Hardware Design

Lab 4 Report

Finite State Machines

Team 01

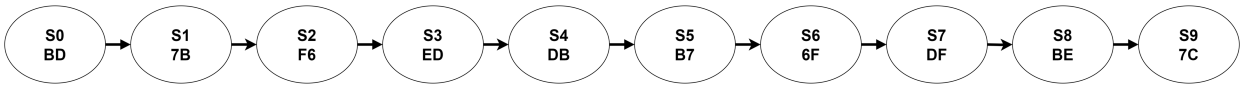
112062122 謝佳晉　112062144 范升維

Table of Contents:

Basic Q3. Many-to-One Linear-Feedback Shift Register (LFSR)

An LFSR is a shift register whose input bit is a linear function (usually XOR) of its previous state. They can be used in digital systems for: pseudo-random number generation, digital counters, cryptography, and so on.

1. State transition diagram:

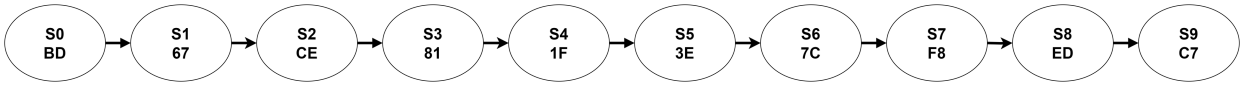


2. What happens if we reset the DFFs to *8'd0*?

   If we reset the DFFs to *8'd0*, the LFSR will keep its state forever. Because there is no *1'b1* in the original state, any of the XOR is impossible to output *1'b1*, so the next state will be the same as the original state. In consequence, the LFSR will keep on shifting, but the states will all be the same, as it's stopped.

Basic Q4. One-to-Many Linear-Feedback Shift Register (LFSR)

1. State transition diagram:

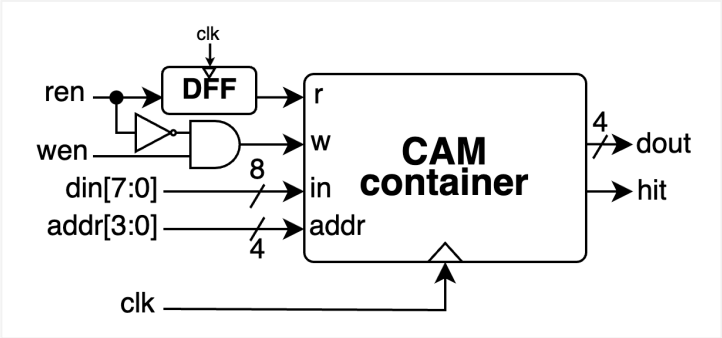


2. What happens if we reset the DFFs to *8'd0*?

   Just like the previous problem, because there is no *1'b1* in the LFSR and we only use XOR operations in the LFSR, the input of all the DFFs will keep being *1'b0*. Therefore, the LFSR will also keep on shifting, but the states will all be the same, as it's stopped, until we change the reset value.

