Hardware Design

Lab 3 Report

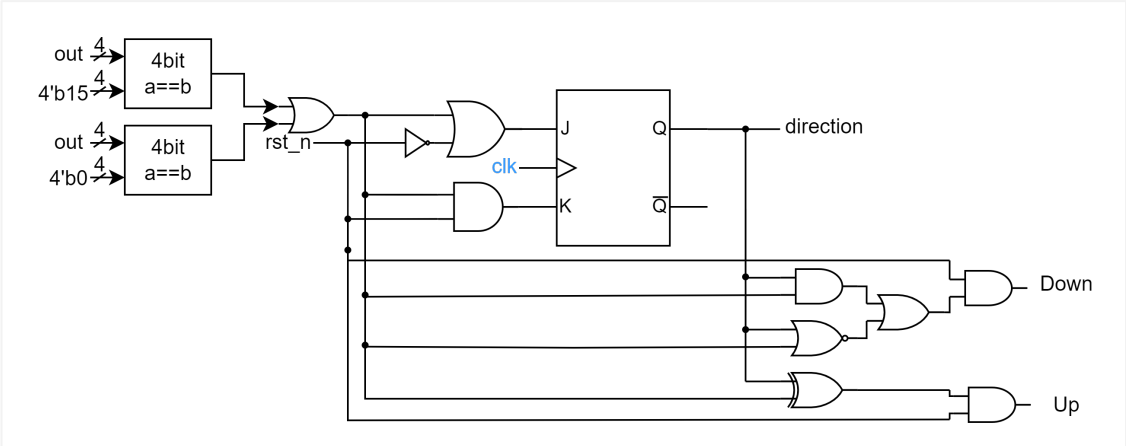
Sequential Circuits

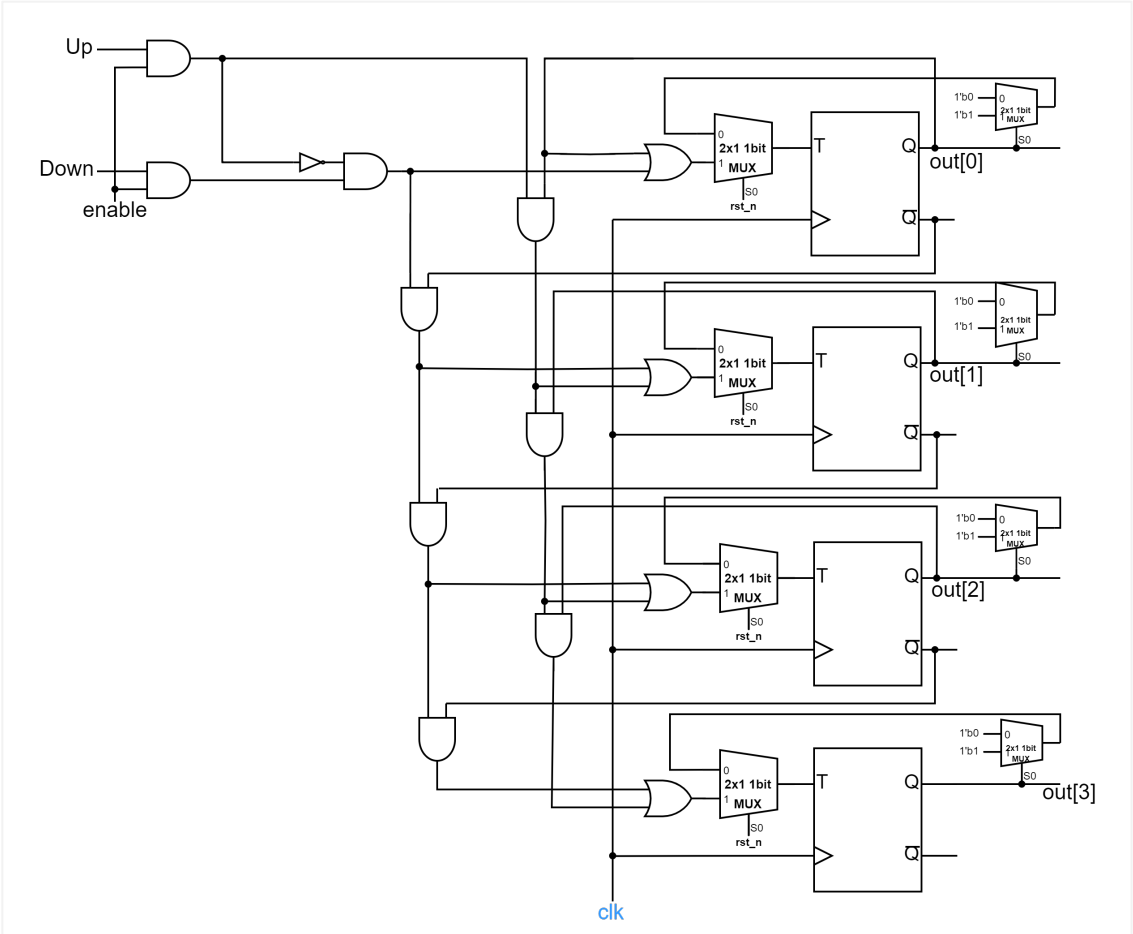
Team 01

112062122 謝佳晉　112062144 范升維

Table of Contents:

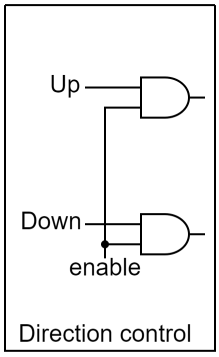
Advanced Q1. 4-bit Ping-Pong Counter





- 3 inputs: *clk, rst_n, enable*
- 2 outputs: *direction, out[3:0]*

In this question, we introduce an Up-down counter to implement the function of changing counting direction.



(Upper-output)

(Lower-output)

The leftmost part of the Up-down counter is the **Direction control**.

Input-Output is shown in the table below. (input, output)

Up	0	0	1	1	0	0	1	1
Down	0	1	0	1	0	1	0	1
Enable	0	0	0	0	1	1	1	1
Lower-output	0	0	0	0	0	1	0	1 (won't happen)
Upper-output	0	0	0	0	0	0	1	1 (won't happen)

The values of Up and Down can be traced back to Figure 1.1. We get their Boolean expressions by **drawing K-map** on **rst_n, direction, and border**(out==15 or out==0).

Direction	border	rst_n	Up	Down
0	0	0	0	0
0	0	1	0	1
0	1	0	0	0
0	1	1	1	0
1	0	0	0	0
1	0	1	1	0
1	1	0	0	0
1	1	1	0	1

The Boolean expressions is shown below:

Up = rst_n & ((!direction & border) | (direction & border)) = rst_n & (direction ^ border)

Down = rst_n & ((direction & border) | (!direction & !border))

For the value of direction, we use JK flip flop. Similarly, we **draw K-map** on **rst_n and border**.

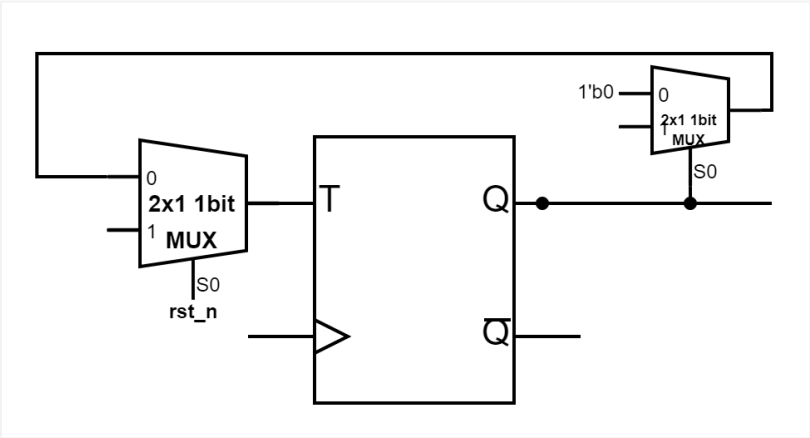
Border	rst_n	J of JKFF	K of JKFF
0	0	1	0
0	1	0	0
1	0	1	0
1	1	1	1

The Boolean expressions is shown below:

J of JKFF = !rst_n | border.

