Lab 4

Objectives

- TSWBAT practice debugging RISC-V assembly code.
- TSWBAT write RISC-V functions that use pointers.

RISC-V Simulator

Like last week, we will be using the **Venus RISC-V simulator** (which can be found online here). Also, please refer to the Venus Guide on our course website when you need a refresher on any of the Venus features.

Exercise 1: Debugging `megalistmanips.s`

In Lab 3, you completed a RISC-V procedure that applied a function to every element of a linked list. In this lab, you will be working with a similar (but slightly more complex) version of that procedure.

Now, instead of having a linked list of `int`'s, our data structure is a linked list of `int` arrays. Remember that when dealing with arrays in `struct`'s, we need to explicitly store the size of the array. In C code, here's what the data structure looks like:

```
struct node {
    int *arr;
    int size;
    struct node *next;
};
```

Also, here's what the new `map` function does: it traverses the linked list and for each element in each array of each `node`, it applies the passed-in function to it, and stores it back into the array.

```
void map(struct node *head, int (*f)(int)) {
    if (!head) { return; }
    for (int i = 0; i < head->size; i++) {
      head->arr[i] = f(head->arr[i]);
    }
    map(head->next, f);
}
```

For the purpose of this lab, don't worry too much about the weird syntax for C function pointers (you are welcome to learn more about them here). Basically, you can pass arguments into function pointers just like you do with normal functions.

Action Item

Record your answers to the following questions in a text file. Some of the questions will require you to run the RISC-V code using Venus' simulator tab.

1. Find the five mistakes inside the map function in `megalistmanips.s`. Read all of the commented lines under the `map` function in `megalistmanips.s` (before it returns with jr ra), and **make sure that the lines do what the comments say**. Some hints:
    - Why do we need to save stuff on the stack before we call `jal`?
    - What's the difference between `add t0, s0, x0` and `lw t0, 0(s0)`?
    - Pay attention to the types of attributes in a `struct node`.
    - *Note*: you need only focus on the `map`, `mapLoop`, and `done` functions but it's worth understanding the full program.

2. **For this exercise, we are requiring that you don't use any extra save registers in your implementation**. While you normally can use the save registers to store values that you want to use after returning from a function (in this case, when we're calling `f` in `map`), we want you to use temporary registers instead and follow their caller/callee conventions. **The provided `map` implementation only uses the `s0` and `s1` registers, so we'll require that you don't use `s2`-`s11`.**

3. **Make an ordered list of each of the five mistakes, and the corrections you made to fix them**.

4. Save your corrected code in the `megalistmanips.s` file. **Use the `-cc` flag to run a basic calling convention check on your code locally**:

   ```
   java -jar venus.jar megalistmanips.s -cc // Append appropriate path to `venus.jar`
   ```

   The CC checker should report 0 warnings.

   Again, the Venus Guide is a great resource if you feel unsure about any of the Venus features.

   Note: The CC checker won't check if you are using registers besides `s0` and `s1`, but you need to implement this requirement in order to pass the autograder.

5. For reference, running `megalistmanips` on the web interface should give the following output:

```
Lists before:
5 2 7 8 1
1 6 3 8 4
5 2 7 4 3
1 2 3 4 7
5 6 7 8 9

Lists after:
30 6 56 72 2
2 42 12 72 20
30 6 56 20 12
2 6 12 20 56
30 42 56 72 90
```

Checkpoint

At this point, make sure that you are comfortable with the following. Note that these will not be part of the lab checkoff, but are meant to benchmark how comfortable you are with the material in the exercise.

- You should know how to debug in Venus, including stepping through code and inspecting the contents of registers.
- You should understand how RISC-V interfaces with memory.
- You should understand CALLER/CALLEE conventions in RISC-V.

Exercise 2: Write a function without branches

Consider the discrete-valued function `f` defined on integers in the set `{-3, -2, -1, 0, 1, 2, 3}`. Here's the function definition:

```
f(-3) = 6
f(-2) = 61
f(-1) = 17
f(0) = -38
f(1) = 19
f(2) = 42
f(3) = 5
```

Action Item

1. Implement the function in `discrete_fn.s` in RISC-V, with the condition that your code may **NOT** use any branch and/or jump instructions!
2. Save your corrected code in a file `discrete_fn.s`.

Hint: How do you load a word from a dynamic address?

Checkoff

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