Logic Design Final Project

8-STORY ELEVATOR

Matthew Kerr & Mario Palacios | EEE 64 | December 5th, 2019

Table of Contents

Specifications

The goal of this design was to create a Verilog-based logic device capable of acting as an 8-story elevator. This would be achieved via a three-state State Machine. The three states would be Idle (when the elevator has no input), Loading (when the elevator is picking up/dropping people off), and Moving (when the elevator is traveling to a different floor).

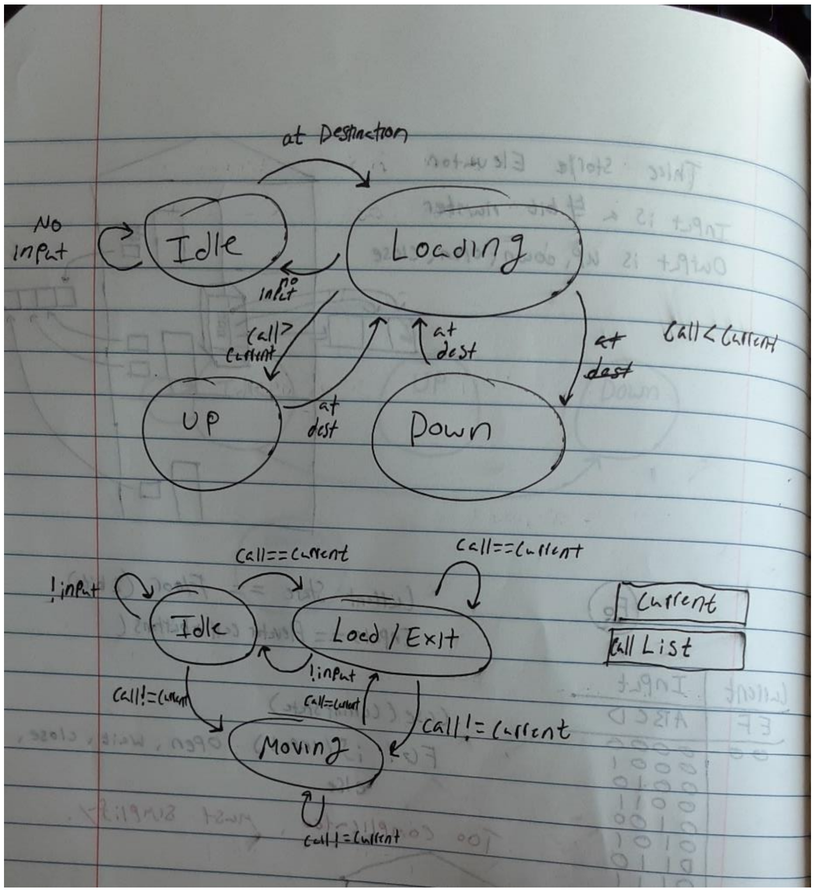
The elevator was to have 8 "Request Buttons" inside the elevator to request the elevator to take a person to a certain floor. The elevator would also have 8 exterior "Call Buttons", one on each floor, which people could use to call the elevator to their floor. Note that in this design, a person simply calls the elevator to their floor; they do not specify the direction they would like to move until they are inside the elevator.

To make the system more efficient, the elevator was to find the nearest requested/called floor and make that a priority before moving on to other floors. This would require the use of a list of required floors which could be modified to keep it updated as the elevator completed its trips.

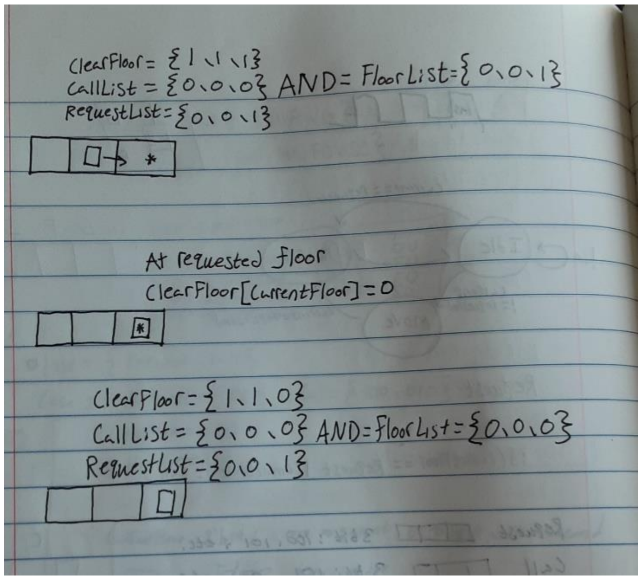
All in all, there would be 16 inputs from the 8 call + 8 request buttons, and a total of 16 outputs from the 7-segment display for the current floor counter, a 2-bit current state indicator, and a 7-bit floor list display. This would be tested via waveform and then implemented using a breadboard and FPGA.

