LAB 6

Team 6

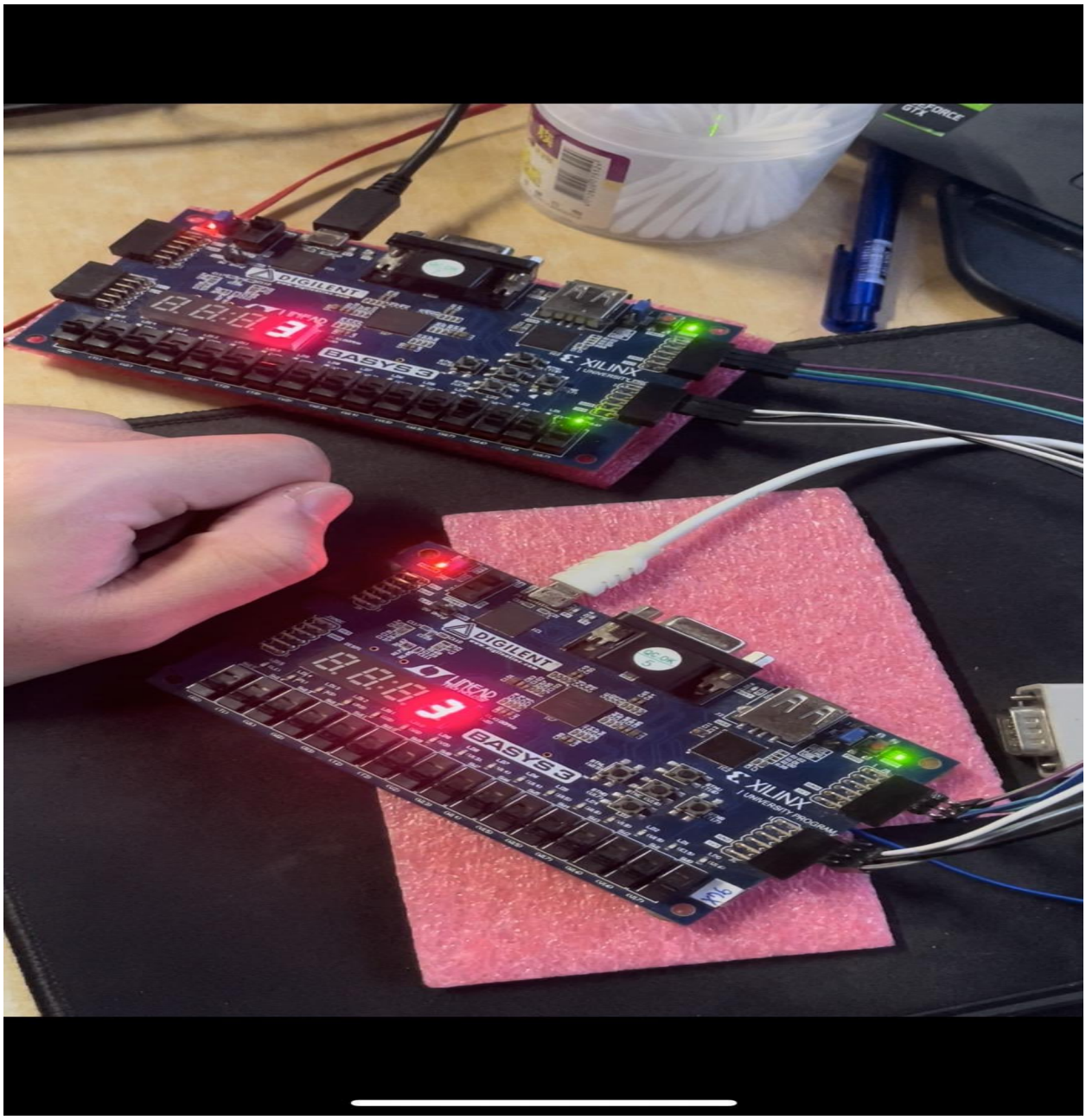
Peripheral Components: VGA, Mouse, DUAL FPGA

鄭聰明 **Jason Theodorus Pratama (109006236)**

鐵特德 **Tuguldur Tserenbaljir (109006271)**

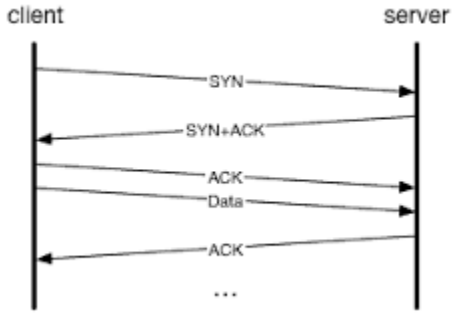
Table of contents:

1. **Dual FPGA Communication**



The concept of Chip2Chip Master-Slave Communication is similar to a class we are taking right now which is computer networks, while on Computer networks are based on Client and Server

Communication, while in Hardware Lab, it is about Master and Slave, but the concept is quite similar.



A. Design

For this advanced question, our assignment was to design a simple FPGA-to-FPGA communication protocol with the Handshake method. We first examined the sample code given to us, drew our Block and State diagrams, and implemented our code.

To start our assignment we first designed our Block and State diagrams. We design our block diagram for the slave and master separately (**Figure 1.1** and **Figure 1.2**). Then we put them together into a big block diagram (**Figure 1.3**). For the slave block diagram (**Figure 1.2**) we have the **Debounce** and **OnePulse** modules for the buttons. Then our slave_control module will control the main algorithm for the block diagram. The algorithm to calculate our slave_control will explain below later on. Once the slave_control outputs the data it will be decoded and passed to the slave top module. Then it will give the ACK to the master and at the same time, it will output the notice and the data on the 7-segment display on the **FPGA**. On the other hand, the Master will first send the request to the slave and get the **ACK** to sending the number to the slave to output. In the master top module, it will get the request, notice, valid, and data-to-slave variables from the **master_control**. The algorithm for the **master_control** where we get the needed variables are explained below later. Similar to the slave, it will have the buttons to request and reset where **Debounce** and **OnePulse** handle it in the master top module. The master top module will first encode the data in order to be sent to the slave.

Upon getting the **ACK** and data to be sent to the slave is shown on the 7-segment display on the **FPGA** where we can control the data by the switches. Finally, the final block diagram is shown down below (**Figure 1.3**).

The next step was to implement our algorithm in code. For our code, we have two Verilog files, master and slave. Each Verilog file has a counter, seven segments, a top, and a slave or a master Verilog module.

The slave and the master have both the same counter where the counter will only start to count when the **reset** is **false (1'b0)** and the **start** is also **true (1'b1)**. Thus there is the condition where if the count reaches **27'd100000000** it will reset the count back to zero and set the variable **done** to **1'b1**. The done is to determine if the **count** has counted to 1 second.

```
            if (start) begin
                if (count == 27'd100000000) begin
                    done = 1;
                    next_count = 0;
                end
                else begin
                    next_count = count + 1;
                    done = 0;
```

In addition, to the **counter** module, they both have the same **decoder** and **seven segment** modules. To get decode the number and display it on the 7-segment on the FPGA board. However, the master has an encoder module that the slave does not have (**Shown below**). The purpose of the encoder is to follow our design where the master can use one hot switch input to binary encoding.

```
module encoder(in, out);
    input [8-1:0] in;
    output reg [3-1:0] out;
    always@(*) begin
        case(in)
            8'b0000_0001: out = 3'd0;
            8'b0000_0010: out = 3'd1;
           /// so on
            default: out = 0;
         endcase
```

Also, both master and slave have a debounce and one pulse module because we need the **reset** button and the **ACK** request button. The purpose of these buttons is to pass signals and perform debounce one pulse.

```
    always @(posedge clk)begin
        DFF[3:1] <= DFF[2:0];
        DFF[0] <= pb;
    end
    assign pb_debounced = ((DFF == 4'b1111) ? 1'b1 : 1'b0);
```

