Hardware Design Lab 4 Report

Group Number: 30

Members: 112062130 侯佑勳
112062326 孔祥光

Contents:

Basic Questions

1. Many-to-one LFSR

State transition diagram of the DFFs in LFSR for the first ten states after reset is shown in Figure 1.

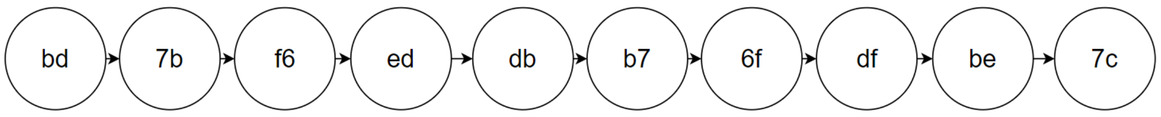


Figure 1

What happens if we reset the DFFs to 8’d0 instead of 8'b10111101?
It would stay in this state because after the next shift, the values of the DFFs would still be 8’b0000_0000. Namely, it sticks in a self-loop.

2. One-to-many LFSR

State transition diagram of the DFFs in LFSR for the first ten states after reset is shown in Figure 2.

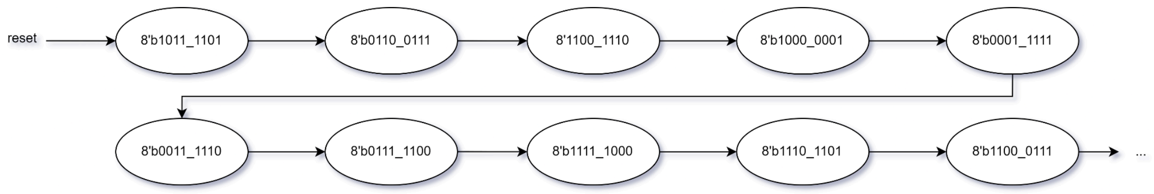


Figure 2

What happens if we reset the DFFs to 8’d0 instead of 8'b10111101?
It would stay in this state because after the next shift, the values of the DFFs would still be 8’b0000_0000.

Design & Testing

1. Content-addressable memory (CAM)
   - input: *clk, ren, wen, addr[3:0], din[7:0]*
     - *addr* is for writing
     - din is for writing or matching
   - output: *dout[3:0], hit*

Design

In this design, we'll take an hierarchical approach, combining four 4x8bit CAMs to form a 16x8bit CAM. The key components in this design are **4x2 priority encoder** and **8-bit equal comparator**.

The priority encoder is simply implemented by some MUXes, as depicted in Figure 3-1. The *dont_care* constant in our code is also set to 2'b00, but its connotation is different from the other option.

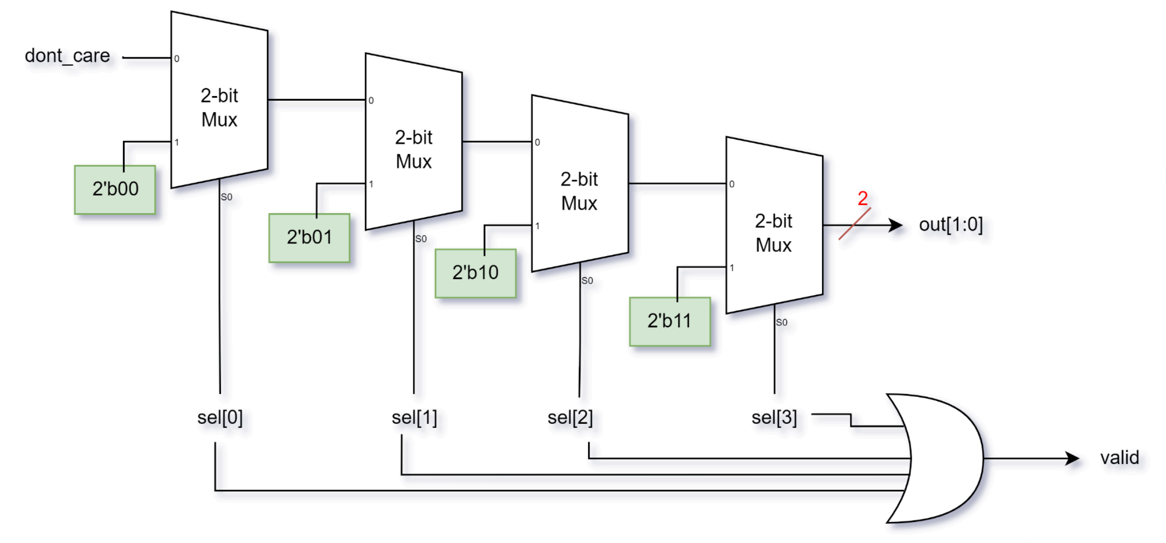


Figure 3-1 4x2 priority encoder

The select signals will be generated from an array of 8-bit comparators, determining whether the content in each memory cell equals to *din*. See Figure 3-2.

