**Q1.**

`timescale 1ns/1ps

`define kBitW 6

`define kBitS `kBitW-1:0

module upDownLFSR(

input wire clk, // Clock signal

input wire rst, // Reset signal

input wire en\_i, // Enable signal

input wire upDown\_i, // upDown\_i = 1 -> up; otherwise down

output reg [`kBitS] cnt\_o, // 6-bit counter output

output reg bitStr\_o // Overflow bit

);

reg [`kBitS] lfsr; // LFSR register

always @(posedge clk or posedge rst) begin

if (rst) begin

lfsr <= 6'b000001; // Initial seed value (non-zero)

cnt\_o <= 6'b000001; // Reset counter

bitStr\_o <= 1'b0; // Reset overflow bit

end else if (en\_i) begin

// Feedback logic for LFSR

if (upDown\_i) begin

// Count up

lfsr <= {lfsr[`kBitW-2:0], lfsr[`kBitW-1] ^ lfsr[4]};

cnt\_o <= cnt\_o + 1;

// Set overflow bit

if (cnt\_o == 6'b111111)

bitStr\_o <= 1'b1;

else

bitStr\_o <= 1'b0;

end else begin

// Count down

lfsr <= {lfsr[`kBitW-1] ^ lfsr[4], lfsr[`kBitW-1:1]};

cnt\_o <= cnt\_o - 1;

// Set overflow bit

if (cnt\_o == 6'b000000)

bitStr\_o <= 1'b1;

else

bitStr\_o <= 1'b0;

end

end

end

endmodule

`timescale 1ns/1ps

module upDownLFSR\_tb;

reg clk; // Clock signal

reg rst; // Reset signal

reg en\_i; // Enable signal

reg upDown\_i; // Up/Down control signal

wire [`kBitS] cnt\_o; // Counter output

wire bitStr\_o; // Overflow bit

// Instantiate the `upDownLFSR` module

upDownLFSR uut (

.clk(clk),

.rst(rst),

.en\_i(en\_i),

.upDown\_i(upDown\_i),

.cnt\_o(cnt\_o),

.bitStr\_o(bitStr\_o)

);

// Clock generation

initial begin

clk = 0;

forever #5 clk = ~clk; // 10 ns clock period

end

// Test sequence

initial begin

// Initialize signals

rst = 1;

en\_i = 0;

upDown\_i = 1; // Start with counting up

// Apply reset

#10 rst = 0;

en\_i = 1;

// Count up for a few cycles

#60 upDown\_i = 0; // Switch to counting down

// Count down for a few cycles

#60 en\_i = 0; // Disable counter

// Hold for observation

#40 en\_i = 1; // Re-enable counter

upDown\_i = 1; // Switch to counting up

#50 $finish; // End simulation

end

// Monitor output signals

initial begin

$monitor("Time=%0t | clk=%b | rst=%b | en\_i=%b | upDown\_i=%b | cnt\_o=%b | bitStr\_o=%b",

$time, clk, rst, en\_i, upDown\_i, cnt\_o, bitStr\_o);