Now that we have every module that we need for our top module we can put them together, exempt to the **master_control** and **slave control** which we will cover after the **top** module. The top module both for the slave and master control was given to us with the **master_control module**. Our most important job is to finish implementing the **salve_control** module for the slave Verilog file. Inside the module, we have three states where they are **state_wait_rqst**, **state_wait_to_send_ack**, and **state_wait_data**. In the first state, we wait until we get the request from the master, and depending on the request we decide to move to the next or stay in the **state_wait_rqst.** The main purpose of the first state is to ensure that we can start our determined flow where the request is needed to be sent, otherwise, it will try again where all the values of variables such as the **ack**, and **notice** are set to their initial 0 except for data. When the request button is pressed it will go to the next state **state_wait_to_send_ack.** In this state, we will wait for 1 sec otherwise **27'd100000000** where the **done** is **true** (**1**) which the counter module calculates for us. Once the counter is done waiting it will go to the next state, otherwise our design will still stay in the same state until the counter is done counting. The final state is **state_wait_data** where we check if it is valid to send, when the value is **true** (1) we set the ack to 1 to tell our program that it can send the signal back to the master where the **ack** is **1** and we wait for our next data by going back the first state and the procedure repeats. If the data is **false (0)** we go back to the same state and try again. Given the sample, we complete the slave_control module by filling in the following values (**Shown below**).

```
            state_wait_rqst: begin
                next_state = (request == 1)? state_wait_to_send_ack:
state_wait_rqst;
                 next_notice = 1'b0;
                 next_ack = 1'b0;
```

```
                next_data = data;
                if(request) begin
                   next_start = 1'b1;
                end

            state_wait_to_send_ack: begin
                next_state = (done == 1)? state_wait_data :
state_wait_to_send_ack;
                next_notice = 1'b1;
                next_ack = 1'b0;
                next_data = data;
                next_start = 1'b1;
            End

            state_wait_data: begin
                next_state = (valid == 1)? state_wait_rqst :
state_wait_data ;
                next_notice = 1'b0;
                next_ack = 1'b1;
                next_data = data_in;
                next_start = 1'b0;
```

Similar to the **master_control** in each state we configure the next-in data such as the **ack**, **notice**, **data**, and **start**. To follow our model design for Handshake protocol using FPGA-to-FPGA communication protocol the master will send a request first by us when we press the request button, then the slave will give the signal of ack, and then the master can send numbers to the slave FPGA to show the numbers on the 7-segment display on the FPGA.

B. Block Diagrams and State Diagrams

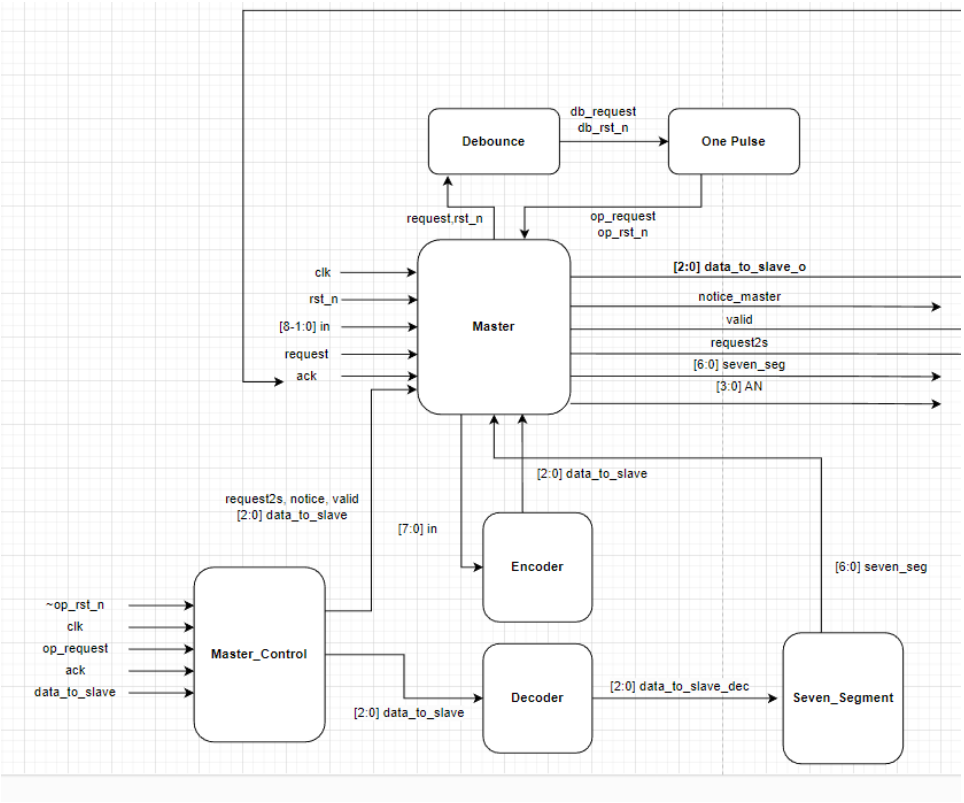


Figure 1.1 Block Diagram of Master (Chip2Chip)

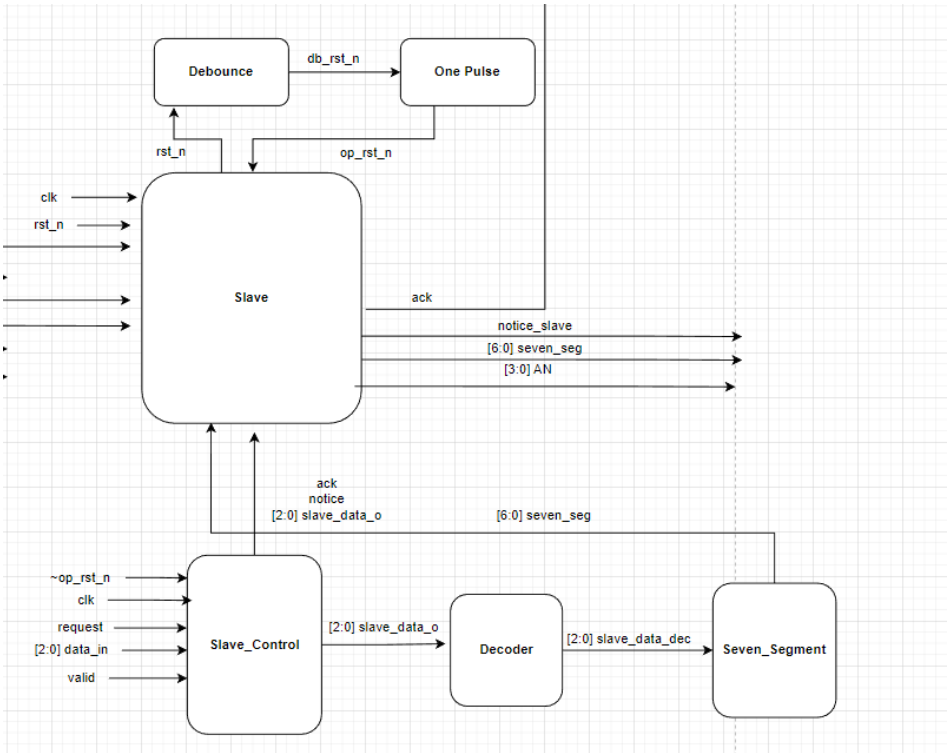


Figure 1.2 Block Diagram of Slave (Chip2Chip)

