Selin Ceydeli

Canberk Tahıl

CS303 Term Project – Elevator Control System

A General Overview of the Modules We Defined

In our project, we have written 3 extra modules in addition to the elevator module. These extra modules include two counter modules (one of which counts up to 250 (corresponding to 5 seconds of waiting time) and the other one counts up to 500 (corresponding to 10 seconds of waiting time)) and a range_checker module which returns a boolean value (1 if true, 0 if false) and checks whether a request made from a particular floor must be accepted or not.

We instantiated the range_checker module once for all five floors so that the checks are made separately for requests from a particular floor.

We instantiated the counter module, which counts up to 250, twice so that we can use one of the timed_up outputs that are returned in maintaining the elevator's floor changes in 5 seconds and the other one in turning the led_busy on for 5 seconds.

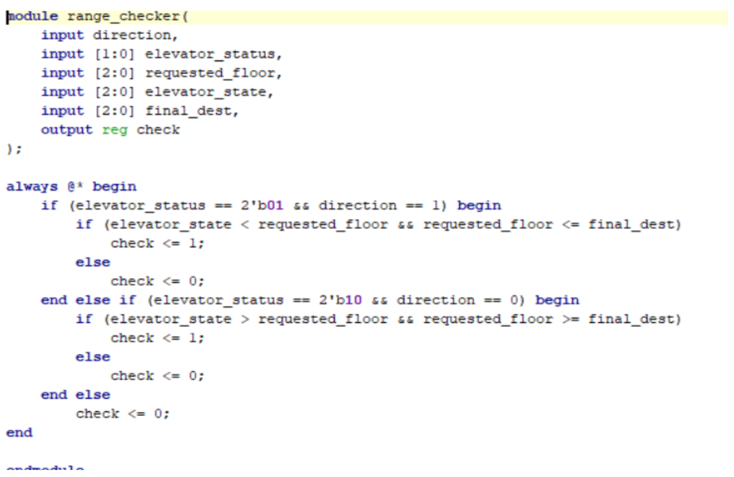
We instantiated the counter_2 module, which counts up to 500, once to use in the elevator's floor changes in 10 seconds for the situation when it is picking up or dropping off someone on a particular floor.

We negated the reset in the top_module to convert it from a negedge reset to a posedge reset and used it accordingly in the always blocks of our elevator module. The reason why we decided to use a posedge reset instead of a negedge reset is that since the rest of the modules are using a posedge reset, we wanted all the modules to be synchronized, and thus, we used a posedge reset in the always blocks of our elevator module and the extra modules we defined, as described above.

The extra modules we have created and the functions we defined in the elevator module are explained in detail section by section below:

Range Checker Module

In the elevator module, we have defined a range checker module which is used to accept/reject the floor requests and turn on the LEDs of the necessary floors accordingly. In the always block that works at all cases, we first compare the direction switch which the person pressed at a particular floor and the elevator's status which both of them can be either UP (2'b01 or 1) or DOWN (2'b10 or 0). If both of them are UP or DOWN, then we check whether the requested floor is between elevator_state (current floor of elevator) and the final_dest (final destination which the elevator will go). If the requested floor is in range then, output check is sent to the elevator module with value 1 indicating that a person will be taken at that floor, otherwise the request is ignored and elevator keeps moving.



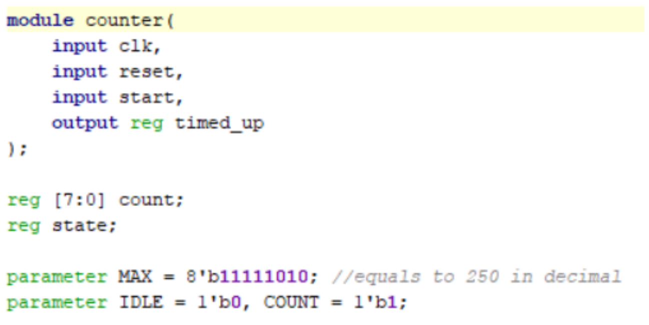
```
module range_checker(
    input direction,
    input [1:0] elevator_status,
    input [2:0] requested_floor,
    input [2:0] elevator_state,
    input [2:0] final_dest,
    output reg check
);

always @* begin
    if (elevator_status == 2'b01 && direction == 1) begin
        if (elevator_state < requested_floor && requested_floor <= final_dest)
            check <= 1;
        else
            check <= 0;
    end else if (elevator_status == 2'b10 && direction == 0) begin
        if (elevator_state > requested_floor && requested_floor >= final_dest)
            check <= 1;
        else
            check <= 0;
    end else
        check <= 0;
end
```

Since the request can come from any floor at any time, we needed a range checker module for each floor hence we used a total of 5 instances of range checker module in our elevator module. As it was stated at the project pdf, direction from floor 0 can only be UP (1) and direction from floor 4 can only be DOWN (0).

