**Hardware Design and Lab: Lab4**

**111060013 EECS 26' 劉祐廷**

**Catalog**

**1. Basic Question:**

**Many-to-one LFSR and One-to-many LFSR………………P3**

**2. Advanced Question:**

**Content-addressable memory (CAM) design…...………….P4**

**3. Advanced Question:**

**Scan Chain Design…….……………………….…………..…...P6**

**4. Advanced Question:**

**Built-in Self Test……………………………………….……..P8**

**5. Advanced Question:**

**Built-in Self Test FPGA…………………………………....P10**

**6. What I Have Learned…...................................................P15**

1. **Basic Question**
2. **Many-to-one LFSR**
3. **State Transition Diagram**

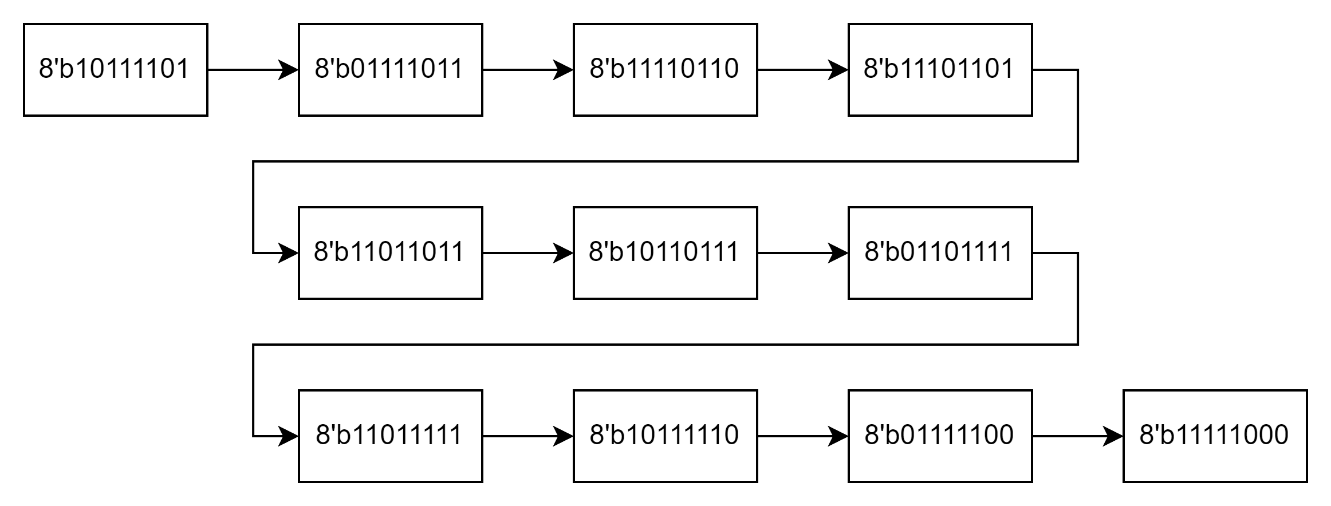
****

Figure 1.1

1. **Basic Question**

If we reset the DFFs to 8'd0, then DFFs will remain 8'd0 because (DFF[1] ^ DFF[2]) ^ (DFF[3] ^ DFF[7]) is 0. Therefore, out will be a constant 1'b0.

1. **One-to-many LFSR**
2. **State Transition Diagram**

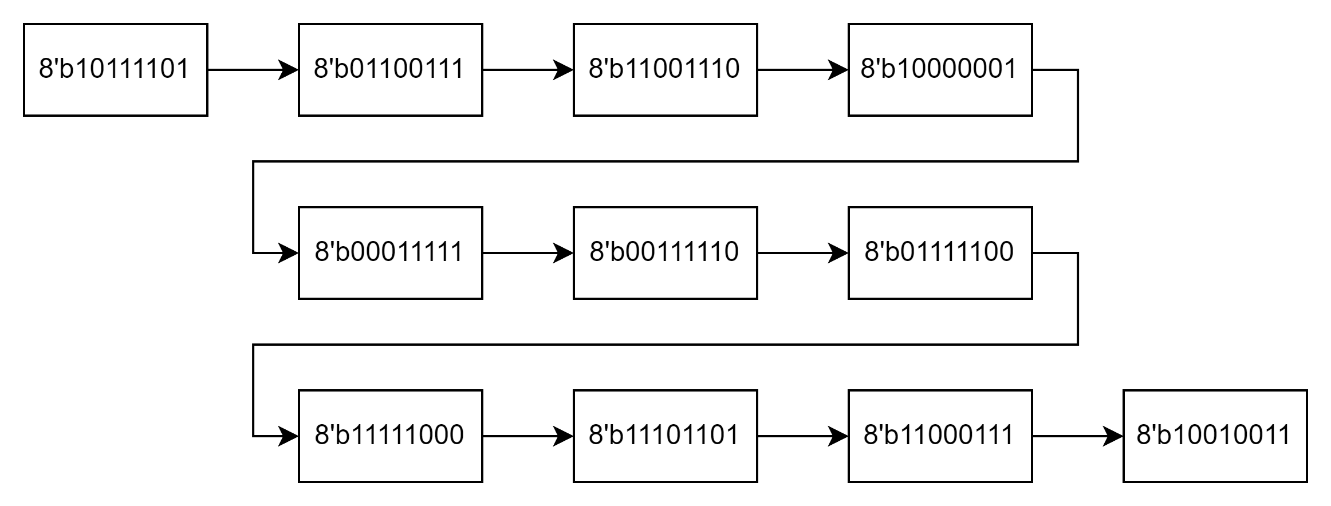
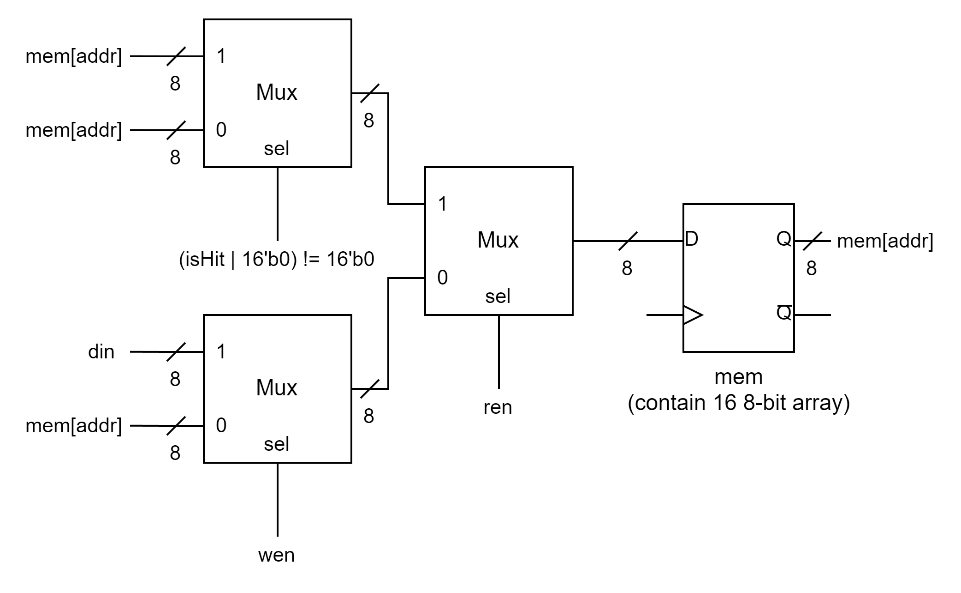


Figure 1.2

1. **Basic Question**

If we reset the DFFs to 8'd0, then DFFs will remain 8'd0 because DFF[7] XOR any other bit is 0. Therefore, out will be a constant 1'b0.

