Hardware Design Lab 3 Report

Group Number: 30

Members: 112062130 侯佑勳
112062326 孔祥光

Contents:

Design & Testing **1**
1. 4-bit Ping-Pong Counter 1
2. First-In First Out (FIFO) Queue 3
3. Multi-Bank Memory 6
4. Round-Robin FIFO Arbiter 7
5. 4-bit Parameterized Ping-Pong Counter 8

FPGA **10**
What Have We Learned? **12**
Contribution **12**

Design & Testing

1. 4-bit Ping-Pong Counter
   - input: *clk, rst_n, enable*
   - output: *out[3:0]* (the count), *direction* (up = 1, down = 0)

Design

We'll first define a signal *next_count* to determine if the counter will be counting up or down, as depicted in Figure 1-1. The counter will change to counting down if *out* == 15, or to counting up if *out* == 0; otherwise, it remains the previous counting direction.

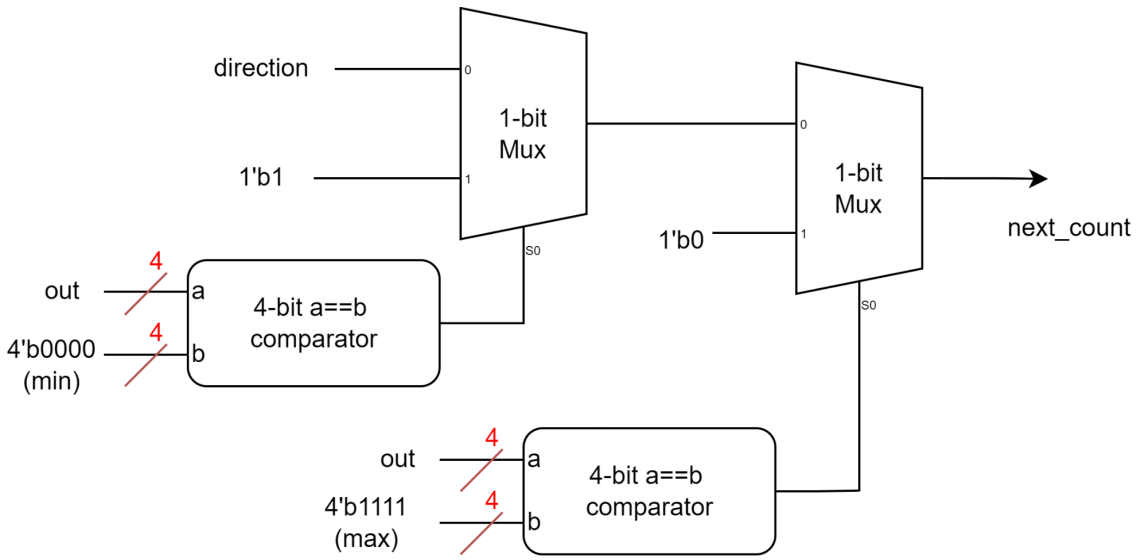

Figure 1-1 Combinational circuit to calculate next_count

Then we use *next_count* as the input *Up* of a 4-bit **up-down counter**, shown in Figure 1-2. When enabled, the counter will count up if *Up*=1 and count down if *Up*=0, due to the toggling-nature of T-Flip-Flop.

