

## **LAB 3**

### **Sequential Circuits**

#### **Team 6**

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

**鐵特德 Tuguldur Tserenbaljir ( 109006271 )**

## **TABLE OF CONTENTS**

<b>1. 4-bit Ping Pong Counter</b>	<b>1</b>
<b>2. First-In First Out (FIFO) Queue</b>	<b>6</b>
<b>4. Round-Robin FIFO Arbiter</b>	<b>9</b>
<b>5. Parameterized Ping Pong Counter</b>	<b>13</b>
<b>6. FPGA Implementation</b>	<b>19</b>

## 1. 4-bit Ping Pong Counter

### a. Design

In this part, Jason was confused about the condition of a State Machine, because he tried to write a code where when the direction is 1 it counts up until 15, when it is 15, it will change the direction, but the waveform was like so.

When it is 15, the direction is 0

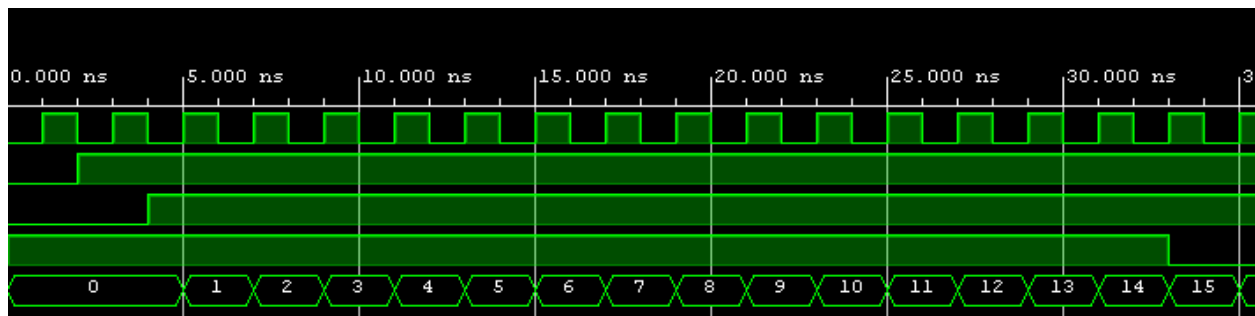
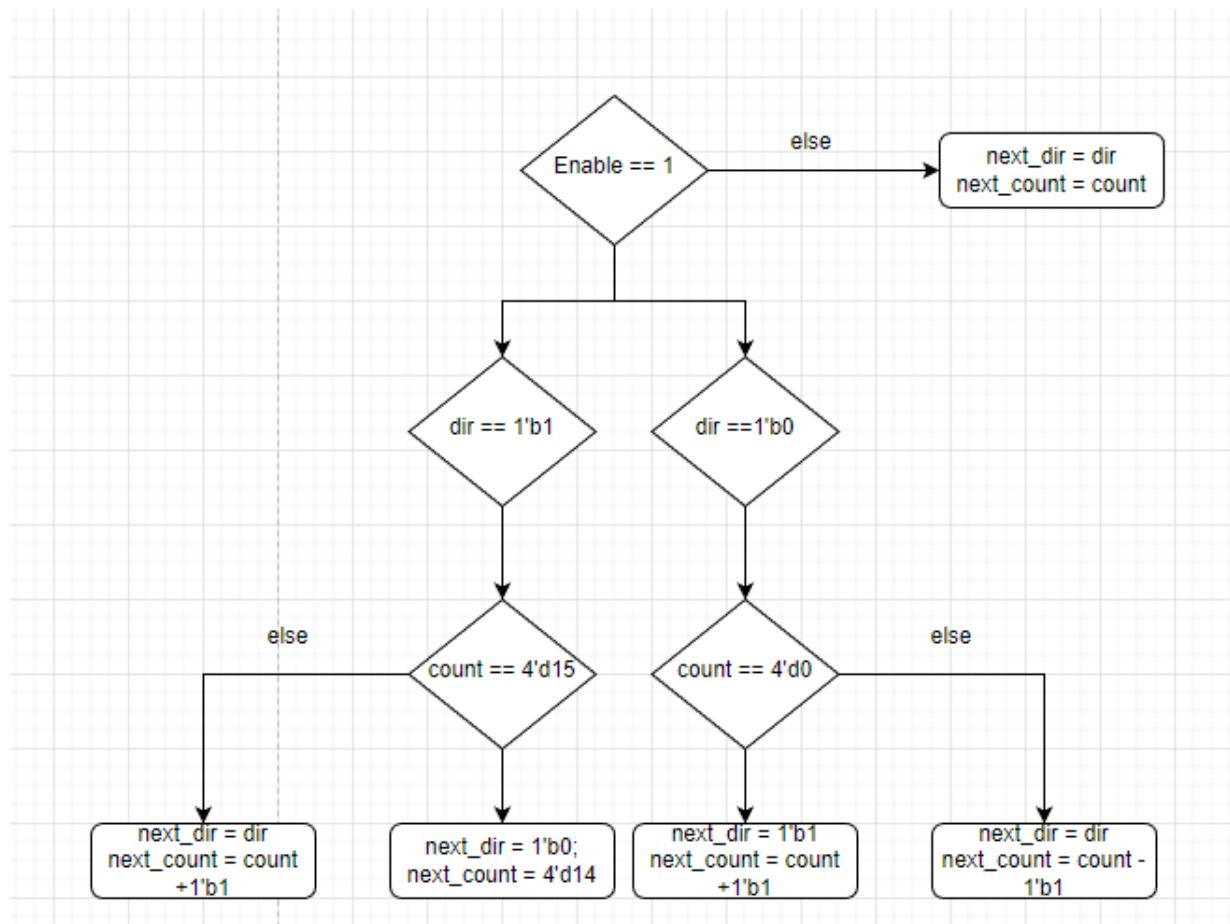
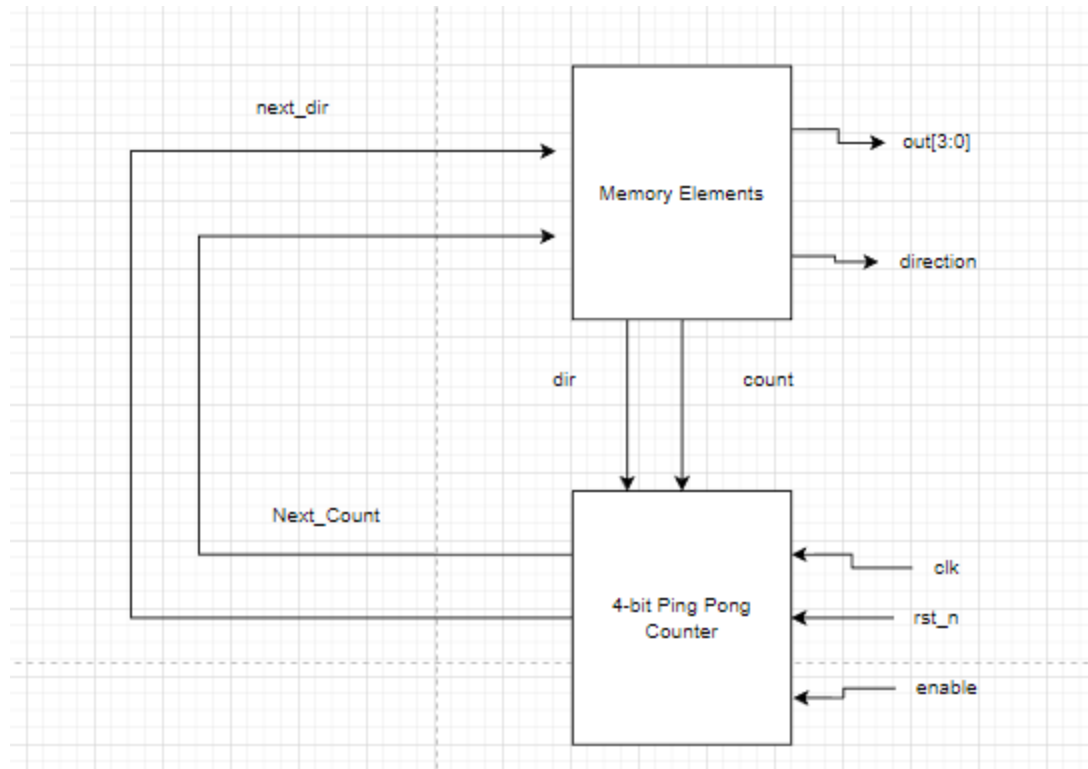