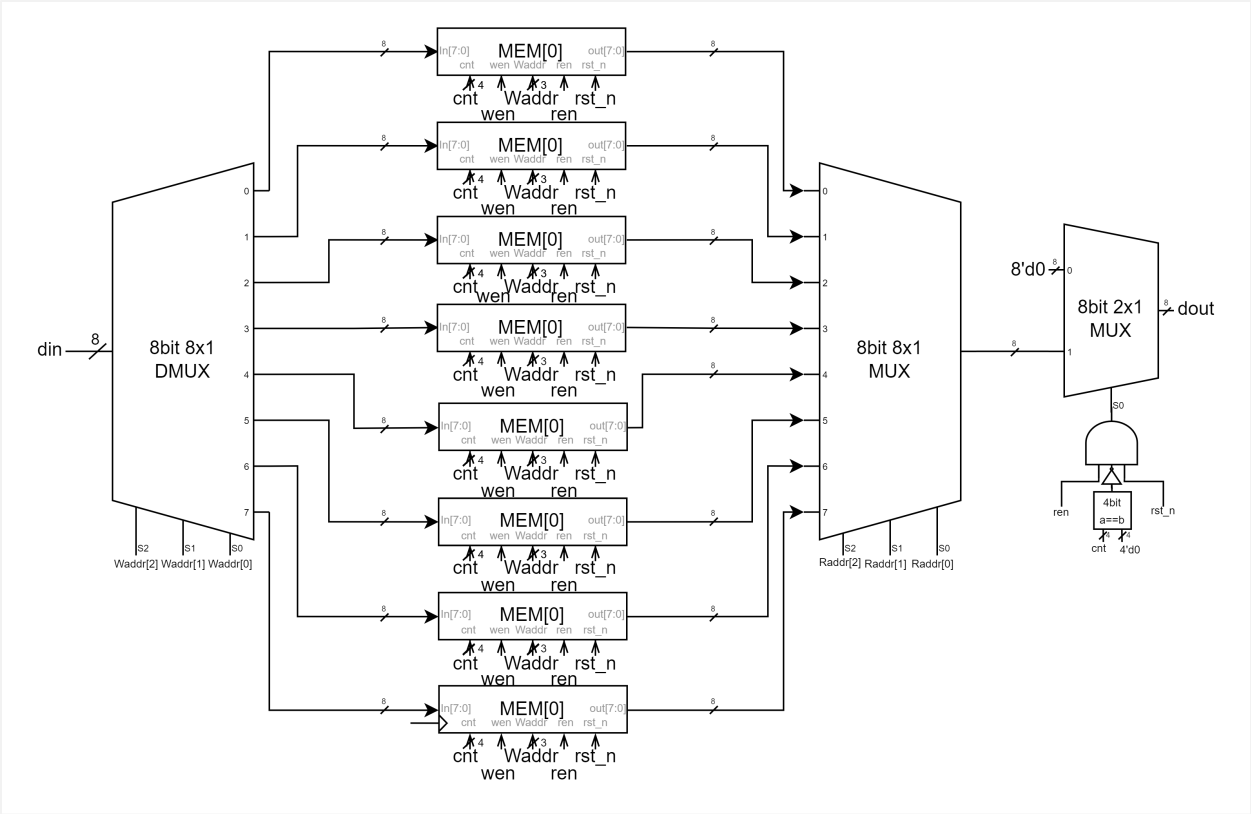
K of JKFF = rst_n & border.

For *out[3:0]*, because it will be set to *4'd0* if *rst_n* == 0, a MUX takes on *rst_n* before TFF input is needed. We want each bit of *out[3:0]* to be *0*, but we can't just simply input *0* into the MUX because of the feature of TFF(Toggle when input *1*, Nothing change when input *0*). This is where another MUX taking on each bit of *out[3:0]* comes in.

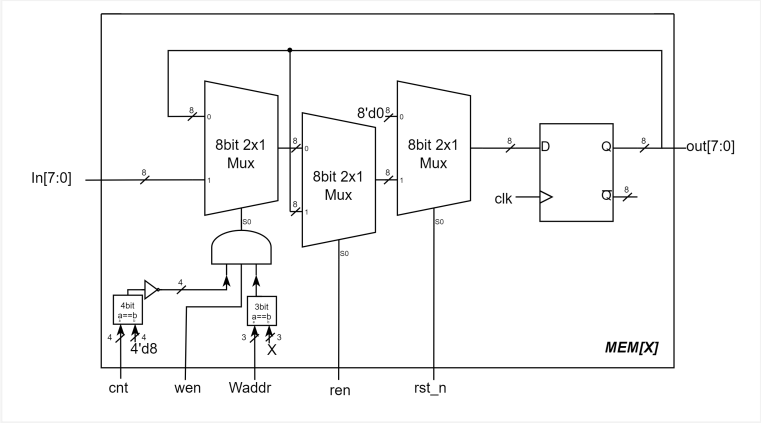


If the bit is *1'b1*, *1'b1* will be set to output in order to toggle the TFF value. If the bit is *1'b0*, *1'b0* will be set to output, and the TFF keep its value. In this way, every bit will all be *1'b0* when *rst_n == 0*.

Advanced Q2. First-In First-Out (FIFO) Queue



▲ Figure 2.1: First-In First-Out Queue



▲ Figure 2.2: Memory Unit

- 3 inputs: *clk, rst_n, din[7:0]*
- 2 outputs: *dout[7:0], error*
- 3 internal signals: *In[7:0], out[7:0], cnt[3:0], Raddr[2:0], Waddr[2:0]*

In order to make *din* and *out* to the correct port, we use **8x1 DMUX** and **8x1 MUX** and take on *Waddr* and *Raddr* respectively as the selection.

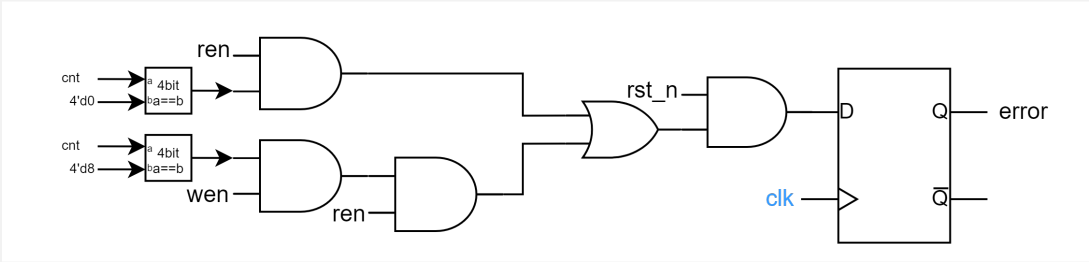
To construct a memory unit, we combine three **2x1 MUX**es and a **DFF**(Figure 2.2). At the first MUX, we check *cnt*'s value to see if the memory is full, check *Waddr*'s value to see if it matches the memory address *X*, and check whether *wen* is equal to *1'b1*. If these cases are all correct, then second MUX will take the din as one of the input.

At the second MUX, we check if *ren* is equal to *1'b1*. If not, then third MUX will take the original value in the memory unit because *ren* is prior to *wen*.

At the third MUX, we check *rst_n*'s value. If it's equal to *1'b0*, it will put *8'd0* to the DFF because of the priority of *rst_n*.

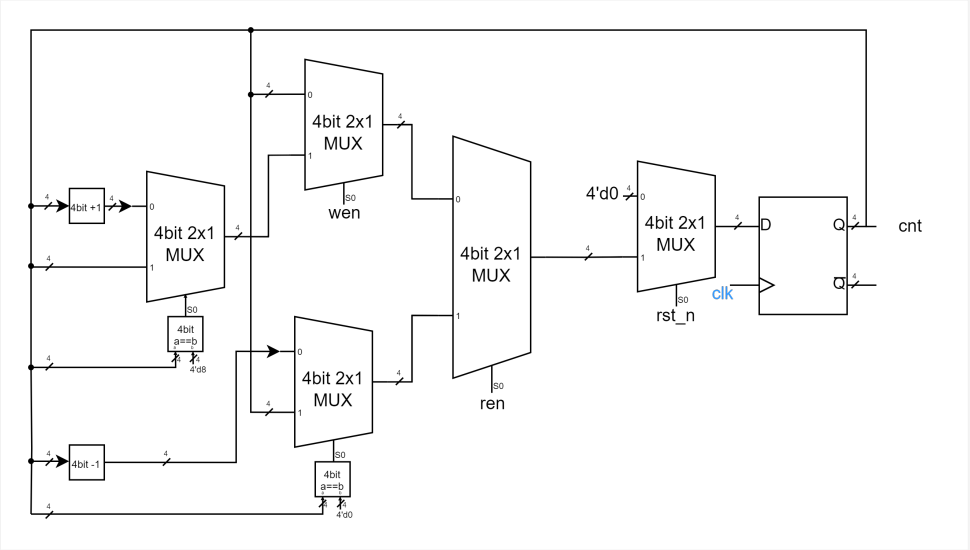
Eventually, out will take on the value at the next clock's posive edge.

At the right most of Figure 2.1, because *dout* only output the value of memory when re*n==1'b1 && cnt!=1'b0 && rst_n==1'b1*, we design a MUX take on this condition and output *8'd0* if the condition isn't met.