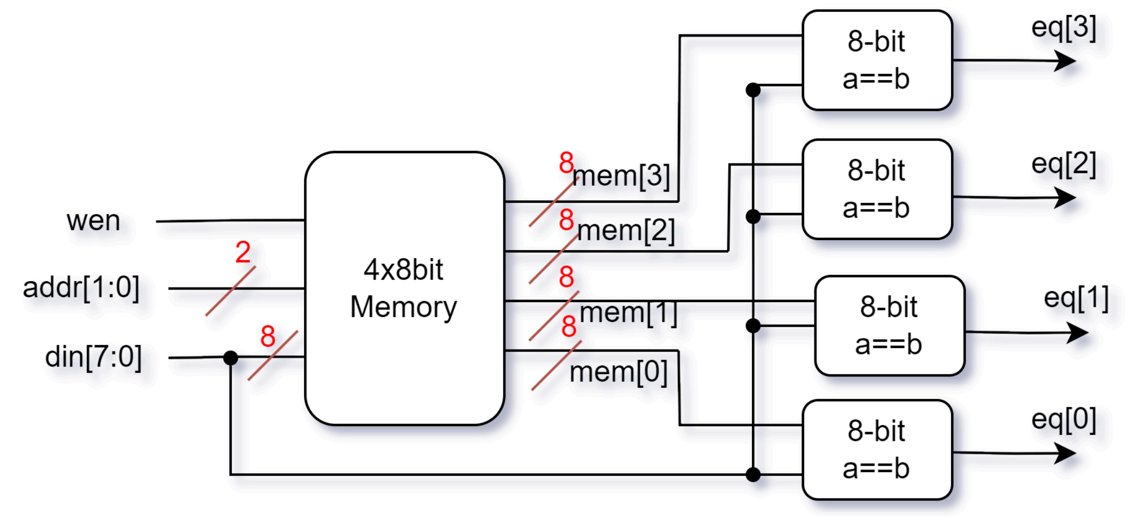


Figure 3-2 Memory block & comparator array

Putting them together, we'll obtain our 4x8bit CAM. Note that the data from the memory is constantly flowing out, but the result address (*out[1:0]*) will only be read into DFFs when *ren* is high and at a positive clock edge.

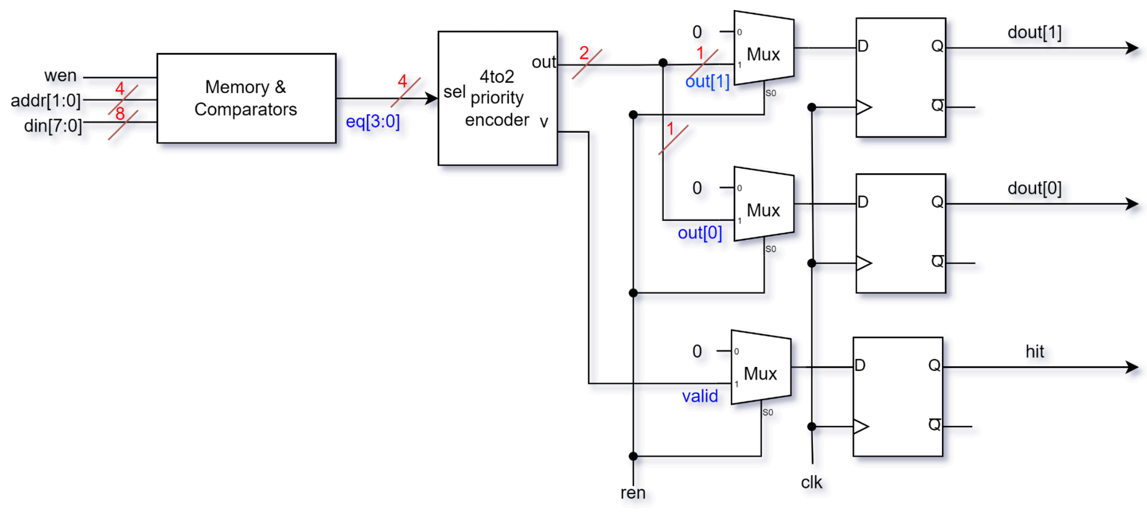


Figure 3-3 4x8bit CAM

Also, we have to slightly modify the priority encoder from Figure 3-1, in order to select outputs from the sub-CAMs instead of the four constants (2'b00~2'b11), shown in Figure 3-4.

