Lab 1: Landing on MARS

Finish by midnight on Saturday 9/8

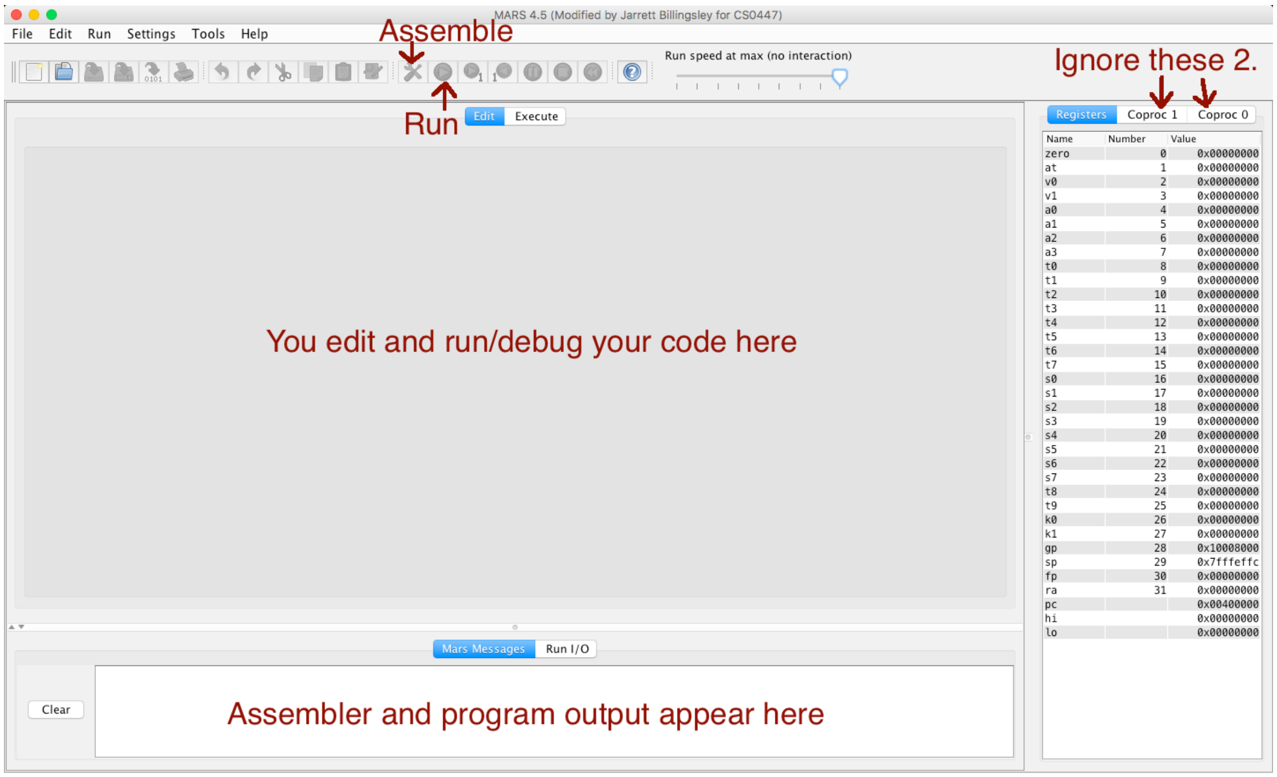
This lab is about familiarizing you with the MARS MIPS simulator software, and getting you started on your FUN ASSEMBLY LANGUAGE JOURNEY.

1. Getting started

There is a new version of MARS, `MARS_2191_b.jar`. If you already downloaded MARS, please redownload this new version and delete the old one.

You must use the modified version of MARS available on the software page. Please make sure you have the newest version of the JRE installed.