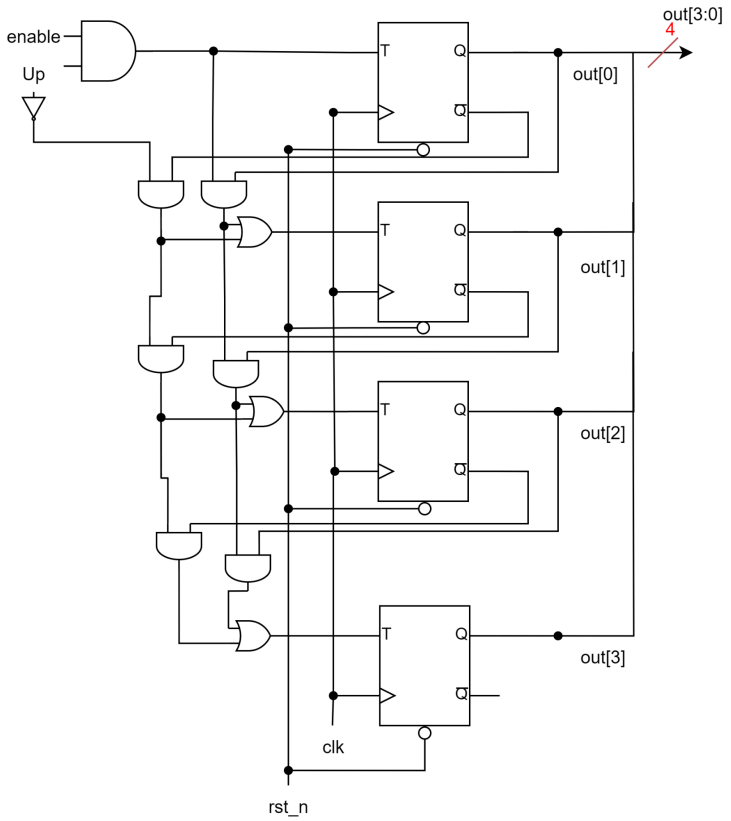

Figure 1-2 4-bit up-down counter, with reset

Finally, we put together both of the submodules to form the whole circuit, as in Figure 1-3.

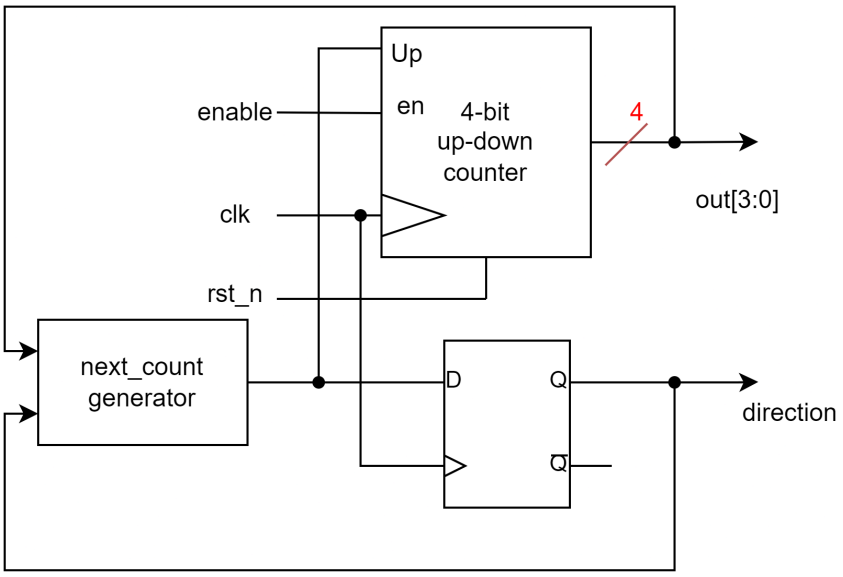

Figure 1-3 4-bit Ping-Pong Counter

Testing

In this testbench, we check the following criteria, as shown in Figure 1-4.

1) whether the “ping-pong” process is achieved.
2) whether the count is holded when disabled.
3) after reset, *out* is set to 0 and *direction* is set to 1

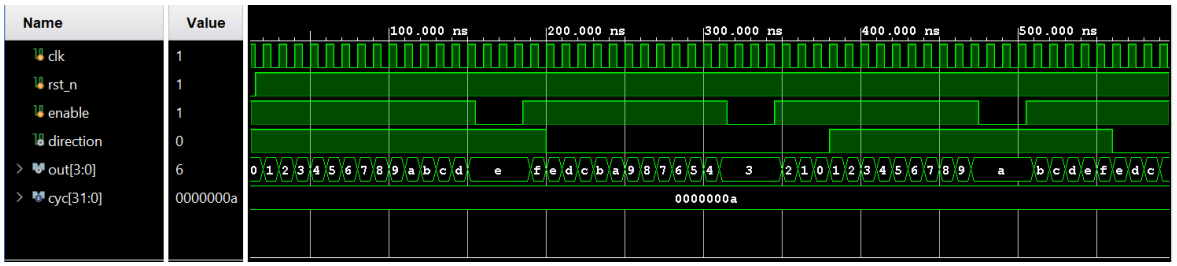

Figure 1-4 4-bit Ping-Pong Counter simulation waveform

2. First-In First Out (FIFO) Queue
   - input: *clk, rst_n, wen, ren, din [7:0]*
   - output: *dout [7:0], error*
   - other variable: [2:0] write_pointer, [2:0] read_poiner, [7:0] Queue [7:0]

   Design

   Below is the overall concept about how to implement a FIFO queue. By using write_pointer and read_pointer to record a queue’s state. Initially, those two pointer’s values are zero. When we write one new thing to the queue’s tail, the write_pointer will increase by one. When we read the oldest data from memory, the read_pointer will increase by one.

