Learning Objectives

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

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. `arraySortCheck_control.v`: A finite state machine that controls the `arraySortCheck_circuit` module.

2. `arraySortCheck_control_tb.v`: A testbench for the same. You should test whether your FSM can correctly check multiple arrays for order without resetting. You will be tested on whether your FSM correctly identifies sorted arrays 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 clean`: removes all executables and v̇cd files

2. `make register`: compiles and runs the register test bench;

3. `make reg_writer`: compiles and runs the `reg_writer` test bench.
4. `make arraySortCheck_control`: compiles and runs the `arraySortCheck_control` test bench.

Register

We have provided in register.v a dffe module which has the implementation desired for a one-bit register. Using this dffe module, create a 32-bit register module.

Register Writer

We have given you an implementation of the MIPS register file (`regfile` module in `arraySortCheck_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.

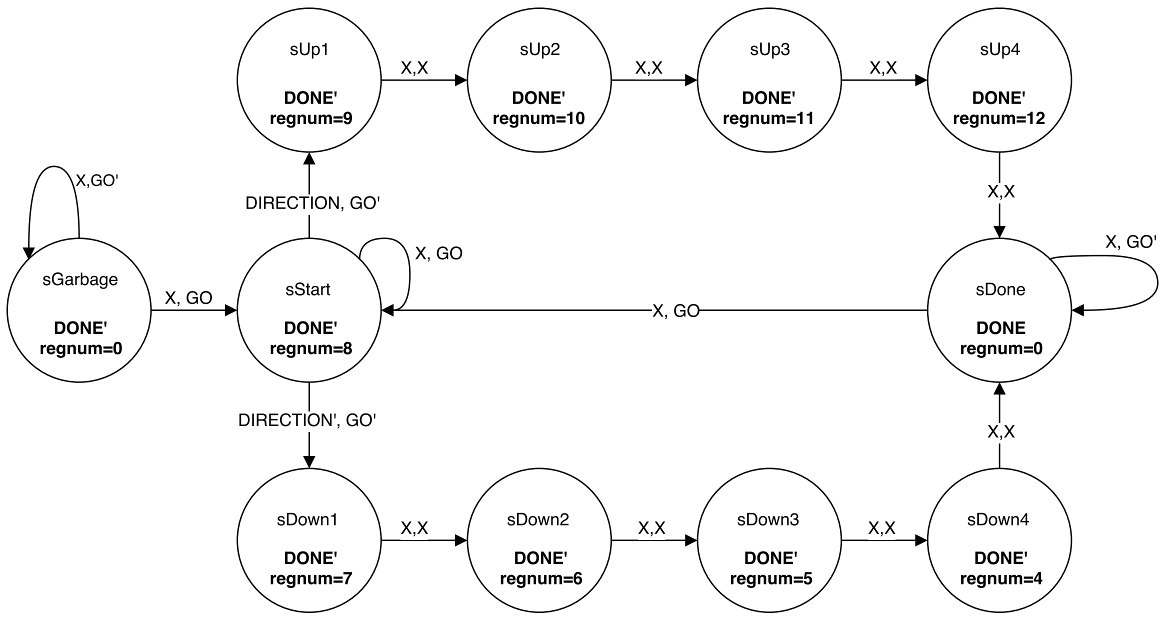

Figure 1. RF Writer FSM

Array Sorting Check

In this lab, you will design a Finite State Machine (FSM) and datapath that can check if an array stored in a register file is sorted or not. The array consists of 32-bit signed numbers. Specifically, it will 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.

Code for testing if an array is sorted in ascending order:

```
bool arraySortCheck(int *array, unsigned int length) {
    unsigned int index = 0;

    if (length != 0){
        while (index < (length - 1)) {
            if (array[index] > array[index + 1]) {
                return false;
            }
            index++;
        }
    }

    return true;
}
```

We can represent this code in hardware by making a circuit and a finite state machine to control this circuit. The circuit is described below.