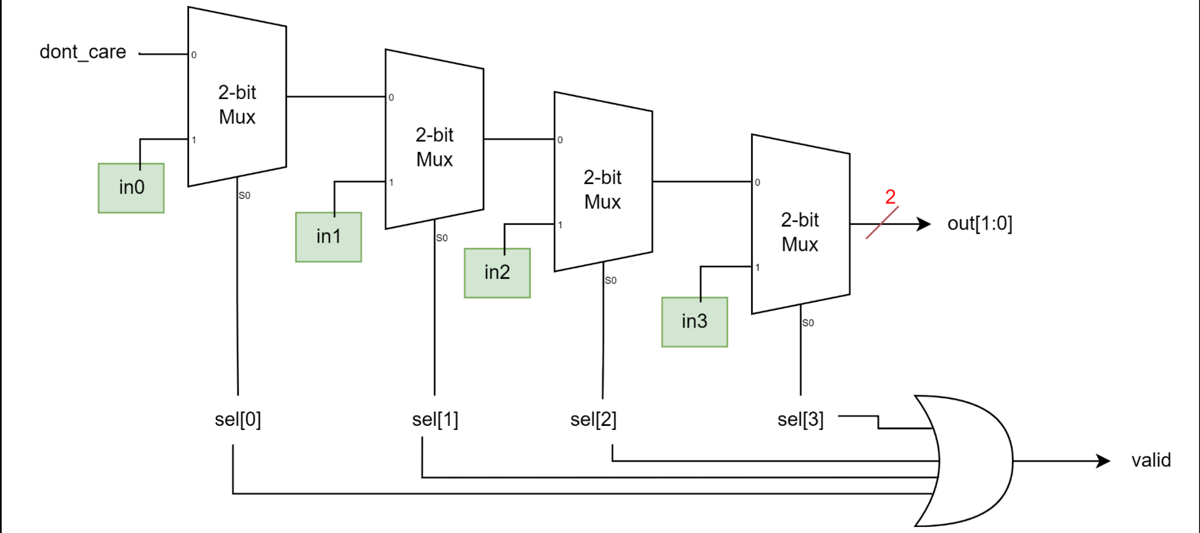


Figure 3-4 4x2 priority encoder with custom encoding

Finally, we compose four of those to form the desired module. The output address *dout[3:2]* consists of two parts: *dout[3:2]* is the address of sub-CAM (2b'00~2b'11) ; *dout[1:0]* is the address of memory cells selected from within the sub-CAM. For instance, if we matched data at 4'b1101, it means we've hit CAM[3] and its mem[1].

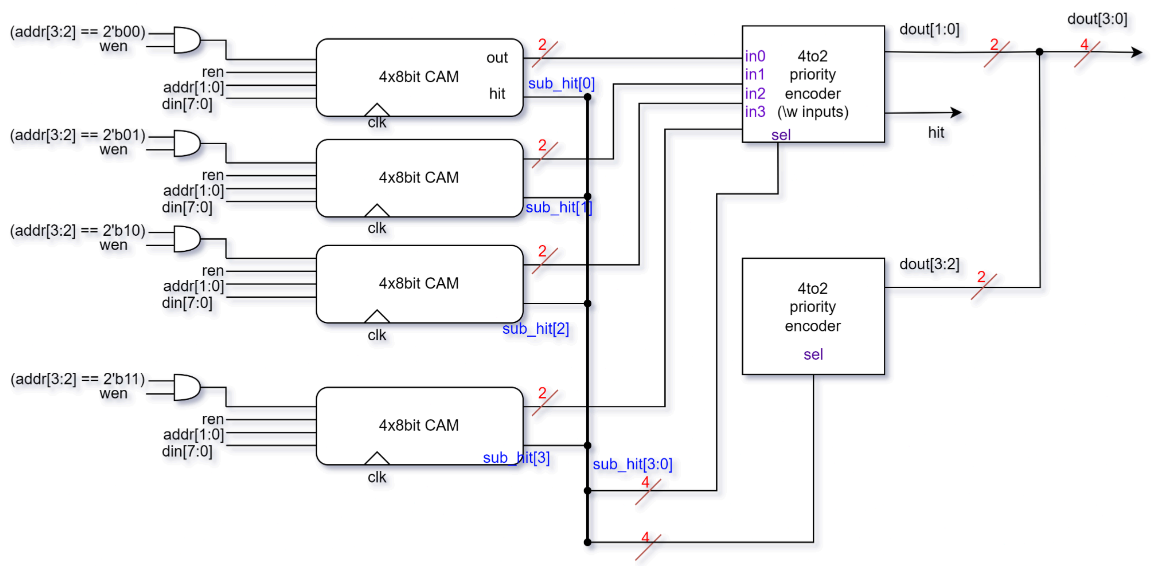


Figure 3-5 16x8bit CAM

Testing

In the testbench we verify whether the following criteria are met:

1) **Read/Write** operations work
2) **Content addressing** works
3) **Priority**: If there are multiple addresses hit, the output should be the largest one

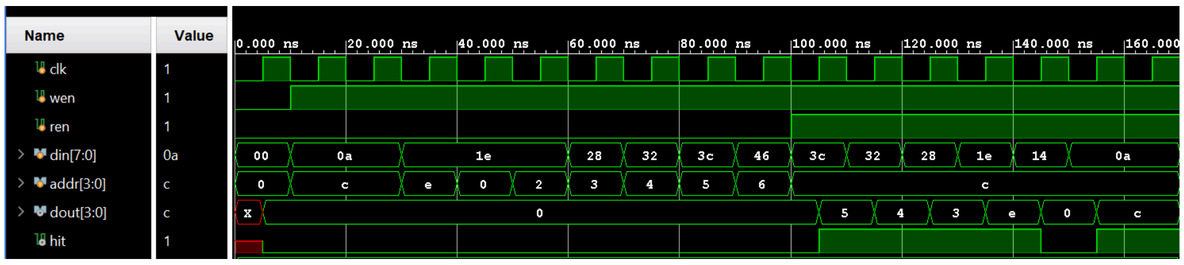


Figure 3-6 Simulation Waveform of 16x8bit CAM

2. Scan Chain
- input: *clk, scan_in, rst_n, scan_en*
- output: *scan_out*

Design

For implementing a scan chain, we first design our Scan DFF, which has two more MUXes compared to the ordinary DFF. If the scan_en is 1, the Scan DFF will transmit the scan_in value, or it will transmit data value.