Design Process

To begin with, a smaller three-story elevator was attempted. A number of different state diagrams were sketched, with varying levels of complexity, until the design shown below was selected. This simple state machine had three states: Idle, Loading, and Moving. The Machine would move between these states according to various logic assignments involving the 16 inputs.

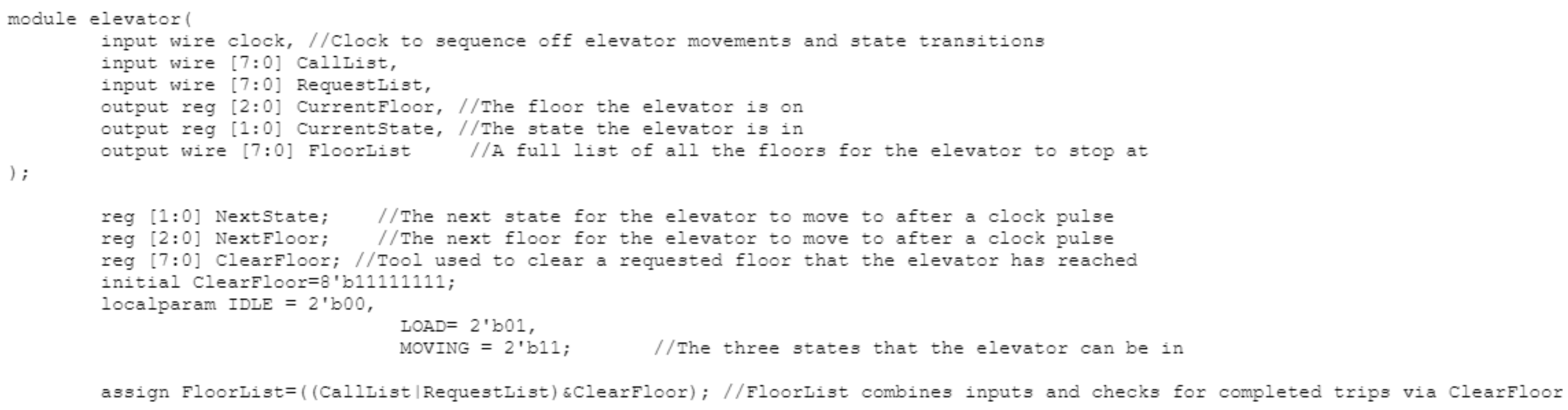


The next step was to determine how to delete a floor from the floor list once the elevator had successfully completed a trip. This was done by using an initially all high 3-bit number which could be ANDed with the floor list to remove specific floors. An example of this is included below.


ClearFloor = {1,1,1}
CallList = {0,0,0} AND= FloorList={0,0,1}
RequestList={0,0,1}
At requested floor
ClearFloor[CurrentFloor]=0
ClearFloor={1,1,0}
CallList={0,0,0} AND=FloorList={0,0,0}
RequestList={0,0,1}

The next task was to design the logic assignments. This was achieved by programing one state at a time. Once completed, the building was expanded to five floors and then the final eight floors. This was done by modifying the moving state and expanding the registers.