   First, we introduce how read_pointer works:
   We use a 3-bit Counter to control the read_pointer’s value. If the ren(read_enable) is true and the queue is not empty (read_pointer does not equal to (catch up) write_pointer), we can enable this counter.

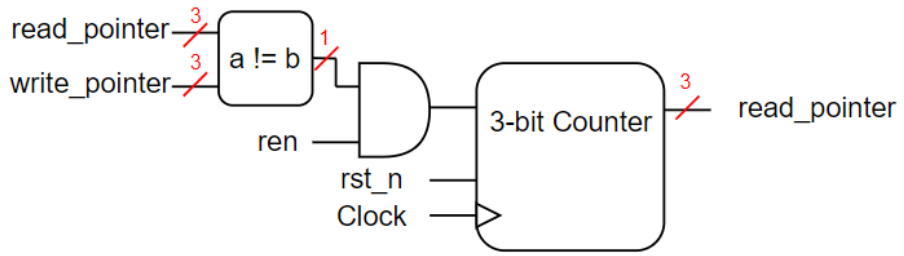

Figure 2-1 read_pointer Generator

Second, we introduce how write_pointer works:
We also use a 3-bit Counter to control the read_pointer's value. If the wen(write_enable) is true and the queue is not full (the write_pointer does not equal to read_pointer minus 1) and the ren equals zero (because if ren is one, we need to read instead of write), we can enable this counter.

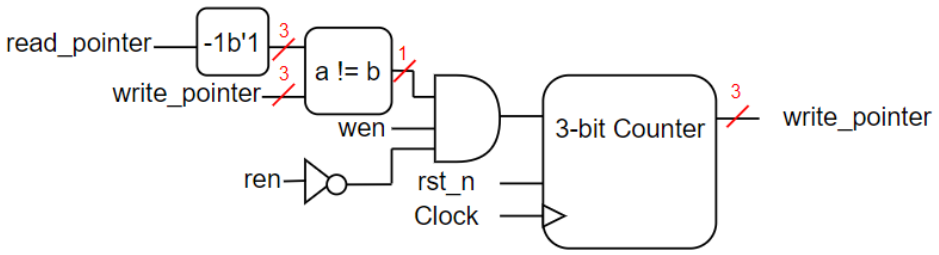

Figure 2-2 write_pointer Generator

Third, we introduce how error signal produced:
There are two situations where the error signal will be one.

1. The queue is already full, but we keep writing data.
2. The queue is already empty, but we keep reading it.

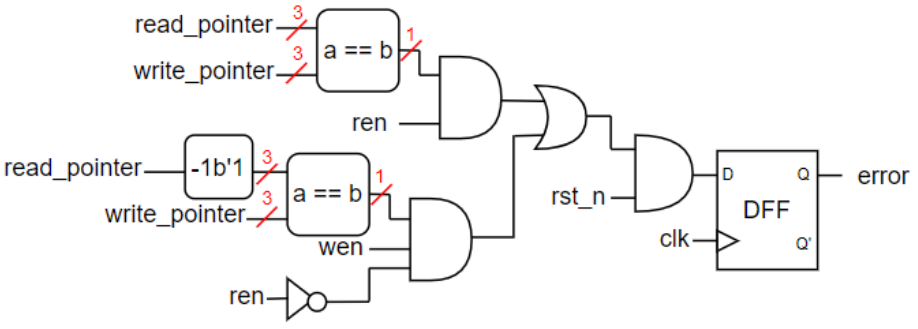

Figure 2-3 error Generator

Fourth, we introduce how to write data to memory:
The condition is the same as the one of write_pointer increasing.
In addition, we add write_pointer as an address to write din.

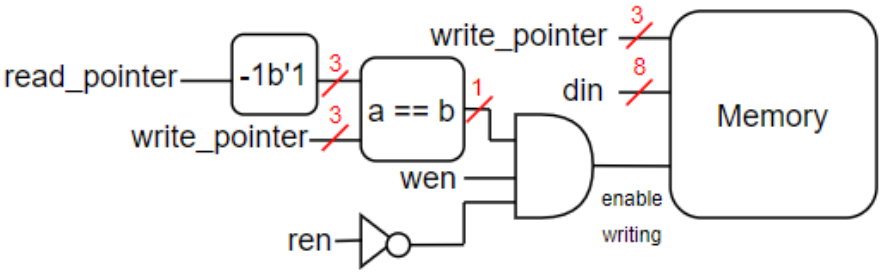

Figure 2-4 Writing Function

Fifth, we introduce how to read a data from memory:
The condition is also the same as the one of reading_pointer increasing. In addition, we add a rst_n to let dout be 0 if rst_n is 0.