Advanced Q1. Content-Addressable Memory (CAM) Design

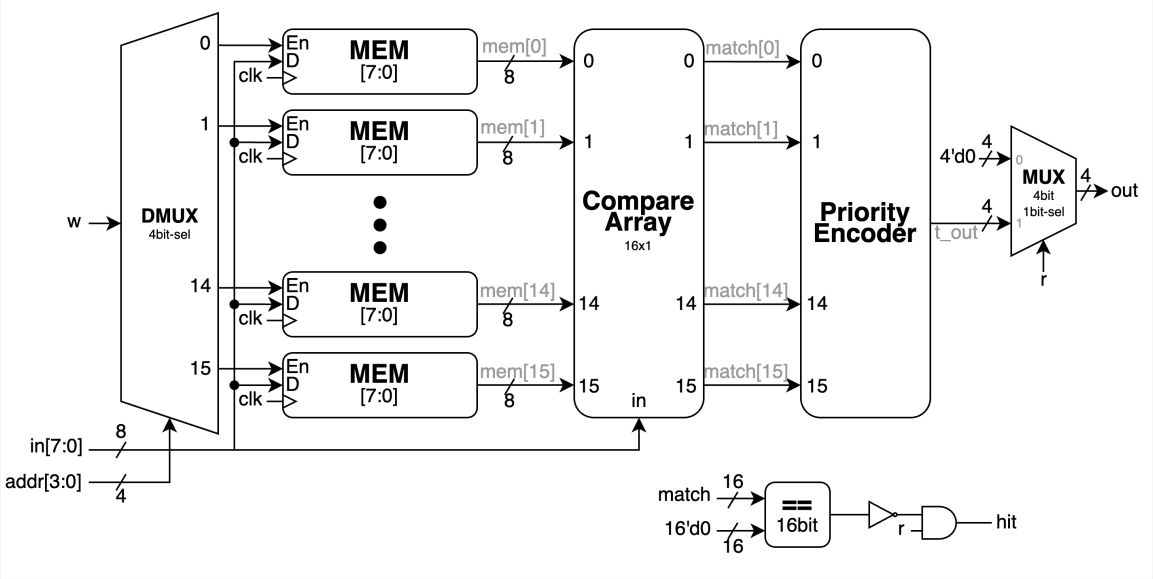
In advanced Q1, we were instructed to create a Content-addressable memory (CAM) design. The search mechanism compares the input data (*din*) with each Stored Data Line in the CAM, returning the address where a match is found. This feels like the reverse of traditional memory access where addresses are used to retrieve stored values.



▲ Figure 1.1: Top Module

- 4 inputs: *clk, wen, ren, addr[3:0]*
- 2 outputs: *hit, dout[3:0]*

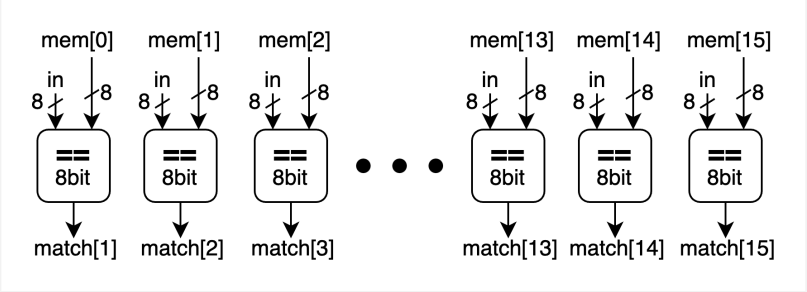
First, let's examine the top module in Figure 1.1. Its main function is to convert *ren* and *wen* into *r* and *w*, ensuring that *r* and *w* cannot be *1'b1* simultaneously. Additionally, the input *ren* passes through a DFF because, as we'll see later in our **CAM container**, it serves directly as the MUX selector. Unlike *w*, which will go through a DFF clock inside the container, we need to add a DFF here to prevent *r* from changing between clock cycles.



▲ Figure 1.2: CAM container

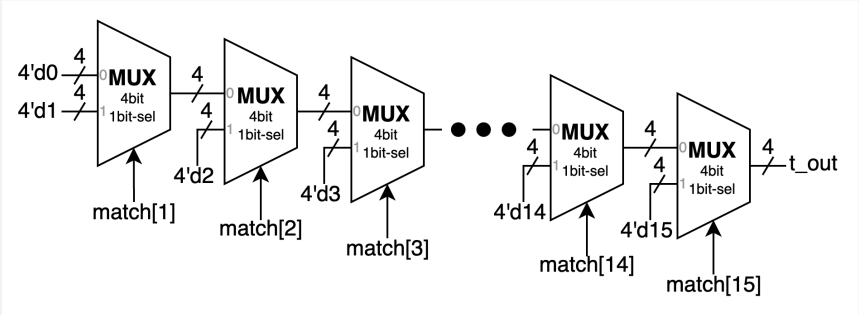
In Figure 1.2, which shows the CAM container, when *w* is *1b'1*, indicating a write operation, the *MEM En* pointed to by *addr* becomes *1'b1*, allowing it to read in. All other *MEM En* remain unenabled, so their values remain unchanged.