Figure 1.1

So he asked a senior about this and he explained about **Finite State Machine**, and when declaring “if” statement it declares the current state, not the next state, and the senior advised me to use 2 variables of the current, and next. So Jason designed it like so. Jason initialized the variables for the Sequential which was the latch to get values from the next\_count and next\_dir



**Figure 1.2** Flow Chart of Ping Pong Counter



**Figure 1.3 Circuit Diagram of Ping the Ping Pong Counter**

**b. Code**

Since it is a 4-bit Ping Pong Counter, the parameters of the counter would be 0-15. So , first we initialize the sequential circuit , if reset is 0 , initialize the count do 0 and direction to 1, if it's 1, set it to the next count. As well as the direction.

```
always @(posedge clk) begin
    if(rst_n == 1'b0) begin
        count <= 4'b0000;
        dir <= 1'b1;
    end
    else begin
        count <= next_count;
        dir <= next_dir;
    end
end
```

This is the core part of the code, where it is the combinational circuit, where all the **Logic Diagram** is placed here. When enable is set to 1, it will start the counter, if it's 0 it will hold the value of count and dir.

The counter when the direction is one will check if the count is 15, it will change the next direction to 0 and change the next count to 14, this is a problem i encountered previously if i just assign it in 1 circuit, not in sequential and combination circuit.

```
always @(*) begin
    if (enable == 1'b1) begin
        if (dir == 1'b1) begin
            if (count == 4'd15) begin
                next_dir = 1'b0;
                next_count = 4'd14;
            end
            else begin
                next_dir = dir;
                next_count = count + 1'b1;
            end
        end
        else begin
            if (count == 4'd0) begin
                next_dir = 1'b1;
                next_count = 4'd1;
            end else begin
                next_dir = dir;
                next_count = count - 1'b1;
            end
        end
    end
end
```

```

    end
end
else begin
    next_dir = dir;
    next_count = count;
end
end
end

```

```

// Output
assign direction = dir;
assign out = count;

```

### c. Testbench

```

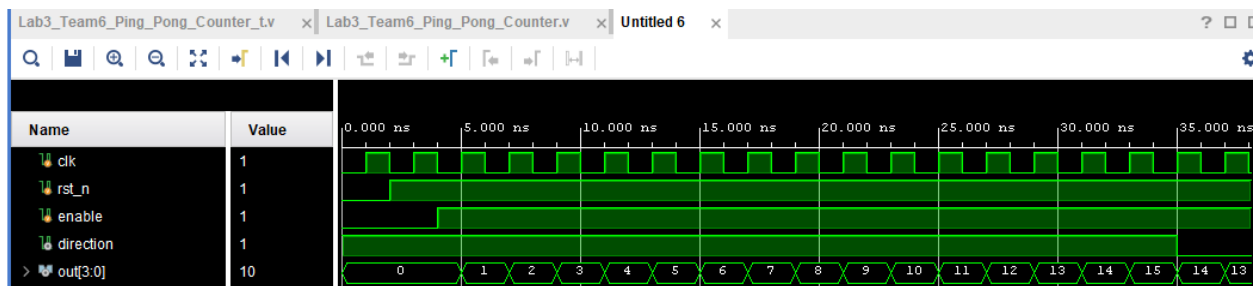
Initial begin
    @(negedge clk)rst_n = 1'b1;
    @(negedge clk)enable = 1'b1;
    #200 $finish;
end

endmodule

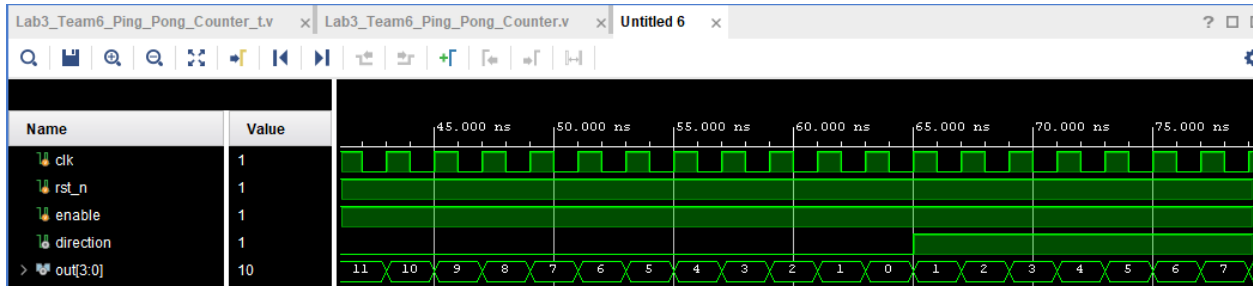
```

I just set the rst\_n , and enable to to 1 to test if if the module works.

### d. Waveform



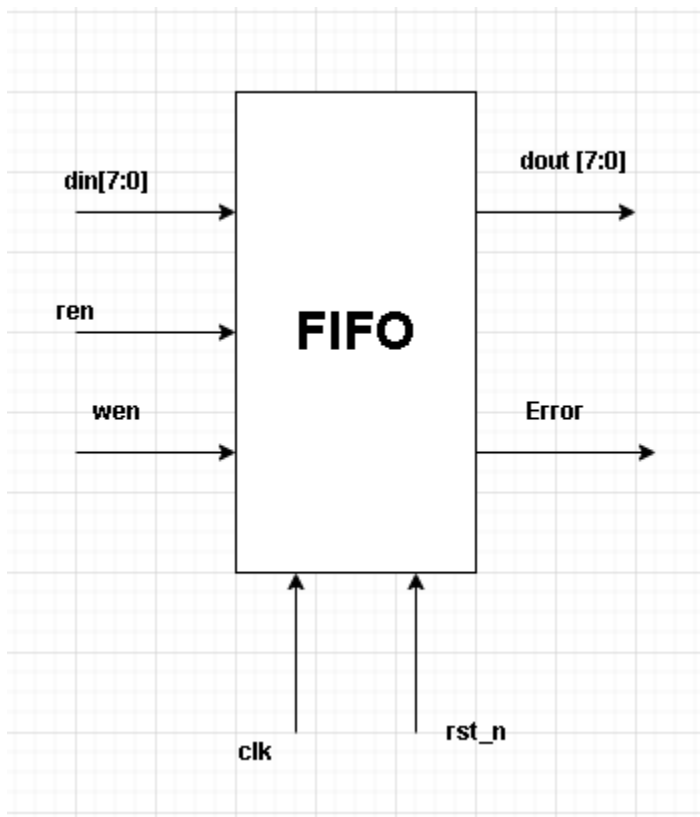
The case begins on 0 when enable is 1, then counts up until 15, after that **changes direction** to down again and starts counting down, similarly when it reaches 0, it will start counting up



## 2. First-In First Out (FIFO) Queue

### a. Design

For this design of the FIFO Queue, we design using behavioral modeling. Our block diagram will look like the following picture (**Figure 2.1**). Our design will only function when the clock is positive whereas if not it will set all of the values to 0.



