Learning Objectives

1. Sequential logic design
2. Register file implementation

Work that needs to be handed in (via SVN)

By the first deadline

1. Complete a reader for I in `i_reader.v` and corresponding test cases in `i_reader_tb.v`. Just as in Discussion, you should aim to begin in the garbage state. At input 000, you should move into the "blank" state that precedes the sequence that would make an I, *i.e.*, you've got a blank space and then you'll read your I.

By the second deadline

1. `luv_reader.v`: a character recognition finite state machine described in the section handout using the provided module definition. You can just copy over the logic for recognizing I from `i_reader.v` once you've completed that file.
2. `luv_reader_tb.v`: a testbench for testing the iluvu reader. We've provided a shell for this file, but you'll have to design your own tests. *We won't be auto grading your test benches, but the quality of your testing will influence the assignment of partial credit;* i.e.*, no tests, no partial credit.*
3. `rf.v`: a MIPS register file. We've provided the module's interface, shown below.

```
module mips_regfile (rd1_data, rd2_data, rd1_regnum, rd2_regnum,
                     wr_regnum, wr_data, writeenable,
                     clock, reset);

    output [31:0] rd1_data, rd2_data;
    input   [4:0] rd1_regnum, rd2_regnum, wr_regnum;
    input  [31:0] wr_data;
    input         writeenable, clock, reset;

endmodule // mips_regfile
```

   The register file has 2 32-bit read ports (`rd1_data` and `rd2_data`) which are independently controlled by `rd1_regnum` and `rd2_regnum`, and 1 32-bit write port (`wr_data` controlled by `wr_regnum` and `writeenable`). It includes 31 registers; reading register $0 always returns zero. All writes are synchronous, so they should only occur at the clock's rising edge.
4. `rf_tb.v`: a testbench for the register file; handed in, not autograded (see `luv_reader_tb.v`). You should test the following functionality (because we will be...):
   - That when you write a register (any of $1-$31), that future reads of that register (from either read port) return the written value and that the contents of no other register are affected.
   - That reading $0 always returns zero even if you try to write to it.
   - That resetting the register file restores all register values to 0.

Compiling

We have provided you with a Makefile that you will use for the compilation of all of your files. Usage:

1. `make i_reader`: compiles and runs the I reader finite state machine.
2. `make luv_reader`: compiles and runs the character recognition finite state machine.
3. `make rf`: compiles and runs the MIPS register file.
4. `make clean`: removes all executables and .vcd files

Tips

As you work through Lab 4, here are two tips that might be helpful:

1. Code generators: Remember code generators from last week. They are a great tool for repetitive work and may save you some time and prove easier to debug.
2. New Verilog notation: If you're declaring a lot of wires, you can use special notation to avoid having to name them individually. For instance, if we have:

   ```
   wire foo0, foo1, foo2, ..., foo31;
   ```

   we can simplify this by using the following array notation:

   ```
   wire foo [0:31];
   ```

   This is different than declaring `foo` as a single 32-bit bus:

   ```
   wire [31:0] foo;
   ```

   where you can refer to all 32 bits together (using `foo[31:0]` or just `foo`) or some subset of the bits (e.g. `foo[5:2]`) or just individual bits (e.g. `foo[10]`).

   Instead, with the array notation, you are declaring 32 individual 1-bit wires, so you can't refer to them collectively. Notice the `[brackets]` are before the name for buses and after the name for arrays, which is what distinguishes them. It's also conventional to number buses in descending order and arrays in ascending order, to further distinguish them. `foo[0]`, `foo[1]`, etc. are the individual wires of the array.

   You can combine the two notations and get something like

   ```
   wire [31:0] bar [0:15];
   ```

   which declares an array of 16 32-bit buses. Thus, e.g. `bar[3]` refers to a 32-bit bus, and you can do things like `bar[3][5:2]`, `bar[3][10]`, etc. to refer to the bits of the bus if needed. You might find this combination to be useful for this lab.

A 32 x 32-bit MIPS register file

Note: register zero always returns the value 0, as discussed in lecture.