Lab 4 **RISC-V Functions, Pointers**

Objective: **Practice debugging RISC-V assembly code**
Write RISC-V functions that use pointers

Setup & Introduction

You already have your local repository of this course. You have also learned how to mount the file on Venus RISC-V simulator. Mount the lab 4 files as you did with Lab 3. Once you've got discrete_fn.s open, you're ready to move on to Exercise 1!

Exercise 1: 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 discrete_fn.s. Make sure you have it saved locally!

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

Exercise 2: 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. Enable the CC checker in settings, then run the program in the simulator.

Alternatively, you can run Venus locally with the following command:

```
$ java -jar tools/venus.jar -cc lab04/cc_test.s
```

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! cc_test.s:58 mv a0, t0
[CC Violation]: (PC=0x0000008C) Setting of a saved register (s0) which has not been saved! cc_test.s:80 li s0, 1
[CC Violation]: (PC=0x00000094) Setting of a saved register (s0) which has not been saved! cc_test.s:83 mul s0, s0, a0
[CC Violation]: (PC=0x00000094) Setting of a saved register (s0) which has not been saved! cc_test.s:83 mul s0, s0, a0
[CC Violation]: (PC=0x00000094) Setting of a saved register (s0) which has not been saved! cc_test.s:83 mul s0, s0, a0
```

```
[CC Violation]: (PC=0x00000094) Setting of a saved register (s0) which has not been saved! cc_test.s:83 mul s0, s0, a0
[CC Violation]: (PC=0x00000094) Setting of a saved register (s0) which has not been saved! cc_test.s:83 mul s0, s0, a0
[CC Violation]: (PC=0x00000094) Setting of a saved register (s0) which has not been saved! cc_test.s:83 mul s0, s0, a0
[CC Violation]: (PC=0x00000094) Setting of a saved register (s0) which has not been saved! cc_test.s:83 mul s0, s0, a0
[CC Violation]: (PC=0x000000A4) Save register s0 not correctly restored before return! Expected 0x00000A3F, Actual 0x00000080. cc_test.s:90 ret
[CC Violation]: (PC=0x000000B0) Setting of a saved register (s0) which has not been saved! cc_test.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! cc_test.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! cc_test.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 '0x14B7A3FD'.
```

Find the source of each of the errors reported by the CC checker and fix it. 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 basic 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 Items

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 discrete_fn.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 command from the beginning of this exercise 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 3: 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 within 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 six mistakes inside the map function in megalistmanips.s. Read all of the commented lines under the map function in megalistmanips.s 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.
    - Why are there a bunch of newlines printed between the Before and After? Where do we print newlines, and why is that function being run?
    - *Note*: All bugs are within the map function, mapLoop, and done 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 six mistakes in the megalistmanips_answers.txt file , 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 tools/venus.jar -cc lab04/megalistmanips.s
```

Again, the Venus reference 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.

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
```

Exercise 4: Finding and solving the bugs

Coding in RISC-V can be tricky; most of the guardrails that are present in higher level languages are missing here. As such, it is very easy to write code that passes most test cases, but still has bugs in them. This exercise will give practice on finding and solving the most common of these bugs.

The function `accumulator` is defined as follows:

```
Inputs: a0 contains a pointer to an array of nonzero integers, terminated with 0
Output: a0 should return the sum of the elements of the array

Example: Let a0 = [1,2,3,4,5,6,7,0]

Then the expected output (in a0) is 1+2+3+4+5+6+7=28
```

Open the file `lotsofaccumulators.s` In this file, there are five versions of `accumulator`, numbered `one` to `five`. Then go to `accumulatortests.s`. You should see a testing template, with a test already written, that tests the function on the array `[1,2,3,4,5,6,7,0]`. If you replace the `jal accumulatorone` line with the other accumulators, you should notice that all five version pass.

However, only one of the five versions we gave you is actually correct. Your task is to find the bugs in the accumulators, and write tests that pass on the correct version, but fail on the buggy versions.

Notably, the CC checker only catches two of these bugs. This serves as a good lesson: The CC checker does not catch everything, and as such is only a basic check. Thus, you may experience passing locally with no warnings and still fail on the autograder. This is commonly due to the following:

1. incorrect jumps
2. incorrect prologue/epilogue setup
3. missed edge cases in basic tests.

Before running the CC checker on the accumulators, try identifying which ones will pass the CC checker but still fails to follow convention. Because of this, you are still responsible for writing strong test suites, so be sure not to be over reliant on the CC checker to catch all of your calling convention mistakes! For more information on why your code may be failing, take a look at the venus reference.

Action items

Answer the following questions. The solutions are provided to allow you to check your understanding. Please take time to answer the question yourself before looking at the solution.

Find the bugs in four of the five accumulators

- accumulatorone
- accumulatortwo
- accumulatorthree
- accumulatorfour
- accumulatorfive

For each broken accumulator, write a test that fails on the broken one, but passes the correct implementation.

Learnings

At this point, make sure that you are comfortable with the following.

- 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.

Tasks

- Complete all action items listed above in exercises