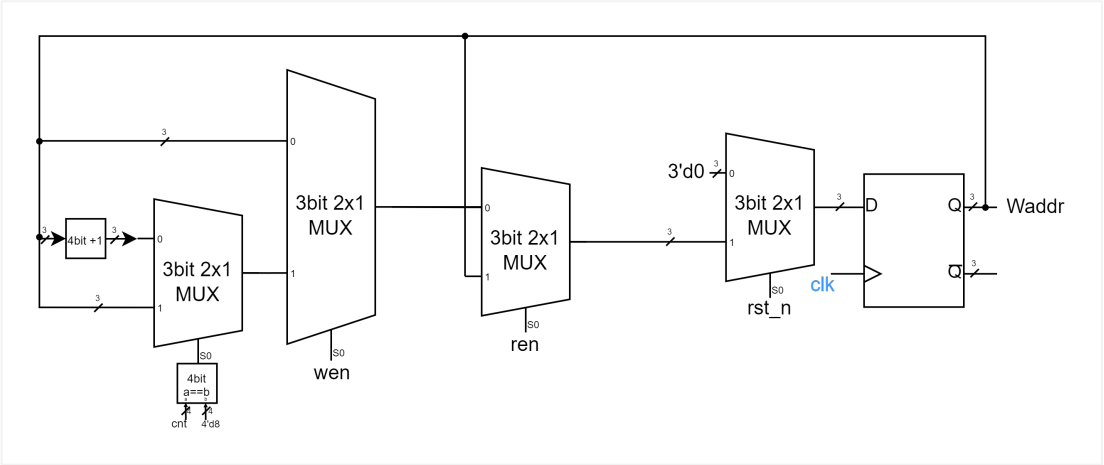


▲ Figure 2.3: Component – *error*

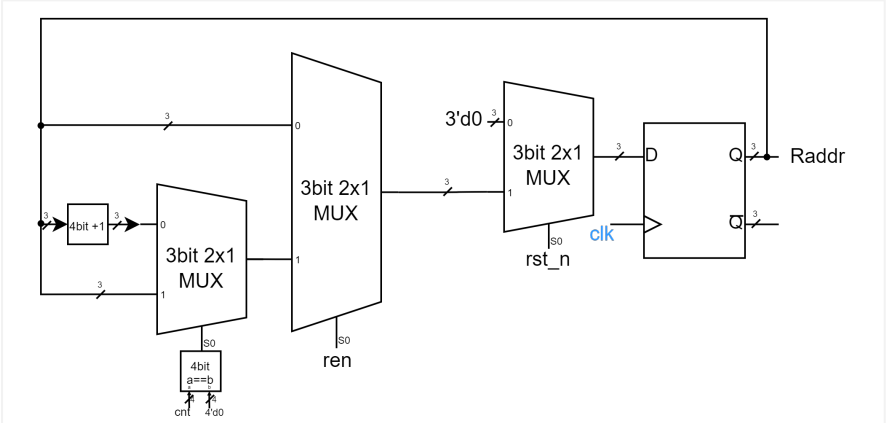
For error signal, because error will change to *1'b0* when *rst_n==1'b0*, we put *rst_n* to the most front and-gate. If *rst_n==1'b1*, we have to consider some more signals. Because the priority order is *ren > wen*, we need to put *ren* after *wen* to see if we can do the write operation. Moreover, border cases are also considered, so we have the border check and the operation which the border is related to linked with an AND gate. Eventually, we can have the *error* signal indicate correctly.



▲ Figure 2.4: Component – *cnt*



▲ Figure 2.5: Component – *Waddr*

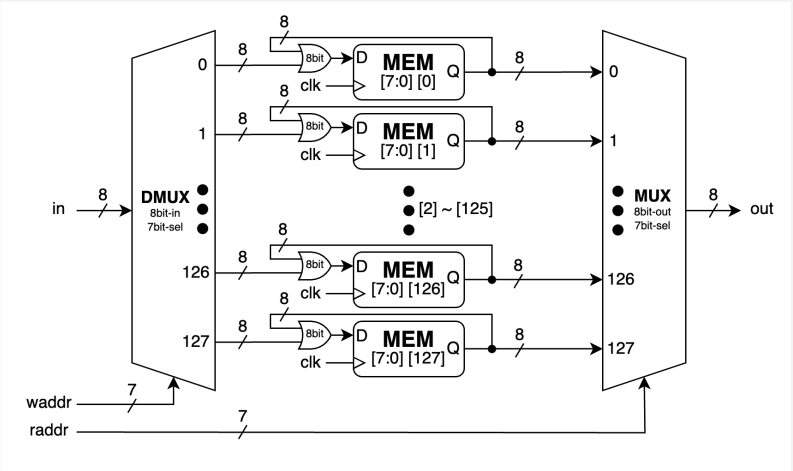


▲ Figure 2.6: Component – *Waddr*

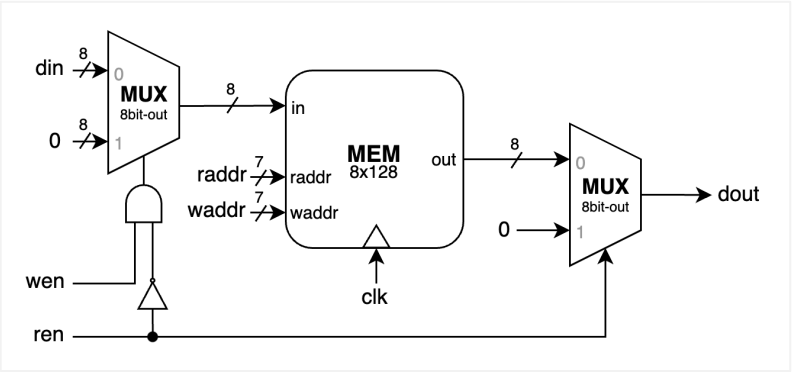
For *cnt*, *Waddr*, and *Raddr*, we use the same method, **MUX by MUX**, to change our if-else part into circuits. The order of MUXes shows the priority order of selection signals. The mechanism of our design is that whenever a write or read operation is done, either *Waddr* or *Raddr* will move forward to the next address(0->1->2->…->6->7->0->…) according to the operation type. Likewise, *cnt* will also +1 or -1 according to the operation type.

Advanced Q3. Multi-Bank Memory

In Advanced Question 3, we are required to construct a multi-bank memory system consisting of 4 **Memory Banks**. Each bank comprises 4 **Memory Units**, and each Memory Unit is an 8-bit x 128 memory module, which we previously implemented in Basic Question 2.



▲ Figure 3.1: MEM 8x128



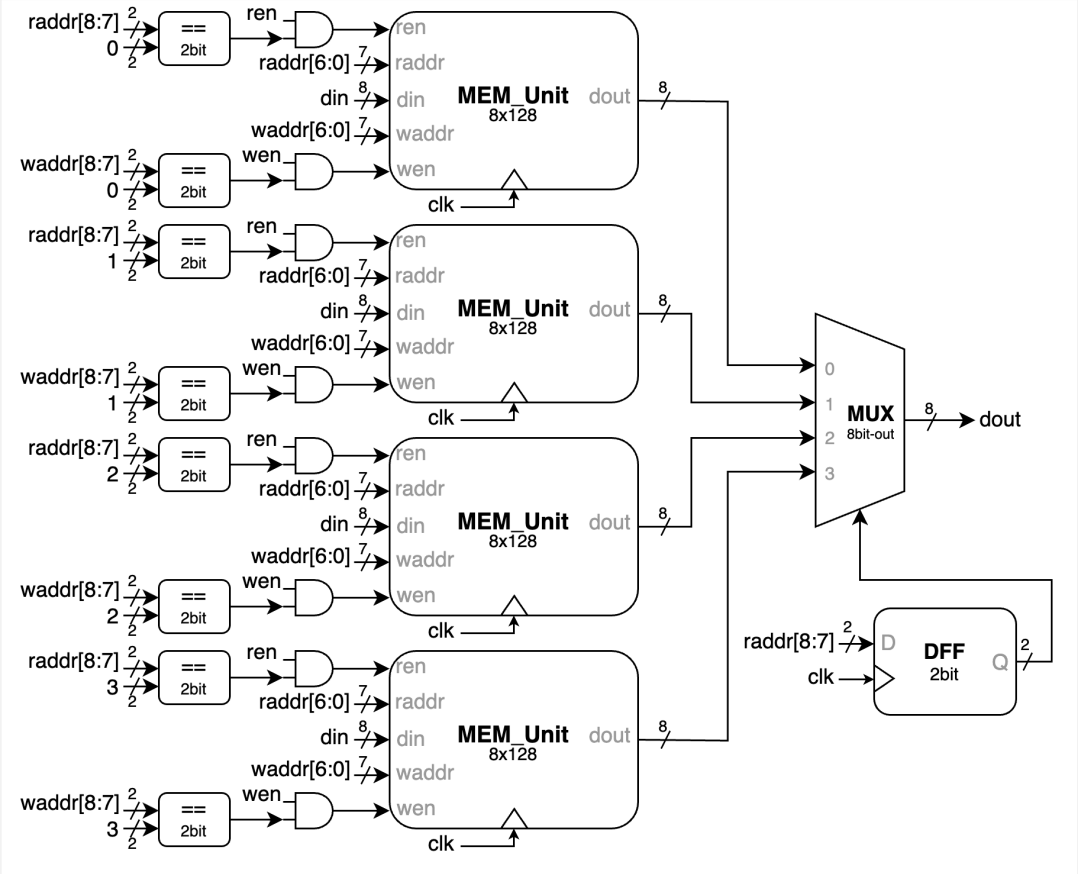
▲ Figure 3.2: Memory Unit (8bit x 128 memory module)

- 6 inputs: *clk, ren, wen, waddr[6:0], raddr[6:0], din[7:0]*
- 1 output: *dout[7:1]*

Figure 3.2 illustrates our implemented **Memory Unit**, which is an 8-bit x 128 memory module. This unit incorporates the **MEM 8x128** component shown in Figure 3.1. The MEM 8x128 is designed to store the input data (*din*) at the memory location specified by the write address (*waddr*). Each memory location within the MEM consists of a synchronized 8-bit

register array, implemented with D Flip Flop. The output (*dout*) is determined by the value stored in the memory location designated by the read address (*raddr*).