1. **Advanced Question: Content-addressable memory (CAM) design**
2. **Block Diagram**

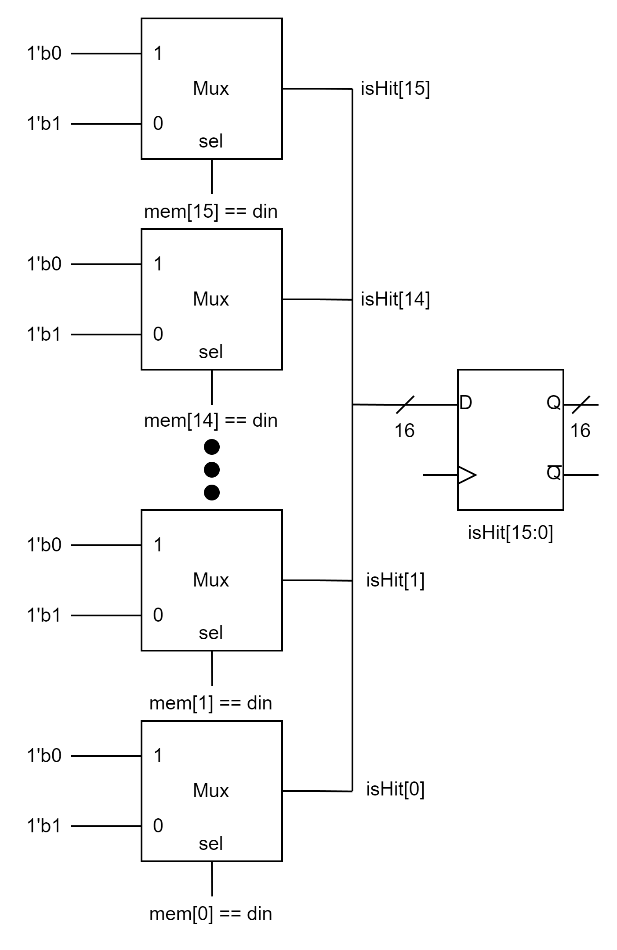
****

Figure 2.2

mem

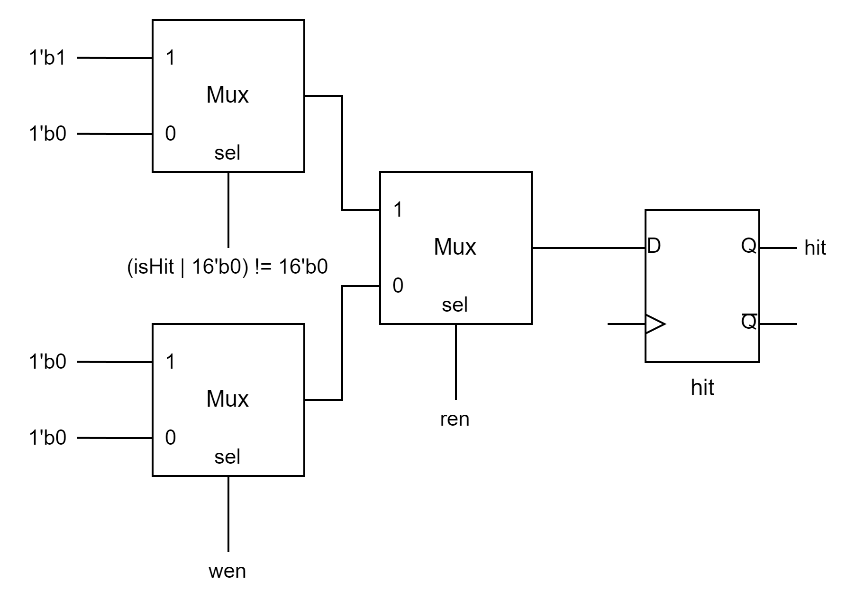
****

Figure 2.1

isHit[15:0]

Figure 2.3

hit

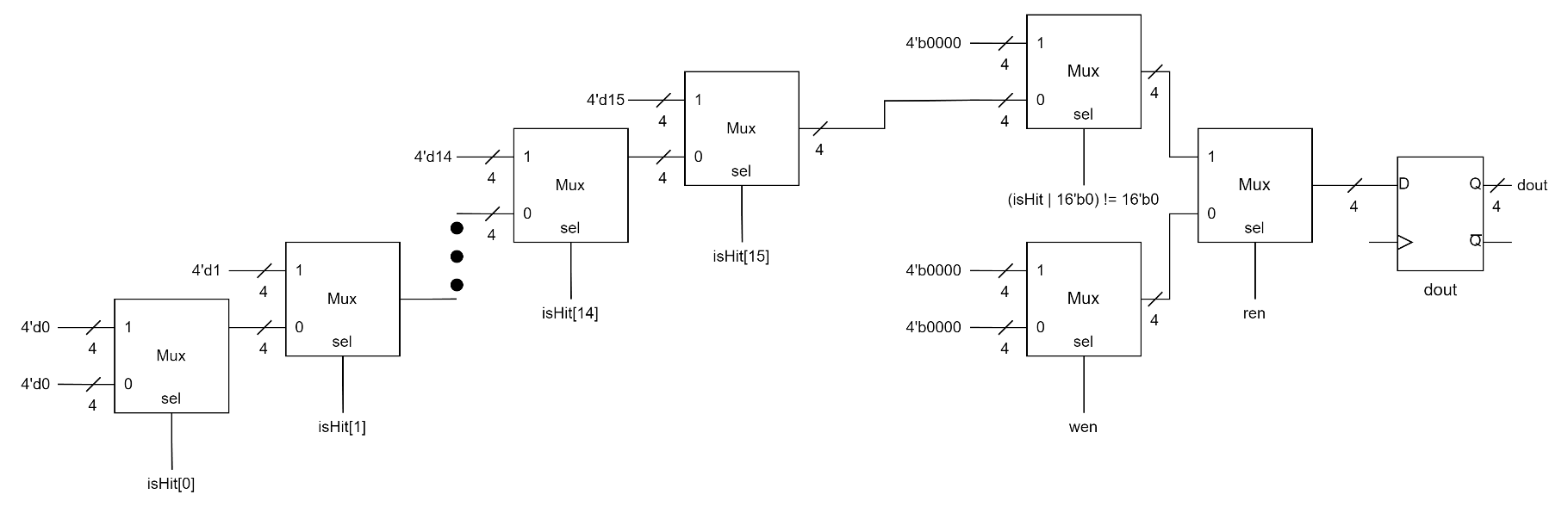
****

Figure 2.4

dout

1. **Explanation**

In this problem, I defined a reg called **isHit[15:0]** as **Figure 2.1** to save the hit status. If the i-th bit in **isHit** equal to **din**, then **isHit[i]** will be set to 1'b1. Otherwise it will be set to 1'b0. **Figure 2.2** shows how I design the memory. If **ren** is 1'b0 and **wen** is 1'b1, then **din** will be written into **mem[addr]**. Otherwise, **mem[addr]** will not change. As mention above, **isHit** == **16'b0** means that there is no hit condition happened. Therefore, I designed **hit** as it shows in **Figure 2.3**. According to the specification, if **hit** == **1'b1**, we should output the largest address of the bits that is hit. So I designed the circuit as it shows in **Figure 2.4** to realize this function.