Functionality

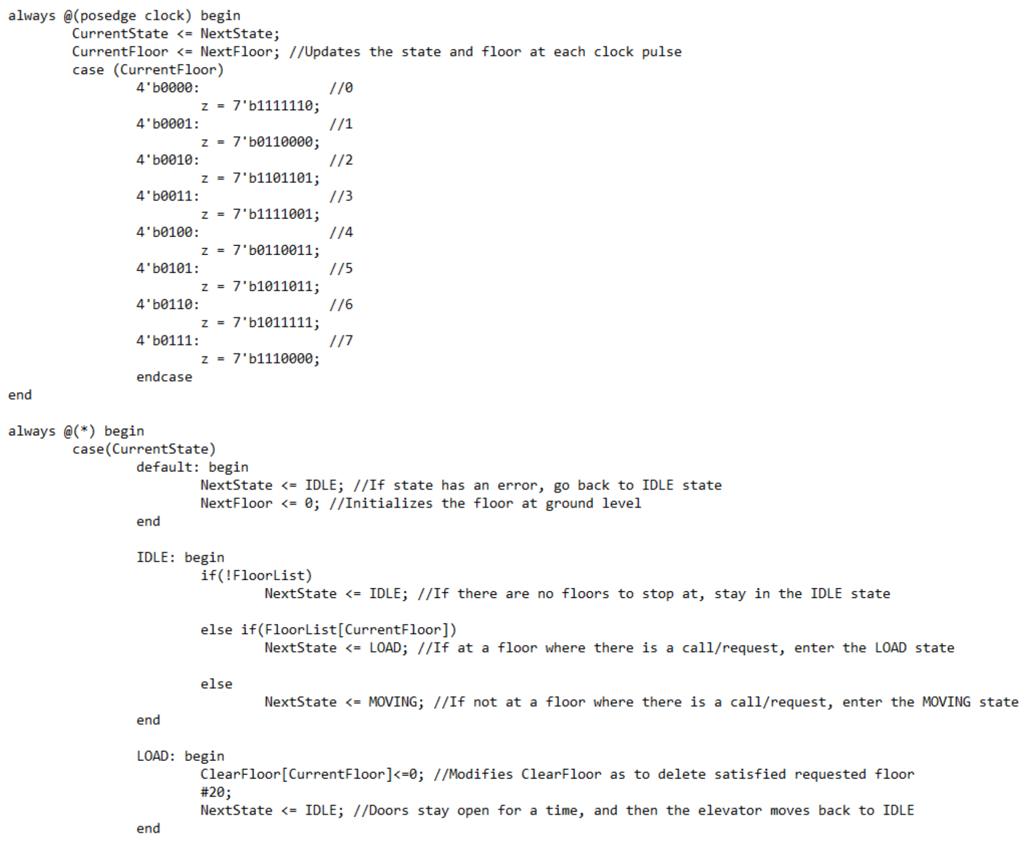


```
module elevator(
        input wire clock, //Clock to sequence off elevator movements and state transitions
        input wire [7:0] CallList,
        input wire [7:0] RequestList,
        output reg [2:0] CurrentFloor, //The floor the elevator is on
        output reg [1:0] CurrentState, //The state the elevator is in
        output wire [7:0] FloorList     //A full list of all the floors for the elevator to stop at
);

        reg [1:0] NextState;    //The next state for the elevator to move to after a clock pulse
        reg [2:0] NextFloor;    //The next floor for the elevator to move to after a clock pulse
        reg [7:0] ClearFloor; //Tool used to clear a requested floor that the elevator has reached
        initial ClearFloor=8'b11111111;
        localparam IDLE = 2'b00,
                                LOAD= 2'b01,
                                MOVING = 2'b11;       //The three states that the elevator can be in

        assign FloorList=((CallList|RequestList)&ClearFloor); //FloorList combines inputs and checks for completed trips via ClearFloor
```

As stated previously, this elevator works on 8 floors. The clock acts to sequence out all state transitions as well as floor movements. There are two 8-bit input registers which keep track of the 16 different buttons the elevator responds to. The CallList input register is for the buttons outside the elevator, while the RequestList input register is for all the buttons inside the elevator.

For the outputs, there is a three-bit register which keeps track of the CurrentFloor, ranging from zero (ground floor) to seven (top floor). There is also a two-bit register which keeps track of the state. These states are defined as local parameters where 2'b00 is Idle, 2'b01 is Loading, and 2'b11 is Moving.

Finally, there is an 8-bit FloorList output register which AND's the two input registers with a ClearFloor register in order to keep an updated list of all required floors for the elevator to stop at.