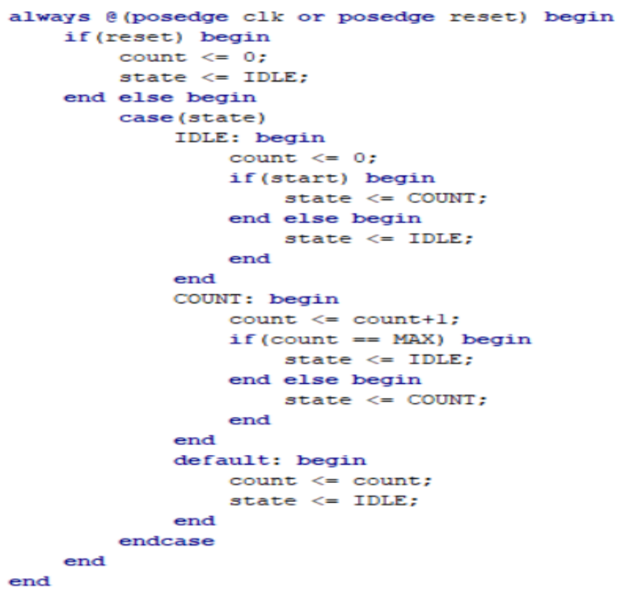
```
range_checker floor_0 (
    .direction(1),
    .elevator_status(status),
    .requested_floor(FLOOR_0),
    .elevator_state(elevator_state),
    .final_dest(final_dest),
    .check(check_0)
);

range_checker floor_1 (
    .direction(direction_1),
    .elevator_status(status),
    .requested_floor(FLOOR_1),
    .elevator_state(elevator_state),
    .final_dest(final_dest),
    .check(check_1)
);

range_checker floor_2 (
    .direction(direction_2),
    .elevator_status(status),
    .requested_floor(FLOOR_2),
    .elevator_state(elevator_state),
    .final_dest(final_dest),
    .check(check_2)
);

range_checker floor_3 (
    .direction(direction_3),
    .elevator_status(status),
    .requested_floor(FLOOR_3),
    .elevator_state(elevator_state),
    .final_dest(final_dest),
    .check(check_3)
);

range_checker floor_4 (
    .direction(0),
    .elevator_status(status),
    .requested_floor(FLOOR_4),
    .elevator_state(elevator_state),
    .final_dest(final_dest),
    .check(check_4)
);
```

Counter Module

In the elevator module, we have defined a counter module to determine the transition between floors, the waiting time at the stopped floor and the duration of the LED to be turned on while waiting at the stopped floor, as 5 seconds. The clock in the counter module and the clock in the elevator module are the same 50 hertz clocks. Since the clock will come in every 50 hertz, that is, in 0.2 seconds, it is necessary to count up to 250 to wait for 5 seconds. Therefore, the MAX value has been determined as 8'b11111010, that is, 250. If the start input from the elevator module is 1, it state is set to COUNT and it start counting otherwise it remains at the IDLE state. In the COUNT state, it increments the count value by 1 each time. If the count value is MAX, it returns to the IDLE state otherwise it remains at the COUNT state. Also, if a reset input is received, the counter set to 0 and state is set to IDLE state.

