of control: relative branches and jumps

For more details, see the *MIPS Architecture and Assembly Language Overview* in our Resources module and the Wikipedia article on the [MIPS CPU](#).

3 Assembly Language

As with all CPUs, the MIPS CPU only understands [machine language](#), where each instruction is a binary pattern which represents the operation to perform, and the operands to perform it on. Fortunately, there is a more textual, human-readable form for this called [assembly language](#), which is what we are going to learn. A tool called an *assembler* translates our assembly language programs into the binary machine code, which is what the CPU will run. The assembler does some other jobs for us, such as:

- choosing RAM locations to store our variables
- laying out the RAM locations for our functions and methods
- implementing some *pseudoinstructions*, where one assembly language instruction is translated into several real machine instructions

All of the above makes our job as assembly language programmers somewhat easier. However, it is still much harder to write in low-level assembly than it is to use a high-level language like C or Java.

3.1 Literals and Registers

Literal integers are written in signed decimal, e.g. 4, 23, 1854, -67. Registers can be named as integers preceded by a '\$' sign, e.g. \$0, \$25, \$16, but it is preferable to use their mnemonic name instead, e.g. \$sp, \$v0, \$t3.

Character and string literals are written the same as in Java: single quotes around characters, double quotes around strings, e.g. 'x', 'c', "hello", "Mary had a little lamb".

3.2 Basic Program Structure

MIPS assembly language programs are stored in plain text files ending with the suffix ".s" or the suffix ".asm". You can use any text editor you want to create and edit your assembly programs. You can use either the MARS Simulator to run and test your programs. This lab we are using the MARS MIPS simulator described below.

The basic structure of a MIPS assembly language program is:

- a **data section**, where your variables and their data sizes are named. The assembler will choose where in RAM to store your variables. The data section is identified by a line with the assembler directive

.data

There can be several data sections, but good style recommends a single data section at the top. This section is followed by

- a **code section**, which contains your assembly language instructions. The code section is identified by a line with the assembler directive

.text

Your code section must contain a starting point for your program's execution, marked by the label

```
main:
```

and your main code must end with a call to the `exit` system call, which tells the CPU to stop the program.

3.3 Comments and a Basic Template

Comments in your program are written starting with the '#' character, and they extend to the end of the line. The '#' character acts just like the '/' characters used by Java comments.

Putting all of the above together, a basic template for a MIPS assembly language program is this:

```
# Comments describing the purpose of this program, how to use it etc.
#
#
        .data            # Data declarations follow this line
                        # ...

        .text            # Instructions follow this line
main:      # First instruction to execute
            # ...
            # ...
```

```
li    $v0, 10 # system call code for exit = 10
syscall # call operating system
```

We have not yet covered the names of the MIPS instructions, nor have we covered how to write them with their operands; we will do this in future labs. For now, you should definitely read and **absorb** as much of the [MIPS Architecture and Assembly Language Overview](#) that you can.

4 The MARS Simulator

We are going to use the [MARS simulator](#) in this lab. This combines a MIPS assembler, and a simulated MIPS CPU, and it allows you to:

- assemble your programs into machine code
- execute them
- see the contents of the CPU registers, the CPU flags and the contents of RAM
- single-step your program, so that you can see the effect that each instruction has on the CPU and RAM
- debug your program by setting breakpoints, single-stepping and inspecting the contents of registers and RAM

4.1 Download the Simulator

MARS is installed in the STEM Success Center Lab, but you may wish to download and use it on your own computer.

Use the above link to the MARS website, go to the Download section and download the *jar* file which is the MARS simulator. You may need to right-click on the "Download MARS" icon and choose "Save As ...". You should be able to download the *jar* file for MARS and run it on your own Windows, OS X or Linux system. Just make sure that you have the Java J2SE 1.5 (or later) SDK installed on your computer.