Before we discuss *hit* and *dout*, let's first examine how **Compare Array** and **Priority Encoder** work and what their functions are.



▲ Figure 1.3: Compare Array

Compare Array compares each input (*MEM[i]*) with *in*, and outputs whether they are equal to the corresponding output (*match[i]*).



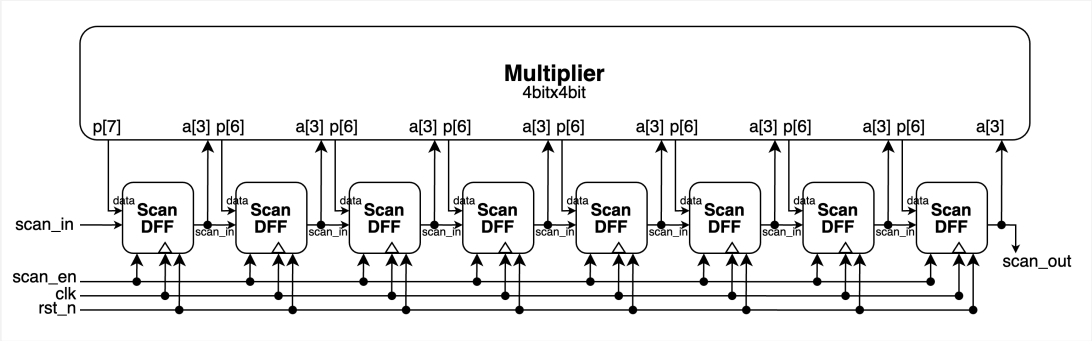
▲ Figure 1.4: Priority Encoder

The Priority Encoder starts checking from the rightmost position. If *match[15]* is *1'b1*, it outputs *4'd15*. If not, it considers *match[14]*. If *match[14]* is *1'b1*, it outputs *4'd14*. If not, it continues checking... This recursive process continues until the last element. If *match[1]* is also false, it directly outputs *4'd0* without considering *match[0]*.

Now that we've covered these two components, we can examine the *hit* and *out* of the CAM container. After examining the Priority Encoder, we can understand the necessity of the *hit* output, as it will output *4'b0* regardless of whether *match[0]* actually matches or not. Therefore, *hit* outputs *1'b0* when none of the matches are successful, and outputs *1'b1* in all other cases. As for *out*, when *r* is enabled, it outputs the result of the Priority Encoder; otherwise, it uniformly outputs *4'b0*.

Advanced Q2. Scan Chain Design

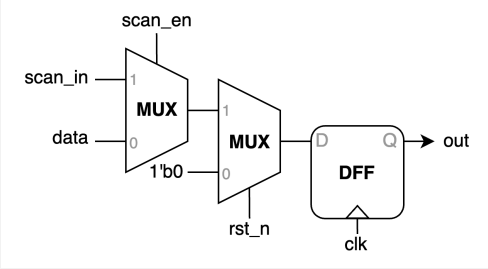
Scan chain is a technique used in design for testing. The objective is to make testing easier by providing a simple way to set and observe every flip-flop in a circuit. The structure of a scan chain is a chain of a special type of DFF, which is called **Scan DFF**. In the question, we are required to build a scan chain for a **4-bit Multiplier**.



▲ Figure 2.1: Top Module

- 4 inputs: *clk, rst_n, scan_en, scan_in*
- 1 output: *scan_out*

To understand how this Top module operates, let's first examine its crucial submodules: the Scan DFF.