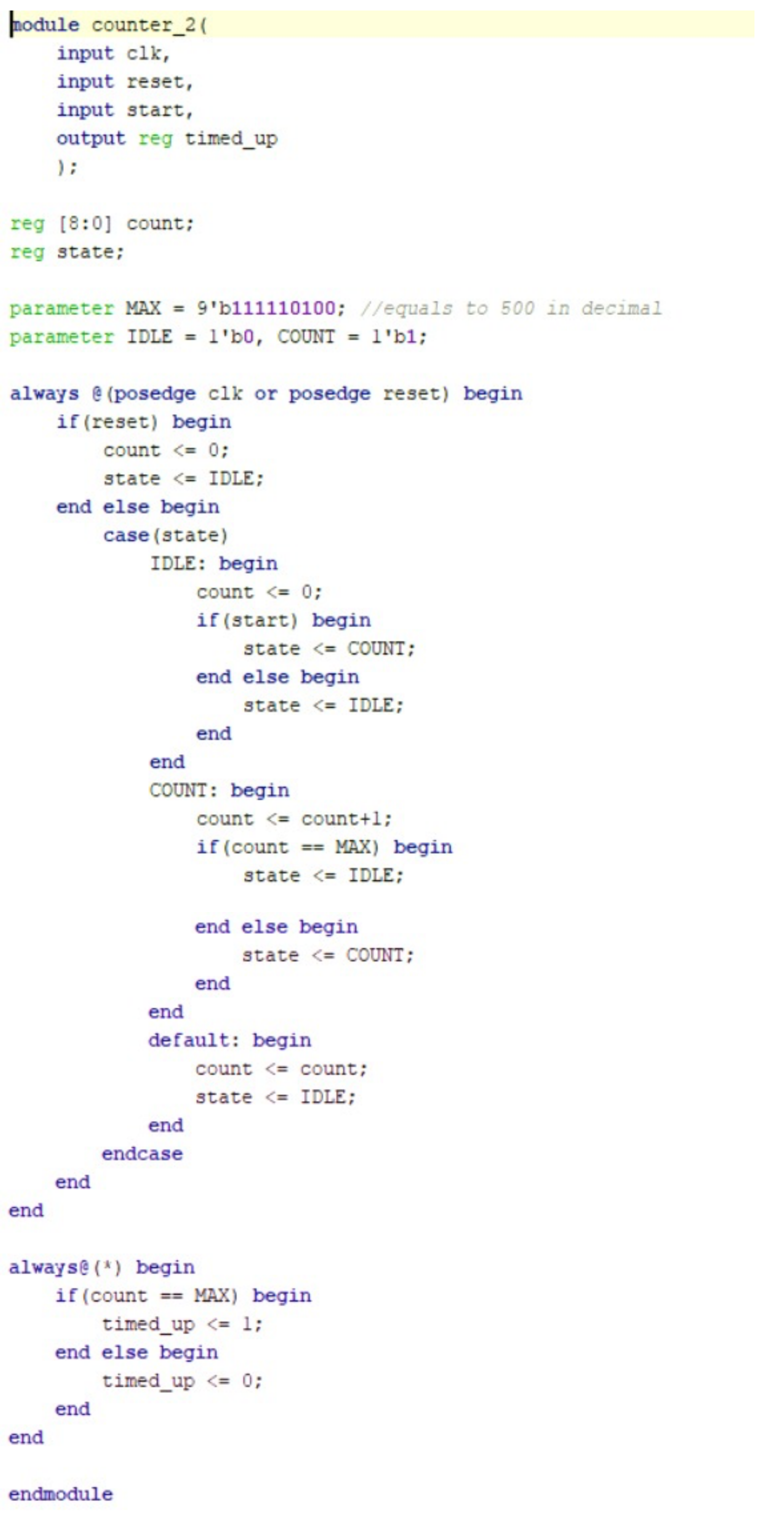


```
module counter(
    input clk,
    input reset,
    input start,
    output reg timed_up
);

reg [7:0] count;
reg state;

parameter MAX = 8'b11111010; //equals to 250 in decimal
parameter IDLE = 1'b0, COUNT = 1'b1;
```



```
always @(posedge clk or posedge reset) begin
    if(reset) begin
        count <= 0;
        state <= IDLE;
    end else begin
        case(state)
            IDLE: begin
                count <= 0;
                if(start) begin
                    state <= COUNT;
                end else begin
                    state <= IDLE;
                end
            end
            COUNT: begin
                count <= count+1;
                if(count == MAX) begin
                    state <= IDLE;
                end else begin
                    state <= COUNT;
                end
            end
            default: begin
                count <= count;
                state <= IDLE;
            end
        endcase
    end
end
```

We defined a second counter module, which has the same logic as in the above described counter module but is used to count up to 500 instead of 250 so that we could maintain the 10 seconds floor transitions when the elevator stops at that particular floor and picks up/drops off a person. The only difference between the codes of the two counter modules is that this time, the MAX value has been determined as 9’b111110100, that is, 500.



```
module counter_2(
    input clk,
    input reset,
    input start,
    output reg timed_up
    );

reg [8:0] count;
reg state;

parameter MAX = 9'b111110100; //equals to 500 in decimal
parameter IDLE = 1'b0, COUNT = 1'b1;

always @(posedge clk or posedge reset) begin
    if(reset) begin
        count <= 0;
        state <= IDLE;
    end else begin
        case(state)
            IDLE: begin
                count <= 0;
                if(start) begin
                    state <= COUNT;
                end else begin
                    state <= IDLE;
                end
            end
            COUNT: begin
                count <= count+1;
                if(count == MAX) begin
                    state <= IDLE;

                end else begin
                    state <= COUNT;
                end
            end
            default: begin
                count <= count;
                state <= IDLE;
            end
        endcase
    end
end

always@(*) begin
    if(count == MAX) begin
        timed_up <= 1;
    end else begin
        timed_up <= 0;
    end
end

endmodule
```

Determining timed_up

In the always block that works in all cases, the timed_up output to be used in the elevator module is sent. If the count is equal to 250, it means that it has waited for 5 seconds and the timed_up value is sent to the elevator module as 1 and is used in various control blocks and statements. Otherwise, timed_up value is set to 0.

```
always@(*) begin
    if(count == MAX) begin
        timed_up <= 1;
    end else begin
        timed_up <= 0;
    end
end
```

