CS303 Term Project: Elevator Module

Team Members:

Görkem Topçu 28862

Can Zunal 29453

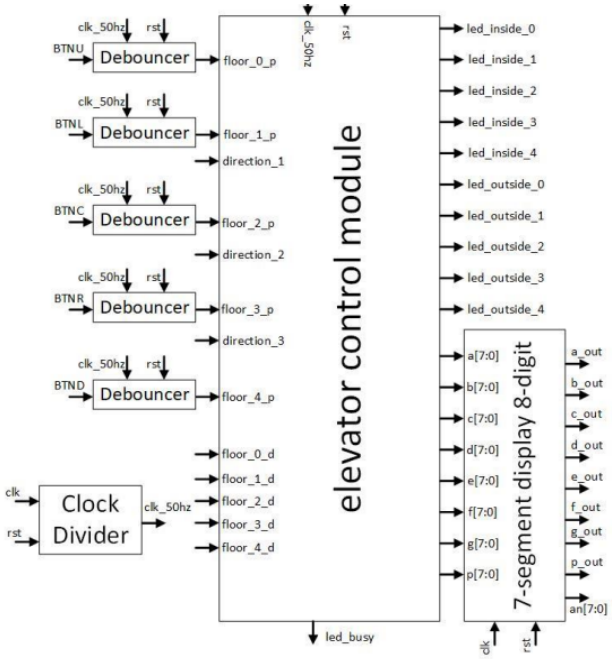
January 19, 2023

Introduction

Implementation of a module that operates an elevator that has 5 floors in verilog programming language and used in an FPGA board for demonstration. The implementation of the elevator can be separated into 4 different modules namely: debouncer which controls external inputs, clock divider which is used to derive another clock using the internal clock of the FPGA, SSD module which manages 7 segment displays to observe the floor numbers in real time and the main elevator control module which operates the floor transitions using external inputs. These modules are combined in a top module for the implementation. This report is divided into 3 parts: implementation, which will explain the individual modules; simulation that shows how the modules work in waveform and constraint file to show how we mapped the inputs and outputs into the FPGA board. The actual input and output names that are used in the code will be in parentheses for improved comprehension.

Inputs and Outputs

The circuit has 15 inputs and 4 different types of outputs for controlling LEDs and a 7-segment displays for our elevator controller design. Some of the outputs will be used as inputs in other modules, like clock dividers output divided clock(clk_50hz) will be used as input for the rest of the modules. Each input and output used in a module will be explained in more detail in the corresponding module. You can observe the input and outputs of the overall circuit of the top module in the following graph.



Implementation

In this section each module's input, outputs and what they do will be explained. Code snippets will be used to demonstrate how the code works. Implementation is divided into 5 parts: clock divider, debouncer, SSD Modul elevator control module and the top module that connects all the individual parts.

Clock Divider

For the clock divider we have a common reset(rst) and a 50 MHz clock(clk) generated by the FPGA board as input . The goal for this module is to change the frequency of the internal clock so that the display works in real time, each transition taking 5 seconds. This is achieved by using a counter that counts to 2000000. When the counter hits 2000000, the divided clock which is the output of circuit, gets complemented.

```
1  `timescale 1ns / 1ps
2  /*
3  clk_divider module reduces 100MHz (10ns) clock
4  to 50Hz (20ms) by dividing the clock with 2000000
5  It is possible to change the frequency of the output clock
6  by changing the toggle_value
7  */
8  module clk_divider(
9      input clk_in,
10     input rst,
11     output reg divided_clk
12     );
13 parameter toggle_value = 2000000;
14 reg [24:0] cnt;
15 always@(posedge clk_in or negedge rst)
16 begin
17     if (~rst) begin
18         cnt <= 0;
19         divided_clk <= 0;
20     end
21     else begin
22         if (cnt==toggle_value) begin
23             cnt <= 0;
24             divided_clk <= ~divided_clk;
25         end
26         else begin
27             cnt <= cnt +1;
28             divided_clk <= divided_clk;
29         end
30     end
31 end
32 endmodule
```

Debouncer

There are a total of 5 debouncer modules created for this project as we have 5 floors with 5 different buttons in each floor. This module has several inputs including the 50Hz (20ms) clock input generated by the clock divider module, common reset (rst) and an input received directly from the FPGA board's push buttons which is used to call an elevator from the outside. The goal is to filter out any noise present on the digital signal caused by multiple transitions when you press or release the button and output the cleaned signal. The cleaned outputs are then used as input for the elevator control module.

```
1  `timescale 1ns / 1ps
2  module debouncer(
3      input       clk,
4      input       rst,
5      input       noisy_in, // port from the push button
6      output reg  clean_out // port into the circuit
7      );
8  reg noisy_in_reg;
9  reg clean_out_tmp1; // will be used to detect rising edge
10 reg clean_out_tmp2; // will be used to detect rising edge
11 always@(posedge clk or negedge rst)
12 begin
13     if (~rst) begin
14         noisy_in_reg <= 0;
15         clean_out_tmp1 <= 0;
16         clean_out_tmp2 <= 0;
17
18         clean_out <= 0;
19     end
20     else begin
21         // store the input
22         noisy_in_reg <= noisy_in;
23         clean_out_tmp1 <= noisy_in_reg;
24
25         // rising edge detect
26         clean_out_tmp2 <= clean_out_tmp1;
27         clean_out <= ~clean_out_tmp2 & clean_out_tmp1; // it produce a single pulse during a risingedge
28     end
29 end
30 endmodule
31
```

Elevator Control Module

This is the main module that governs the state transition of the elevator. It uses different status machines, with combinational and sequential parts to achieve the behavior of an elevator. The module takes inputs and outputs directly from the FPGA board, the debouncer module and the clock divider module. Module checks if the calls from the different floors are valid, and changes the status of the elevator depending on the operation criteria. There are 2 different statuses that we consider and display in the 7 segment decoders: which floor the elevator currently is and its status(UP, DOWN or IDLE). Parameters are used for easier understanding in the code snippets.

```
1  `timescale 1ns / 1ps
2
3  module elevator_control_module(
4      input rst,
5      input clk,
6      input floor_0_p, floor_1_p, floor_2_p, floor_3_p, floor_4_p,
7      input direction_1, direction_2, direction_3,
8      input floor_0_d ,floor_1_d, floor_2_d,floor_3_d, floor_4_d,
9      output reg led_inside_0, led_inside_1, led_inside_2, led_inside_3, led_inside_4,
10     output reg led_outside_0, led_outside_1, led_outside_2, led_outside_3, led_outside_4,
11     output reg led_busy,
12     output reg [7:0] a, b, c, d, e, f, g, p
13     );
14
15     parameter FL_IDX0 = 3'b000, FL_IDX1 = 3'b001, FL_IDX2 = 3'b010, FL_IDX3 = 3'b011, FL_IDX4 = 3'b100; // floor indices
16     parameter FLOOR0 = 5'b00001, FLOOR1 = 5'b00010, FLOOR2 = 5'b00100, FLOOR3 = 5'b01000, FLOOR4 = 5'b10000; // floors
17     parameter IDLE = 2'b00, UP = 2'b01, DOWN = 2'b10; // states
18
19     reg [4:0] current_floor;
20     reg [4:0] next_floor;
21     reg [2:0] current_fl_idx, next_fl_idx;
22
23     reg [1:0] current_state;
24     reg [1:0] next_state;
25     reg [1:0] intermediate_state;
26
27     //counter to slow the input clock
28     reg [7:0] counter;
29
30     // floor states
31     reg [4:0] call_floor;
```