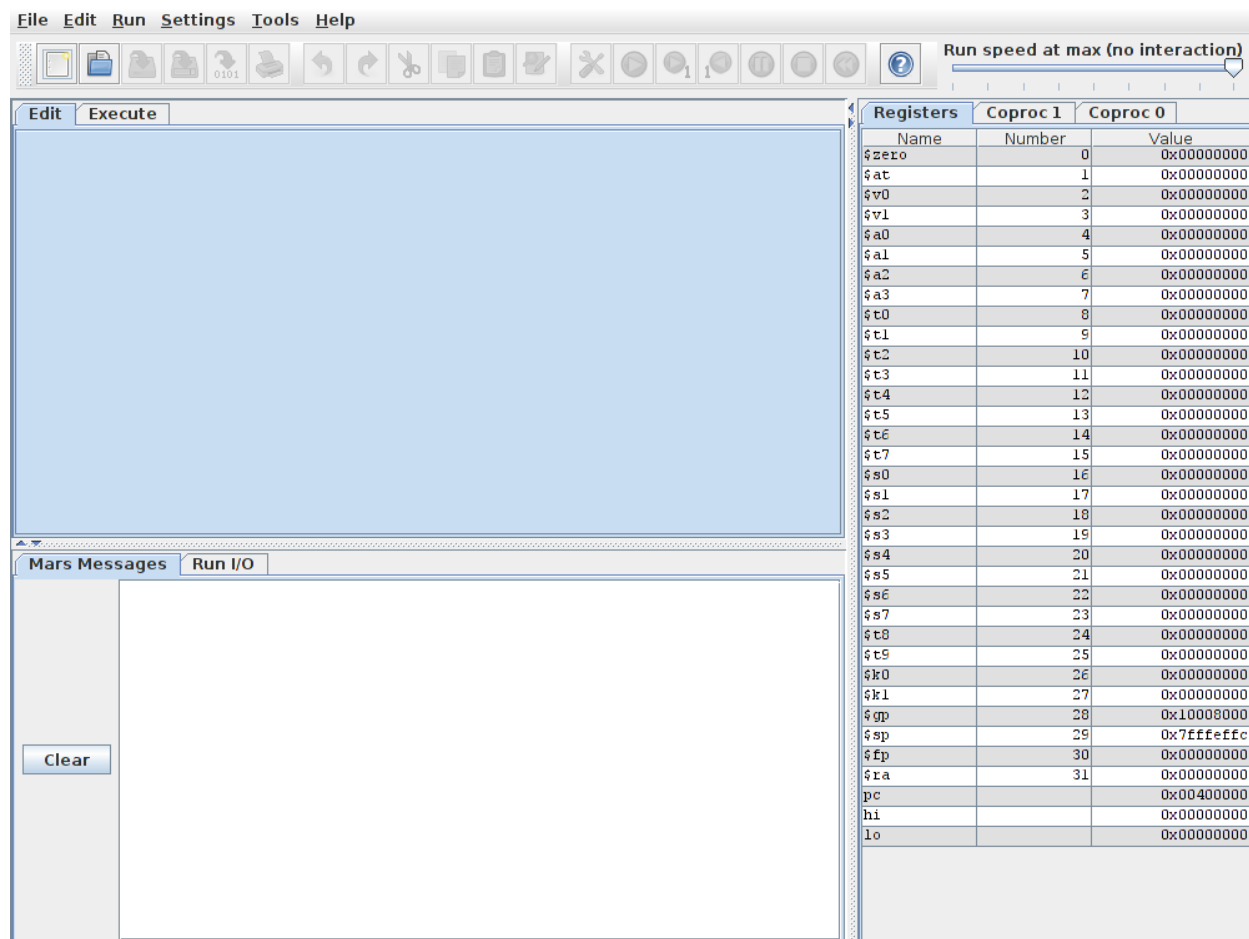
4.2 Running the Simulator

On Windows, you should be able to run MARS by finding the *jar* file in its folder, and double-clicking on it; you might even be able to do this on OS X. On Ubuntu Linux, you can use the Nautilus file browser to find the *jar* file, right-click on it and choose "Open with Java Runtime".

For OS X and Linux, if you can't get MARS running via the graphical user interface, you can open up a terminal window. Change directory (*cd*) into the folder where the *jar* file is, and run the command:

```
$ java -jar Mars.jar          # or whatever the jar file is called
```

When MARS runs, you should see a splash screen for a bit, and then you will get a window like this one:



On the left, the blue Edit tab will show you your assembly program. Once assembled, the Execute tab will display the RAM locations which hold the machine instructions and your program's data. On the right is a display of the 32 registers in the CPU and their values; the other tabs will reveal the floating-point registers.

