ECE 510: Foundations of Computer Engineering

Project 1: Elevator Operation

Nick Name: ….Maxy…. **Date: …2020/10/24…**

First Name: ...Maoqin... **Last Name: ….Zhu.....**

1. Problem Statement

This assignment requires an elevator operation to be simulated in Verilog. There are 4 floors between which the elevator can move and there are buttons on each floor, all external to the elevator. A button press implies certain action to be taken, which are described in detail in the following sections. The elevator system can be modeled as a Finite State Machine that undergo state transitions and produces outputs depending on the inputs provided. Design the Finite state machine and describe its behavior either by using a state diagram or a table. Implement the system according to the specifications mentioned below, in Verilog and submit the code as well as the waveforms along with a detailed write up describing the same.

1.1 Input description

There are 6 inputs to the system in addition to a clock signal. The clock signal is used for synchronization purposes. The button 1U is in the first floor, 2U & 2D in the second floor, 3U & 3D in the third floor, and 4D in the 4_{th} floor. A button press and its implication is tabulated below.

Note that the switches with the same name at different floors are electrically connected together. So, if, for example, 1U is pressed at any floor just one signal will be active and be fed to the elevator controller.

INPUT (Button Press)	IMPLICATION
1U	Person in the first floor wants to go to the second floor
2U	Person in the second floor wants to go to the third floor
3U	Person in the third floor wants to go to the fourth floor
2D	Person in the fourth floor wants to go to the third floor
3D	Person in the third floor wants to go to the second floor
4D	Person in the second floor wants to go to the first floor

1.2 Output description

There are 3 outputs indicating the action that is to be taken and is tabulated below.

OUTPUT (elevator action)	IMPLICATION
UP	elevator moves in the upward direction
DOWN	elevator moves in the downward direction
STAY	elevator stays in the same floor

1.3 System behavior

The elevator can be in any of the 4 floors; Floor 1, Floor 2, Floor 3, or Floor 4. At each clock cycle, depending on whether a button is pressed or not, certain action has to be taken which is tabulated below.

Current Position of elevator	Input	Action sequence to be taken
First floor	1U	Move the elevator to the Second floor
	2U	Move the elevator to the Second floor and then to the Third
	3U	Move the elevator to the 3rd floor and then to the 4th
	2D	Move the elevator to the Second floor and then to the First
	3D	Move the elevator to the Third floor and then to the Second
	4D	Move the elevator to the 4th floor and then to the 3rd
Second floor	1U	Move the elevator to the First floor and then to the Second
	2U	Move the elevator to the Third floor
	3U	Move the elevator to the 3rd floor and then to the 4th
	2D	Move the elevator to the First floor
	3D	Move the elevator to the Third floor and then to the Second
	4D	Move the elevator to the 4th floor and then to the 3rd
Third Floor	1U	Move the elevator to the First floor and then to the Second
	2U	Move the elevator to the Second floor and then to the Third
	3U	Move the elevator to the 4th floor
	2D	Move the elevator to the Second floor and then to the First
	3D	Move the elevator to the 2nd floor
	4D	Move the elevator to the 4th floor and then to the 3rd

Fourth Floor	1U	Move the elevator to the First floor and then to the Second
	2U	Move the elevator to the Second floor and then to the Third
	3U	Move the elevator to the 3rd floor and then to the 4th
	2D	Move the elevator to the Second floor and then to the First
	3D	Move the elevator to the 3rd floor and then to Second floor
	4D	Move the elevator to the the 3rd floor

1.4 Processing multiple inputs: Buffer

In order to account for the cases when multiple buttons are pressed at the same time or one after the other, we need to have a mechanism to keep track of the input requests made (a buffer) and means to prioritize them. In order to simplify the code, consider the following to be the order of priority: 1U, 2U, 3U, 2D, 3D, 4D.

For instance, consider the following requests to be made one after the other: 2U, 3D and 1U. The order of processing them would be as follows: 1U, 2U and 3D irrespective of the current position of the elevator.

While designing the mechanism to keep track of the input requests made, note that the order of arrival of requests and the current position of the elevator is irrelevant in deciding the order of execution. This is because our system is supposed to follow the order of priority mentioned above. Also ignore an input request if the same request is already present in the buffer.

2. Way to approach

In this section, I'll talk about my appoach for solving the problem. The steps I took to find the result will be explained as follow.

