Lab 4

Computer Architecture I ShanghaiTech University

Goals

These exercises are intended to give you more practice with

- Download source code from here
- calling conventions, including prologue and epilogue.
- function calls and manipulating pointers in RISC-V.
- implementing extended bitwise operations based on existing instructions.

Exercises

Exercise 1: calling convention checker

This exercise uses the file `ex1.s`.

A quick recap of RISC-V calling conventions: 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 `ex1.s` file, particularly at the `main`, `simple_fn`, `naive_pow`, `inc_arr`, and `helper_fn` functions. Enable the CC checker in Venus-Settings-Calling Convention, then run the program in the simulator. You should see an output similar to the following:

```
[CC Violation]: (PC=0x00000080) Usage of unset register t0! editor.S:58 mv a0, t0
[CC Violation]: (PC=0x0000008C) Setting of a saved register (s0) which has not been saved! editor.S:80 li s0, 1
[CC Violation]: (PC=0x00000094) Setting of a saved register (s0) which has not been saved! editor.S:83 mul s0, s0, a0
......
```

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 sanity 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 Item

Resolve all the calling convention errors in `ex1.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.)

Testing

After fixing the errors with `FIXME` in `ex1.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!
```

Checkoff

- Show your TA your code and its test run.
- Provide answers to the questions.

Exercise 2: RISC-V function calling with `map`

This exercise uses the file `ex2.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.

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. 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 `TODO: 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)`.

Checkoff

- Show your TA your test run.

Exercise 3: implementing bitwise reverse operation in two ways

This exercise uses the file `ex3.s`.

Write two versions of a bitwise reverse function that, given a value in `a0`, returns the value in `a0` with the bits reversed, e.g. 01011101 to 10111010.

- In the first version `bitrev1`, a loop is required to generate the reversed value bit by bit.
- In the second version `bitrev2`, only use `li`, `and`, `or`, `slli`, `srli` instructions. Any branch and jump instructions are not allowed (except the last `ret`), and manually expanding the loop in `bitrev1` is also not allowed.
  The solution is a little tricky. You can think of this solution as a kind of divide and conquer. The input is `k` binary bits, and the output is the reversed bits.

  ```
  Algorithm bitwise-reverse( b[0:k] )
      if k==1:
          return the single bit
      b[0:k/2] = bitwise-reverse( b[0:k/2] )
      b[k/2:k] = bitwise-reverse( b[k/2:k] )
      swap(b[0:k/2], b[k/2:k])
      return b[0:k]
  ```

  However, you are not going to implement this recursive algorithm. By using `li` to load several specific constants and `and`, `or`, `slli`, `srli` to modify the bits, the whole procedure can be parallelized. Here is the example:

  ```
  0101-1101
  1101-0101        # step1, k=8

  11-01 01-01
  01-11 01-01      # step2, k=4, parallelized

  0-1 1-1 0-1 0-1
  1-0 1-1 1-0 1-0  # step3, k=2, parallelized
  ```

When you've finished the two functions, running the code should provide you with the following output:

```
0x12345678 0x1E6A2C48 0x1E6A2C48 0x1E6A2C48
0x71C924BF 0xFD24938E 0xFD24938E 0xFD24938E
0x19260817 0xE8106498 0xE8106498 0xE8106498
```

Checkoff

- Show your TA both versions of the function and its test run.