Figure 3.2 illustrates the implementation that fulfills the specified requirements of the problem. The circuit is configured to output *dout[7:0]* only when the read enable signal (*ren*) is asserted to *1'b1*; otherwise, it outputs *8'b0*. Furthermore, the circuit permits data input operations under the condition that the input data (*din*) is stored in the memory location designated by the write address (*waddr*) exclusively when the write enable signal (*wen*) is asserted to *1'b1* and, concurrently, the read enable signal (*ren*) is de-asserted to *1'b0*. This control scheme ensures proper read and write operations while preventing potential conflicts between simultaneous read and write attempts.

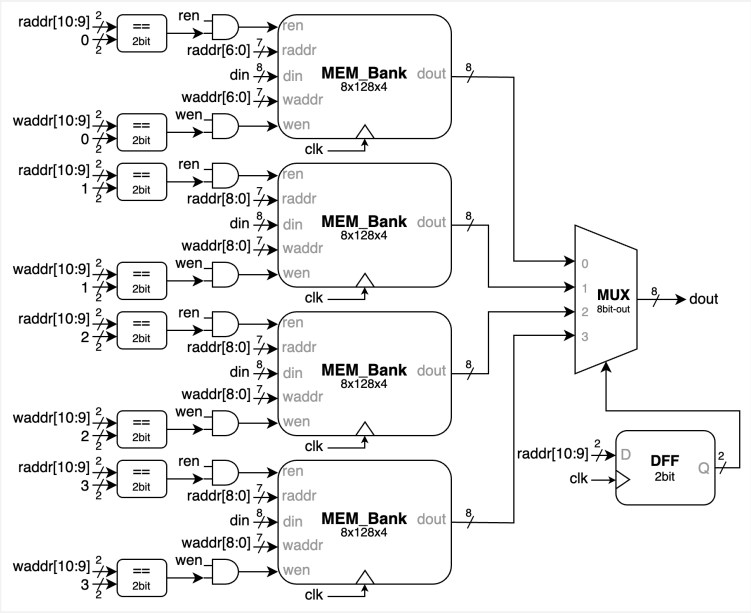


▲ Figure 3.3: Memory Bank

- 6 inputs: *clk, ren, wen, waddr[8:0], raddr[8:0], din[7:0]*
- 1 output: *dout[7:1]*

Figure 3.3 depicts the circuit diagram of the **Memory Bank**, which integrates four Memory Units. The *waddr* and *raddr* signals have been expanded by two bits to specify the target Memory Unit. Consequently, the write enable (*wen*) and read enable (*ren*) signals for each Memory Unit are activated (set to *1'b1*) only when this module's *wen*/*ren* is *1'b1* and the two most significant bits of *waddr*/*raddr* correspond to that particular Memory Unit.

The output (*dout*) of this module utilizes a multiplexer (**MUX**) to select the *dout* from the Memory Unit indicated by the two most significant bits of *raddr*. It is important to note that the *select* signal for this MUX is not directly connected to *raddr*. Instead, it passes through a D flip-flop (**DFF**). This design choice ensures that fluctuations in the *raddr* signal between clock cycles do not inadvertently alter the Memory Unit connected to the module's *dout*, thereby maintaining signal integrity and preventing glitches in the output.

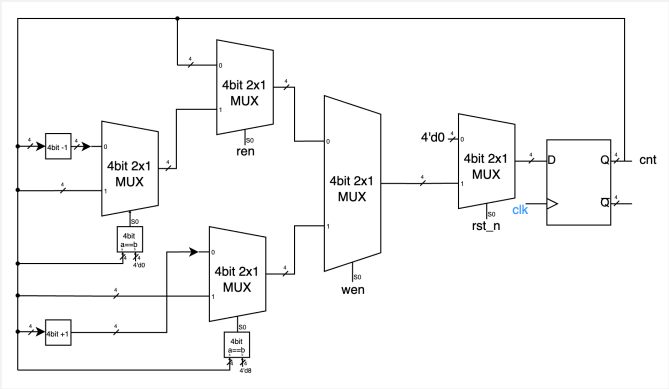


▲ Figure 3.4: Multi-Bank Memory

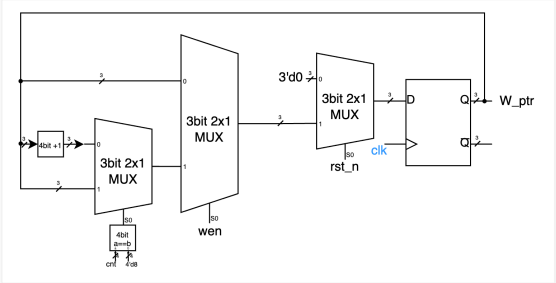
Figure 3.4 illustrates the **Multi-Bank Memory** module, which represents the final implementation of our design. The structure of this module bears a strong resemblance to that of the Memory Bank connected to four Memory Units, as previously discussed. The primary distinction lies in the further extension of *waddr* and *raddr* signals by an additional two bits. Given the structural similarities, an additional description of the design is deemed unnecessary.

Advanced Q4. Round-Robin FIFO Arbiter

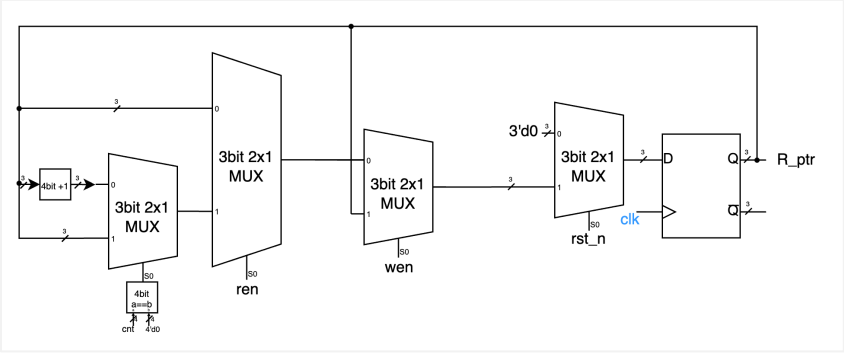
To create the Round-Robin FIFO Arbiter as specified in the problem, we first need to make some minor modifications to the FIFO from Advanced Q2, as follows:



▲ Figure 4.1: Component – *cnt*

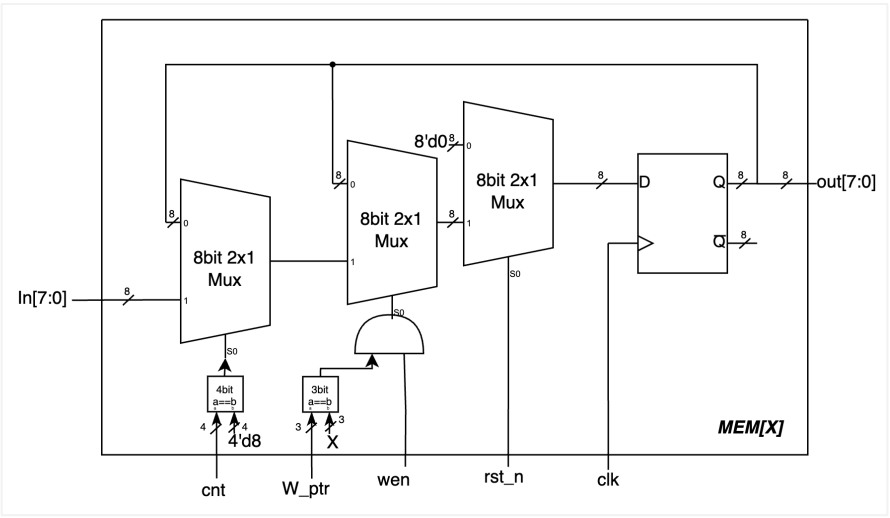


▲ Figure 4.2: Component – *W_ptr*



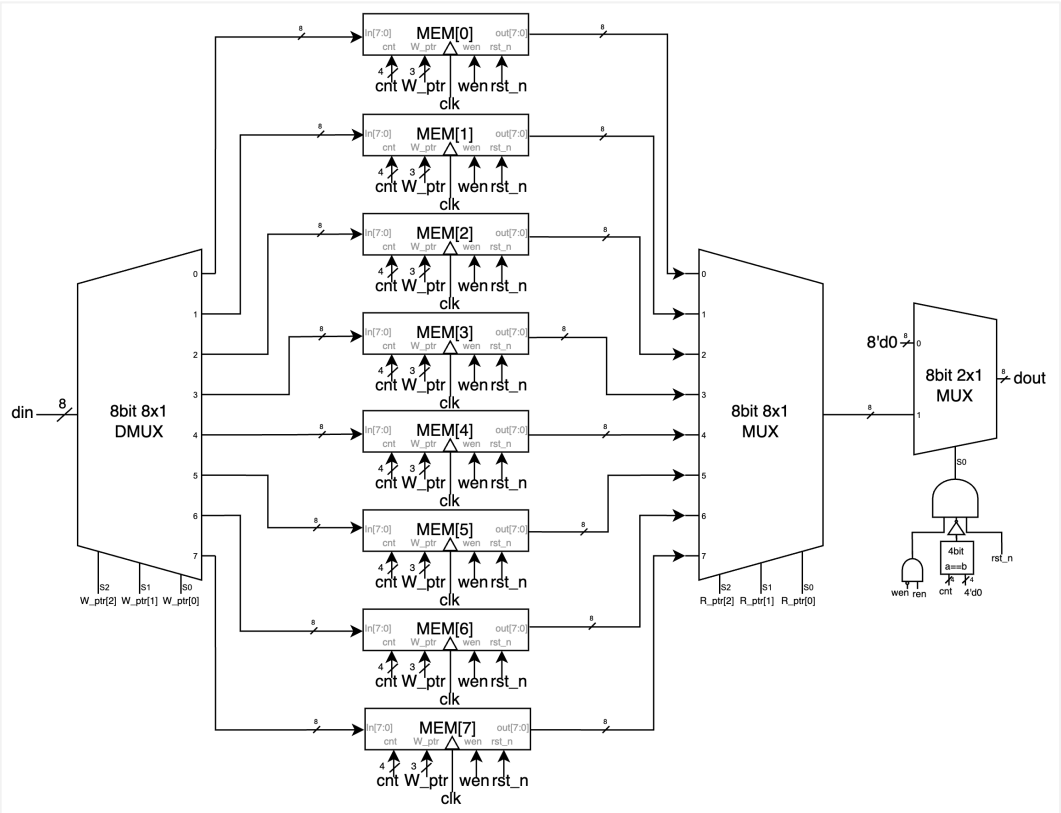
◀ Figure 4.3: Component – *R_ptr*

In the above figures, Figures 4.1, 4.2, and 4.3 are slightly modified versions of Figures 2.4, 2.6, and 2.5 from Advanced Q2, respectively. The main modification is adjusting the order of wen and ren as MUX selectors. This is because in the second question, when both are enabled, ren takes precedence, while in the fourth question, wen takes precedence, hence this adjustment.



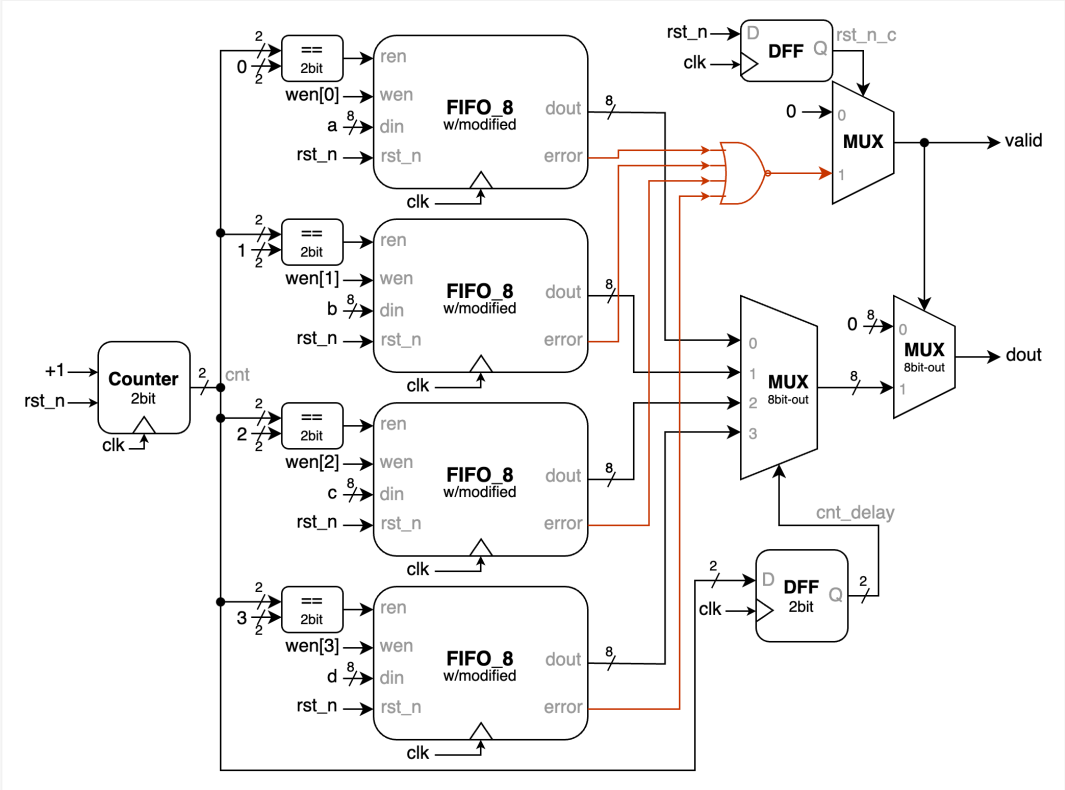
▲ Figure 4.4: Component – MEM

Figure 4.4 above shows the MEM cell in the FIFO, similar to Figure 2.2 from the second question. The only difference is the removal of the MUX that used ren as a selector, for the same reason as mentioned in the previous paragraph. The other functions remain the same.



▲ Figure 4.5: FIFO w/modified

In Figure 4.5 above, we have finally completed the **FIFO w/modified**. It is similar to Figure 2.1 from Q2, with the following changes: the input to **MEM** no longer includes *ren*, and the final **2x1 MUX**'s selector has been changed from just *ren* to *!wen & ren*. The reason for these modifications is the same as mentioned previously - in this question, *wen* has a higher priority than *ren*.



▲ Figure 4.6: Round Robin FIFO Arbiter

We can now discuss the main content of this question. On the far left is a 2-bit counter (*cnt*) that increments with each positive clock edge, serving as the *ren* condition for the four central FIFOs. The four FIFOs in the middle have their corresponding inputs connected as shown. Their *ren* signals are activated only when the *cnt* signal equals their respective number, at which point they output and pop (i.e., advance the FIFO's *R_ptr* by one position).

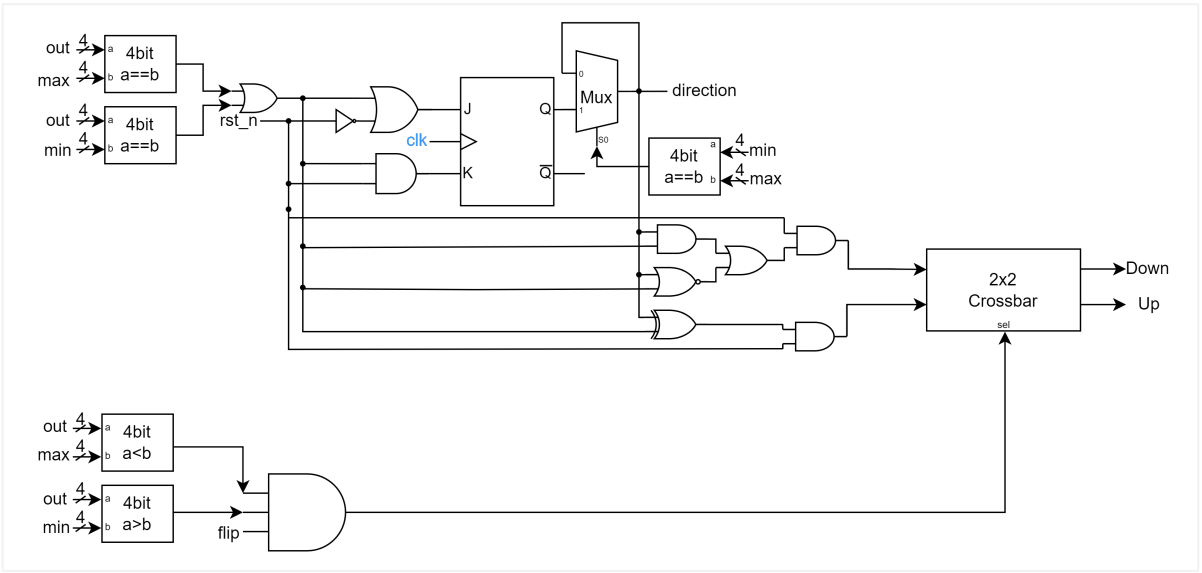
The outputs of the four FIFOs pass through a MUX, with *cnt_delay* as the selector to determine which FIFO's output is currently selected. The delay is necessary because there is a

one clock cycle delay from input to output in the FIFO, and using *cnt_delay* ensures correct correspondence.