At the elevator module, we used 3 different instances of the counter modules in order to determine the transition between floors, the waiting time at the stopped floor and the duration of the LED to be turned on while waiting at the stopped floor, as 5 seconds. Count function is used for changing the elevator state after 5 seconds. Count_2 function is used for changing the elevator state and stopping at the floor for 5 seconds each making a total of 10 seconds. Count_3 function is used for changing the value of led_busy which is 1 if elevator stopped at floor and is on for 5 seconds.

```
counter count (
     .clk(clk_50hz),
     .reset(rst),
     .start(start_timer),
     .timed_up(timed_up)
);

counter count_3 (
     .clk(clk_50hz),
     .reset(rst),
     .start(start_timer_3),
     .timed_up(timed_up_3)
);

counter_2 count_2 (
     .clk(clk_50hz),
     .reset(rst),
     .start(start_timer_2),
     .timed_up(timed_up_2)
);
```

Moreover, to start the timer in these counter models, we have written three always blocks, specifying the conditions to when these counter modules should start counting. Based on these conditions, we start the timers of the counter modules, and they start counting. Here are the always blocks handling these timers:

```
always @* begin
    if (elevator_state != final_dest || led_busy == 1)
        start_timer <= 1;
    else
        start_timer <= 0;
end

always @* begin
    if (led_busy == 1)
        start_timer_2 <= 1;
    else
        start_timer_2 <= 0;
end

always @* begin
    if (led_busy == 1)
        start_timer_3 <= 1;
    else
        start_timer_3 <= 0;
end
```

Elevator Module

Displaying led for requested floor

This always block is for displaying the led for the floor if there was a request made to that floor and elevator has reached to that particular floor. If the request comes from the floor which the elevator currently is, then led_busy will be 1, otherwise if led_busy is 1, that is elevator stopped at that floor, led_busy should be kept 1 for 5 seconds and then be set to 0 using the timed_up_3 value which comes from the counter_3 module. This explanation can be generalized for all floors at this project which can be seen below:

```
always @(posedge clk_50hz or posedge rst) begin
    if (rst) begin
        led_busy <= 0;
    end else begin
        case (elevator_state)
            FLOOR_0: begin
                if (led_inside_0 || led_outside_0) begin
                    led_busy <= 1;
                end else begin
                    if (led_busy == 1) begin
                        if (timed_up_3)
                            led_busy <= 0;
                        else
                            led_busy <= 1;
                    end else
                        led_busy <= 0;
                end
            end
```

```
FLOOR_1: begin
    if (led_inside_1 || led_outside_1) begin
        led_busy <= 1;
    end else begin
        if (led_busy == 1) begin
            if (timed_up_3)
                led_busy <= 0;
            else
                led_busy <= 1;
        end else
            led_busy <= 0;
    end
end

FLOOR_2: begin
    if (led_inside_2 || led_outside_2) begin
        led_busy <= 1;
    end else begin
        if (led_busy == 1) begin
            if (timed_up_3)
                led_busy <= 0;
            else
                led_busy <= 1;
        end else
            led_busy <= 0;
    end
end
```

```
FLOOR_3: begin
    if (led_inside_3 || led_outside_3) begin
        led_busy <= 1;
    end else begin
        if (led_busy == 1) begin
            if (timed_up_3)
                led_busy <= 0;
            else
                led_busy <= 1;
        end else
            led_busy <= 0;
    end
end

FLOOR_4: begin
    if (led_inside_4 || led_outside_4) begin
        led_busy <= 1;
    end else begin
        if (led_busy == 1) begin
            if (timed_up_3)
                led_busy <= 0;
            else
                led_busy <= 1;
        end else
            led_busy <= 0;
    end
end
```

Turning the LEDs Inside and Outside the Elevator On and Off

The logic of our elevator pick-up/drop-off system is that we open the LEDs (either the outside LEDs or the inside LEDs) based on the requests that are accepted and the elevator stops at a particular floor if either the outside led or the inside led in that floor is turned on. Thus, we use the LEDs as an indicator for the elevator to pick up/drop off someone on a particular floor.

Firstly, we were turning the LEDs on and off according to the changes in the push buttons and the destination switches but while implementing the design on the FPGA on the Simulation Demo day, we realized that the push buttons do not stay pressed but go back to zero logic instantaneously. Therefore, to overcome this problem, we defined registers for each push button and destination switch to store the current status of the push buttons and destination switches. In total, we have 5 registers for the push buttons and 5 for the destination switches, corresponding to each floor.

