Lab4 Report

Department: CS

Student’s Id: 110062271

Name: 林奕為

內容

What I have learned from the lab:

In this lab, I have learned about content addressable memory, scan_chain_design, build_in_self_test and mealy sequence detector, and the FPGA implementation regarding LEDs. I have becoming more familiar with this modules and Verilog also.

Advanced Question 1, CAM design:

How does my modules work:

I designed a total of 3 modules, they are *Comparator_Array*, *Priority* and *Content_Addressable_Memory* respectively.

In CAM module, first there are 16 8-bits stored data line, if write is enabled, din is write into the register, else if read is enabled, dout is wired from Priority and updated.

In *Comparator_Array*, each stored data line is compared to din, if they are equal the output from that line is 1'b1 otherwise 1'b0.

In *Priority,* the output from *Comparator_Array* wired in, and the output is determined by the descending order from line 15 to data line 0. If the data line with higher order is 1'b1, dout would be the address of that data line.

Here's the source code:

```
`timescale 1ns/1ps

module Comparator_Array(din, in0, in1, in2, in3, in4, in5, in6, in7,
in8,
in9, in10, in11, in12, in13, in14, in15, out0, out1, out2, out3, out4,
out5,
out6, out7, out8, out9, out10, out11, out12, out13, out14, out15);

input [7:0] in0, in1, in2, in3, in4, in5, in6, in7, in8,
in9, in10, in11, in12, in13, in14, in15;
input [7:0] din;
output out0, out1, out2, out3, out4, out5,out6, out7, out8,
out9, out10, out11, out12, out13, out14, out15;

assign out0 = (in0 === din) ? 1'b1 : 1'b0;
assign out1 = (in1 === din) ? 1'b1 : 1'b0;
assign out2 = (in2 === din) ? 1'b1 : 1'b0;
```

```
assign out3 = (in3 === din) ? 1'b1 : 1'b0;
assign out4 = (in4 === din) ? 1'b1 : 1'b0;
assign out5 = (in5 === din) ? 1'b1 : 1'b0;
assign out6 = (in6 === din) ? 1'b1 : 1'b0;
assign out7 = (in7 === din) ? 1'b1 : 1'b0;
assign out8 = (in8 === din) ? 1'b1 : 1'b0;
assign out9 = (in9 === din) ? 1'b1 : 1'b0;
assign out10 = (in10 === din) ? 1'b1 : 1'b0;
assign out11 = (in11 === din) ? 1'b1 : 1'b0;
assign out12 = (in12 === din) ? 1'b1 : 1'b0;
assign out13 = (in13 === din) ? 1'b1 : 1'b0;
assign out14 = (in14 === din) ? 1'b1 : 1'b0;
assign out15 = (in15 === din) ? 1'b1 : 1'b0;

endmodule



module Priority(in0, in1, in2, in3, in4, in5, in6, in7, in8,
in9, in10, in11, in12, in13, in14, in15, dout, signal);

input in0, in1, in2, in3, in4, in5, in6, in7, in8,
in9, in10, in11, in12, in13, in14, in15;
output [3:0] dout;
output signal;

assign dout = (in15 & 1'b1) ? 4'd15 :
              (in14 & 1'b1) ? 4'd14 :
              (in13 & 1'b1) ? 4'd13 :
              (in12 & 1'b1) ? 4'd12 :
              (in11 & 1'b1) ? 4'd11 :
              (in10 & 1'b1) ? 4'd10 :
              (in9 & 1'b1) ? 4'd9 :
              (in8 & 1'b1) ? 4'd8 :
              (in7 & 1'b1) ? 4'd7 :
              (in6 & 1'b1) ? 4'd6 :
              (in5 & 1'b1) ? 4'd5 :
              (in4 & 1'b1) ? 4'd4 :
              (in3 & 1'b1) ? 4'd3 :
```

```
            (in2 & 1'b1) ? 4'd2 :
            (in1 & 1'b1) ? 4'd1 :
            (in0 & 1'b1) ? 4'd0 :
            4'd15;


assign signal = in0 | in1 | in2 | in3 | in4 | in5 | in6 | in7 | in8 |
in9 | in10 | in11 | in12 | in13 | in14 | in15;

endmodule



module Content_Addressable_Memory(clk, wen, ren, din, addr, dout, hit);
input clk;
input wen, ren;
input [7:0] din;
input [3:0] addr;
output [3:0] dout;
output reg hit;

reg [3:0] dout;

wire [3:0] out_temp;
wire signal;
reg [7:0] CAM [15:0];

wire [3:0] out0, out1, out2, out3, out4, out5,
out6, out7, out8, out9, out10, out11, out12, out13, out14, out15;

// Comparator_Array(in_data, din, dout);
Comparator_Array C1(din,
CAM[0], CAM[1], CAM[2], CAM[3], CAM[4], CAM[5], CAM[6], CAM[7], CAM[8],
CAM[9], CAM[10], CAM[11], CAM[12], CAM[13], CAM[14], CAM[15],
out0, out1, out2, out3, out4, out5, out6, out7, out8, out9, out10,
out11, out12, out13, out14, out15);

//Priority(in0, in1, in2, in3, in4, in5, in6, in7, in8,
//in9, in10, in11, in12, in13, in14, in15, dout, signal);
```

```
Priority P1(out0, out1, out2, out3, out4, out5, out6, out7, out8, out9,
out10, out11, out12, out13, out14, out15,
out_temp, signal);

wire write_enabled, read_enabled;
assign write_enabled = wen && !ren;
assign read_enabled = (!wen && ren) || (wen && ren);


// CAM logic
always @(posedge clk) begin
    if(write_enabled) begin
        dout <= 1'b0;
        hit <= 1'b0;
        CAM[addr] = din;
    end
    else if(read_enabled) begin
        if(signal == 1'b1) begin
            hit <= 1'b1;
            dout <= out_temp;
        end
        else begin
            hit <= 1'b0;
            dout <= 1'b0;
        end
    end
    else begin // !wen && !ren
        dout <= 1'b0;
        hit <= 1'b0;
    end
end


endmodule
```

How do I design my testbench:

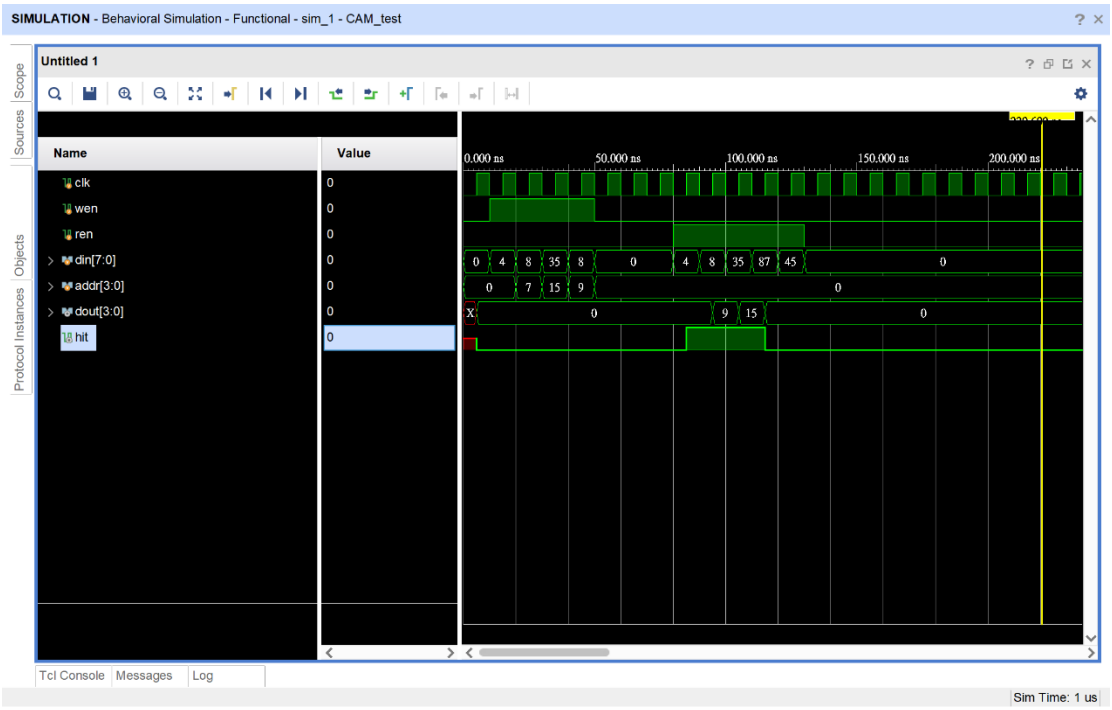
I make my testbench the same as the one in the pdf, and it can test the function of write and read, and the reading of data with multiple occurrences.