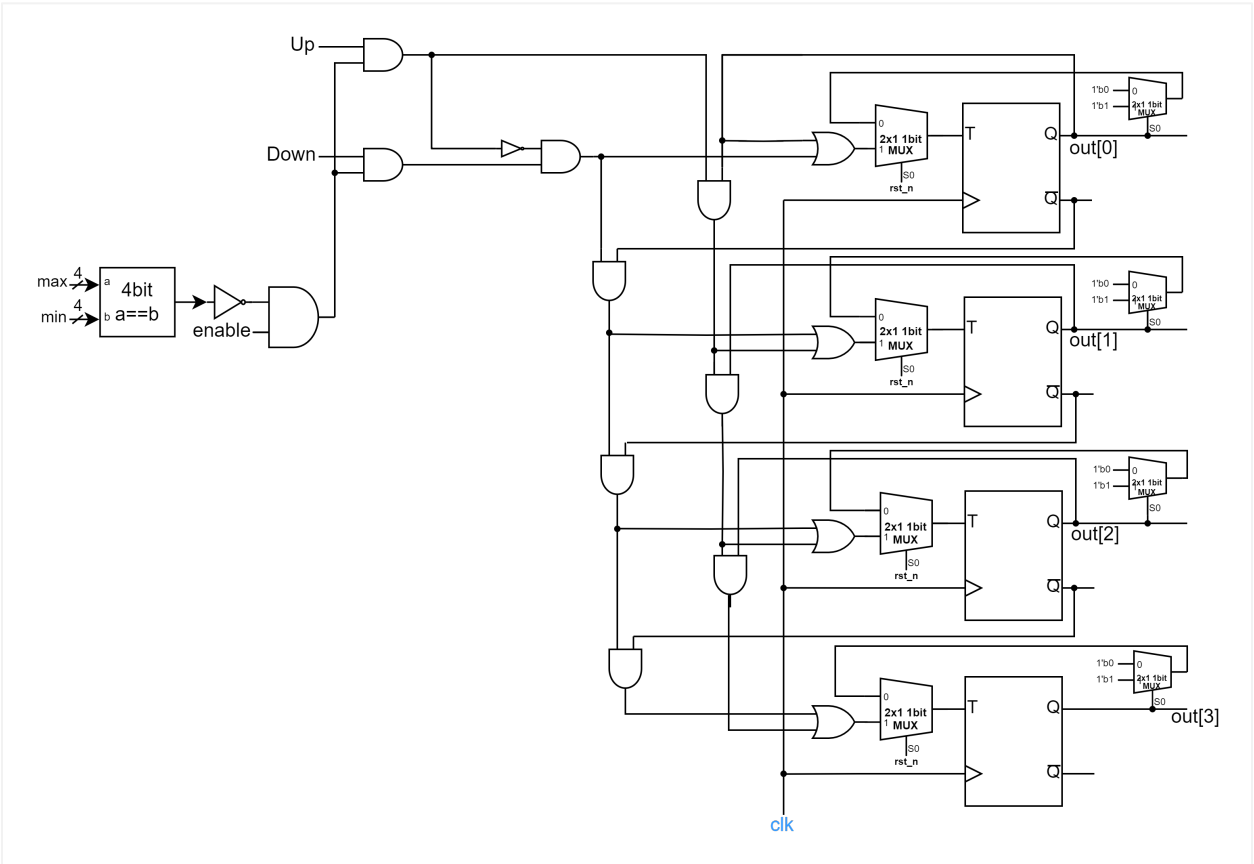
The valid signal is determined by first NORing the error signals from all four FIFOs, then checking *rst_n_c* (which is the result of *rst_n* after passing through a DFF, done to ensure synchronization and prevent valid signal fluctuations between clock cycles due to *rst_n* changes) to determine if *valid* should be forced to *1'b0*.

Finally, *dout* is only output when *valid* is high; otherwise, it outputs *8'b0*. With this, our Round-Robin FIFO Arbiter is complete.

Advanced Q5. 4-bit Parameterized Ping-Pong Counter

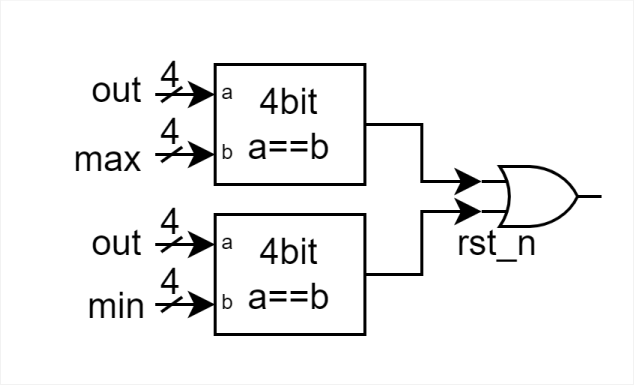


▲ Figure 5.1: Direction Control with flip



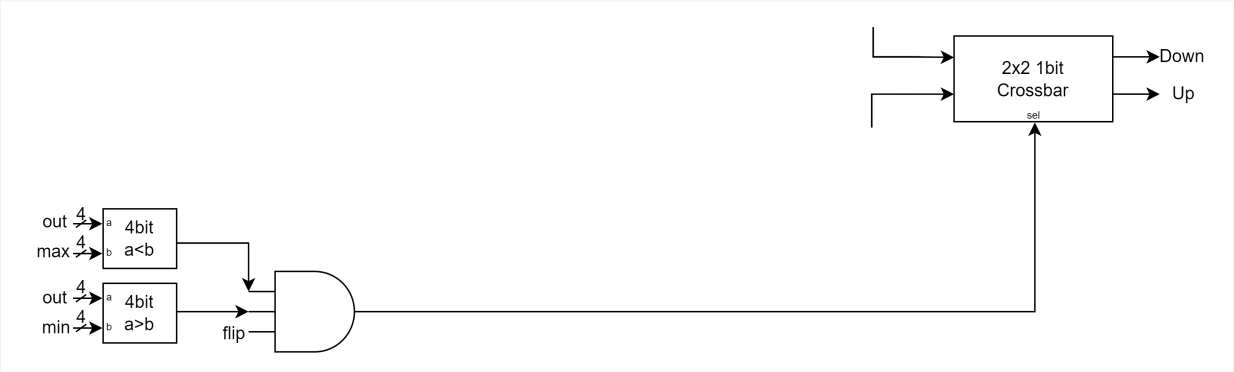
▲ Figure 5.2: new version Up-down counter

We design our Parameterized Ping-Pong Counter based on Question 1's Ping-Pong Counter, and modify some places according to the new features. Below is what we have modified.



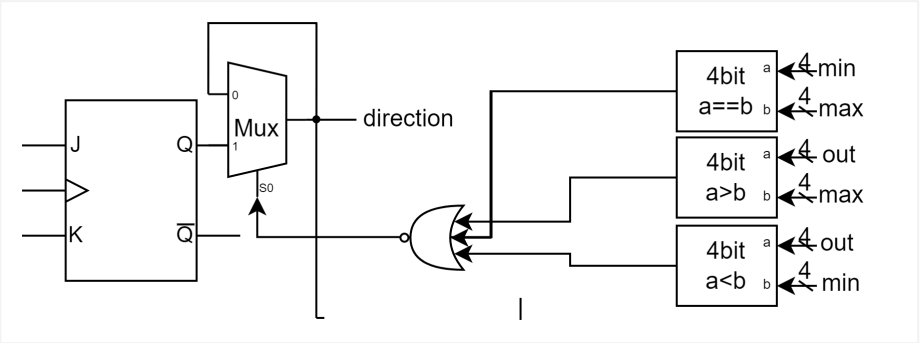
▲ Figure 5.3: direction with flexible max and min

We modify *4'd15* to *max* and *4'd0* to *min*. By making constants into variables, we can improve the module's flexibility.

