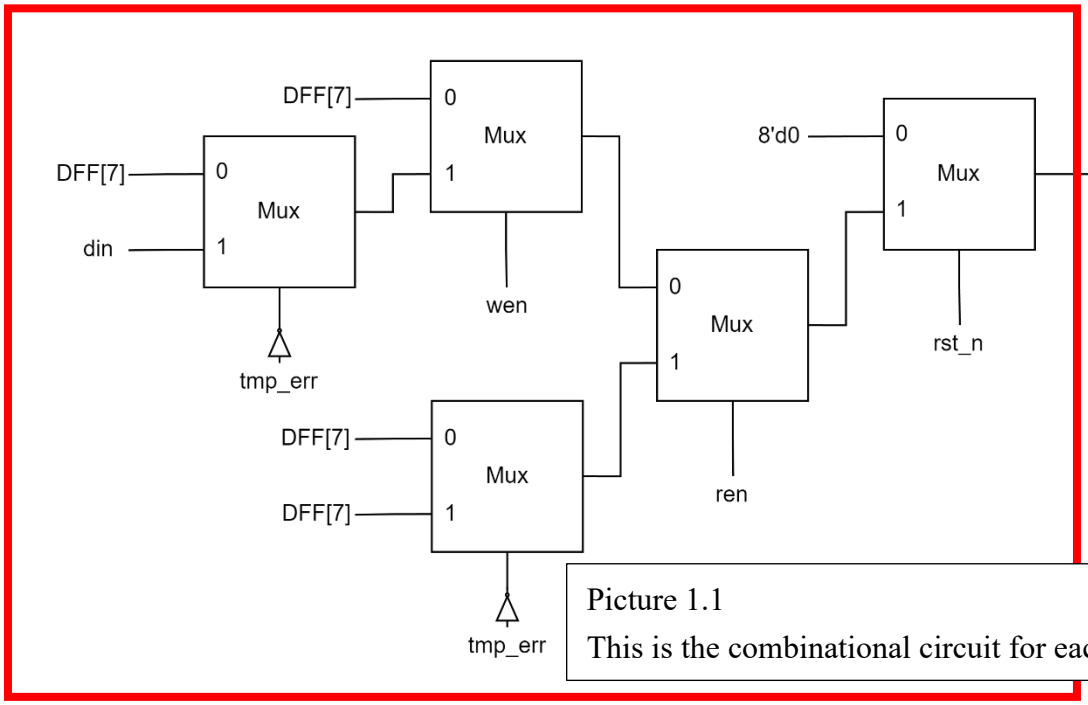
Hardware Design and Lab: Lab3

111060013 EECS 26' 劉祐廷

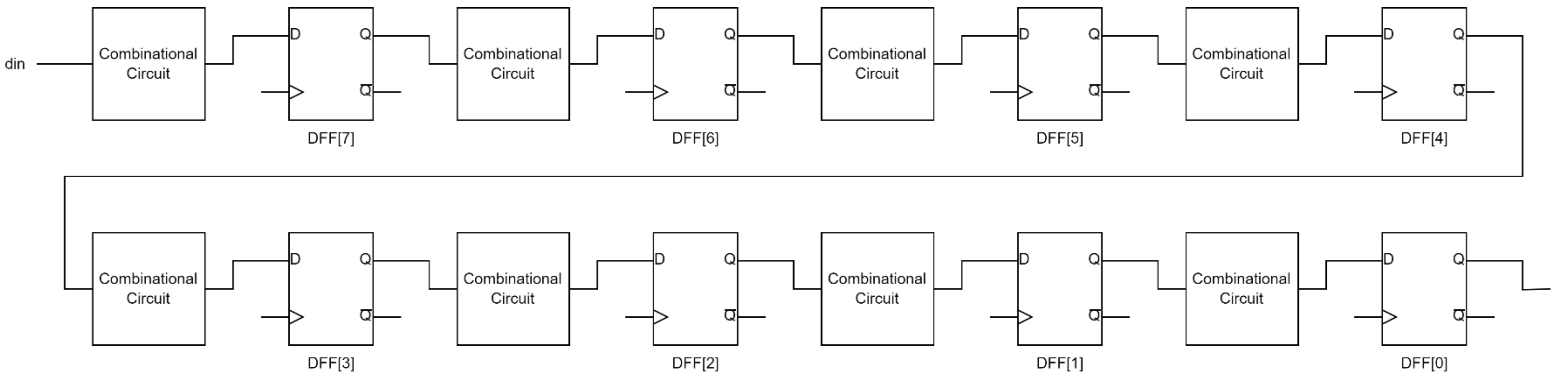
Catalog

1. Advanced Question: First-In First Out (FIFO) Queue

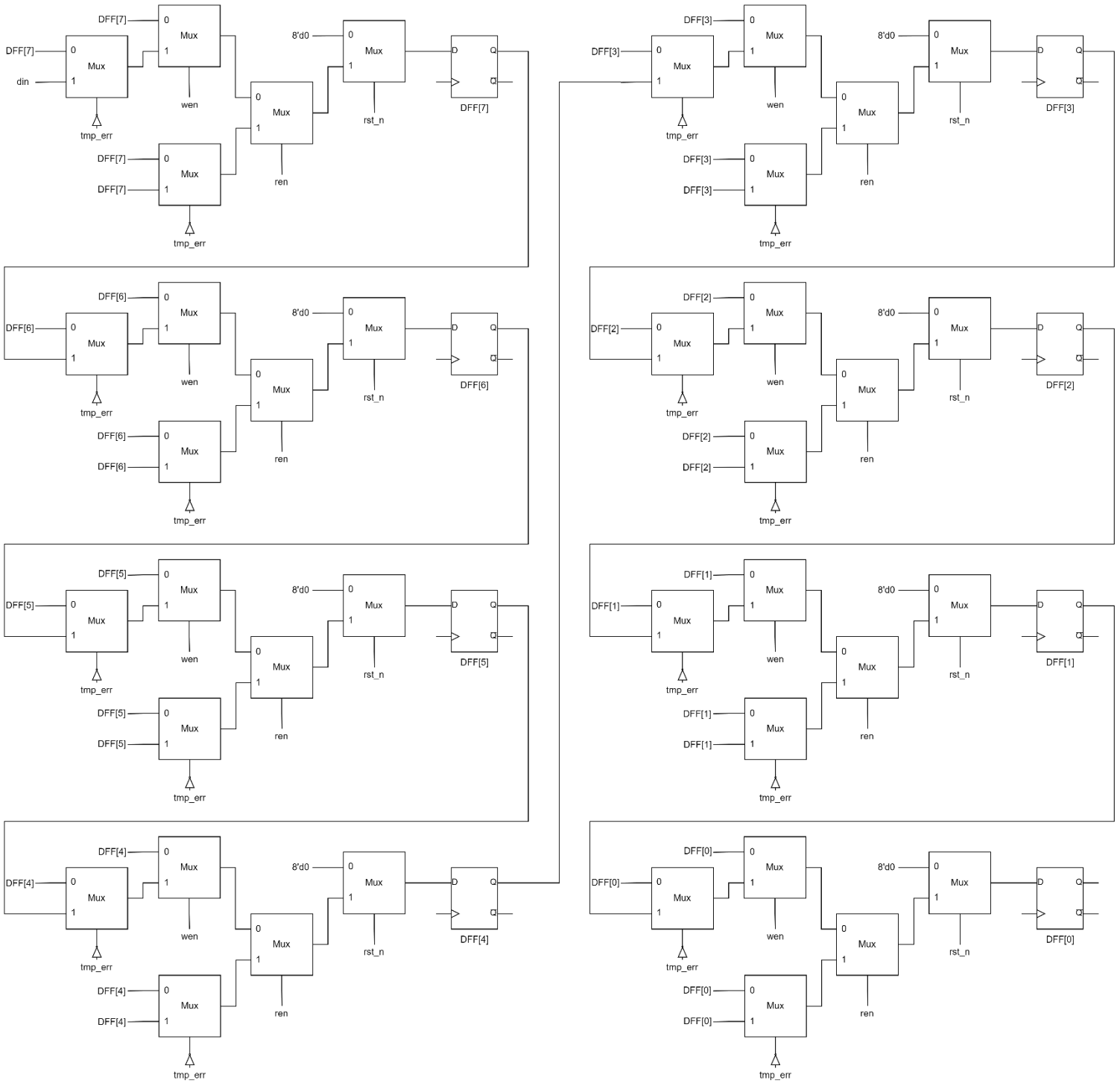
A. Block Diagram



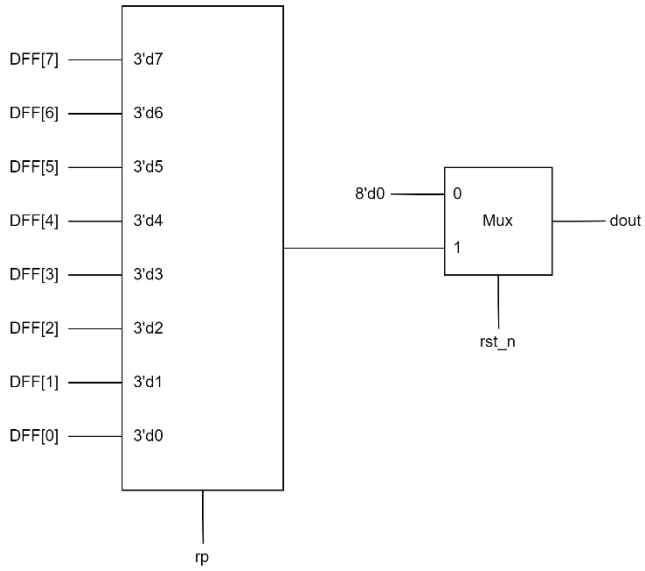
Picture 1.1
This is the combinational circuit for each DFF in the queue.



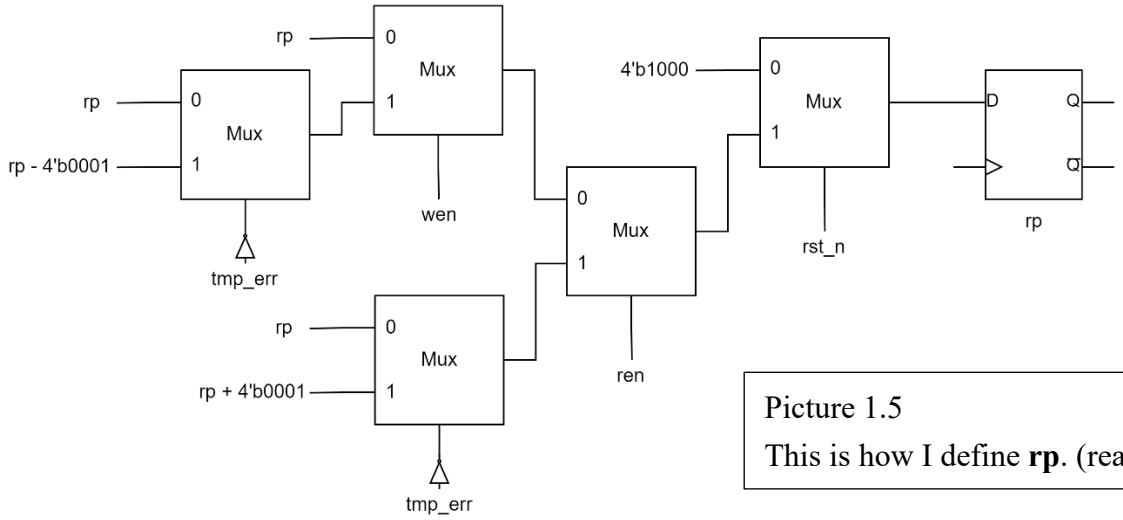
Picture 1.2
This is the architecture of the 8-bits FIFO Queue.