Inputs

This module has 15 inputs in total, a common clock (clk_50hz) from the clock divider module, a common reset (rst) that resets the circuit, 5 inputs for calling the elevator from outside from each floor (floor_x_p), 3 directional switches used when calling the elevator from outside for floors 1 to 3(direction_x) and 5 inputs for calling the elevator from outside from each floor(floor_x_d). Consider x as the corresponding floor number.

Outputs

This module has 4 different types of output, 5 LEDs to show which switches are used to go to destination floors for the people inside the elevators (led_inside_x). Another 5 LEDs to show which push buttons are pressed to call the elevator for the people outside the elevator (led_outside_x). These outputs should only go high when the call from the outside and inside inputs are valid according to the operation criteria which will be checked by a status machine. There is also another output used to show that the elevator is busy after reaching and stopping on a floor(led_busy). The last output type of this module is used to govern the 7 segment decoder used for observing which floor the elevator is in real life and its status.

Sequential Part - State Transitions

In this part we have an always block that is used to govern state transitions. Common reset signal resets the states to the initial state, in this case floor 0 and idle. Next state and next floor are assigned to the current state and current floor respectively. Also we used a counter to increase the time period between transitions as the elevator needs to change floors and states every 5 seconds in the FPGA implementation.

```
33      //busy state
34      reg busy_state;
35
36      //sequential part - state transitions
37      always@(posedge clk or negedge rst)
38      begin
39          if (~rst)begin
40              current_floor <= FLOOR0;
41              current_fl_idx <= FL_IDX0;
42              current_state <= IDLE;
43              led_busy <= 0;
44              counter <= 0;
45          end
46          else begin
47              if(counter == 8'b11111010) begin     // if equal to 250
48                  // status change
49                  current_state <= next_state;
50                  // led busy change
51                  led_busy <= busy_state;
52                  // floor change
53                  current_floor <= next_floor;
54                  current_fl_idx <= next_fl_idx;
55                  // restart counter
56                  counter <= 0;
57              end
58              else counter <= counter + 1; // if not 250 add 1
59          end
60      end
```

Next Floor Definitions

In this part depending on the current floor, current state and a register we hold (call_floor) we decide on the next floor of the elevator. Different cases are checked for each floor. For instance, if the elevator is on floor 0 and the current state is IDLE or there is a call from the same floor, the next floor should stay the same. Another example is if the elevator is on the 2nd floor and the next state is up elevator needs to go to the 3rd floor. Depending on the current state and floor next_floor changes. Next_fl_idx, which is used for comparisons in another state machine, is also changed in this part.

```
62      // next floor definitions
63      always @(*)
64      begin
65        case(current_floor)
66          FLOOR0:begin
67              if(current_state == IDLE || call_floor[current_fl_idx])begin
68                  next_floor <= FLOOR0;
69                  next_fl_idx <= FL_IDX0;
70              end
71              else if(current_state == UP)begin
72                  next_floor <= FLOOR1;
73                  next_fl_idx <= FL_IDX1;
74              end
75              else begin
76                  next_floor <= FLOOR0;
77                  next_fl_idx <= FL_IDX0;
78              end
79           end
80          FLOOR1:begin
81              if(current_state == IDLE || call_floor[current_fl_idx])begin
82                  next_floor <= FLOOR1;
83                  next_fl_idx <= FL_IDX1;
84              end
85              else if(current_state == UP)begin
86                  next_floor <= FLOOR2;
87                  next_fl_idx <= FL_IDX2;
88              end
89              else if(current_state == DOWN)begin
90                  next_floor <= FLOOR0;
91                  next_fl_idx <= FL_IDX0;
92              end
93              else begin

94                  next_floor <= FLOOR1;
95                  next_fl_idx <= FL_IDX1;
96              end
97          end
98          FLOOR2:begin
99              if(current_state == IDLE || call_floor[current_fl_idx])begin
100                 next_floor <= FLOOR2;
101                 next_fl_idx <= FL_IDX2;
102             end
103             else if(current_state == UP)begin
104                 next_floor <= FLOOR3;
105                 next_fl_idx <= FL_IDX3;
106             end
107             else if(current_state == DOWN)begin
108                 next_floor <= FLOOR1;
109                 next_fl_idx <= FL_IDX1;
110             end
111             else begin
112                 next_floor <= FLOOR3;
113                 next_fl_idx <= FL_IDX3;
114             end
115         end
116         FLOOR3:begin;
117             if(current_state == IDLE || call_floor[current_fl_idx])begin
118                 next_floor <= FLOOR3;
119                 next_fl_idx <= FL_IDX3;
120             end
121             else if(current_state == UP)begin
122                 next_floor <= FLOOR4;
123                 next_fl_idx <= FL_IDX4;
124             end
125             else if(current_state == DOWN)begin
```

```
                next_floor <= FLOOR2;
                next_fl_idx <= FL_IDX2;
            end
            else begin
                next_floor <= FLOOR3;
                next_fl_idx <= FL_IDX3;
            end
        end
        FLOOR4:begin
            if(current_state == IDLE || call_floor[current_fl_idx])begin // if idle state or there is a call here
                next_floor <= FLOOR4;
                next_fl_idx <= FL_IDX4;
            end
            else if(current_state == DOWN)begin
                next_floor <= FLOOR3;
                next_fl_idx <= FL_IDX3;
            end
            else begin
                next_floor <= FLOOR4;
                next_fl_idx <= FL_IDX4;
            end
        end
        default: begin
            next_floor <= FLOOR0;
            next_fl_idx <= FL_IDX0;
        end
      endcase
    end
```