(Figure 2.1)

First inside the “always” block, we check if the reset is false or not, and if it is false we set all of the storage, output, and error values to zero. Following the next condition, we check whether ren and wen are 1 or 0 with the counter being null or full. For example, if both ren and wen are 1 and the Counter is not empty, our code reads. In addition, we write to the input into the storage if the Counter is not full (Count == 8),

wen is 1 and ren is false. Finally, the final condition is that when the we only output the storage value if ren is equal to 1 and wen is equal to 0.

After checking the conditions for the assignment, we use nested “if-else” statements to get the correct counter number following the above operation whether it is a read or write operation.

In contrast, we check whether we are allowing an error. To check, we first assign variables Empty and Full to 1 and 0 using by checking if the Counter is at a max value of 8 or a min value of 0. For example, it would be like reading in empty storage or writing in full storage. Finally, we assign the output and error from our calculations.

## **b. Code**

```
assign Empty = (Count==0)? 1'b1:1'b0;
assign Full = (Count==8)? 1'b1:1'b0;

always @(posedge clk)begin
    if(rst_n == 1'b0)begin
        FIFO[0] = 1'b0;
        ...//
        FIFO[7] = 1'b0;

        read_Counter = 1'b0;
        write_Counter = 1'b0;

        D_dout <= 1'b0;
        D_error <= 1'b0;
    end

    else if( ren == 1'b1) begin
        if(Count != 1'b0 && wen == 1'b1)
            D_dout = FIFO[read_Counter];
            read_Counter = read_Counter + 1'b1;
        end

    else if ( wen == 1'b1 ) begin
        if(Count < 4'b1000 && ren == 1'b0)
            FIFO [write_Counter] <= din;
            write_Counter = write_Counter + 1'b1;
        end

    else if(ren == 1'b1 && wen == 1'b0) begin
        D_dout = FIFO[1];
```



```

end

if(read_Counter > write_Counter)begin
    Count = read_Counter - write_Counter;
end else if(write_Counter > read_Counter)begin
    Count = write_Counter - read_Counter;
end
if(Empty || Full)begin
    D_error = 1'b1;
end else begin
    D_error = 1'b0;
end
end
end
assign dout = D_dout;
assign error = D_error;
endmodule

```

### c. Test-bench

```

always #1 clk = ~clk;
initial begin
    @(negedge clk) rst_n = 1'b1;

    @(negedge clk) wen = 1'b1;
    repeat(2 ** 4) begin
        #2 din = din + 1'b1;
    end

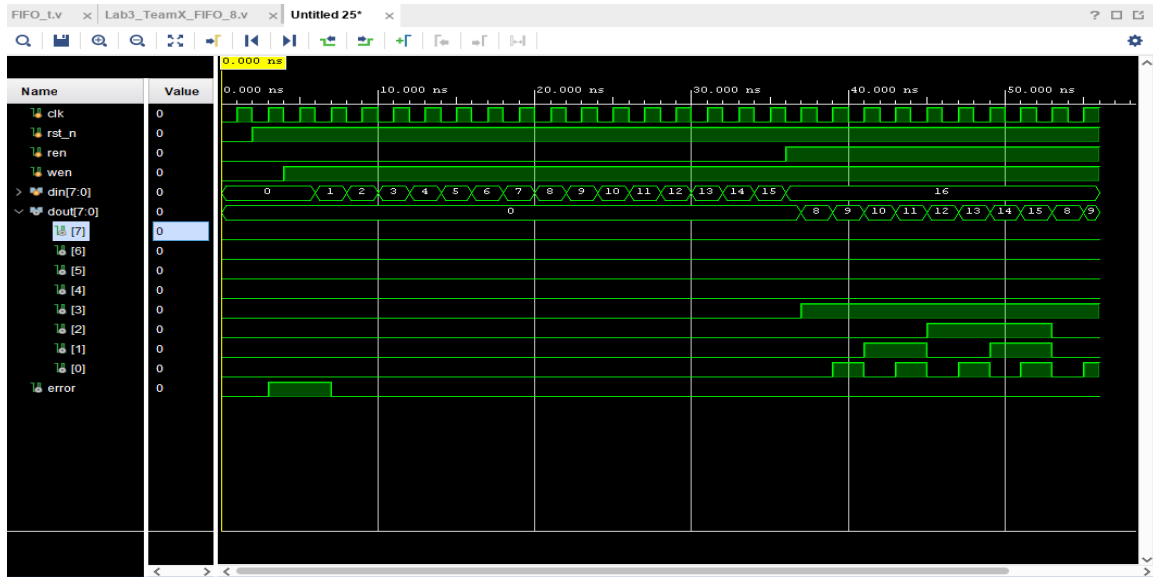
    @(negedge clk) ren = 1'b1;

    #20 $finish;
end

```

After coding the main algorithm for our FIFO verilog, we design a test-bench in the above so. We calculate until 2 to the power of 4 where the rst\_n, wen, and ren will all have a value of 1'b1 at first. In contrast, when the input increments, simultaneously other values will also increment and change. After running a successful simulation in Vivado, we get the waveform for our design. We test our design for each bit from the beginning and check from the examples given in the slides. At first, when we try to read, it raises an error. In addition, we can see from the waveform that it follows our algorithm that we created for FIFO Queue.

### d. Waveform



### 3. Multi-bank

#### a.Design

For the Multi-bank, we have four banks where in those 4 banks each have their own 4 banks. To design we have the main module Multi-bank, Bank-Memory, and finally the Memory Model.

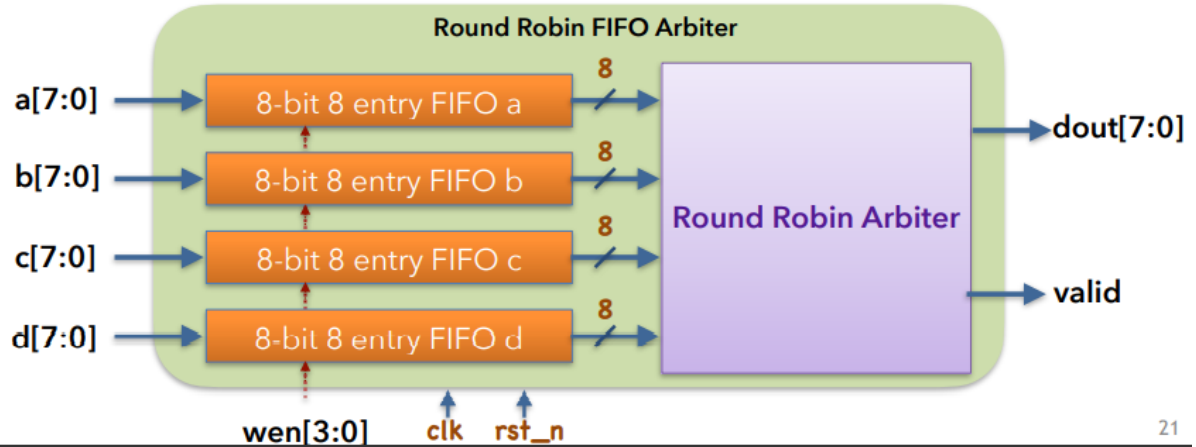
In the Multi-bank module, we initialize every bank's read and write corresponding to their waddr and raddr with their respective wen with the values of 1's and 0's. After which it will pass it onto the next module Bank-Memory where it will do the same functions in the last module but in a sense it Multi-bank is in a higher hierarchy. Then it passes on the final module Memory which we implemented on our Lab from last week.