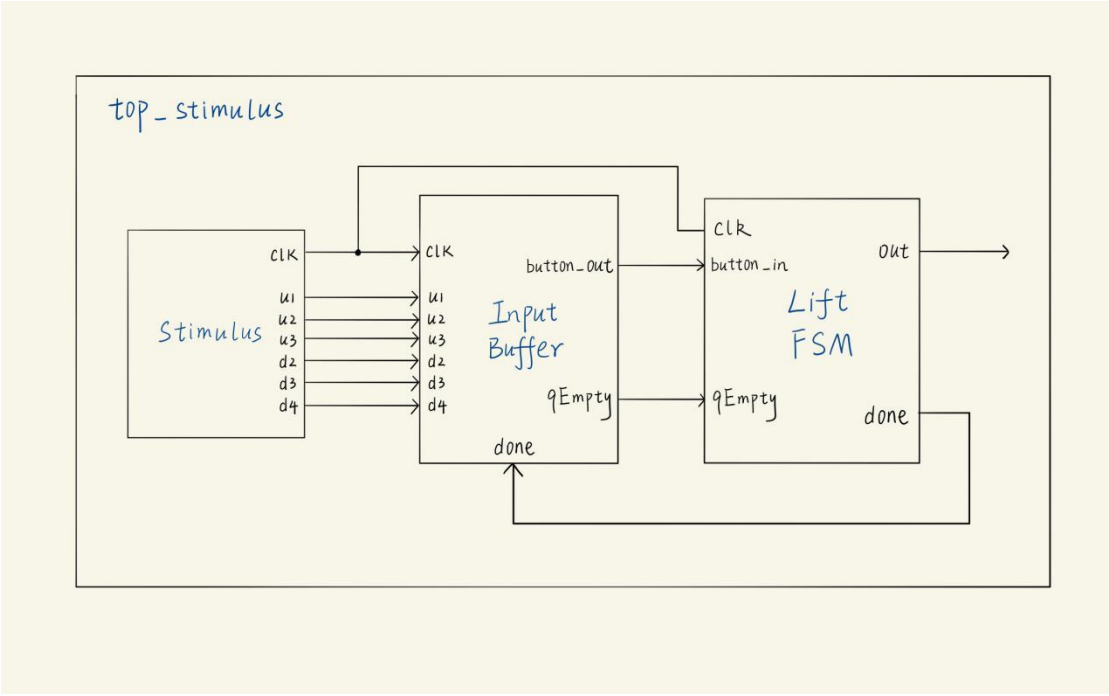
2.1 Separate Module for Lift FSM and Queue and stimulus

Firstly, it is better to have separate modules for Lift FSM, Input Buffer and Stimulus. Then we implement the first two modules separately and using the last module to make a connection between the modules above. The main function of each module is listed below.

• LiftFSM Module : Keeping track of the current floor of the lift, accepting inputs one by one from the InputBuffer Module, making corresponding state transitions and generating the required output.

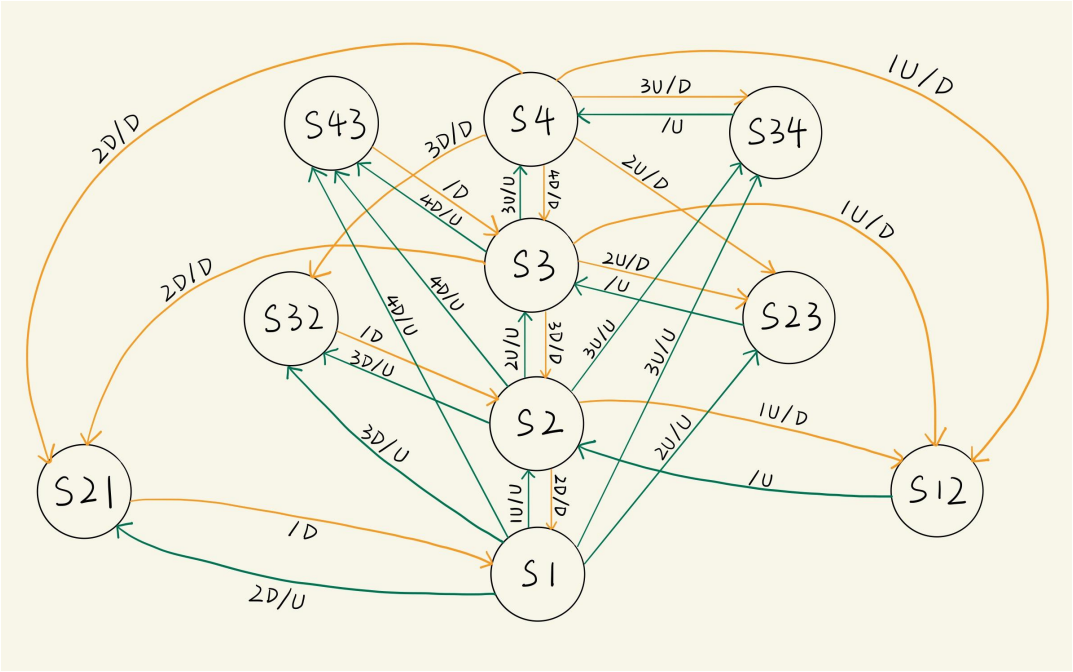
• InputBuffer Module : Keeping track of all the pending inputs to be processed(if any), prioritizing the inputs and providing one input at a time to the LiftFSM Module for it to be processed.