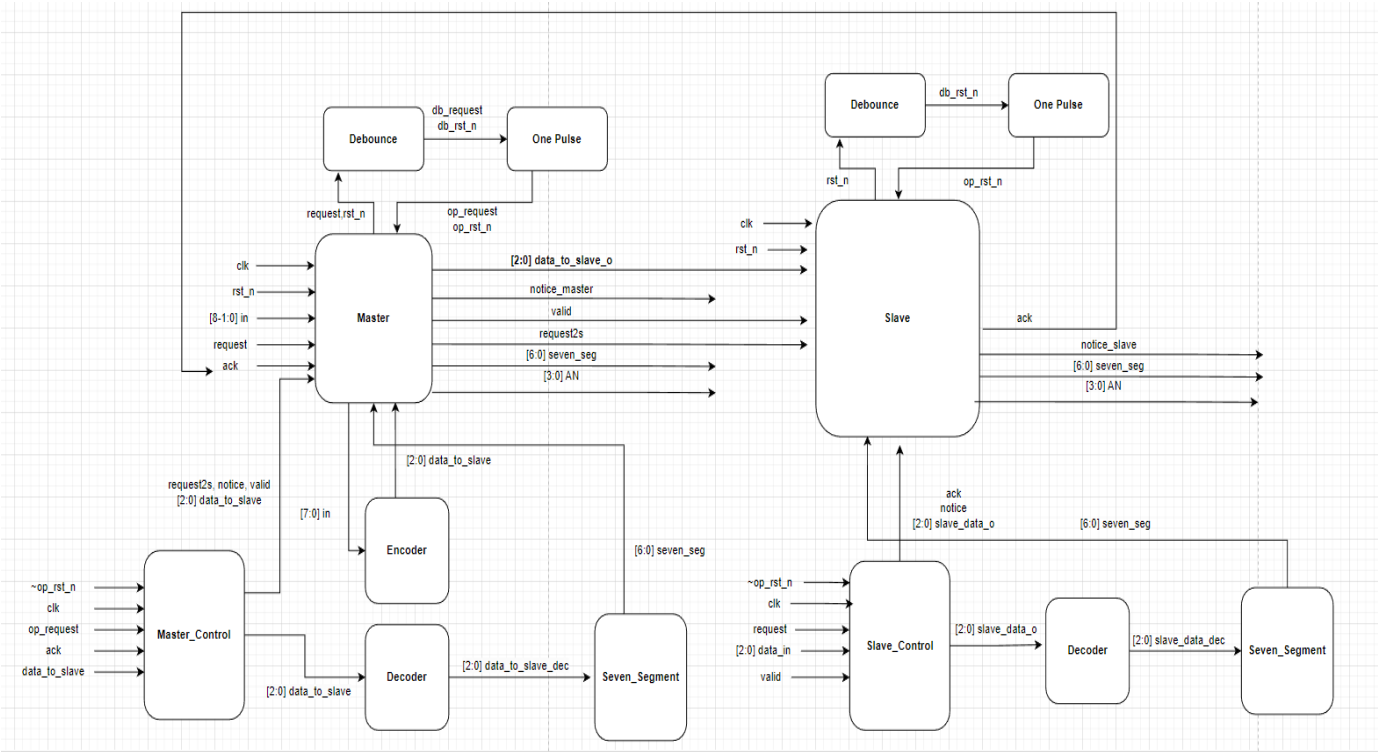
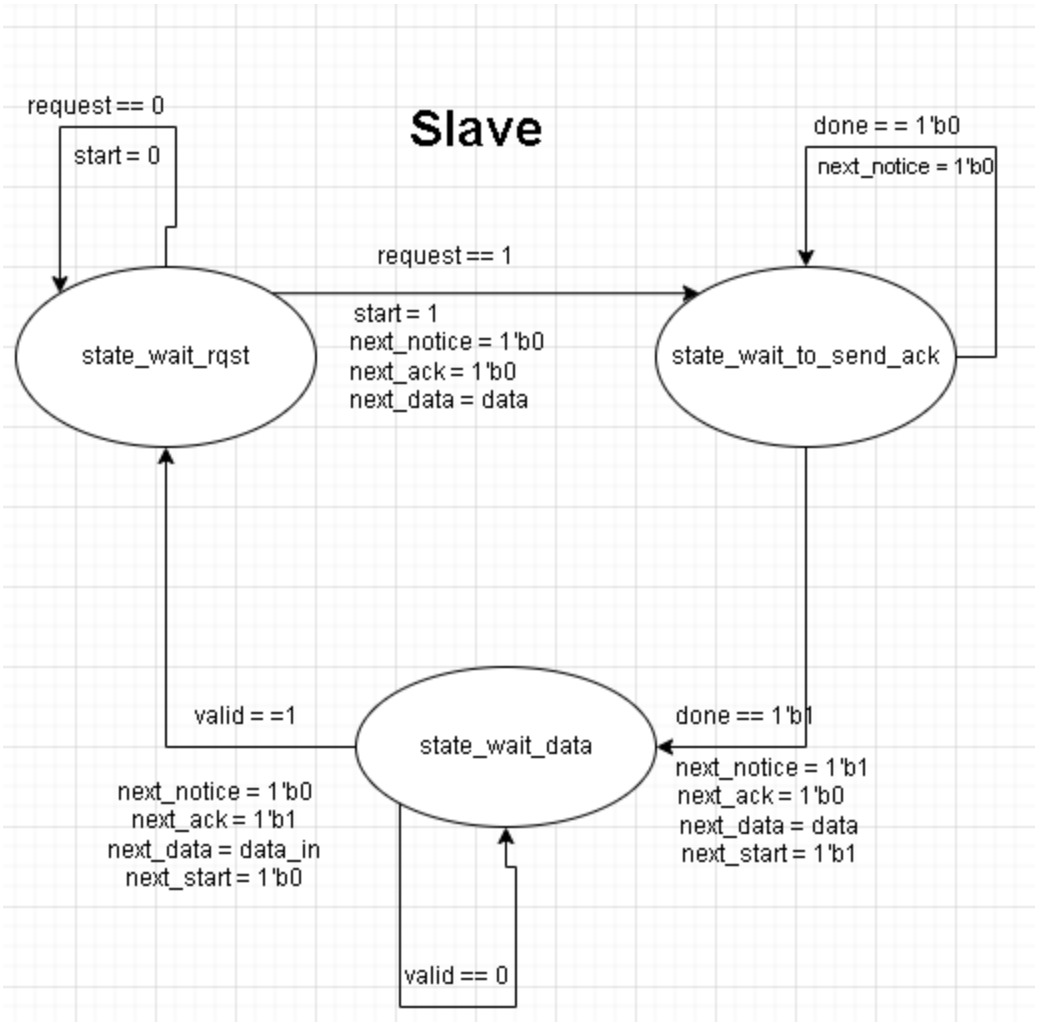
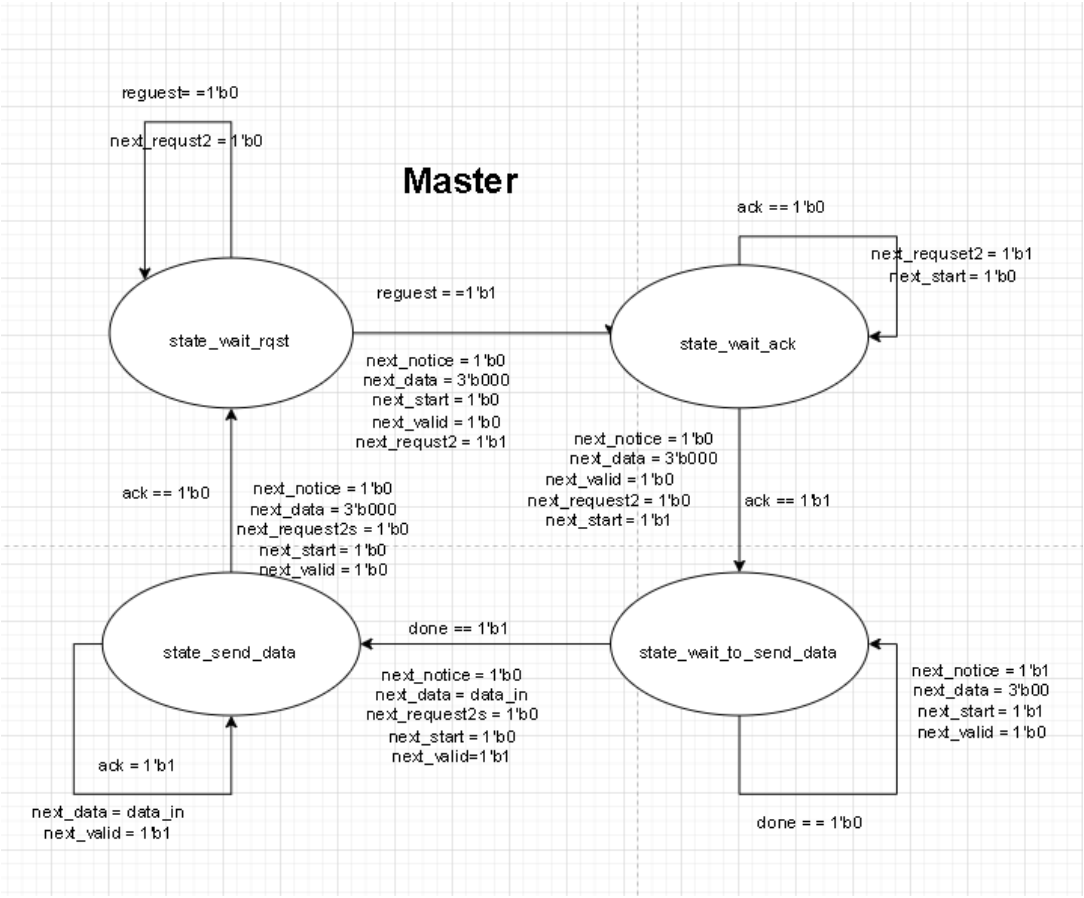