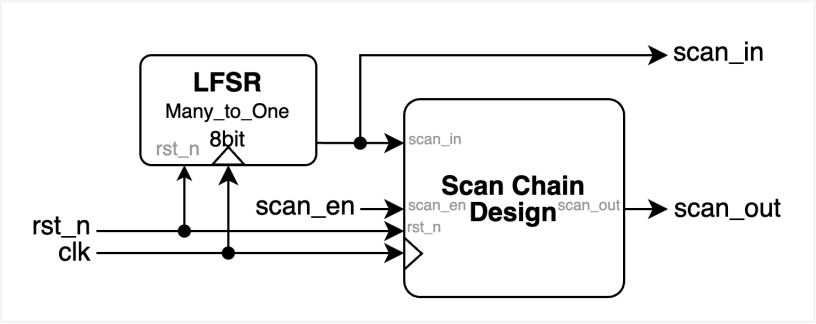
▲ Figure 2.2: Scan DFF

The Scan DFF is built upon the basic DFF with the addition of an *rst_n* signal (which sets the DFF value to *1'b0* when it's *1'b0*), and a *scan_en* control signal. When *scan_en* is enabled, it takes input from *scan_in*; otherwise, it takes input from *data*.

After understanding the I/O and function of the Scan DFF, let's return to the Scan Chain. It connects all SDFF outputs and *scan_in* in series. Therefore, when *scan_en* is *1'b1* at the positive clock edge, the SDFFs perform what is essentially a right shift operation, outputting the value of each SDFF one by one through the overall output (*scan_out*). When *scan_en* is *1'b0* at the positive clock edge, the device under test executes its actual operation, which is multiplying the corresponding *a[3:0]* and *b[3:0]* and storing the result in *p[7:0]*.

Advanced Q3. Built-In Self Test (BIST)

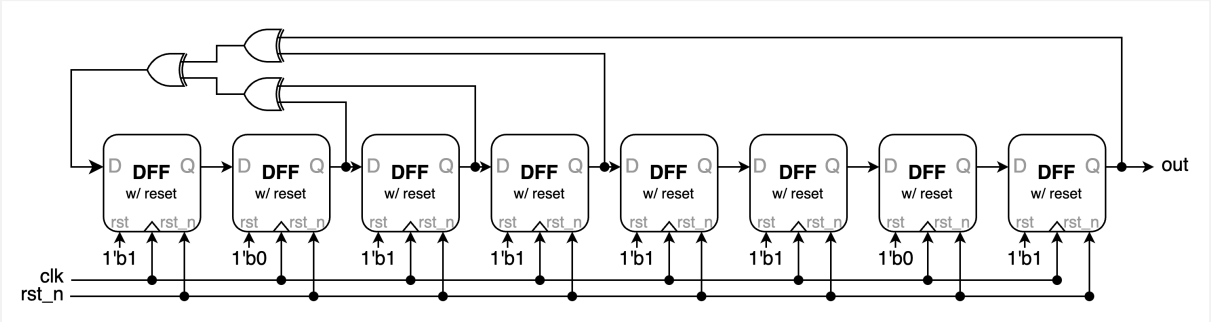
In this question, we are required to construct a Built-In Self Test (BIST) module, namely instead of using actual data input, we utilize a module to generate input for the Scan Chain. In this problem, we use the Many-to-One Linear-Feedback Shift Register that we created in basic question 3 as the module for feeding input data.



▲ Figure 3.1: Top Module

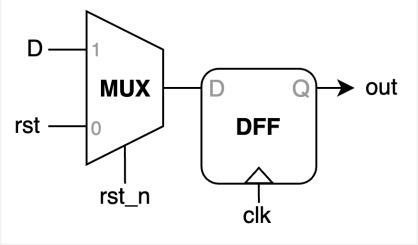
- 3 inputs: *clk, rst_n, scan_en*
- 2 outputs: *scan_in, scan_out*

This is our Top Module, where the Scan Chain Design is the module, we created in the previous question. What's notable here is that *scan_in* is also included as one of the outputs, which allows us to verify whether our module is operating correctly. In actual BIST implementation, this output is not necessary. The circuit diagram of the LFSR we created in basic Q3 is shown below.



▲ Figure 3.2: 8bit Many-to-One LFSR

Figure 3.2 above shows our previously created LFSR, but with a minor modification. Originally, it output all 8 bits, but due to our Scan Chain requirements, it has been modified to output only the MSB (Most Significant Bit) which serves as the *scan_in* input to the Scan Chain. The **DFF with reset** shown in the diagram sets its value to *rst* when both the positive clock cycle occurs and *rst_n* is *1'b0*. By the requirement of the question, we set the *rst* to *8'b10111101*, and its circuit diagram is shown in the figure below.

