Lab 3

Before you begin

This lab is somewhat longer than the previous labs (and longer than most future labs as well). This is in order to give everyone a chance to get comfortable with writing RISC-V and using the Venus simulator before the release of project 2, which will require you to write a fair amount of assembly.

To this end, we recommend that you watch this week's lectures and get started on this assignment as early as possible. Good luck! If you have any questions or run into any issues, then please feel free to ask in office hours or on Piazza.

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.
- Get familiar with using the Venus simulator

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/fa20-lab-starter.git
```

and run the original command again.

Intro to Assembly with RISC-V Simulator

In this course, we have so far dealt mostly with C programs (with the `.c` file extension), used the `gcc` program to compile them to machine code, and then executed them directly on your computer or hive machine. Now, we're shifting our focus to the RISC-V assembly language, which is a lower-level language much closer to machine code. We can't execute RISC-V code directly because your computer and the hives are built to run machine code from other assembly languages - most likely x86 or ARM.

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

To get started with Venus, please take a look at "The Editor Tab" and "The Simulator Tab" in the Venus reference. We recommend that you read this whole page at some point, but these sections should be enough to get started.

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

Exercise 0: Connecting your files to Venus

In previous iterations of the course, the Venus web editor allows you to write assembly code from scratch or to manually upload and download files, but thanks to a recent update, you can now "mount" a folder from your local device onto Venus's web frontend! This means that any changes you make in Venus's editor will be reflected in your local files, and vice versa.

This exercise will walk you through the process of connecting your file system to Venus, which should save you a lot of trouble copy/pasting files between your local drive and the Venus editor.

If for some reason this feature ends up not working for you (it's relatively new, and there's a chance there might still be bugs), then for the rest of this assignment, wherever it says to open a file in Venus, you should copy/paste the contents into the Venus web editor, and manually copy/paste those changes back to your local machine.

Here's what you need to do:

- In the labs folder on your local machine, run `java -jar tools/venus.jar . -dm`. This will expose your lab directory to Venus on a network port (6161 by default).
  - You should see the message `To connect, enter `mount http://localhost:6161 vmfs` on Venus.`, as well as a a big "Javalin" logo.
  - If you see a message along the lines of "port unable to be bound", then you can specify the port number explicitly by appending `--port <port number>` to the command (for example, `java -jar tools/venus.jar . -dm --port 6162` will expose the file system on port 6162)
- Open https://venus.cs61c.org in your web browser. In the Venus terminal, run `mount local vmfs` (if you chose a different port, replace "local" with the full URL, such as `http://localhost:6162`). This connects Venus to your file system.
- Go to the "Files" tab. You should now be able to see your labs directory under the `vmfs` folder.
- Navigate to `lab03`, and make sure it works by hitting the `Edit` button next to `ex1.s`. This should open in the `Editor` tab.
  - If you make any changes to the file in the `Editor` tab, hitting command-s on a Mac and ctrl-s on Windows/Linux will update your local copy of the file. To check if the save was successful, open the file on your local machine to see if it matches what you have in the web editor (unfortunately no feedback message has been implemented yet).
  - **Note:** If you make any changes to a file in your local machine, if you had the same file open in the Venus editor, you'll need to reopen it from the "Files" menu to get the new changes.
- To make it so that the file system will attempt to remount automatically whenever you close and reopen Venus, enable "Save on Close" in the Settings pane (again in the Venus tab). This will make the Venus web client attempt to locate the file system exposed by running the Venus jar, and will pop up an error saying that it couldn't connect to the server if it doesn't see it running. If this happens, just follow the above steps to manually remount the file system.

Once you've got `ex1.s` open, you're ready to move on to Exercise 1!

Exercise 1: Familiarizing yourself with Venus

Getting started:

1. Open `ex1.s` into the Venus editor. If you were unable to mount the filesystem in Exercise 0, then you can copy/paste `ex1.s` from your local machine into the Venus editor directly.
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. Note that undo may or may not undo operations performed by `ecall`, such as exiting the program or printing to console.
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

Open `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's 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.

In addition to opening a file in the "Editor" tab and then running in the "Simulator" tab as described above, you can also run `ex2.s` directly within the Venus terminal by `cd`ing into the appropriate folder, then running `run ex2.s` or `./ex2.s`. Typing `vdb ex2.s` will also assemble the file and take you to the "Simulator" tab directly.

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. You may also assume that the `factorial` function will only be called on positive values with results that won't overflow a 32-bit two's complement integer.

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: Calling Convention Checker

In this exercise, we'll be looking at the code in `cc_test.s`. We'll be using a feature that's only available on the command line version of Venus, so if you're still using the Venus web editor, make sure you hit cmd-s or ctrl-s to make sure your changes are reflected in your local files. Likewise, if you modify your local files and want to use the Venus web simulator, make sure to reopen your file through the simulator to make sure the changes are reflected.

Throughout this course, we will be running automated checks to make sure your assembly complies with RISC-V calling conventions, as described in lecture and discussion. Here's a quick recap: all functions that overwrite registers that are preserved by convention must have a prologue, which saves those register values to the stack at the start of the function, and an epilogue, which restores those values for the function's caller. You can find a more detailed explanation along with some concrete examples in these notes.

Bugs due to calling convention violations can often be difficult to find manually, so Venus provides a way to automatically report some of these errors at runtime.

Take a look at the contents of the `cc_test.s` file, particularly at the `main`, `simple_fn`, `naive_pow`, `inc_arr`, and `helper_fn` functions. Now, run Venus locally with the following command:

```
$ java -jar tools/venus.jar -cc lab03/cc_test.s
```

Note: The `-cc` flag **MUST** go between `tool/venus.jar` and the file you're running. If it goes before `venus.jar` it will be treated as an argument to the `java` program; if it goes after the file name it will treated as an argument to the `main` function of your assembly program.

The `-cc` flag enables the calling convention checker, and detects some basic violations. You should see an output similar to the following:

```
[CC Violation]: (PC=0x00000080) Usage of unset register t0! main.S:58 mv a0, t0
[CC Violation]: (PC=0x0000008C) Setting of a saved register (s0) which has not been saved! main.S:80 li s0, 1
[CC Violation]: (PC=0x00000094) Setting of a saved register (s0) which has not been saved! main.S:83 mul s0, s0, a0
[CC Violation]: (PC=0x00000094) Setting of a saved register (s0) which has not been saved! main.S:83 mul s0, s0, a0
[CC Violation]: (PC=0x00000094) Setting of a saved register (s0) which has not been saved! main.S:83 mul s0, s0, a0
[CC Violation]: (PC=0x00000094) Setting of a saved register (s0) which has not been saved! main.S:83 mul s0, s0, a0
[CC Violation]: (PC=0x00000094) Setting of a saved register (s0) which has not been saved! main.S:83 mul s0, s0, a0
[CC Violation]: (PC=0x00000094) Setting of a saved register (s0) which has not been saved! main.S:83 mul s0, s0, a0
[CC Violation]: (PC=0x00000094) Setting of a saved register (s0) which has not been saved! main.S:83 mul s0, s0, a0
[CC Violation]: (PC=0x000000A4) Save register s0 not correctly restored before return! Expected 0x00000A3F, Actual
0x00000080. main.S:90 ret
[CC Violation]: (PC=0x000000B0) Setting of a saved register (s0) which has not been saved! main.S:106 mv s0, a0 # Copy
start of array to saved register
[CC Violation]: (PC=0x000000B4) Setting of a saved register (s1) which has not been saved! main.S:107 mv s1, a1 # Copy
length of array to saved register
[CC Violation]: (PC=0x000000E4) Setting of a saved register (s0) which has not been saved! main.S:142 addi s0, t1, 1
Venus ran into a simulator error!
Attempting to access uninitialized memory between the stack and heap. Attempting to access '4' bytes at address
'347579389'.
```

Find the source of each of the errors reported by the CC checker and fix it. Even though the output says `main.S`, the line number reported should still correspond to the correct line in `cc_test.s`: this is just an artifact of the way Venus works in order to support combining multiple assembly files together. You can find a list of CC error messages, as well as their meanings, in the Venus reference.

Once you've fixed all the violations reported by the CC checker, the code might still fail: this is likely because there's still some remaining calling convention errors that Venus doesn't report. Since function calls in assembly language are ultimately just jumps, Venus can't report these violations without more information, at risk of producing false positives.

The fixes for all of these errors (both the ones reported by the CC checker and the ones it can't find) should be added near the lines marked by the `FIXME` comments in the starter code.

Note: Venus's calling convention checker will not report all calling convention bugs; it is intended to be used primarily as a sanity check. Most importantly, it will only look for bugs in functions that are exported with the `.globl` directive - the meaning of `.globl` is explained in more detail in the Venus reference.

Action Item

Resolve all the calling convention errors in `cc_test.s`, and be able to answer the following questions:

- What caused the errors in `simple_fn`, `naive_pow`, and `inc_arr` that were reported by the Venus CC checker?
- In RISC-V, we call functions by jumping to them and storing the return address in the `ra` register. Does calling convention apply to the jumps to the `naive_pow_loop` or `naive_pow_end` labels?
- Why do we need to store `ra` in the prologue for `inc_arr`, but not in any other function?
- Why wasn't the calling convention error in `helper_fn` reported by the CC checker? (Hint: it's mentioned above in the exercise instructions.)

Once you have answered these, run Venus with the calling convention checker on `factorial.s` from the last exercise as well. Make sure to fix any bugs you find.

Testing

After fixing the errors in `cc_test.s`, run Venus locally with the above command to make sure the behavior of the functions hasn't changed and that you've remedied all calling convention violations.

Once you have fixed everything, running the above Venus command should output the following:

```
Sanity checks passed! Make sure there are no CC violations.
Found 0 warnings!
```

Exercise 5: 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 in the Venus web simulator, run `list_map.s` and examine the output. 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

In the future, we'll be working with more complex RISC-V programs that require multiple files of assembly code. To prepare for this, we recommend looking over the following sections of the Venus reference:

- Writing Larger RISC-V Programs
- Using the Web Terminal
- Passing Command Line Arguments

Checkoff

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

Checkoff questions for lab 3:

- Make sure you understand and can answer the questions in exercises 1, 2, and 4!