Here's the testbench:

```
module CAM_test;
reg clk, wen, ren;
reg [7:0] din;
reg [3:0] addr;
wire [3:0] dout;
wire hit;

// Content_Addressable_Memory(clk, wen, ren, din, addr, dout, hit);
Content_Addressable_Memory D1(clk, wen, ren, din, addr, dout, hit);

always #5 clk = ~clk;

initial begin
    clk = 1'b0;
    ren = 1'b0;
    wen = 1'b0;
    addr = 4'd0;
    din = 8'd0;

    @(negedge clk)
    wen = 1'b1;
    din = 8'd4;
    @(negedge clk)
    addr = 4'd7;
    din = 8'd8;
    @(negedge clk)
    addr = 4'd15;
    din = 8'd35;
    @(negedge clk)
    addr = 4'd9;
    din = 8'd8;
    @(negedge clk)
```

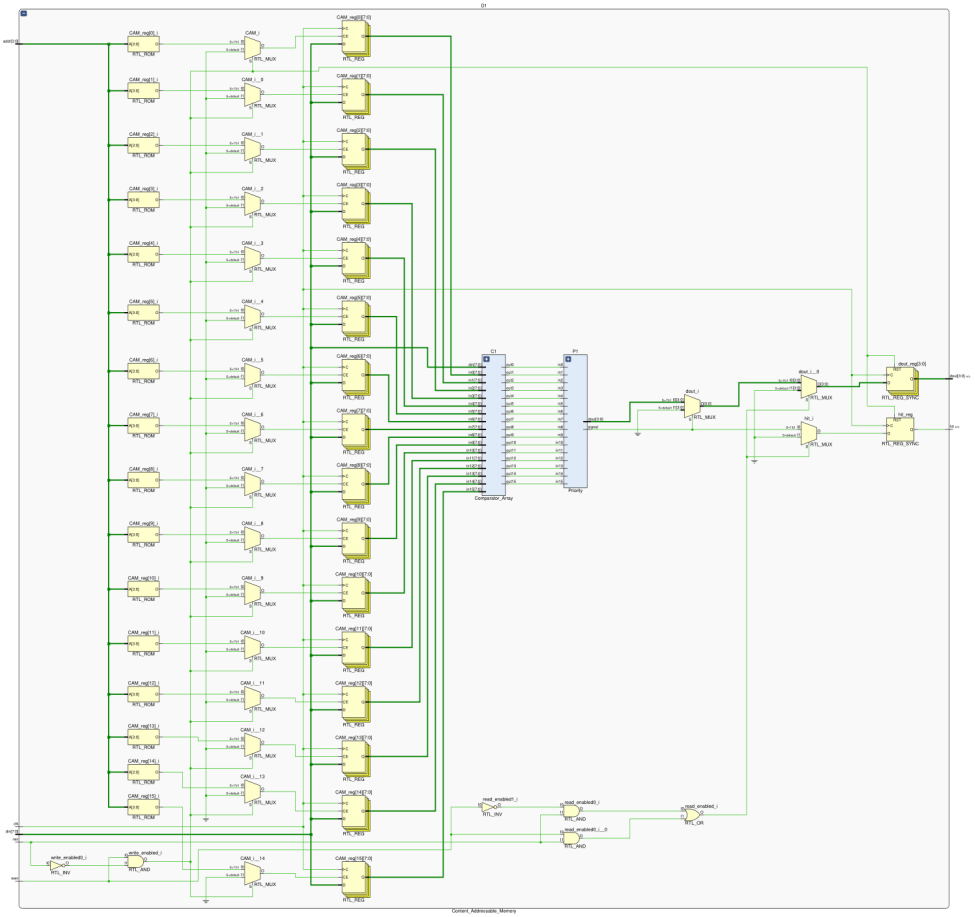
```
    addr = 4'd0;
    din = 8'd0;
    wen = 1'b0;
    @(negedge clk)
    wen = 1'b0;
    @(negedge clk)
    wen = 1'b0;
    @(negedge clk)
    din = 8'd4;
    ren = 1'b1;
    @(negedge clk)
    din = 8'd8;
    ren = 1'b1;
    @(negedge clk)
    din = 8'd35;
    ren = 1'b1;
    @(negedge clk)
    din = 8'd87;
    ren = 1'b1;
    @(negedge clk)
    din = 8'd45;
    ren = 1'b1;
    @(negedge clk)
    din = 8'd0;
    ren = 1'b0;
    @(negedge clk)
    ren = 1'b0;
    @(negedge clk)
    ren = 1'b0;
    @(negedge clk)
    ren = 1'b0;

end
endmodule
```

Here’s the waveform result:

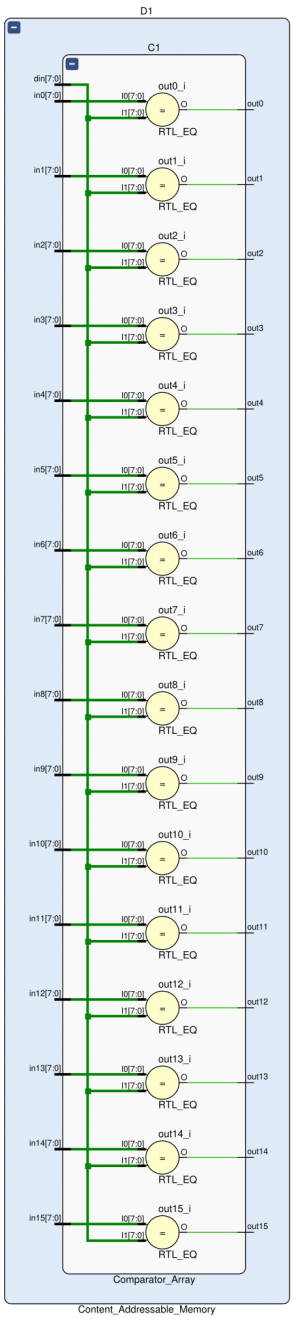


Block Diagram:

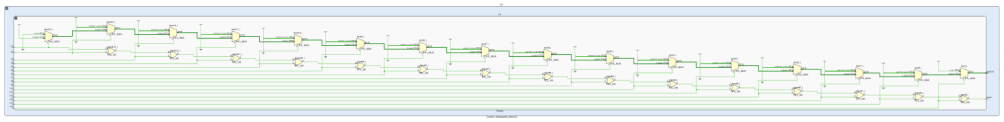
CAM:



Comparator_Array:



Priority:



Advanced question 2, Scan Chain Design:

How does my modules work:

There are 8 SCFF in total, at every posedge clk signal the bit will shifted to right, if the scan_en equals 1'b1 the content would be scan_in otherwise data.

After shifted every bits in, if you turn off scan_en the multiplier will work and stored back the corresponding result to the SCFF, after turning on the scan_en and shifted all the 8 bits out we can get the result.