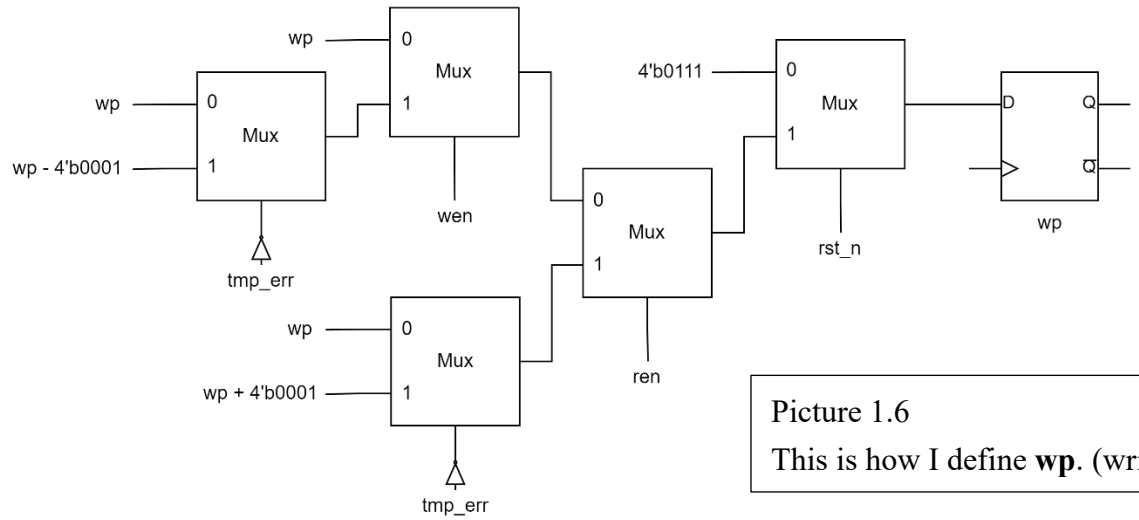
Picture 1.3
This is the block diagram of the 8-bits FIFO Queue.



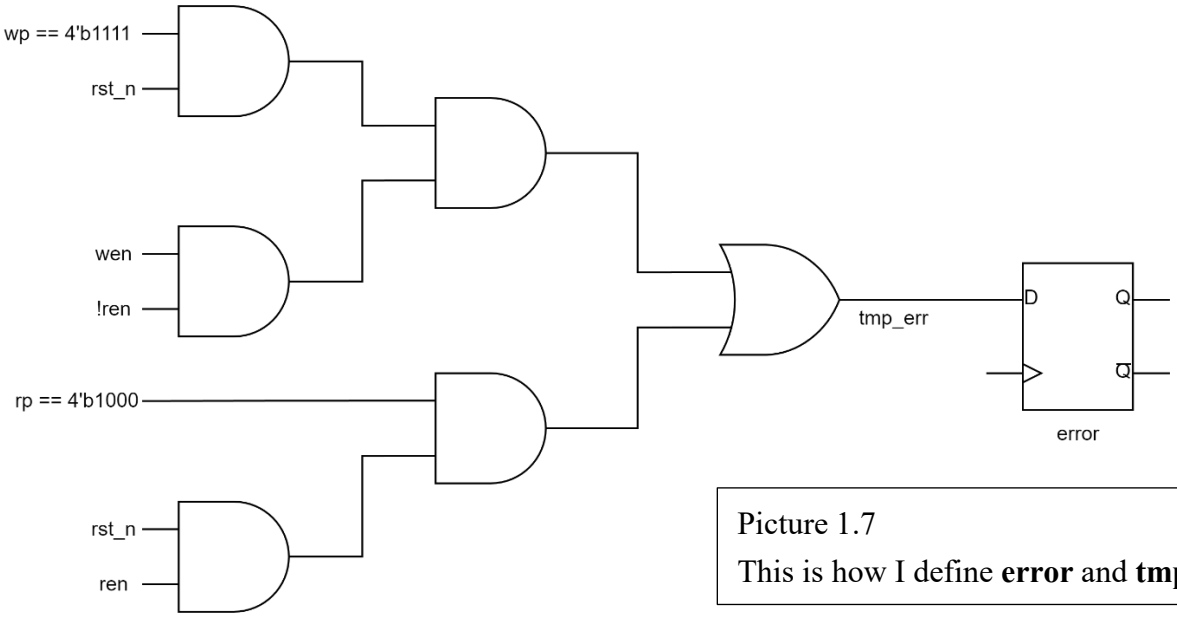
Picture 1.4
Choose the **dout** by **rp (read pointer)**.



Picture 1.5
This is how I define **rp**. (read pointer)



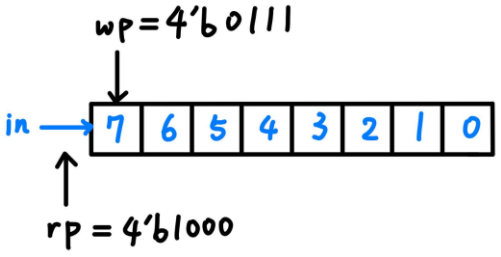
Picture 1.6
This is how I define **wp**. (write pointer)



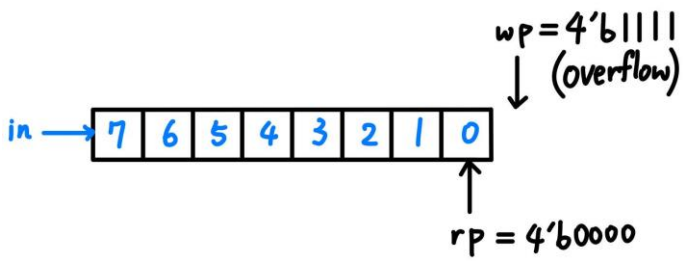
Picture 1.7
This is how I define **error** and **tmp_err**.

B. Explanation

In the beginning, I reset **rp** (read pointer) and **wp** (write pointer) to **4'b0111** and **4'b1111** respectively. If writing data into queue is successful, both of **rp** and **wp** will -1 as moving the pointers to the next DFF. If reading data out of queue is successful, both of **rp** and **wp** will +1 as moving the pointers to the previous DFF. The data in the queue will only be passed to the next DFF while writing is successful. Otherwise, they will be kept in the same DFF. The error cases are that **rp == 4'b1000** and **ren == 1'b1**, which means reading failed, or **wp == 4'b1111 (overflow when writing)**, **ren == 1'b0** and **wen == 1'b1**, which means writing failed. In these cases, **error** will be pulled up for a clock cycle.

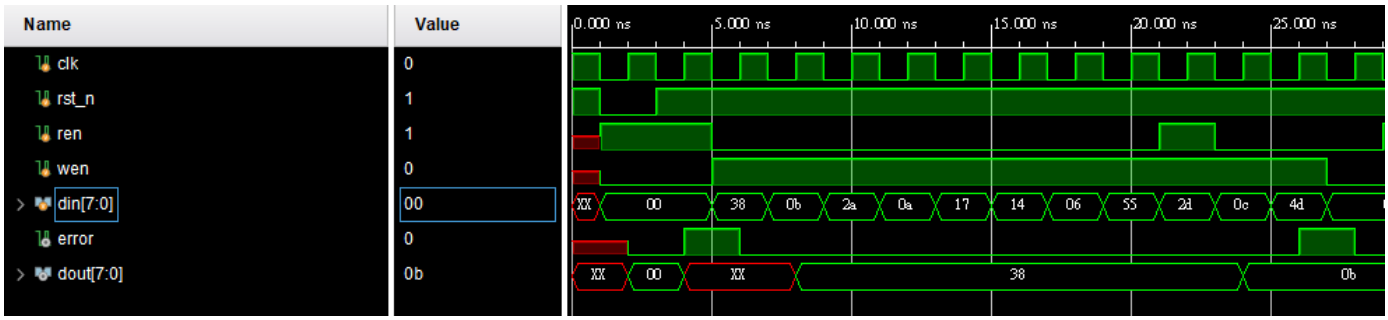


Picture 1.8 Reading failed



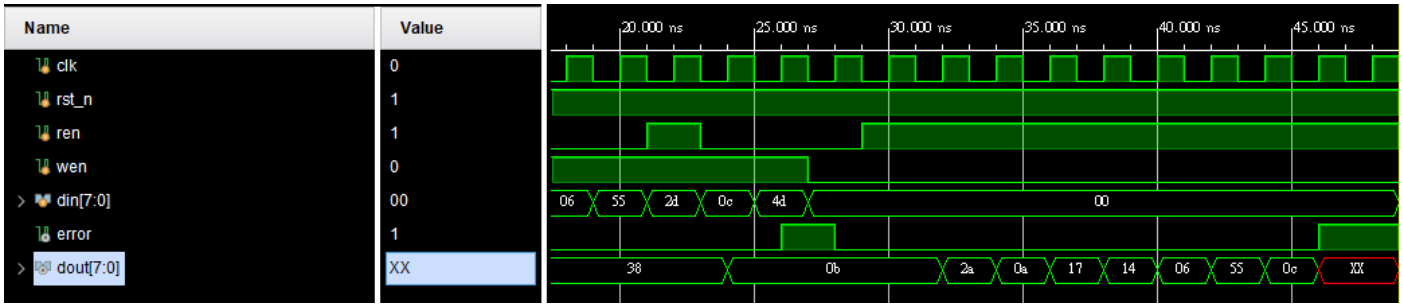
Picture 1.9 Writing failed

C. Testbench



Picture 1.10 Wave form 1

I test my design with the input in the lecture slide to see if there is something wrong or not. As the wave form showed in Picture 1.10m it looks like everything is correct. However, it only tests the condition of writing failed. Therefore, I let it read out all the data in the queue to check the condition of reading failed.