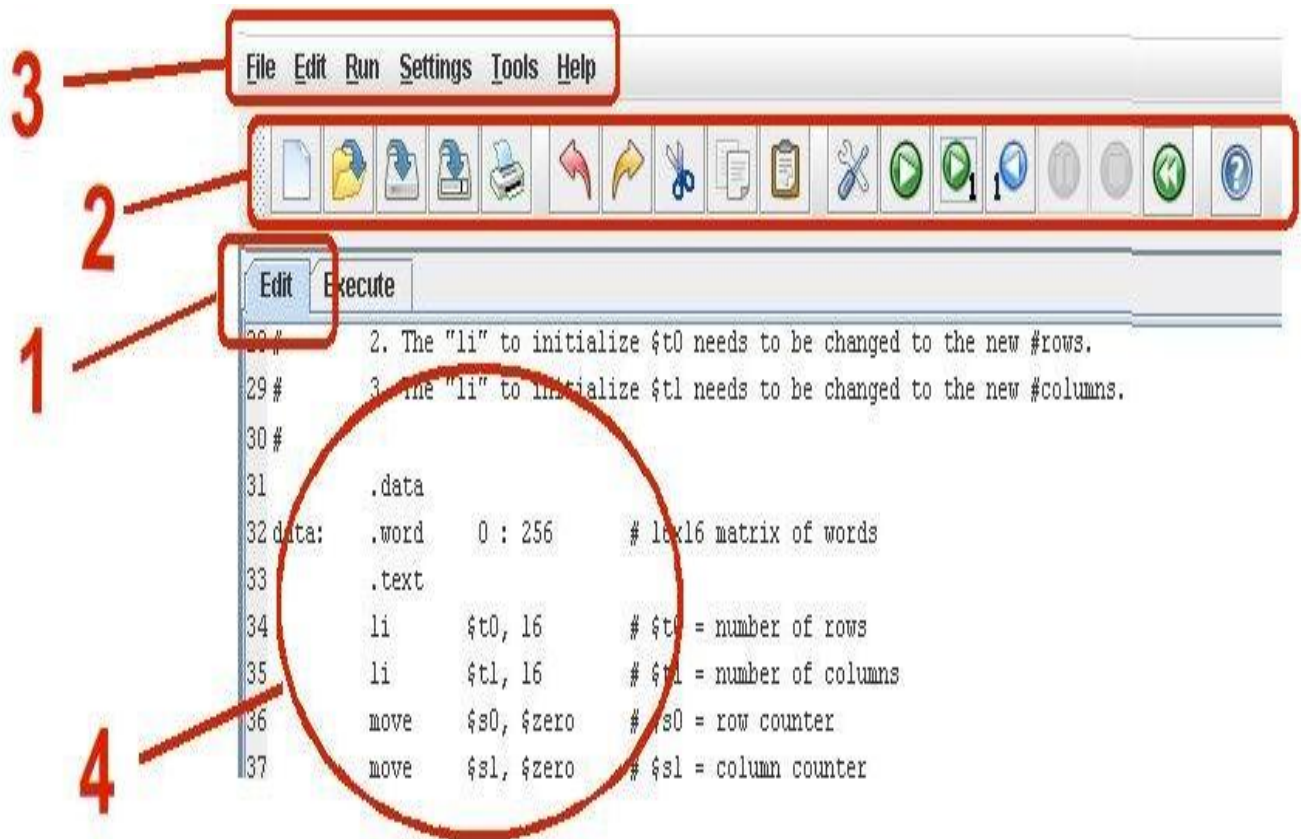
Down the bottom, the Mars Message tab shows messages from the MARS program, e.g. errors it finds when you try to assemble your program. The Run I/O tab shows the input to your program (i.e. what you type in), and any output that your program prints out.

4.3 Getting Help

The Help menu item at the top of the main MARS window brings up a second window with a heap of concise cheat sheets which are very useful. The main set of instructions for MARS is available on the [MARS Website](#) you should bookmark this link. However, they are quite terse.