1. **Testbench**

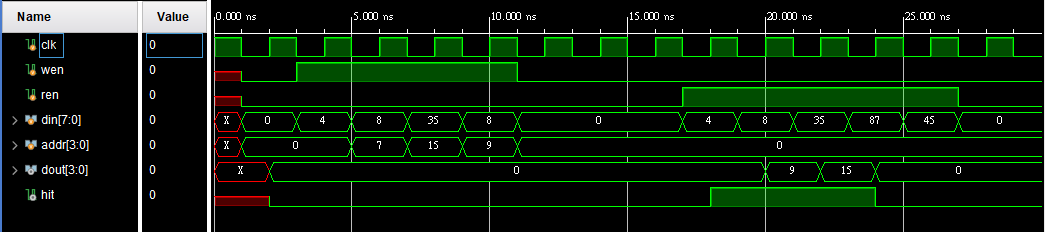
****

Figure 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 **Figure 2.5**, we can see that everything works correctly. And then I also add some extra testcases to check if the design work correctly. As **Figure 2.6** shows, it seems that it works correctly.

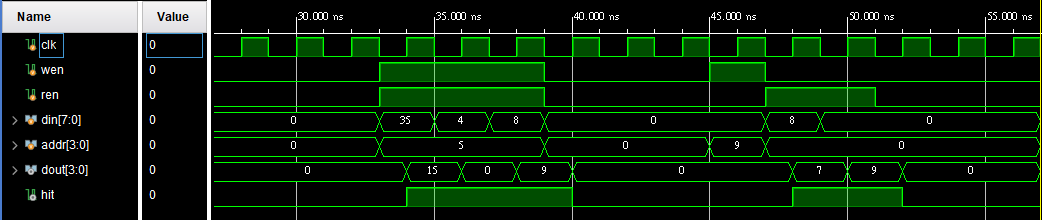


Figure 2.6 wave form 2

1. **Advanced Question: Scan Chain Design**
2. **Block Diagram**

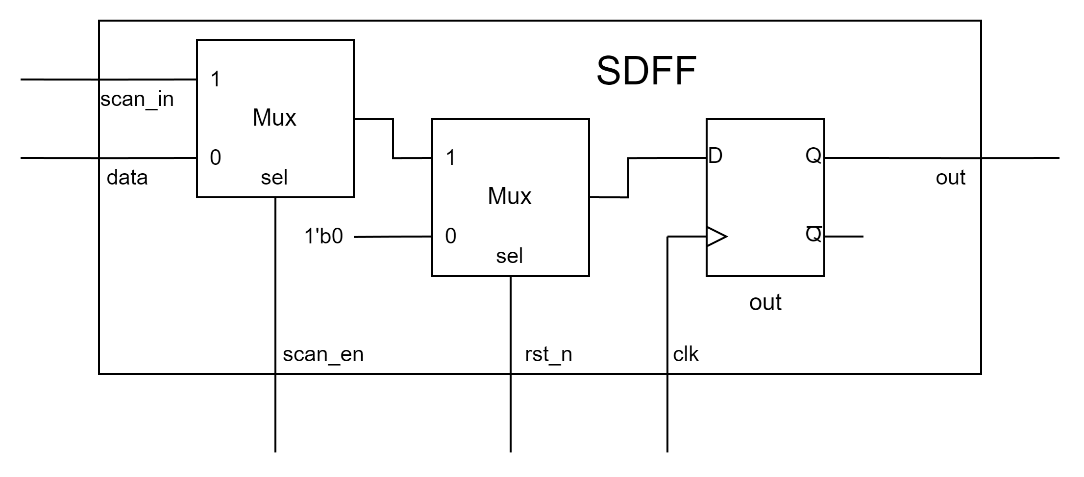
****

Figure 3.1

SDFF

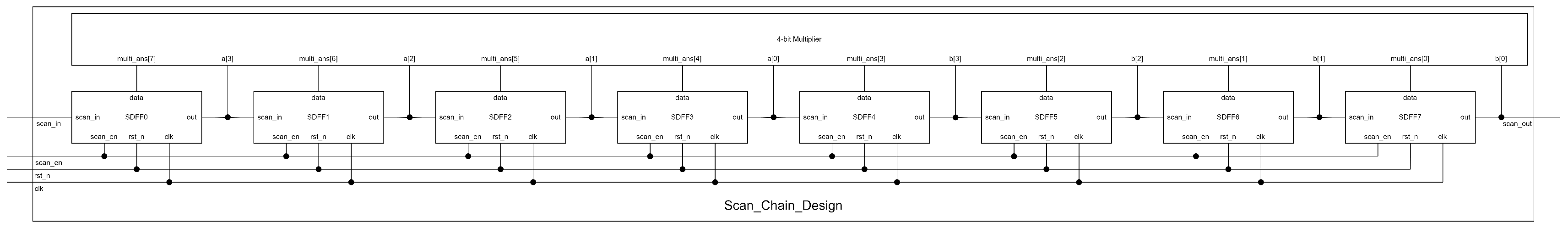
****

Figure 3.2

Scan\_Chain\_Design (Left Part)

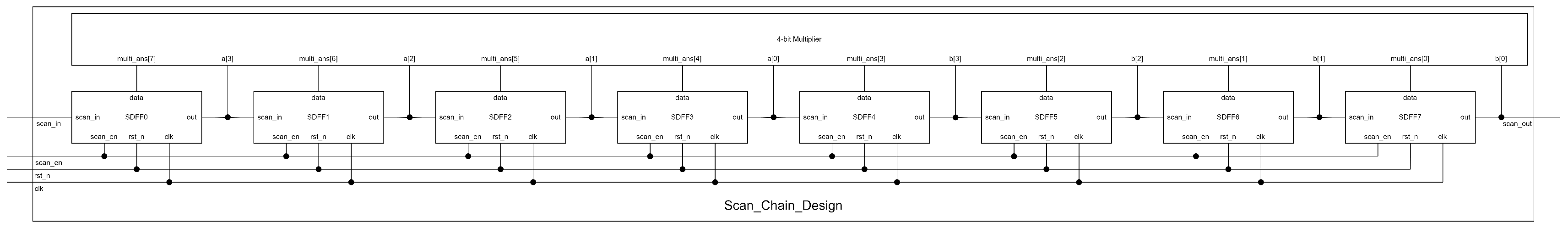
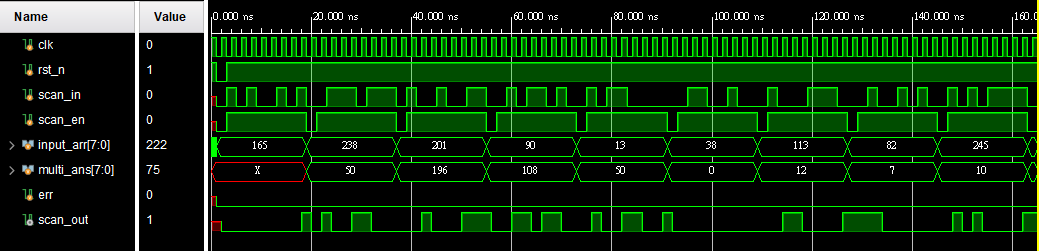
****

Figure 3.3

Scan\_Chain\_Design (Right Part)

1. **Explanation**

In this problem, the lecture slides provide a complete design. Thus I realize the circuit by Verilog according to the **Figure 3.1**, **Figure 3.2**, and **Figure 3.3**. Note that **4-bit Multiplier** is designed by behavioral-level code.