• Stimulus(Tesebench) : Instantiating the above two modules, generating the clock signal and the inputs for the system.



2.2 Draw state diagram and code it in Verilog

Secondly, I start by designing the state machine that models the behavior described in the specifications. Implement the state machine in Verilog and verify its behavior. The state gram is given as follow:



There are totally 10 states. I use the green arrow to indicate “up” output and use the orange arrow to indicate “down” output. The system can be implemented as a Finite State Machine with idle states - S1 , S2, S3, S4 which indicate that the current position of the lift is in the first, second, third, or the fourth floor, respectively and that the system is ready to process the next input. In addition to these, the system also requires busy states – S12, S21 S23, S32, … which indicate that the system is currently responding to a particular input and hence cannot process the next input in this state promptly.
And then, I implement the state machine in Verilog and verify its behavior. Partial verilog code for module LiftFSM is given below. The complete code please check the file “LiftFSM.v” in other folder.

```
//Define module parameters for each state in this FSM

localparam s1=4'b0000;

localparam s2=4'b0001;

localparam s3=4'b0010;

localparam s4=4'b0011;

localparam s12=4'b0100;

localparam s23=4'b0101;

localparam s34=4'b0110;

localparam s21=4'b1000;

localparam s32=4'b1001;
```

```
                                localparam s43=4'b1010;
                        //Define module parameters for output in this FSM
                                 localparam STAY=2'b00;
                                localparam UP=2'b01;
                                localparam DOWN=2'b10;
//Initialize parameters in this module
initial begin
      ps=s1;
      ns=s1;
      out=STAY;
end
always @ (qEmpty or ps or button_in) begin
       //Stay in the same state when stystem is idle and no further input requests
      if(qEmpty && done) begin
            ns=ps;
            out<=STAY;
      end
      else begin
            case (ps)
            //Describe the behavior according to the state diagram
            s1: begin
                  case (button_in)A
                  6'b000001: begin   //button 1U
                        ns=s2;
                        out=UP;
                        done=1;
                  end
                  6'b000010: begin   //button 2U
                        ns=s23;
                        out=UP;
                        done=0;
                  end
                  6'b000100: begin   //button 3U
                        ns=s34;
```

```
                out=UP;
                done=0;
            end
            6'b001000: begin   //button 2D
                ns=s21;
                out=UP;
                done=0;
            end
            6'b010000: begin   //button 3D
                ns=s32;
                out=UP;
                done=0;
            end
            6'b100000: begin   //button 4D
                ns=s43;
                out=UP;
                done=0;
            end
            default: begin        //other invalid input request
                ns=s1;
                out=STAY;
                done=1;
            end
            endcase
        end
    .....
    .....
    end
```

2.3 Design input buffer and code it in Verilog

Then I proceed to implement the buffering system and verify its behavior as before. The conventional buffer consists of FIFO structure. However, there also exist different ways to implement this buffer module. I select a way only using basic verilog syntax such as case, if, if else (The key questions about the

buffer module will be discussed in section3). In this way, the amount of parameters and the number of code line is small. The complete code for this part please check the file “InputBuffer.v” in other folder.

2.4 Design a top module to connect above two modules

Once that is done, integrate the elevator state machine and the buffering system. Decide on the number of control signals that would be needed to coordinate both and verify the behavior of the entire system by applying the necessary stimuli.

Two control signals - done and qEmpty are required to coordinate between the LiftFSM and InputBuffer module.

• done – This signal has to be generated by the LiftFSM module and received by the InputBuffer module when the LiftFSM module has finished processing an input and is ready to process the next. In other words, when the system is in an Idle state.

• qEmpty – This signal has to be generated by the InputBuffer module and received by the LiftFSM module when there are no further input requests to be processed. This will enable the LiftFSM module to stay in the same state and produce the ‘STAY’ output.