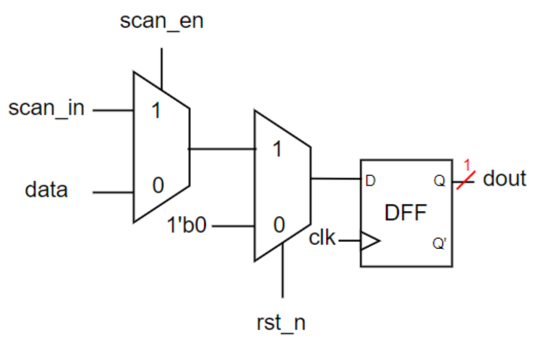


Figure 4-1 Scan DFF

To design a scan chain for a multiplier, we connect the scan_in to the number we want to multiply, and connect data to the product. While the scan_en is 1. We can sequentially receive data one bit at a time. If scan_en becomes 0, we can pass the product into the whole chain, and the chain will sequentially output the product. While outputting the product, we also receive and transmit data for the next calculation.

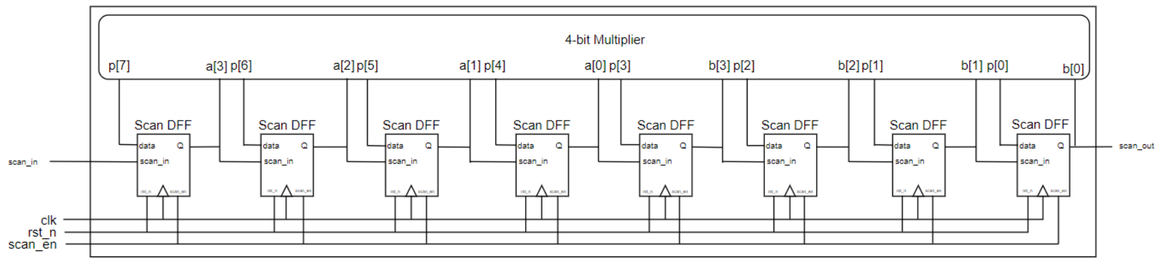


Figure 4-2 Scan Chain

Testing
By passing a[3:0] = 4’0101 and b[3:0] 4’b0101, we can get a product (scan_out) which is 0001 1001. We also pass the scan_in for the next multiplication while there’s a scan_out to prove input won’t influence output at the same time.

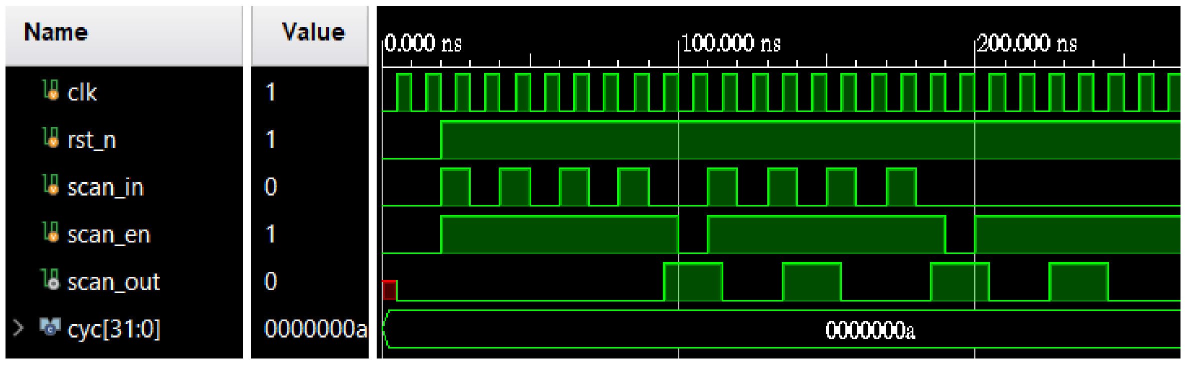


Figure 4-3 Simulation Waveform of Scan Chain

3. Built-in Self Test
    - input: *clk, rst_n, scan_en*
    - output: *scan_in, scan_out*

Design
To connect the Many_To_One_LFSR to the Scan_Chain, we can just take the last DFF’s output as the scan_in for Scan_Chain. The LFSR will generate different output by itself at the positive edge and also change to the next state simultaneously. We also take the scan_in signal to check the data provided by LFSR.