The always code blocks of these 5 registers are the same for all five push button and 5 destination switches. Thus, here I only put the always blocks for the registers corresponding to the push button in floor-0 and the destination switch at floor-0 as example code blocks:

```
always @(posedge clk_50hz or posedge rst) begin
    if (rst) begin
        floor_0_p_reg <= 0;
    end else begin
        if(floor_0_p && check_0)
           floor_0_p_reg <= 1;
        else if(elevator_state == FLOOR_0)
           floor_0_p_reg <= 0;
        else
           floor_0_p_reg <= floor_0_p_reg;
    end
end
```

```
always @(posedge clk_50hz or posedge rst) begin
    if (rst) begin
        floor_0_d_reg <= 0;
    end else begin
        if(floor_0_d && check_0)
           floor_0_d_reg <= 1;
        else if(elevator_state == FLOOR_0)
           floor_0_d_reg <= 0;
        else
           floor_0_d_reg <= floor_0_d_reg;
    end
end
```

Once a push button outside the elevator or a destination switch inside the elevator is turned on, we process the request, conduct a range check and decide whether to accept the request or not. If we accept the request, we make the value of the corresponding register logic 1 and then turn on the corresponding LED in another always block. In this way, we achieve to turn on the LEDs once a request is accepted and hold them turned on until the elevator reaches that particular floor and picks up/drops off the person.

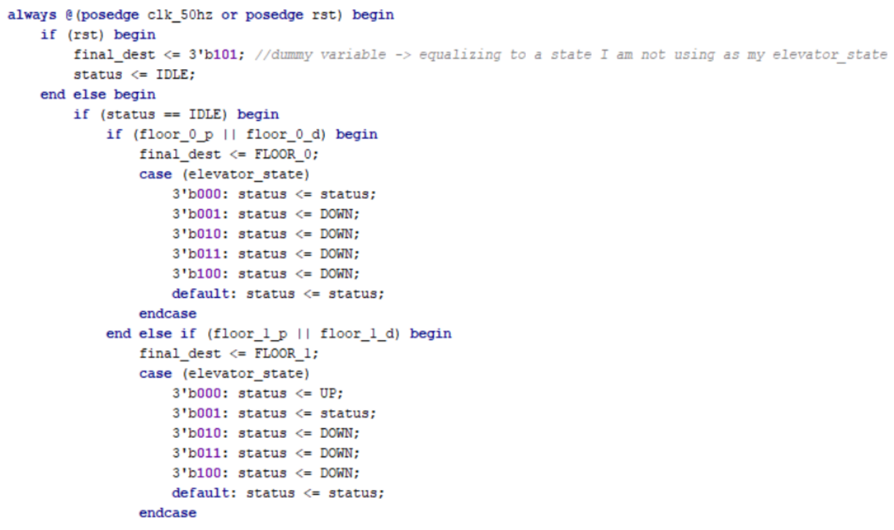
Likewise the always blocks we assign values to the registers, the always blocks where we make assignments to the LEDs are the same for all the LEDs inside and outside the elevator. So, here I only put the always blocks for the registers corresponding to the LED inside corresponding to floor-0 and the LED outside at floor-0 as example code blocks:

```
always @(posedge clk_50hz or posedge rst) begin
    if (rst) begin
        led_inside_0 <= 0;
    end else begin
        if(floor_0_d_reg) //turn on the inside led when the floor 0 destination switch is turned on
           led_inside_0 <= 1;
        else
           led_inside_0 <= 0;
    end
end
```

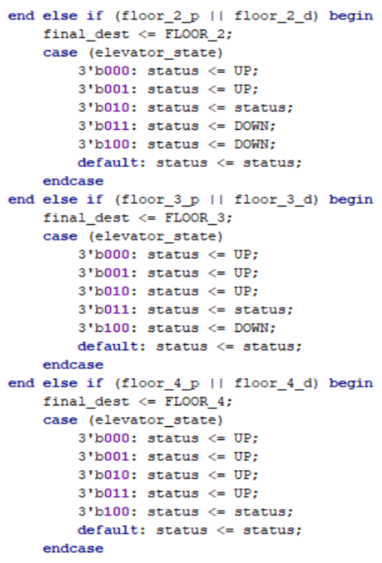
```
always @(posedge clk_50hz or posedge rst) begin
    if (rst) begin
        led_outside_0 <= 0;
    end else begin
        if(floor_0_p_reg) //turn on the led when the push button at floor 0 is pressed and floor 0 is in range
           led_outside_0 <= 1;
        else
           led_outside_0 <= 0;
    end
end
```