Here we should instantiate the above two modules and generate the clock signal, the inputs signals for the system. The complete code for this part please check the file “top_stimulus.v” and file “Stimulus.v” in other folder.

2.5 Use testbench for simulation in Quatus and take wave screenshots

Finally, I set “top_stimulus.v” as the top-level entity of this project. And then set “Stimulus.v” as the testbench file. Take screenshots of those waveform before analyzing the outcomes (The details will be shown in section4).

3. Problems

In this section, I'll talk about problems I faced and my solution to them.

3.1 How to implement functions of InputBuffer?

On balance, there are two main requirements for this buffer:

1) Follow the order of priority: 1U, 2U, 3U, 2D, 3D, 4D.
2) Ignore an input request if the same request is already present in the buffer.

For the first one, what comes to my mind is using "if...else" syntax. However, when we consider the second requirement at the same time. That's not so easy to come up with a solution at once.

After looking through different buffer samples on the websites and discussing with other ECE510 classmates, one way makes sense, that is

a) Set the size of buffer register as 6-bit, because there are 6 input buttons in total. b) Record input requirements in this clock cycle according to the button with "if" syntax.

Partial verilog code for module InputBuffer is given below. The complete code please check the file "InputBuffer.v" in other folder.

```
    //Specify primary datatype in Verilog for parameter inputbuffer which is 6 bits
    reg [5:0] inputbuffer;
.....
    always @ (posedge clk) begin
    //Reset the inputbuffer parameters at positive clk edge if last request has been processed
        case (button_out)
            6'b000001:    inputbuffer[0]=0;
            6'b000010:    inputbuffer[1]=0;
            6'b000100:    inputbuffer[2]=0;
            6'b001000:    inputbuffer[3]=0;
            6'b010000:    inputbuffer[4]=0;
            6'b100000:    inputbuffer[5]=0;
        endcase
        //Record all the inputs in this clock cycle according to the button
        if(u1) inputbuffer[0]=1;
        if(u2) inputbuffer[1]=1;
        if(u3) inputbuffer[2]=1;
```

```
        if(d2) inputbuffer[3]=1;
        if(d3) inputbuffer[4]=1;
        if(d4) inputbuffer[5]=1;
        //Deliver input request to LiftFSM module in order of priority:1U, 2U, 3U, 2D, 3D, 4D
        if(done) begin
            if(inputbuffer[0])          button_out=6'b000001;
            else if(inputbuffer[1]) button_out=6'b000010;
            else if(inputbuffer[2]) button_out=6'b000100;
            else if(inputbuffer[3]) button_out=6'b001000;
            else if(inputbuffer[4]) button_out=6'b010000;
            else if(inputbuffer[5]) button_out=6'b100000;
        end
    end
```

3.2 Is FIFO structure simple enough?

At the beginning, I coded to implement a FIFO buffer. However, since there were supposed to create several new parameters and code for this module was not actually short, my logic seemed not so clear.

By learning from different buffer samples and asking predecessors for help, the, as you can see, the way I choose to design a buffer does not look like a conventional FIFO structure buffer. There is no reading pointer or writing pointer. With the simple syntax, it still can satisfy the requirements of buffer function.