Figure 1.3 Combined Block Diagram of Master and Slave (Chip2Chip)



(Figure 1.4 State Diagram Chip2Chip Slave)

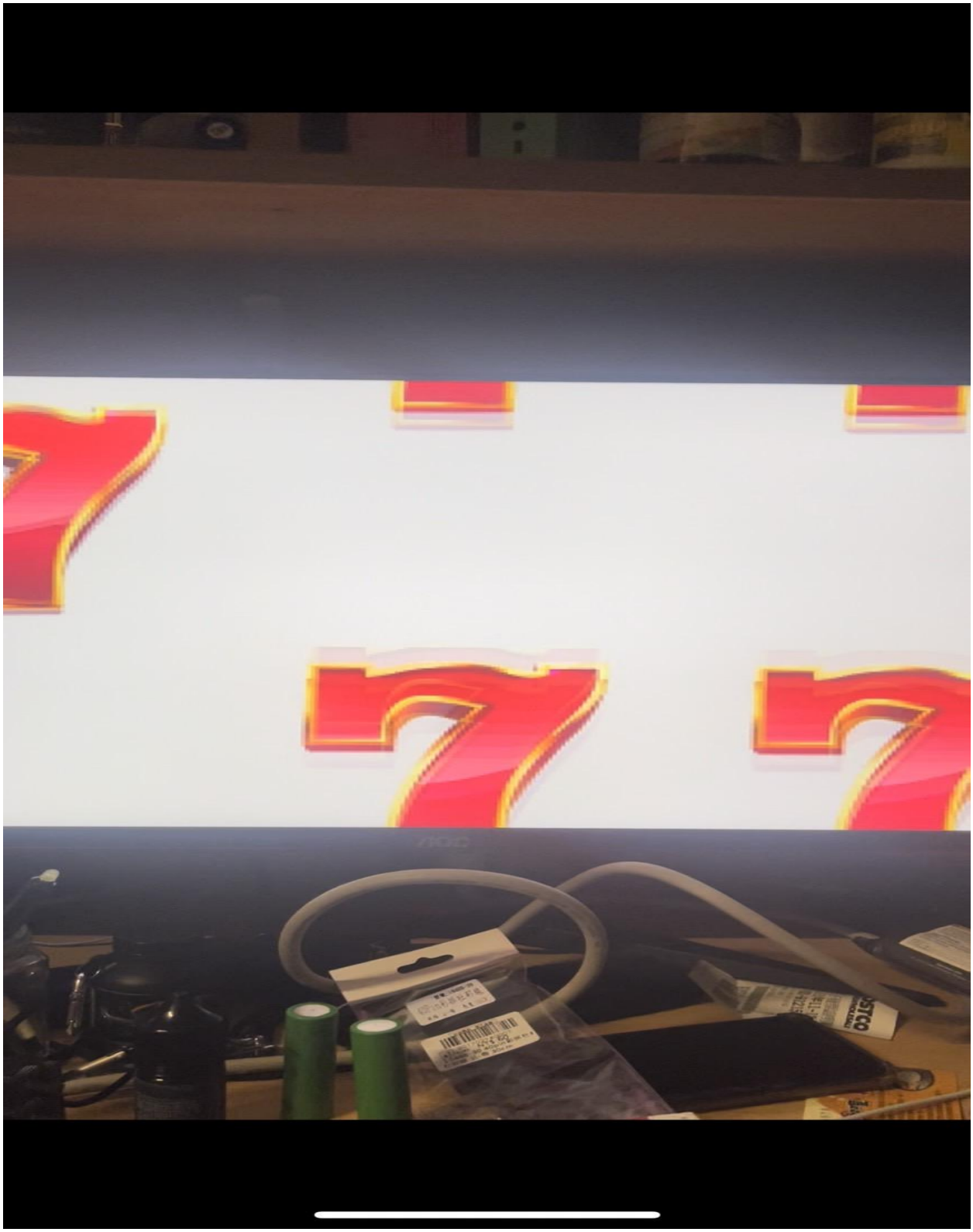


(Figure 1.5 State Diagram Chip2chip Master)

C. Test

To test our implementation of the chip-to-chip problem, we first double-check our code to our block diagram and state diagram to ensure that we did not leave anything behind or gave out the wrong values to the wrong modules. After thoroughly checking our code with our diagrams, we configure our FPGA for the master and the slave. Then we connect the PMOD of the two FPGAs to each other following the slides. After we successfully connect our PMODs and configure the I/O ports in the **Vivado** we run our code. We check with every number and test with all the different buttons. After making sure of the final product, we conclude that we successfully finished this advanced question.

2. **The Slot Machine**



3.

A. Design

For this advanced question, we first look at our objectives which are to create and implement two buttons such that one will run upward in direction and the other will run downward, depending on the button in the slot machine. Then we downloaded our sample code and studied it for a bit. We got a pretty good picture of how to implement it. Before we started coding, we drew our block diagram and state diagram to have a better understanding **(Figure 2.1, Figure 2.2)**. We realize that the **Top** module will control our buttons with the clock and reset whereas from the Top module the clock will be input for the **state_control** and **vga_controller** modules each. Inside the state_control module, we will calculate the speed and algorithm for the slot machine whereas it will give out inputs for the variables **A, B**, and **C** to the memory adder. Only after generating the memory in the memory block will it give the needed pixels to the top module. In addition to the **Top** module giving a clock to **state_control** and **VGA control** modules, it will also configure the buttons with **debounce** and o**ne pulse modules**.

Firstly, we add our two buttons (start_down, start_up) as inputs in the "**Top**" module as variables "**bLeft**" and "**bRight**" (button left, button right). We thought it would be easier to name it as the left button and right button than the start-up and start-down. After we declare our new variables we add new buttons to the **FPGA** by using the "**debounce**" and "one **pulse**" modules (**Shown below**).

```
debounce DB0(.s(bLeft), .s_db(bLeft_db), .clk(clk));
debounce DB1(.s(bRight), .s_db(bRight_db), .clk(clk));
debounce DB2(.s(rst), .s_db(rst_db), .clk(clk));
onepulse OP0(.s(bLeft_db), .s_op(bLeft_op), .clk(clk_d22));
onepulse OP1(.s(bRight_db), .s_op(bRight_op), .clk(clk_d22));
onepulse OP2(.s(rst_db), .s_op(rst_op), .clk(clk_d22));
```

After finishing the first few steps, we realized that only the "**state_control**" module needed to be implemented to complete the task. So, we add the inputs for the buttons and feed them as inputs ("**bLeft_op"**, "**bRight_op"**) to the module. Since we already have a reset button, we wire a variable "**start**" to be either the left button or the right button. Thus, we also add our states to the first "**always block**" where it will transition to the next state if reset is not triggered.

