Lab 2: Review of Finite State Machines (FSMs)

1 Introduction

The purpose of this lab is to review Finite State Machines (FSMs), which are a critical component of hardware design. You will also learn about the proper design of FSMs using control/datapath separation. Another purpose of this first lab is to (re)familiarize you with good hardware design practices, which include good use of module hierarchy and the naming and purpose of signals.

2 Elevator Controller

In this lab, you must design the hardware to control an elevator in a 4-storey building. Each floor of the building has a single elevator call button which a person can press to bring the elevator to that floor. Once inside the elevator, they select the floor they want to go to. For this design, both the call buttons and the buttons inside the elevator are combined into a single input.

You `elevator` module consists of the following inputs and outputs:

1. `i_clock`: Clock input.
2. `i_reset`: Active high asynchronous reset.
3. `i_buttons[1:0]`: The elevator buttons.
4. `o_current_floor[1:0]`: Output from your design showing which floor the elevator is on.

Design overview. We begin by providing an overview of how your design should work.

1. On reset, the elevator will start at the ground floor. You do not need to move the elevator to the ground floor on reset. You can assume it will only be reset when it is already on the ground floor.
2. When the call button is pressed from a floor, the elevator must go to that floor.
3. The `o_current_floor` output should indicate the current floor that the elevator is on. When moving between floors, it should show the last floor it passed.
4. When moving across floors (without stopping at them), it should show the floors it is passing. For example, when going from floor 1 to 3, `o_current_floor` should be 2 when it passes that floor.
5. The call button will remain ON until the elevator reaches that floor. Buttons pressed while the elevator is moving will not change the `i_button` input. For example, if the elevator is on the ground floor and it is moving to the fourth floor, `i_button` will remain 4 until it reaches the fourth floor, even if someone on floors 2 or 3 presses the button.

Using the example of the elevator controller, this lab will show you how to use hierarchical design of modules and how to separate your design into a control path and a datapath.

3 Understanding the template code.

To help you get started, you have been provided two template files on Quercus: `elevator.sv` and `elevator_tb.sv`. The code in these files follows the hardware design shown in Figure 1. We will start by going over the code provided to you in `elevator.sv`.

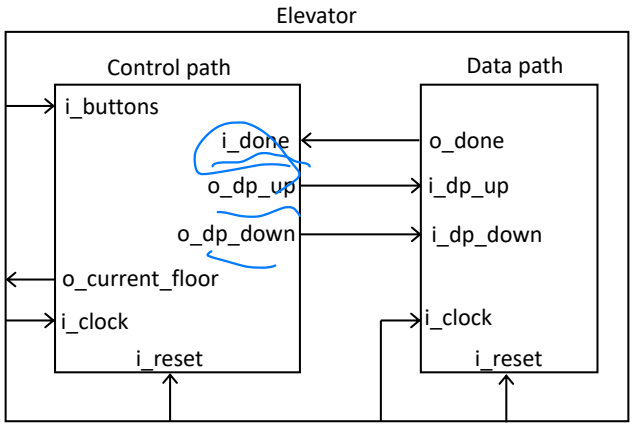

Figure 1: Hardware overview, showing control path, data path and all module inputs and outputs.

3.1 Design template

`elevator.sv` provides the template for your hardware. All the code in this file must be **synthesizable.**

Elevator module: The first module `elevator` is your top level module. This is the module you will instantiate in your testbench. (It is also the module you should set as your `top level module` in Quartus, if you want to run this code on an FPGA.) The `elevator` module contains the top-level inputs and outputs listed above. It then instantiates the `controlpath` and `datapath` for your design. Lastly, this module also declares some signals: `dp_up`, `dp_down` and `done_moving`. These are signals going between the control path and the data path.

Note that all module inputs and outputs are pre-prended with `i_` or `o_` to indicate the signal direction. For example, the `dp_up` signal connects to the `o_dp_up` port of the control path (indicating that it is an output from the control path) and to the `i_dp_up` port of the data path (indicating that it is an input to the data path). This is particularly important for designs with many modules as the output of one module is the input to another module. Note that these directions are also shown on Figure 1. You are strongly encouraged to do the same, for both code and figures, for all future labs as well as on your exams.

Controlpath module: Next, in the `controlpath` module, you will see the code to declare the states of the FSM. Recall that in ECE241, you declared states using the `localparam` command. Here, we use `enum`, introduced in System Verilog instead. Using `enum` has one major advantage; you will see the actual name of each state in the waveform window in ModelSim. If you were to use `localparam`, it would show the actual value (i.e., 0 or 1) assigned to each state, instead of the name of the state (i.e., `S_G`). If you use descriptive state names (e.g., `S_START, S_DONE`), seeing these in the waveform window can help you during debug.

The next `always_ff` block changes your state on each clock cycle. You will not need to change this code. Lastly, the `always_comb` block calculates your next state combinationally. You will list all the states in the `case` statement and write logic to calculate the next state as well as the FSM outputs here. The assignments at the top of the always block are the default assignments. Recall that you must have default statements in combinational blocks to avoid inferring latches. So any outputs that are not set inside a case statement will be set to this default value. When you have many outputs from an always block, this can result in much neater (and more readable) code compared to writing out every output inside every case statement.