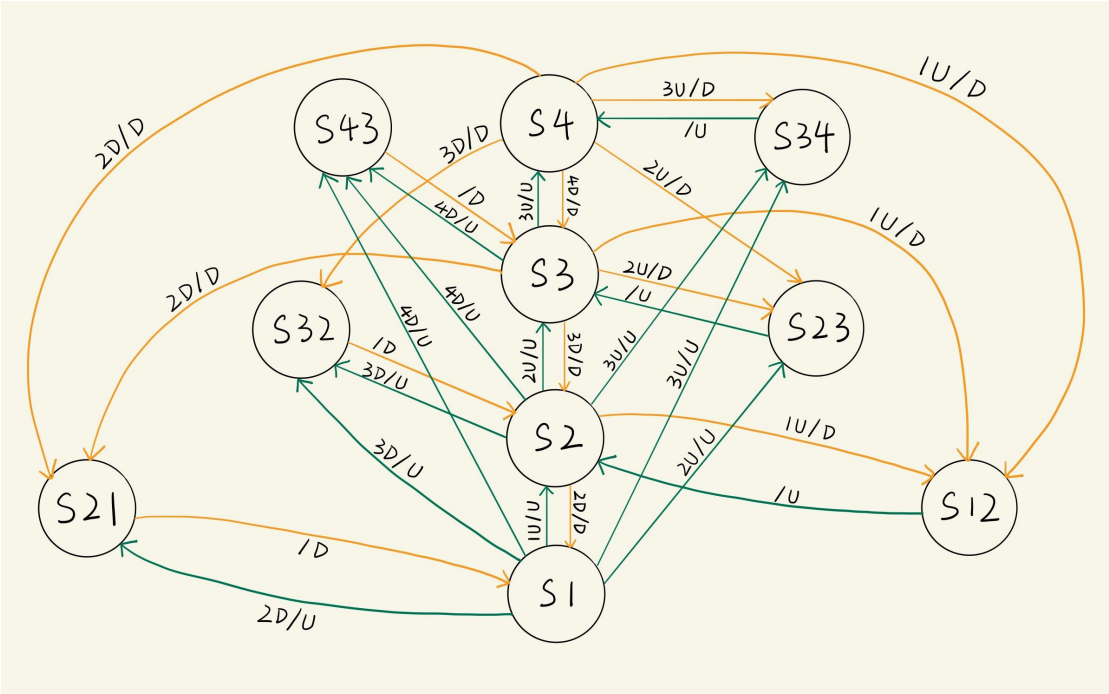
4. Analysis

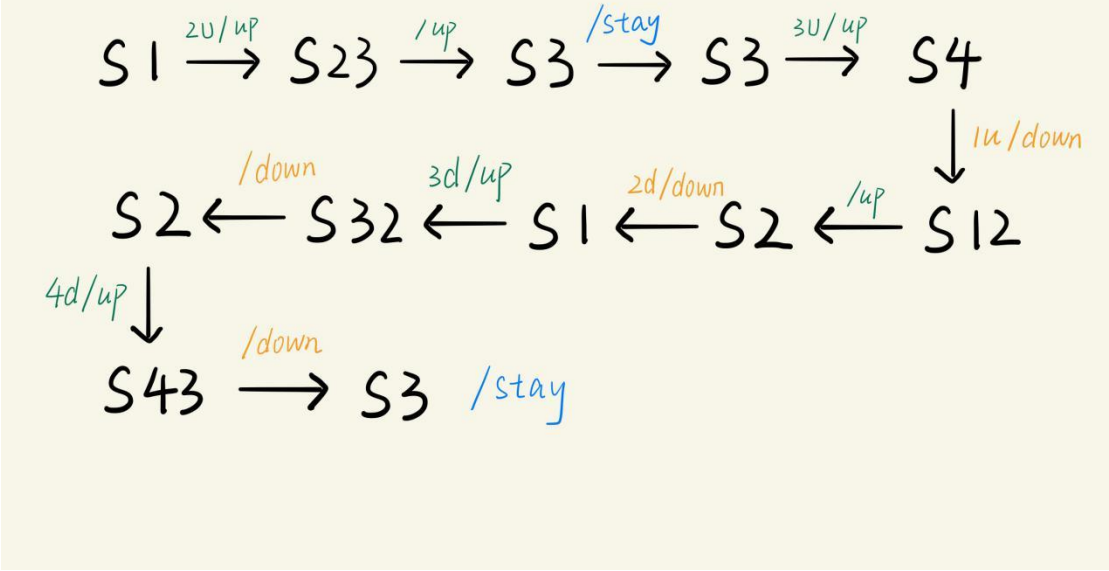
In this section, I'll show you the procedure that I used to test my solution to show it works properly.

Firstly, we generate and initialize clock signal and input signals for the simulation. The partial code for testbench is given below. The complete code for this part please check the file "Stimulus.v" in other folder.

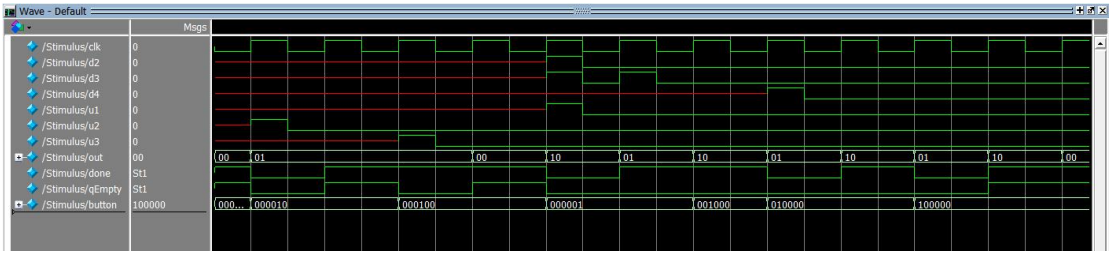
```
//Declare clock cycle
parameter CYCLE = 2;
.....
//Generate and initialize the clock signal
initial begin
    clk = 0;
    forever
    #(CYCLE/2)
    clk=~clk;
end
//Initialize input signals for the simulation
initial begin
    #1      u2=1;
    #1      u2=0;
    #3      u3=1;
    #1      u3=0;
    #3      u1=1;
            d2=1;
            d3=1;
    #1      u1=0;
            d2=0;
            d3=0;
    #1      d3=1;
    #1      d3=0;
    #3      d4=1;
    #1      d4=0;
$display("Stimulus has already finished.");
end
```