1. **Testbench**

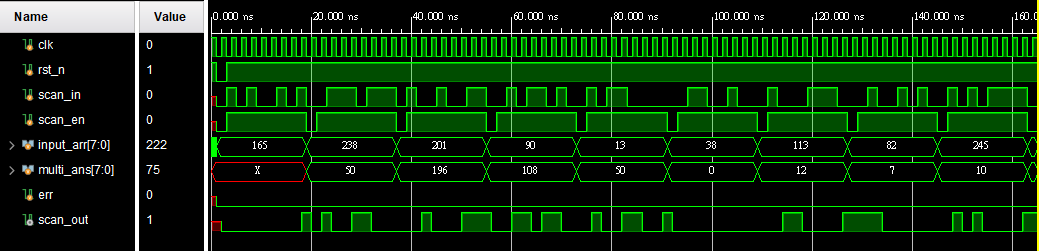


Figure 3.4

Wave form

In this problem, I found that it is difficult to judge the results with the naked eyes. Therefore, as I've learned form Lab2, I defined a signal called **err** to help me check the result. In testbench, the input sequence will be stored in **input\_arr[7:0]** first. And then it will be scan bit by bit into the DFFs. After input 8 bits of data, it will count **multi\_ans[7:0]** as **input\_arr[7:4]** \* **input\_arr[3:0]** and then **input\_ans** will be set to the next testcase. After **scan\_en** pulled up again, the testbench will check the output bit by bit compare to **multi\_ans**. If detecting any errors, **err** will be pulled up. Note that **err** will be reset to 1'b0 while **scan\_en** is **1'b0**. In this way, I can check the output more efficiently and precisely.

1. **Advanced Question: Built-in Self Test**
2. **Block Diagram**

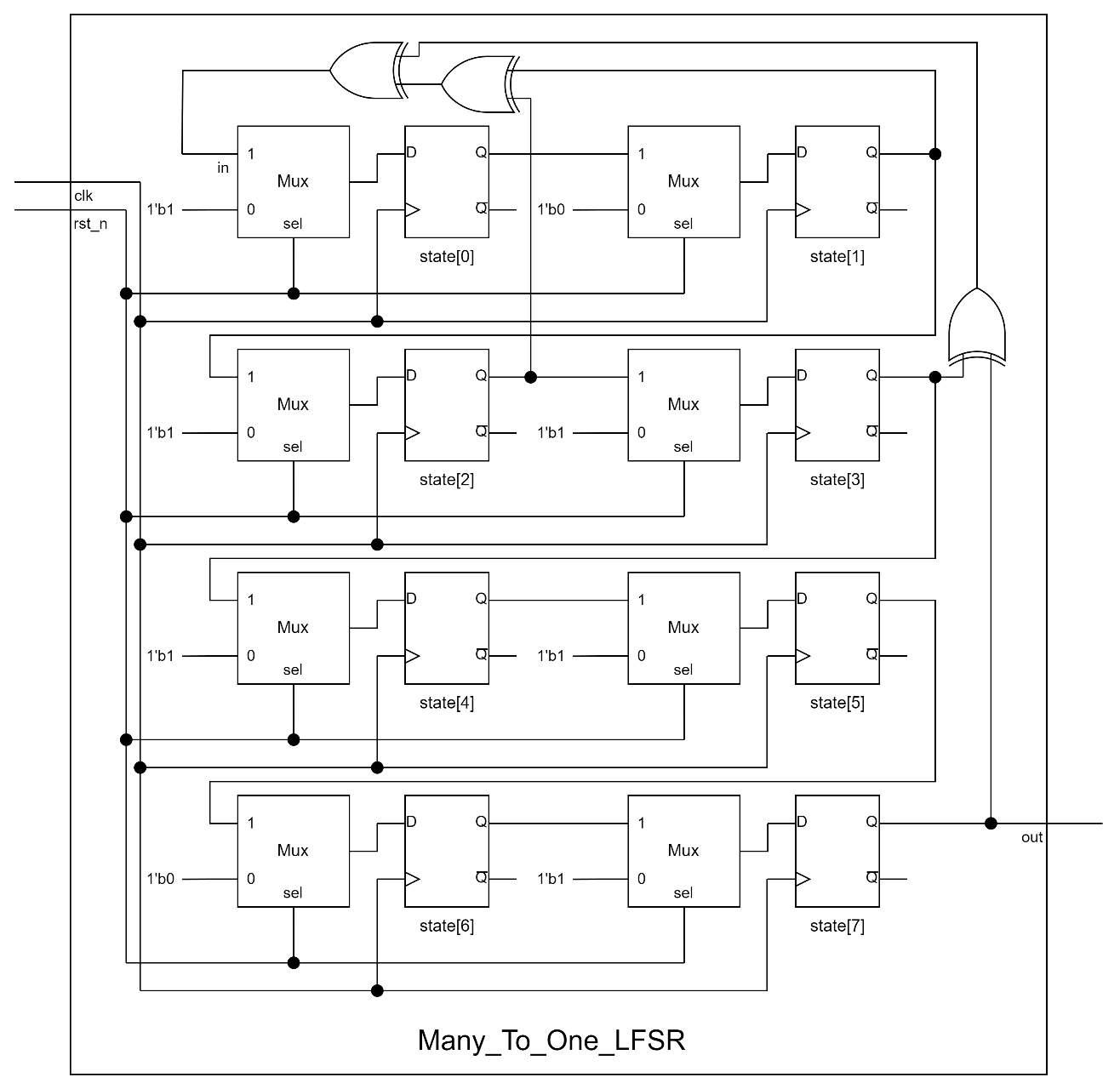
****

Figure 4.1 Many\_To\_One\_LFSR

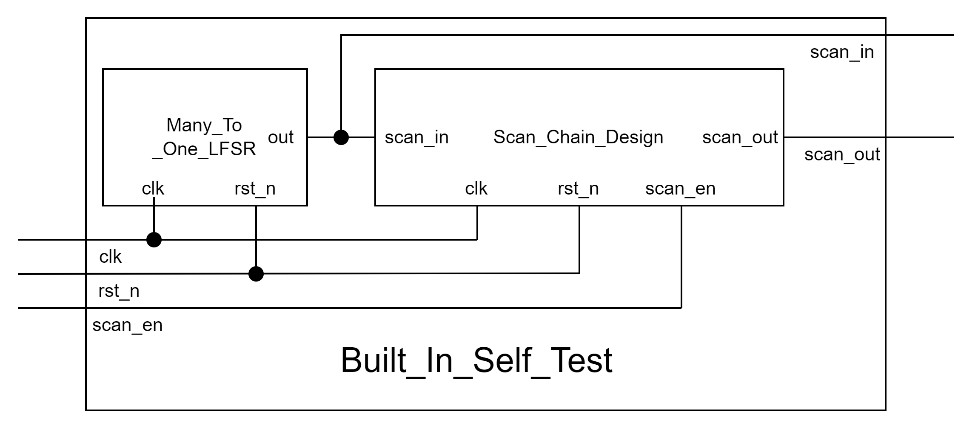
****

Figure 4.2 Built\_In\_Self\_Test

1. **Explanation**

In this problem, I only have to combine to modules I 've designed in the previous problems. **Figure 4.1** shows how I designed **Many\_To\_One\_LFSR**. **Figure 4.2** shows how I connect these two modules together.

1. **Testbench**

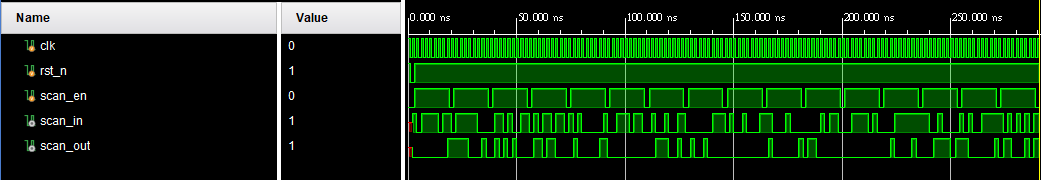


Figure 4.3 wave form

In this design, testcases are generated by **Many\_To\_One\_LFSR**. It seems that the method of how I designed in the testbench of **Scan\_Chain\_Design** is risky because there are some chances that I have some wrong cases in my testbench. Therefore, I write a code in C++ to simulate the testcases and check the results one by one. And every result seems right.