Status State Machine

In this part we change the state of the elevator depending on an array that we hold called call_floor that stores valid calls made from different floors and current state of the elevator. In order to change floors the following logic is used. If the current state is IDLE and there is a call from the current floor the next state should be IDLE. If the current state is IDLE and there is a call from a floor that is higher than the current floor, the current state should change to up. If the current state is IDLE and there is a call from a floor that is lower than the current floor, the current state should change to DOWN. Notice that while the current state is UP or DOWN and the next state is IDLE meaning that the elevator stops on a floor the busy_state changes to 1. Busy state is used to light led_busy.

```
    // Status state machine
    always@ (*)begin
        case (current_state)
            IDLE:begin
                busy_state <= 0;
                if(call_floor)begin // there is a call
                    if(call_floor[current_fl_idx]) next_state = IDLE; // call is at current floor
                    else if(call_floor > current_floor) next_state = UP; // call is at upper floors
                    else if (call_floor < current_floor) next_state = DOWN; // call is at lower floors
                    else next_state = IDLE;
                end
                else next_state <= IDLE;
            end
            UP:begin
                if(call_floor[current_fl_idx+1])begin
                    next_state <= IDLE; // there is a call at one upper floor
                    busy_state <= 1;
                end
                else next_state <= UP;
            end
            DOWN:begin
                if(call_floor[current_fl_idx-1])begin
                    next_state <= IDLE; // there is a call at one lower floor
                    busy_state <= 1;
                end
                else next_state <= DOWN;
            end
            default: next_state <= IDLE;
        endcase
    end
```

Elevator Calling

In this status machine the goal is to change an array called call_floor that is used in another status machine that decides state changes and it also changes inside(led_inside_x) and outside LEDs (led_outside_x) depending on inputs. Common reset(rst) is used to reset the LEDS and call_floor array. The operation is fairly simple, we just need to check which input is high. After that, depending on the current floor, call_floor's status as well as direction inputs it decides whether the call is viable and adds it into the call_floor array which is then used for operating status state machines.

```
186     // ELEVATOR CALLING
187     always @(posedge clk or negedge rst)begin
188     if(~rst)begin
189         call_floor <= 0;
190         intermediate_state <= IDLE; // NEW
191         led_outside_0 <= 0;
192         led_outside_1 <= 0;
193         led_outside_2 <= 0;
194         led_outside_3 <= 0;
195         led_outside_4 <= 0;
196         led_inside_0 <= 0;
197         led_inside_1 <= 0;
198         led_inside_2 <= 0;
199         led_inside_3 <= 0;
200         led_inside_4 <= 0;
201     end
202     else if((floor_0_p || floor_0_d) && current_floor != FLOOR0 && ~call_floor[0] && intermediate_state != DOWN)begin // floor_0_p (lowermost floor)
203         if(current_state != UP)begin
204             intermediate_state <= UP;
205             call_floor[0] <= 1;
206             if(floor_0_p)led_outside_0 <= 1;
207             else if (floor_0_d) led_inside_0 <= 1;
208         end
209         else begin
210             call_floor[0] <= 0;
211             if(floor_0_p)led_outside_0 <= 0;
212             else if(floor_0_d) led_inside_0 <= 0;
213         end
214     end
215     else if((floor_1_p || floor_1_d) && current_floor != FLOOR1 && ~call_floor[1])begin // floor_1_p
216         if(current_state == IDLE)begin
217             if((~direction_1 && intermediate_state!=UP) || (direction_1 && intermediate_state != DOWN))begin

218                 if(~direction_1) intermediate_state <= DOWN;
219                 else intermediate_state <= UP;
220                 call_floor[1] <= 1;
221                 if(floor_1_p)led_outside_1 <= 1;
222                 else if(floor_1_d) led_inside_1 <= 1;
223             end
224             else begin
225                 call_floor[1] <= 0;
226                 if(floor_1_p)led_outside_1 <= 0;
227                 else if(floor_1_d)led_inside_1 <= 0;
228             end
229         end
230         else if(~direction_1)begin // go down
231             if(current_state == UP)begin
232                 call_floor[1] <= 0;
233                 if(floor_1_p)led_outside_1 <= 0;
234                 else if(floor_1_d)led_inside_1 <= 0;
235             end
236             else begin
237                 call_floor[1] <= 1;
238                 if(floor_1_p)led_outside_1 <= 1;
239                 else if(floor_1_d) led_inside_1 <= 1;
240             end
241         end
242         else if(direction_1)begin // go up
243             if(FLOOR1 < current_floor)begin
244                 call_floor[1] <= 0;