**Tuguldur was not able to finish the assignment due to time.**

### 4. Round-Robin FIFO Arbiter

#### a. Design

For this design we follow the slides, where the model diagram is drawn below.



In our code, we have two “always” blocks. The first “always” block is set to the positive edge of the clock where it determines the initial values of the counter and ren whether it is valid or not for the output. The second “always” block calculates each 4 FIFO block with their respective 8-bit input (a,b,c,d). We see if ren[0] is true or not and then wen[0] or if it is an error for each 4 inputs. If the error is not valid, thus the output is equal to zero in the algorithm. Else, we set the value for the output to a wire and pass it on into the FIFO block that we implemented earlier for question two and it is valid.

## b. Code

```
always @(posedge clk) begin
    ren = counter;
    if(rst_n == 1'b0)begin
        counter <= 4'b0001;
        ren <= 4'b0000;
    end
    else begin
        if(counter == 4'b1000) counter <= 4'b0001;
        else counter <= counter << 1;
    end

    dout <= D_dout;
    valid <= D_valid;
end

always @(*)begin
    if(ren == 4'b0000)begin
        D_dout = 1'b0;
```

```

        D_valid = 1'b0;
    end
    else begin
        if(ren[0] == 1'b1 && (wen[0] == 1'b1 || error__a == 1'b1))begin
            D_dout = 1'b0;
            D_valid = 1'b0;
        end else begin
            D_dout = dout_a;
            D_valid = 1'b1;
        end
    end

    if(ren[0] == 1'b1 && (wen[0] == 1'b1 || error__b == 1'b1))begin
        D_dout = 1'b0;
        D_valid = 1'b0;
    end else begin
        D_dout = dout_b;
        D_valid = 1'b1;
    end
    ...///
end
end
end

FIFO_8 A(clk, rst_n, wen[0], ren[0], a, dout_a, error__a);
FIFO_8 B(clk, rst_n, wen[1], ren[1], b, dout_b, error__b);
FIFO_8 C(clk, rst_n, wen[2], ren[2], c, dout_c, error__c);
FIFO_8 D(clk, rst_n, wen[3], ren[3], d, dout_d, error__d);

endmodule

```

### c. Test-bench

```

always#(cyc/2)clk = !clk;

initial begin
    clk = 1'b0;
    rst_n = 1'b0;
    wen = 1'b0;
    a = 8'd87; b = 8'd56; c = 8'd9; d = 8'd13;

    @(posedge clk) begin
        rst_n = 1'b1;
        wen = 4'b1111;
        a = 8'd87; b = 8'd56; c = 8'd9; d = 8'd13;
    end

    @(posedge clk)begin

```

```

        wen = 4'b1000;
        a = 8'd0; b = 8'd0; c = 8'd0;
        d = 8'd85;
    end

    @(posedge clk)begin
        wen = 4'b0100;
        a = 8'd0; b = 8'd0; d = 8'd0;
        c = 8'd139;
    end

    @(posedge clk) wen = 4'b0000;

    #(cyc * 4)

    @(posedge clk)begin
        wen = 4'b0001;
        a = 8'd51;
        b = 8'd0; d = 8'd0;
        c = 8'd0;
    end

    @(posedge clk) wen = 4'b0000;

    wen = 1'b0;
    a = 8'd0; b = 8'd0; c = 8'd0; d = 8'd0;

    repeat (2**3) begin
        #(cyc*2)
        wen = wen + 1'b0;
        a = a + 1'b0;
        b = b + 1'b0;
        c = c + 1'b0;
        d = d + 1'b0;
    end

    #(cyc*4) $finish;
end

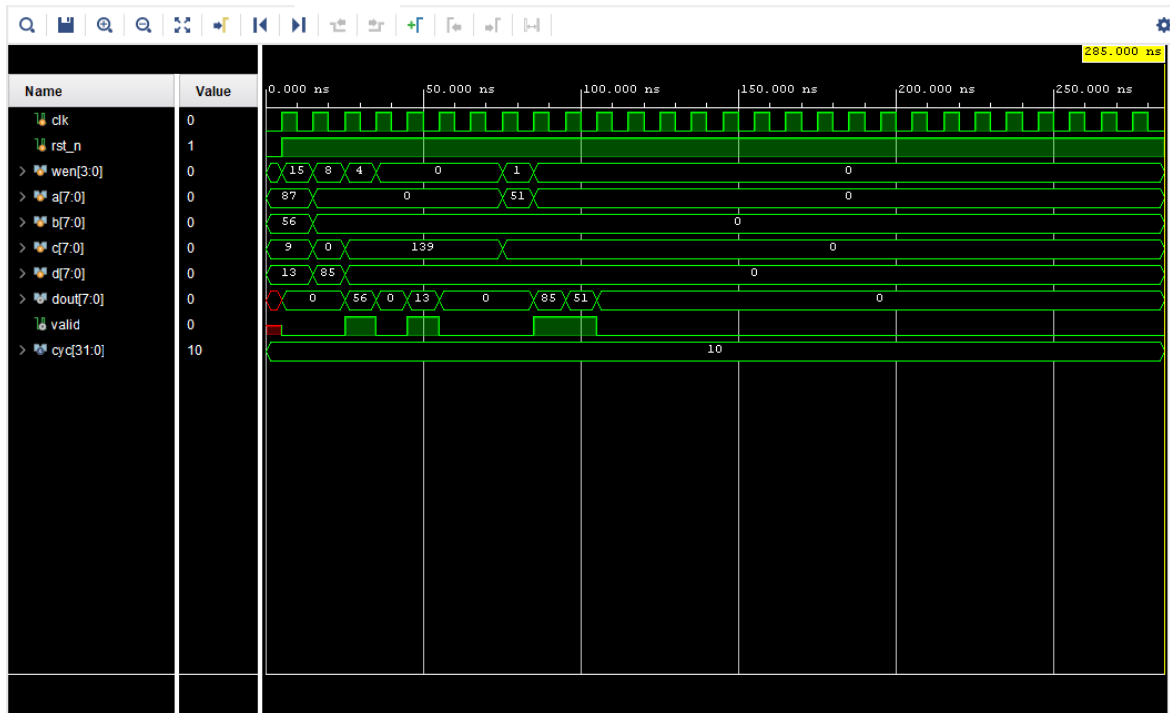
endmodule

```

For the test case, we manually set the values of the inputs(a,b,c,d) whereas the “0” represents the null from the slides. We prioritize the values of the wen of 1111, 1000,

0100, 0000, 0001, and 0000. If we check our waveform, we see that it is the same as the slide and we can continue to debug our verilog algorithm for the FIFO whereas the test case is set to the sample waveform.