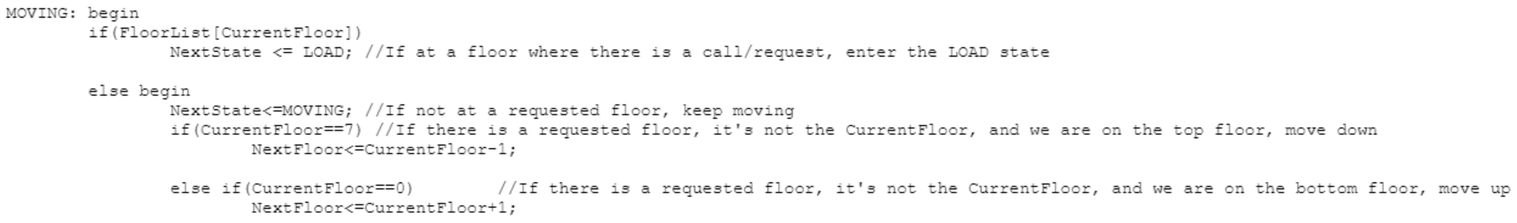


```
always @(posedge clock) begin
        CurrentState <= NextState;
        CurrentFloor <= NextFloor; //Updates the state and floor at each clock pulse
        case (CurrentFloor)
                4'b0000:                //0
                        z = 7'b1111110;
                4'b0001:                //1
                        z = 7'b0110000;
                4'b0010:                //2
                        z = 7'b1101101;
                4'b0011:                //3
                        z = 7'b1111001;
                4'b0100:                //4
                        z = 7'b0110011;
                4'b0101:                //5
                        z = 7'b1011011;
                4'b0110:                //6
                        z = 7'b1011111;
                4'b0111:                //7
                        z = 7'b1110000;
                endcase
end

always @(*) begin
        case(CurrentState)
                default: begin
                        NextState <= IDLE; //If state has an error, go back to IDLE state
                        NextFloor <= 0; //Initializes the floor at ground level
                end

                IDLE: begin
                        if(!FloorList)
                                NextState <= IDLE; //If there are no floors to stop at, stay in the IDLE state

                        else if(FloorList[CurrentFloor])
                                NextState <= LOAD; //If at a floor where there is a call/request, enter the LOAD state

                        else
                                NextState <= MOVING; //If not at a floor where there is a call/request, enter the MOVING state
                end

                LOAD: begin
                        ClearFloor[CurrentFloor]<=0; //Modifies ClearFloor as to delete satisfied requested floor
                        #20;
                        NextState <= IDLE; //Doors stay open for a time, and then the elevator moves back to IDLE
                end
```

The body of the program consists of two always blocks. The first block serves to update the CurrentState and CurrentFloor. It also translates CurrentFloor into a form the 7-segment display can read. The second always block is the case system which contains the majority of the state machine.

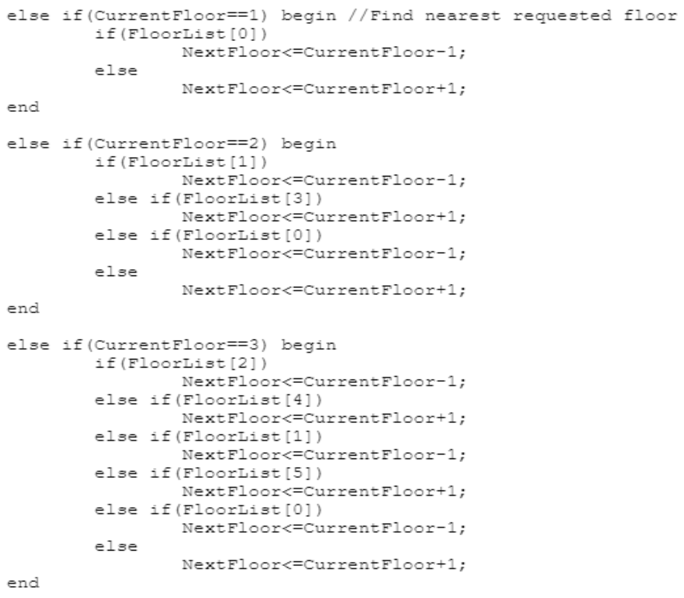
The Idle state serves to act as a stationary period where the elevator awaits input from the 16 buttons. The elevator remains Idle if there are no inputs (i.e. the floor list is empty). If the floor list is not empty, the program checks to see if the current floor is a requested floor. If this is the case then the elevator transfers to the Loading state. If neither of these are the case, the elevator transfers to the Moving state.