245                 if(floor_1_p)led_outside_1 <= 0;
246                 else if(floor_1_d) led_inside_1 <= 0;
247             end
248             else begin
249                 call_floor[1] <= 1;
```

```
            if(floor_1_p)led_outside_1 <= 1;
            else if(floor_1_d) led_inside_1 <= 1;
        end
    end
end
else if((floor_2_p || floor_2_d) && current_floor != FLOOR2 && ~call_floor[2])begin // floor_2_p
    if(current_state == IDLE) begin
        if((direction_2 && intermediate_state!=DOWN)|| (~direction_2 && intermediate_state != UP))begin
            if(~direction_1) intermediate_state <= DOWN;
            else intermediate_state <= UP;
            call_floor[2] <= 1;
            if(floor_2_p)led_outside_2 <= 1;
            else if(floor_2_d) led_inside_2 <= 1;
        end
        else begin
            call_floor[2] <= 0;
            if(floor_2_p)led_outside_2 <= 0;
            else if(floor_2_d) led_inside_2 <= 0;
        end
    end
    else if(~direction_2)begin // go down
        if(FLOOR2 > current_floor)begin
            call_floor[2] <= 0;
            if(floor_2_p)led_outside_2 <= 0;
            else if(floor_2_d) led_inside_2 <= 0;
        end
        else if(current_state == UP)begin
            call_floor[2] <= 0;
            if(floor_2_p)led_outside_2 <= 0;
            else if(floor_2_d) led_inside_2 <= 0;
        end
        else begin


            call_floor[2] <= 1;
            if(floor_2_p)led_outside_2 <= 1;
            else if(floor_2_d) led_inside_2 <= 1;
        end
    end
    else if(direction_2)begin // go up
        if(FLOOR2 < current_floor)begin
            call_floor[2] <= 0;
            if(floor_2_p)led_outside_2 <= 0;
            else if(floor_3_d) led_inside_2 <= 0;
        end
        else if(current_state == DOWN)begin
            call_floor[2] <= 0;
            if(floor_2_p)led_outside_2 <= 0;
            else if(floor_3_d) led_inside_2 <= 0;
        end
        else begin
            call_floor[2] <= 1;
            if(floor_2_p)led_outside_2 <= 1;
            else if(floor_3_d) led_inside_2 <= 1;
        end
    end
end
else if((floor_3_p || floor_3_d) && current_floor != FLOOR3 && ~call_floor[3])begin// floor_3_p
    if(current_state == IDLE)begin
        if((direction_3 && intermediate_state!=DOWN) || (~direction_3 && intermediate_state!=UP))begin
            if(~direction_1) intermediate_state <= DOWN;
            else intermediate_state <= UP;
            call_floor[3] <= 1;
            if(floor_3_p)led_outside_3 <= 1;
            else if(floor_3_d) led_inside_3 <= 1;
        end


        else begin
            call_floor[3] <= 0;
            if(floor_3_p)led_outside_3 <= 0;
            else if(floor_3_d) led_inside_3 <= 0;
        end
    end
    else if(~direction_3)begin // go down
        if(FLOOR3 > current_floor)begin
            call_floor[3] <= 0;
            if(floor_3_p)led_outside_3 <= 0;
            else if(floor_3_d) led_inside_3 <= 0;
        end
        else begin
            call_floor[3] <= 1;
            if(floor_3_p)led_outside_3 <= 1;
            else if(floor_3_d) led_inside_3 <= 1;
        end
    end
    else if(direction_3)begin // go up
        if(current_state == DOWN)begin
            call_floor[3] <= 0;
            if(floor_3_p)led_outside_3 <= 0;
            else if(floor_3_d) led_inside_3 <= 0;
        end
        else begin
            call_floor[3] <= 1;
            if(floor_3_p)led_outside_3 <= 1;
            else if(floor_3_d) led_inside_3 <= 1;
        end
    end
end
else if((floor_4_p || floor_4_d) && current_floor != FLOOR4 && ~call_floor[4] && intermediate_state != UP)begin // floor_4_p (uppermost floor)
```

```
            intermediate_state <= DOWN;
            call_floor[4] <= 1;
            if(floor_4_p)led_outside_4 <= 1;
            else if(floor_4_d) led_inside_4 <= 1;
        end
        else if(current_state != DOWN)begin
            call_floor[4] <= 1;
            if(floor_4_p)led_outside_4 <= 1;
            else if(floor_4_d) led_inside_4 <= 1;
        end
        else begin
            call_floor[4] <= 0;
            if(floor_4_p)led_outside_4 <= 0;
            else if(floor_4_d) led_inside_4 <= 0;
        end
    end
    else begin
        if(current_floor == FLOOR1)begin // floor is 1
            call_floor[1] <= 0;
            led_outside_1 <= 0;
            led_inside_1 <= 0;
        end
        else if(current_floor == FLOOR2)begin // floor is 2
            call_floor[2] <= 0;
            led_outside_2 <= 0;
            led_inside_2 <= 0;
        end
        else if(current_floor == FLOOR3)begin // floor is 3
            call_floor[3] <= 0;
            led_outside_3 <= 0;
            led_inside_3 <= 0;
        end

         else if(current_floor == FLOOR4)begin // floor is 4
             call_floor[4] <= 0;
             led_outside_4 <= 0;
             led_inside_4 <= 0;
         end
         else begin // floor is 0
             call_floor[0] <= 0;
             led_outside_0 <= 0;
             led_inside_0 <= 0;
         end
         if(~call_floor) intermediate_state <= IDLE;
         else begin end
     end
end
```