You should be able to double-click the JAR file to run it. You will see this:


MARS 4.5 (Modified by Jarrett Billingsley for CS0447)
File Edit Run Settings Tools Help
Assemble
Run
Run speed at max (no interaction)
Ignore these 2.
Edit Execute
Registers Coproc 1 Coproc 0
You edit and run/debug your code here
Mars Messages Run I/O
Clear
Assembler and program output appear here

Name	Number	Value
zero	0	0x00000000
at	1	0x00000000
v0	2	0x00000000
v1	3	0x00000000
a0	4	0x00000000
a1	5	0x00000000
a2	6	0x00000000
a3	7	0x00000000
t0	8	0x00000000
t1	9	0x00000000
t2	10	0x00000000
t3	11	0x00000000
t4	12	0x00000000
t5	13	0x00000000
t6	14	0x00000000
t7	15	0x00000000
s0	16	0x00000000
s1	17	0x00000000
s2	18	0x00000000
s3	19	0x00000000
s4	20	0x00000000
s5	21	0x00000000
s6	22	0x00000000
s7	23	0x00000000
t8	24	0x00000000
t9	25	0x00000000
k0	26	0x00000000
k1	27	0x00000000
gp	28	0x10008000
sp	29	0x7fffeffc
fp	30	0x00000000
ra	31	0x00000000
pc		0x00400000
hi		0x00000000
lo		0x00000000

Setting things up (important!)

In the **Settings** menu, **make sure the following things are checked (enabled):**

- Show Labels Window (symbol table)
- Clear Run I/O upon assembling
- Initialize Program Counter to global 'main' if defined

Leave the other settings unchanged.

---

2. Hello, nothing

1. Make a new file, and save it with the name `username_lab1.asm`, where `username` is your username. (I would put `jfb42_lab1.asm`.)

2. In MIPS, comments start with a `#` sign. At the top of the file, put your name and username in a comment.

3. Just like in Java or C, our assembly programs start at a `main` function. Type these lines into your file:

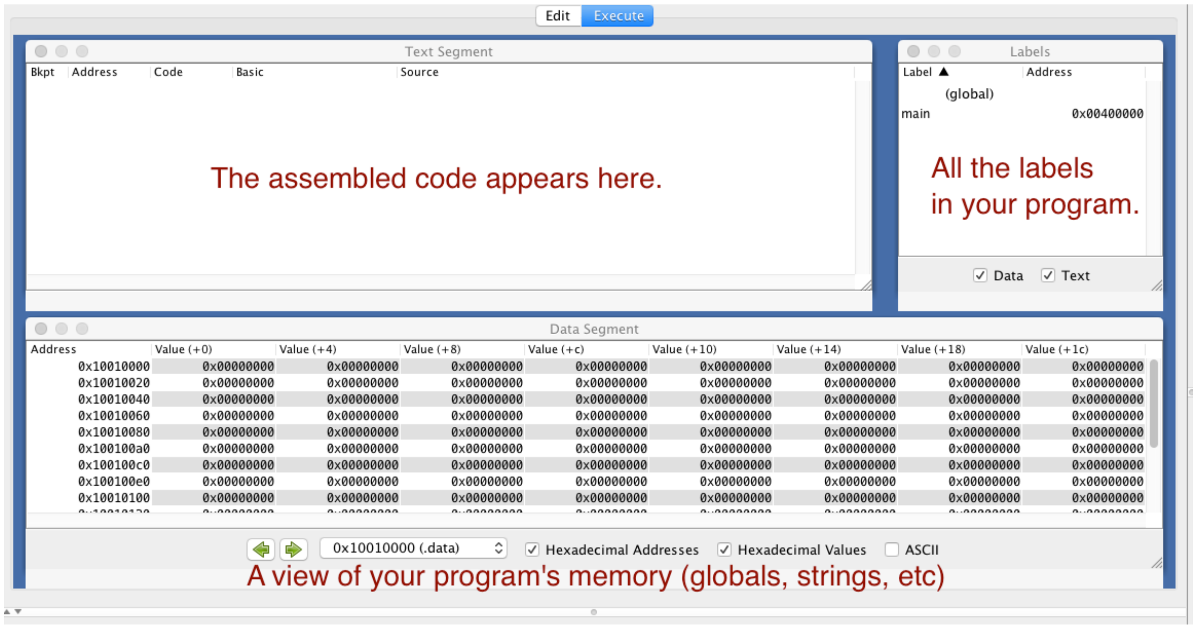
   ```
   .globl main
   main:
   ```

   The `.globl` directive is only needed for `main`. Most of your functions won't need it.

   That is not a typo: it is `.globl`, ***not `.global.***

   `main:` is a **label.** Labels name parts of your code. Whatever you write after the label will be the code of the `main` function.