The Loading state was designed to act as the time where the elevator allows people to enter or leave by opening its doors. The Loading state is also when a trip has been successfully completed, so it is tasked with deleting the floor from the FloorList. In order to simulate the elevator doors opening and closing before awaiting input, the Loading state has a time delay and then transfer back to the Idle state.



```
MOVING: begin
        if(FloorList[CurrentFloor])
                NextState <= LOAD; //If at a floor where there is a call/request, enter the LOAD state

        else begin
                NextState<=MOVING; //If not at a requested floor, keep moving
                if(CurrentFloor==7) //If there is a requested floor, it's not the CurrentFloor, and we are on the top floor, move down
                        NextFloor<=CurrentFloor-1;

                else if(CurrentFloor==0)        //If there is a requested floor, it's not the CurrentFloor, and we are on the bottom floor, move up
                        NextFloor<=CurrentFloor+1;
```

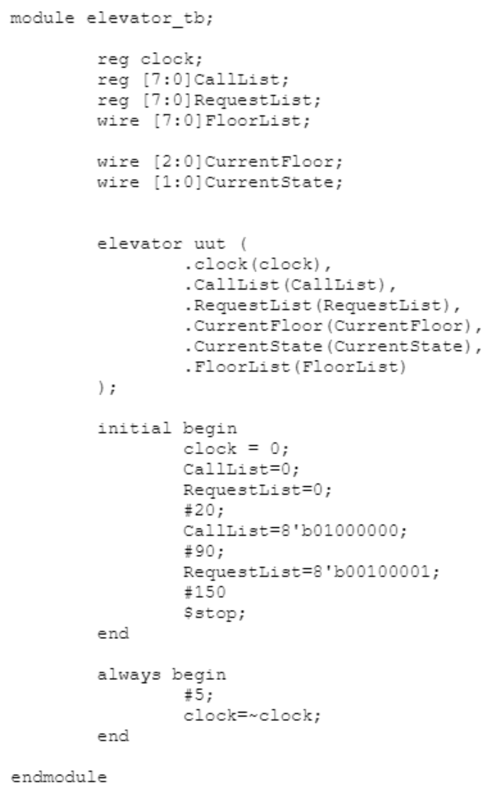
The Moving state is defined as the state where the elevator is moving towards a floor which is on the FloorList. Its first task is to check if it has completed its trip. If this is the case, the elevator transfers to the Loading state. If not, the elevator must determine which direction to move. There are two situations where this is simple. If the elevator is on the top floor, there is a requested floor, and that request is not the current floor, then the elevator must have to move down. Similar logic can be applied if the elevator is on the ground floor, except the elevator must now move up.



```
else if(CurrentFloor==1) begin //Find nearest requested floor
        if(FloorList[0])
                NextFloor<=CurrentFloor-1;
        else
                NextFloor<=CurrentFloor+1;
end

else if(CurrentFloor==2) begin
        if(FloorList[1])
                NextFloor<=CurrentFloor-1;
        else if(FloorList[3])
                NextFloor<=CurrentFloor+1;
        else if(FloorList[0])
                NextFloor<=CurrentFloor-1;
        else
                NextFloor<=CurrentFloor+1;
end

else if(CurrentFloor==3) begin
        if(FloorList[2])
                NextFloor<=CurrentFloor-1;
        else if(FloorList[4])
                NextFloor<=CurrentFloor+1;
        else if(FloorList[1])
                NextFloor<=CurrentFloor-1;
        else if(FloorList[5])
                NextFloor<=CurrentFloor+1;
        else if(FloorList[0])
                NextFloor<=CurrentFloor-1;
        else
                NextFloor<=CurrentFloor+1;
end
```

If the elevator is not on the top/ bottom floor, there must be more thorough analysis to determine the closest requested floor. This is done by alternating checks of the floors above and below the elevator. The more floors above/ below, the more checks the elevator has to complete. This method ensures that the elevator is never searching for floors higher than the top floor or lower than the ground floor.