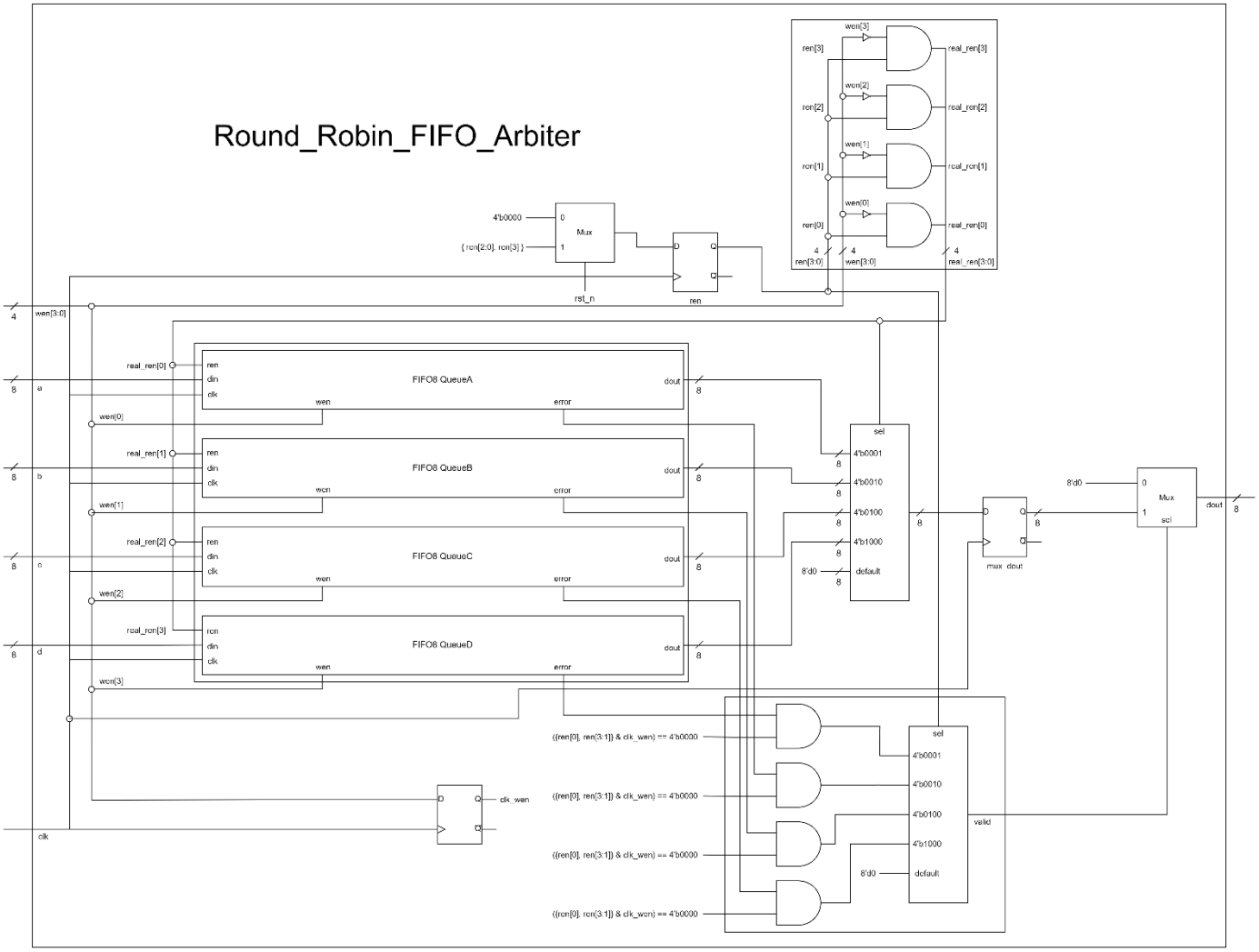


Picture 1.11 Wave form 2

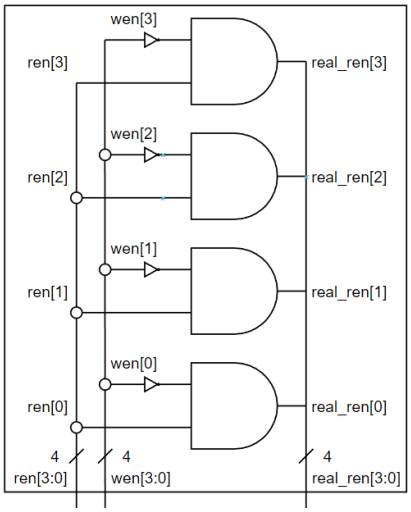
As the result showed in Picture 1.11, we can see that the condition of reading fail is also correct.

2. Advanced Question: Round-Robin FIFO Arbiter

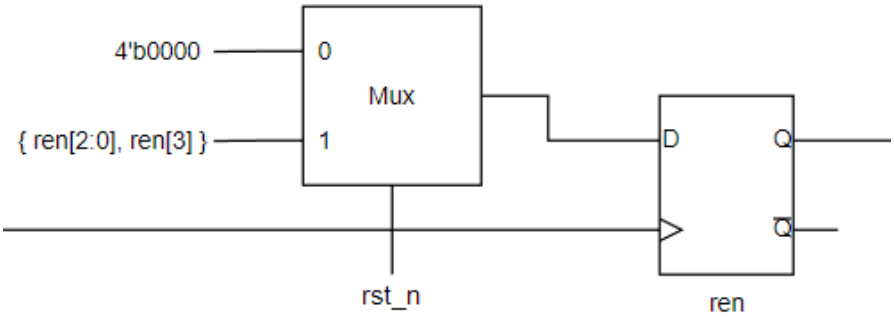
A. Block Diagram



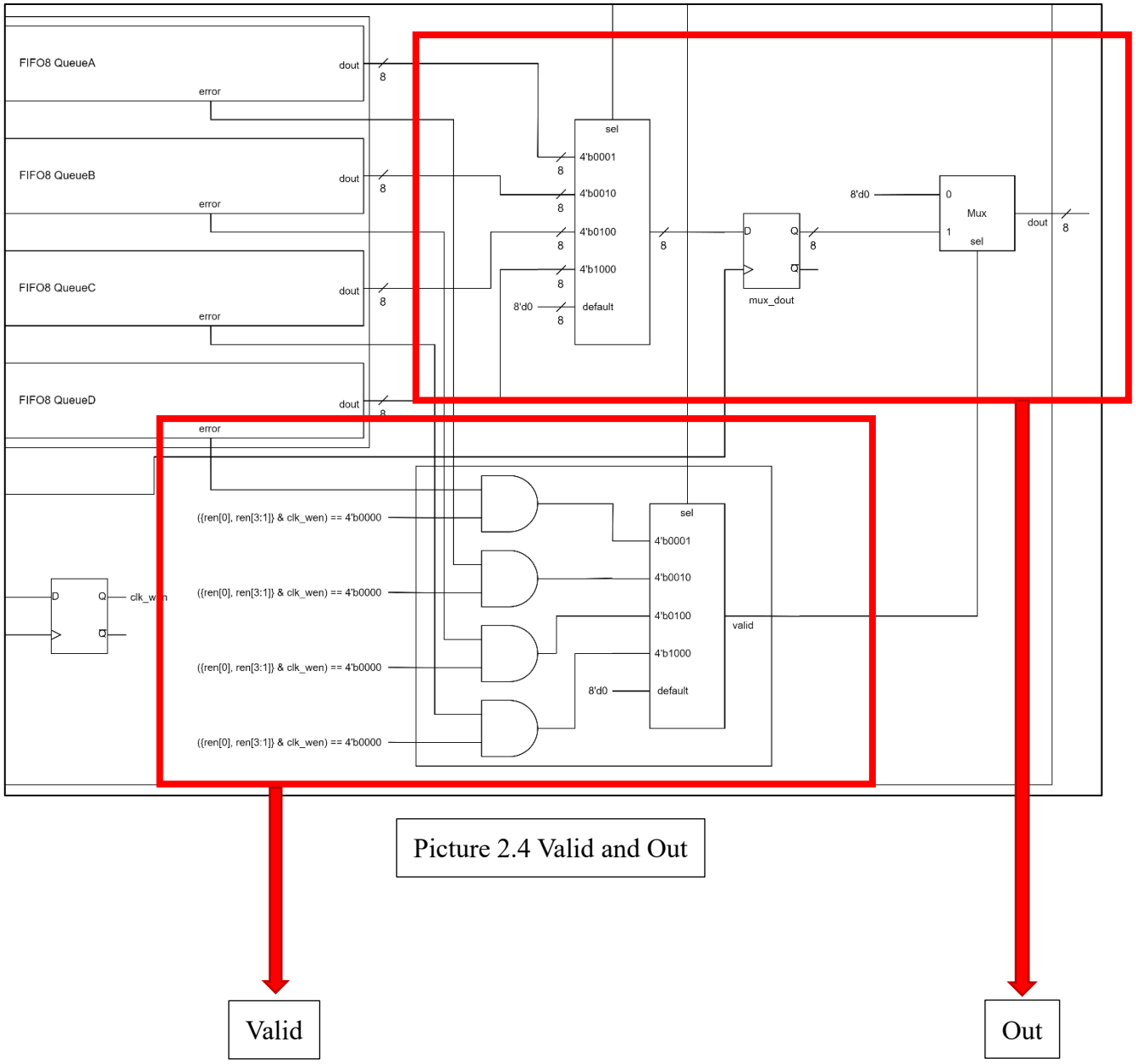
Picture 2.1 The whole design of Round-Robin FIFO Arbiter