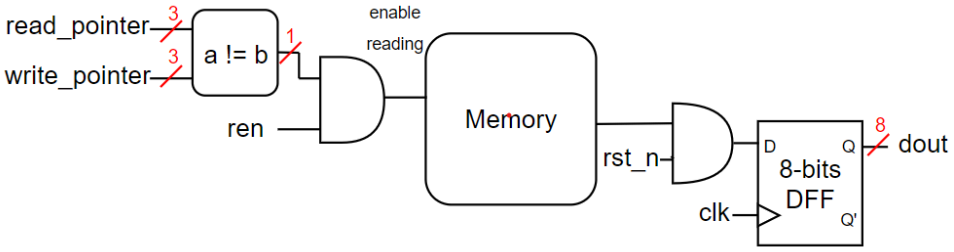

Figure 2-5 Reading Function

Testing

In this testbench, we test these situation:

1) The read and write functions can work, and the read function has higher priority.
2) If the queue is empty && keep reading or full && keep writing, the error will become one.
3) While rst_n == 0, the error and dout become zero and memory clear

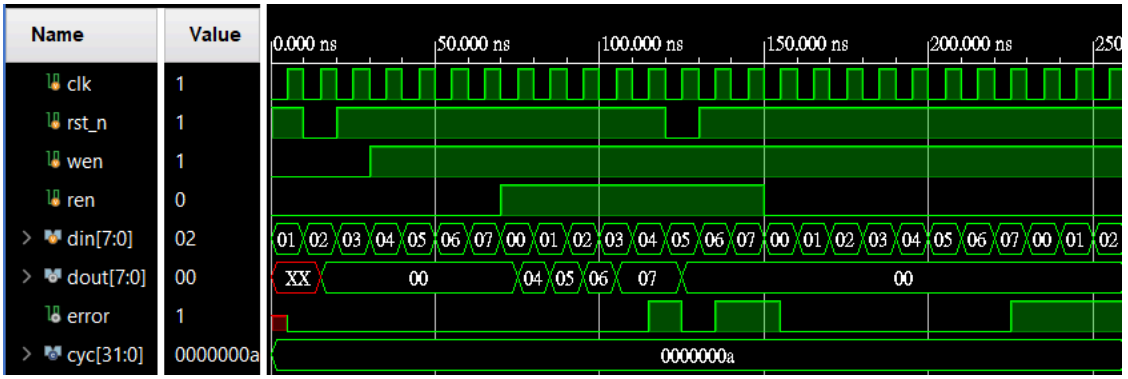

Figure 2-6 FIFO simulation waveform

3. Multi-Bank Memory
   - input: *clk, ren, wen, waddr[10:0], raddr[10:0], din[7:0]*
   - output: *dout[7:0]*

Design

Because the Multi-Bank Memory is a **hierarchical structure**, we divide it into three elements: Multi-Bank Memory, Bank Memory, Sub Bank Memory. and three memories' output names respectively are *dout, tdout, fdout*.

First, we introduce the Sub Bank Memory(the smallest one):
We take Sub_Bank_Memory3 for example.
The address is determined by if *raddr[8:7]* is equal to 2'b11. If it's true, then we input the reading address, or input the writing address.
In addition, whether to write or read is also some part determined by if *raddr[8:7]* is equal to 2'b11.The read and write functions are similar to FIFO without the full and empty situation. Moreover, our design fits that **read has higher priority than write**.