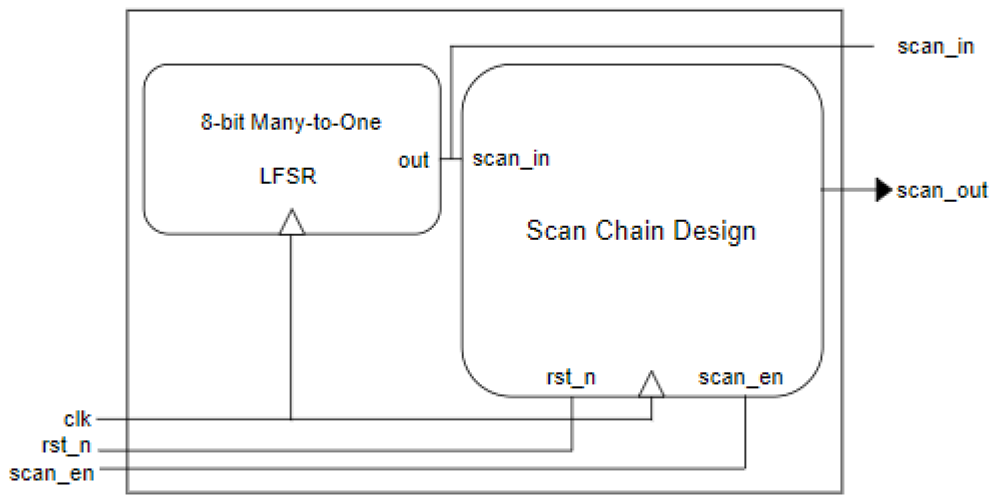


Figure 5-1 Built in Self Test (BIST)

Testing

We set the initial set as 8’b10111101 to the LFSR and let it generate subsequent scan_in. Same as the scan chain’s tb, we give 7 clock cycles to receive data, and then output them.

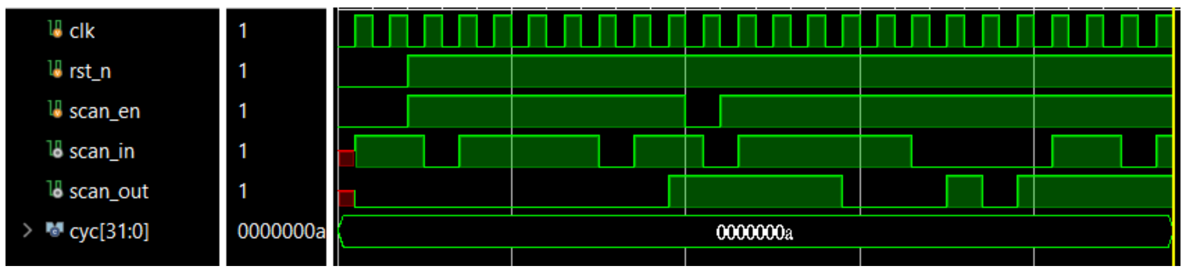


Figure 5-2 Simulation Waveform of BIST

4. Mealy Machine Sequence Detector
   - input: *clk, rst_n, in*
   - output: *dec*

Design

First we draw the state diagram before implementing the Mealy FSM, as shown in Figure 6-1.

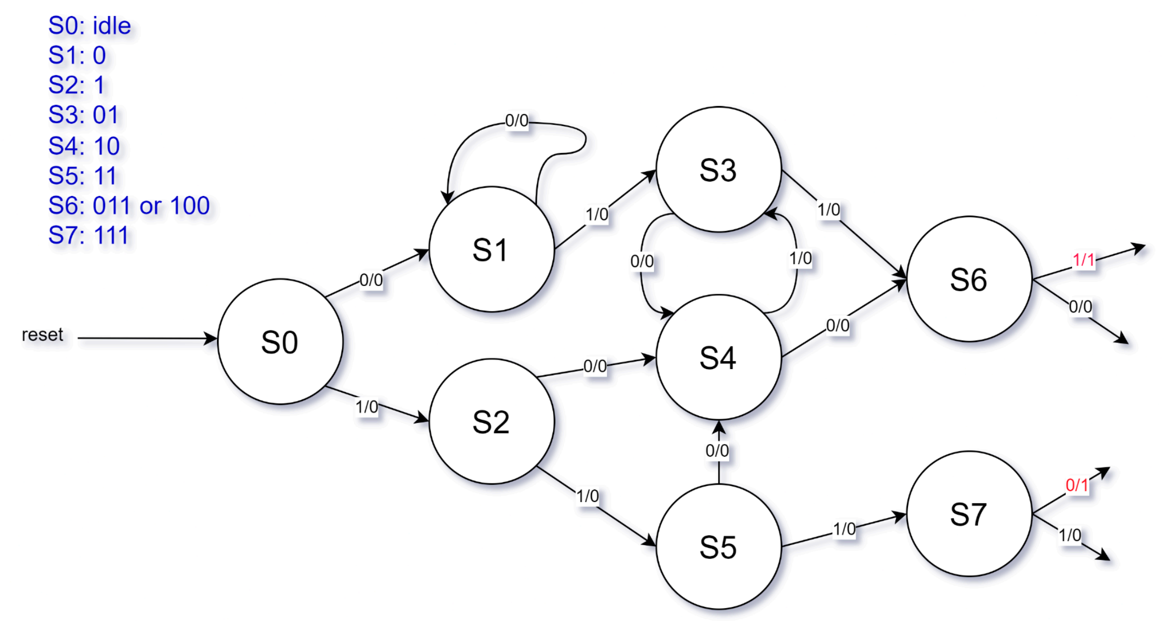


Figure 6-1 State Diagram of Mealy 4-bit Pattern Detector for 0111, 1001, and 1110

Then we encode the states with 3-bit binaries (could be any other encodings as well) like this: S0~S7 corresponds to 3’b000~3’b111, respectively. Since the next state is determined by the current state and input, we’ll use a MUX to generate the *next_state* signal, shown in Figure 6-2