Here's the source code:

```
`timescale 1ns/1ps

module AND(a, b, out);
input a, b;
output out;
wire temp;

nand n1(temp, a, b);
nand n2(out, temp, temp);
endmodule

module OR(a, b, out);
input a, b;
output out;
wire temp1, temp2;

nand n1(temp1, a, a);
nand n2(temp2, b, b);
nand n3(out, temp1, temp2);
endmodule

module NOT(a, out);
input a;
output out;
```

```
nand n1(out, a, a);
endmodule

module Majority(a, b, c, out);
input a, b, c;
output out;
wire w1, w2, w3, w4;

AND a1(a, b, w1);
AND a2(a, c, w2);
AND a3(b, c, w3);
OR o1(w1, w2, w4);
OR o2(w3, w4, out);

endmodule

module Full_Adder (a, b, cin, cout, sum);
input a, b, cin;
output cout, sum;
wire w1, w2, w3;
wire not_cin, not_cout;

Majority m1(a, b, cin, w1);
AND a1(w1, 1'b1, cout);
NOT n1(cin, not_cin);
NOT n2(w1, not_cout);
Majority m2(a, b, not_cin, w2);
Majority m3(not_cout, cin, w2, sum);

endmodule

module Multiplier_4bit(a, b, p);
input [4-1:0] a, b;
output [8-1:0] p;

wire a0b0, a0b1, a0b2, a0b3, a1b0, a1b1, a1b2, a1b3, a2b0, a2b1, a2b2,
     a2b3, a3b0, a3b1, a3b2, a3b3;
```

```
wire c1,c2,c3,c4,c5,c6,c7,c8,c9,c10,c11;
wire t1,t2,t3,t4,t5,t6;

AND a1(a[0], b[0], p[0]);
AND a2(a[0], b[1], a0b1);
AND a3(a[0], b[2], a0b2);
AND a4(a[0], b[3], a0b3);
AND a5(a[1], b[0], a1b0);
AND a6(a[1], b[1], a1b1);
AND a7(a[1], b[2], a1b2);
AND a8(a[1], b[3], a1b3);
AND a9(a[2], b[0], a2b0);
AND a10(a[2], b[1], a2b1);
AND a11(a[2], b[2], a2b2);
AND a12(a[2], b[3], a2b3);
AND a13(a[3], b[0], a3b0);
AND a14(a[3], b[1], a3b1);
AND a15(a[3], b[2], a3b2);
AND a16(a[3], b[3], a3b3);

// Full_Adder (a, b, cin, cout, sum);

Full_Adder F1(a1b0, a0b1, 1'b0, c1, p[1]);
Full_Adder F2(a2b0, a1b1, c1, c2, t1);
Full_Adder F3(a3b0, a2b1, c2, c3, t2);
Full_Adder F4(a3b1, 1'b0, c3, c4, t3);
Full_Adder F5(t1, a0b2, 1'b0, c5, p[2]);
Full_Adder F6(t2, a1b2, c5, c6, t4);
Full_Adder F7(t3, a2b2, c6, c7, t5);
Full_Adder F8(c4, a3b2, c7, c8, t6);
Full_Adder F9(t4, a0b3, 1'b0, c9, p[3]);
Full_Adder F10(t5, a1b3, c9, c10, p[4]);
Full_Adder F11(t6, a2b3, c10, c11, p[5]);
Full_Adder F12(c8, a3b3, c11, p[7], p[6]);


endmodule
```

```
module SCFF(clk, rst_n, scan_in, data, scan_en, scan_out);
input clk, rst_n, scan_in, data, scan_en;
output reg scan_out;

wire w1, w2;
assign w1 = (scan_en == 1'b1) ? scan_in : data;
assign w2 = (rst_n == 1'b1) ? w1 : 1'b0;

always @(posedge clk) begin
    scan_out <= w2;
end

endmodule


module Scan_Chain_Design(clk, rst_n, scan_in, scan_en, scan_out);
input clk;
input rst_n;
input scan_in;
input scan_en;
output scan_out;

wire out1, out2, out3, out4, out5, out6, out7, out8;
wire [3:0] a, b;
wire [7:0] p;

SCFF s1(clk, rst_n, scan_in, p[7], scan_en, out1);
SCFF s2(clk, rst_n, out1, p[6], scan_en, out2);
SCFF s3(clk, rst_n, out2, p[5], scan_en, out3);
SCFF s4(clk, rst_n, out3, p[4], scan_en, out4);
SCFF s5(clk, rst_n, out4, p[3], scan_en, out5);
SCFF s6(clk, rst_n, out5, p[2], scan_en, out6);
SCFF s7(clk, rst_n, out6, p[1], scan_en, out7);
SCFF s8(clk, rst_n, out7, p[0], scan_en, out8);

assign scan_out = out8;
assign a = {out1,out2,out3,out4};
assign b = {out5,out6,out7,out8};
```

```
Multiplier_4bit M1(a, b, p);


endmodule
```

How Do I Design Testbench:

I test 2 cases, 1010 * 1010 = 01100100 and the other 0011 * 1100 = 0100100.
Both results are correct on the waveform diagram.

Here's the testbench:

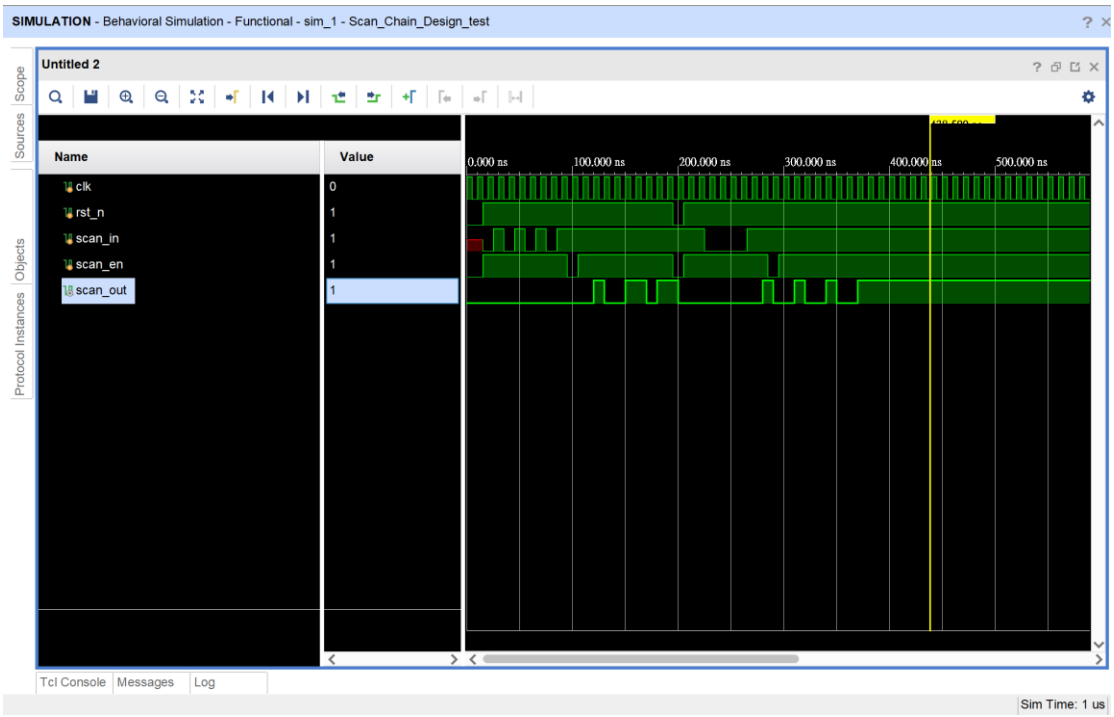
```
`timescale 1ns/1ps

module Scan_Chain_Design_test;

reg clk, rst_n, scan_in, scan_en;
wire scan_out;

Scan_Chain_Design S1(clk, rst_n, scan_in, scan_en, scan_out);

always #5 clk = ~clk;

initial begin
    clk = 1'b1;
    rst_n = 1'b0;
    scan_en = 1'b0;

    // 1010 * 1010 = 01100100 pass

    @(negedge clk)
        rst_n = 1'b0;
    @(negedge clk)
        rst_n = 1'b1;
        scan_en = 1'b1;
        scan_in = 1'b0;
    @(negedge clk)
        rst_n = 1'b1;
        scan_in = 1'b1;
```

```
    @(negedge clk)
        scan_in = 1'b0;
    @(negedge clk)
        scan_in = 1'b1;
    @(negedge clk)
        scan_in = 1'b0;
    @(negedge clk)
        scan_in = 1'b1;
    @(negedge clk)
        scan_in = 1'b0;
    @(negedge clk)
        scan_in = 1'b1;
    @(negedge clk)
        scan_en = 1'b0;
    @(negedge clk)
        scan_en = 1'b1;
    @(negedge clk)
        rst_n = 1'b1;
    @(negedge clk)
        rst_n = 1'b1;
    @(negedge clk)
        rst_n = 1'b1;
    @(negedge clk)
        rst_n = 1'b1;
    @(negedge clk)
        rst_n = 1'b1;
    @(negedge clk)
        rst_n = 1'b1;
    @(negedge clk)
        rst_n = 1'b1;
    @(negedge clk)
        rst_n = 1'b1;
    @(negedge clk)
        rst_n = 1'b1;

    // 0011 * 1100 = 0100100 pass

    @(negedge clk)
        rst_n = 1'b0;
        scan_en = 1'b0;
```

```
    @(negedge clk)
        rst_n = 1'b1;
        scan_en = 1'b1;
        scan_in = 1'b1;
    @(negedge clk)
        rst_n = 1'b1;
        scan_in = 1'b1;
    @(negedge clk)
        scan_in = 1'b0;
    @(negedge clk)
        scan_in = 1'b0;
    @(negedge clk)
        scan_in = 1'b0;
    @(negedge clk)
        scan_in = 1'b0;
    @(negedge clk)
        scan_in = 1'b1;
    @(negedge clk)
        scan_in = 1'b1;
    @(negedge clk)
        scan_en = 1'b0;
    @(negedge clk)
        scan_en = 1'b1;
    @(negedge clk)
        rst_n = 1'b1;
    @(negedge clk)
        rst_n = 1'b1;
    @(negedge clk)
        rst_n = 1'b1;
    @(negedge clk)
        rst_n = 1'b1;
    @(negedge clk)
        rst_n = 1'b1;
    @(negedge clk)
        rst_n = 1'b1;
    @(negedge clk)
        rst_n = 1'b1;
    @(negedge clk)
        rst_n = 1'b1;
    @(negedge clk)
```