Determining final destination and status (behavior) of the elevator

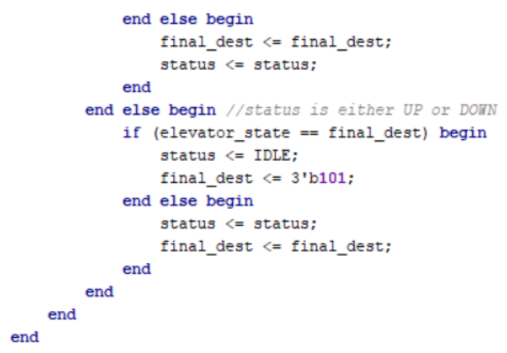
At this always block, which works at positive edge of clock or positive edge of reset, the final destination of elevator and status of the elevator is determined. Final destination can only be set to a floor when the current status of elevator is IDLE, hence we first check the status of elevator. If there is a request from floor n and elevator is at a floor below floor n, then status of the elevator should be UP with final destination being set to floor n. Similarly, if there is a request from floor n and elevator is at a floor above floor n, then status of the elevator should be DOWN with final destination being set to floor n. If the request comes from the floor in which the elevator is currently at, then status is preserved. When status of elevator is IDLE and there aren't any requests coming from any floor, then both status and final destination of elevator is preserved. Another possibility is that the status of elevator also can be UP or DOWN hence those conditions should also be considered that is while the elevator hasn't reached its final destination, both status and final destination of elevator is preserved. Otherwise if elevator has reached its final destination, then the status of elevator is set to IDLE and final destination is set to 3'b101 which is another way of saying that elevator is at IDLE status. At the screenshots below, you can see the implementation of the explanation above:



```
always @(posedge clk_50hz or posedge rst) begin
    if (rst) begin
        final_dest <= 3'b101; //dummy variable -> equalizing to a state I am not using as my elevator_state
        status <= IDLE;
    end else begin
        if (status == IDLE) begin
            if (floor_0_p || floor_0_d) begin
                final_dest <= FLOOR_0;
                case (elevator_state)
                    3'b000: status <= status;
                    3'b001: status <= DOWN;
                    3'b010: status <= DOWN;
                    3'b011: status <= DOWN;
                    3'b100: status <= DOWN;
                    default: status <= status;
                endcase
            end else if (floor_1_p || floor_1_d) begin
                final_dest <= FLOOR_1;
                case (elevator_state)
                    3'b000: status <= UP;
                    3'b001: status <= status;
                    3'b010: status <= DOWN;
                    3'b011: status <= DOWN;
                    3'b100: status <= DOWN;
                    default: status <= status;
                endcase
```



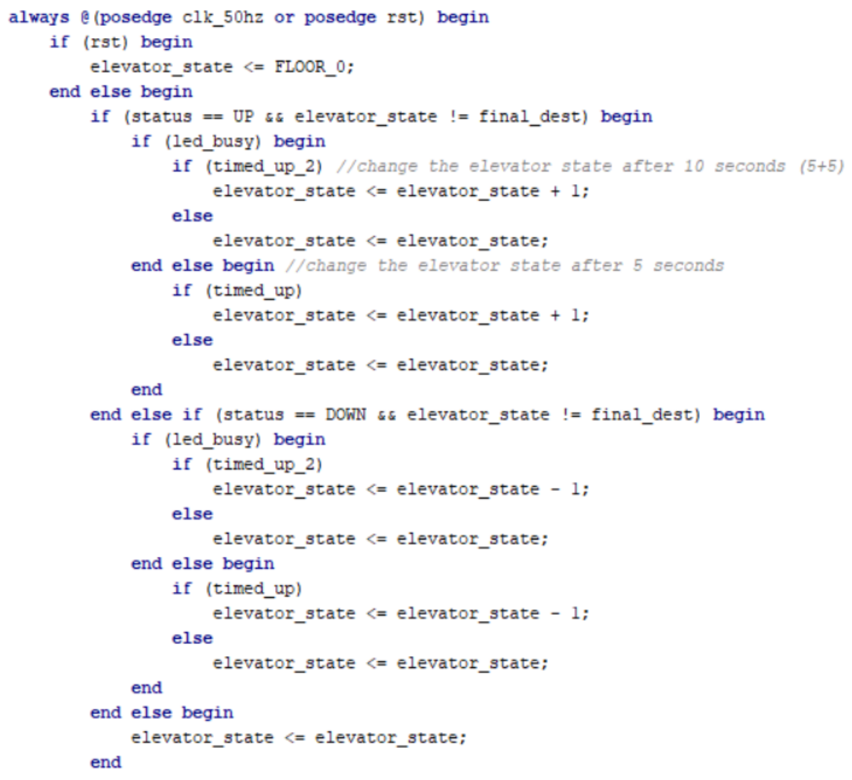
```
            end else if (floor_2_p || floor_2_d) begin
                final_dest <= FLOOR_2;
                case (elevator_state)
                    3'b000: status <= UP;
                    3'b001: status <= UP;
                    3'b010: status <= status;
                    3'b011: status <= DOWN;
                    3'b100: status <= DOWN;
                    default: status <= status;
                endcase
            end else if (floor_3_p || floor_3_d) begin
                final_dest <= FLOOR_3;
                case (elevator_state)
                    3'b000: status <= UP;
                    3'b001: status <= UP;
                    3'b010: status <= UP;
                    3'b011: status <= status;
                    3'b100: status <= DOWN;
                    default: status <= status;
                endcase
            end else if (floor_4_p || floor_4_d) begin
                final_dest <= FLOOR_4;
                case (elevator_state)
                    3'b000: status <= UP;
                    3'b001: status <= UP;
                    3'b010: status <= UP;
                    3'b011: status <= UP;
                    3'b100: status <= status;
                    default: status <= status;
                endcase
```



```
            end else begin
                final_dest <= final_dest;
                status <= status;
            end
        end else begin //status is either UP or DOWN
            if (elevator_state == final_dest) begin
                status <= IDLE;
                final_dest <= 3'b101;
            end else begin
                status <= status;
                final_dest <= final_dest;
            end
        end
    end
end
```