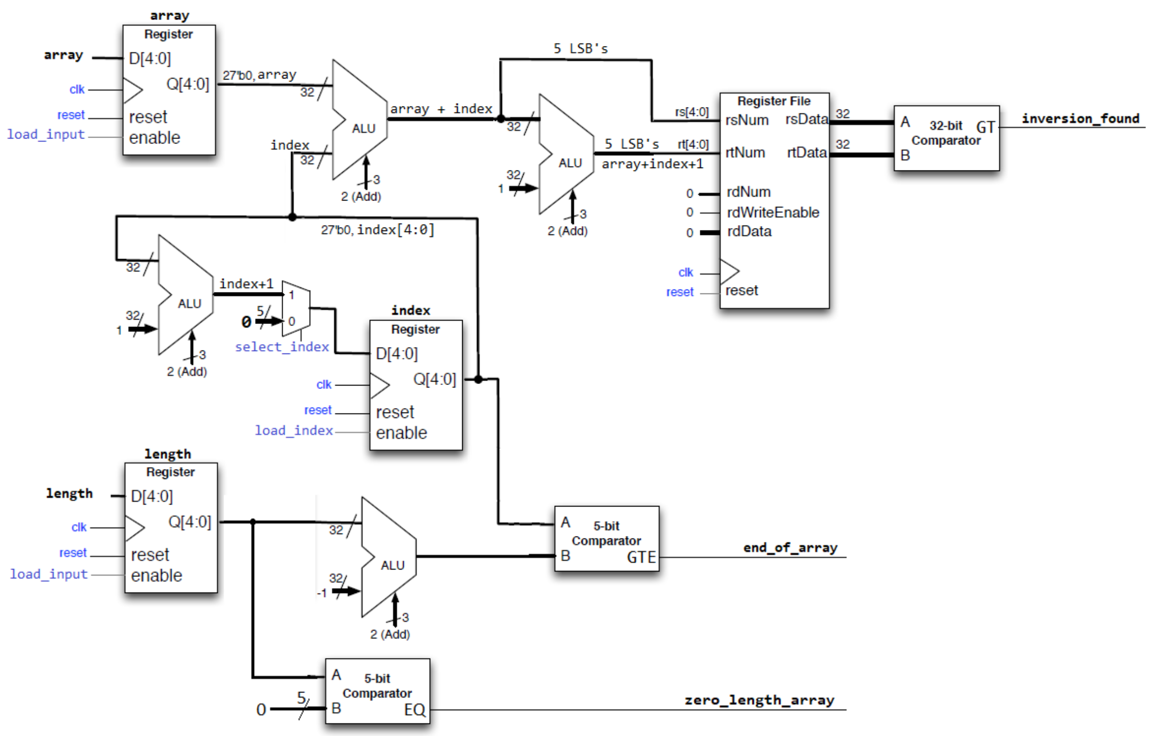

Figure 2. Array Sort Check Datapath Circuit

Array Sort Checking Circuit

The datapath (Figure 2) receives two 5-bit data input signals (array and length), and three 1-bit control signals load_input, load_index and select_index. The 5-bit inputs are stored in the 5-bit registers array and length. load_input signal controls these registers. The register for local variable index is controlled by the two signals load_index and select_index. Specifically, load_index controls when the index register is written and select_index decides what is written to index register. The datapath will generate three 1-bit output signals. The first signal is inversion_found, which is 1 when the array[index] > array[index + 1]. The values array[index] and array[index + 1] are the read outputs of the register file. The second signal is end_of_array, which is 1 when the contents of the register containing index is >= (length - 1). This indicates that the array has been fully traversed. The third signal, zero_length_array, is 1 when the length register is 0. A comparator circuits (given in arraySortCheck_lib.v) are used to produce inversion_found, end_of_array and zero_length_array 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. In our implementation, the reset signal will be synchronous i.e. the reset signal will affect the state on a positive clock edge.

Array Sort Check Control

Your job is to build the finite state machine that controls the above circuit in arraySortCheck_control.v.

Your machine will receive four inputs. The input signal go will come from a user and will become 1 when your circuit should be ready to start checking the array, dropping to 0 to start a checking run on your `arraySortCheck_circuit`. Once a checking run starts, it must be completed irrespective of the changes on the `go` signal. If the `go` signal is 1 at the end of the run, then your circuit should get ready to start another checking run. The FSM will also receive three signals from `arraySortCheck_circuit`. The signals are `inversion_found`, `end_of_array` and `zero_length_array`.

Your FSM must produce five output signals. Three output signals are the control signals (`load_input`, `load_index` and `select_index`) that are sent to `arraySortCheck_circuit`). The `done` signal should be 1 when your algorithm has terminated, with the `sorted` signal being 1 if the given array is sorted (or 0 if it is not sorted). The `sorted` signal should be 0 when it is not indicating the result of the checking run. 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. The transition to "garbage" state should be synchronous i.e. the state should change to "garbage" state on the next clock edge `reset` is set to 1 rather that going to "garbage" state as soon as `reset` is set to 1.

Your sequential circuit `arraySortCheck_control` should be synchronized with the `arraySortCheck_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 `arraySortCheck_circuit` module), there are no wires attached to the register file for writing data (as writing tests and initializing array values would be painful). Instead, you can use the syntax demonstrated in the testbench for initializing the register file. Example:

```
circuit.rf.r[11] <= 32'd1;
circuit.rf.r[12] <= 32'd2;
circuit.rf.r[13] <= 32'd3;
circuit.rf.r[14] <= 32'd4;
```

The above section of behavioral Verilog initializes in the circuit module, the register file's registers 11, 12, 13, and 14 to values 1, 2, 3, and 4 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 3, 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 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.