Picture 2.2 real_ren



Picture 2.3 ren

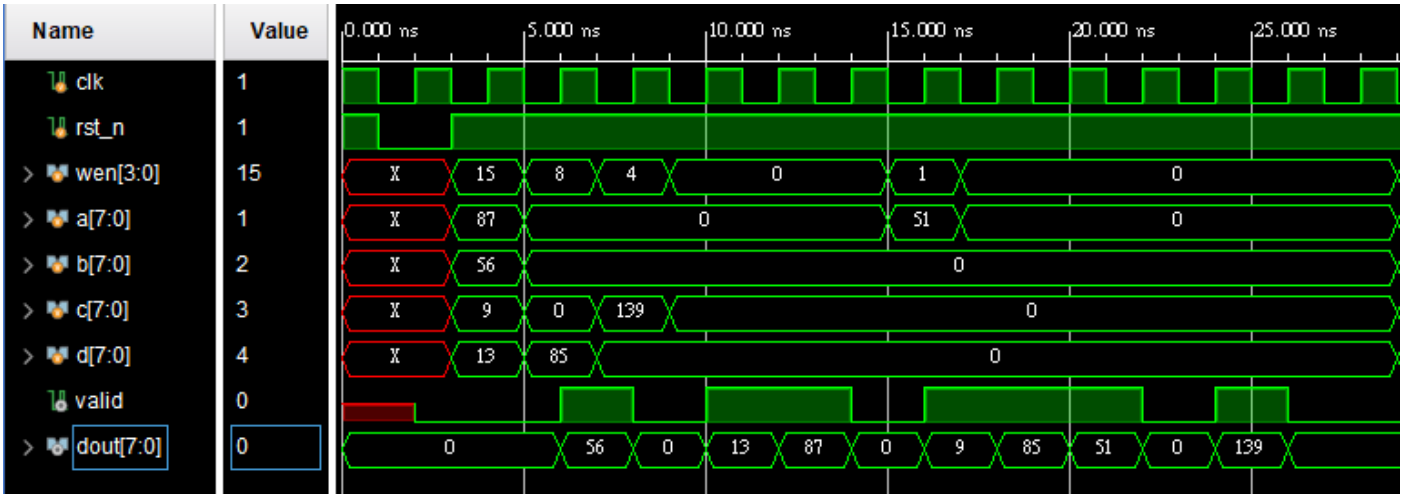


Picture 2.4 Valid and Out

B. Explanation

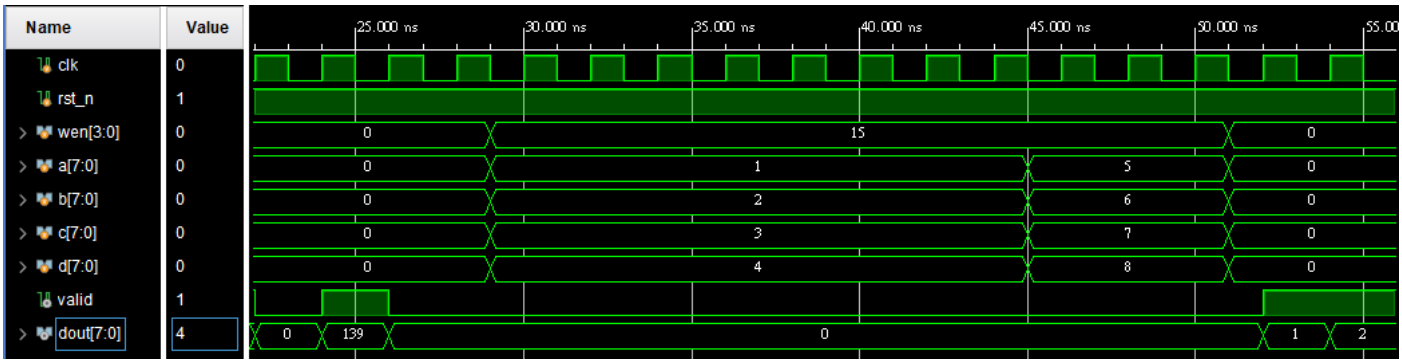
FIFO8 here is totally the same as the **FIFO8** in the advanced question, First-In First Out (FIFO) Queue. For **valid**, **error** in **FIFO8** is synchronized, so **valid** isn't needed to be contained in a DFF. However, because **valid** is not contained in a DFF, which means that it will change if **wen** changes, so I use a DFF to keep **wen** for a clock cycle which called **clk_wen** to help me define **valid** correctly. For **dout**, I use a 4-bit counter called **ren** to decide which queue should be access. Different from the advanced question, First-In First Out (FIFO) Queue, if both **ren** and **wen** is 1'b1, it will write instead of read. Therefore, I define an extra reg called **real_ren** to fix this problem.

C. Testbench

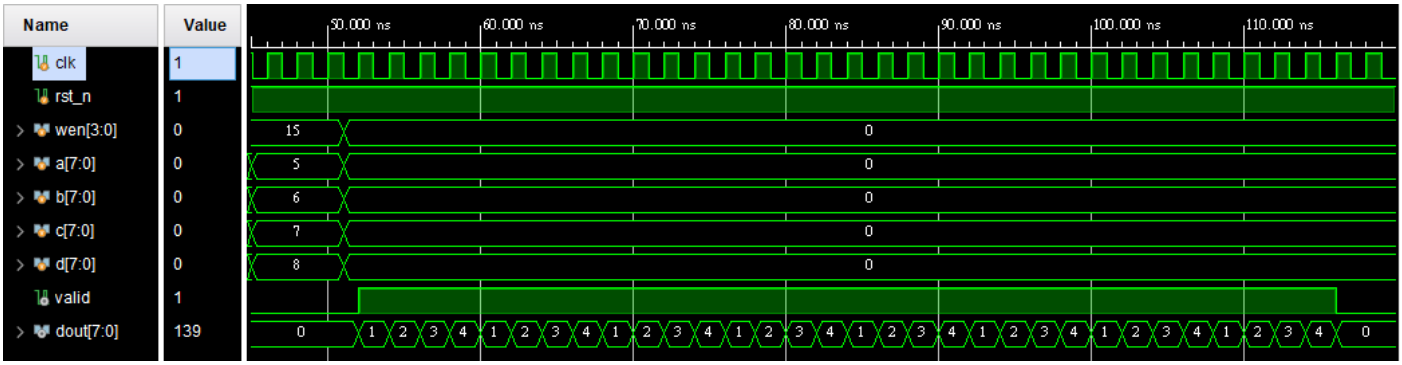


Picture 2.5 wave form 1

First, I use the input data from lecture slide to test if there is something wrong or not. As the result showed in Picture 2.5, we can see that everything works correctly. However, here we can't test the situation of writing failed. Therefore, after clear the queues, I let it write 8 times of same numbers and 1 times of different numbers. After that, I pop every queue for 9 times to check if everything is correct.



Picture 2.6 wave form 2

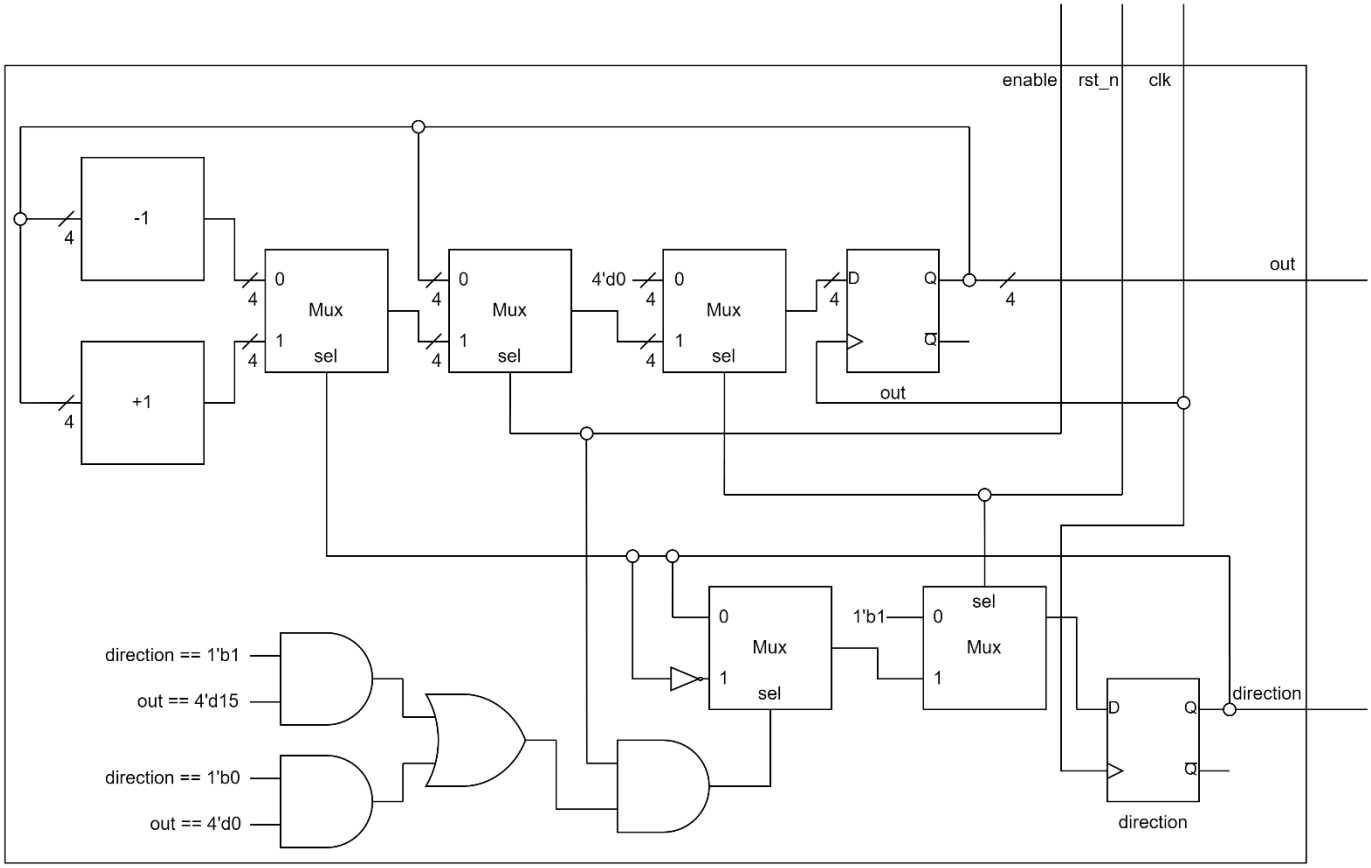


Picture 2.7 wave form 3

From Picture 2.6 and Picture 2.7, we can see that when the condition of reading failed or writing failed happened, my design can work correctly.

3. Advanced Question: 4-bit Ping-Pong Counter

A. Block Diagram

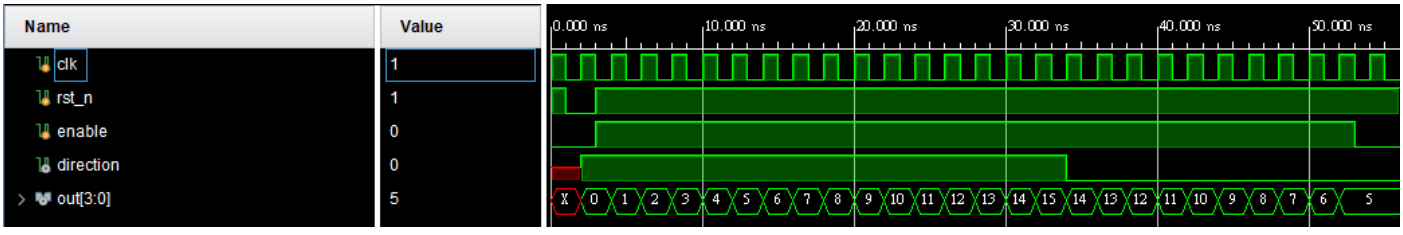


Picture 3.1 The whole design of 4-bit Ping-Pong Counter

B. Explanation

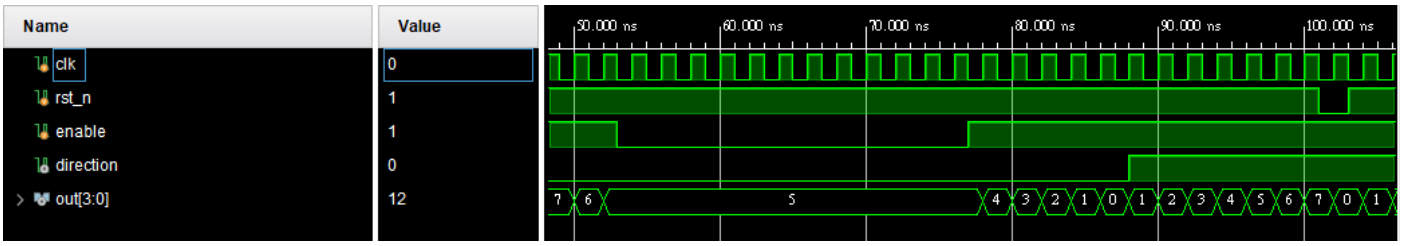
I use two DFF to contain **out** and **direction** respectively. If **direction == 1'b1** and **out == 4'd15**, which means the counter hits the upper bound, and **direction == 1'b0** and **out == 4'd0**, which means the counter hits the lower bound, **direction** should be inverted.