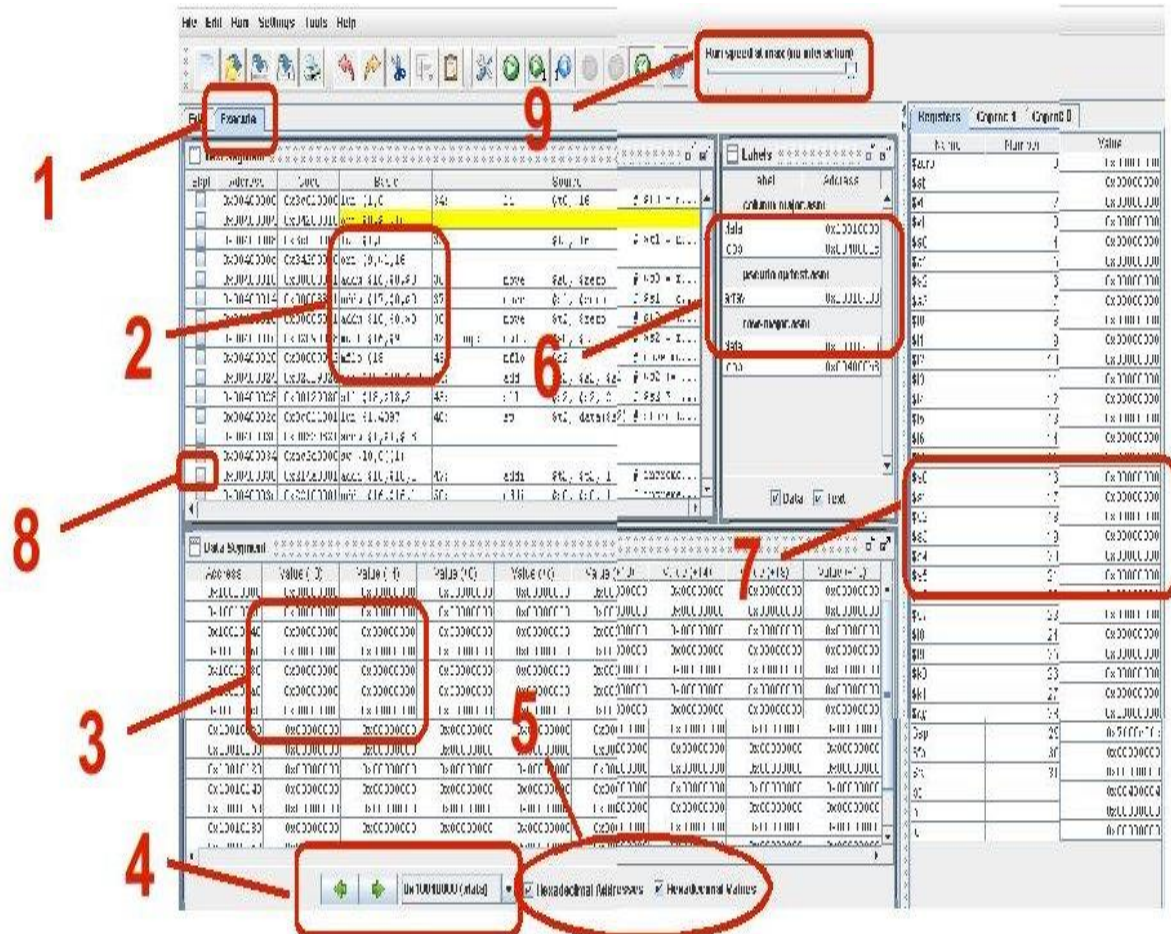
There is a [MARS Tutorial](#). The features map document gives a description of the Edit and Execute screens as follows.

4.3.1 Edit Screen



1. Edit display is indicated by highlighted tab.
- 2, 3. Typical edit and execute operations are available through icons and menus, dimmed-out when unavailable or not applicable.
4. WYSIWYG editor for MIPS assembly language code.

4.3.2 Execute Screen





1. Execute display is indicated by highlighted tab.
2. Assembly code is displayed with its address, machine code, assembly code, and the corresponding line from the source code file. (Source code and assembly code will differ when pseudoinstructions have been used.)
3. The values stored in Memory are directly editable (similar to a spreadsheet).
4. The window onto the Memory display is controlled in several ways: previous/next arrows and a menu of common locations (e.g., top of stack).
5. The numeric base used for the display of data values and addresses (memory and registers) is selectable between decimal and hexadecimal.
6. Addresses of labels and data declarations are available. Typically, these are used only when single-stepping to verify that an address is as expected.
7. The values stored in Registers are directly editable (similar to a spreadsheet).
8. Breakpoints are set by a checkbox for each assembly instruction. These checkboxes are always displayed and available.
9. Selectable speed of execution allows the user to "watch the action" instead of the assembly program finishing directly.

4.4 The MARS Tutorial

I am reproducing a modified version of the tutorial from the [MARS Tutorial](#), but leaving out the material which is not relevant to us at this time. Before following the tutorial, you should save the provided **fibonacci.asm** assembly program into the same area as you have the MARS simulator *jar* file:

The example program is **fibonacci.asm** which computes everyone's favorite number sequence up to the highest number below one million.