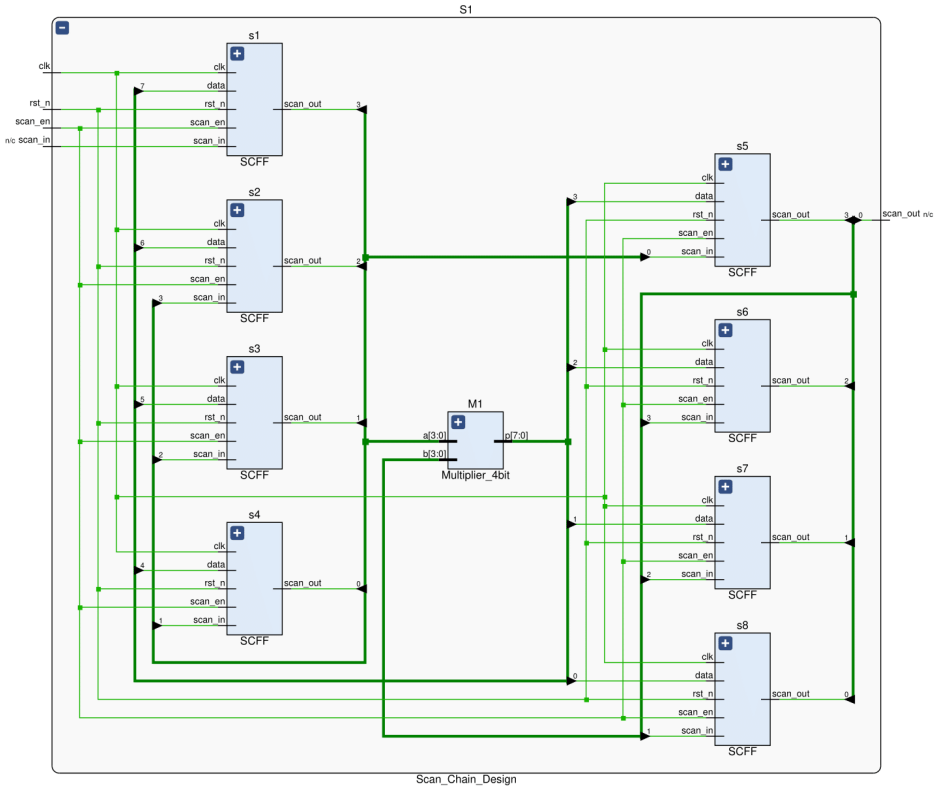
```
        rst_n = 1'b1;

end
endmodule
```

Here’s the waveform diagram:

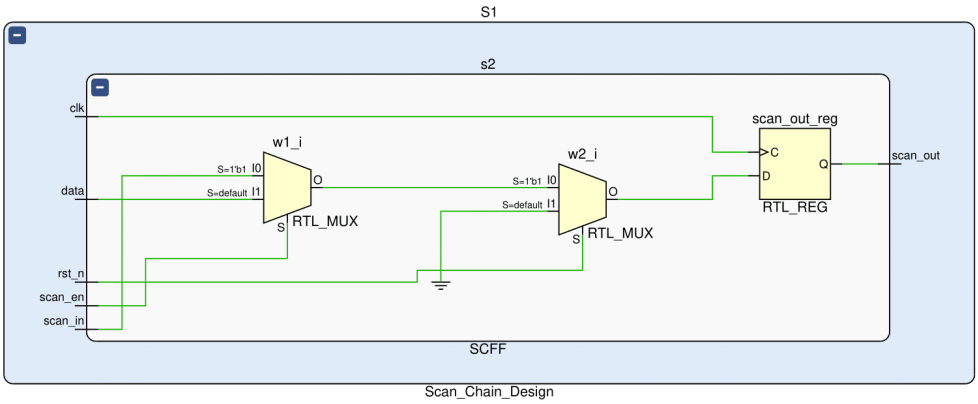


Block Diagram:

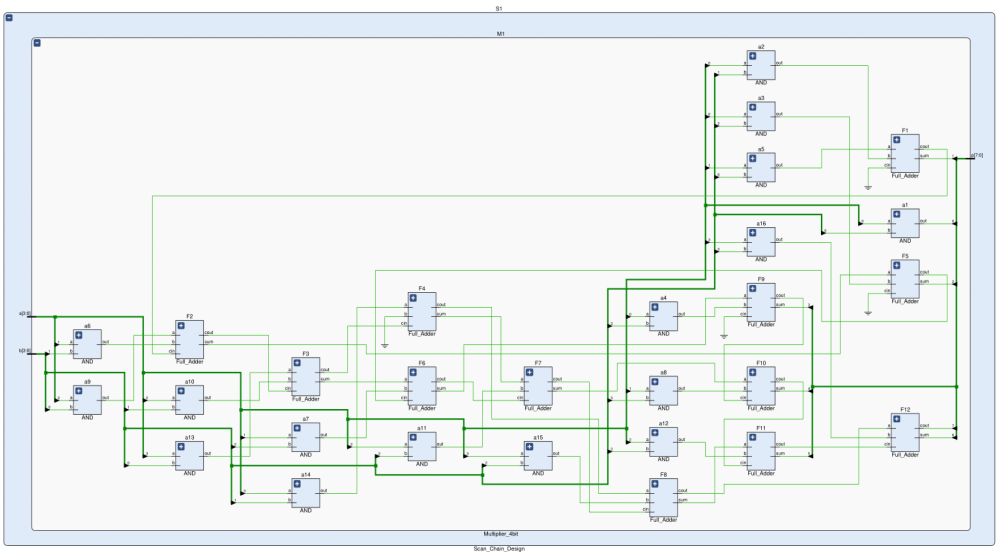
Scan_Chain_Design:



SCFF:


S1
s2
clk
data
rst_n
scan_en
scan_in
w1_i
S=1'b1
I0
S=default
I1
O
S
RTL_MUX
w2_i
S=1'b1
I0
S=default
I1
O
S
RTL_MUX
scan_out_reg
C
D
Q
RTL_REG
scan_out
SCFF
Scan_Chain_Design

Multiplier: (Reuse from the previous lab)



Advanced question 3, Built-in self-test (BIST):

How my modules work:

There are two main modules in BIST, they are LFSR and Scan chain design respectively. Every posedge clk, LFSR will shift for one time and pass one bit into the Scan chain design.

The scan chain design does the same function, after all the bits are scan in, turn off the scan_in and the multiplier combinational circuit will perform its function and store the results in 8 SCFF, after turning back on the scan_en and 8 clock cycles, the result will be manifested using scan_out signal.