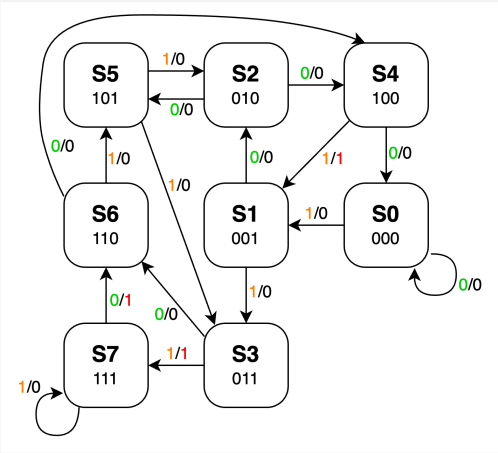


▲ Figure 3.3: DFF with Reset

Advanced Q4. Mealy Machine Sequence Detector

In this problem, we need to create a Mealy Machine sequence detector that differs from the conventional Shift Queue pattern. It must re-detect sequences every 4 clock cycles, meaning the queue needs to be cleared. At each positive clock edge, the input should be stored in the queue with the new input becoming the last digit. The output should be *1'b1* when these 4 digits exactly match any of the specified sequences: *4'b0111*, *4'b100*1, or *4'b1110*; otherwise, it should output *1'b0*.

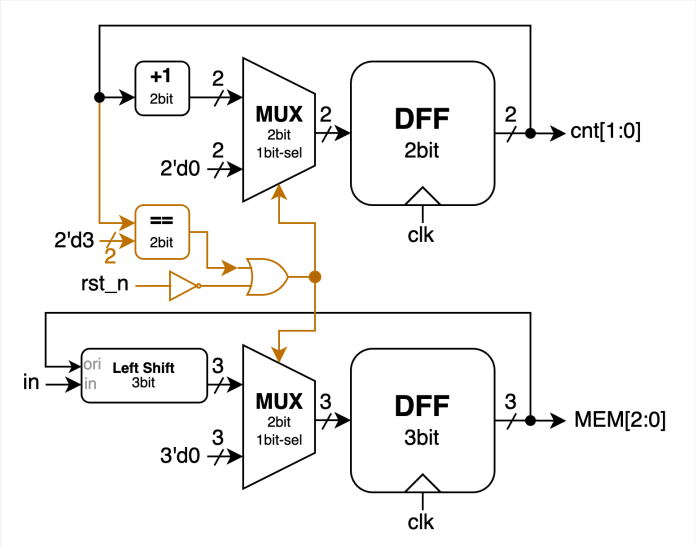
First, let's examine this problem's state diagram. Since the 4-clock-cycle reset is handled separately, this state transition diagram only shows the relationship between the three bits in the queue plus the current input bit, along with their corresponding output values:



▲ Figure 4.1: State Diagram

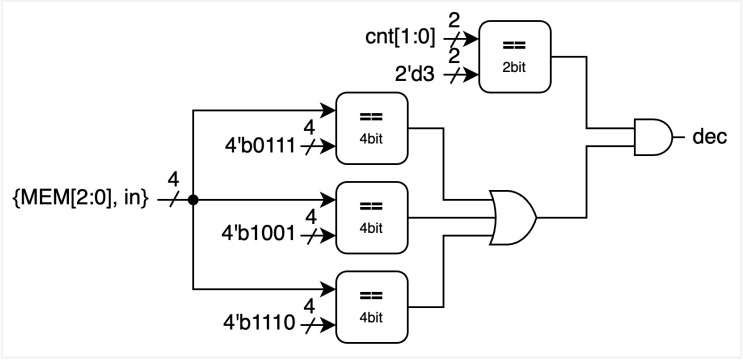
Since this Mealy machine doesn't have irregular transitions but operates similarly to a queue, our implementation is as follows:

- 1 input: *in*
- 1 output: *dec*
- 2 internal signals: *cnt[1:0], MEM[2:0]*



▲ Figure 4.2: generate internal signals

First, we create a reset mechanism that triggers every four clock cycles and a *MEM* to store the three bits. We use a 2-bit counter (DFF) named *cnt* that counts from *2'd0* to *2'd3*, covering four clock cycles. When *cnt* reaches *2'd3* or *rst_n* is triggered (becomes *1'b0*), *cnt* resets to *1'b0* and *MEM* is cleared. Outside of these conditions, MEM performs a left shift by 1 bit at each positive clock edge, with the input bit (LSB) provided by *in*.



▲ Figure 4.3: generating output *dec*

With *cnt* and *MEM* established, we can concatenate *MEM[2:0]* and *in* to create a 4-bit signal for comparison with the given sequences. The results are ORed together, and if any sequence matches when *cnt[1:0]* is *2'd3*, the output becomes *1'b1*. The comparison with *cnt[1:0]* prevents matches involving data in *MEM[2:0]* that wasn't input through *in* (i.e., zeros left by the 4-clock-cycle reset) from causing *dec* to output *1'b1*.

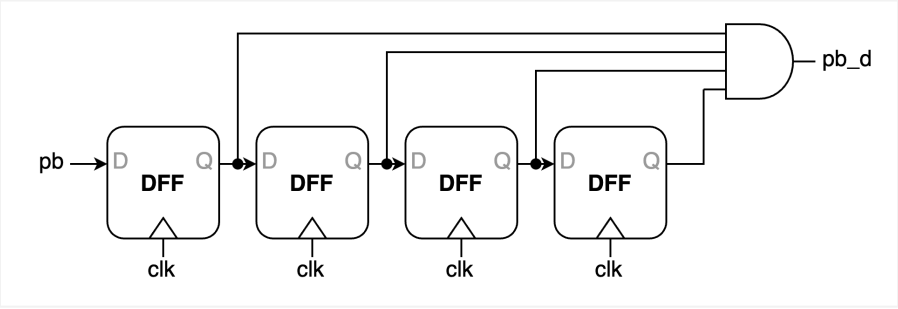
FPGA: Built-In Self Test (BIST)

This Lab's FPGA problem doesn't need brand new concepts to study; it mainly focuses on controlling and displaying the inputs and outputs of our advanced Q3 module through the FPGA board.

- 5 inputs: *clk, d_clk, reset, scan_en, rst_value*
- 4 outputs: *a_out, b_out, AN, seg*

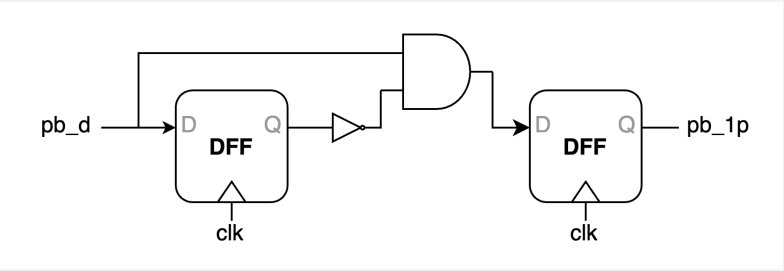
What's unique this time is the *d_clk* signal, which is used to advance our Q3 module to the next clock cycle. Since the built-in clock runs at 100MHz, making it impossible for us to visually observe the Q3 module's operation at this frequency, *d_clk* effectively serves as its clock. We say "effectively" because we've learned that only the system clock can be used as the driving clock signal for modules. We want to pass in an additional signal to act as the clock and use an if statement (which translates to a MUX in the circuit) to control when it drives the module.

This is where we'll apply the Debounce and One Pulse concepts learned in Lab3. Since *d_clk* is triggered by a button press, and button components generate glitches at their edges, we use Debounce to prevent unexpected signals. Debounce ensures that a *1'b1* is only registered when the input remains stable at *1'b1* for 8 clock cycles (the diagram below shows a 4-clock-cycle example), thus filtering out glitches.