In this case, we can transfer the state according to the state diagram. The state transfer behavior of this FSM in every positive clock edge is shown as follow.





The screenshots of wave diagram is shown as follow:



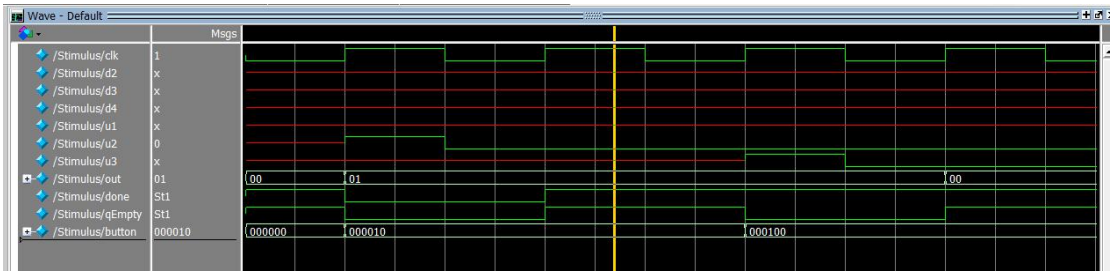
Especially, I wanna point out some interesting clock cycle here to show my solution works properly.

At the 1st clk cycle: S1 to S23

Only button 2U sends input request.

Out indicates up and qEmpty is 0.

Since it moves to S23, done is 0.

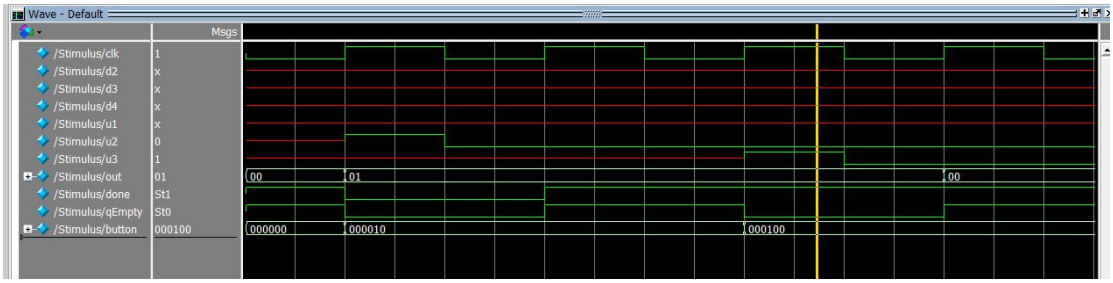


At the 2nd clk cycle: S23 to S2

No button sends input request.

Out indicates up and qEmpty is 1.

Since it moves to S2, done is 1.



At the 3rd clk cycle: S2 to S4

Only button 3U sends input request.

Out indicates up and qEmpty is 0.

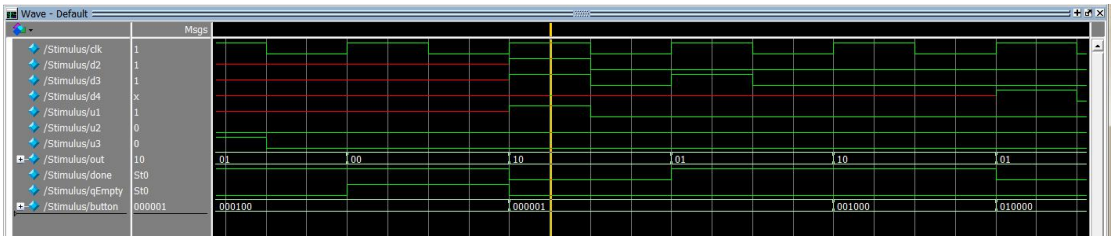
Since it moves to S4, done is 1.

At the 4th clk cycle: stay at S4

No button sends input request.

Out indicates stay and qEmpty is 1.

