Learning Objectives

1. You will design a finite state machine that controls a simple datapath using sequential logic
2. You will implement a register

Work that needs to be handed in

By the first deadline

1. `register.v`: A single 32-bit register. We've provided the module's interface, shown below.

```
module register(q, d, clk, enable, reset);
    output [31:0] q;
    input [31:0] d;
    input clk, enable, reset;
endmodule // register
```

   The register module has a 32-bit write port (`d`) and outputs the value it holds on a 32-bit read port (`q`). All writes to this module are synchronous, so they should occur only at the clock's rising edge and only when enable is high (Hint: this is the same exact behavior that the dffe already has). You **need** to make this module using the included `dffe` module.

2. `register_tb.v`: a testbench for the register ; handed in, not autograded. You should test the following functionality (because we will be...):
   - When you write to the register when it is enabled, future reads of that register should return the written value.
   - If you attempt to write to the register when it's not enabled, future reads should not be affected by that write.
   - Resetting the register restores the register value to 0.
3. `reg_writer.v`: We have provided you a broken implementation of a "register writer" finite state machine. You need to fix the implementation that we give you so that it follows the finite state machine diagram we give you for this module (it doesn't).

By the second deadline

1. `palindrome_control.v`: A finite state machine that controls the `palindrome_circuit` module.
2. `palindrome_control_tb.v`: A testbench for the same. You should test whether your FSM can correctly detect multiple palindrome calculations in a row without resetting. You will be tested on whether your FSM correctly identifies palindromes for a variety of inputs and register file states.

Compiling

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

1. `make reg_writer`: compiles and runs the `reg_writer` test bench.
2. `make palindrome_control`: compiles and runs the `palindrome_control` test bench.

3. `make register`: compiles and runs the register test bench;
4. `make clean`: removes all executables and v̇cd files

Register Writer

We have given you an implementation of the MIPS register file (`regfile` module in `palindrome_lib.v`) and a finite state machine that controls a register file (`reg_writer` module). The MIPS register file contains 32 registers as described in lecture. The finite state machine has two inputs: `Go` and `Direction`. The finite state machine will begin writing 5 values in sequence starting from register 8 of the register file when `Go` is 1 on a positive clock edge. The machine will then wait in an initialization state until `Go` is 0. Once `Go` is 0, the machine **should** write values into register 8 through register 12 if the `Direction` signal is 1 at the start of the writing run, or into registers 8 down to 4 if `Direction` is 0. The finite state machine has two outputs, a 5-bit signal indicating which register to write to and a `Done` signal that should be 1 only when the finite state machine has finished writing all 5 values into the register file. Finally, the `reset` signal should return the FSM to the garbage state and clear the register file. Figure(1) provides the state diagram for the FSM that solves this problem. We have given you a buggy implementation of the register writer, your task is to fix the implementation.