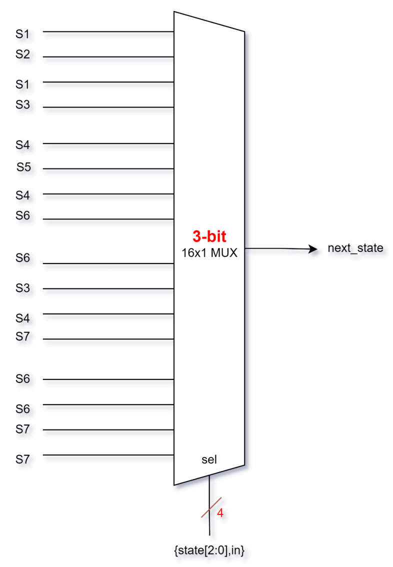


Figure 6-2 Comb. Circuit for Determining the *next_state*

Also, we'll reset the detection every 4 bits, so a 2-bit counter is needed. Finally put everything together and assemble the FSM.

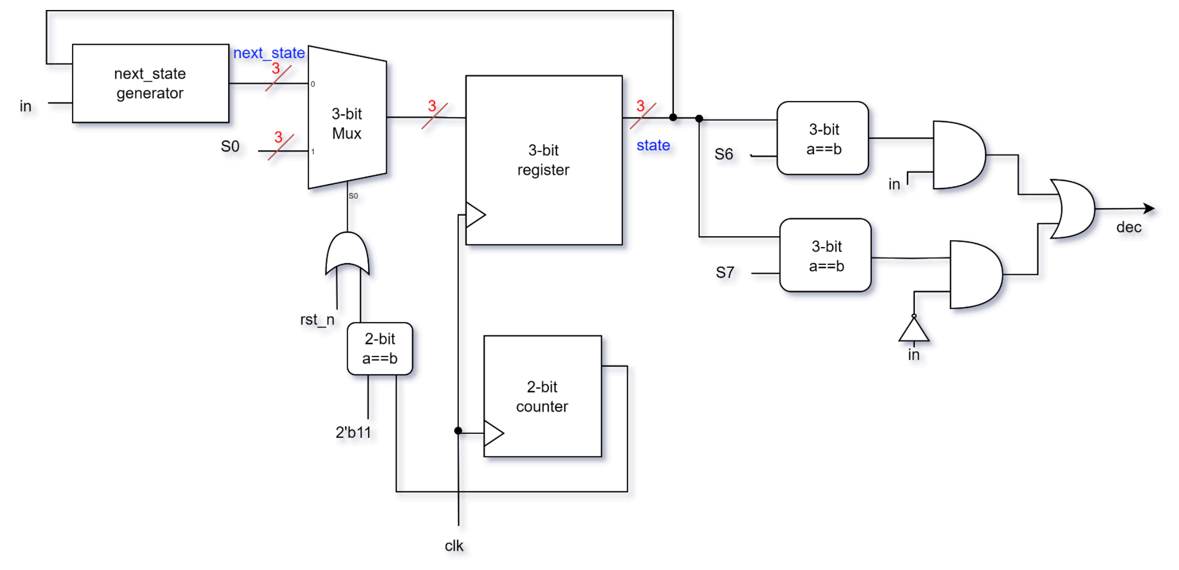
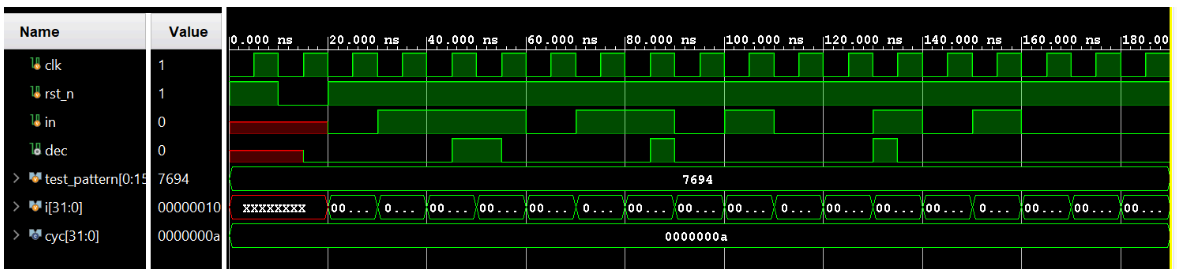


Figure 6-3 Mealy Machine

Testing

For the verification of this Mealy Machine, we check whether it correctly detects the patterns 0111, 1001, and 1110 and whether it actually follows our state transition diagram.



Name	Value
clk	1
rst_n	1
in	0
dec	0
test_pattern[0:15]	7694
i[31:0]	00000010
cyc[31:0]	0000000a

Figure 6-4 Simulation Waveform of Mealy Pattern Detector

FPGA

Built-in Self Test on FPGA

(*clk for SDFF and LFSR is triggered by d_clk button)

To implement BiST on the FPGA, we design some submodules to handle different functions. Similar to Lab3, we also use debounce (using 16 DFFs) and onepulse to stabilize button inputs, and a clock_divider for time-multiplexing the 7-seg display. The clock_divider produces a clk_2_17 signal (which is about 381 Hz) for concurrent digit display.