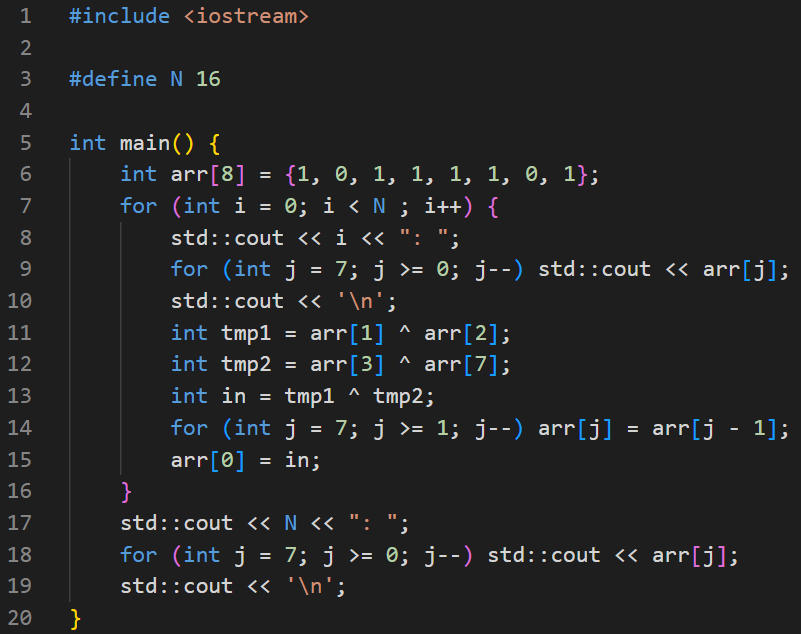


Figure 4.4

C++ code for simulating the testcases

1. **Advanced Question: Built-in Self Test FPGA**
2. **Block Diagram**

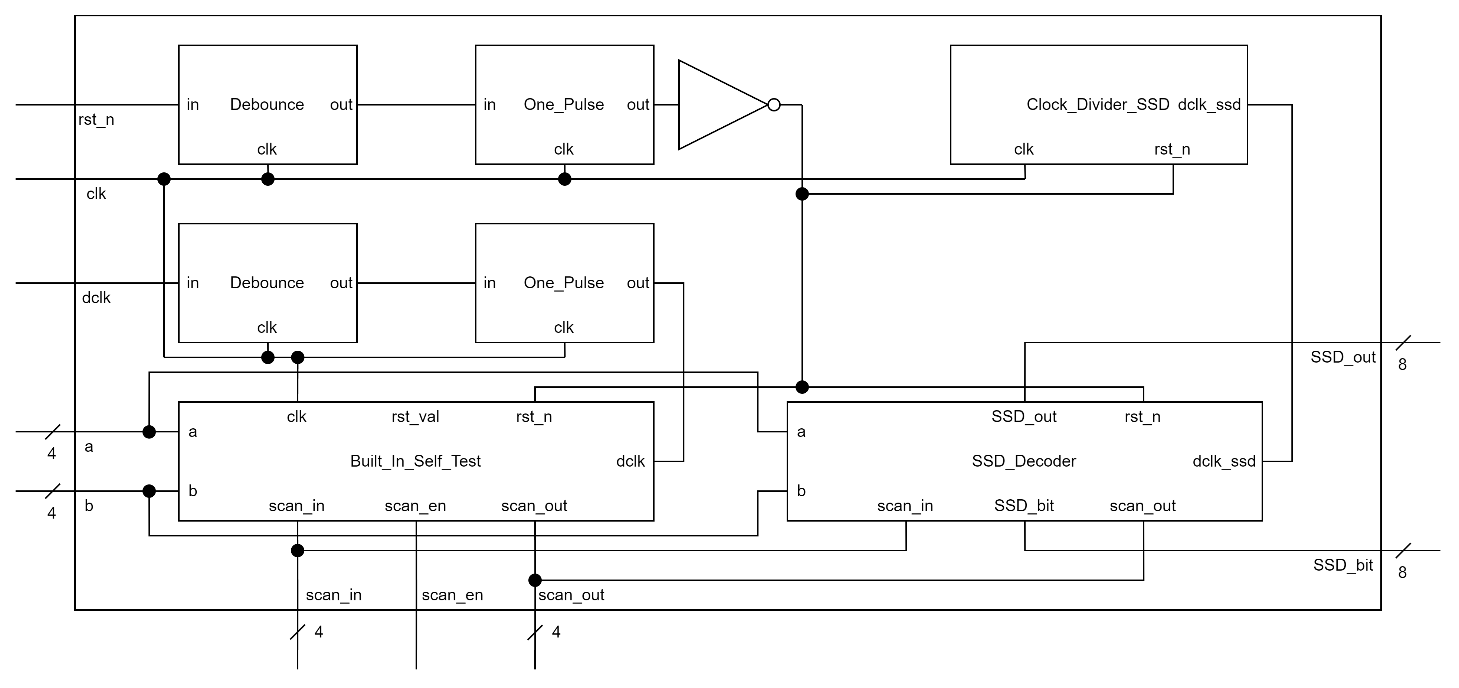
****

Figure 5.1