Finally, at the end of the module, we add our calculation which consists of three always blocks. We thought that the initial always block for the counter is necessary because without

constraint the counter would be difficult to control. Instead of assigning the next counter right away, we set an "if" statement where if the present counter is more or equal to "**10'd1000**" the next counter is **0**. Otherwise, it will follow the previous logic of the sample code (**Our implementation is shown below)**.

```
        if(counter >= 10'd1000) begin
            next_counter = 1'b0;
        end else begin
            next_counter = ((start==1'b0 && counter==10'd0) || (counter >=
10'd1000))? counter : counter + 1'b1;
        end
```

The next step was to add our logic for the Upward direction whereas we do not need to calculate the Downward direction because, in the given sample code, it was already assigned to the Downwards direction. Our job in this always block was to calculate the next **A**, **B**, and **C** for the slot machine. From carefully examining the Downwards direction logic, we quickly implemented our Upwards logic (**Shown Below**).

```
        UP :  begin
            //next_A_v_count = (A_v_count + A_state >= 10'd240)? A_v_count
+ A_state - 10'd240: A_v_count + A_state;
            //if A_V_count < A_state, next_A_v --> A_v_count - A_state +
10'd240
            // else A_V_Count - A_State;
                 if(A_v_count < A_state) begin
                      next_A_v_count = A_v_count - A_state + 10'd240;
                 end
                 else begin
                      next_A_v_count = A_v_count - A_state;
                 end
            //next_B_v_count = (B_v_count + B_state >= 10'd240)? B_v_count
+ B_state - 10'd240: B_v_count + B_state;
            //if B_V_count < B_state, next_B_v --> B_v_count - B_state +
10'd240
            // else B_V_Count - B_State;
                 if(B_v_count < B_state) begin
                      next_B_v_count = B_v_count - B_state + 10'd240;
```

```
                end
                else begin
                     next_B_v_count = B_v_count - B_state;
                end
                //next_C_v_count = (C_v_count + C_state >= 10'd240)?
C_v_count + C_state - 10'd240: C_v_count + C_state;
           //if C_V_count < C_state, next_C_v --> C_v_count - C_state +
10'd240
           // else C_V_Count - C_State;
                if(C_v_count < C_state) begin
                   next_C_v_count = C_v_count - C_state + 10'd240;
                end
                else begin
                next_C_v_count = C_v_count - C_state;
                end
       end
```

Finally our last always block objective was to activate the switching of buttons and states. We implemented a logic that if the **start** button is pressed the **start** variable will be either button left or the button right concerning the counter (**Shown Below**).

```
        if (start==1'b1 && counter==10'd0)begin
            next_state = bRight;
        end else begin
            next_state = state;
```

(Note: we switch some wires to registers in the state_control module. E.g.: **next_counter**, **next_A_v_count**, and so on.)