Here's the source code:

```
`timescale 1ns/1ps

`timescale 1ns/1ps

module AND(a, b, out);
input a, b;
output out;
wire temp;

nand n1(temp, a, b);
nand n2(out, temp, temp);
endmodule

module OR(a, b, out);
input a, b;
output out;
wire temp1, temp2;

nand n1(temp1, a, a);
nand n2(temp2, b, b);
nand n3(out, temp1, temp2);
```

```
endmodule

module NOT(a, out);
input a;
output out;

nand n1(out, a, a);
endmodule

module Majority(a, b, c, out);
input a, b, c;
output out;
wire w1, w2, w3, w4;

AND a1(a, b, w1);
AND a2(a, c, w2);
AND a3(b, c, w3);
OR o1(w1, w2, w4);
OR o2(w3, w4, out);

endmodule

module Full_Adder (a, b, cin, cout, sum);
input a, b, cin;
output cout, sum;
wire w1, w2, w3;
wire not_cin, not_cout;

Majority m1(a, b, cin, w1);
AND a1(w1, 1'b1, cout);
NOT n1(cin, not_cin);
NOT n2(w1, not_cout);
Majority m2(a, b, not_cin, w2);
Majority m3(not_cout, cin, w2, sum);

endmodule

module Multiplier_4bit(a, b, p);
```

```
input [4-1:0] a, b;
output [8-1:0] p;

wire a0b0, a0b1, a0b2, a0b3, a1b0, a1b1, a1b2, a1b3, a2b0, a2b1, a2b2,
     a2b3, a3b0, a3b1, a3b2, a3b3;

wire c1,c2,c3,c4,c5,c6,c7,c8,c9,c10,c11;
wire t1,t2,t3,t4,t5,t6;

AND a1(a[0], b[0], p[0]);
AND a2(a[0], b[1], a0b1);
AND a3(a[0], b[2], a0b2);
AND a4(a[0], b[3], a0b3);
AND a5(a[1], b[0], a1b0);
AND a6(a[1], b[1], a1b1);
AND a7(a[1], b[2], a1b2);
AND a8(a[1], b[3], a1b3);
AND a9(a[2], b[0], a2b0);
AND a10(a[2], b[1], a2b1);
AND a11(a[2], b[2], a2b2);
AND a12(a[2], b[3], a2b3);
AND a13(a[3], b[0], a3b0);
AND a14(a[3], b[1], a3b1);
AND a15(a[3], b[2], a3b2);
AND a16(a[3], b[3], a3b3);

// Full_Adder (a, b, cin, cout, sum);

Full_Adder F1(a1b0, a0b1, 1'b0, c1, p[1]);
Full_Adder F2(a2b0, a1b1, c1, c2, t1);
Full_Adder F3(a3b0, a2b1, c2, c3, t2);
Full_Adder F4(a3b1, 1'b0, c3, c4, t3);
Full_Adder F5(t1, a0b2, 1'b0, c5, p[2]);
Full_Adder F6(t2, a1b2, c5, c6, t4);
Full_Adder F7(t3, a2b2, c6, c7, t5);
Full_Adder F8(c4, a3b2, c7, c8, t6);
Full_Adder F9(t4, a0b3, 1'b0, c9, p[3]);
Full_Adder F10(t5, a1b3, c9, c10, p[4]);
```

```
Full_Adder F11(t6, a2b3, c10, c11, p[5]);
Full_Adder F12(c8, a3b3, c11, p[7], p[6]);


endmodule


module SCFF(clk, rst_n, scan_in, data, scan_en, scan_out);
input clk, rst_n, scan_in, data, scan_en;
output reg scan_out;

wire w1, w2;
assign w1 = (scan_en == 1'b1) ? scan_in : data;
assign w2 = (rst_n == 1'b1) ? w1 : 1'b0;

always @(posedge clk) begin
    scan_out <= w2;
end

endmodule


module Scan_Chain_Design(clk, rst_n, scan_in, scan_en, scan_out);
input clk;
input rst_n;
input scan_in;
input scan_en;
output scan_out;

wire out1, out2, out3, out4, out5, out6, out7, out8;
wire w1, w2, w3, w4, w5, w6, w7, w8;
wire [3:0] a, b;
wire [7:0] p;

SCFF s1(clk, rst_n, scan_in, p[7], scan_en, out1);
SCFF s2(clk, rst_n, w1, p[6], scan_en, out2);
SCFF s3(clk, rst_n, w2, p[5], scan_en, out3);
SCFF s4(clk, rst_n, w3, p[4], scan_en, out4);
SCFF s5(clk, rst_n, w4, p[3], scan_en, out5);
```

```
SCFF s6(clk, rst_n, w5, p[2], scan_en, out6);
SCFF s7(clk, rst_n, w6, p[1], scan_en, out7);
SCFF s8(clk, rst_n, w7, p[0], scan_en, out8);

Multiplier_4bit M1({w1,w2,w3,w4}, {w5,w6,w7,w8}, p);

assign w1 = out1;
assign w2 = out2;
assign w3 = out3;
assign w4 = out4;
assign w5 = out5;
assign w6 = out6;
assign w7 = out7;
assign w8 = out8;

assign scan_out = out8;


endmodule

module Many_To_One_LFSR(clk, rst_n, out);
input clk;
input rst_n;
reg [8-1:0] temp;
output out;

always @(posedge clk) begin
    if (~rst_n) temp <= 8'b10111101;
    else temp <= {temp[6:0], temp[1] ^ temp[2] ^ temp[3] ^ temp[7]};
end

assign out = temp[7];

endmodule


module Built_In_Self_Test(clk, rst_n, scan_en, scan_in, scan_out);
input clk;
input rst_n;
```

```
input scan_en;
output scan_in;
output scan_out;

wire w1;

Many_To_One_LFSR M1(clk, rst_n, w1);
assign scan_in = w1;

Scan_Chain_Design S1(clk, rst_n, w1, scan_en, scan_out);


endmodule
```

How Do I Design my Testbench:

Because there are already reset value in the LFSR, I only need to manipulate rst_n and scan_en signal. Using this module in Basic question, it would equals 1011 * 1101 = 010001111.

Since the LFSR will alternate the data itself contains by using XOR operation, the result of the waveform diagram is not fixed to one set of multiplier output.

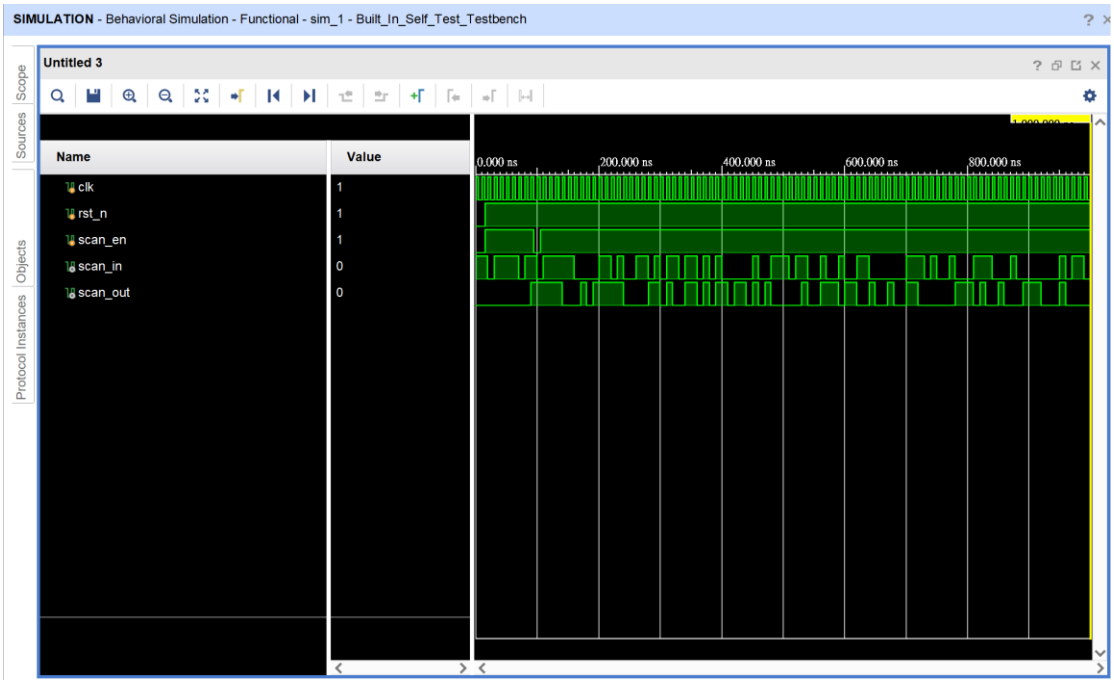
Here's the testbench:

```
`timescale 1ns/1ps

module Built_In_Self_Test_Testbench;

reg clk, rst_n, scan_en;
wire scan_in, scan_out;

Built_In_Self_Test B1(clk, rst_n, scan_en, scan_in, scan_out);

always #5 clk = ~clk;

initial begin
    clk = 1'b1;
```

```
    rst_n = 1'b0;
    scan_en = 1'b0;

    // 1011 * 1101 = 010001111

    @(negedge clk)
        rst_n = 1'b0;
    @(negedge clk)
        rst_n = 1'b1;
        scan_en = 1'b1;
    @(negedge clk)
        rst_n = 1'b1;
    @(negedge clk)
        rst_n = 1'b1;
    @(negedge clk)
        rst_n = 1'b1;
    @(negedge clk)
        rst_n = 1'b1;
    @(negedge clk)
        rst_n = 1'b1;
    @(negedge clk)
        rst_n = 1'b1;
    @(negedge clk)
        rst_n = 1'b1;
    @(negedge clk)
        rst_n = 1'b1;
    @(negedge clk)
        scan_en = 1'b0;
    @(negedge clk)
        scan_en = 1'b1;
    @(negedge clk)
        rst_n = 1'b1;
    @(negedge clk)
        rst_n = 1'b1;
    @(negedge clk)
        rst_n = 1'b1;
    @(negedge clk)
        rst_n = 1'b1;
    @(negedge clk)
        rst_n = 1'b1;
```