C. Testbench



Picture 3.2 wave form 1

For the beginning of the testbench, I test the counter without changing **enable** and **rst_n**. As the result showed in Picture 3.2, we can see that everything works properly.

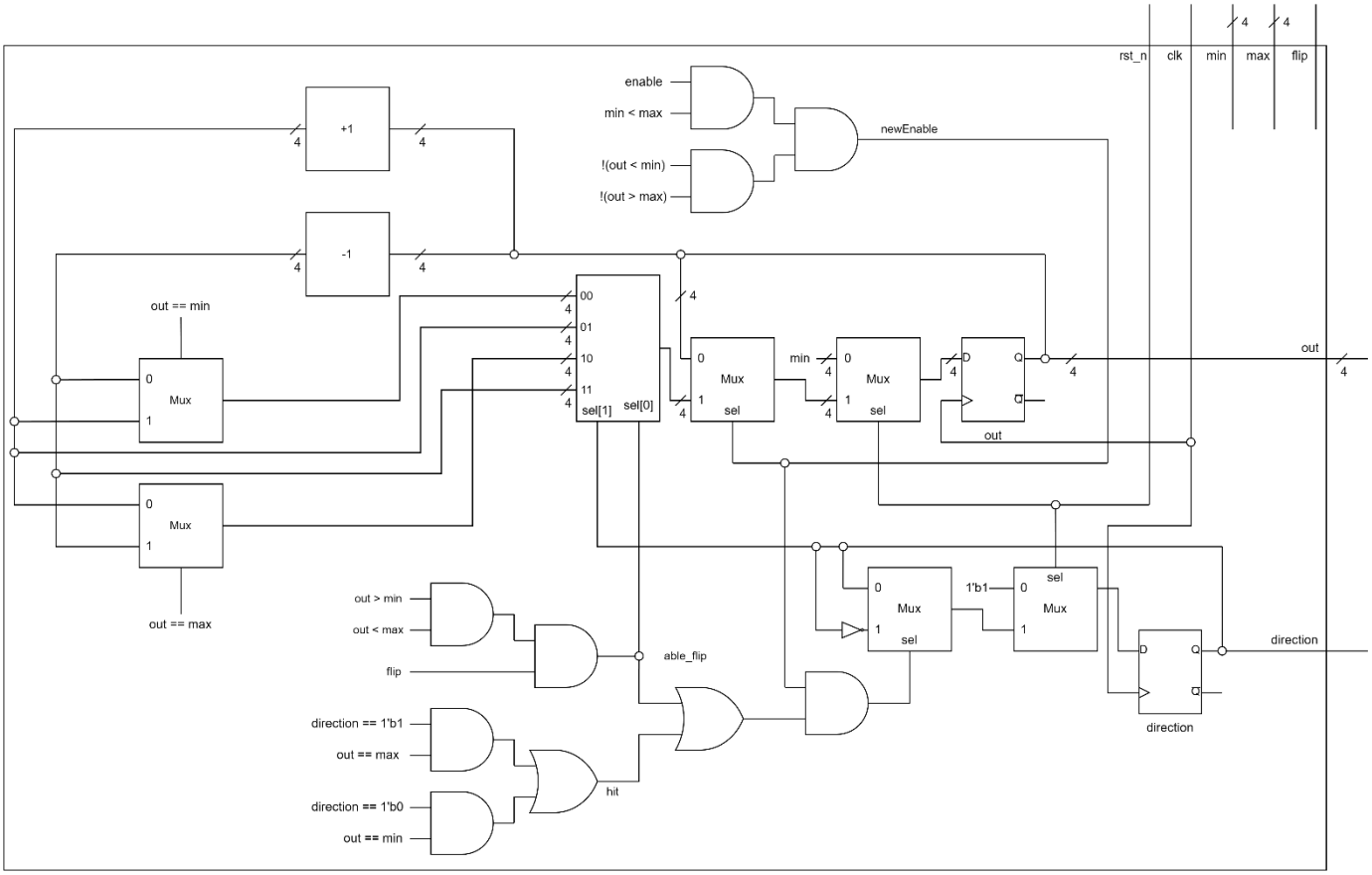


Picture 3.3 wave form 2

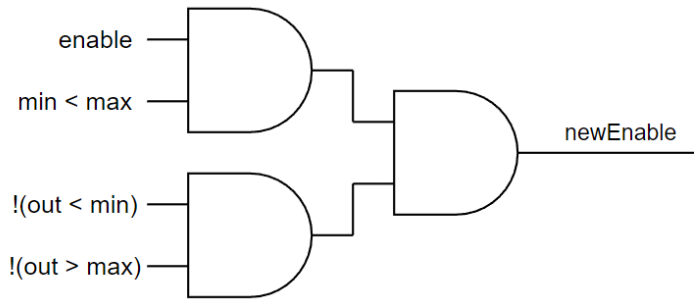
And then I test **enable** and **rst_n** to check if they work or not. From the result in Picture 3.3. It seems that everything is correct.

4. Advanced Question: 4-bit Parameterized Ping-Pong Counter

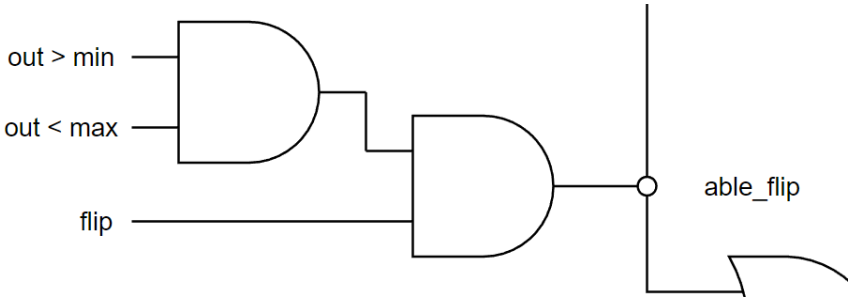
A. Block Diagram



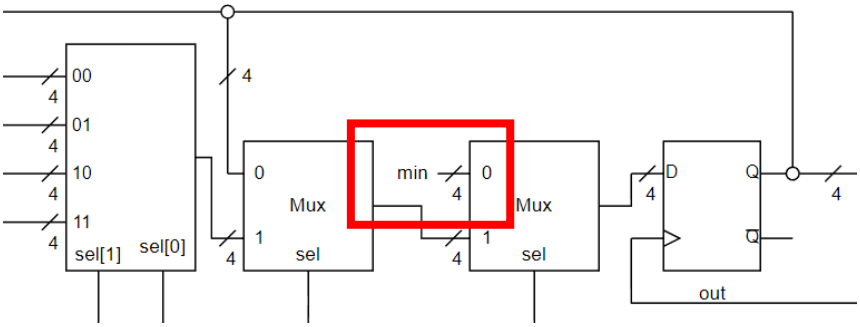
Picture 4.1 The whole design of 4-bit Parameterized Ping-Pong Counter



Picture 4.2 newEnable



Picture 4.3 able_flip

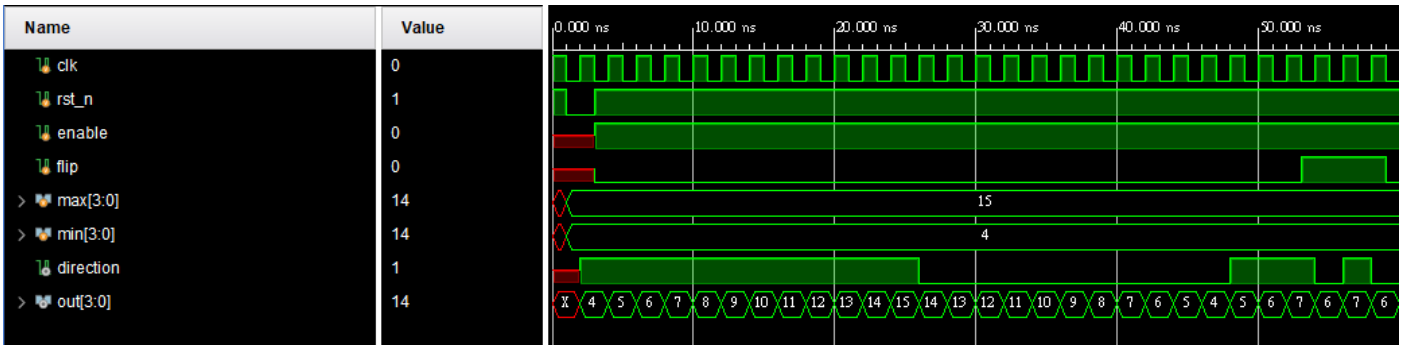


Picture 4.4 reset to min

B. Explanation

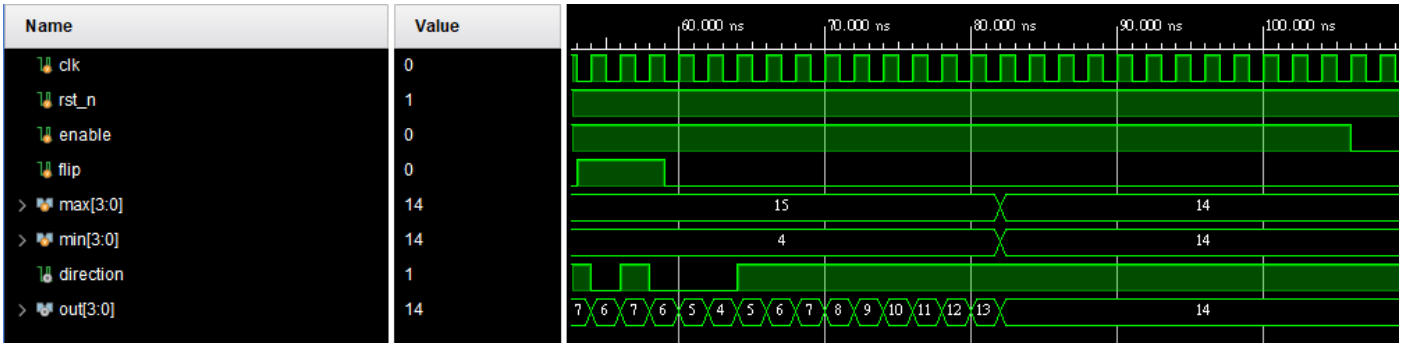
This module extends from the advanced question, 4-bit Ping-Pong Counter. Most of its architecture and methods are the same. There are three different things between them. First, in this problem, **enable** is not the only factor of triggering counter. Thus, I define a reg called **newEnable** like Picture 4.2. Second, there is an extra signal called **flip** in this problem. When **flip == 1'b1** and **min < out < max**, it will trigger the counter to change it direction of counting. I define a reg called **able_flip** to store this factor as Picture 4.2 shows. Third, according to the problem, while **rst_n == 1'b0**, I should set **out** to **min**. So I change the reset case of **out** as it shows in Picture 4.4.