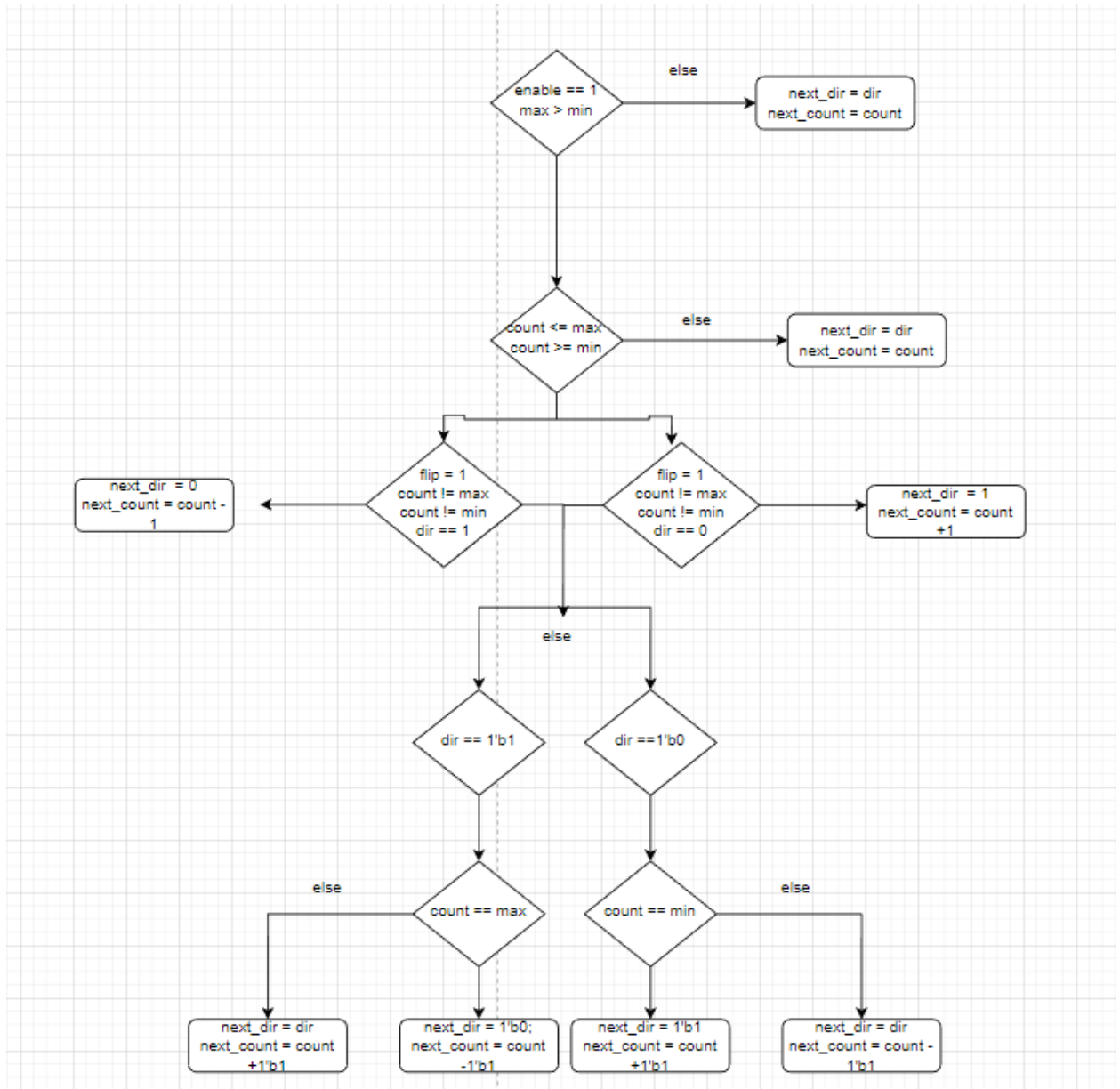
#### d. Waveform



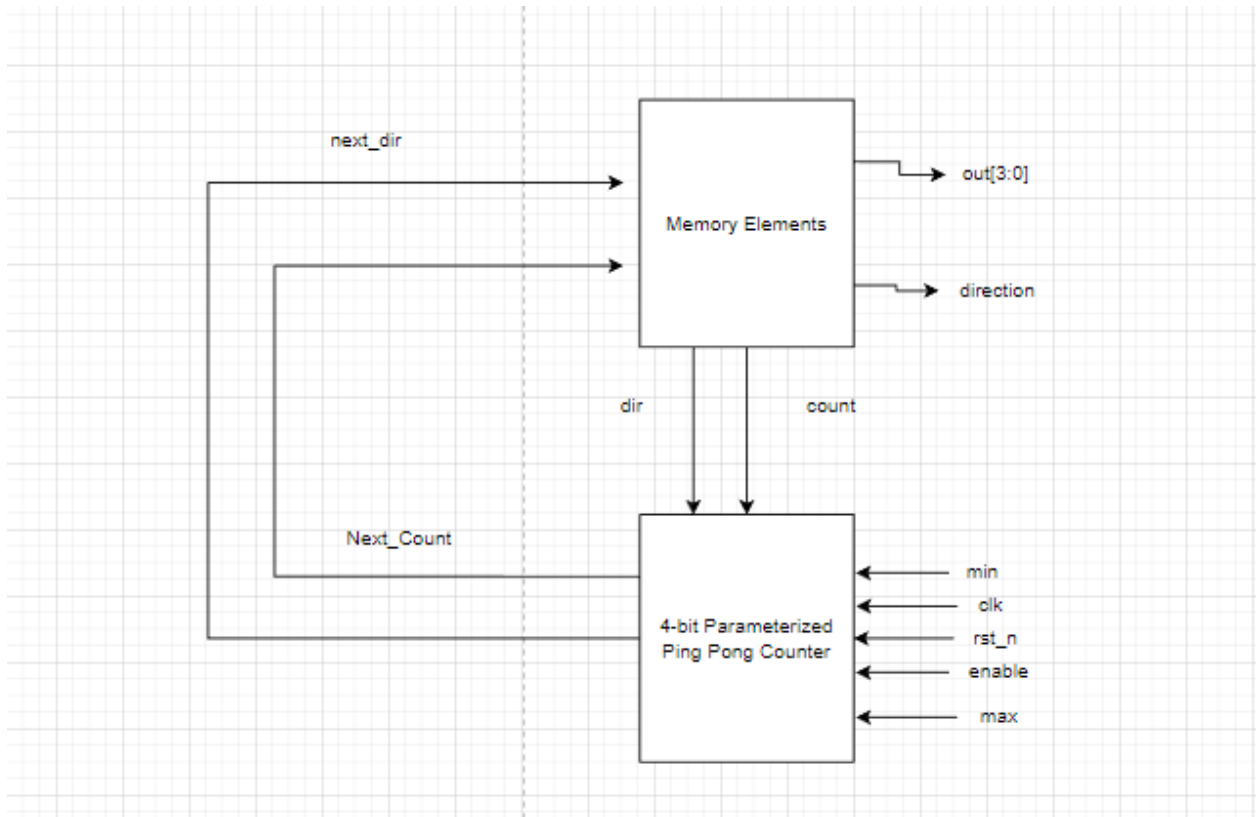
## 5. Parameterized Ping Pong Counter

### a. Design

The Parameterized Ping Pong Counter is similar to the Ping Pong Counter, but it has parameters of Min and Max on it. In this problem, Jason reused his Ping Pong Counter module and add parameters and additional if elses for the flip.



**Figure 5.1 Flowchart of Parameterized Ping Pong Counter**



**Figure 5.2 Circuit Diagram of Parameterized Ping Pong Counter**

#### **b. Code**

Like other circuits, we have to declare 1st the Sequential circuit where in this case, if the reset is 0, it will reset the count to min, and direction to 1. it will reinitialize to the min and also reinitialize the direction to 1. Otherwise, the count will hold its value, as well as the direction.

```
always @(posedge clk) begin
    if(rst_n == 1'b0) begin
        count <= min;
        dir <= 1'b1;
    end
    else begin
        count <= next_count;
        dir <= next_dir;
    end
end
```



```
end
```

This is the core of the code, where we give the conditions for the ping pong counter, so it will begin its operations if enable is 1 and the max is bigger than min. After checking that, we also check if the count is less/equal to max and count bigger or equal to min., if these 2 conditions does not pass, it will hold the counter and direction value which is colored in red After that the Ping Pong Counter may initiate its operations.