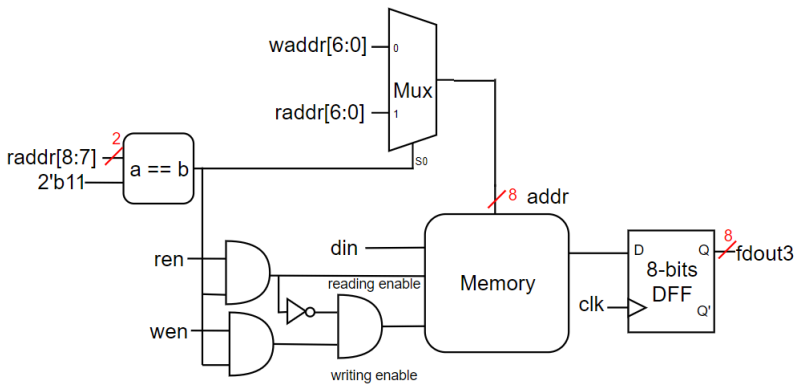

Figure 3-1 Sub_Bank_Memory

Second, we introduce the Bank Memory(the middle one):
In each Bank Memory, there are four Sub Bank Memories. And we use a 4-1 Mux to connect them. The structure of Bank Memory is very similar to Sub_Bank_Memory with some slight adjustments including: changing *raddr[8:7]* to *raddr[10:9]* and changing *fdout* to *tdout*.

Finally, we introduce the Multi_Bank_Memory:
Each four Bank Memories are connected with one Multi_Bank_Memory. And we connect them with 4-1 MUX and a DFF to ensure the final output will change with the clock's positive edge.

Testing
We consider those four situation
ren = 0, wen = 1, write in
ren = 1, wen = 0, read out
ren = 1, wen = 1, read and write different memory
ren = 1, wen = 1, read and write same memory

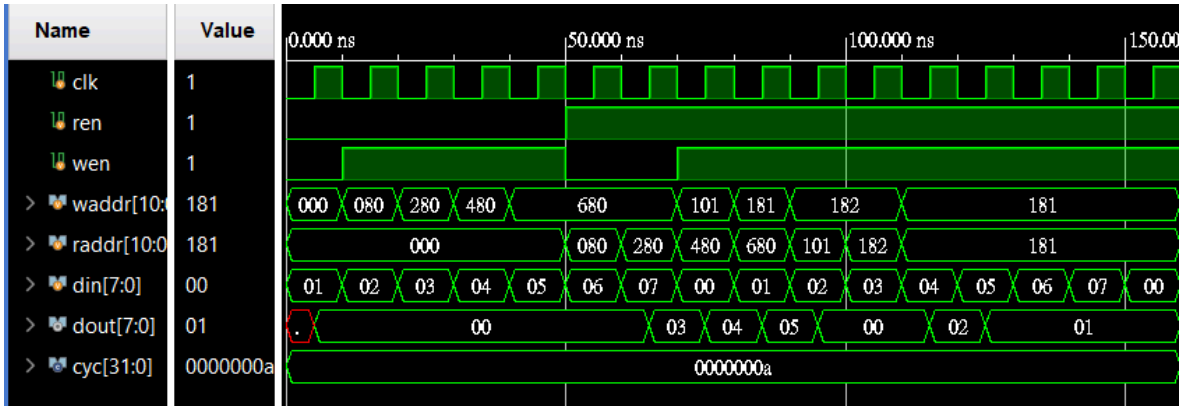

Figure 3-2 Mult_Bank_Memory simulation waveform

4. Round-Robin FIFO Arbiter

Design
😬😬😬
Testing
😢😢😢

5. 4-bit Parameterized Ping-Pong Counter
   - input: *clk, rst_n, enable, flip, min[3:0], max[3:0]*
   - output: *out[3:0], direction*

Design

Using a similar strategy in Q1, we define a signal *next_count*, to determine the output *direction,* and whether we count-up or count-down. Shown in Figure 5-1 is the module to accomplish that.