B. Block Diagrams and State Diagrams

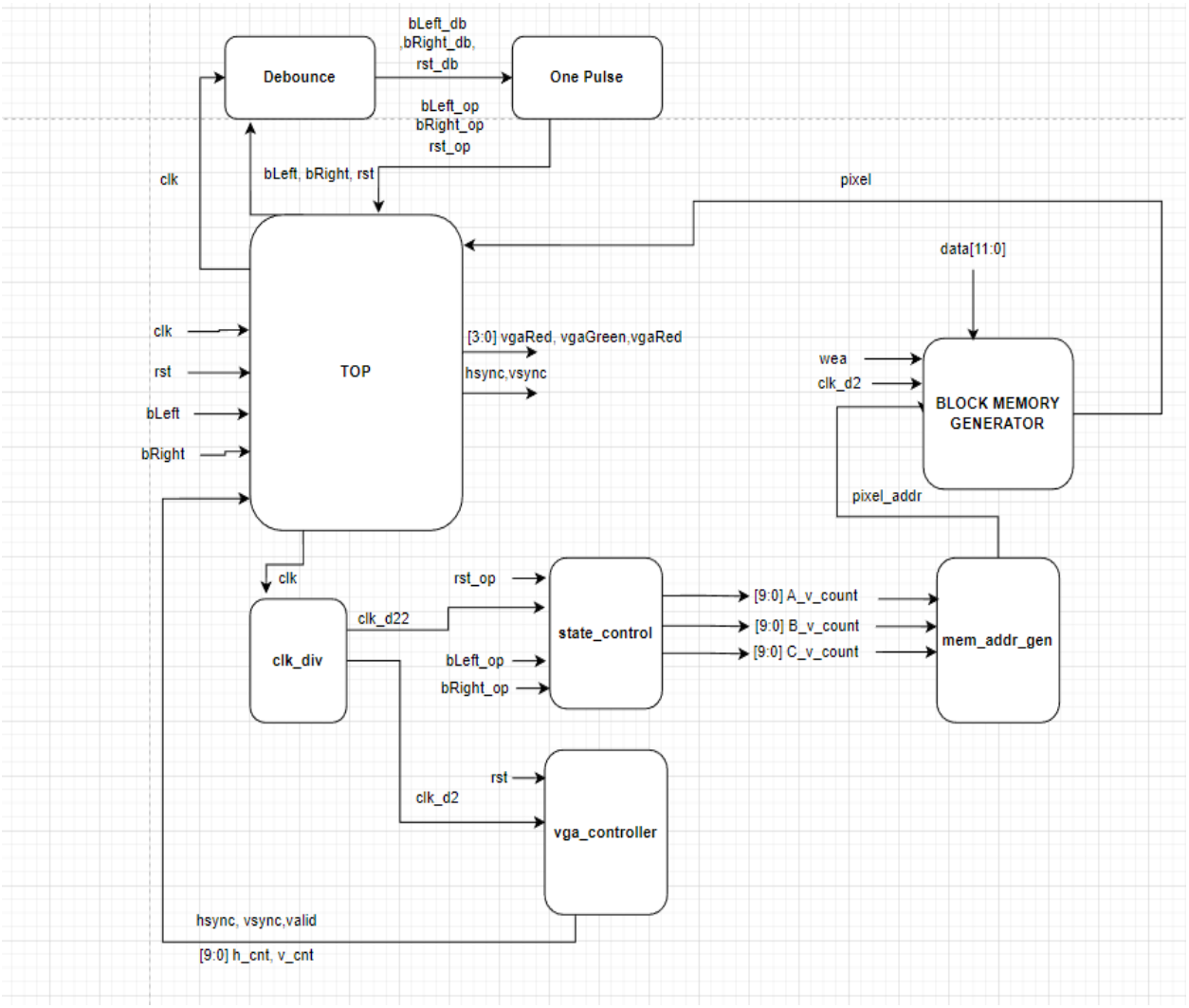


Figure 2.1 Block Diagram of Slot Machine

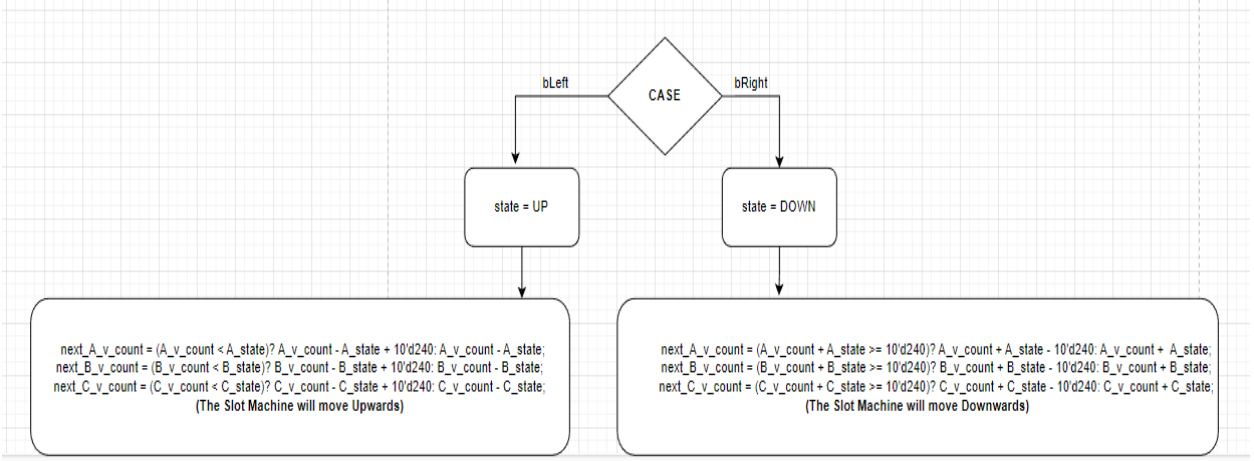
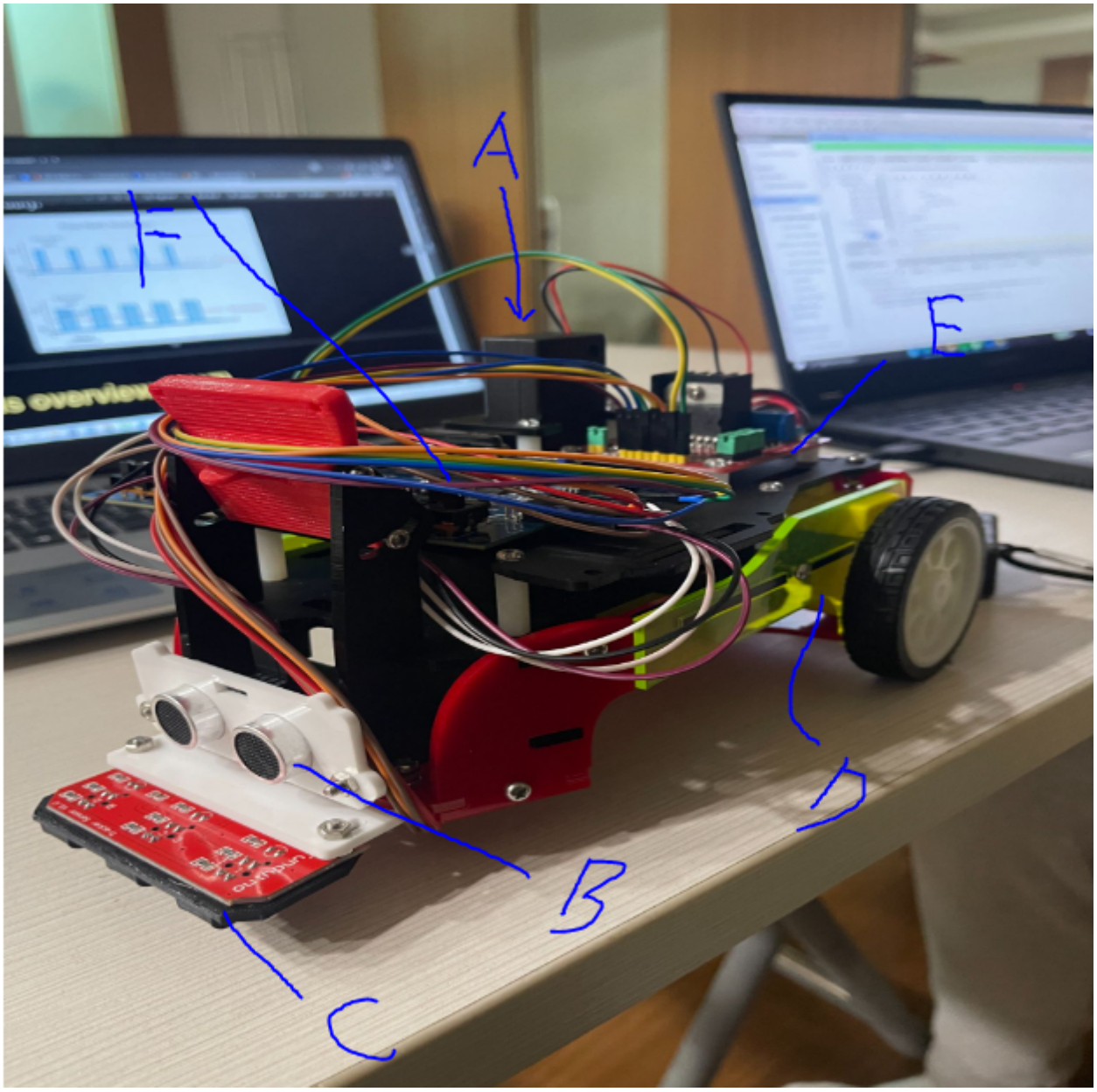


Figure 2.2 State Diagram of Slot Machine

C. Test

To test our design, we used the method from the basic lab of this advanced lab. We used our FPGA board and HDMI cables to make sure it is running correctly. Then we tested if the code still works and we pressed the buttons to test our logic. Finally, we double-checked it with our State Diagrams and saw that it was running smoothly like the demo that was shown to us by the TAs in class.

3. The Car



Here is a Picture of our car, and we labeled it as so

A = Battery Pack

B = Ultrasonic Sensor

C = Infrared Sensor / Tracker Sensor

D = Gear Motor / Dynamo

E = L298N Motor Driver Module

F = FPGA

A. Design

To begin designing the car, the first saw the demonstration vidoes from online. Thus, read the slides for the components that we are using for the car project to have better understanding of what we are doing in this advanced question. After carefully reading and examining our project, we downloaded the sample code file for the car. When we downloaded our code we saw that our car module will consist of 3 major modules for the 3 components, top module to put everything together, and few modules to calculate some variables that are needed for the modules.

Before coding our algorithm, we first designed our block and state diagram (**Figure 3.1** and **Figure 3.2**). Given the modules in the sample code, we completed our block diagram. The top module will first have the inputs of clk, rst_n, echo, left_signal, right_signal, mid_signal, and the values from the **debounce** and **onepulse** modules.Then it will pass the right values to the **motor**, **tracker_sensor**, and **sonic_top** modules (the algorithm for them will be explained below). The 3-way tracker will be given the values of **right**, **left**, and **mid** signals with rst_n and clk values where it will track the line to follow. The sonic_top module will be given the **clk**, **rst_n**, **trig**, and **stop** to sense if there is an obstacle in front using the sensor. Finally, the motor will have the **clk, rst, mode** , and **speed** to calculate the right speed for the motors.

In the sonic_top module, we were to calculate the right distance to trigger stoping the car. We assigned the **stop** variable to output to distance where the distance will give out **1'b1** or **1'b0** if there is an obstacle in the near 40 cm. The module that calculates the distance by the module **PosCounter** which was given to us.

```
assign stop = (dis <= 20'd4000 )?(1'b1):(1'b0);
```

Next, module to implement was the **tracker_sensor**. We created an case statement with three signals we receive from the top module. Where starting with the left to mid to right, if the values are 001 it would mean turning right and if 100 be turning left and so on. We parameter set the values for **RIGHT**, **LEFT**, **FRONT**, and **STOP**.

```
            case({left_signal,mid_signal,right_signal})
                3'b001: state <= RIGHT;
                 // so on
                3'b111: state <= STOP;
```

```
        endcase
```

The final module to complete before configuring everything in the top module is the motor module. The purpose for this module was to take right speed for different situations. We thought it would be easier to use another case statement for the motor mode where we set the parameter values to 10'd600 when fast and 10'd750 when it is fast. The mode will have **FRONT**, **LEFT**, **RIGHT**, and **STOP** modes where it will decide the speed for the next left and right motor (shown below). The module to calculate the speed is **motor_pwn** and **PWR_GEN**.

```
    case(mode)
        FRONT:begin
            next_left_motor = FAST;
            next_right_motor= FAST;
        end
        LEFT:begin
            next_left_motor = SLOW;
            next_right_motor= FAST;
        ///
        end
```

Finally, next module to put together everything together is top module. Our idea was to manipulate the values for the left and right engines. First, we check if the stop value from our sonic_top module. If the stop is **1'b1** we stop the car if not we continue our calculation with the left and right motors using the 3 major components that we configures in the previous modules above. The speed variable that we declared in the beginning which is two bit where it stores the values of the left and right motors. Then after calculation we assign the values back to the left and right motor. Then we use case statement from the state we got from the **tracker_sensor** to configure the right 1's and 0's for our two motors. The example of our case statement is shown below.

Although our car can move forward and detect objects to stop, sadly it was not able to turn left or right where it kept going forward on our demo day.

```
FRONT :  begin
    left[0] = 1'b0;
    left[1] = 1'b1;
    right[0] = 1'b0;
    right[1] = 1'b1;

end
LEFT: begin
    left[0] = 1'b1;
    left[1] = 1'b0;
 //////
```