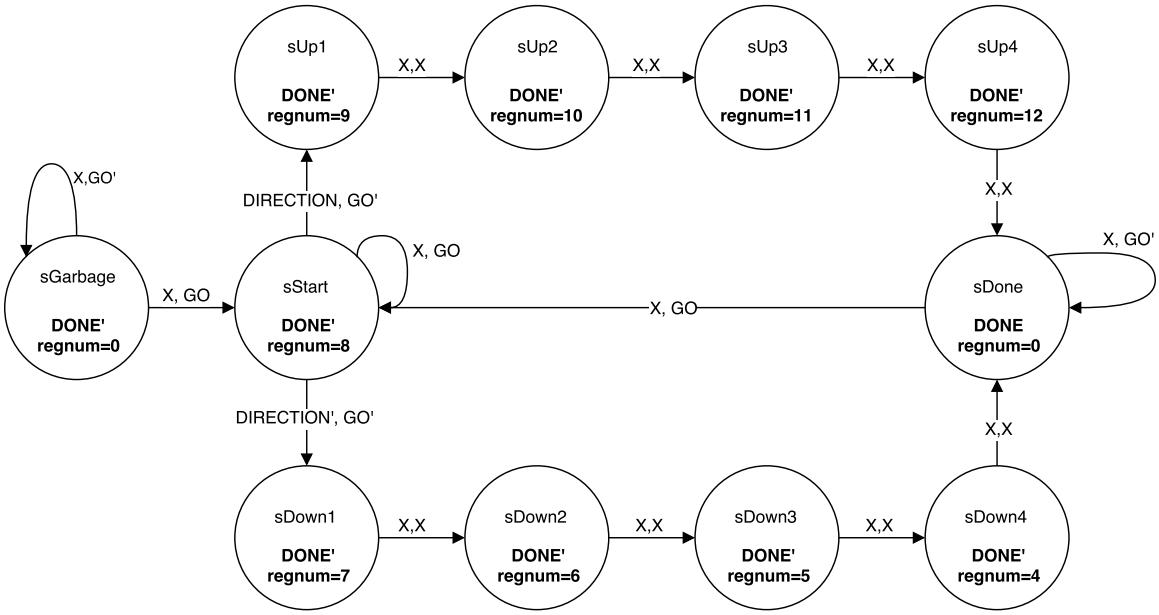

Figure 1. RF Writer FSM

Palindrome

A palindrome is a string that reads the same forwards and backwards such as "toot" or "radar". In this lab, you will design a Finite State Machine (FSM) and datapath that can execute the following C code in hardware. This is good practice for what we will be doing the rest of the semester - implementing C code in assembly language to control a data path.

Example code for testing if a string is a palindrome follows:

```
bool palindrome(char *base, char *end) {
    char *start = base;
    char *ending = end;

    while (start < ending) {
        if (*start != *ending) {
            return false;
        }
        start++;
        ending--;
    }

    return true;
}
```

We can represent this code in hardware by making a circuit and a finite state machine to control this circuit. Note: our definition for a palindrome for building this circuit/FSM is looser than comparing ASCII values. Instead, interpret a palindrome to just be 32-bit integers. An example palindrome with this definition could be `0xFFFFFFFF, 0x12341100, 0xABCD1111, 0x12341100, 0xFFFFFFFF`.

Palindrome Circuit

The datapath (Figure 2) receives two 5-bit data input signals (`base` and `end`), and 1-bit control signals `load` and `select`. The datapath will generate two 1-bit data output signals `front_ge_back`, which is 1 when the register number stored in the front counter is greater than or equal to the register number stored in the back counter, and `a_ne_b`, which is 1 if the contents of the register pointed at by the front counter is not equal to the contents of the register pointed at by the back counter. To manage these counters, this circuit uses two 5-bit registers which are enabled by `load`. To dictate which value gets written to these registers, the `select` signal is used to decide between initial values for each counter (front: `base`, back: `end` or incremented/decremented values accordingly. A comparator circuit (given in `palindrome_lib.v`) is used to produce the `front_ge_back` and `a_ne_b` signals.

The circuit also has a `reset` signal and a `clock` signal. `reset` will clear the registers to 0 and the `clock` signal synchronizes the registers.