end

// Generate waveform

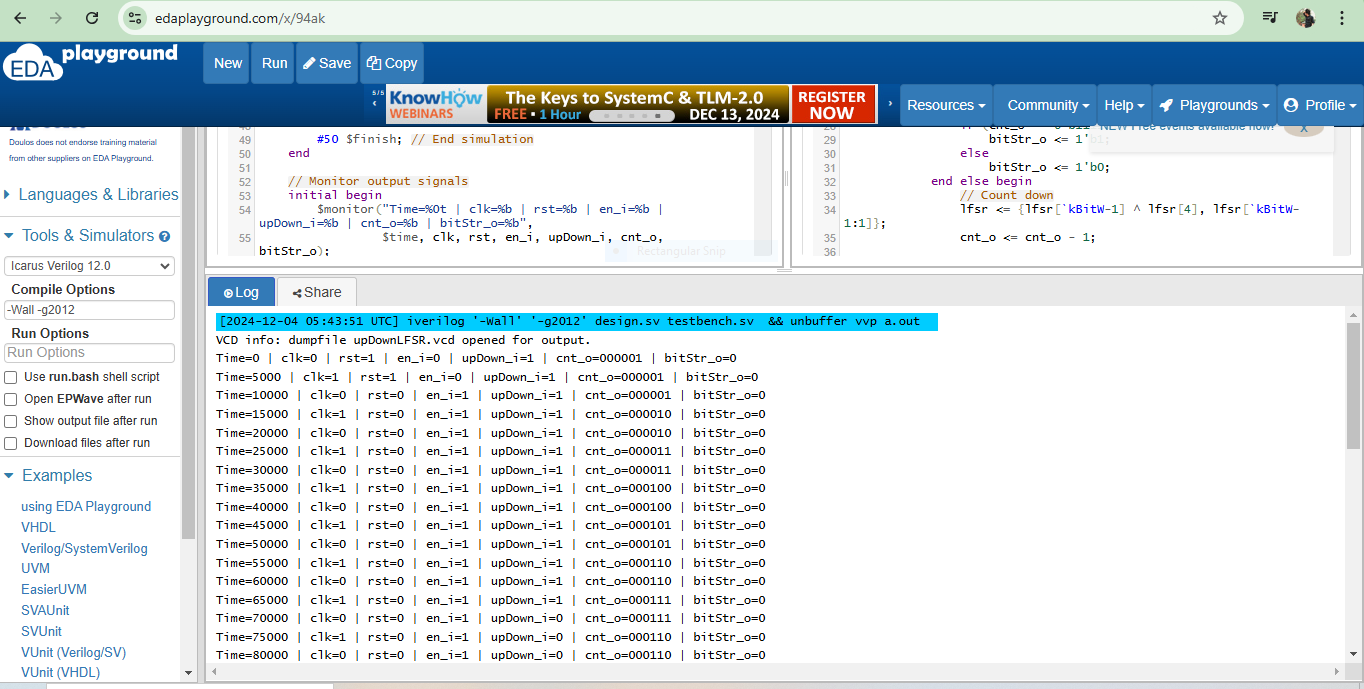
initial begin

$dumpfile("upDownLFSR.vcd");

$dumpvars(0, upDownLFSR\_tb);

end

endmodule



**Q2.**

`timescale 1ns/1ps

module freq\_div7 (

input wire clk\_in, // Input clock

input wire rst, // Reset signal

output reg clk\_out // Divided clock output

);

reg [2:0] counter; // 3-bit counter to count from 0 to 6

always @(posedge clk\_in or posedge rst) begin

if (rst) begin

counter <= 3'b000; // Reset counter

clk\_out <= 1'b0; // Reset output clock

end else begin

if (counter == 3'b110) begin // Counter reaches 6

counter <= 3'b000; // Reset counter

clk\_out <= ~clk\_out; // Toggle output clock

end else begin

counter <= counter + 1; // Increment counter

end

end

end

endmodule

`timescale 1ns/1ps

module freq\_div7\_tb;

reg clk\_in; // Input clock signal

reg rst; // Reset signal

wire clk\_out; // Divided clock output

freq\_div7 uut (

.clk\_in(clk\_in),

.rst(rst),

.clk\_out(clk\_out)

);

// Clock generation

initial begin

clk\_in = 0;