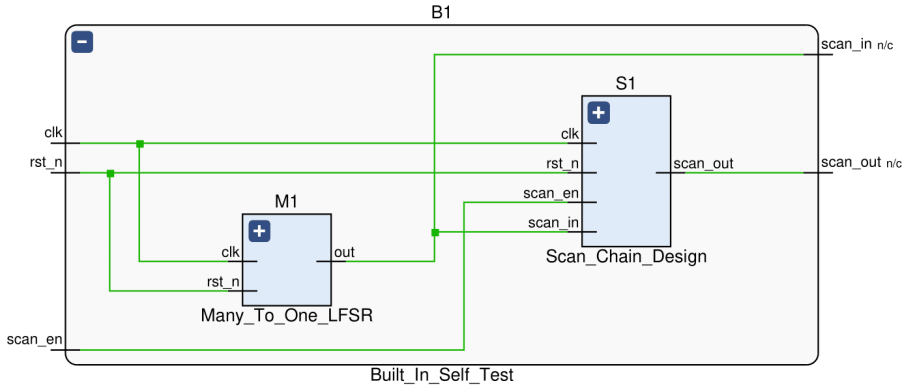
```
    @(negedge clk)
        rst_n = 1'b1;
    @(negedge clk)
        rst_n = 1'b1;
    @(negedge clk)
        rst_n = 1'b1;

end
endmodule
```

Here’s the waveform diagram:

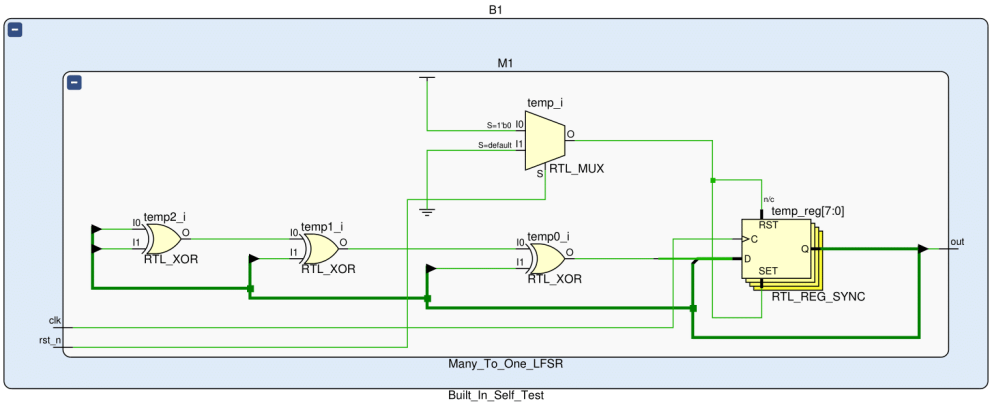


Block Diagram:

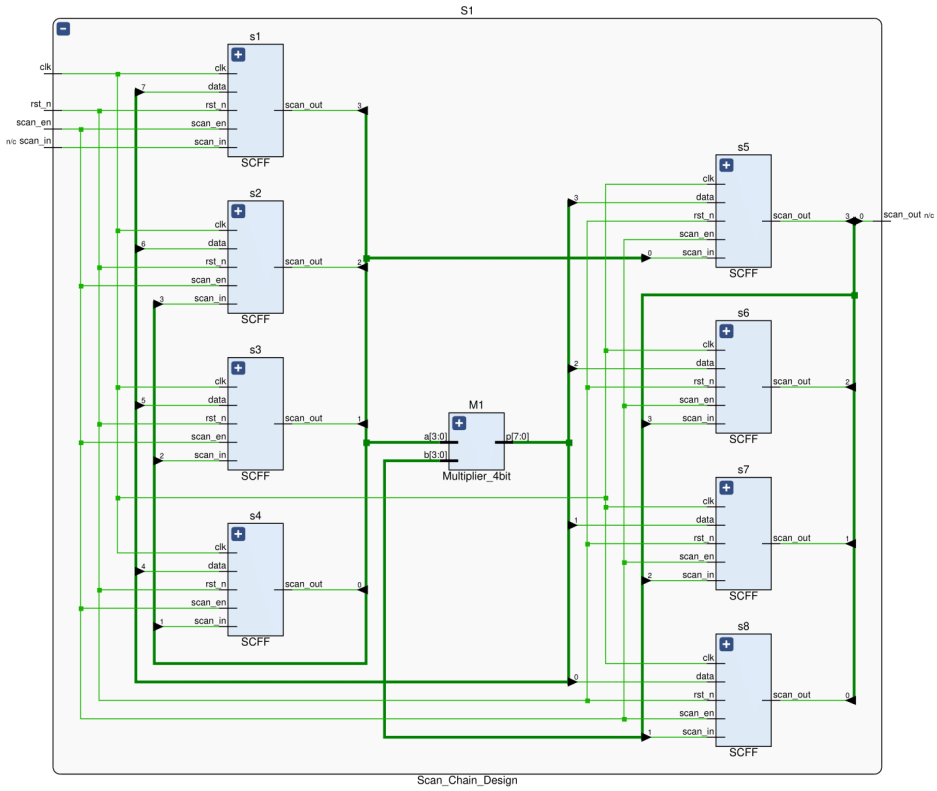
BIST:



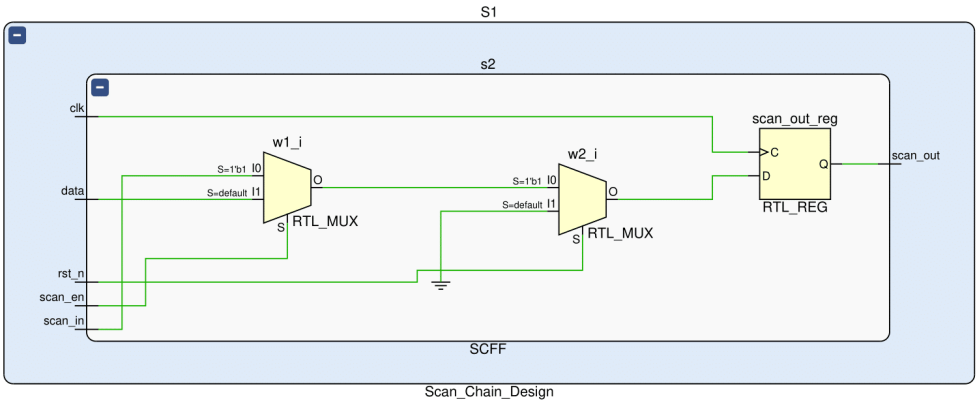
LFSR:


B1
M1
temp_i
S=1'b0
I0
S=default
I1
O
S
RTL_MUX
temp2_i
I0
I1
O
RTL_XOR
temp1_i
I0
I1
O
RTL_XOR
temp0_i
I0
I1
O
RTL_XOR
n/c
temp_reg[7:0]
RST
C
D
Q
SET
RTL_REG_SYNC
out
clk
rst_n
Many_To_One_LFSR
Built_In_Self_Test

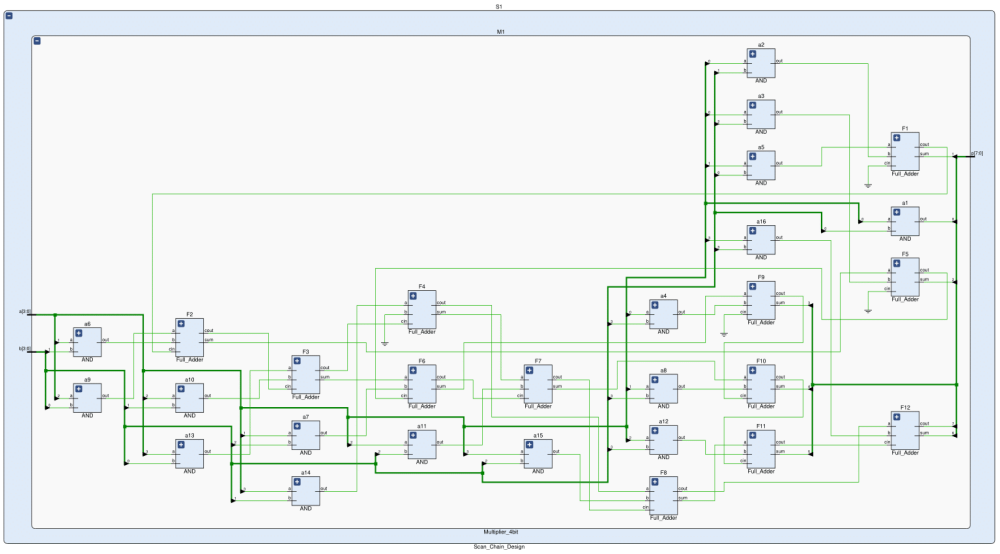
Scan chain design:



SCFF:


S1
s2
clk
data
rst_n
scan_en
scan_in
w1_i
S=1'b1 I0
S=default I1
O
S
RTL_MUX
w2_i
S=1'b1 I0
S=default I1
O
S
RTL_MUX
scan_out_reg
C
D
Q
RTL_REG
scan_out
SCFF
Scan_Chain_Design

Multiplier:



Advanced question 4, Mealy machine sequence detector:

How my modules work:

My code is short, there's a mealy sequence going, and I design my mealy sequence so that it can detect 0111, 1001, or 1110, and the dec is determined by combinational logic so that if the state changed it would change accordingly.