Then after this the Ping Pong Counter Starts, but 1st for Good Programming Style, we put the one with most parameters for if else, at the front. In this case, it is for the flip so i set the condition for flip = 1, count != max, count != min, and **direction** to 1 it will set the next direction to 0, which is down and will minus the next\_count so it wil initiate the count direction downwards. It is similar when **direction** is 0. But then, the counter will also flip, but not through flip = 1, if the counter has reached its max/min. So we put another condition if the direction is 1, and it has reached its max, it will set its direction to 0, and start to count downwards. Then **a good programming style** is about putting an else after if. Even though, if else is not recommended for hardware languages. I think in this counter, i am forced to use else if. So on the else statement, it is just the opposite of the if statement, which elaborates if the counter isn't max yet it will keep on the direction to up and the direction to 1.

Then there is an else for the if direction = 1. So it implies if the direction is 0. Just like the max, if it has reaches **min**, it will change the next direction to 1 and the next count to count += 1, which means its counting up. And if it has not reached the min, it will keep **counting down and the next\_dir to min**. And that concludes the code explanation.

```
always@(*)begin
    if(enable == 1'b1 && max > min)begin
        if(count <= max && count >= min)begin
            //DEBUG DONE
            if(flip == 1'b1 & count != max & count != min & dir == 1'b1) begin
```

```

    next_dir = 1'b0;
    next_count = count - 1'b1;
end
else if (flip == 1'b1 & count != max & count != min & dir == 1'b0)begin
    next_dir = 1'b1;
    next_count = count + 1'b1;
end
// DEBUG DONE
else begin
    if (dir == 1'b1)begin
        if (count == max) begin
            next_dir = 1'b0;
            next_count = count - 1'b1;
        end
        else begin
            next_count = count + 1'b1;
            next_dir = dir;
        end
    end
    // DEBUG DONE
    else begin
        if (count == min) begin
            next_dir = 1'b1;
            next_count = count + 1'b1;
        end
        else begin
            next_count = count - 1'b1;
            next_dir = dir;
        end
    end
    // DEBUG DONE

```

```

        end
    end
    else begin
        next_count = count;
        next_dir = dir;
    end
end
else begin
    next_count = count;
    next_dir = dir;
end
end
end

```

### c. Testbench

For the testbench, i put the values manually with the help of **cycle** to define about how many clocks / cycle so i can give room for the counter to make a move.

An example of the testbench is like so

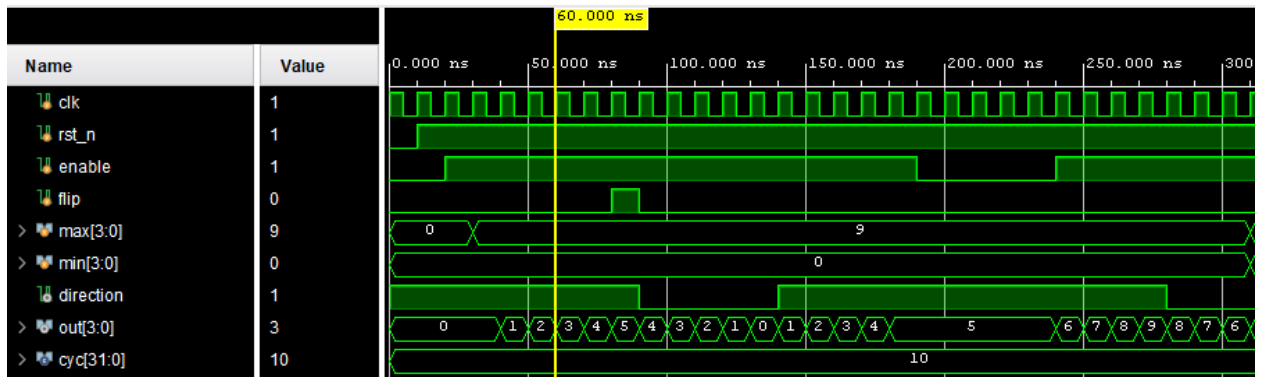
```

always#(cyc/2)clk = !clk;
initial begin
    clk = 1'b1;
    rst_n = 1'b0;
    enable = 1'b0;
    flip = 1'b0;
    max = 1'b0;
    min = 1'b0;
    @(posedge clk) rst_n = 1'b1;
    @(posedge clk) enable = 1'b1;
    @(posedge clk)begin
        max = 4'd9;
        min = 4'b0;
    end
end

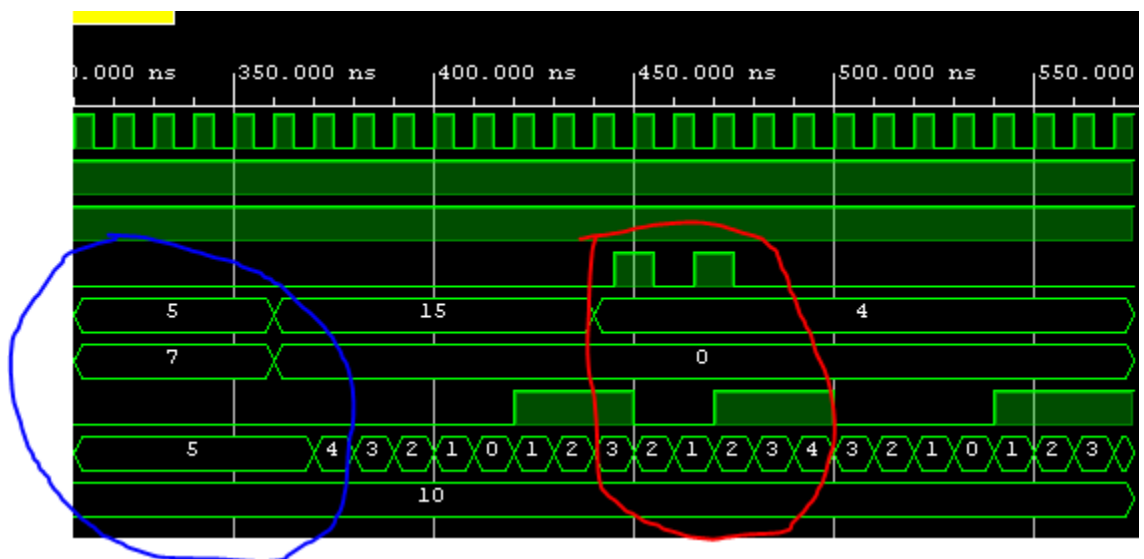
```

Where i set the parameter of cyc to be 10, so which means a cycle is a clock posedge and negedge. Where i declare it with **red**.

#### d. Waveforms



The cases that i tested, is when there is a flip occurring, i set the max = 9 and min = 0, then the flip occurs on the 5th cycle which is on 5, so it will start counting down. Then after it reaches 0, it will go back counting upwards, but it is a parameterized flip, not a manual flip, then when the enable is 0, it will hold its value which is currently on 5. And starts counting again when the enable has been turned on again, and goes back down from 9 because it has reached the parameter.