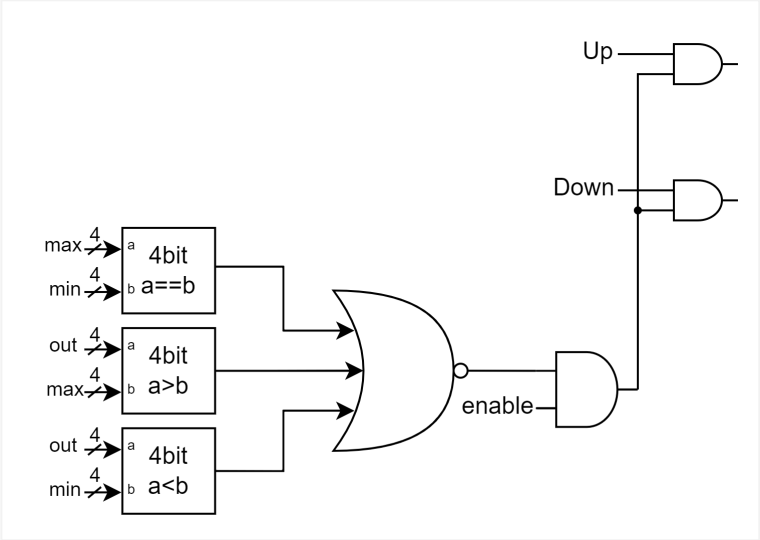


▲ Figure 5.4: flip with Crossbar on directions

Figure 5.4 shows the mechanism of flip. If *flip==1'b1* and *out* is in (*min*, *max*), then the Crossbar will exchange the original Up and Down signals, causing the counting direction to change.



▲ Figure 5.5: direction with special cases



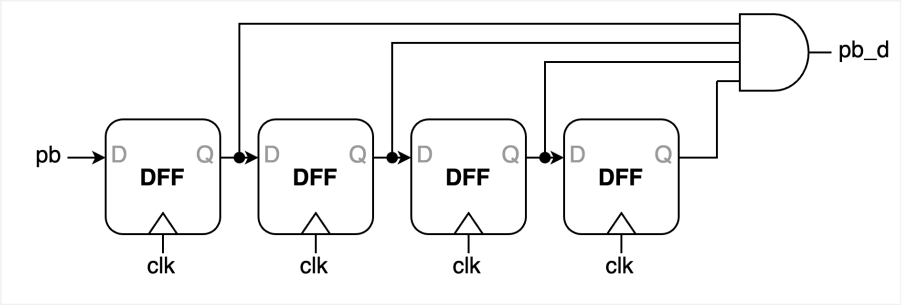
▲ Figure 5.6: enable with special case

In Figure 5.5 and 5.6, we take more conditions into the *enable* and *direction*. If either *min==max*, *out>max*, or *out<min* occurs, the *direction* will keep its value, and the *enable* will lose its effect, making the Up-down counter stop counting. So, *out[3:0]* will also keep its value.

FPGA: 4-bit Parameterized Ping-Pong Counter

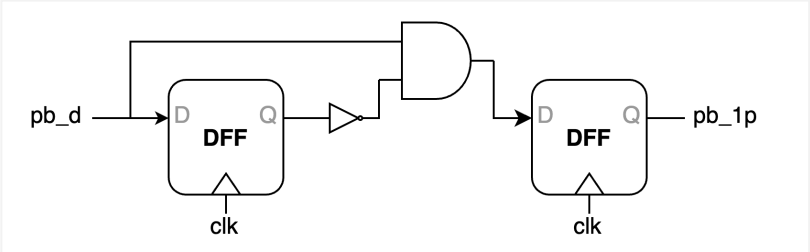
To implement sequential circuits on FPGA, we first have to solve the clock problem. Because the default clock frequency is 100MHz, which is much greater than usual, so we have to use some methods to extend the clock period. We use two registers called **timer** and **timer2** with **17 bits counter** and **26 bits counter** to solve it. When the counters got all their bits 1, then they will toggle the original values of timer and timer2. Therefore, the period of timer is extended to 0.0026secs, and the period of timer2 is extended to 1.34secs.

After we solve the clock period problem, we then have to solve glitches and multiple triggering problems.



▲ Figure 6.1: Debounce Circuit

To solve glitches problem, we introduce the **debounce circuit**. Debounce circuit use multiple DFFs linked back to back to prevent the signal value from changing too quick and causing some annoying bugs. Our implementation uses ten DFFs linked back to back. If one wants to trigger the button's function, they need to press the button for at least **ten clock periods**, therefore significantly reduced the possibility of glitches.



▲ Figure 6.2: One-pulse Circuit

To solve multiple triggering problems, we use the **one-pulse circuit**. One pulse circuit handle the problem by focusing on the changing moment of signal. When pressing the button, the *pb_debounced* comes up, and A comes down at the next clock positive edge. By using an and-gate takes on A and *pb_debounced*, B can come up exactly one clock cycle, and after the rightmost DFF takes B, the output will also come up exactly one clock cycle, therefore trigger exactly once.

We have to set the 7-segment display. To make the four LEDs look like they light up at the same time, we use **timer** to be *c_light[3:0]*'s clock input(period: 0.0026secs) in order to trigger the "**Persistence of vision**". The initial value of *AN[3:0]* is *4'b1110*, and it will change to *1101*, *1011*, *0111*, *1110*, …, periodically based on timer.

Lastly, we have to get the 7 segments' Boolean expression according to which LED is lighted exactly. To solve this, we refer to the design on lecture slide and get the Boolean expressions:

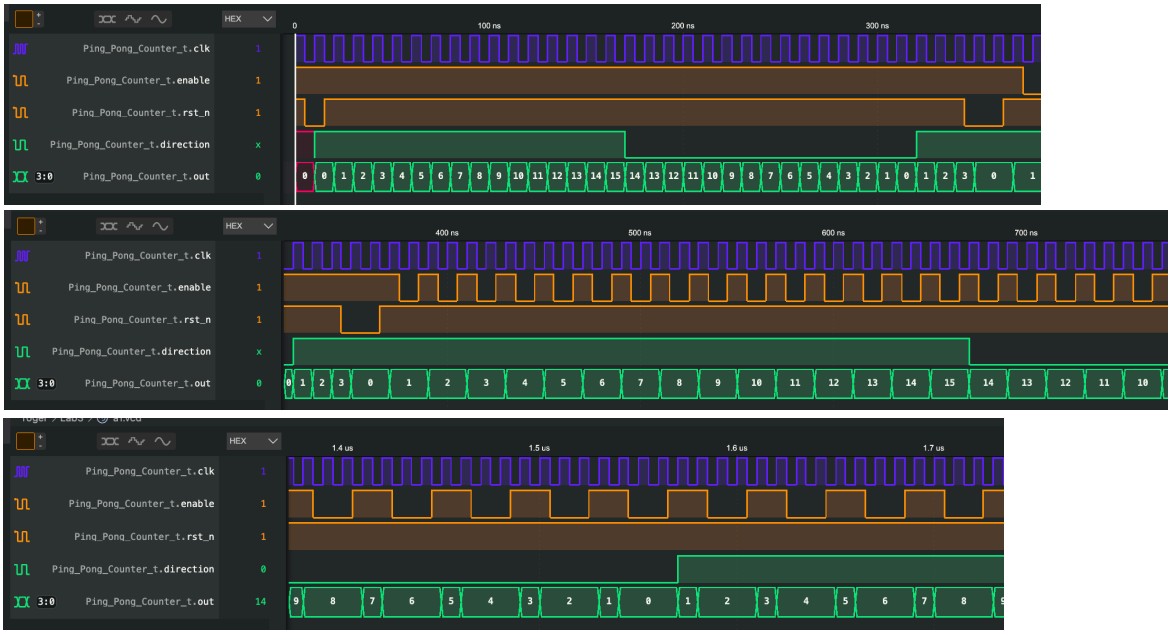
```
segment[0] = !((!c_light[0] & direction) | (!c_light[1] & direction) | (!c_light[2] & !(out==1
|| out==4 || out==11 || out==14)) | (!c_light[3] & !(out==10 || out==11 ||out==12 || out==13 ||
out==14 || out==15)))
segment[1] = !((!c_light[0] & direction) | (!c_light[1] & direction) | (!c_light[2] & !(out==5
|| out==6 || out==15)) | (!c_light[3]))
segment[2] = !((!c_light[0] & !direction) | (!c_light[1] & !direction) | (!c_light[2]
& !(out==2 || out==12)) | (!c_light[3]))
segment[3] = !((!c_light[0] & !direction) | (!c_light[1] & !direction) | (!c_light[2]
& !(out==1 || out==4 || out==7 || out==11 || out==14)) | (!c_light[3] & !(out==10 || out==11
|| out==12 || out==13 || out==14 || out==15)))
segment[4] = !((!c_light[0] & !direction) | (!c_light[1] & !direction) | (!c_light[2] & (out==0
|| out==2 || out==6 || out==8 || out==10 || out==12)) | (!c_light[3] & !(out==10 || out==11 ||
out==12 || out==13 || out==14 || out==15)))
segment[5] = !((!c_light[0] & direction) | (!c_light[1] & direction) | (!c_light[2] & !(out==1
|| out==2 || out==3 || out==7 || out==11 || out==12 || out==13)) | (!c_light[3] & !(out==10 ||
out==11 || out==12 || out==13 || out==14 || out==15)))
segment[6] = !(!c_light[2] & !(out==0 || out==1 || out==7 || out==10 || out==11))
```

After we get the above expressions, we can construct the FPGA module.