In this design, we modified the modules from the previous question. First, we use d_clk to enable SDFF, just like the picture below.

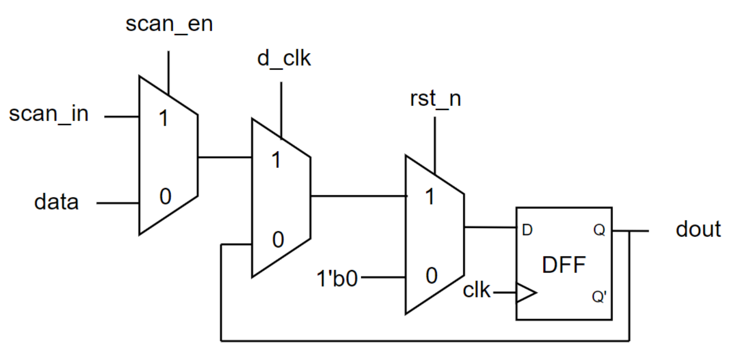


Figure 7-1 Scan DFF for d_clk triggered

As for the scan chain, we add outputs a and b to record the a , b status.

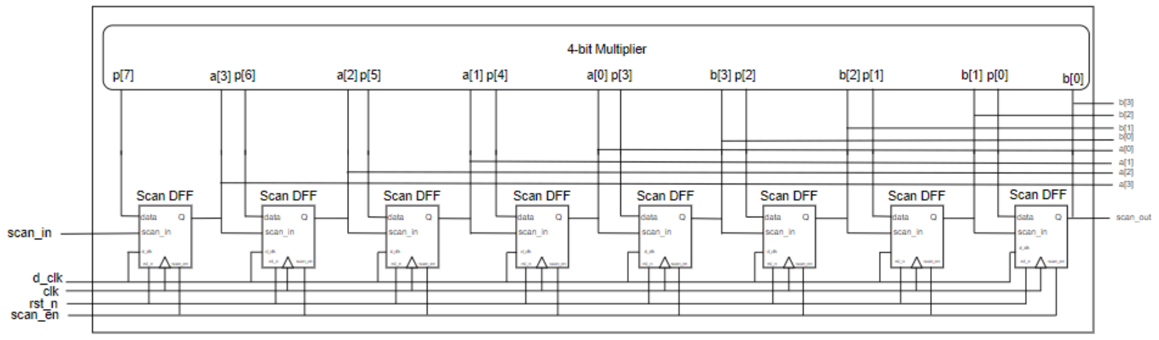


Figure 7-2 Scan Chain for FPGA

For LFSR, we can assign an initial_state while rst_n is zero by using MUXes, and the module’s cycle also depends on *d_clk*.

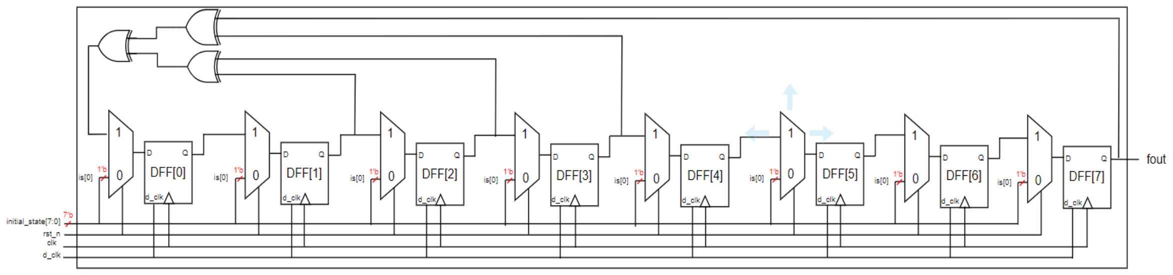


Figure 7-3 LFSR for FPGA

By using the components above, we can construct a new BIST for FPGA.