Datapath module: In a real elevator, the datapath would be the signals to the motor to move the elevator. The datapath output would be a 'done' signal to indicate that the elevator has moved to the required floor. We mimic this using a counter. To reduce simulation time, this has been set to 5 cycles per floor. We set the counter to 0 on reset or when the control path signals the datapath to move a floor. Then, after 5 cycles, the datapath asserts the `o_done` signal which tells the control path that the elevator has finished moving. **You should not modify the provided datapath code.**

3.2 Testbench template

`elevator_tb.sv` provides the template for your testbench. The structure of the testbench code should be familiar to you from Lab 1, apart from the `task` block which we explain below. At the top, we declare signals to drive the inputs and test the outputs of the hardware being tested. We instantiate the hardware, generate a clock and start by resetting the design for 1 cycle. We then need to change the inputs to our design and check if we get the right outputs. To properly test a design, you should provide a variety of inputs (including corner cases) to make sure your design works robustly. In the case of the adder we saw in Lab 1, we can use a for loop to generate many possible inputs. However, in many cases this is not possible; either because the values do not cover a linear range or because testing every possible combination would take too much time. In such cases, we want to test certain specific inputs, one after the other. For example, we could write the following code, starting at line 52:

```
$display("Pressing button on floor 1.");
i_buttons = 2'd1;
repeat(10) @(posedge i_clock);
if (o_current_floor !== floor)
  $display("ERROR: Elevator should be floor %d, but was on floor %d instead.",
           floor, o_current_floor);
else
  $display("Elevator correctly moved to floor %d.", o_current_floor);
end
```

This would move the elevator from the ground floor to the first floor and check if it had moved correctly. Since we want to wait some cycles for the elevator to finish moving, we need to add several `@(posedge i_clock)` commands. The `repeat()` command is a shortcut to do this, as it repeats the command given right afterwards several times. In this case, we repeat the `@(posedge i_clock)` command 10 times to wait 10 cycles. You may be wondering why we wait 10 cycles? As mentioned above, the datapath takes 5 cycles to move between floors. However, there may be delays due to state changes in your FSM. So, just to be safe, we wait 10 cycles to check if the elevator reached the right floor. Note that this is 10 cycles per floor. Thus if you move from the ground floor to say the 3rd floor, you will need to wait at least 15 cycles. So you should adjust this delay accordingly.

You will see that we will have to repeat this section of code for each floor we want to move the elevator to. This can quickly make your testbench very long (and increase the chance of mistakes). To prevent this, we can use `task` blocks, which function just like functions in C++.

Tasks Tasks allows us to repeat the same operations many times, with different inputs. The `move_floor` task takes one input: the floor to move to. The code in the task is very similar to the code shown above. The only difference is the code involving `floor_change`. This is to set the appropriate delay for the elevator to move from one floor to any other floor. Since floors can be moved up or down, we need the absolute value of floors being moved. Lines 30 and 31 calculate the absolute value of floors moved. Using this, we can wait the appropriate number of cycles to see if the elevator has reached the right floor.

One key difference between tasks and functions in C++ is that tasks can be `static` or `automatic`. `Static` tasks use the same memory each time they are called. Thus, any variables declared in them retain their values when the `task` is called again. In contrast, `automatic` tasks work like functions in C++. The variables in their scope are only live within the task itself. This makes `automatic` tasks more 'intuitive' to use, from a programming perspective.

Note that System Verilog also has `functions`, which operate differently from tasks. So if you want to use blocks which are similar to C++ functions in your testbenches, stick to tasks.

4 Implementing your design

Now that we have gone through the provided code, you should think about how to implement the FSM. The template has four states: `S_G, S_1, S_2` and `S_3`, corresponding to the four floors. You are, however, free to add more states as needed. Here is a suggested sequence of steps to work on your design:

1. First, think about what type of FSM (Mealy or Moore) you would like to build for this design. **Be prepared to justify your choice of FSM to your TA during marking.**
2. Next, draw the FSM state diagram showing all the inputs and outputs. Even if your FSM changes as you work on your design, you should always have an initial state diagram before you start coding.
3. You may wish to start with a design involving just two floors to start with. Once you get that working, you can then expand it to more floors.
4. Test as you go! Do not write up all your code before running the testbench.
5. Make sure to test corner cases to ensure your design is robust.

5 In-lab demo

(6 marks) Show that your elevator design works using appropriate test cases.

5.1 (OPTIONAL) FPGA pin mappings

To run this code on an FPGA, you can use either a KEY as a clock and manually press the key to toggle the clock, or you can use `CLOCK_50` as a clock. However, with `CLOCK_50` the changes will happen too fast for you to see, so it is preferable to use a KEY as a 'manual clock' for this design. You can also decrease the count in the datapath to say 2 cycles to speed things up, when using KEY as a clock.

Signal name	i_clock	i_reset	i_buttons	o_current_floor
FPGA port	KEY[0]	KEY[3]	SW[1:0]	HEX0

Table 1: Signal mapping for Part I