Lab 3 **RISC-V Assembly**

Objective: **Practice running and debugging RISC-V assembly code on Venus**
Write RISC-V functions with the correct function calling procedure

Setup & Introduction

You already have your local repository of this course. You just need to explore this lab using cd and ls commands. Go to fa21-lab-starter/lab03 and list all the exercises.

Introduction to Assembly

In this course so far, we have 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.

For the next few labs, we will work with several RISC-V assembly files, each of which has a .s file extension. To run these, we will be using Venus, an educational RISC-V assembler and simulator. You can run Venus locally from your own terminal or on the Venus website, and the following instructions will guide you through the steps to set it up. Though you may find using the web editor easier to use for this lab, *please go through these instructions for local setup regardless*: these steps will also set up other infrastructure needed for future projects and labs.

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.

Exercise 1: Connecting your files to Venus

You can "mount" a folder from your local device onto Venus's web frontend, so that edits you make within the browser Venus editor are reflected in your local file system, and vice versa. If you don't do this step, files created and edited in Venus will be lost each time you close the tab, unless you copy/paste them to a local file.

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:

- If you don't already have your labs repo cloned on your local machine, open a terminal on your local machine and clone it.
- `cd` into your labs repo folder, and run `java -jar tools/venus.jar . -dm`. This will expose your lab directory to Venus on a network port.
    - You should see a big "Javalin" logo.
    - If you see a message along the lines of "port unable to be bound", then you can specify another 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 (Chrome or Firefox are recommended). In the Venus web terminal, run `mount local labs` (if you chose a different port, replace "local" with the full URL, such as `http://localhost:6162`). This connects Venus to your file system.
    - In your browser, you may see a prompt saying `Key has been shown in the Venus mount server! Please copy and paste it into here.`. You should be able to see a key in the most recent line of your local terminal output; just copy and paste it into the dialog.
- Go to the "Files" tab. You should now be able to see your labs directory under the `labs` 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 Venus locally, 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 2!

Exercise 2: Familiarizing yourself with Venus

Getting started:

1. Open ex1.s into the Venus editor. If you were unable to mount the filesystem in Exercise 1, 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 answers 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 3: 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 cding 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 and identify the following components of this assembly file, and be able to explain how they work.

- 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 4: 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.

The argument that is passed into the function is located at the label n. You can modify n to test different factorials. To implement, you will need to add instructions under the factorial label. Note that you may find it helpful to add additional labels to simplify control flow. You may solve this problem using either recursion or iteration and you can 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 can test this using the online version of Venus, but as promised, we've also provided Venus for you to test locally! Make sure to update your factorial.s file and run the following command before you submit to verify that the output is correct (You will need to run this from the labs directory).

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

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 (how does jalr work?).

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 that takes an int as an argument (you may recall that Philphix makes use of function pointers as well). We can call this function f just like any other.

There are exactly ten (10) markers (1 in done, 7 in map, and 2 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 ten markers with the appropriate code. Furthermore, provide a 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
80 63 48 35 24 15 8 3 0 -1
```

The first line is the original list, and the second line is the list with all elements squared after calling map(head, &square), and the third is the list with all elements decremented after now calling map(head, &decrement).

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 labs 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 Venus reference.

Tasks

- Answer all the questions Exercise 3 and show
- Run and show the output of Exercise 4
- Run and show the output of Exercise 5