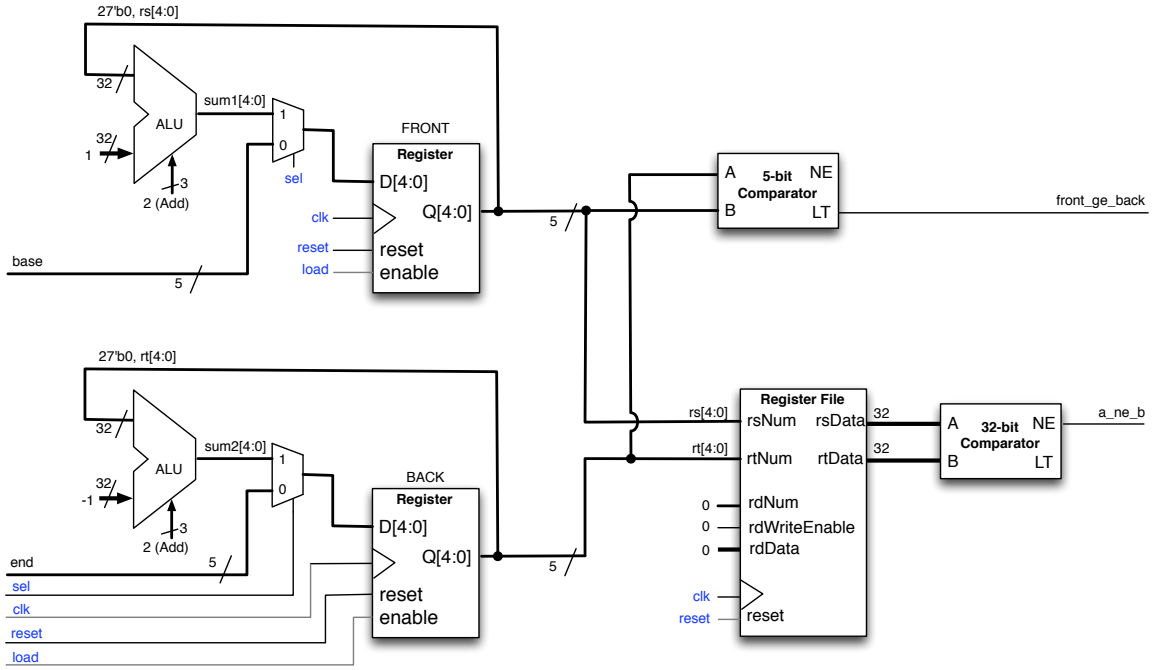

Figure 2. Palindrome Datapath Circuit

Palindrome Control

Your job is to build the finite state machine that controls the above circuit in `palindrome_control.v`. Your machine will receive three inputs. The input signal `go` will come from a user and will become 1 when your circuit should be ready to start detecting a palindrome, dropping to 0 to start a detection run on your `palindrome_circuit`. The FSM will also receive two flag signals (`a_ne_b` and `front_ge_back`) from your `palindrome_circuit`. Your FSM must produce four output signals. Two output signals are the control signals (`select` and `load`) that are sent to `palindrome_circuit`). The `done` signal should be 1 when your algorithm has terminated, with the `palindrome` signal being 1 if the given run contains a palindrome (or 0 if it did not). These signals should maintain their value until you receive receive a new `go` signal, indicating that your machine should be ready to start a new run (with the signal dropping back to 0 to begin the run, as above).

Your FSM should start in a garbage state and should return to the garbage state whenever the reset signal is 1. Your sequential circuit `palindrome_control` should be synchronized with the `palindrome_circuit` using the clock signal.

We provided for you a test bench that goes over three simple cases. Note that in the test bench (as well as the `palindrome_circuit` module), there are no wires attached to the register file for writing data (as writing tests and initializing values for palindrome runs would be painful). Instead, you can use the syntax demonstrated in the testbench for initializing the register file. Example:

```
circuit.rf.r[11] <= 32'h12344321;
circuit.rf.r[12] <= 32'h00000000;
circuit.rf.r[13] <= 32'h00000000;
circuit.rf.r[14] <= 32'h12344321;
```

The above section of behavioral Verilog initializes in the circuit module, the register file's registers 11, 12, 13, and 14 to values 0x12344321, 0x00000000, 0x00000000, and 0x12344321 respectively. Don't worry if this syntax seems unfamiliar, treat is as if you're assigning the value on the right to the register on the left.

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. GTKWave Data Formats: After you "append" wires to your main view, you can right click their names, and set the "Data Format" to be something more useful. (You might find the decimal mode handy if you can't read hex.)
3. 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 befor 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.