Status Machine to Display the Outputs

In this part we give outputs to the SSD module to observe the status of the elevator in real time. The outputs are 8 bit arrays that map the 7 segment decoder. The right and left block have 8 SSD in total. The 4 leftmost SSD's show the status and 4 right most SSD’s show current floor number. There are a total of 13 cases as there are 5 floors, and 3 states for floors 1 to 3 and 2 states for 0th and 4th floor. Depending on these 13 cases, the status machine changes the displays. SSD module’s operation will be explained in the corresponding part.

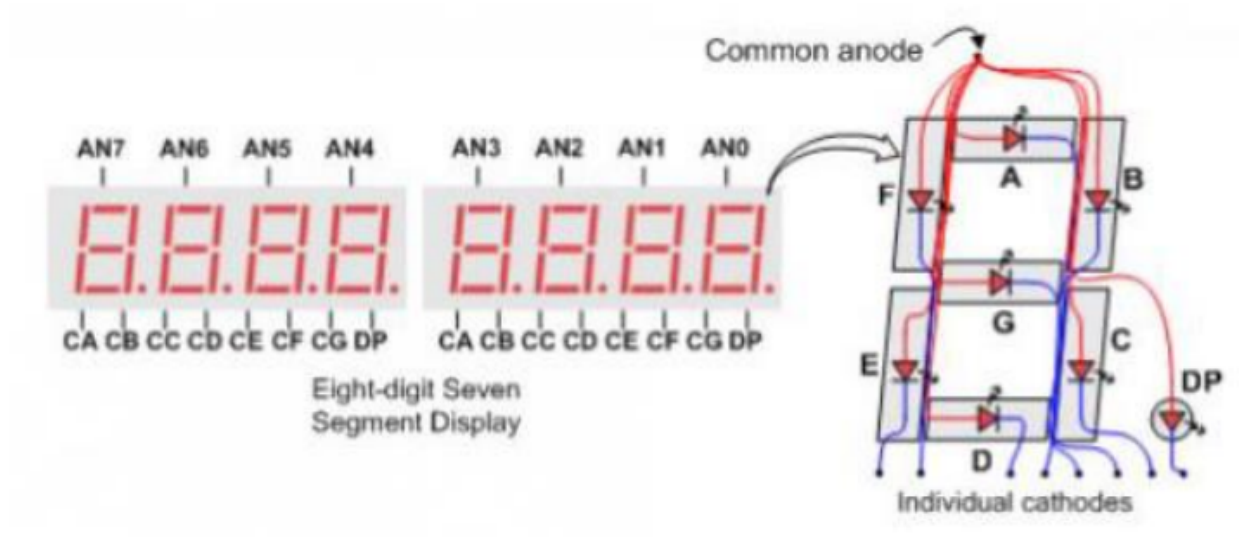
```
// Seven segment display
always @ (posedge clk or negedge rst) begin
    if(~rst) begin
        a <= 8'b11111111;
        b <= 8'b11111111;
        c <= 8'b11111111;
        d <= 8'b11111111;
        e <= 8'b11111111;
        f <= 8'b11111111;
        g <= 8'b11111111;
        p <= 8'b11111111;
    end
    else begin
    case(current_floor)
       FLOOR0: begin // FL-0
           case(current_state)
            IDLE: begin //--Id
               a <= 8'b11110110;
               b <= 8'b11001110;
               c <= 8'b11001110;
               d <= 8'b11101010;
               e <= 8'b11100010;
               f <= 8'b11110010;
               g <= 8'b11100111;
               p <= 8'b11111111;
           end
           UP: begin //--UP
               a <= 8'b11100110;
               b <= 8'b11001110;
               c <= 8'b11011110;
               d <= 8'b11011010;
               e <= 8'b11000010;

                f <= 8'b11000010;
                g <= 8'b11100111;
                p <= 8'b11111111;
           end
           DOWN: begin //--d0
               a <= 8'b11100110;
               b <= 8'b11001110;
               c <= 8'b11001110;
               d <= 8'b11001010;
               e <= 8'b11000010;
               f <= 8'b11100010;
               g <= 8'b11010111;
               p <= 8'b11111111;
           end
           endcase
       end
       FLOOR1: begin // FL-1
           case(current_state)
            IDLE: begin //--Id
               a <= 8'b11110111;
               b <= 8'b11001110;
               c <= 8'b11001110;
               d <= 8'b11101011;
               e <= 8'b11100011;
               f <= 8'b11110011;
               g <= 8'b11100111;
               p <= 8'b11111111;
           end
           UP: begin //--UP
               a <= 8'b11100111;
               b <= 8'b11001110;
               c <= 8'b11011110;
```

```
458                 d <= 8'b11011011;
459                 e <= 8'b11000011;
460                 f <= 8'b11000011;
461                 g <= 8'b11100111;
462                 p <= 8'b11111111;
463             end
464             DOWN: begin //--d0
465                 a <= 8'b11100111;
466                 b <= 8'b11001110;
467                 c <= 8'b11001110;
468                 d <= 8'b11001011;
469                 e <= 8'b11000011;
470                 f <= 8'b11100011;
471                 g <= 8'b11010111;
472                 p <= 8'b11111111;
473             end
474             endcase
475         end
476         FLOOR2: begin // FL-2
477              case(current_state)
478              IDLE: begin //--Id
479                 a <= 8'b11110110;
480                 b <= 8'b11001110;
481                 c <= 8'b11001111;
482                 d <= 8'b11101010;
483                 e <= 8'b11100010;
484                 f <= 8'b11110011;
485                 g <= 8'b11100111;
486                 p <= 8'b11111111;
487             end

488              UP: begin //--UP
489                  a <= 8'b11100110;
490                  b <= 8'b11001110;
491                  c <= 8'b11011111;
492                  d <= 8'b11011010;
493                  e <= 8'b11000010;
494                  f <= 8'b11000011;
495                  g <= 8'b11100111;
496                  p <= 8'b11111111;
497              end
498              DOWN: begin //--d0
499                  a <= 8'b11100110;
500                  b <= 8'b11001110;
501                  c <= 8'b11001111;
502                  d <= 8'b11001010;
503                  e <= 8'b11000010;
504                  f <= 8'b11100011;
505                  g <= 8'b11010111;
506                  p <= 8'b11111111;
507              end
508             endcase
509         end
510         FLOOR3: begin  // FL-3
511          case(current_state)
512             IDLE: begin //--Id
513                  a <= 8'b11110110;
514                  b <= 8'b11001110;
515                  c <= 8'b11001110;
516                  d <= 8'b11101010;
517                  e <= 8'b11100011;
518                  f <= 8'b11110011;
519                  g <= 8'b11100110;
```

```
                p <= 8'b11111111;
            end
            UP: begin //--UP
                a <= 8'b11100110;
                b <= 8'b11001110;
                c <= 8'b11011110;
                d <= 8'b11011010;
                e <= 8'b11000011;
                f <= 8'b11000011;
                g <= 8'b11100110;
                p <= 8'b11111111;
            end
            DOWN: begin //--dO
                a <= 8'b11100110;
                b <= 8'b11001110;
                c <= 8'b11001110;
                d <= 8'b11001010;
                e <= 8'b11000011;
                f <= 8'b11100011;
                g <= 8'b11010110;
                p <= 8'b11111111;
            end
            endcase
        end
        FLOOR4: begin // FL-4
            case(current_state)
             IDLE: begin //--Id
                a <= 8'b11110111;
                b <= 8'b11001110;
                c <= 8'b11001110;
                d <= 8'b11101011;
                e <= 8'b11100011;

                f <= 8'b11110010;
                g <= 8'b11100110;
                p <= 8'b11111111;
              end
              UP: begin //--UP
                a <= 8'b11100111;
                b <= 8'b11001110;
                c <= 8'b11011110;
                d <= 8'b11011011;
                e <= 8'b11000011;
                f <= 8'b11000010;
                g <= 8'b11100110;
                p <= 8'b11111111;
              end
              DOWN: begin //--dO
                a <= 8'b11100111;
                b <= 8'b11001110;
                c <= 8'b11001110;
                d <= 8'b11001011;
                e <= 8'b11000011;
                f <= 8'b11100010;
                g <= 8'b11010110;
                p <= 8'b11111111;
              end
              endcase
      end
      endcase
      end
endmodule
```