Then on this next waveform, the count is on 5, but the min is 7, so it is out of bounds, the waveform is like the one circled in **blue**. If its out of bounds it will hold its current value, then will move on until the **max or min is initialized again**. Then when 2 flips occur the

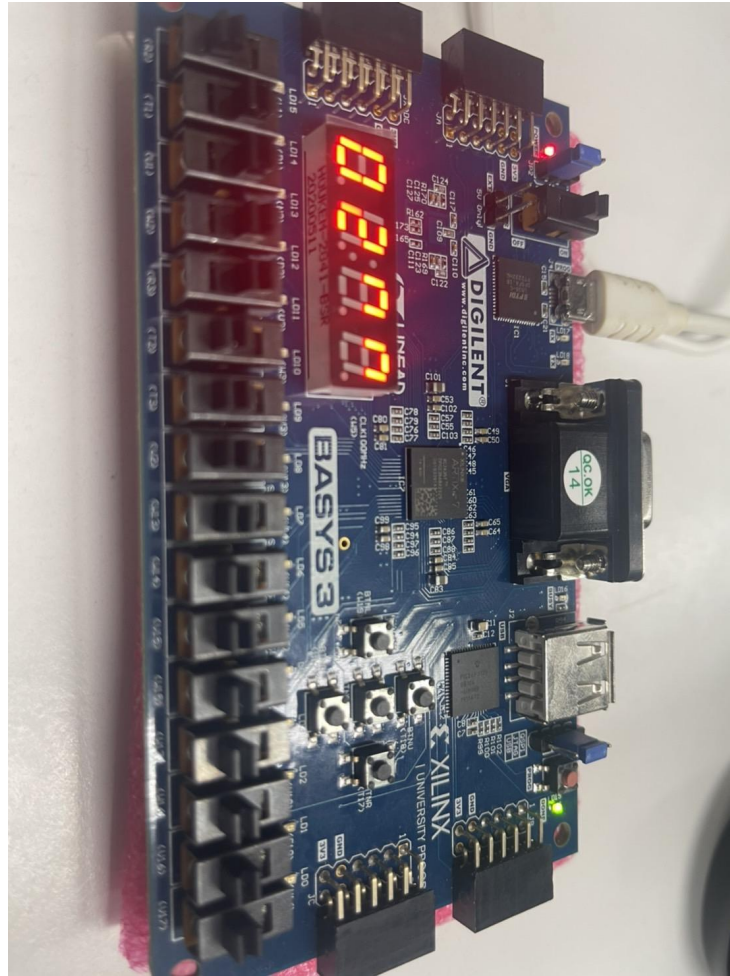
waveform is like the one circled in red. And that concludes the waveforms of the test cases i have imagined.

## 6. FPGA Implementation

For the FPGA Implementation, it is very challenging for me, especially the buttons which until now i couldn't fix because of clock. I knew i had to use the FPGA Clock which is 100Mhz, but the **button just wouldn't register**, the **register** only works if i hold in so i am forced to use the clock which is used for the LED, so it changes. The solutions which i have thought in my mind to solve the button problem was like so

Using Debounce and One-Pulse for the buttons	Doesn't Work because i havent init the clock
Without Using Debounce & One-Pulse	Nothing Works :')
Different Clock Division for the Button and the LEDs	LED works, the button is messed up for some reason.
Using Debounce with normal clock, and one pulse using divided clock ( $2 \times 25$ ) which implies around 0,5 second	LED Works, But the button has to be held, because used the same clock for 0,5s
Apply Debounce and One Pulse with the normal clock	LED Works, but the button didn't work.

These are the things that i have tried to fix the button, but it failed sadly, so i still have to hold the button.



**Figure 6.1 Basys3 Board with Parameterized Ping Pong Counter Implementation.**

#### a. Code

At first, i was kind of confused to assign the Anodes, but thankfully **fpga4student.com** saves the day again. Where the clock refreshes, from left to right so you can assign which anodes you can turn on

```
2'b00: begin
    Anode = 4'b01111;
    LED_bcd = out/10;
end
```

This is an example of the very most left LED, when the counter is 0, it will turn on the leftmost LED .

And this is for changing the **bcd outputs** into **cathodes**. Which here is declared as  
LEDOut

```
4'b0000: LEDOut = 7'b0000001; // 0
4'b0001: LEDOut = 7'b1001111; // 1
4'b0010: LEDOut = 7'b0010010; // 2
4'b0011: LEDOut = 7'b0000110; // 3
4'b0100: LEDOut = 7'b1001100; // 4
4'b0101: LEDOut = 7'b0100100; // 5
4'b0110: LEDOut = 7'b0100000; // 6
4'b0111: LEDOut = 7'b0001111; // 7
4'b1000: LEDOut = 7'b0000000; // 8
4'b1001: LEDOut = 7'b0000100; // 9
4'b1110: LEDOut = 7'b1100011; // down
4'b1111: LEDOut = 7'b0011101; // up
default: LEDOut = 7'b1111111; // dead
```

**//Debounce and OnePulse for Flip and Reset**

```
debounce debounce_flip (.pb_debounced(flip_db), .pb(flip), .clk(clk));
debounce debounce_reset (.pb_debounced(reset_db), .pb(rst_n), .clk(clk));
onepulse onepulse_flip (.pb_debounced(flip_db), .clk(dclk), .pb_one_pulse(flip_op));
onepulse_reset(.pb_debounced(reset_db), .clk(dclk), .pb_one_pulse(reset_op));
```

This is where i assign the debounce and onepulse, where the onepulse is the clock which is divided into 0,5 second

```
module debounce(pb_debounced, pb, clk);
    input pb, clk;
    output pb_debounced;

    reg [3:0] DFF;
```

```

always @(posedge clk) begin
    DFF[3:1] <= DFF[2:0];
    DFF[0] <= pb;
end

assign pb_debounced = ((DFF == 4'b1111) ? 1'b1 : 1'b0);

endmodule

```

This is the debounce module which i took from the professor's slide, which is used to stabilize the signals. So it doesn't keep changing from 0 to 1 or 1 to 0.

And this is how i divide the clock , where it is declared on **dclk**.

```

always @(posedge clk) begin
    refresh_counter <= refresh_counter + 1;
    ClkCounter = ClkCounter + 1'b1;
    dclk <= (ClkCounter % ( 2 ** 25)) ? 1'b0 : 1'b1;
end