The whole design of this problem

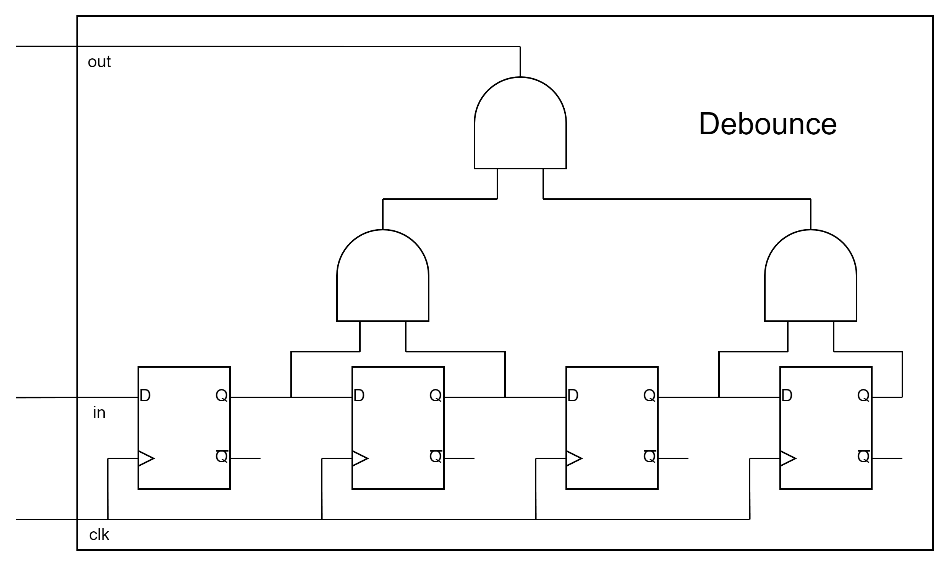
****

Figure 5.2

Debounce

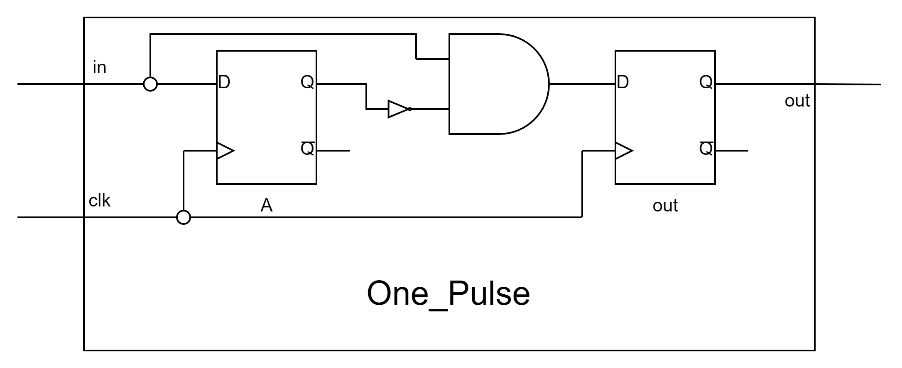
****

Figure 5.3

One\_Pulse

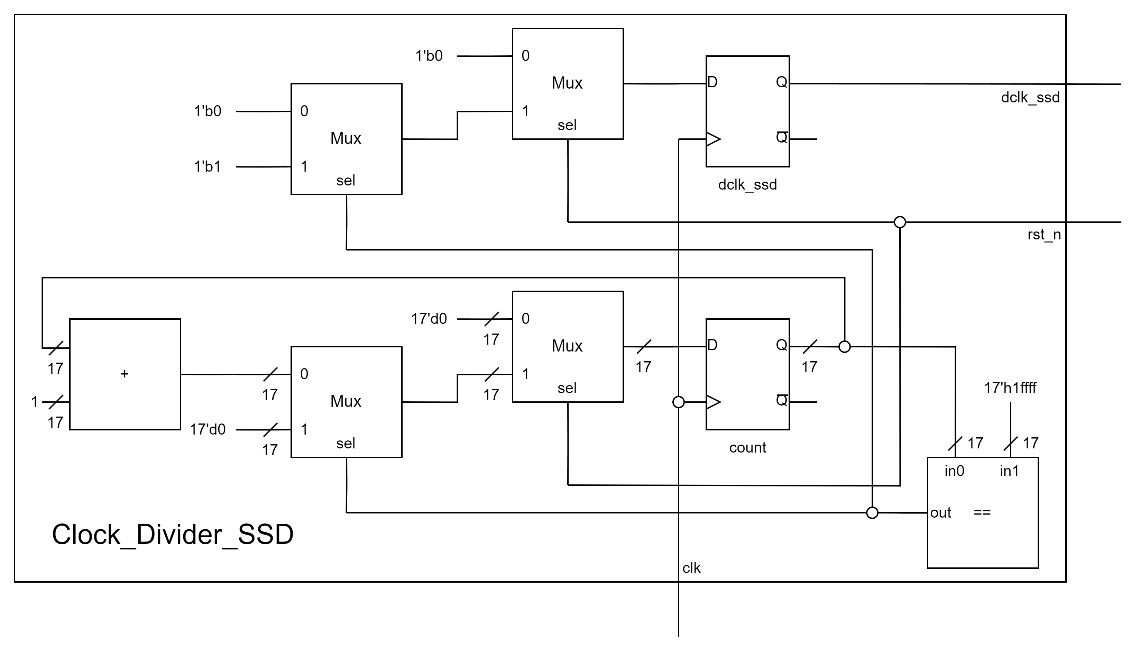
****

Figure 5.4

Clock\_Devider\_SSD