SSD Module

Each SSD (digit) has 8 diodes (A, B, C, D, E, F, G, P). The anodes of all 8 diodes are connected, so they have a common anode. Their cathodes are operated by the module. To activate anything you need to activate the common anode of that SSD. It works active low. Common reset input reset all the displays as usual.




```
`timescale 1ns / 1ps
module ssd(clk,reset,a,b,c,d,e,f,g,p,
        a_out,b_out,c_out,d_out,e_out,f_out,g_out,p_out,
        an);
input clk, reset;// set clock and reset as input(1 bit)
input [7:0] a,b,c,d,e,f,g,p;
output reg a_out,b_out,c_out,d_out,e_out,f_out,g_out,p_out;
output reg [7:0] an;

reg [3:0] state;//holds state number (3 bit)
reg [19:0] counter;//counter to slow the input clock

// in this always block the speed of the clock reduced by 100000
// times so that display works properly
always @ (posedge clk or negedge reset) begin //state counter
    if(~reset) begin    //synchronus reset
        state <= 0; //if reset set state and counter to zero
        counter <= 0;
    end
    else begin //else the counter untill 100000
        if(counter == 20'h186A0) begin   //if equal to 100000
            if (state == 4'b1000)
                state <= 1;
            else
                state <= state + 1;
            counter <= 0;
        end
        else
            counter <= counter + 1; //if not 100000 add 1
    end
end
```

```
33  //in this always block we give the inputs to the leds by choosing
34  //different display segment in each time
35
36  always@(posedge clk or negedge reset)
37  begin
38      if(~reset)// if reset initilize the outputs
39      begin
40          an[7:0] <= 8'b11111111;
41          a_out <= 1;
42          b_out  <= 1;
43          c_out  <= 1;
44          d_out  <= 1;
45          e_out  <= 1;
46          f_out  <= 1;
47          g_out  <= 1;
48          p_out  <= 1;
49      end
50      else if(state == 4'b0001)
51  //state 1 gives the led outputs to the AN0
52      begin
53          an[7:0] <= 8'b11111110;
54          a_out  <= a[0];
55          b_out  <= b[0];
56          c_out  <= c[0];
57          d_out  <= d[0];
58          e_out  <= e[0];
59          f_out  <= f[0];
60          g_out  <= g[0];
61          p_out  <= p[0];
62      end

63      else if(state == 4'b0010)
64  //state 2 gives the led outputs to the AN1
65      begin
66          an[7:0] <= 8'b11111101;
67          a_out  <= a[1];
68          b_out  <= b[1];
69          c_out  <= c[1];
70          d_out  <= d[1];
71          e_out  <= e[1];
72          f_out  <= f[1];
73          g_out  <= g[1];
74          p_out  <= p[1];
75
76      end
77      else if(state == 4'b0011)
78  //state 3 gives the led outputs to the AN2
79      begin
80          an[7:0] <= 8'b11111011;
81          a_out  <= a[2];
82          b_out  <= b[2];
83          c_out  <= c[2];
84          d_out  <= d[2];
85          e_out  <= e[2];
86          f_out  <= f[2];
87          g_out  <= g[2];
88          p_out  <= p[2];
89
90      end
```

```
77      else if(state == 4'b0011)
78    //state 3 gives the led outputs to the AN2
79      begin
80          an[7:0] <= 8'b11111011;
81          a_out  <= a[2];
82          b_out  <= b[2];
83          c_out  <= c[2];
84          d_out  <= d[2];
85          e_out  <= e[2];
86          f_out  <= f[2];
87          g_out  <= g[2];
88          p_out  <= p[2];
89
90      end
91      else if(state == 4'b0100)
92    //state 4 gives the led outputs to the AN3
93      begin
94          an[7:0] <= 8'b11110111;
95          a_out  <= a[3];
96          b_out  <= b[3];
97          c_out  <= c[3];
98          d_out  <= d[3];
99          e_out  <= e[3];
100         f_out  <= f[3];
101         g_out  <= g[3];
102         p_out  <= p[3];
103
104     end

105     else if(state == 4'b0101)
106   //state 4 gives the led outputs to the AN4
107     begin
108         an[7:0] <= 8'b11101111;
109         a_out  <= a[4];
110         b_out  <= b[4];
111         c_out  <= c[4];
112         d_out  <= d[4];
113         e_out  <= e[4];
114         f_out  <= f[4];
115         g_out  <= g[4];
116         p_out  <= p[4];
117
118     end
119
120     else if(state == 4'b0110)
121   //state 4 gives the led outputs to the AN5
122     begin
123         an[7:0] <= 8'b11011111;
124         a_out  <= a[5];
125         b_out  <= b[5];
126         c_out  <= c[5];
127         d_out  <= d[5];
128         e_out  <= e[5];
129         f_out  <= f[5];
130         g_out  <= g[5];
131         p_out  <= p[5];
132
133     end
```

```
    else if(state == 4'b0111)
//state 4 gives the led outputs to the AN6
    begin
        an[7:0] <= 8'b10111111;
        a_out  <= a[6];
        b_out  <= b[6];
        c_out  <= c[6];
        d_out  <= d[6];
        e_out  <= e[6];
        f_out  <= f[6];
        g_out  <= g[6];
        p_out  <= p[6];

    end

    else if(state == 4'b1000)
//state 4 gives the led outputs to the AN7
    begin
        an[7:0] <= 8'b01111111;
        a_out  <= a[7];
        b_out  <= b[7];
        c_out  <= c[7];
        d_out  <= d[7];
        e_out  <= e[7];
        f_out  <= f[7];
        g_out  <= g[7];
        p_out  <= p[7];

    end

     else //For other states default inputs and outputs
     begin
         an[7:0] <= 8'b11111111;
         a_out  <= 1;
         b_out  <= 1;
         c_out  <= 1;
         d_out  <= 1;
         e_out  <= 1;
         f_out  <= 1;
         g_out  <= 1;
         p_out  <= 1;
     end
end

endmodule
```

Top Module

In this part all the modules explained thus far are combined into a top module. Clock divider module is used to create a new clock(clk50hz) that is distributed to other modules. 5 debouncers are used to filter unwanted inputs, These inputs are then fed into the elevator control module which then creates input for the SSD module.

```
1  `timescale 1ns / 1ps
2
3  module top_module(
4      input clk,
5      rst,
6
7      floor_0_p,
8      floor_1_p,
9      floor_2_p,
10     floor_3_p,
11     floor_4_p,
12
13     direction_1,
14     direction_2,
15     direction_3,
16
17     floor_0_d,
18     floor_1_d,
19     floor_2_d,
20     floor_3_d,
21     floor_4_d,
22
23     output led_inside_0,
24     led_inside_1,
25     led_inside_2,
26     led_inside_3,
27     led_inside_4,
28
29     led_outside_0,
30     led_outside_1,
31     led_outside_2,
32     led_outside_3,
```

```
66        .direction_2(direction_2),
67        .direction_3(direction_3),
68        .floor_0_d(floor_0_d),
69        .floor_1_d(floor_1_d),
70        .floor_2_d(floor_2_d),
71        .floor_3_d(floor_3_d),
72        .floor_4_d(floor_4_d),
73        .led_inside_0(led_inside_0),
74        .led_inside_1(led_inside_1),
75        .led_inside_2(led_inside_2),
76        .led_inside_3(led_inside_3),
77        .led_inside_4(led_inside_4),
78        .led_outside_0(led_outside_0),
79        .led_outside_1(led_outside_1),
80        .led_outside_2(led_outside_2),
81        .led_outside_3(led_outside_3),
82        .led_outside_4(led_outside_4),
83        .led_busy(led_busy),
84        .a(a),
85        .b(b),
86        .c(c),
87        .d(d),
88        .e(e),
89        .f(f),
90        .g(g),
91        .p(p)
92        );
93    endmodule
94
```