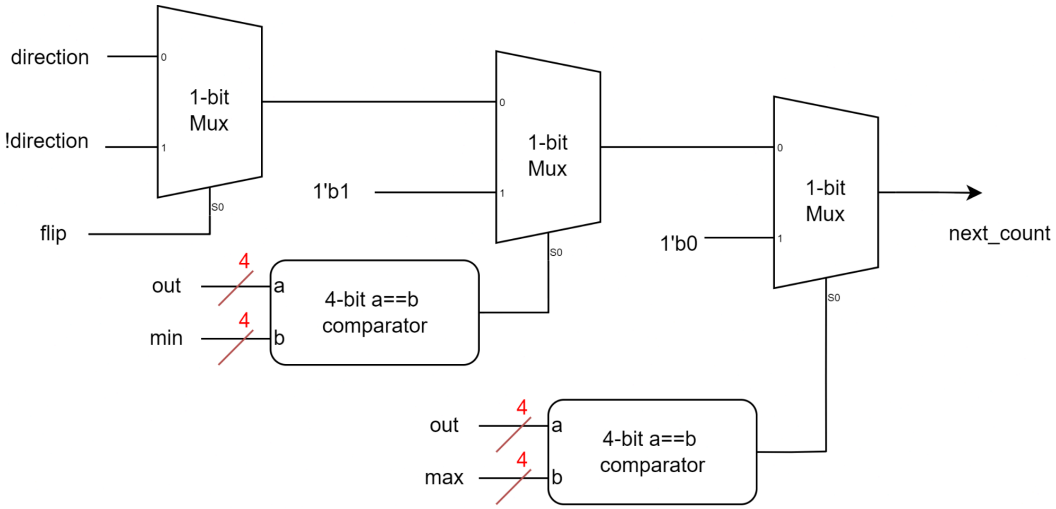

Figure 5-1 Comb. Circuit for generating *next_count*, with *flip*

The rest is similar to Q1, but we have to make some slight adjustments, adding a **reset with load** feature to our up-down counter. Also, only enable the counter when the range is valid ($max > min$) or when the count is in-range ($max \geq out \geq min$). To achieve greater-or-equal to, we'll just negate less-than.

The final circuit will look like this:

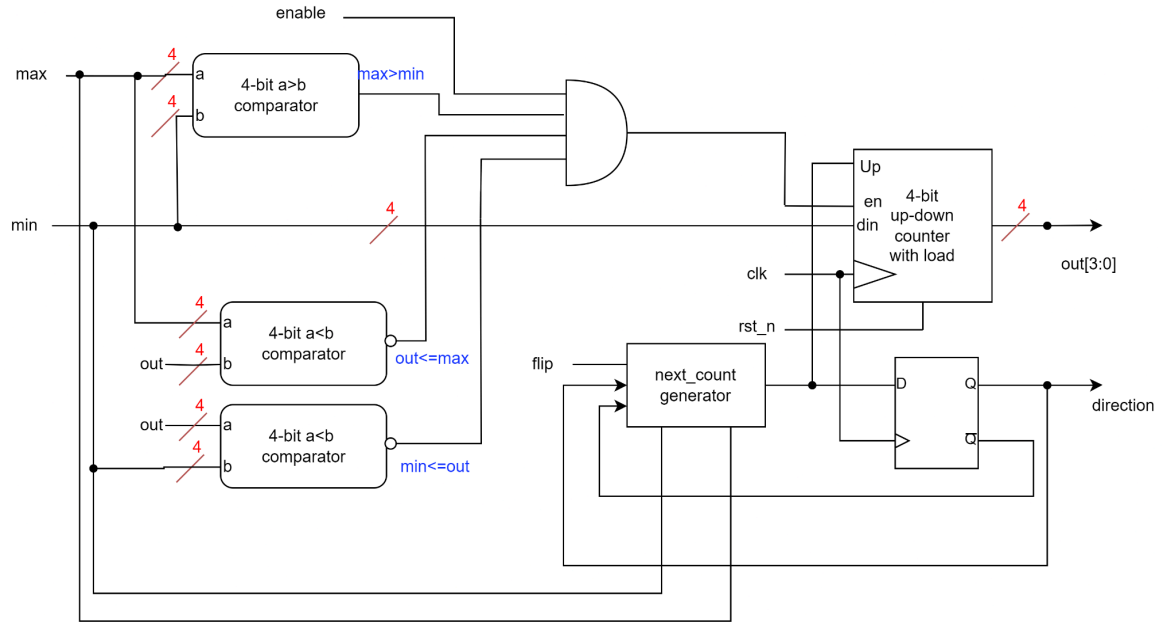

Figure 5-2 4-bit Parameterized Ping-Pong Counter

Testing

In this testbench, we check the following criteria, as shown in Figure 5-3.

1) the counter counts up and down when enabled & in range
2) it holds the value when disabled, out-of-range, or invalid range ($max > min$)
3) after reset, *out* is set to *min* and *direction* is set to 1
4) when *flip* is high and the count is in range, flip the counting direction.