4. Instead of "compiling," we **"assemble"** an asm program. When you assemble the program with , it'll switch to the Execute tab.


Edit
Execute
Text Segment
Bkpt
Address
Code
Basic
Source
The assembled code appears here.
Labels
All the labels in your program.
Data Segment
A view of your program's memory (globals, strings, etc)

Label	Address
(global)	
main	0x00400000

☑ Data ☑ Text

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010100	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

0x10010000 (.data) ☑ Hexadecimal Addresses ☑ Hexadecimal Values ☐ ASCII

5. Now run with ▶. In the “Run I/O” at the bottom of the screen, it’ll say

   ```
   -- program is finished running (dropped off bottom) --
   ```

   It’s a completely empty program that does nothing. Yay!

---

3. Hello, *a number*

Now we want to do the equivalent of `System.out.print(1234);`.

1. Don’t copy and paste this, *type it in* after your `main:` line:

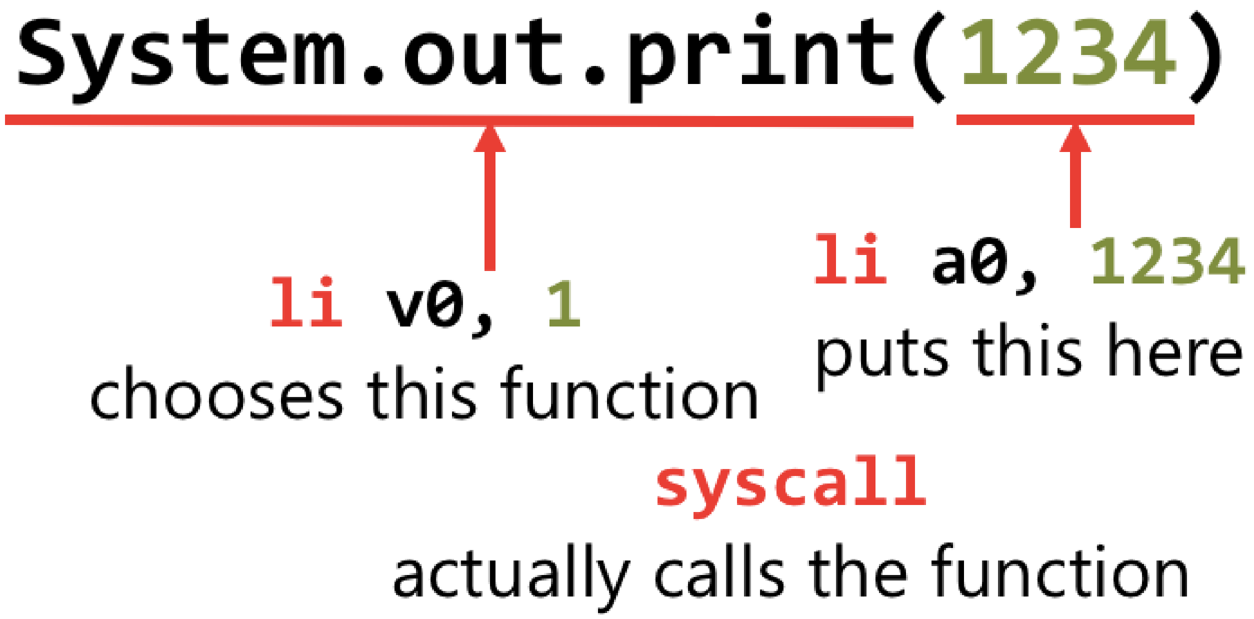
   ```
       li a0, 1234
       li v0, 1
       syscall
   ```

2. If you assemble and run again, you’ll see the number `1234` printed in the “Run I/O”.

3. Try assembling, and then *stepping through* instruction-by-instruction with this button: ▶1. **Watch the values of the registers** as you do so.

What the heck is going on?

A **single line** of high-level code will usually become **multiple lines** of assembly.



`li` stands for “load immediate”, and it puts a **constant value** into a register.

“Immediate” is an assembly term for “a constant that is written inside the instruction.”

`a0` is an **argument register.** When you put things in argument registers, you know that a **function call** is about to happen.

`syscall` stands for “system call” and you’ll learn about those in 449.