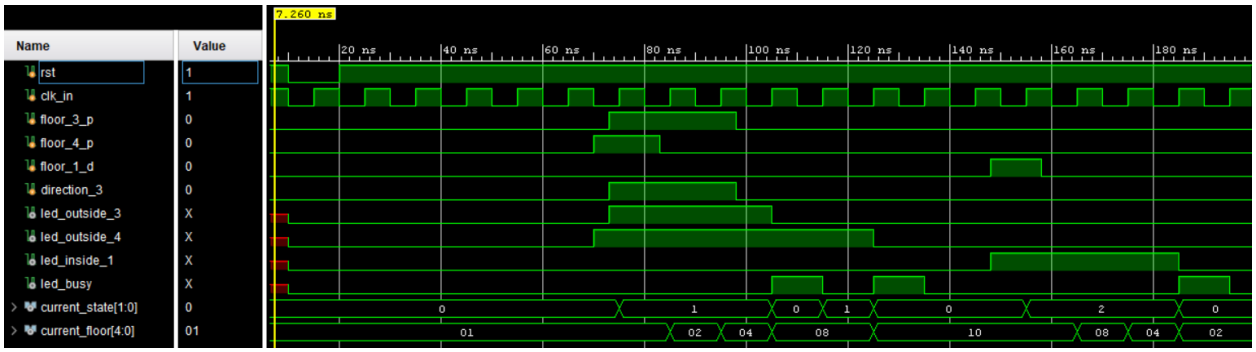
```
34
35      led_busy,
36
37      a_out,b_out,c_out,d_out,e_out,f_out,g_out,p_out,
38      [7:0]an
39  );
40
41      wire [7:0] a,b,c,d,e,f,g,p;
42      wire clk50hz;
43      wire floor_0_p_clean, floor_1_p_clean, floor_2_p_clean, floor_3_p_clean, floor_4_p_clean;
44
45      clk_divider divider(.clk_in(clk), .rst(rst), .divided_clk(clk50hz));
46
47      debouncer pb0(.clk(clk50hz), .rst(~rst), .noisy_in(floor_0_p), .clean_out(floor_0_p_clean));
48      debouncer pb1(.clk(clk50hz), .rst(~rst), .noisy_in(floor_1_p), .clean_out(floor_1_p_clean));
49      debouncer pb2(.clk(clk50hz), .rst(~rst), .noisy_in(floor_2_p), .clean_out(floor_2_p_clean));
50      debouncer pb3(.clk(clk50hz), .rst(~rst), .noisy_in(floor_3_p), .clean_out(floor_3_p_clean));
51      debouncer pb4(.clk(clk50hz), .rst(~rst), .noisy_in(floor_4_p), .clean_out(floor_4_p_clean));
52
53      ssd seven_segment_display(clk,rst,a,b,c,d,e,f,g,p,
54          a_out,b_out,c_out,d_out,e_out,f_out,g_out,p_out,
55          an);
56
57      elevator_control_module elevator(
58      .clk(clk50hz),
59      .rst(rst),
60      .floor_0_p(floor_0_p),
61      .floor_1_p(floor_1_p),
62      .floor_2_p(floor_2_p),
63      .floor_3_p(floor_3_p),
64      .floor_4_p(floor_4_p),
65      .direction_1(direction_1),
```

Simulations

Simulations are conducted in order to see if the circuit is working as expected for the operation criteria. As the original code is used for FPGA board implementation, current state and floor names are slightly different in simulation. For the current state 0 is for IDLE, 1 is for UP

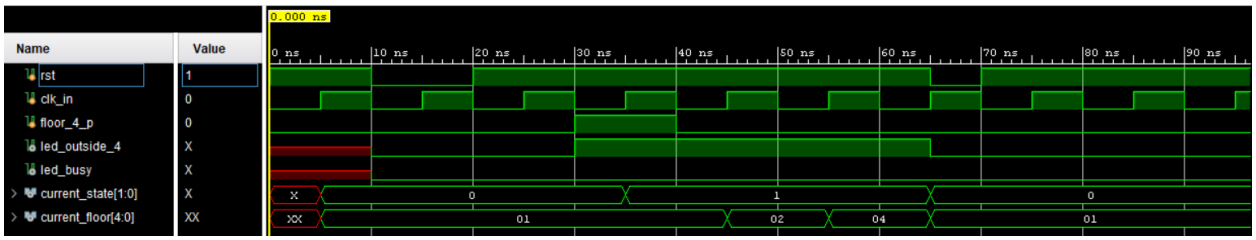
and 2 is for DOWN. For the current floor 01 is 0th floor, 02 is 1st floor, 04 is 2nd floor, 08 is for 3rd floor and 10 is for the 4th floor. As you can see floor numbers are currently represented in hexadecimal which is not the case for the original FPGA implementation. Unnecessary inputs and outputs are omitted for improved clarity.

First Case



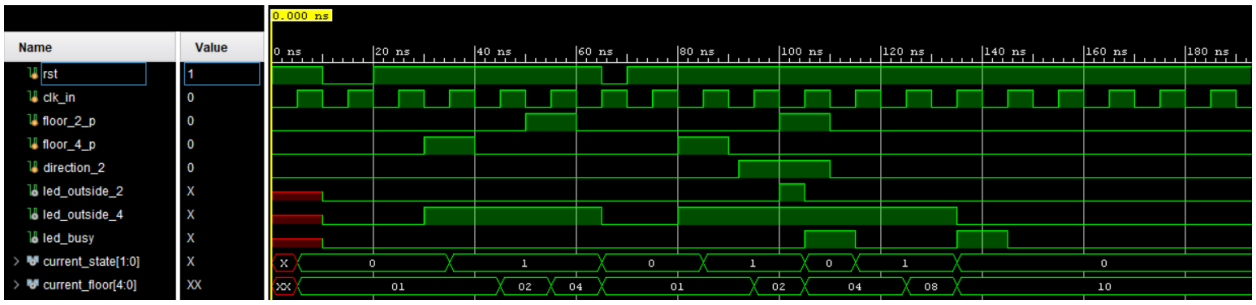
As you might have observed there is a call from outside of the 4th floor and 3rd floor respectively. As it can be seen on floors where an elevator call was made previously, elevator stops and led busy starts to light for one clock cycle. Since there were two calls from outside, led busy lights twice. After reaching the final destination which is the 4th floor, there is a call to the first floor from inside. Calls from inside of the elevator operate exactly the same as previous situations.