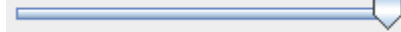
1. Start MARS by double-clicking on its icon, or using `java -jar Mars.jar`.
2. Use the menubar File -> Open or the Open icon  to open *fibonacci.asm*.
3. In the Edit view, look at the .data section at the top of the file. You should see two ASCII strings, one labeled *message* and the other labeled *newline*.
4. You can use the Settings menu option to configure the MARS displays. The settings will be retained for the next MARS session.
 - The Labels display contains the addresses of the assembly code statements with a label, but the default is to not show this display: **make sure that you enable it**.
 - You can select your preference for allowing pseudo-instructions (programmer-friendly instruction substitutions and shorthand).
 - You can select your preference for assembling only one file, or many files together (all the files in the current folder). This feature is useful for subroutines contained in separate files, etc.
 - You can select the startup display format of addresses and values (decimal or hexadecimal). Choose hexadecimal for now.
5. The provided assembly program *fibonacci.asm* is complete. Assemble the program using Run -> Assemble or click on the Assemble icon . The display should switch from the Edit view to the Execute view.
6. In the Execute view, there are 3 sub-windows:
 - a. **Text Segment**, which shows the machine instructions.
 - b. **Labels**, which shows the addresses of labeled items, i.e. variables and jump endpoints.
 - c. **Data Segment**, which shows the RAM locations that hold variables and other data.

Using the Labels sub-window, identify the locations of the *message* and *newline* strings. Single-click on each name, and the RAM location which holds this value will be highlighted with a blue border in the Data Segment sub-window. You can use the **Hexadecimal Values** tickbox to toggle between decimal and hexadecimal display of the values in RAM.





The *prompt* string starts at address 0x10010000, and is null-terminated (i.e. ends with a byte with value 0). The *newline* string starts at 0x1001002f.

Locate the Registers display, which shows the 32 common MIPS registers. Other tabs in the Registers display show the floating-point registers (Coproc 1) and status codes (Coproc 0).

Run speed at max (no interaction)

Use the slider bar  to change the run speed to about 10 instructions per second. This allows us to watch the action instead of the assembly program finishing directly.

Choose how you will execute the program:


- The Run -> Go option and the  icon runs the program to completion. Using this icon, you should observe the yellow highlight showing the programs progress and the values of the Fibonacci sequence appearing in the \$t2 register in the Registers display. Turn off **Hexadecimal Values** to see the values in decimal.
- The Run -> Reset option and the  icon resets the program and simulator to initial values. Memory contents are those specified within the program, and register contents are generally zero.
- The Run -> Step option and the  icon performs single-step: one instruction is performed and then the simulation stops.
- The complement to "single-step" is Run -> Backstep and the  icon; this undoes each single operation.


Run the program to completion. Observe the output of the program in the Run I/O display window:

```
The Fibonacci numbers below 1 million are
1
1
2
3
5
8
...
```

Modify the contents of a register. (Modifying a memory value is exactly the same.)

- In the **Text Segment** window, each instruction line has a checkbox on the left. If you select this checkbox, it sets a breakpoint, and the execution will stop when the program reaches this checkpoint.
- Set a checkpoint on address 0x00400050 which is the `b loop` instruction.
- Now reset and re-run the program, which will now stop at the breakpoint.

- Double-click on the value the \$t1 register's value in the **Registers** window. The cell will be highlighted and will accept keyboard entry, similar to a spreadsheet. Enter some noticeably different value, and use the Enter key or click outside the cell to indicate that the change is complete.
- Do Run -> Go or click on  to continue from the breakpoint. The program now uses your entered value instead of the computed Fibonacci number.

Open the Help menu option or use the  icon for information on MIPS instructions, pseudoinstructions, directives, and syscalls.

4.5 Understanding the Fibonacci Program

You are not expected to grasp the full intricacies of the Fibonacci program this week, but you should read through the code and pick these things up:

- load (l) and store (s) instructions move data between memory and registers. They work on such things as **w**ords, **i**mmEDIATE literal values and **a**ddresses.
- the add instruction can add registers together, or it can add **i**mmEDIATE literal values to registers.
- there are no high-level control structures such as loops or method/function calls. These have to be performed using instructions which **b**ranch if one register is, for example, **g**reater than a second register.
- the program relies on an *operating system* to provide functionality such as printing to the terminal and reading from the terminal. We invoke the operating system through system calls, but we have to load values into specific registers to tell the operating system exactly what system call we want to perform, and the arguments to the system call.

5 Outlook for the Next Lab

In our next lab, we will learn the basic set of instructions in the MIPS assembly language so that you can begin to write your own programs. Until then, please go through all of the information above and try to absorb as much of it as you can.