The `syscall` instruction doesn’t let you specify which function to call, so instead, we tell MARS which system call we want by **putting the number of the syscall in the `v0` register.**

This is kind of a weird choice, since the `v` registers are usually used for return values. I’m not sure why they did this.

Hit F1 to open MARS help. Then click “Syscalls”. This tells you about what syscalls are, which ones are available, what their numbers are, and their arguments and return values.

We put 1 into `v0`. **Which syscall is 1?**

---

4. Making your *own* function

1. **Before your** `.globl main` **line,** type the following:

```
# ---------------------------------
print_int:

    jr ra
# ---------------------------------
```

I like to put those comment lines to visually separate functions, since assembly is so freeform.

You now have another label, `print_int`, which we'll be able to use as a function. But what is `jr ra`?

In a higher level language, you would write a function like:

```
void print_int() {
    // ...some code...
}
```

When the execution reaches the closing brace, the function is done, right? This is because *your high level language puts a* `return;` *for you* before the closing brace.

`jr ra` is the "return" instruction: it returns to the function that called this one. **So when you make a function, write the** `jr ra` **first so you won't forget,** and then write all the code between the label and the `jr ra`.

If you don't put `jr ra` **at the end of your function, bad things will happen.** You will get infinite loops and functions running when they weren't called and your program ending for no reason and it's just bad ok???

2. Now put the following between the `print_int:` and `jr ra` lines:

```
    li v0, 1
    syscall
```

So your `print_int` function will have 3 instructions.

3. Now let's go back to `main`, *after the* `syscall`, and do the equivalent of `print_int(5678)`.

   `print_int` takes one *argument*, which goes into `a0`. **Write the instruction to put 5678 into `a0`.**

4. After that, write:

   ```
   	jal		print_int
   ```

   This is a *regular* function call. `jal` is an instruction which calls the function you name.

5. When you assemble and run your program, it should display:

   ```
   12345678
   ```

   If you get `Error in : invalid program counter value: 0x00000000`, don't forget to turn on the "Initialize program counter to global `main`" setting.

6. Now **change the previous syscall in `main` to call your `print_int` function with `1234` instead.** Your program should still print `12345678`.

---

5. Making `print_int` put a newline after the number

1. Look at the "Syscalls" section of the MARS help. Find one that lets you print *a single character*. **Read what register it expects its argument to be in.**

2. Inside `print_int`, between `syscall` and `jr ra`, **call that "print a character" syscall** with the newline character.

   Hint: you can put a single character in a register using single quotes, like `li a0, 'A'`. Do you remember how to write a newline character? It starts with `\`

3. Done correctly, your program will now output:

```
1234
5678
```

---

6. Printing a string

A string is an array of characters, and cannot possibly fit into a register. Instead, we deal with the string's *address* - where, in memory, the string begins.

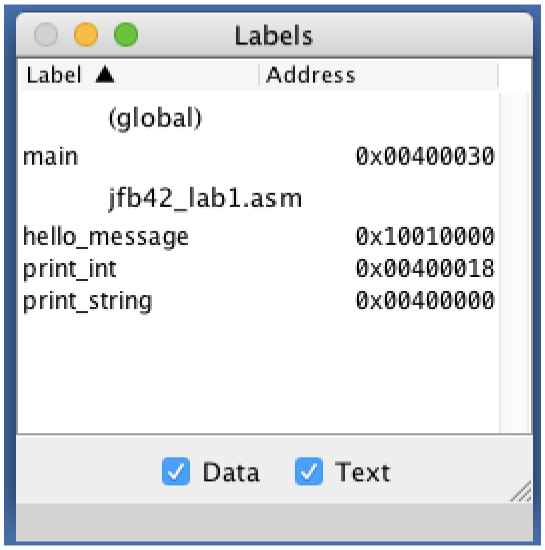
1. To put a string into memory, we have to switch to the **data segment.** At the top of your program, do this:

   ```
   .data
   hello_message: .asciiz "Hello, world!"

   .text
   ```

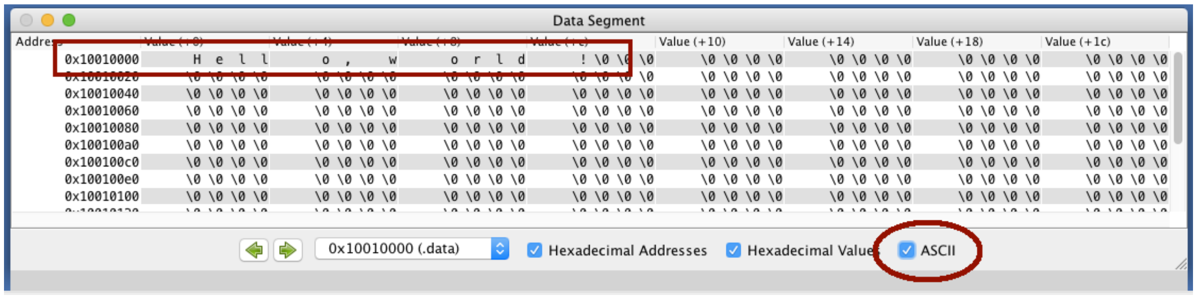
   The `.data` and `.text` directives tell the assembler to switch to the data and text (code) segments. **Any time you switch to the data segment, you must switch back to the text segment to write more code.**

2. Try assembling, and look at the labels and memory views. The labels view shows your new `hello_message` label at address `0x10010000`.



Label ▲	Address
(global)	
main	0x00400030
jfb42_lab1.asm	
hello_message	0x10010000
print_int	0x00400018
print_string	0x00400000

In the memory view, make sure the "ASCII" option is checked, and you can see your string in memory starting at address `0x10010000`.



```
Data Segment
Address     Value (+0)  Value (+4)  Value (+8)  Value (+c)  Value (+10) Value (+14) Value (+18) Value (+1c)
0x10010000     H e l l     o ,   w     o r l d   ! \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
0x10010020  \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
0x10010040  \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
0x10010060  \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
0x10010080  \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
0x100100a0  \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
0x100100c0  \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
0x100100e0  \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
0x10010100  \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
```

3. **Make a new function,** `print_string`, which works like `print_int`. But instead of using syscall 1 to print an int, **look up the syscall to print a string.** `print_string` will also print a newline after printing the string, just like `print_int` did.

4. In `main`, you want to call `print_string`, but to set up its argument `a0`, you can't use `li`. We want the string's *address* to go into `a0`, so we use a new instruction, `la` (load address):

```
    la a0, hello_message
    jal     print_string
```

5. Assemble and run. It should print the string on its own line, wherever you decided to call it in `main`. I put mine at the beginning:

```
Hello, world!
1234
5678
```

6. **Assemble and step through your program line-by-line, and watch the registers change.** In particular, watch what happens on the `la` line you just added. See what value actually goes into `a0`?

---

7. Nested function calls

1. Make a new function called `print_123` which does the assembly equivalent of:

```
print_int(1);
print_int(2);
print_int(3);
```

2. Call `print_123` at the **beginning** of `main`.

3. Run it. ...Wait a second. Why did it print this and then stop?

   ```
   1
   2
   3
   ```

   Notice that *the program is still running*. In fact, it's trapped in an infinite loop.

4. **Hit the stop button** to stop it. Now modify `print_123` to look like this:

   ```
   print_123:
       push      ra

       #...the calls to print_int, but NOT the jr ra

       pop       ra
       jr        ra
   ```

   That is, insert those `push ra` and `pop ra` instructions at the beginning and just before the end of the function, respectively.

5. When you run it now, it should work perfectly. **It should print 1, 2, 3, "Hello, world!", 1234, 5678, and then stop.**

Why did this happen? `ra` is like a "bookmark" that the CPU uses to get back to where it left off, when you return from a function. But when you `jal print_int` inside of `print_123`, `ra` is **replaced by a bookmark back to** `print_123`. So the `jr ra` at the end of `print_123` just kept going back to the `print_123`, forever. Pushing and popping `ra` inside `print_123` lets us save the bookmark to `main` on the stack.

If that didn't make any sense... that's fine. We'll get to it in class ;)

Submitting

Make sure your file is named `username_lab1.asm`, like `jfb42_lab1.asm`.

Submit here.

Drag your asm file into your browser to upload. **If you can see your file, you uploaded it correctly!**

You can also re-upload if you made a mistake and need to fix it.