```

```

module onepulse(pb_debounced, clk, pb_one_pulse);
    input pb_debounced;
    input clk;
    output reg pb_one_pulse;
    reg pb_debounced_delay;

    always @(posedge clk) begin
        pb_one_pulse <= pb_debounced & (!pb_debounced_delay);
        pb_debounced_delay <= pb_debounced;
    end

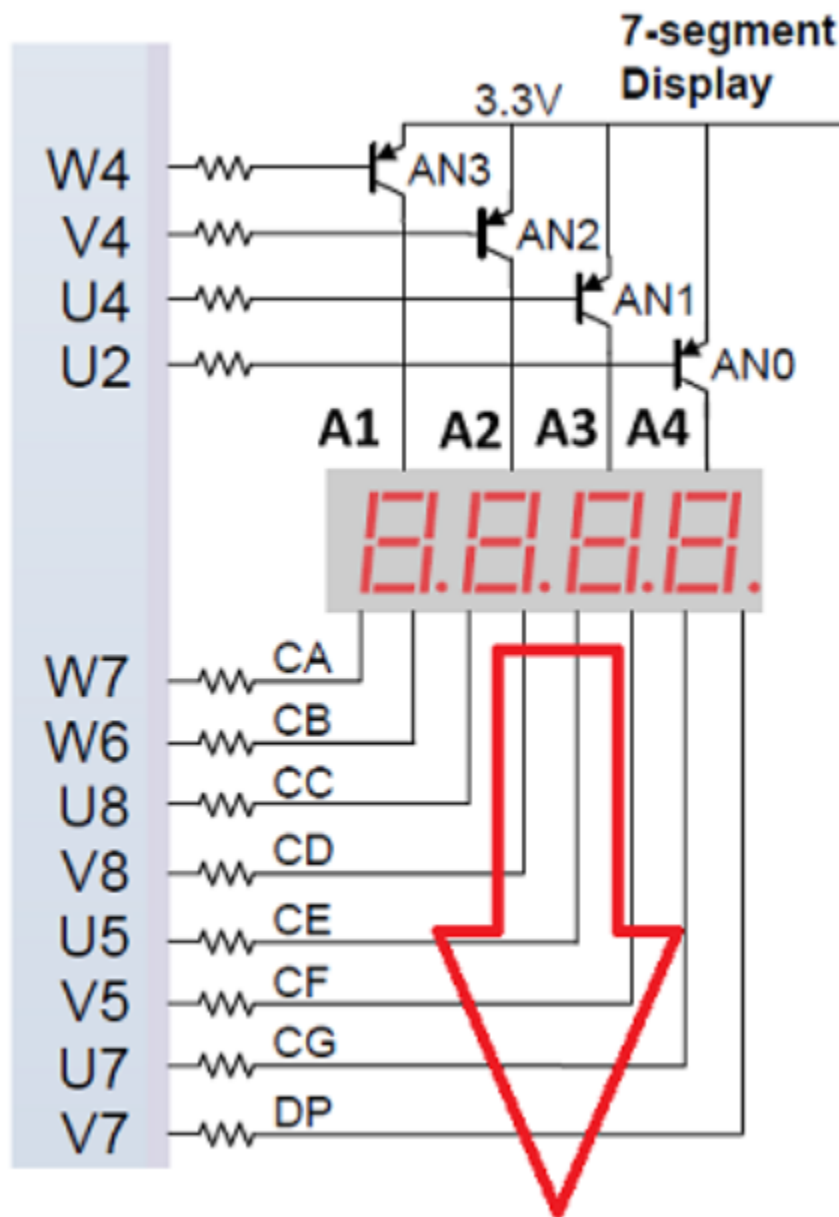
endmodule

```

And this is the onepulse module to register only 1 time input even though the button is held or pressed multiple times.





















Then to assign the LEDs to the destined ports i take the basys3 specification from [fpga4student.com](http://fpga4student.com) website which is so








**Figure 6.2 Ports for LED**

to assign the I/O Ports to use in the FPGA Board. Which is like so.

▼  Anode (4)	OUT				<input checked="" type="checkbox"/>	34	LVC MOS33*	▼
 Anode[3]	OUT		W4	▼	<input checked="" type="checkbox"/>	34	LVC MOS33*	▼
 Anode[2]	OUT		V4	▼	<input checked="" type="checkbox"/>	34	LVC MOS33*	▼
 Anode[1]	OUT		U4	▼	<input checked="" type="checkbox"/>	34	LVC MOS33*	▼
 Anode[0]	OUT		U2	▼	<input checked="" type="checkbox"/>	34	LVC MOS33*	▼
▼  LEDOut (7)	OUT				<input checked="" type="checkbox"/>	34	LVC MOS33*	▼
 LEDOut[6]	OUT		W7	▼	<input checked="" type="checkbox"/>	34	LVC MOS33*	▼
 LEDOut[5]	OUT		W6	▼	<input checked="" type="checkbox"/>	34	LVC MOS33*	▼
 LEDOut[4]	OUT		U8	▼	<input checked="" type="checkbox"/>	34	LVC MOS33*	▼
 LEDOut[3]	OUT		V8	▼	<input checked="" type="checkbox"/>	34	LVC MOS33*	▼
 LEDOut[2]	OUT		U5	▼	<input checked="" type="checkbox"/>	34	LVC MOS33*	▼
 LEDOut[1]	OUT		V5	▼	<input checked="" type="checkbox"/>	34	LVC MOS33*	▼
 LEDOut[0]	OUT		U7	▼	<input checked="" type="checkbox"/>	34	LVC MOS33*	▼
▼  max (4)	IN				<input checked="" type="checkbox"/>	34	LVC MOS33*	▼
 max[3]	IN		T1	▼	<input checked="" type="checkbox"/>	34	LVC MOS33*	▼
 max[2]	IN		U1	▼	<input checked="" type="checkbox"/>	34	LVC MOS33*	▼
 max[1]	IN		W2	▼	<input checked="" type="checkbox"/>	34	LVC MOS33*	▼
 max[0]	IN		R3	▼	<input checked="" type="checkbox"/>	34	LVC MOS33*	▼

▼  Anode (4)	OUT				<input checked="" type="checkbox"/>	34	LVC MOS33*	▼
 Anode[3]	OUT		W4	▼	<input checked="" type="checkbox"/>	34	LVC MOS33*	▼
 Anode[2]	OUT		V4	▼	<input checked="" type="checkbox"/>	34	LVC MOS33*	▼
 Anode[1]	OUT		U4	▼	<input checked="" type="checkbox"/>	34	LVC MOS33*	▼
 Anode[0]	OUT		U2	▼	<input checked="" type="checkbox"/>	34	LVC MOS33*	▼









▼  LEDOut (7)	OUT				<input checked="" type="checkbox"/>	34	LVC MOS33*	▼
 LEDOut[6]	OUT		W7	▼	<input checked="" type="checkbox"/>	34	LVC MOS33*	▼
 LEDOut[5]	OUT		W6	▼	<input checked="" type="checkbox"/>	34	LVC MOS33*	▼
 LEDOut[4]	OUT		U8	▼	<input checked="" type="checkbox"/>	34	LVC MOS33*	▼
 LEDOut[3]	OUT		V8	▼	<input checked="" type="checkbox"/>	34	LVC MOS33*	▼
 LEDOut[2]	OUT		U5	▼	<input checked="" type="checkbox"/>	34	LVC MOS33*	▼
 LEDOut[1]	OUT		V5	▼	<input checked="" type="checkbox"/>	34	LVC MOS33*	▼
 LEDOut[0]	OUT		U7	▼	<input checked="" type="checkbox"/>	34	LVC MOS33*	▼

Figure 6.3 Sample of Assigning I/O Ports

## 7. Workload

**Jason** was in charge of Part 1,5,FPGA and to write the report of 1,5,FPGA and helped Tundo with checking FIFO.

While **Tuguldur** was in charge of Part 2,3,4 and writing the report of 2,3,4

## **8. Problems We Encountered and What We Learn.**

We have spent countless hours, multiple days and nights. And Jason had a midterm during the week, so he wasn't able to allocate more time into the lab. For Tuguldur, his roommate had covid, so he was also busy quarantining and moving things around.

Unable to come to the university for 7 days, Tuguldur had a hard time alone in quarantine. With not many friends to discuss and ask about his bugs. Also, there was not much helpful information on the internet. Due to this, Tuguldur was unable to completely debug his code where there are still some bugs inside.

The day before demo day, a disaster struck where Jason's External SSD where he kept Vivado and all the lab files, broke **T\_T**. Jason was literally panicking and stayed up until 4 trying to recover the files, but no luck. But, thankfully since he coded using Visual Studio Code, the cache was still there. So, This Lab was saved by VSC's Cache :) .

Jason had problems at first about the condition of Finite State Machine, and how to do a better coding style for Sequential Circuits and the Combinational, but eventually he did it? ( we hope so ).

For Tuguldur, he was struggling with the rest of the questions, but we hope that he did it too.

In this Lab, we have learned to use Sequential Circuit, and Combinational Circuit to implement different Circuits.