Lab 3

Deadline: Monday, July 6th

Goals

- Practice running and debugging RISC-V assembly code.
- Write RISC-V functions with the correct function calling procedure.
- Get an idea of how to translate C code to RISC-V.

Getting the files

To get the starter files for this lab, run the following command in your `labs` directory.

```
$ git pull starter master
```

If you get an error like the following:

```
fatal: 'starter' does not appear to be a git repository
fatal: Could not read from remote repository.
```

make sure to set the starter remote as follows:

```
git remote add starter https://github.com/61c-teach/su20-lab-starter.git
```

and run the original command again.

Intro to Assembly with RISC-V Simulator

So far, we have been dealing with C program files (`.c` file extension), and have been using the `gcc` compiler to execute these higher-level language programs. Now, we are learning about the RISC-V assembly language, which is a lower-level language much closer to machine code. For context, `gcc` takes the C code we write, first compiles this down to assembly code (e.g. x86, ARM), and then assembles this down to machine code/binary.

In this lab, we will deal with several RISC-V assembly program files, each of which have a `.s` file extension. To run these, we will need to use **Venus**, a RISC-V simulator that you can find here. There is also a `.jar` version of Venus that we have provided in the `tools` folder under your base lab repository.

Assembly/Venus Basics:

- Enter your code in the "Editor" tab
- Programs start at the first line regardless of the label. That means that the `main` function must be put first.
  - Note: Sometimes, we want to pre-allocate some items in memory before the program starts executing (more on this in Exercise 1 below!). Since this allocation isn't actual code, we can place it before the `main` function.
- Programs end with an `ecall` with argument value 10. This signals for the program to exit. The `ecall` instructions are analogous to "System Calls" and allow us to do things such as print to the console or request chunks of memory from the heap.
- Labels end with a colon (`:`).
- Comments start with a pound sign (`#`).
- You CANNOT put more than one instruction per line.

- When you are done editing, click the “Simulator” tab to prepare for execution.

For the following exercises, please save your completed code in a file on your local machine. Otherwise, we will have no proof that you completed the lab exercises.

Exercise 1: Familiarizing yourself with Venus

Getting started:

1. Paste the contents of `ex1.s` into the Venus editor.
2. Click the “Simulator” tab and click the “Assemble & Simulate from Editor” button. This will prepare the code you wrote for execution. If you click back to the “Editor” tab, your simulation will be reset.
3. In the simulator, to execute the next instruction, click the “step” button.
4. To undo an instruction, click the “prev” button.
5. To run the program to completion, click the “run” button.
6. To reset the program from the start, click the “reset” button.
7. The contents of all 32 registers are on the right-hand side, and the console output is at the bottom
8. To view the contents of memory, click the “Memory” tab on the right. You can navigate to different portions of your memory using the dropdown menu at the bottom.

Action Item

Paste the contents of `ex1.s` in Venus and record your answers to the following questions. Some of the questions will require you to run the RISC-V code using Venus’ simulator tab.

1. What do the `.data`, `.word`, `.text` directives mean (i.e. what do you use them for)? *Hint*: think about the 4 sections of memory.
2. Run the program to completion. What number did the program output? What does this number represent?
3. At what address is `n` stored in memory? **Hint**: Look at the contents of the registers.
4. Without actually editing the code (i.e. without going into the “Editor” tab), have the program calculate the 13th fib number (0-indexed) by *manually* modifying the value of a register. You may find it helpful to first step through the code. If you prefer to look at decimal values, change the “Display Settings” option at the bottom.

Exercise 2: Translating from C to RISC-V

Open the files `ex2.c` and `ex2.s`. The assembly code provided (.s file) is a translation of the given C program into RISC-V.

Action Item

Find/explain the following components of this assembly file.

- The register representing the variable `k`.
- The register representing the variable `sum`.
- The registers acting as pointers to the `source` and `dest` arrays.
- The assembly code for the loop found in the C code.
- How the pointers are manipulated in the assembly code.

Exercise 3: Factorial

In this exercise, you will be implementing the `factorial` function in RISC-V. This function takes in a single integer parameter `n` and returns `n!`. A stub of this function can be found in the file `factorial.s`.

You will only need to add instructions under the `factorial` label, and the argument that is passed into the function is configured to be located at the label `n`. You may solve this problem using either recursion or iteration.

Testing

As a sanity check, you should make sure your function properly returns that `3! = 6`, `7! = 5040` and `8! = 40320`.

You have the option to test this using the online version of Venus, but we've provided a `.jar` for you to test locally! We'll be using the `.jar` in the autograder so make sure to update your `factorial.s` file and run the following command before you submit to verify that the output is correct. Note that you will need to have java installed to run this command.

```
$ java -jar tools/venus.jar lab03/factorial.s
```

Exercise 4: RISC-V function calling with `map`

This exercise uses the file `list_map.s`.