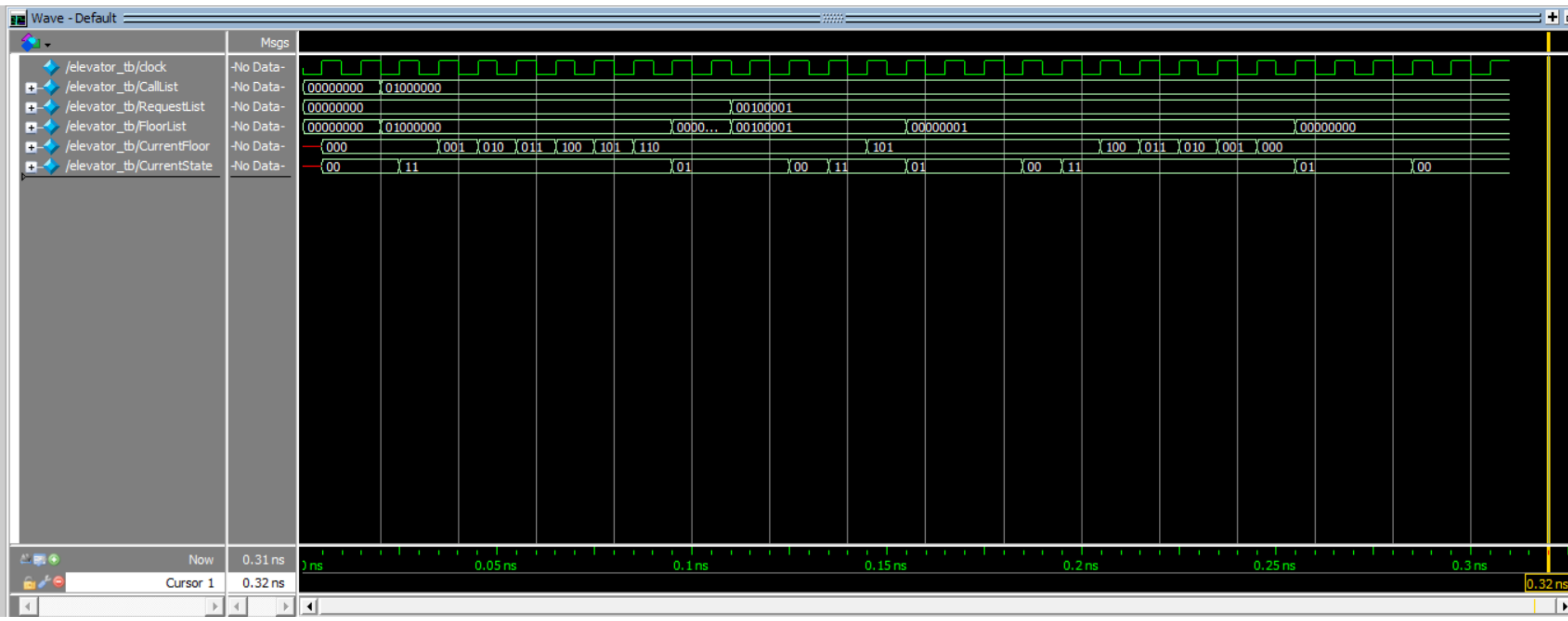
Test Bench



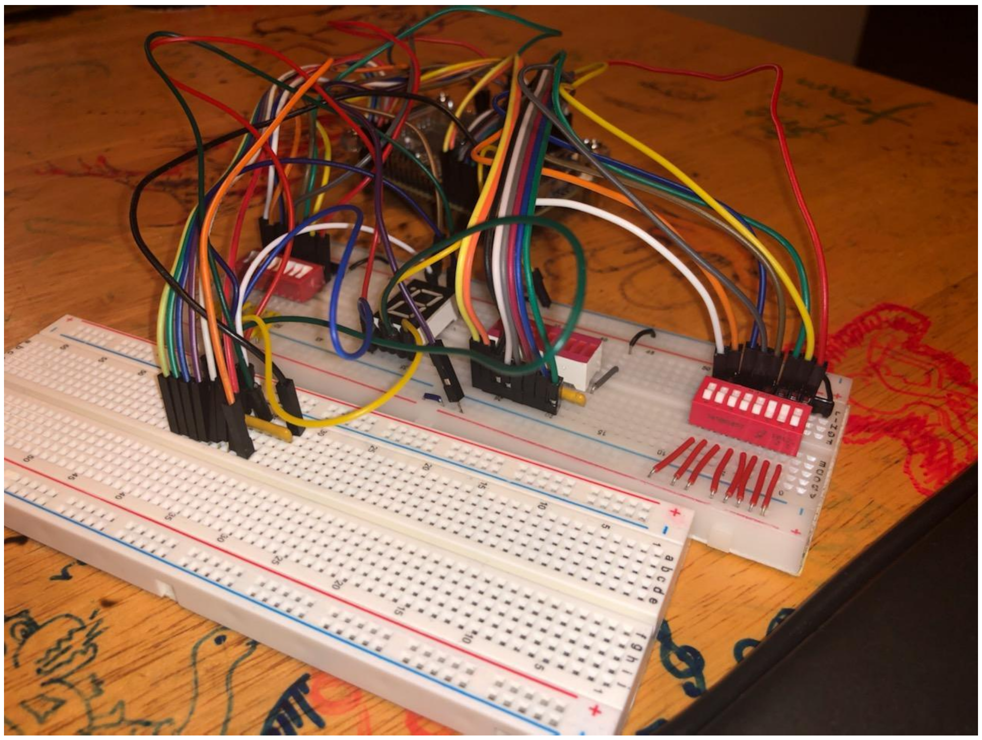
```
module elevator_tb;

        reg clock;
        reg [7:0]CallList;
        reg [7:0]RequestList;
        wire [7:0]FloorList;

        wire [2:0]CurrentFloor;
        wire [1:0]CurrentState;


        elevator uut (
                .clock(clock),
                .CallList(CallList),
                .RequestList(RequestList),
                .CurrentFloor(CurrentFloor),
                .CurrentState(CurrentState),
                .FloorList(FloorList)
        );

        initial begin
                clock = 0;
                CallList=0;
                RequestList=0;
                #20;
                CallList=8'b01000000;
                #90;
                RequestList=8'b00100001;
                #150
                $stop;
        end

        always begin
                #5;
                clock=~clock;
        end

endmodule
```

The test bench is initialized with all the input and output variables from the main program. The clock has been set to a 5ms delay in between inversions. Initially, the clock, CallList, and RequestList are all set to zero to ensure the elevator starts on a clean slate. After a short delay, the elevator receives a call on the sixth floor (as seen by the b'01000000 CallList assignment). The elevator is then given a larger delay to respond to this call. Next, two requests are made for the fifth floor and ground floor (b'00100001). This time the elevator is given an even larger delay to respond to the multiple requests.

Demonstration