Since it stays at S4, done is 1.



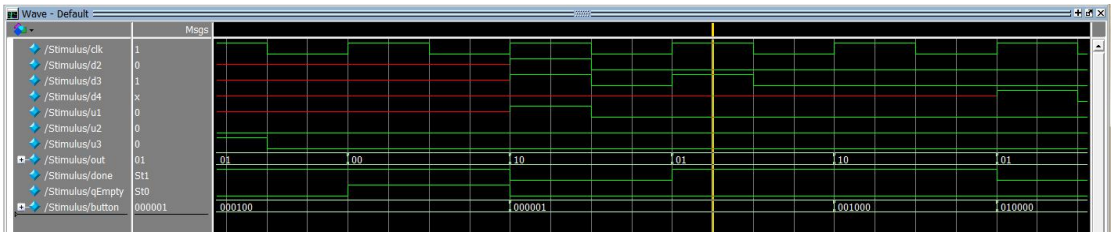
At the 5th clk cycle: S4 to S12 (Succeed in implementing the order of priority)

2D button, 3D button and 1U button send input requests at the same time.

Considering the following order of priority: 1U, 2U, 3U, 2D, 3D, 4D, the input request from 1U is being processed at first, and input requests from 2D and 3D is waiting in the buffer in order.

Thus, out indicates down and qEmpty is 0.

Since it moves to at S12, done is 0.



At the 6th clk cycle: S12 to S2 (Succeed in implementing the order of priority and ignoring the same request which is already in the buffer)

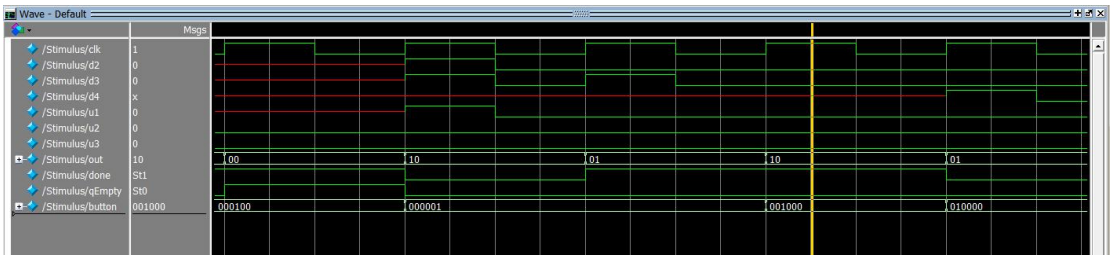
3D button sends input requests again.

Since we design a mechanism ignoring an input if the same request is already present in the buffer, this 3D input is invalid.

Input requests from 2D and 3D is waiting in the buffer in order.

Thus, out indicates down and qEmpty is 0.

Since it moves to at S2, done is 1.



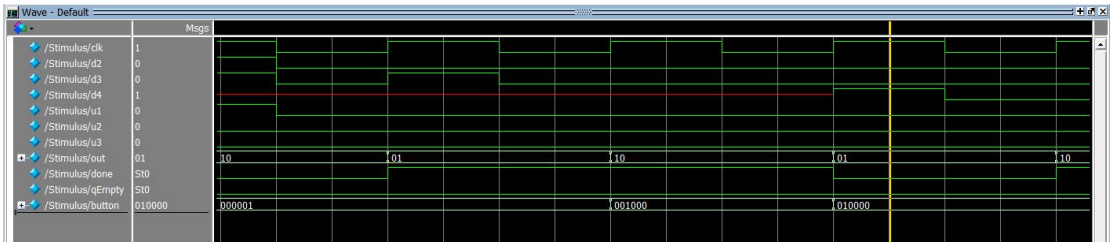
At the 7th clk cycle: S2 to S1

No button sends input requests.

Considering the following order of priority: 1U, 2U, 3U, 2D, 3D, 4D, the input request from 2D is being processed at first, and input request from 3D is still waiting in the buffer.

Thus, out indicates down and qEmpty is 0.

Since it moves to at S1, done is 1.