In this exercise, you will complete an implementation of map on linked-lists in RISC-V. Our function will be simplified to mutate the list in-place, rather than creating and returning a new list with the modified values.

You will find it helpful to refer to the RISC-V green card to complete this exercise. If you encounter any instructions or pseudo-instructions you are unfamiliar with, use this as a resource.

Our `map` procedure will take two parameters; the first parameter will be the address of the head node of a singly-linked list whose values are 32-bit integers. So, in C, the structure would be defined as:

```
struct node {
    int value;
    struct node *next;
};
```

Our second parameter will be the **address of a function** that takes one int as an argument and returns an int. We'll use the `jalr` RISC-V instruction to call this function on the list node values.

Our `map` function will recursively go down the list, applying the function to each value of the list and storing the value returned in that corresponding node. In C, the function would be something like this:

```
void map(struct node *head, int (*f)(int))
{
    if(!head) { return; }
    head->value = f(head->value);
    map(head->next,f);
}
```

If you haven't seen the `int (*f)(int)` kind of declaration before, don't worry too much about it. Basically it means that `f` is a pointer to a function, which, in C, can then be used exactly like any other function.

There are exactly nine (9) markers (8 in `map` and 1 in `main`) in the provided code where it says `YOUR CODE HERE`.

Action Item

Complete the implementation of `map` by filling out each of these nine markers with the appropriate code. Furthermore, provide a sample call to `map` with `square` as the function argument. There are comments in the code that explain what should be accomplished at each marker. When you've filled in these instructions, running the code should provide you with the following output:

```
9 8 7 6 5 4 3 2 1 0
81 64 49 36 25 16 9 4 1 0
```

The first line is the original list, and the second line is the modified list after the map function (in this case square) is applied.

Testing

To test this locally, run the following command in your root lab directory (much like the one for `factorial.s`):

```
$ java -jar tools/venus.jar lab03/list_map.s
```

Transitioning to More Complex RISC-V Programs.

Once we begin to write more complex RISC-V programs, it will become necessary to split our code into multiple files and debug/test them individually. This is impossible to do in the regular Venus web editor, but if we open the "Venus" tab, we'll see that there is an online web terminal (with a corresponding virtual filesystem) that we can use to edit and test multiple files at once.

Note: To ensure that you don't lose any of your work, make sure to save local copies of your files *frequently*.

Working with Multiple Files

- The way we work with multiple files in RISC-V is with the `.import` and `.globl` keywords.
- The `.globl` keyword can be placed in a file and defines the labels we want to expose to other files when they import this file; it's analogous to defining a function in a header file.
    - If you open the `factorial.s` file, you'll notice that we export your `factorial` label at the top with the `.globl factorial` line at the top. This is actually how the autograder tests your function so don't remove it!
- The `.import` line allows us to import other files into the current one. Specifically, it will import only the labels that the file specifies with the `.globl` keyword. So, if we wanted to import the `factorial` label in a different file, we would add the line `.import factorial.s` at the top of the new file.

Using the Venus Web Terminal

- So far, we've spent most of our time in the `Editor` and `Simulator` tabs on Venus.
- If we open the `Venus` tab, we'll see a terminal-looking interface. Entering `help` will show us some of the commands we can run in this terminal environment.
- The commands you'll likely use most are `ls`, `touch`, `edit`, `upload`, and `download`.
    - `ls`: Lists the contents of the current directory.

  - `touch`: Creates a new empty file that we can modify later.
  - `edit`: Opens up the specified file in the "Editor" tab (Ex. `edit file.s`). Note that the `Cmd` + `S` and `Ctrl` + `S` shortcuts will work to save the file in the virtual filesystem if you choose to edit a different file later. But, to ensure that your work is saved, we recommend that you periodically save local copies of every file.
  - `upload`: Opens a prompt to upload files from our local computer to the virtual filesystem. You can hold down the `Ctrl` or `Cmd` keys to select multiple files to upload. Once a file is uploaded, we can edit it in the Venus editor using the `edit` command.
  - `download`: Downloads the specified files locally. You can specify multiple files to download, e.g. `download file1.s file2.s`, and each one will have a corresponding prompt to ask you where to download the files.

Passing in Command Line Arguments

- If you're working locally, then the Venus jar will interpret anything after the filename and relevant flags to be command-line arguments. For instance, if our `factorial.s` accepted command-line arguments, we could run the following:

```
$ java -jar tools/venus.jar lab03/factorial.s arg1 arg2 arg3
```

- If Venus detects that command-line arguments are being passed, then it will set `a0` to be the equivalent of `argc` and `a1` to be the equivalent of `argv`.
- We can pass command-line arguments to our programs by using the "Simulator Default Args" field and then you can run the program using those arguments by going to the "Simulator" tab.

Checkoff

Please submit to the Lab Autograder assignment (same as last week!).

Checkoff questions for lab 4:

- Make sure you understand and can answer the questions in exercises 1 and 2!