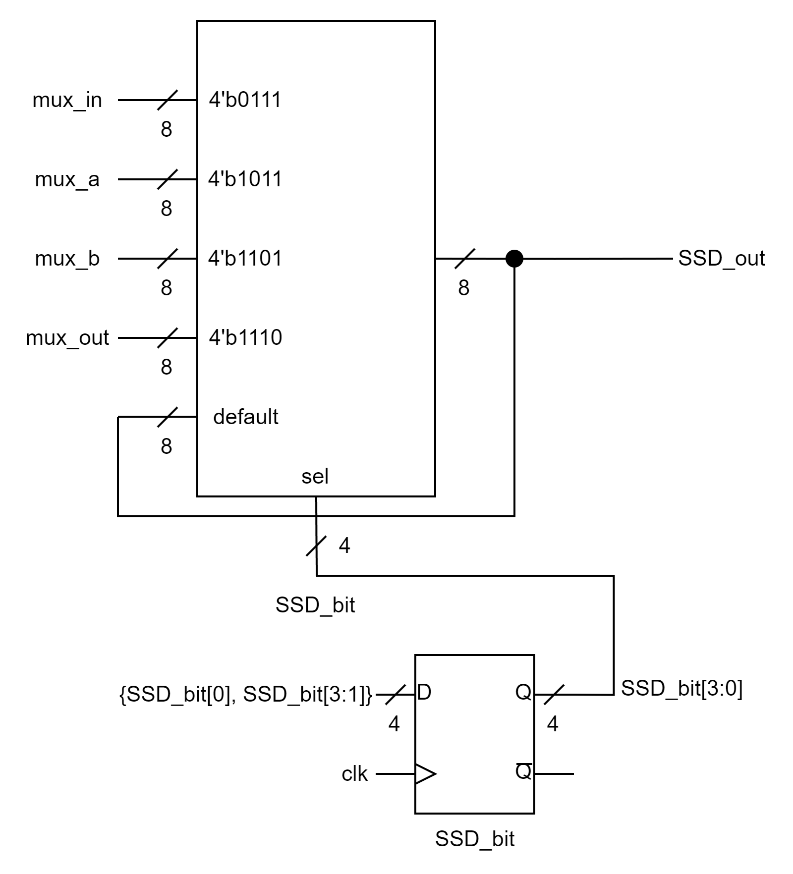
****

Figure 5.5

SSD\_Decoder

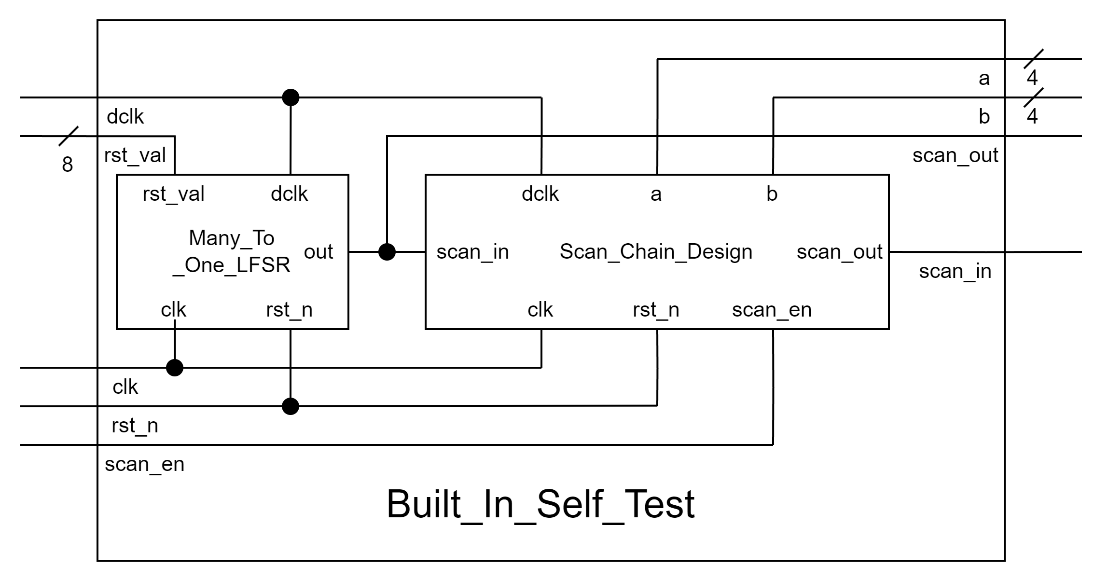
****

Figure 5.6

Built\_In\_Self\_Test

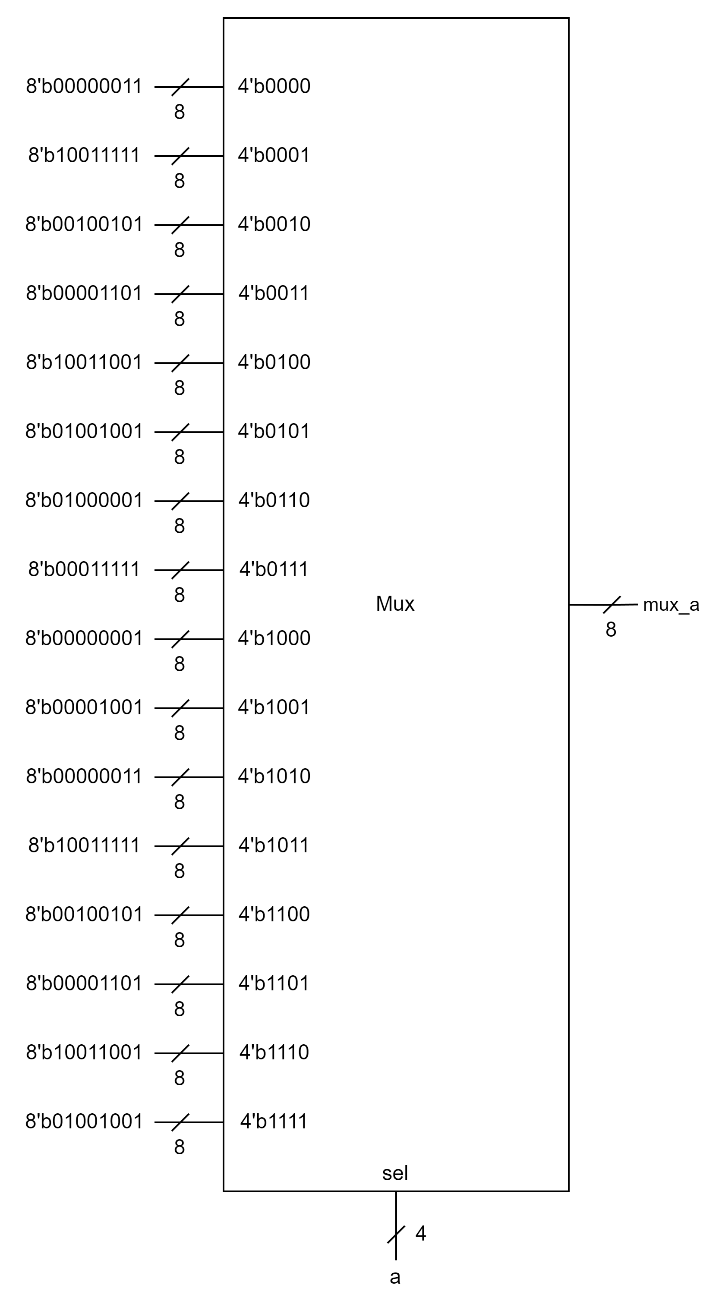
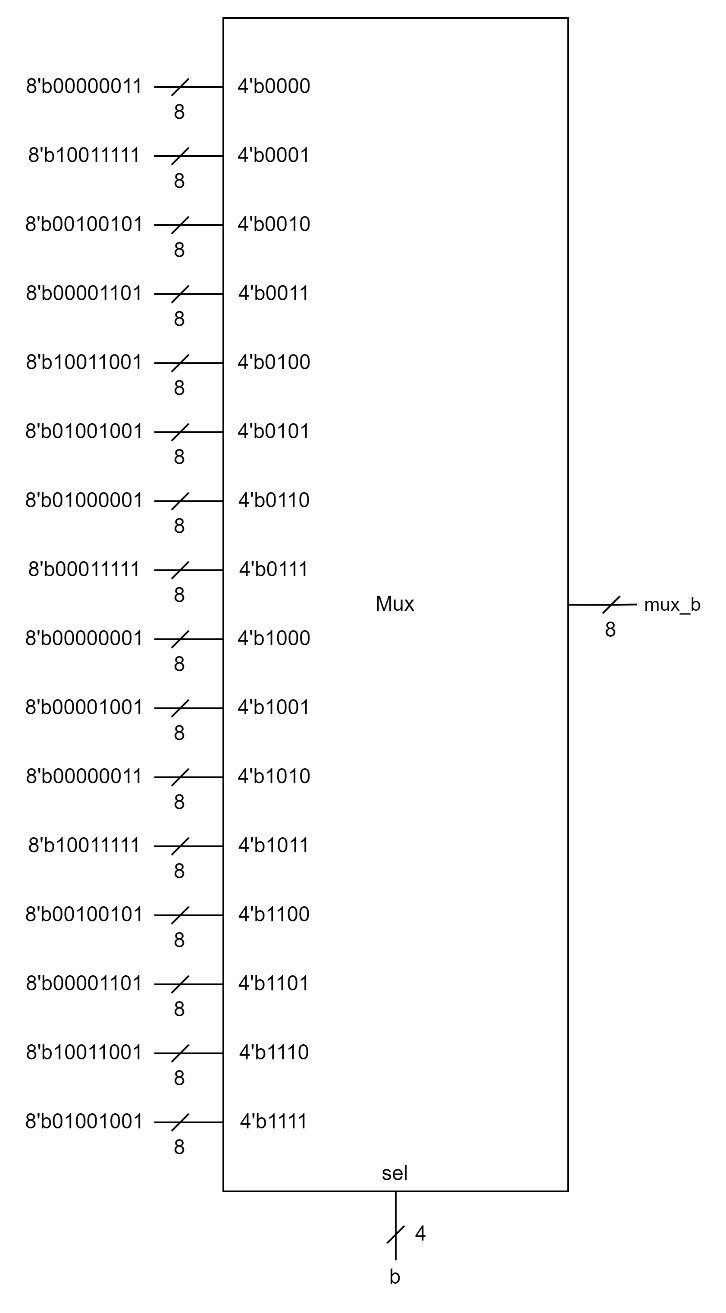
****