C. Testbench



Picture 4.5 wave form 1

First of all, I test the counter counting from min to max back and forth. And then test if **flip** works properly. As Picture 4.5 shows, these two functions work properly.



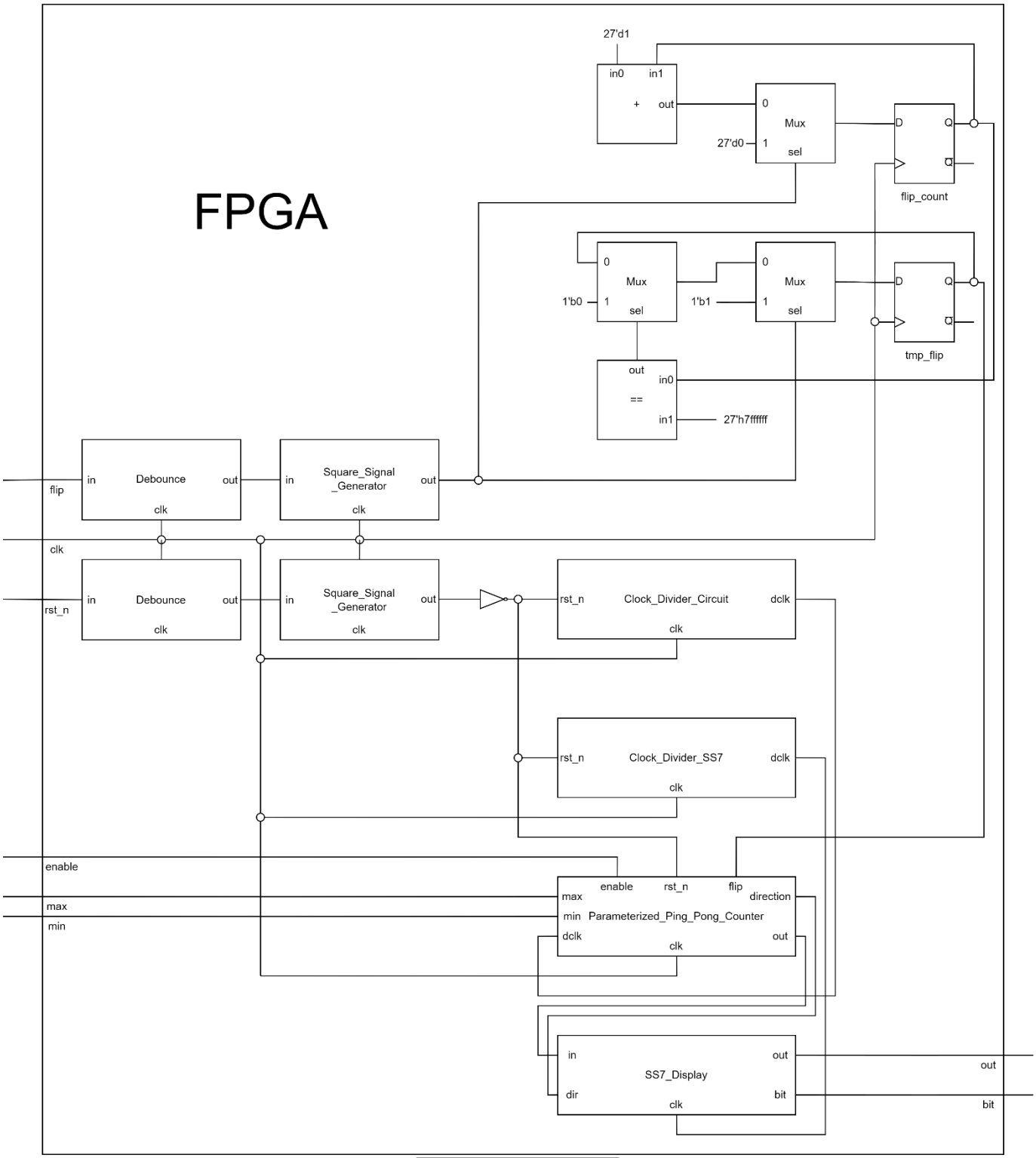
Picture 4.6 wave form 2

After that, I try to change **min** and **max** to check if **newEnable** works correctly or not. As Picture 4.6 shows, we can see that it is correct.

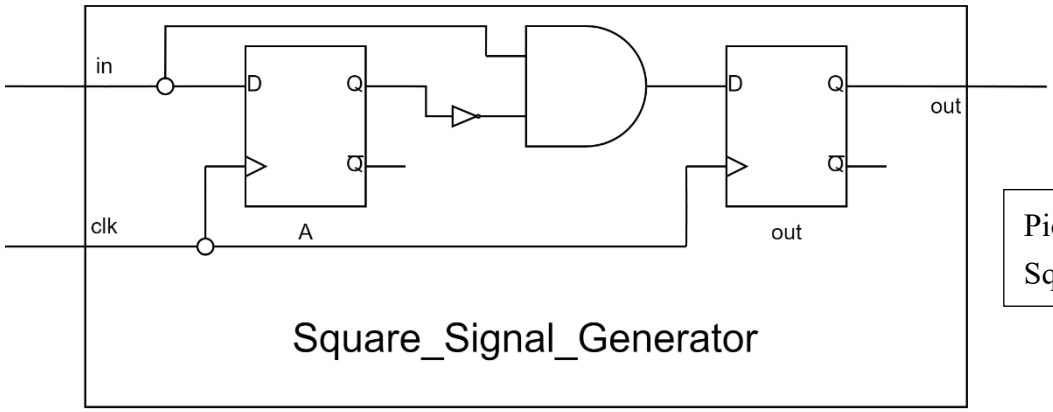
5. Advanced Question: 4-bit Parameterized Ping-Pong Counter