▲ Figure 5.1: Debounce

Afterwards, since we want each button press to trigger only once, we need to use One Pulse. Its principle is similar to creating a DFF edge trigger using the *En* of two latches, as shown in the figure below.



▲ Figure 5.2: One Pulse

After handling the button inputs (*reset* and *d_clk*), *scan_en* and *rst_value* are input via switches, with no complex technical details involved. The *rst_value* is the value to be set when the LFSR is reset. For the outputs, *a_out* and *b_out*, which are the Multiplier's *a* and *b* values in the BIST, are displayed using LEDs for easy verification.

Regarding the LED display, the specification requires the first and fourth digits to show the states of *scan_in* and *scan_out* respectively, while the second and third digits display the hexadecimal values of *a_out* and *b_out*.

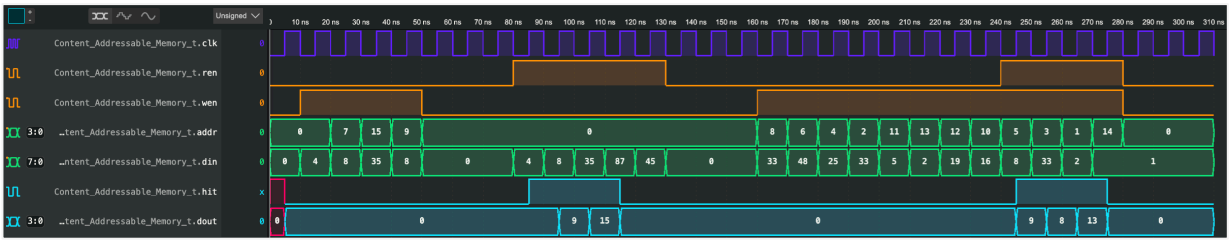
To enable all four digits of the LED display to function, we need to create a counter generating values from *2'b0* to *2'b3* at an appropriate frequency. We implement this as a 19-bit counter, with its two most significant bits serving as the *2'b0* to *2'b3* sequence for cycling through the LED display screens.

The counter is then used as the selector for a 4-to-1 MUX to sequentially display different information on each of the four screens. The display content is converted using a separate BCD_to_seg module that maps 4-bit numbers to corresponding *seg[6:0]* signals, similar to what we did in the Midterm.

With this, we have completed this FPGA implementation problem.

Testbenches

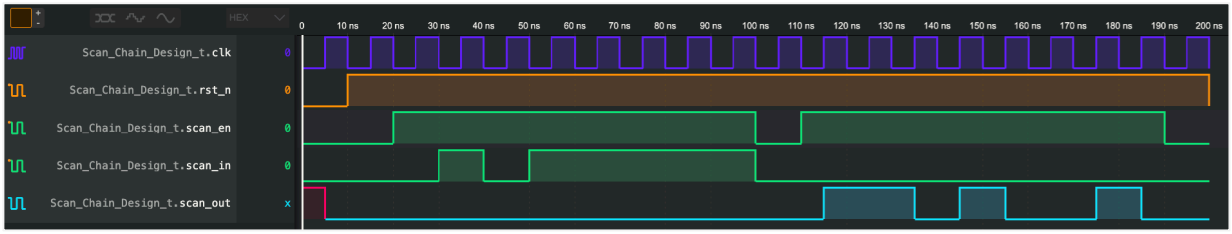
A. Content-Addressable Memory (CAM) Design



In this testbench, we combine the waveform on the spec and some special cases:

1. same data in different stored data lines
2. data matching fails
3. ren and wen both positive simultaneously

