CS61C Fall 2018 Lab 3 - RISC-V

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

Like with previous weeks, cd to your repository and then get the lab 3 starter files via:

```
$ git fetch starter
$ git merge starter/master
```

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 (gcc uses a more complex assembly language than RISC-V), 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 a RISC-V simulator. The simulator we will use was developed by Keyhan Vakil (now a CS161 TA) and improved by Stephan Kaminsky (one of your CS61C TAs). The simulator is called **Venus** and can be found online here.

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.
- Programs end with an `ecall` with argument value 10. This signals for the program to exit. The ecall instructions are analagous 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.

Checkoff Checklist:

Have the following files open BEFORE asking for checkoff.

- `lab3_ex1.s`
- `lab3_ex2_c.c`
- `lab3_ex2_assembly.s`
- `factorial.s`
- `list_map.s`

Exercises

Exercise 1: Familiarizing yourself with Venus

Getting started:

1. Paste the contents of `lab3_ex1.s` into the editor or click this magic link to autofill the assembly code into venus.
2. Click the "Simulator" tab. 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 nagivate to different portions of your memory using the dropdown menu at the bottom.

Task: Paste the contents of `lab3_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 using the "Edit" 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.

Checkoff [1/4]

- Demonstrate that you are able to run through the above steps and provide answers to the questions.

Exercise 2: Translating from C to RISC-V

Open the files `lab3_ex2_c.c` and `lab3_ex2_assembly.s`. The assembly code provided (.s file) is a translation of the given C program into RISC-V (Venus Magic Autofill Link).

Task: Find/explain the following components of this assembly file.

- The register representing the variable `k`.
- 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.

Checkoff [2/4]

- Find the section of code in `lab3_ex2.s` that corresponds to the copying loop and explain how **each** line is used in manipulating the pointer.

Exercise 3: Factorial

Task: In this exercise, you will be implementing a function `factorial` in RISC-V that has a single integer parameter `n` and returns `n!`. A stub of this function can be found in the file `factorial.s` (Venus Magic Autofill Link). 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.

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

Checkoff [3/4]

- Display your code for the factorial function, as well as the outputs for the test cases listed above.

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

This exercise uses the file `list_map.s` (Venus Magic Link here).

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".

Task: 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.

Checkoff [4/4]

- Show your TA your test run.