Figure 5.8

mux\_b

Figure 5.7

mux\_a

****

Figure 5.10

mux\_out

Figure 5.9

mux\_in

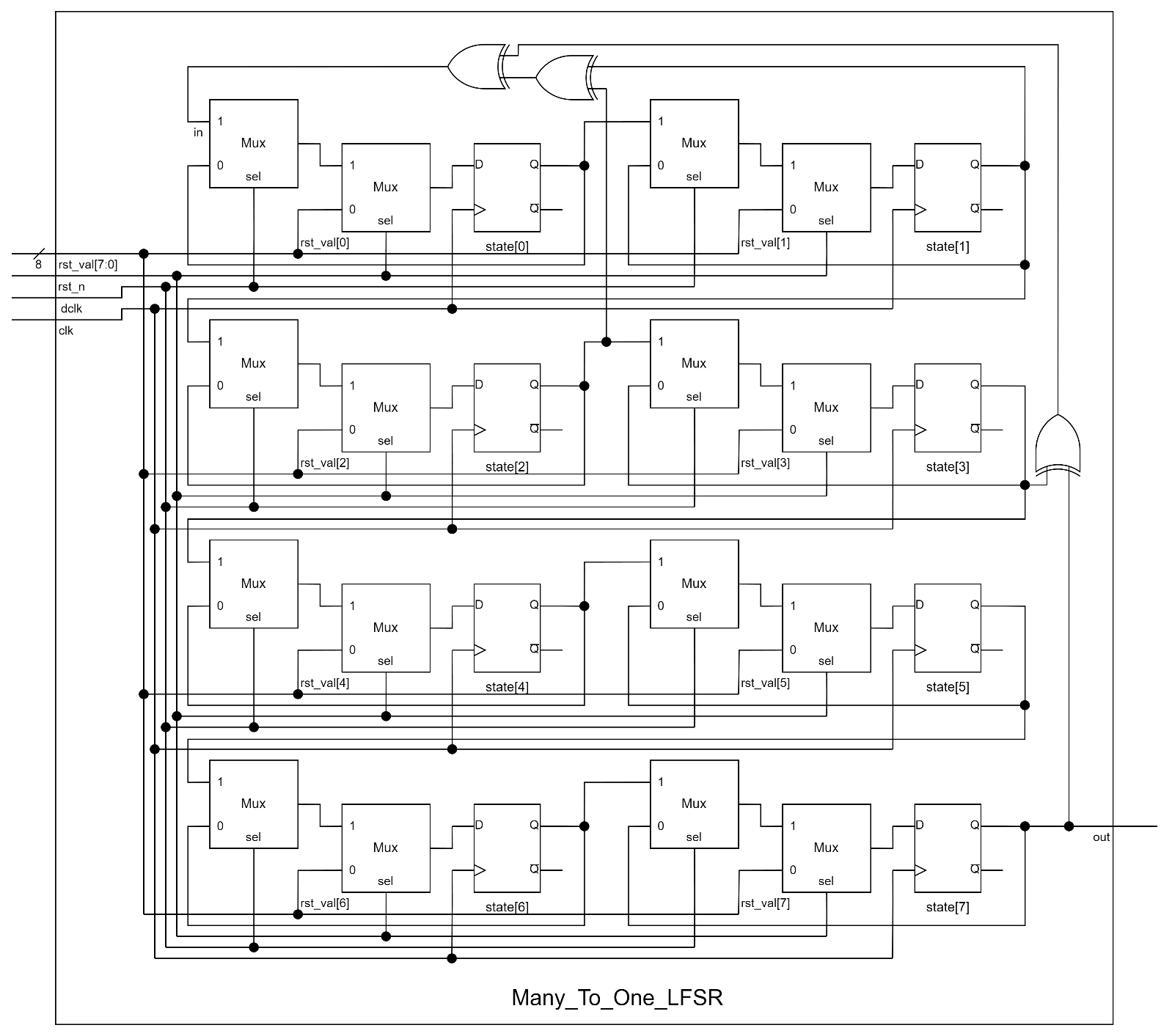
****

Figure 5.11

Many\_To\_One\_LFSR

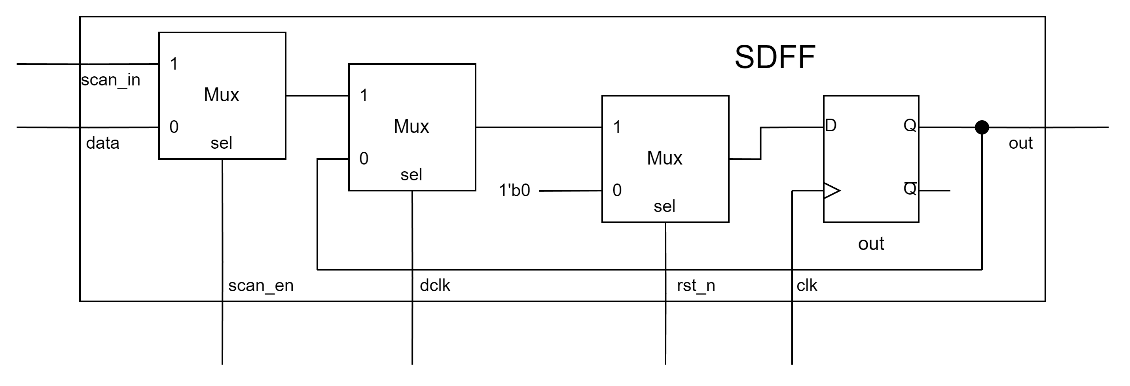
****

Figure 5.12

SDFF

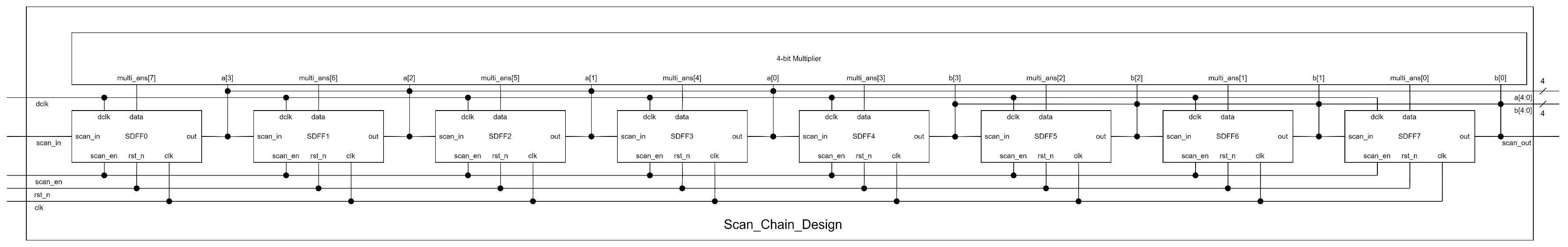
****

Figure 5.13

Scan\_Chain\_Design (Left Part)

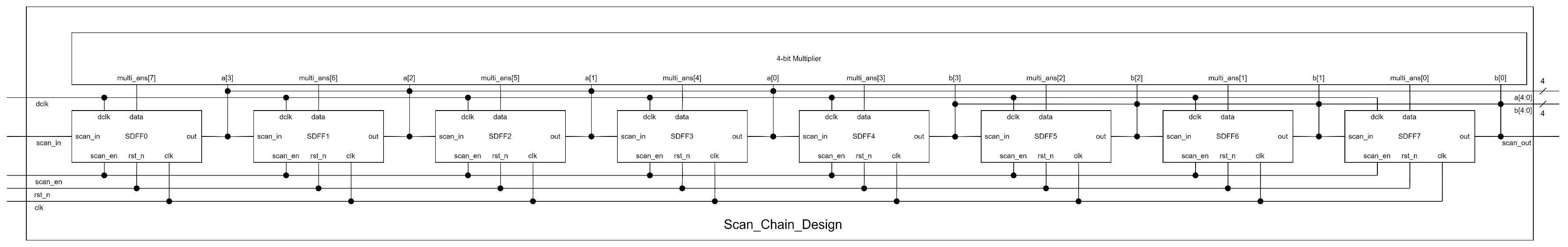
****

Figure 5.14

Scan\_Chain\_Design (Left Part)

1. **Explanation**

According to the specification, I divided the circuit into different modules as **Figure 5.1** shows. In order to work properly on FPGA boards, divided clock is needed. Most of the modules above are modified from the designs of the previous problems by adding a mux which selected by **dclk** before connect into DFFs. However, after testing on the FPGA board, I noticed that there is some chance that the DFFs will go through two states while I only press the button one time. To solve this problem, I use a divided clock to control **Debounce** module as **Figure 5.15** shows. In this way, it seems the condition mentioned above doesn't happen again while I was testing it on the FPGA board.

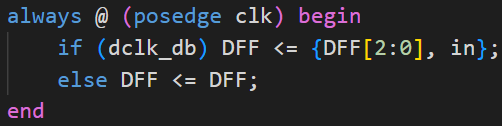


Figure 5.15

1. **What I Have Learned**

Most of modules are provided in the lecture slides, thus I spent much less time designing modules. However, it is difficult to design a testbench in this lab. In order to check the results, I used the method that I've learned from Lab2 and also wrote a C++ program to generate testcases correctly to help me debug. In this lab, I've learned that how to combined the skills that I've learned before to solve problems. I hope that I can use these skills more flexible in the future.