B. Block Diagrams and State Diagrams

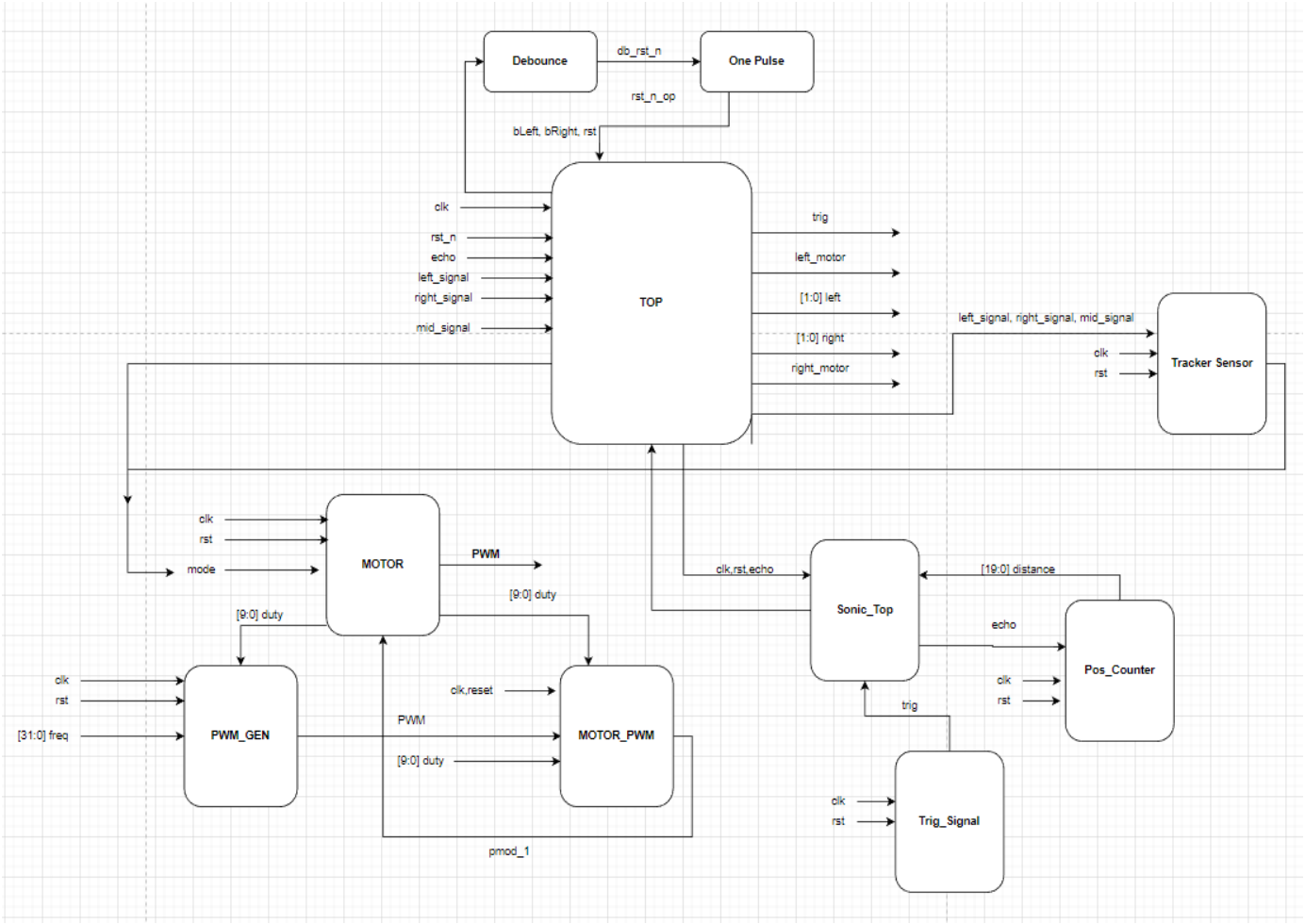


Figure 3.1 Block Diagram of Car

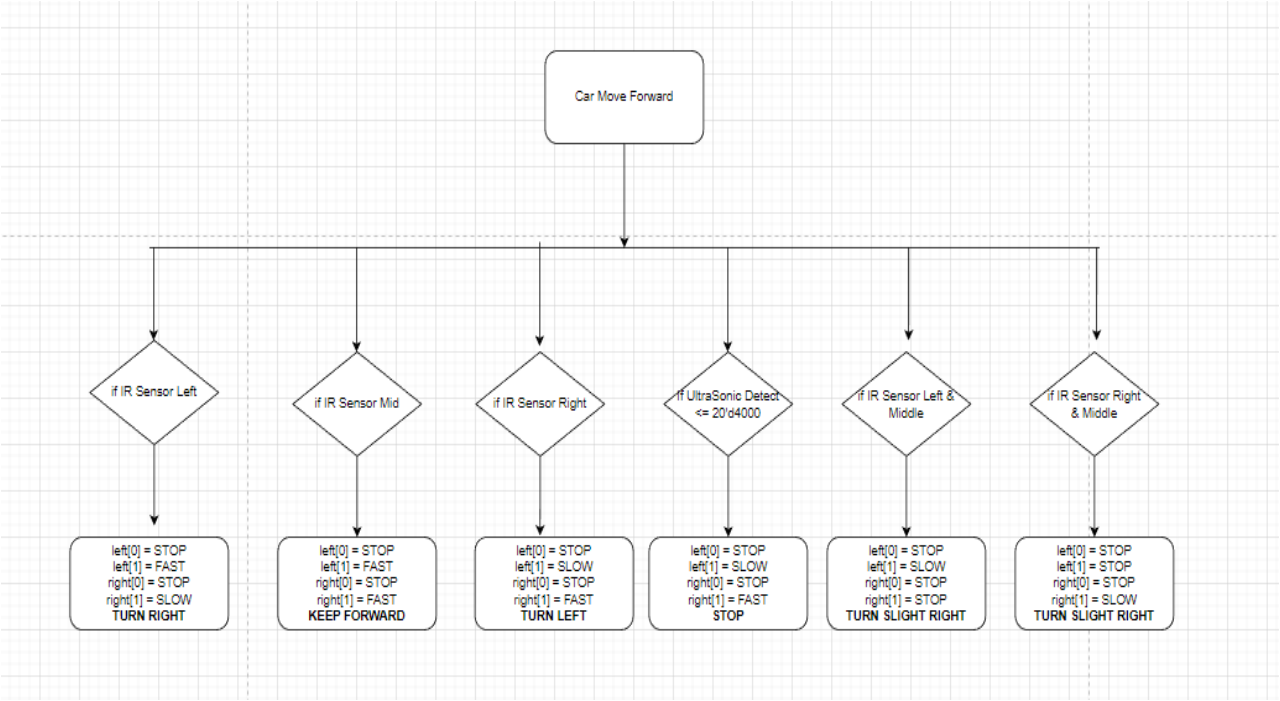


Figure 3.2 State Diagram of Car

C. Modules

FPGA

We are mounting the FPGA onto the car, but we weren't able to screw it onto the Dashboard of the car because the length of the white plastic screw was too short For the nut be able to hold its place into, but we had a really fine cable management so we were able to hold the fpga in place. And the FPGA should be configurated to Flash Memory, so it can save the program that is programmed previously via Vivado, so when running the car we just need a power source in able to run the FPGA, the power source itself can be a powerbank.

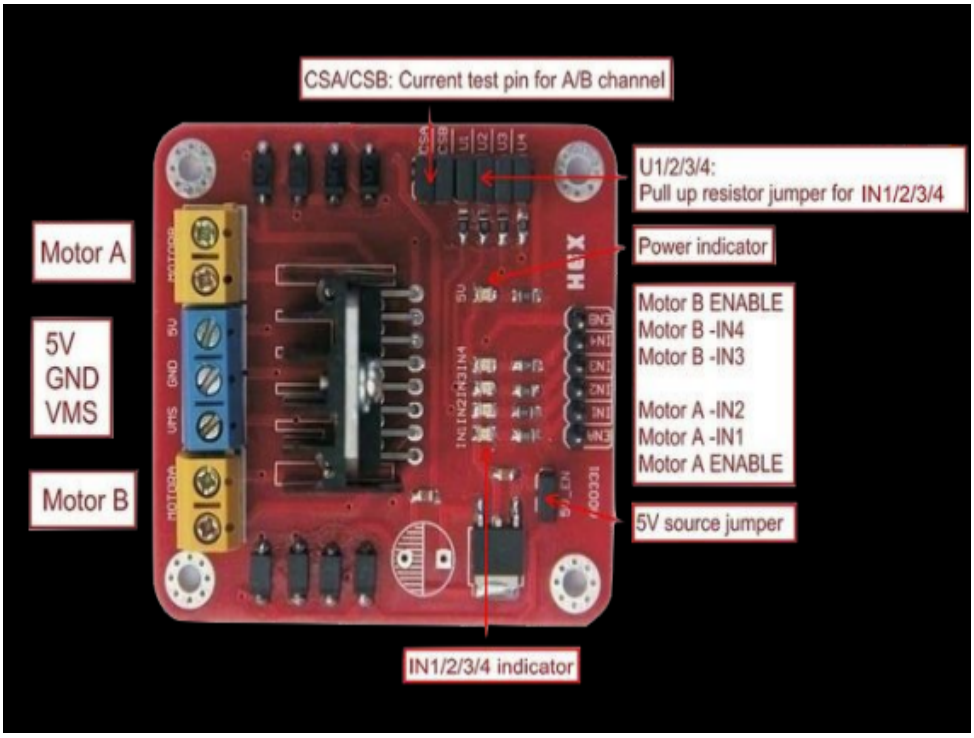
Gear Motor + L298N Motor Driver Module

The gear motor is self explanatory, it is used to move the wheels of the car, like a dynamo, the Motor Driver is the important part, where it has the pins :