(output: c_light[3:0] -> AN[3:0], segment[6:0] -> 'g', 'f', 'e', 'd', 'c', 'b', 'a')

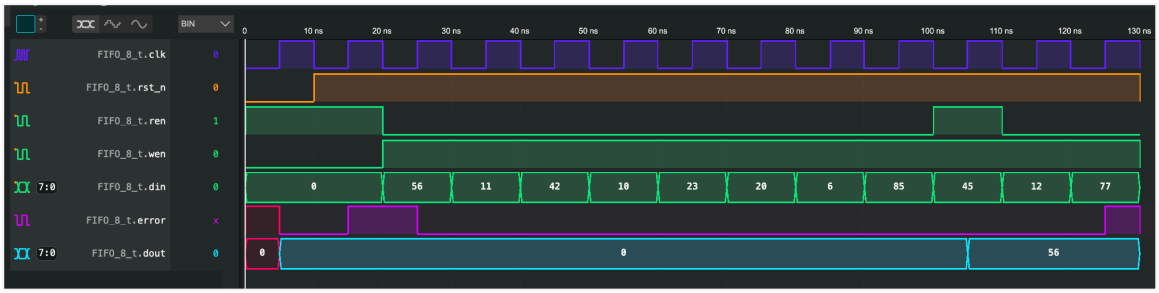
Testbenches

Q1. 4-bit Ping-Pong Counter



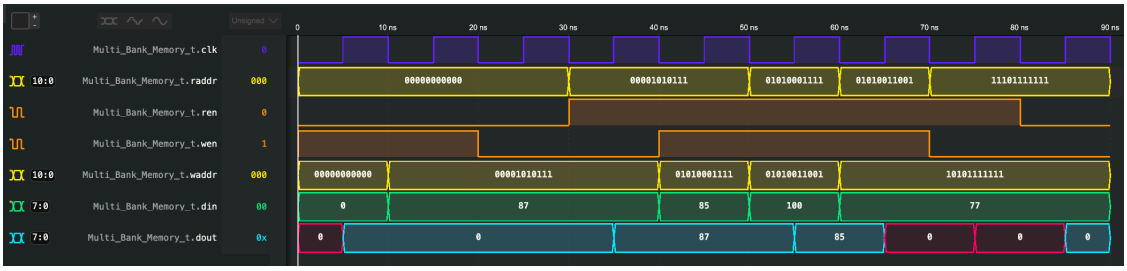
In this testbench, we initially use *rst_n* to reset all values. We then run 32 clock cycles. Subsequently, we test the enable signal by turning it on for one clock cycle and off for one clock cycle, followed by turning it on for two clock cycles and off for two clock cycles. All operations function normally.

Q2. First-In First Out (FIFO) Queue



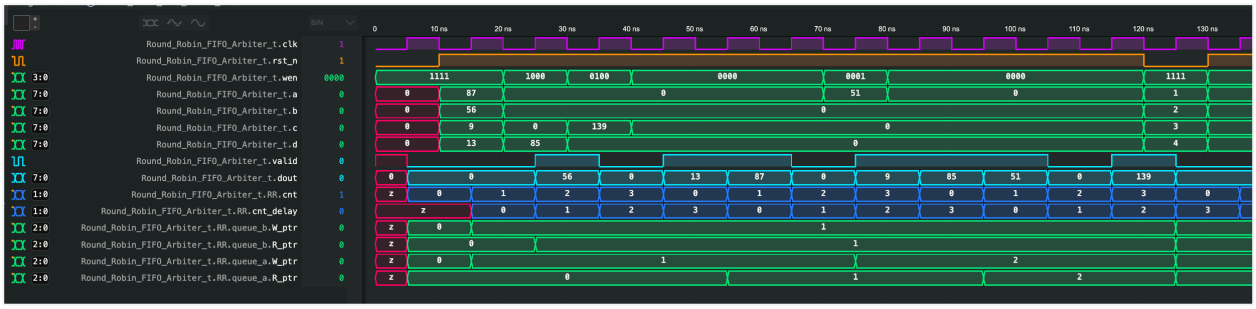
The testbench for this Advanced Q2 has been constructed in strict accordance with the module diagrams provided in the assignment presentation. As such, further elaboration on its structure is not deemed necessary.

Q3. Multi-Bank Memory



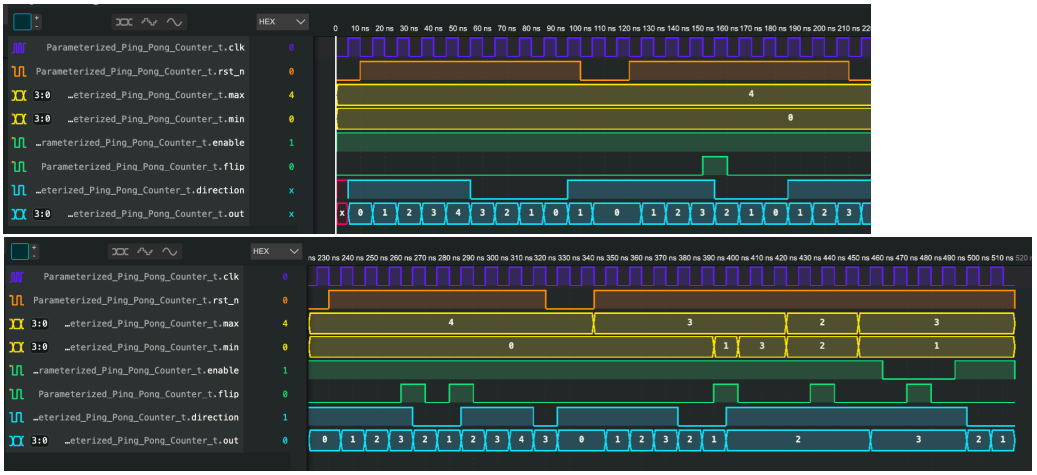
Given that it is not feasible to comprehensively test all possible values for this problem, we generate and test edge cases. These include scenarios where both *ren* and *wen* signals are set to one, cases where only one of them is set to one, and instances where neither is set to one.

Q4. Round-Robin FIFO Arbiter



The testbench for this Advanced Q4 has been constructed in strict accordance with the module diagrams provided in the assignment presentation. As such, further elaboration on its structure is not deemed necessary.

Q5. 4-bit Parameterized Ping-Pong Counter



Initially, we keep the *max*, *min*, and *enable* signals constant to verify if the Ping-Pong Counter operates as expected and if *rst_n* functions normally. In the second figure, we test multiple *flip* signal and modify *max*/*min* during the process. We check whether the *out* changes accordingly and if the current value is held when the range is incorrect.

What have we learned from Lab 3?

This Lab 3 differs from previous labs that used gate-level design. Although writing Verilog at the gate level was more complex and time-consuming, requiring pre-planning of the circuit to achieve the desired functionality before expressing it in gate-level representation, it made drawing diagrams relatively intuitive. This time, we were able to use Data flow modeling and Behavior modeling. While this allowed for more concise and intuitive code writing, converting it into circuit diagrams proved to be a significant challenge. The focus on sequential code in this lab required particular attention to ensure clock behavior aligned with our expectations. Understanding the differences between blocking and non-blocking assignments was also a key learning point. In summary, this Lab 3 truly enabled us to learn many new syntaxes and concepts. Importantly, this included the process of translating Verilog into circuit diagrams.

Contributions

謝佳晉：

wrote Verilog modules: Q1, Q2, Q3, Q4, Q5
wrote testbenches: Q1, Q2, Q3, Q4, Q5
performed simulation: Q1, Q2, Q3, Q4, Q5
drew diagrams: Q1, Q2, Q5
wrote report: Q1, Q2, Q5, FPGA
wrote and made FPGA implementation

范升維：

wrote Verilog modules: Q1, Q2, Q3, Q4, Q5
wrote testbenches: Q1, Q2
performed simulation: Q1, Q2, Q3, Q4, Q5
drew diagrams: Q3, Q4
wrote report: Q3, Q4, testbenches, what we learned
organized whole report

	Q1	Q2	Q3	Q4	Q5	FPGA	What we learned
wrote Verilog module							
wrote testbench							
performed simulation							
drew diagram							
wrote report							
wrote testbench report							
organized whole report							

Both
謝佳晉
范升維