Second case



In this case there is a call from the 4th floor. While the elevator is going up reset input is asserted which resets the entire circuit.

Third case



In this case, to observe the difference between using direction switches during the call, we can divide the case into 2 parts before and after 70 ns. As you might observe in the first case (before 70ns) as direction_2 is off which means the caller wants to go down, which will cause the input to be discarded since it is not viable according to the operation criteria. In the other case as the direction_2 is high it works as intended.

Constraint File

We used the following constraint file in order to map the implementation to the FPGA Board. We created a clock to use in the FPGA board. Directions and floor_x_d are switches in the FPGA board.Floor_x_p and reset(rst) are push buttons. Outputs of the top module are mapped to the SSD's while led_inside_x, led_outside_x and led_busy are mapped to different leds. In the following code snippets specific pins can be observed.

```
set_property -dict { PACKAGE_PIN E3    IOSTANDARD LVCMOS33 } [get_ports { clk }];
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {clk}];

set_property -dict { PACKAGE_PIN C12  IOSTANDARD LVCMOS33 } [get_ports { rst }];
set_property -dict { PACKAGE_PIN M18  IOSTANDARD LVCMOS33 } [get_ports { floor_0_p }];
set_property -dict { PACKAGE_PIN P17  IOSTANDARD LVCMOS33 } [get_ports { floor_1_p }];
set_property -dict { PACKAGE_PIN N17  IOSTANDARD LVCMOS33 } [get_ports { floor_2_p }];
set_property -dict { PACKAGE_PIN M17  IOSTANDARD LVCMOS33 } [get_ports { floor_3_p }];
set_property -dict { PACKAGE_PIN P18  IOSTANDARD LVCMOS33 } [get_ports { floor_4_p }];

set_property -dict { PACKAGE_PIN L16  IOSTANDARD LVCMOS33 } [get_ports { direction_1 }];
set_property -dict { PACKAGE_PIN M13  IOSTANDARD LVCMOS33 } [get_ports { direction_2 }];
set_property -dict { PACKAGE_PIN R15  IOSTANDARD LVCMOS33 } [get_ports { direction_3 }];

set_property -dict { PACKAGE_PIN V10  IOSTANDARD LVCMOS33 } [get_ports { floor_0_d }];
set_property -dict { PACKAGE_PIN U11  IOSTANDARD LVCMOS33 } [get_ports { floor_1_d }];
set_property -dict { PACKAGE_PIN U12  IOSTANDARD LVCMOS33 } [get_ports { floor_2_d }];
set_property -dict { PACKAGE_PIN  H6  IOSTANDARD LVCMOS33 } [get_ports { floor_3_d }];
set_property -dict { PACKAGE_PIN T13  IOSTANDARD LVCMOS33 } [get_ports { floor_4_d }];

set_property -dict { PACKAGE_PIN V11  IOSTANDARD LVCMOS33 } [get_ports { led_inside_0 }];
set_property -dict { PACKAGE_PIN V12  IOSTANDARD LVCMOS33 } [get_ports { led_inside_1 }];
set_property -dict { PACKAGE_PIN V14  IOSTANDARD LVCMOS33 } [get_ports { led_inside_2 }];
set_property -dict { PACKAGE_PIN V15  IOSTANDARD LVCMOS33 } [get_ports { led_inside_3 }];
set_property -dict { PACKAGE_PIN T16  IOSTANDARD LVCMOS33 } [get_ports { led_inside_4 }];

set_property -dict { PACKAGE_PIN H17  IOSTANDARD LVCMOS33 } [get_ports { led_outside_0 }];
set_property -dict { PACKAGE_PIN K15  IOSTANDARD LVCMOS33 } [get_ports { led_outside_1 }];
set_property -dict { PACKAGE_PIN J13  IOSTANDARD LVCMOS33 } [get_ports { led_outside_2 }];
set_property -dict { PACKAGE_PIN N14  IOSTANDARD LVCMOS33 } [get_ports { led_outside_3 }];
set_property -dict { PACKAGE_PIN R18  IOSTANDARD LVCMOS33 } [get_ports { led_outside_4 }];
```

```
set_property -dict { PACKAGE_PIN N16  IOSTANDARD LVCMOS33 } [get_ports { led_busy }];

# outputs to seven-segment display
set_property -dict { PACKAGE_PIN H15  IOSTANDARD LVCMOS33 } [get_ports { p_out }]
set_property -dict { PACKAGE_PIN L18  IOSTANDARD LVCMOS33 } [get_ports { g_out }]
set_property -dict { PACKAGE_PIN T11  IOSTANDARD LVCMOS33 } [get_ports { f_out }]
set_property -dict { PACKAGE_PIN P15  IOSTANDARD LVCMOS33 } [get_ports { e_out }]
set_property -dict { PACKAGE_PIN K13  IOSTANDARD LVCMOS33 } [get_ports { d_out }]
set_property -dict { PACKAGE_PIN K16  IOSTANDARD LVCMOS33 } [get_ports { c_out }]
set_property -dict { PACKAGE_PIN R10  IOSTANDARD LVCMOS33 } [get_ports { b_out }]
set_property -dict { PACKAGE_PIN T10  IOSTANDARD LVCMOS33 } [get_ports { a_out }]

# outputs to seven-segment display segment select
set_property -dict { PACKAGE_PIN U13  IOSTANDARD LVCMOS33 } [get_ports { an[7] }]
set_property -dict { PACKAGE_PIN K2   IOSTANDARD LVCMOS33 } [get_ports { an[6] }]
set_property -dict { PACKAGE_PIN T14  IOSTANDARD LVCMOS33 } [get_ports { an[5] }]
set_property -dict { PACKAGE_PIN P14  IOSTANDARD LVCMOS33 } [get_ports { an[4] }]
set_property -dict { PACKAGE_PIN J14  IOSTANDARD LVCMOS33 } [get_ports { an[3] }]
set_property -dict { PACKAGE_PIN T9   IOSTANDARD LVCMOS33 } [get_ports { an[2] }]
set_property -dict { PACKAGE_PIN J18  IOSTANDARD LVCMOS33 } [get_ports { an[1] }]
set_property -dict { PACKAGE_PIN J17  IOSTANDARD LVCMOS33 } [get_ports { an[0] }]
```

Conclusion

In this project we implement an elevator controller in verilog programming language for a 5 floor elevator that has buttons and displays both inside and outside. It had both combinational and sequential parts. Different modules are combined into a top module. Test cases are performed in the simulations to check whether the module works as intended.