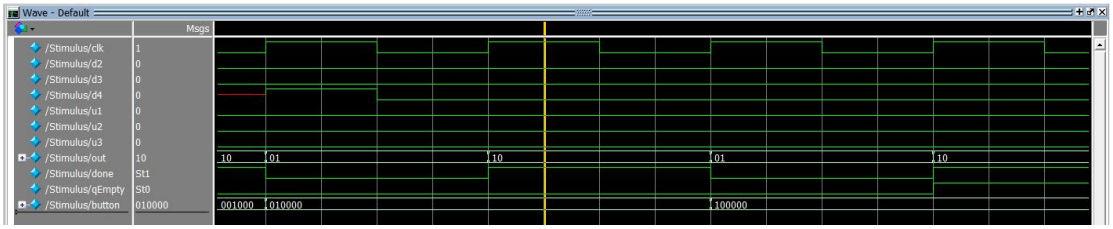
At the 8th clk cycle: S1 to S32

4D button sends input request which is supposed to be placed in buffer in order.

Input request from 3D in the buffer can be processed in this clk cycle.

Thus, out indicates up and qEmpty is 0.

Since it moves to at S32, done is 0.



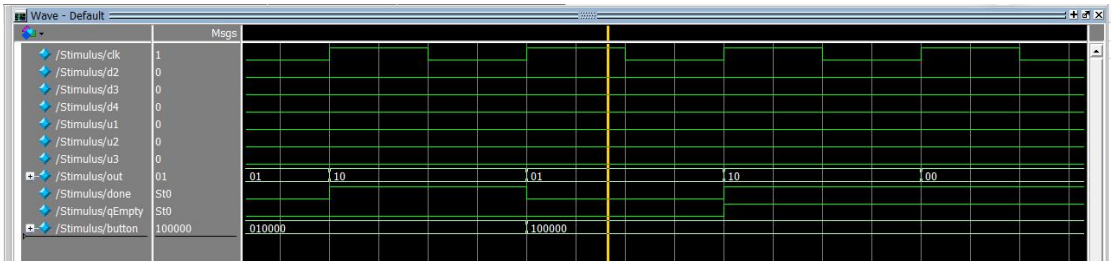
At the 9th clk cycle: S32 to S2

No button sends input request.

Input request from 4D is still in the buffer.

Thus, out indicates down and qEmpty is 0.

Since it moves to at S2, done is 1.



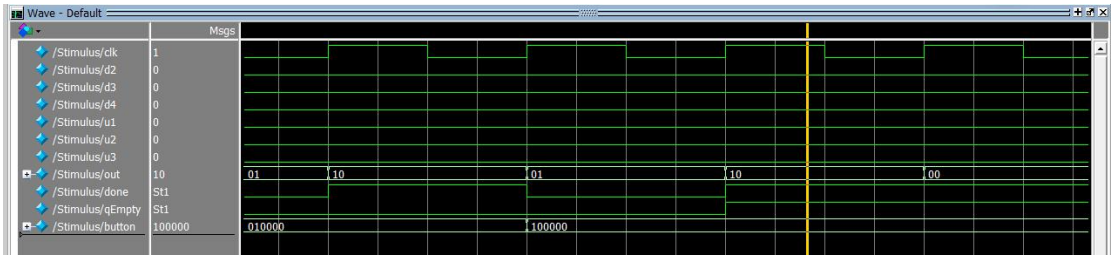
At the 10th clk cycle: S2 to S43

No button sends input request.

Input request from 4D in the buffer can be processed in this clk cycle.

Thus, out indicates up and qEmpty is 0.

Since it moves to at S43, done is 0.



At the 11st clk cycle: S43 to S3

No button sends input request.

Thus, out indicates down and qEmpty is 1.

Since it moves to at S3, done is 1.

As for the following clk cycles: stay at S3

No button sends input request.

Also, qEmpty and done are both 1.

It keeps stay.

The detail description of each clock cycle is given in the file “Screenshots of Testbench output waveforms” in other folder. Please check that file for more details.

5. Conclusion

This project assignment actually give me an experience in Verilog by implementing a Finite State Machine using of a simplified elevator operation. It also expose me to buffering and serialization when multiple elevator buttons are pressed simultaneously. Thus after this project I get familiar with the FSM design, the buffer design and the verilog coding.

For the first part, I actually use what I learned in the ECE510 live lecture. I design the state machine that models the behavior described in the specifications. Implement the state machine in Verilog and verify its behavior.

And then, I'm looking into different buffer designs. By comparing conventional FIFO buffer and my implementation of buffer, I get a deeper understanding of this module. In the process, implementing various functions of buffer is something sort of challenging for me. By searching on the Internet and discussing with classmates, we finally solve the problem in our own way.

Finally, I would say I'm getting more familiar with the operations of Quartus and ModelSim. When I coded for each module, I found that if I wanna have a deeper understanding of the knowledge in textbook, I'm supposed to use it in practice.

Thus this verilog project is a great project experience for me, which helps me know what to learn and what to do research about next.