There's a counter that will be reset every 4 cycles, if counter == 2'd3 It would change the state to S0 on posedge clk.

Here's my source code:

```
`timescale 1ns/1ps

module Mealy_Sequence_Detector (clk, rst_n, in, dec);
input clk, rst_n;
input in;
output dec;

parameter S0 = 4'd0;
parameter S1 = 4'd1;
parameter S2 = 4'd2;
parameter S3 = 4'd3;
parameter S4 = 4'd4;
parameter S5 = 4'd5;
parameter S6 = 4'd6;
parameter S7 = 4'd7;
parameter S8 = 4'd8;

reg dec;

reg [1:0] counter, next_counter;;
```

```
reg [3:0] state, next_state;

always @(posedge clk) begin
    if(~rst_n) begin
        state <= S0;
        dec <= 1'b0;
        counter <= 2'd0;
        next_counter <= 2'd0;
    end
    else begin
        if(counter == 2'd3) begin
            counter <= 2'd0;
            state <= S0;
        end
        else begin
            counter <= counter + 2'd1;
            state <= next_state;
        end

    end

end

always @(*) begin
    case(state)
    S0: begin
        if(in == 1'b1) begin
            next_state = S1;
            dec = 1'b0;
        end
        else begin
            next_state = S4;
            dec = 1'b0;
        end
    end
    S1: begin
        if(in == 1'b1) begin
            next_state = S7;
```

```
            dec = 1'b0;
        end
        else begin
            next_state = S2;
            dec = 1'b0;
        end
    end
    S2: begin
        if(in == 1'b1) begin
            next_state = S2;
            dec = 1'b0;
        end
        else begin
            next_state = S3;
            dec = 1'b0;
        end
    end
    S3: begin
        if(in == 1'b1) begin
            next_state = S0;
            dec = 1'b1;
        end
        else begin
            next_state = S3;
            dec = 1'b0;
        end
    end
    S4: begin
        if(in == 1'b1) begin
            next_state = S5;
            dec = 1'b0;
        end
        else begin
            next_state = S4;
            dec = 1'b0;
        end
    end
    S5: begin
```

```
        if(in == 1'b1) begin
            next_state = S6;
            dec = 1'b0;
        end
        else begin
            next_state = S5;
            dec = 1'b0;
        end
    end
    S6: begin
        if(in == 1'b1) begin
            next_state = S0;
            dec = 1'b1;
        end
        else begin
            next_state = S6;
            dec = 1'b0;
        end
    end
    S7: begin
        if(in == 1'b1) begin
            next_state = S8;
            dec = 1'b0;
        end
        else begin
            next_state = S7;
            dec = 1'b0;
        end
    end
    S8: begin
        if(in == 1'b1) begin
            next_state = S8;
            dec = 1'b0;
        end
        else begin
            next_state = S0;
            dec = 1'b1;
        end
```

```
    end
    endcase
end

endmodule
```

How Do I Design Testbench:

I made my testbench the same as the one on the pdf, it tests the module if it can detect 3 sequences 0111, 1001, or 1110, and 1 error set.

In addition, after the error sequence, I add another 1001 sequences to test if the counter is working correctly so that it can still test the correct sequence after a wrong one.

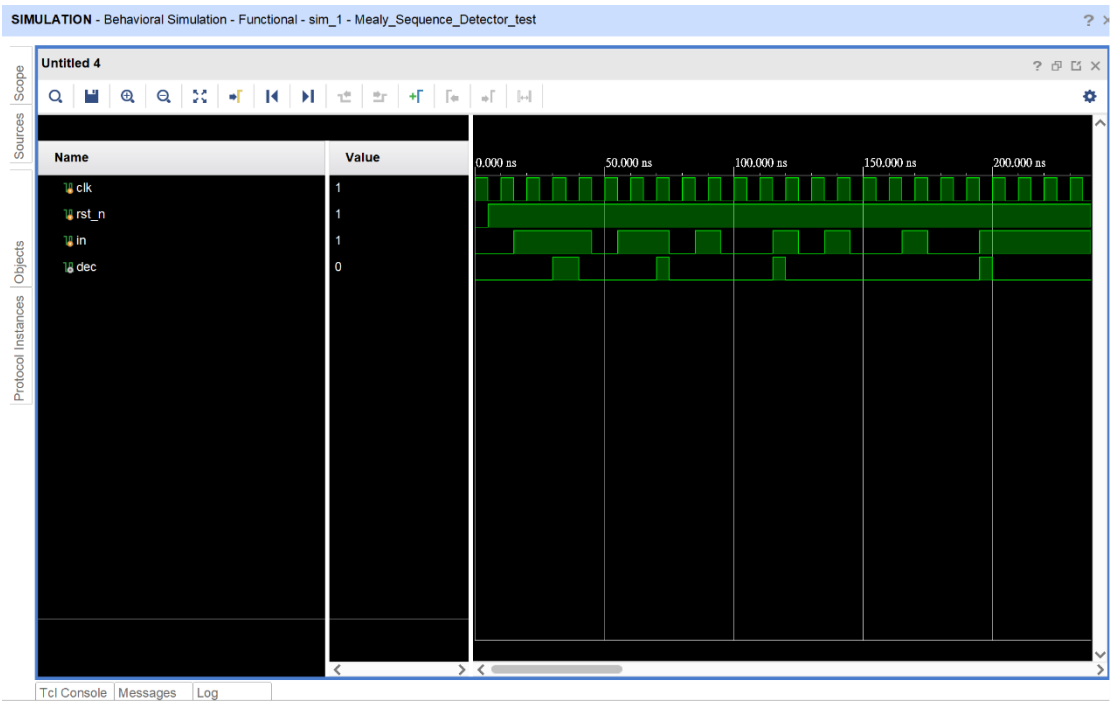
Here's the testbench:

```
`timescale 1ns/1ps

module Mealy_Sequence_Detector_test;
reg clk, rst_n, in;
wire dec;

always #5 clk = ~clk;

Mealy_Sequence_Detector M1(clk, rst_n, in, dec);

initial begin
    clk = 1'b1;
    in = 1'b0;
    rst_n = 1'b0;

    #5
    rst_n = 1'b1;

    #10
    in = 1'b1;
    #10
    in = 1'b1;
```

```
    #10
    in = 1'b1;

    #10
    in = 1'b0;
    #10
    in = 1'b1;
    #10
    in = 1'b1;
    #10
    in = 1'b0;

    #10
    in = 1'b1;
    #10
    in = 1'b0;
    #10
    in = 1'b0;
    #10
    in = 1'b1;

    #10
    in = 1'b0;
    #10
    in = 1'b1;
    #10
    in = 1'b0;
    #10
    in = 1'b0;

    #10
    in = 1'b1;
    #10
    in = 1'b0;
    #10
    in = 1'b0;
    #10
    in = 1'b1;
```

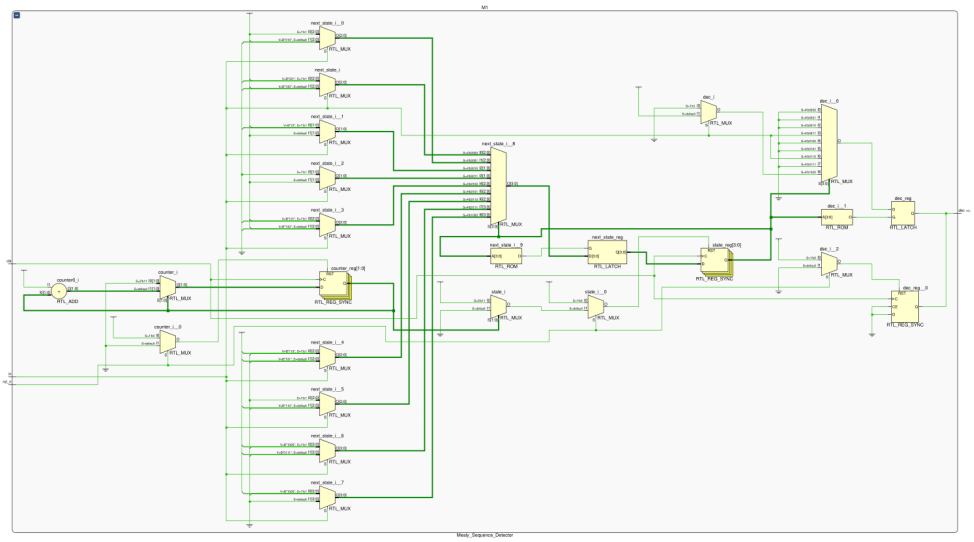
```
end

endmodule
```

Here’s the waveform diagram:



Block Diagram:

Mealy sequence detector:



Advanced Question 5, FPGA:

How my modules work:

My modules didn't work, I did not pass the FPGA demonstration due to unexpected error. However, I do design the modules, in which I modified the LFSR that it can be reset with input value, and some modifications on LED.