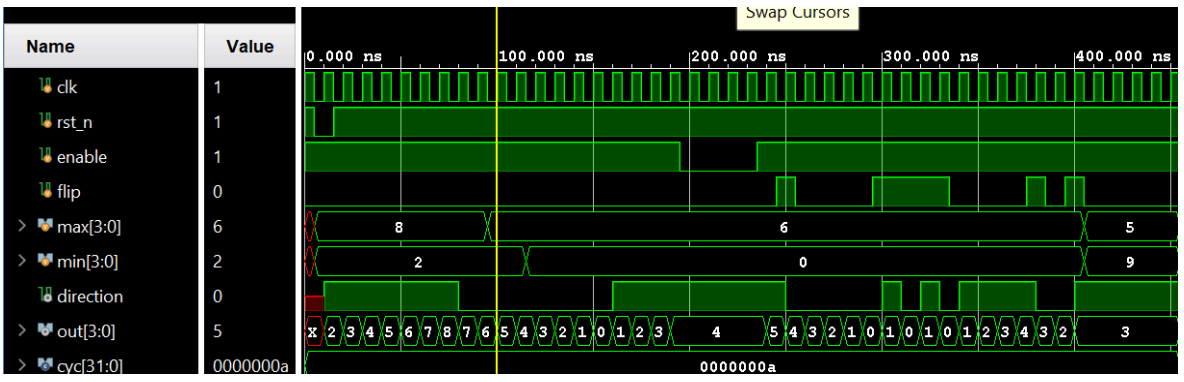

Figure 5-3 PPP counter simulation waveform

FPGA

4-bit Parameterized Ping-Pong Counter on FPGA

- input
  - switch: *enable, max[3:0], min[3:0]*
  - button: *flip, reset*
  - clock: *clk* (100MHz)
- output
  - 7-seg display: *an[3:0], outLED[6:0]*
- internal signals:
  - clock_2_17, clock_2_26
  - processed inputs
  - from counter: *out[3:0], direction*
  - processed outputs

The approach is rather simple, seen in Figure 6-1, we process the inputs from the FPGA, feed it to the PPP-counter module from Q5 (with some slight variation), then process the output of that before displaying it on the FPGA,

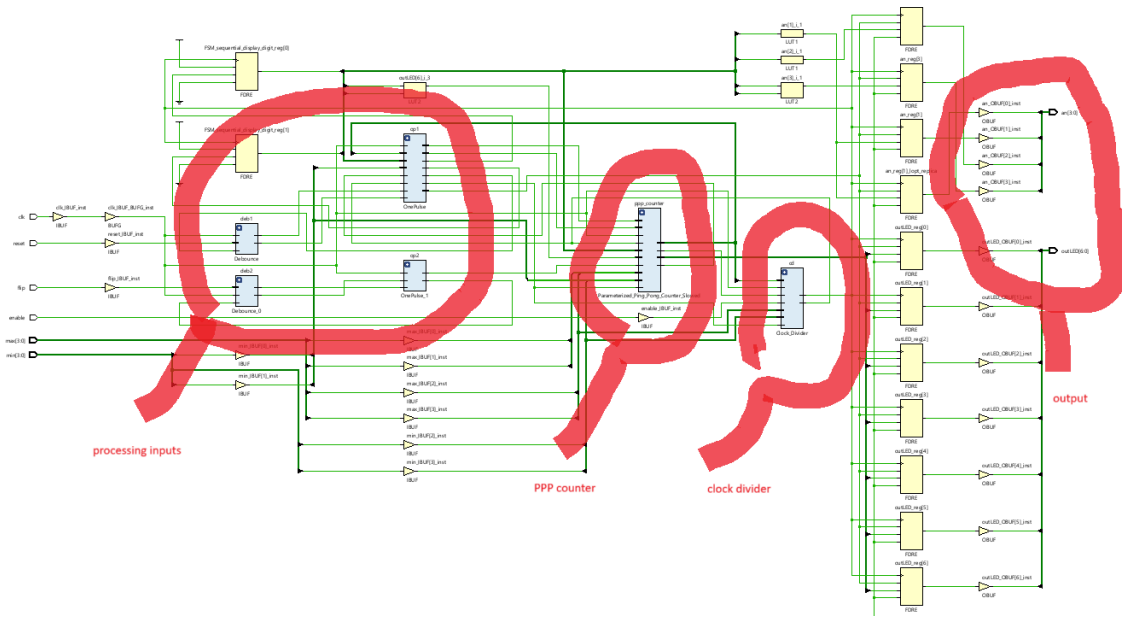

Figure 6-1 overview of the whole circuit

First, notice that the *reset* signal generated from pushing button will be high, but all of our modules are designed with **active-low reset**, so we'll keep in mind to invert it to *rst_n*.

