Project 2: RISC-V Assembly

Goals

In this project, you will practice running and debugging RISC-V assembly code. Also, you will write RISC-V functions with the correct function calling procedure, and get an idea of how to translate C code to RISC-V.

Getting the files

To get the starter files for this lab, please refer to the following link:

https://drive.google.com/file/d/1uM8SS04nuAuCLxSceaaxWS7zi7e1kISQ/view?usp=sharing

Intro to Assembly with RISC-V Simulator

In this project, you will be 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.

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 you can find 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.
    - Note: Sometimes, we want to pre-allocate some items in memory before the program starts executing (more on this in Part 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.

Part 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.

Now 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 different 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.

Part 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.

Now 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.

Part 3: Factorial

In this part, 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 can you also use the .jar version to test locally. Note that you will need to have java installed to run this command.

```
$ java -jar venus-jvm-latest.jar factorial.s
```

The .jar version of Venus you downloaded here.

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

This part uses the file `list_map.s`.

Now 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`.

Now 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 venus-jvm-latest.jar list_map.s
```

How to submit your work

Using Google Classroom, you will submit your answers for Parts 1 and 2 in text files named `part1.txt` and `part2.txt`. For Parts 3 and 4, you will need to submit your modified version of `factorial.s` and `list_map.s`