FPGA

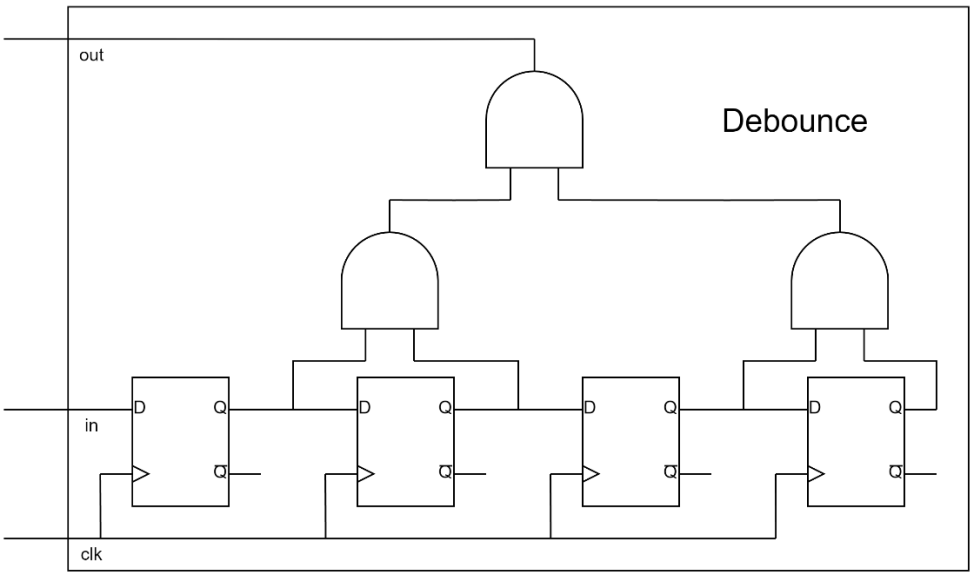
A. Block Diagram



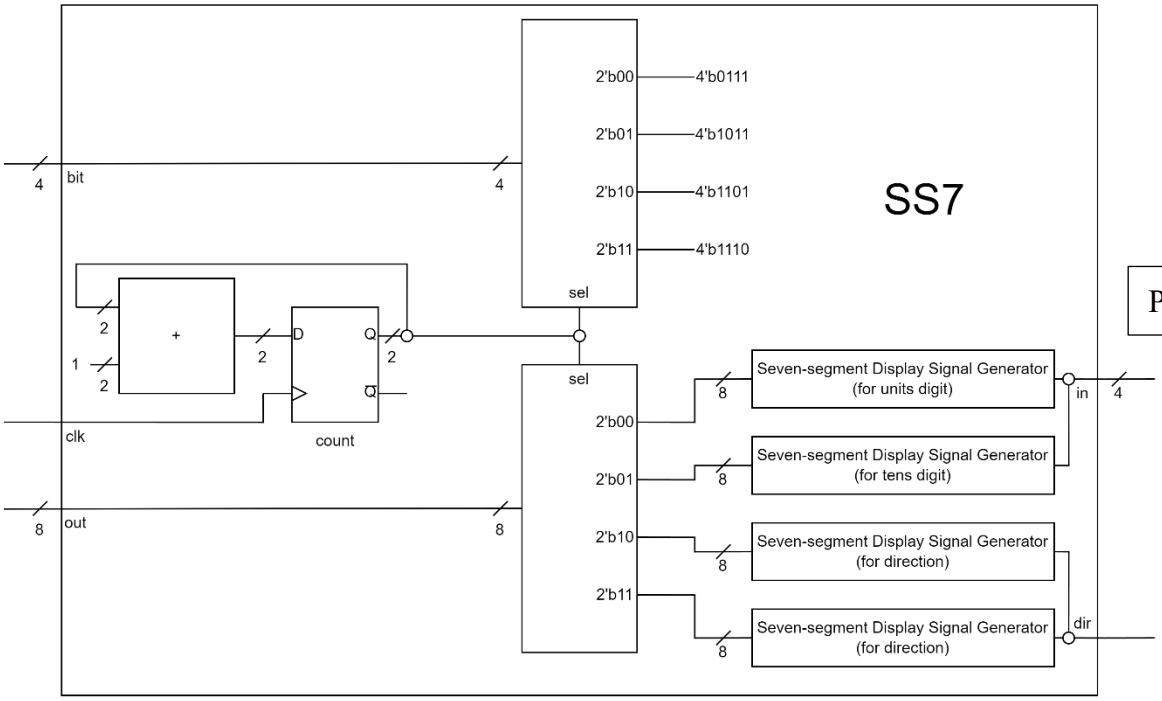
Picture 5.1 FPGA



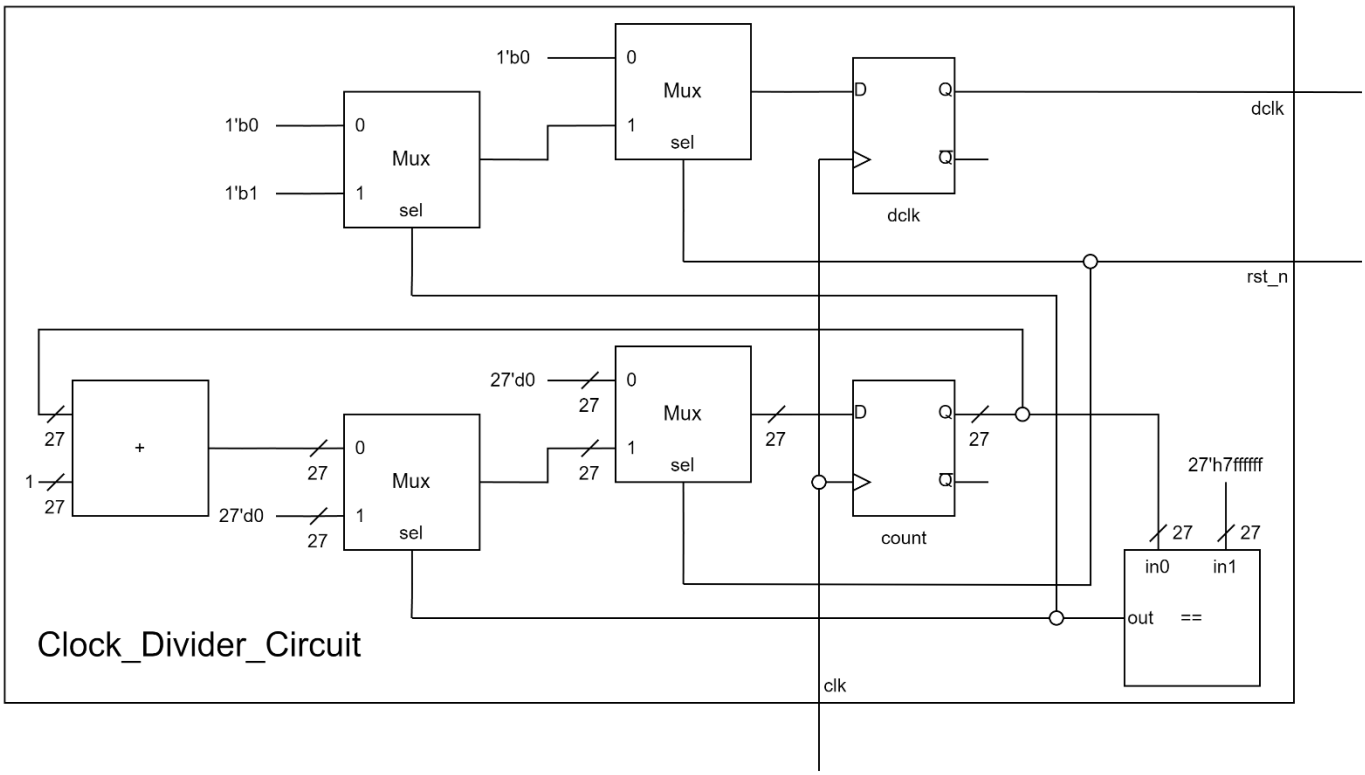
Picture 5.2
Square_Signal_Generator



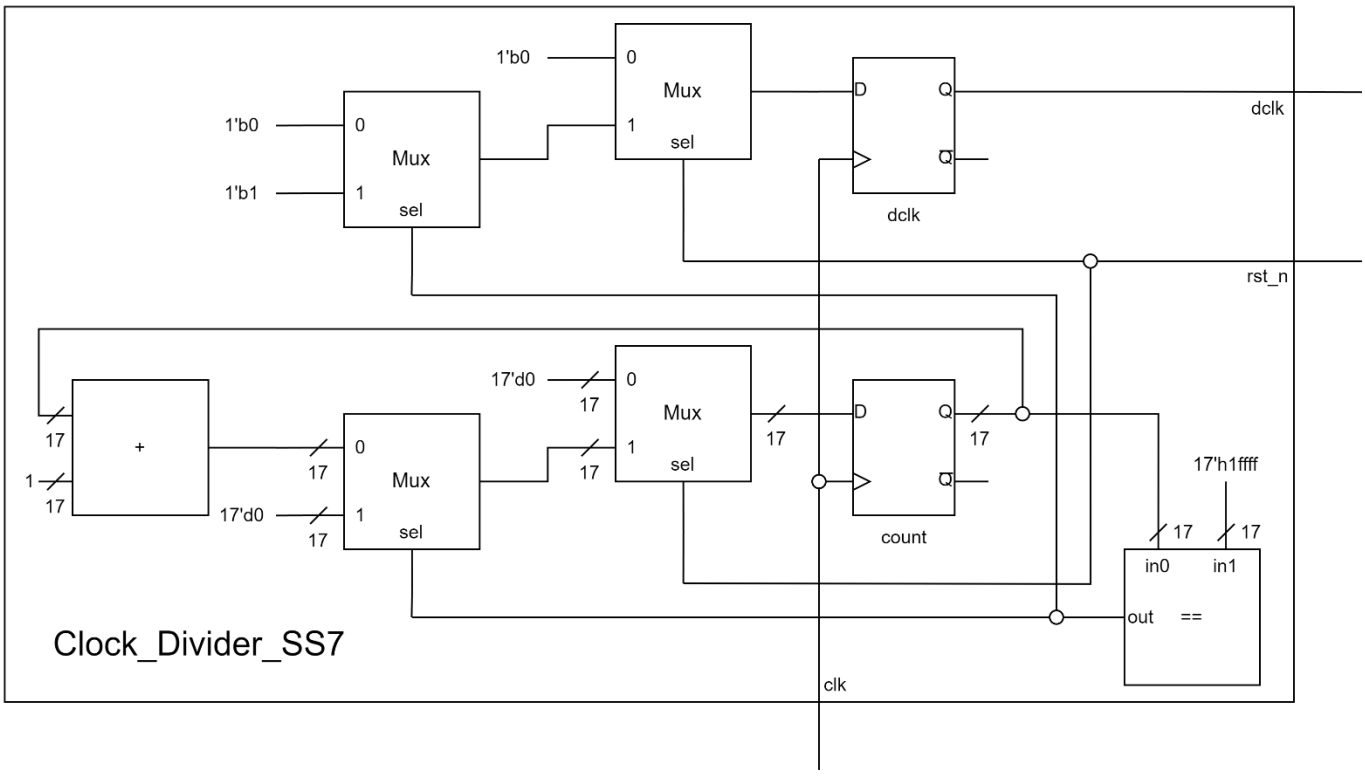
Picture 5.3 Debounce



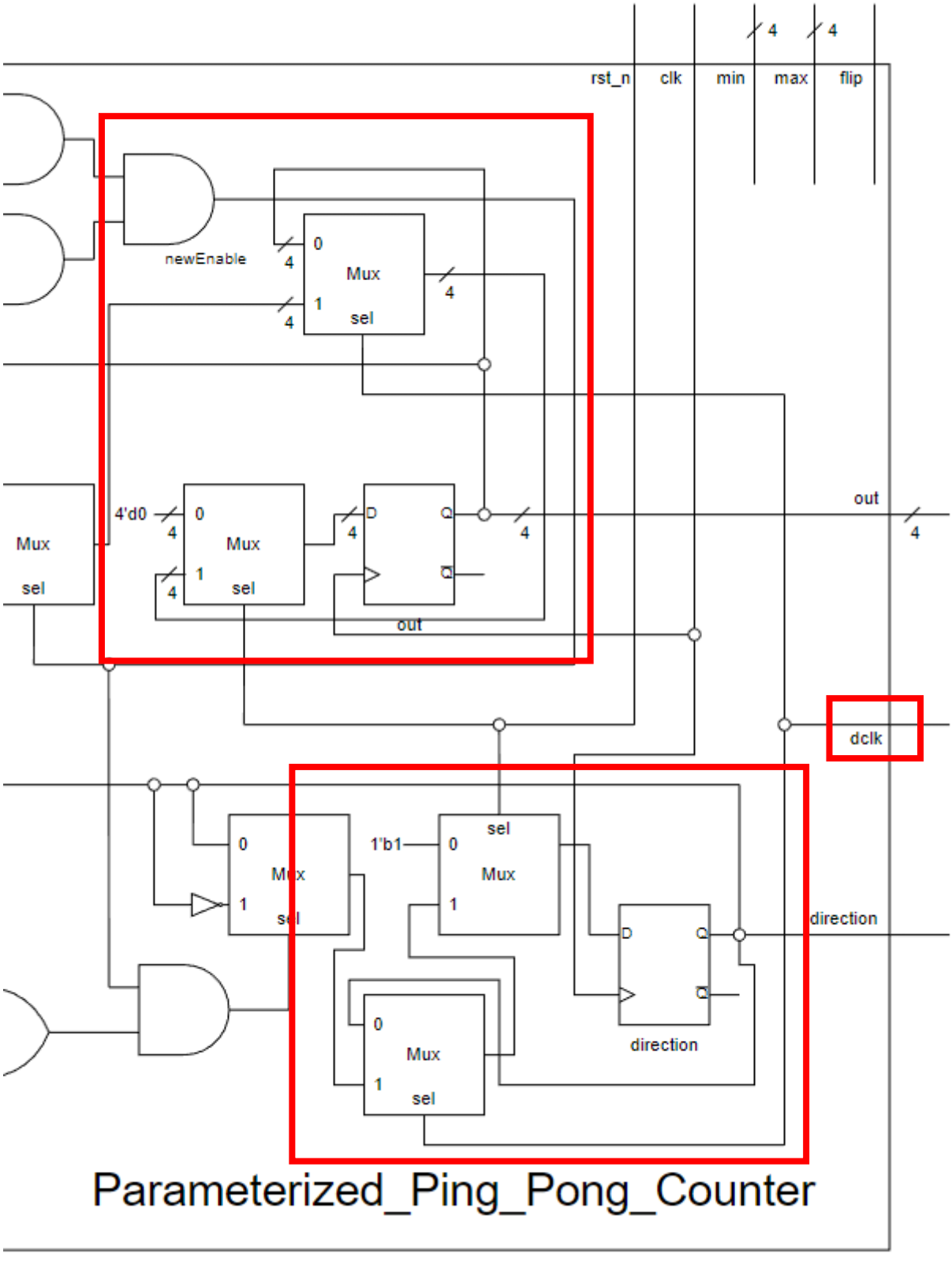
Picture 5.4 SS7



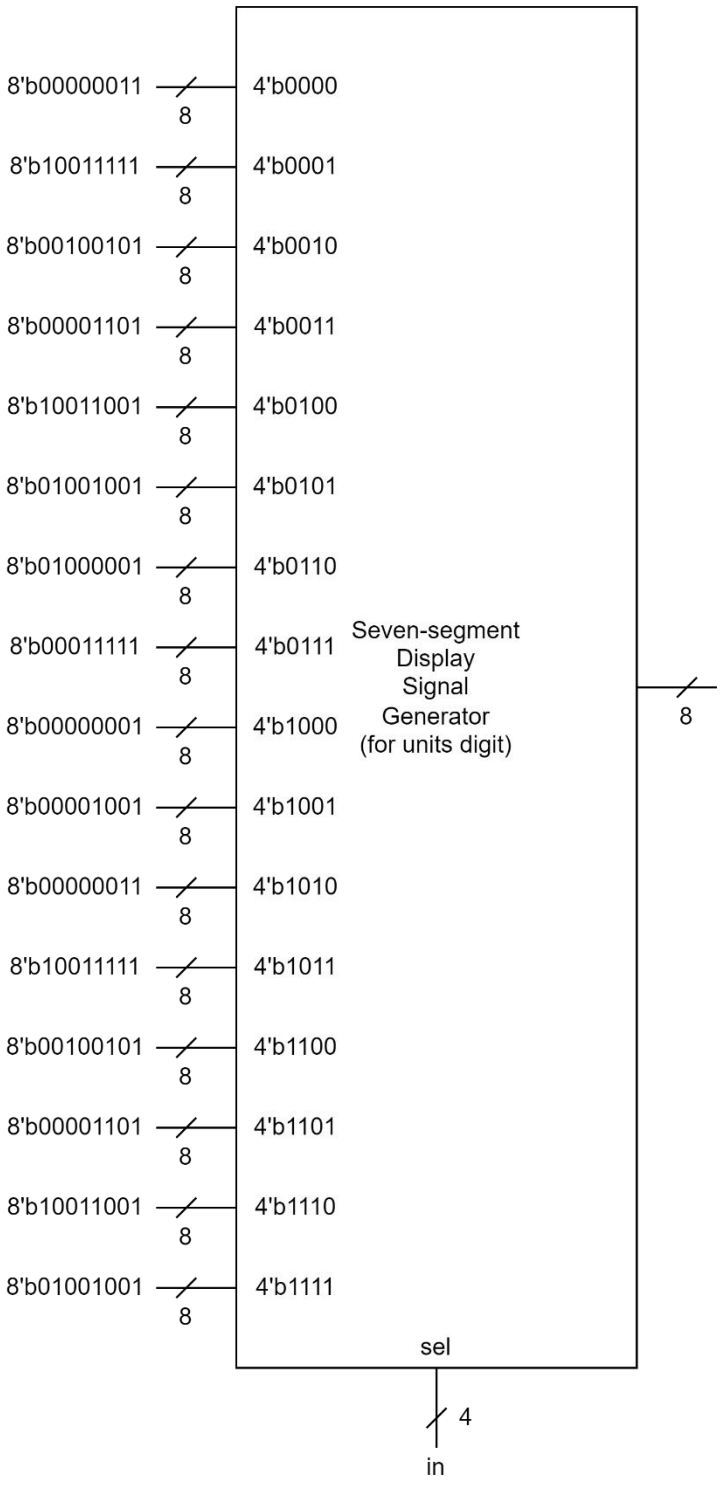
Picture 5.5 Clock_Divider_Circuit

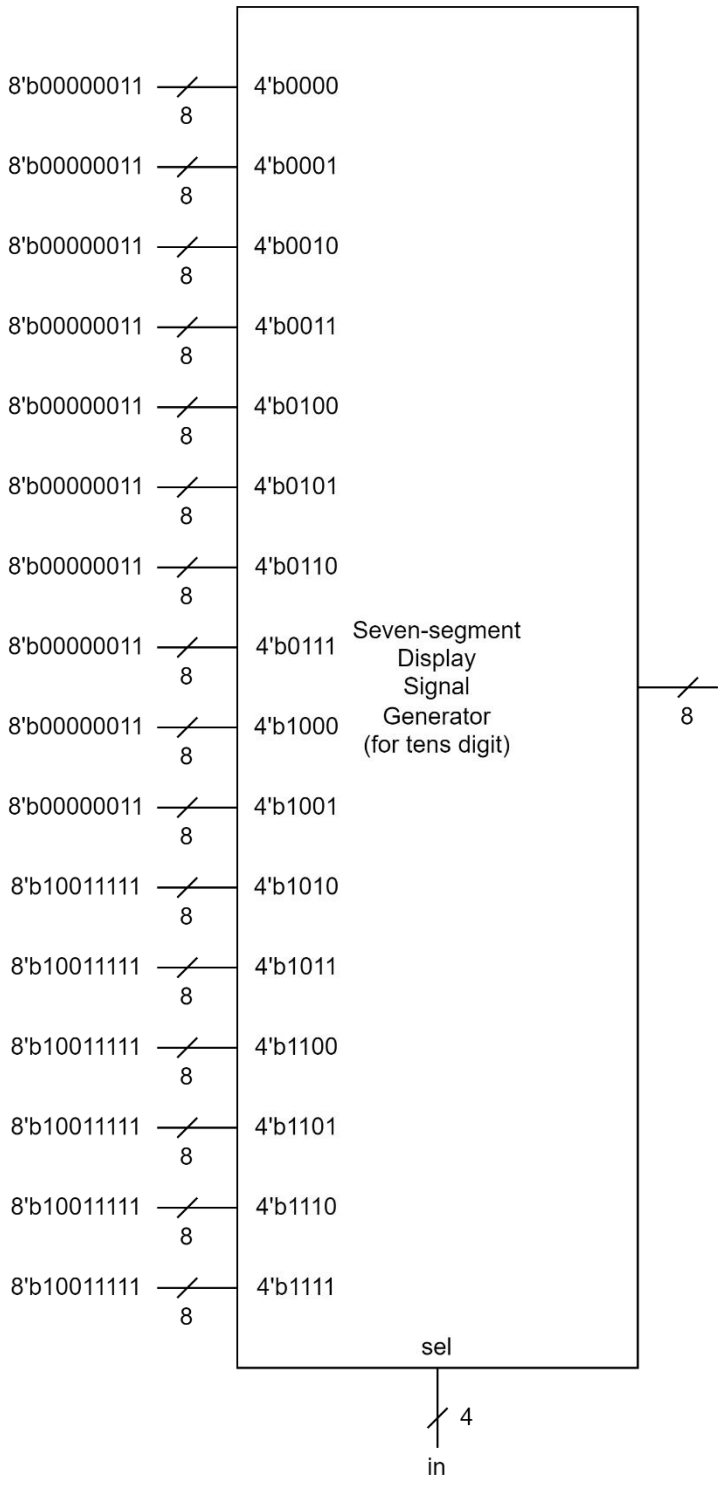


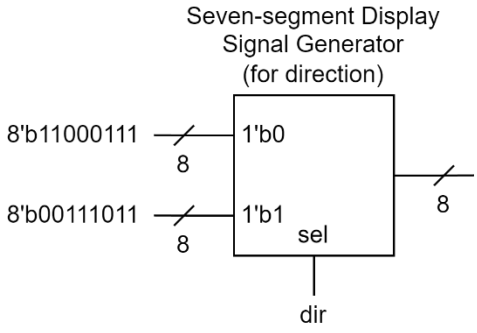
Picture 5.6 Clock_Divider_SS7



Picture 5.7 Parameterized_Ping_Pong_Counter for FPGA



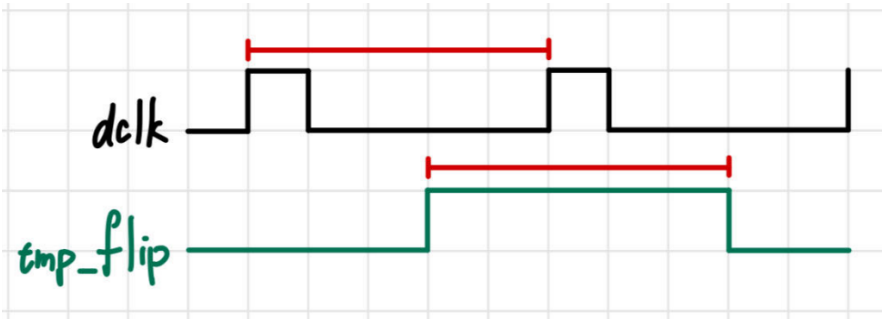




Picture 5.8 Seven-segment Display Signal Generator

B. Explanation

In this problem, there are three clocks with different frequency. **Clock_Divider_Circuit** generates **dclk** for 4-bit Parameterized Ping-Pong Counter. **Clock_Divider_SS7** generates **dclk** for seven-segment display. For **rst_n**, it will reset the counter output to **min** and **direction** to 1'b1 depends on the original clock generated by the FPGA board. Therefore, it seems like reset action happens immediately after the reset button being pushed. However, for **flip**, it is synchronized according to the **dclk**. Once it detects a one-pulse signal of **flip** (**ssg_flip**), **tmp_flip** will be pulled up and keep for a **dclk** cycle of the 4-bit Parameterized Ping-Pong Counter. In this way, I can ensure that it must be caught by positive edge of **dclk** as Picture 5.9.



Picture 5.9

For **Parameterized_Ping_Pong_Counter** module, I change three different things of it. First is that I add an input port called **dclk** to access the **dclk** generated from **Clock_Divider_Circuit**. And in order to synchronize to the **dclk**, I change the circuit as Picture 5.7 shows to make it work properly.

6. What I Have Learned

In this lab, I learned how to write my code in a good code style, separating combinational circuit and sequential circuit. In **Round-Robin FIFO Arbiter**, I had a big obstacle about how to let **valid** work properly. Therefore, I drew out wave form and analyzed it to help me solve this problem. In FPGA question, I had some strange results while I was debugging. The result seemed not related to my code. Instead, it turned out unexpected. I spent two whole days to think which part went wrong. Finally, I found that according to my original code, if **rst_n** was pulled down, **dclk** would be reset and the **tmp_rst_n** would be set at the same time, which was ambiguous to Verilog. (Originally, I designed **tmp_rst_n** as **tmp_flip** as mentioned above) From this event, I will be more careful about this problem and try not make this error again.