The waveform shows the successful simulation of the elevator as described in the test bench described earlier. The elevator starts on the ground floor (CurrentFloor=3'b000) and in the Idle state (CurrentState=2'b00). Note that the elevator doesn't start moving (CurrentState=2'b11) until the rising clock edge after an input has been applied. The elevator moves up the building one floor at a time at each rising clock edge, as seen by the CurrentFloor value increasing. Once the elevator reaches the floor, it goes into the Load state (CurrentState=2'b01), and waits for a period of time before moving to the Idle state. Note also that the FloorList is modified to that floor six has been removed because the trip has been completed. The elevator is also demonstrated to be functional when two separate requests are made simultaneously.



For the FPGA implementation, two separate breadboards were used to allow for more room. Two 8-input switches were used for RequestList and CallList. A seven-segment display was used to display the CurrentFloor and an 8 in-line LED bar was used to display the FloorList. All of these were connected to the FPGA via male-female wires.

Conclusion

In the end, the elevator functioned as we wanted. It is capable of transitioning between the three Idle, Loading, and Moving states according to its 16 inputs and also determines which path is optimal in order to pick up and drop off passengers efficiently. This project was a perfect example of how we take everyday technology for granted. To get to lab every week most people use an elevator, with many of us hardly considering how it works. Working on this project has humbled me, as even this simplified version took plenty of work and a lot of planning to function properly. It is easy to consider everyday objects to be simple until one attempts to create it themselves.