Transition between floors

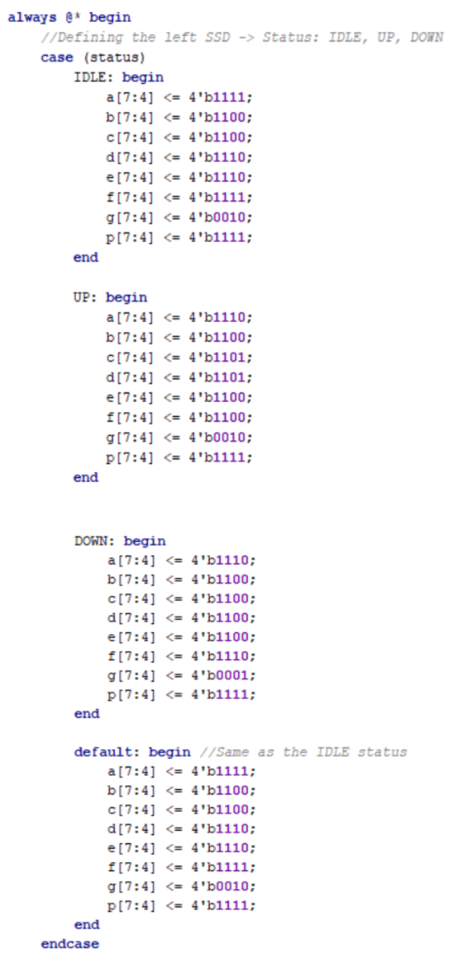
This always block, which works at positive edge of clock or positive edge of reset, is used for transition between floors of the elevator. If the elevator is moving up and didn’t reach its final destination and there is a valid request from the floors between current floor and final destination, then it waits at the requested floor for 5 seconds and then goes to the floor above at 5 seconds making a total of 10 seconds. However, if there wasn’t any request from the floors between, then the elevator will go the floor above at 5 seconds. Similarly, if the elevator is moving down and didn’t reach its final destination and there is a valid request from the floors between current floor and final destination, then it waits at the requested floor for 5 seconds and then goes to the floor below at 5 seconds making a total of 10 seconds. However, if there wasn’t any request from the floors between, then the elevator will go the floor below at 5 seconds. If there is a reset input give to the module, then the current floor of elevator is set to floor 0.



```
always @(posedge clk_50hz or posedge rst) begin
    if (rst) begin
        elevator_state <= FLOOR_0;
    end else begin
        if (status == UP && elevator_state != final_dest) begin
            if (led_busy) begin
                if (timed_up_2) //change the elevator state after 10 seconds (5+5)
                    elevator_state <= elevator_state + 1;
                else
                    elevator_state <= elevator_state;
            end else begin //change the elevator state after 5 seconds
                if (timed_up)
                    elevator_state <= elevator_state + 1;
                else
                    elevator_state <= elevator_state;
            end
        end else if (status == DOWN && elevator_state != final_dest) begin
            if (led_busy) begin
                if (timed_up_2)
                    elevator_state <= elevator_state - 1;
                else
                    elevator_state <= elevator_state;
            end else begin
                if (timed_up)
                    elevator_state <= elevator_state - 1;
                else
                    elevator_state <= elevator_state;
            end
        end else begin
            elevator_state <= elevator_state;
        end
```