- Motor_B Enable
- Motor B - IN4
- Motor B -IN3
- Motor A -IN2

- Motor A -IN1
- Motor A - ENABLE

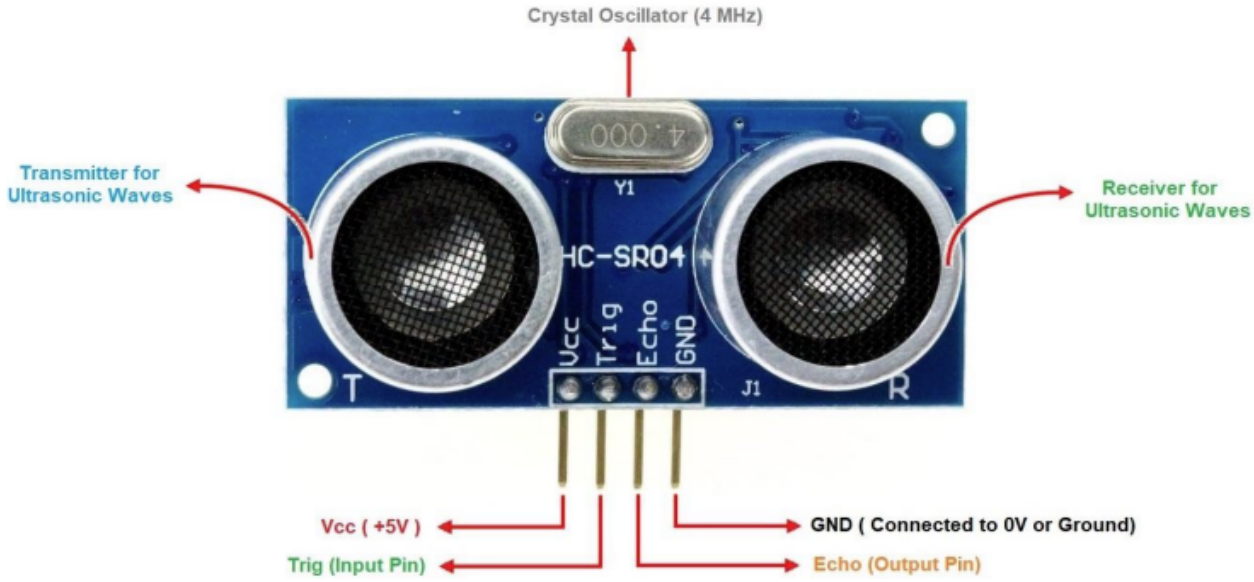
Then the Ground and 5V Source jumper, is connected to the FPGA GND and PWR Pin to the one below the On/Off Switch of the FPGA,
The Motor Driver itself controls the movement of the Gear Motor, where it can Control which Gear Motor to move, and how fast the gear motor spins.



Ref : Picture from Lecture Slides (Car Tutorial)

Ultrasonic Sensor : HC-SR04

The ultrasonic sensor itself is to detect the distance between the front of the car which is the ultrasonic sensor and an object in front of it, our goal is to stop the car when the ultrasonic's echo is below 40cm, so it won't crash the object infront.



Ref : Picture from Lecture Slides (Car Tutorial)

The Pins for the Ultrasonic Sensor are :

- VCC
- Trig
- GND
- Echo

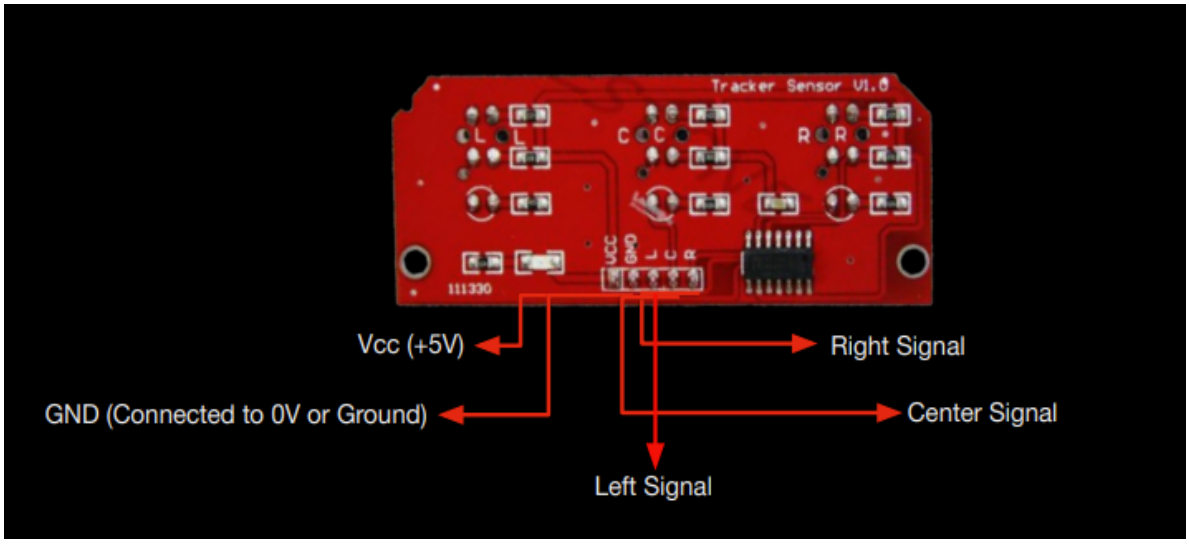
Which the VCC and GND are connected to the GND and PWR pins of the FPGA side pins, and the trig and echo are connected to the non-GND and PWR pins

3-Way Line Tracking IR Sensor : TCRT5000

The IR Sensor itself is used to detect the path of the car, since the track is a large Black line so we just have to detect the white line, and turn on and off the gear Motors accordingly,

The pins for the IR Sensor are :

- VCC
- GND
- Left Signal
- Right Signal
- Center Signal



Ref : Picture from the Lecture Slides (Car Tutorial)

D. Test

Before assembling or running the car, we make sure that our wire is wired correctly according to the slides. Then we check if our logic is following our logical Block Diagram and State Diagram. After we are sure that we did not do anything wrong, we adjust our pins to our model. Finally, run our car on a custom-made course which is

similar to the tracks shown on the slides. Our first objective was to run the car and then test turning left and right with the right speed. To conclude, we run numerous test drives with our car. Within the bugs we had with our car detecting or turning, we fixed on the spot whereas each bug took from two to three hours. However, when we tried to test our car, the car did not turn and kept running forwards. Although the car can detect if there is an obstacle in front of it, the major flaw was that our car would not turn right or left.

4. **Contributions**

Jason did advanced questions 1 and 2 while Tuguldur was stuck on debugging the car. Once Jason finished his advanced question, he also helped Tuguldur debug his car modules. For the report, Tuguldur wrote questions 1,2, and 3 while Jason drew the block diagrams and state diagrams, and helped Tuguldur explain 1,2,3

5. **Problems We Faced**

Lab6 in our opinion is the most fun of all labs, we get to experiment with different kinds of stuff, spend sleepless nights, and make a stronger bond in lab partnership. Even though we failed the car, we knew that we have tried, and at least the car could move forward. We had to rush our coding and spend a long amount of hours on our advanced questions. Sometimes our code has Syntax Errors where our code won't run successfully or our implemented algorithm will show an error where it would not show the correct output.

6. **What We Have Learned From Lab 6**

We learned how to work with sensors and HDMI with our FPGA boards. Also, we learned how to code more efficiently in Verilog. From the sample code, got more comfortable with reading and understanding other people's code. Although we did not understand the logic at first, we learned to adapt to the code and implement it to our objective.