Second, all the inputs from buttons need to be **debounced and onepulsed**, so that they'll be stable (no glitches), high for only one clock-cycle, and can be utilized by our PPP-counter module. See Figure 6-2 and 6-3 below.

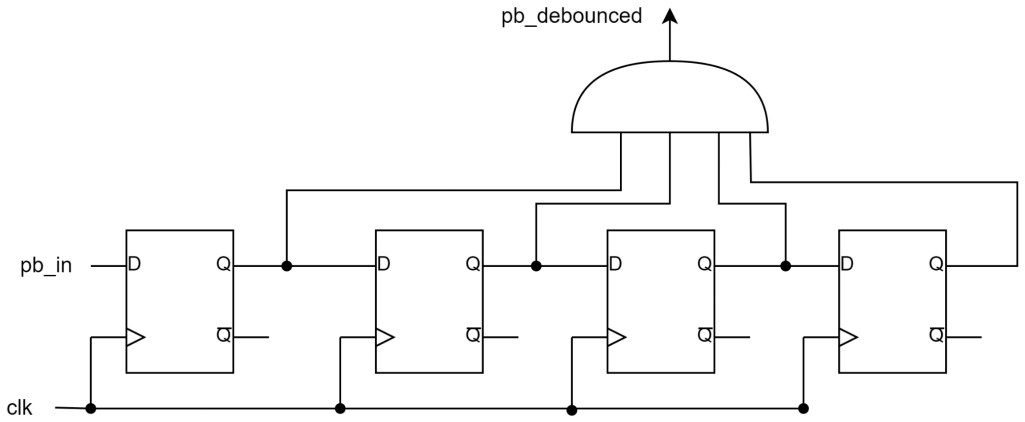

Figure 6-2 Debounce module for button input

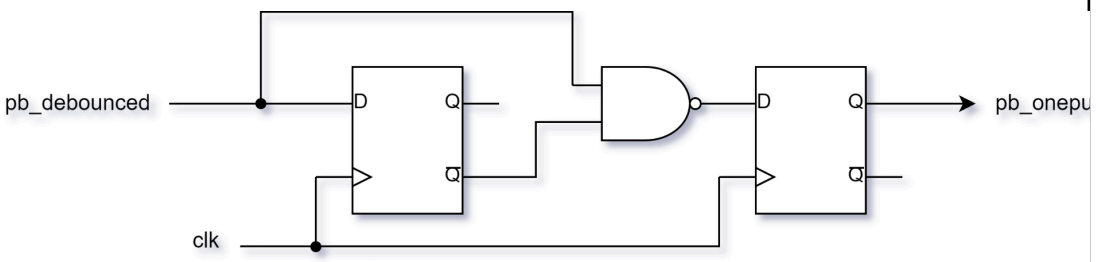

Figure 6-3 Onepulse module for button input

Third, to concurrently display 4 digits on the 7-segment display with different contents, we'll need to do time-multiplexing to refresh *an* and *outLED* signals around every 1ms~16ms. Also, the counting should be slowed to a rate observable by the naked eye. Therefore, we will use a clock divider to generate **two slowed clock rate**: some 1.3KHz & some 2.6Hz, by dividing the given 100Mhz master clock with 2^{17} & 2^{26}, using counters of 17-bit and 26-bit, respectively.

What Have We Learned?

112062130 侯佑勳:
Through this lab, we tried lots of sequential and combinational circuit design which is far different from gate level design. I think the most interesting one is Multi-Bank Memory. We need to think about how to initiate each submodule under the top module and let the correct signal transport. We also need to start the Lab earlier because this time we are stuck for a long time because of some tiny bugs, and don't have time for the last question.

112062326 孔祥光:
During this lab, I've learned two more techniques for modeling a digital circuit: dataflow modeling & behavioral modeling, and got familiar with best practices of using the always-block. In using combinational & sequential circuit design for problem solving, it is quite challenging to picture the final circuit, but once the logic is simplified and the design is complete, it all worked like a charm. Plus, I've also got to know how to use the 7-segment display in a concurrent manner. Though we didn't complete all of the problem set, I've certainly gained a lot.

Contribution

112062130 侯佑勳
First-In First Out (FIFO) Queue (+tb)
Multi-Bank Memory (+tb)

112062326 孔祥光
4-bit Ping-Pong Counter (+tb)
4-bit Parameterized Ping-Pong Counter (+tb)
4-bit Parameterized Ping-Pong Counter on FPGA