Left side of SSD

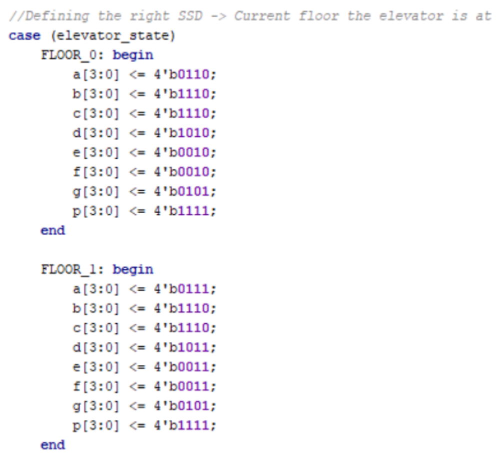
For the left side of the SSD, we had to define left side according to the elevator’s status which can be Idle, UP, or Down and since it had to shown at all times, the always block was used without any constraints coming from clock or reset button that is it always checked for the current status of elevator. The coding of the letters was done as requested in the project file which can be seen below.



```
always @* begin
    //Defining the left SSD -> Status: IDLE, UP, DOWN
    case (status)
        IDLE: begin
            a[7:4] <= 4'b1111;
            b[7:4] <= 4'b1100;
            c[7:4] <= 4'b1100;
            d[7:4] <= 4'b1110;
            e[7:4] <= 4'b1110;
            f[7:4] <= 4'b1111;
            g[7:4] <= 4'b0010;
            p[7:4] <= 4'b1111;
        end

        UP: begin
            a[7:4] <= 4'b1110;
            b[7:4] <= 4'b1100;
            c[7:4] <= 4'b1101;
            d[7:4] <= 4'b1101;
            e[7:4] <= 4'b1100;
            f[7:4] <= 4'b1100;
            g[7:4] <= 4'b0010;
            p[7:4] <= 4'b1111;
        end


        DOWN: begin
            a[7:4] <= 4'b1110;
            b[7:4] <= 4'b1100;
            c[7:4] <= 4'b1100;
            d[7:4] <= 4'b1100;
            e[7:4] <= 4'b1100;
            f[7:4] <= 4'b1110;
            g[7:4] <= 4'b0001;
            p[7:4] <= 4'b1111;
        end

        default: begin //Same as the IDLE status
            a[7:4] <= 4'b1111;
            b[7:4] <= 4'b1100;
            c[7:4] <= 4'b1100;
            d[7:4] <= 4'b1110;
            e[7:4] <= 4'b1110;
            f[7:4] <= 4'b1111;
            g[7:4] <= 4'b0010;
            p[7:4] <= 4'b1111;
        end
    endcase
```

Right side of SSD

For the right side of the SSD, we had to define right side according to the current floor of elevator which can be Floor 0, Floor 1, Floor 2, Floor 3 or Floor 4 and since it had to shown at all times, the always block was used without any constraints coming from clock or reset button that is it always checked for the current floor of elevator. The coding of the letters was done as requested in the project file which can be seen below.



```
//Defining the right SSD -> Current floor the elevator is at
case (elevator_state)
    FLOOR_0: begin
        a[3:0] <= 4'b0110;
        b[3:0] <= 4'b1110;
        c[3:0] <= 4'b1110;
        d[3:0] <= 4'b1010;
        e[3:0] <= 4'b0010;
        f[3:0] <= 4'b0010;
        g[3:0] <= 4'b0101;
        p[3:0] <= 4'b1111;
    end

    FLOOR_1: begin
        a[3:0] <= 4'b0111;
        b[3:0] <= 4'b1110;
        c[3:0] <= 4'b1110;
        d[3:0] <= 4'b1011;
        e[3:0] <= 4'b0011;
        f[3:0] <= 4'b0011;
        g[3:0] <= 4'b0101;
        p[3:0] <= 4'b1111;
    end
```

```
        FLOOR_2: begin
            a[3:0] <= 4'b0110;
            b[3:0] <= 4'b1110;
            c[3:0] <= 4'b1111;
            d[3:0] <= 4'b1010;
            e[3:0] <= 4'b0010;
            f[3:0] <= 4'b0011;
            g[3:0] <= 4'b0100;
            p[3:0] <= 4'b1111;
        end

        FLOOR_3: begin
            a[3:0] <= 4'b0110;
            b[3:0] <= 4'b1110;
            c[3:0] <= 4'b1110;
            d[3:0] <= 4'b1010;
            e[3:0] <= 4'b0011;
            f[3:0] <= 4'b0011;
            g[3:0] <= 4'b0100;
            p[3:0] <= 4'b1111;
        end


        FLOOR_4: begin
            a[3:0] <= 4'b0111;
            b[3:0] <= 4'b1110;
            c[3:0] <= 4'b1110;
            d[3:0] <= 4'b1011;
            e[3:0] <= 4'b0011;
            f[3:0] <= 4'b0010;
            g[3:0] <= 4'b0100;
            p[3:0] <= 4'b1111;
        end

        default: begin //Same as FLOOR-0
            a[3:0] <= 4'b0110;
            b[3:0] <= 4'b1110;
            c[3:0] <= 4'b1110;
            d[3:0] <= 4'b1010;
            e[3:0] <= 4'b0010;
            f[3:0] <= 4'b0010;
            g[3:0] <= 4'b0101;
            p[3:0] <= 4'b1111;
        end
    endcase
end
```