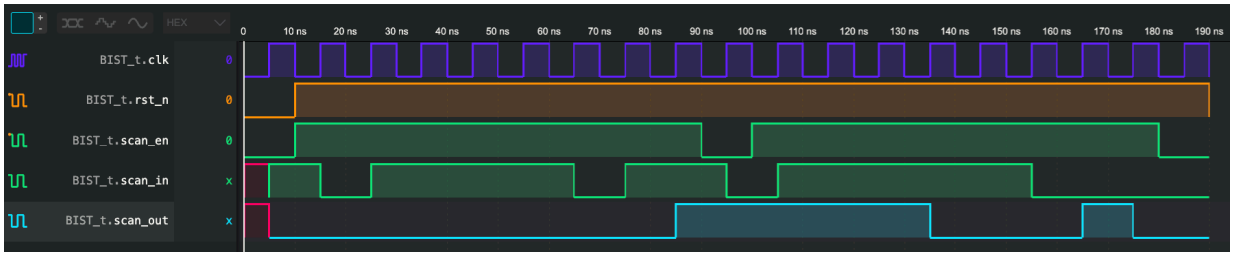
B. Scan Chain Design



In question B, we wrote our testbench fully complying with the input in the spec and receive the same waveform.

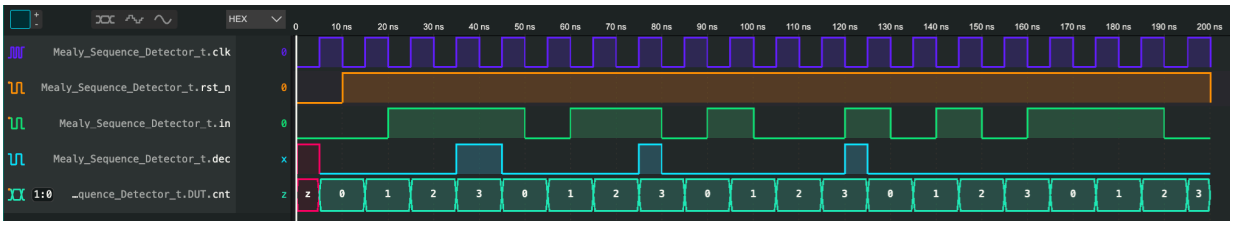
$$a = 4'b1111 = 4'd15$$
$$b = 4'b1010 = 4'd10$$
$$\Rightarrow p = a * b = 8'd150 = 8'b10010110$$

C. Built-In Self Test (BIST)



In this testbench, we modify the *rst_n* and *scan_en* according to the working pattern of Scan Chain Design (reset → scan in → capture → scan out). We can see that the sequence of *scan_in* output is *8'b10111101*, which is the same as the reversed reset value of LFSR.

D. Mealy Machine Sequence Detector



In this testbench, we reference the waveform on the spec and get the exactly waveform.

What have we learned from Lab 4?

In this lab, we learned how to implement Mealy and Moore Machines using Verilog, and we explored several interesting concepts including CAM and BIST. Drawing from our previous lab experience, where we struggled to convert our code into circuit diagrams due to insufficient planning, we adopted a more structured approach this time. We thoroughly planned our design before implementation. Additionally, leveraging the coding style principles learned during the Lab 1-3 Review before midterm - such as proper if-else hierarchical structuring and the limit of only using system clock for module driving - we were able to complete the programming and create the circuit diagram more efficiently (though admittedly, this might be partially due to the more streamlined circuit design).

Furthermore, we encountered a valuable learning point in the first problem. Due to the absence of a reset input, the Compare Array output could potentially produce unexpected results due to residual signals. To address this, we implemented the === operator instead of == in Verilog to filter out 'z' and 'x' signals. It's worth noting that while this approach isn't replicable in physical circuits, it serves as a useful tool for simulation observation purposes.

Contributions

謝佳晉：

wrote Verilog modules: Q1, Q2, Q3, Q4, fpga
wrote testbenches: Q1, Q2, Q3, Q4
performed simulation: Q1, Q2, Q3, Q4
drew diagram: Q3
wrote report: basic Q3, basic Q4, Q3, tb

范升維：

wrote Verilog modules: Q1, Q2, Q3, Q4, fpga
performed simulation: Q1, Q2, Q3, Q4
drew diagram: Q1, Q2, Q3, Q4
wrote report: Q1, Q2, Q3, Q4, tb, what we learned
made FPGA demonstration
organized whole report