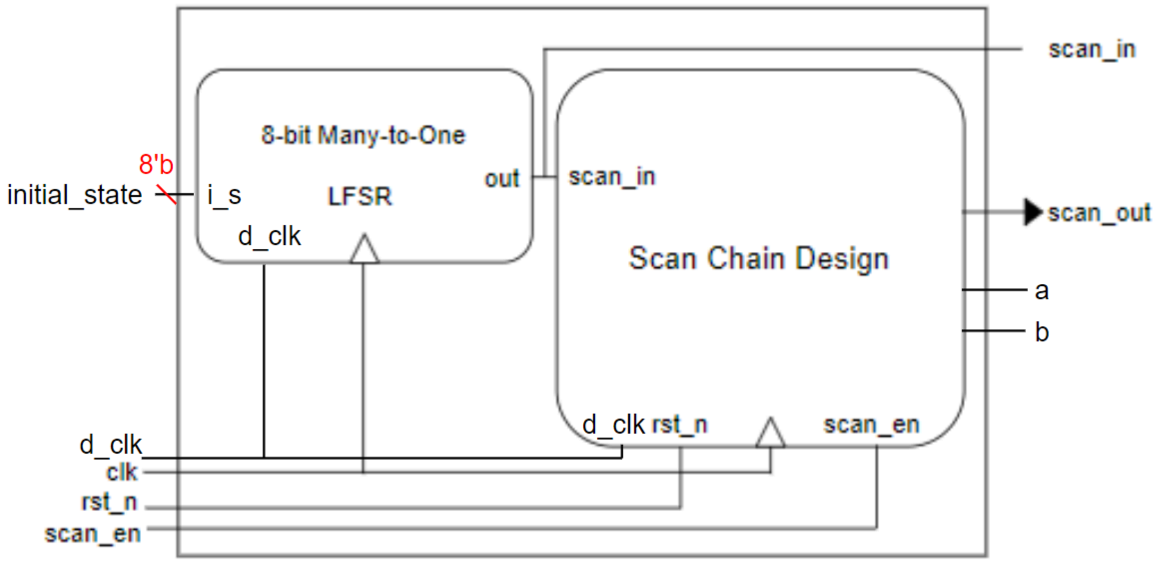


Figure 7-4 Built in Self Test for FPGA

For the 7 segments display mechanism, we use the signal a or b as the selection signal for the16 to 1 multiplexer to determine which number to display (from 0 ~ F) and store it into aLED or bLED.

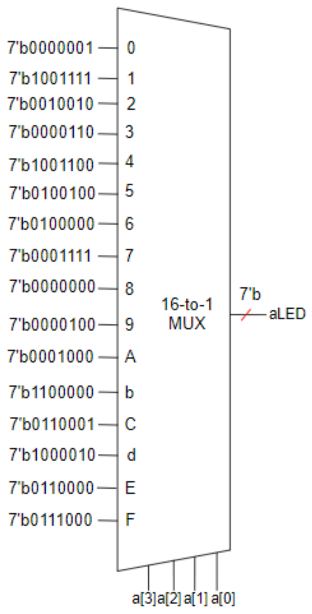


Figure 7-5 16-to-1 MUX for 7 segments display

We also use the signal generated from a four state moore machine (which changes state according to the d_clk signal) to determine which digit to display.

Finally, we connect all the submodules together to implement this FPGA module.

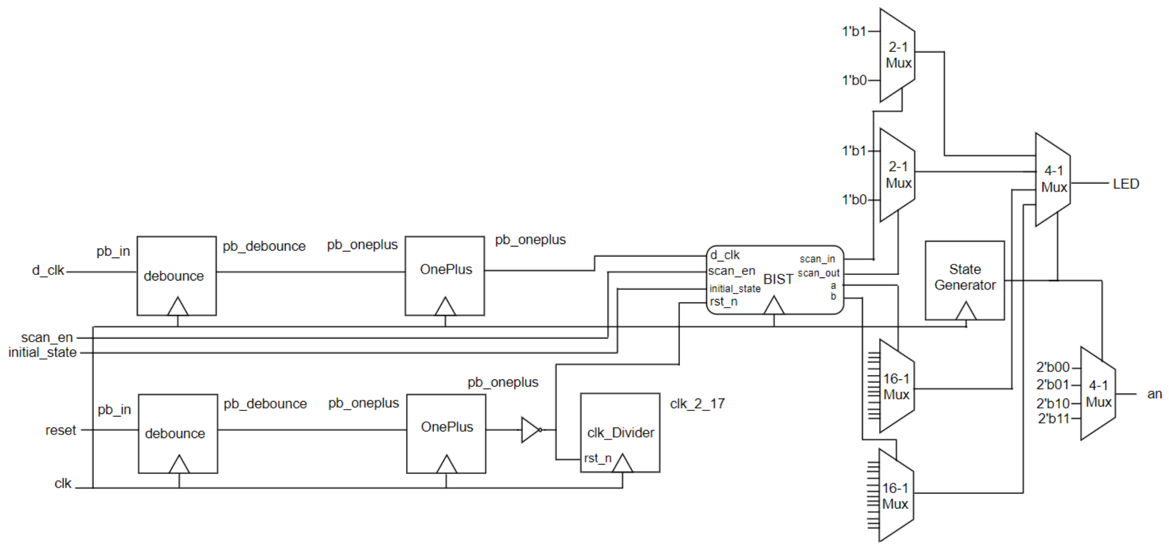


Figure 7-6 16-to-1 MUX for 7 segments display

What Have We Learned?

112062130 侯佑勳

It’s a great chance for me to implement the code on the FPGA. There were lots of different things you need to notice like how to use the divided clk signals, how to handle button press signals and how to display 4 digits concurrently. This Lab also taught me to imagine the whole circuit or block diagram before writing code. After the concept is thought through, writing Verilog is just connecting and integrating the submodules together!

112062326 孔祥光

In this lab I’ve delved deeper into the design methodology of sequential circuits, especially in the part of designing a Mealy state machine. Based on the experience learnt from previous practices, I now sketch out the whole module, submodules used, and the state diagram before writing any line of code. Also, I’ve managed to follow a clean coding style as strictly as possible. For future reports, we might need to learn LaTeX so that writing the report, especially the formatting and typesetting, wouldn’t be such a hassle as it is.

Contribution

112062130 侯佑勳

- Scan Chain & tb
- BIST & tb
- BIST on FPGA

112062326 孔祥光

- CAM & tb
- Mealy Sequence Detector & tb