Theoretically, when I push the upward button, I can reset the module with the signal process by *debounce* and *onepulse* module. After that, each time hitting the right button will cause the clk signal to increment by one cycle.
(I combine *debounce* and *onepulse* modules in one module called *one_signal*)

After using the 8 switches to set the reset value, and one additional SW to control scan_en signal, the modules should work like the BIST, also the 7 segment on the BASYS 3 board will display a[3:0] and b[3:0] respectively.

Here's the source code: (only the FPGA top module part)

```
module debounce(pb_debounced, pb, clk);
input pb, clk;
output pb_debounced;

reg[3:0] DFF;

always @(posedge clk) begin
    DFF[3:1] <= DFF[2:0];
    DFF[0] <= pb;
end

assign pb_debounced = (DFF == 4'b1111) ? 1'b1 : 1'b0;

endmodule


module one_pulse(pb_debounced, pb_onepulse, clk);
input pb_debounced, clk;
output pb_onepulse;

reg pb_onepulse;
reg pb_debounced_delay;
```

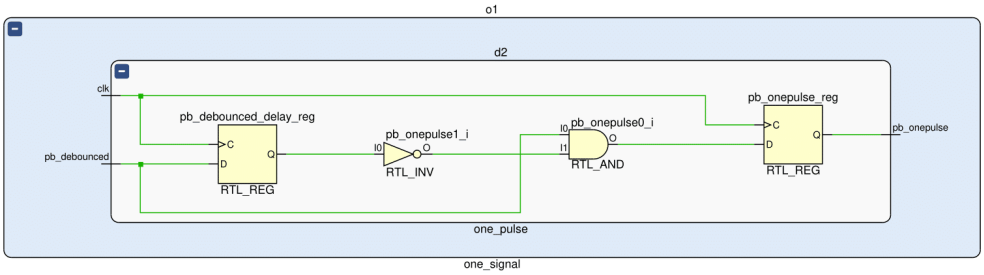
```
always @(posedge clk) begin
    pb_onepulse <= pb_debounced & (!pb_debounced_delay);
    pb_debounced_delay <= pb_debounced;
end
endmodule

module one_signal(pb, clk, pb_onepulse);
input pb, clk;
output pb_onepulse;

wire w1;

debounce d1(w1, pb, clk);
one_pulse d2(w1, pb_onepulse, clk);

endmodule


module Build_In_Self_Test_FPGA(SW, btn, clk, LED, digit, segment);
input [8:0] SW;
input [1:0] btn;
input clk;
output reg [6:0] segment;
output reg [3:0] digit;
output [9:0] LED;


// btn[0]=rst_n, btn[1]=d_clk;
wire rst_n, dclk;

one_signal o1(btn[0], clk, rst_n);
one_signal o2(btn[1], clk, dclk);


//Built_In_Self_Test(clk, rst_n, scan_en, def_value, scan_in, scan_out,
LED)
Built_In_Self_Test B1(dclk, !rst_n, SW[8], SW[7:0], LED[8], LED[9],
LED[7:0]);
```

```
reg [19:0] refresh_counter;
wire [1:0] sel;

always @(posedge clk or posedge rst_n) begin
    if(rst_n) refresh_counter <= 0;
    else refresh_counter <= refresh_counter + 1;
end

assign sel = refresh_counter[19:18];

always @(sel) begin
        if(sel == 2'b00) begin
            digit = 4'b1110;
            case(SW[3:0])
                4'd0: segment = 7'b0000001;
                4'd1: segment = 7'b1001111;
                4'd2: segment = 7'b0010010;
                4'd3: segment = 7'b0000110;
                4'd4: segment = 7'b1001100;
                4'd5: segment = 7'b0100100;
                4'd6: segment = 7'b0100000;
                4'd7: segment = 7'b0001111;
                4'd8: segment = 7'b0000000;
                4'd9: segment = 7'b0000100;
                4'd10: segment = 7'b0001000;
                4'd11: segment = 7'b1100000;
                4'd12: segment = 7'b0110001;
                4'd13: segment = 7'b1000010;
                4'd14: segment = 7'b0110000;
                4'd15: segment = 7'b0111000;
                default: segment = 7'b0000000;
            endcase
        end
        else if(sel == 2'b01) begin
            digit = 4'b1101;
            case(SW[7:4])
                4'd0: segment = 7'b0000001;
                4'd1: segment = 7'b1001111;
```

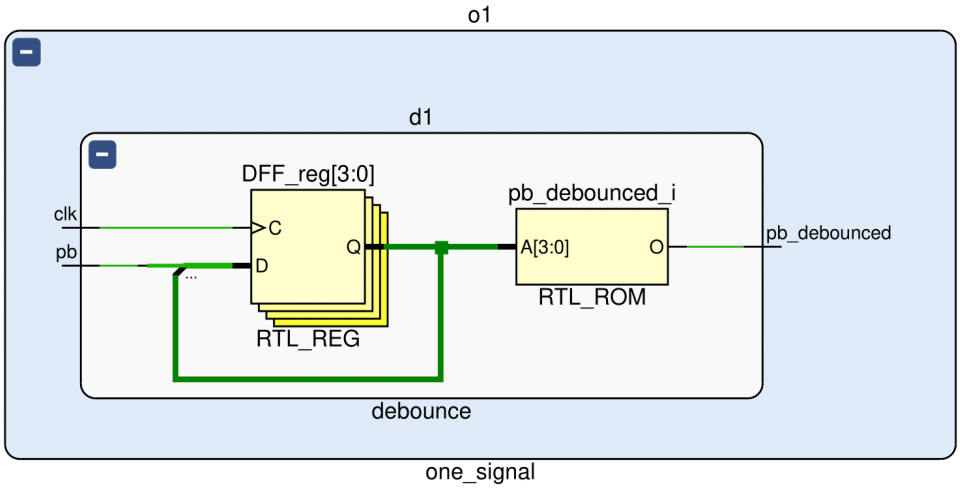
```
                4'd2: segment = 7'b0010010;
                4'd3: segment = 7'b0000110;
                4'd4: segment = 7'b1001100;
                4'd5: segment = 7'b0100100;
                4'd6: segment = 7'b0100000;
                4'd7: segment = 7'b0001111;
                4'd8: segment = 7'b0000000;
                4'd9: segment = 7'b0000100;
                4'd10: segment = 7'b0001000;
                4'd11: segment = 7'b1100000;
                4'd12: segment = 7'b0110001;
                4'd13: segment = 7'b1000010;
                4'd14: segment = 7'b0110000;
                4'd15: segment = 7'b0111000;
                default: segment = 7'b0000000;
            endcase
        end
        else if(sel == 2'b10) begin
            digit = 4'b1011;
            segment = 7'b0000000;
        end
        else begin
            digit = 4'b0111;
            segment = 7'b0000000;
        end
end


endmodule
```

Block Diagram:

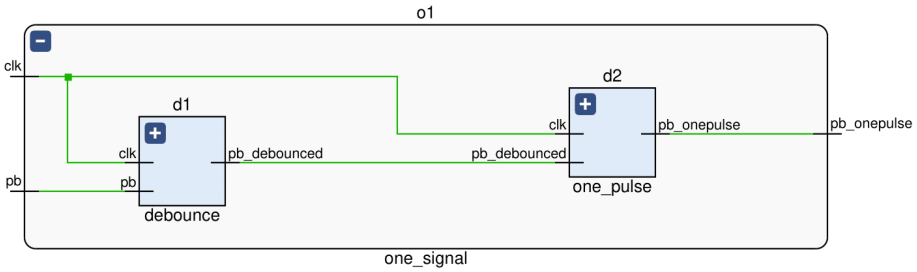
onepulse:


o1
d2
clk
pb_debounced
pb_debounced_delay_reg
C
D
Q
RTL_REG
pb_onepulse1_i
I0
O
RTL_INV
pb_onepulse0_i
I0
I1
O
RTL_AND
pb_onepulse_reg
C
D
Q
RTL_REG
pb_onepulse
one_pulse
one_signal

debounce:


o1
d1
DFF_reg[3:0]
clk
pb
C
D
Q
RTL_REG
pb_debounced_i
A[3:0]
O
RTL_ROM
pb_debounced
debounce
one_signal

one_signal:


o1
clk
d1
clk
pb
pb
pb_debounced
debounce
d2
clk
pb_debounced
pb_onepulse
one_pulse
pb_onepulse
one_signal