forever #5 clk\_in = ~clk\_in; // Toggle every 5ns

end

// Test sequence

initial begin

rst = 1; // Hold reset high

#10 rst = 0; // Release reset

#300 $finish; // End simulation

end

initial begin

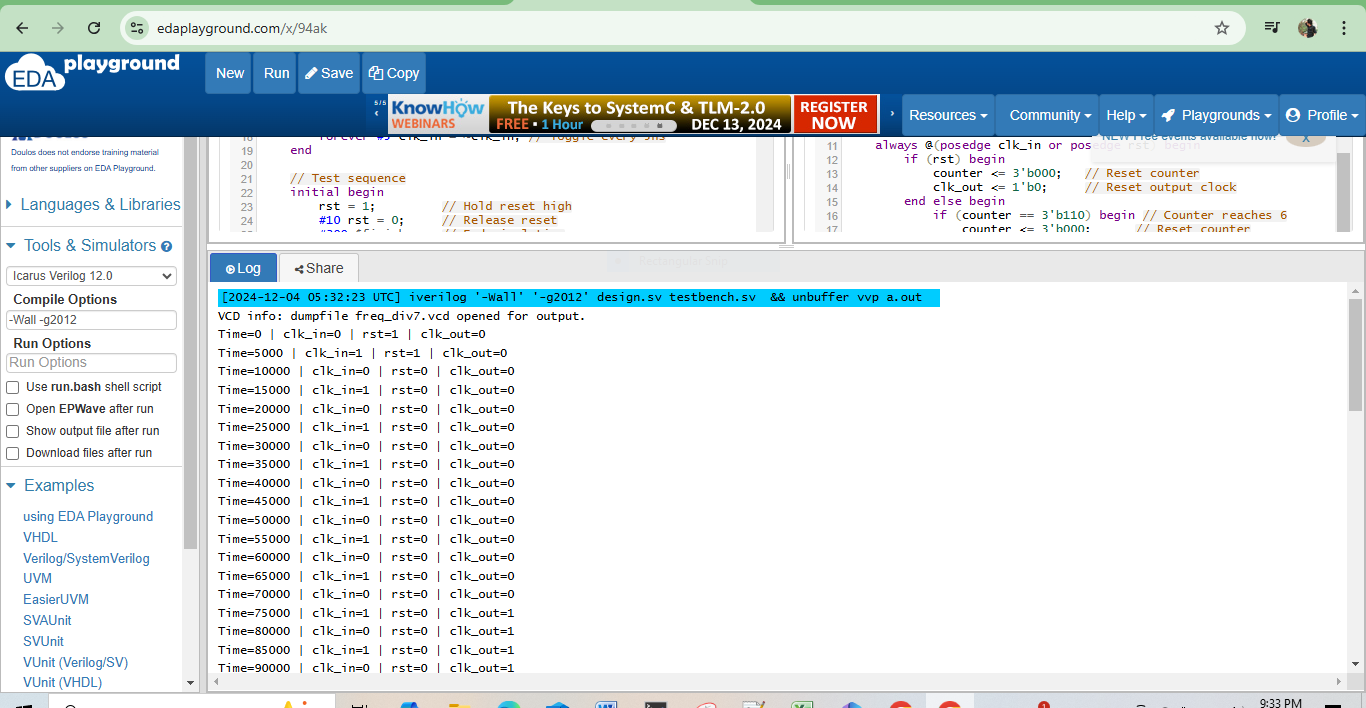
$monitor("Time=%0t | clk\_in=%b | rst=%b | clk\_out=%b", $time, clk\_in, rst, clk\_out);

$dumpfile("freq\_div7.vcd");

$dumpvars(0, freq\_div7\_tb);

end

endmodule



**Q3.**

module crc4\_decoder(

input wire [7:0] data\_in,

output wire [3:0] data\_out,

output wire crc\_error

);

parameter POLYNOMIAL = 5'b10011;

wire [4:0] remainder;

assign remainder = crc\_calc(data\_in);

assign data\_out = data\_in[7:4];

assign crc\_error = (remainder != 5'b00000);

function automatic [4:0] crc\_calc;

input [7:0] data;

reg [4:0] current\_remainder;

integer i;

begin

current\_remainder = data[7:4];

for (i = 0; i < 4; i = i + 1) begin

if (current\_remainder[4] == 1'b1)

current\_remainder = {current\_remainder[3:0], data[3-i]} ^ POLYNOMIAL;

else

current\_remainder = {current\_remainder[3:0], data[3-i]};

end

return current\_remainder;

end

endfunction

endmodule

module tb\_crc4\_decoder;

reg [7:0] data\_in;

wire [3:0] data\_out;

wire crc\_error;

crc4\_decoder uut (

.data\_in(data\_in),

.data\_out(data\_out),

.crc\_error(crc\_error)

);

initial begin

// Test Case 1: Valid Data

data\_in = 8'b10101010;

#10;

$display("Test Case 1:");

$display("Input: %b, Data Out: %b, CRC Error: %b", data\_in, data\_out, crc\_error);

// Test Case 2: Data with Error

data\_in = 8'b10111010;

#10;

$display("Test Case 2:");

$display("Input: %b, Data Out: %b, CRC Error: %b", data\_in, data\_out, crc\_error);

// Test Case 3: Another Valid Data

data\_in = 8'b11001100;

#10;

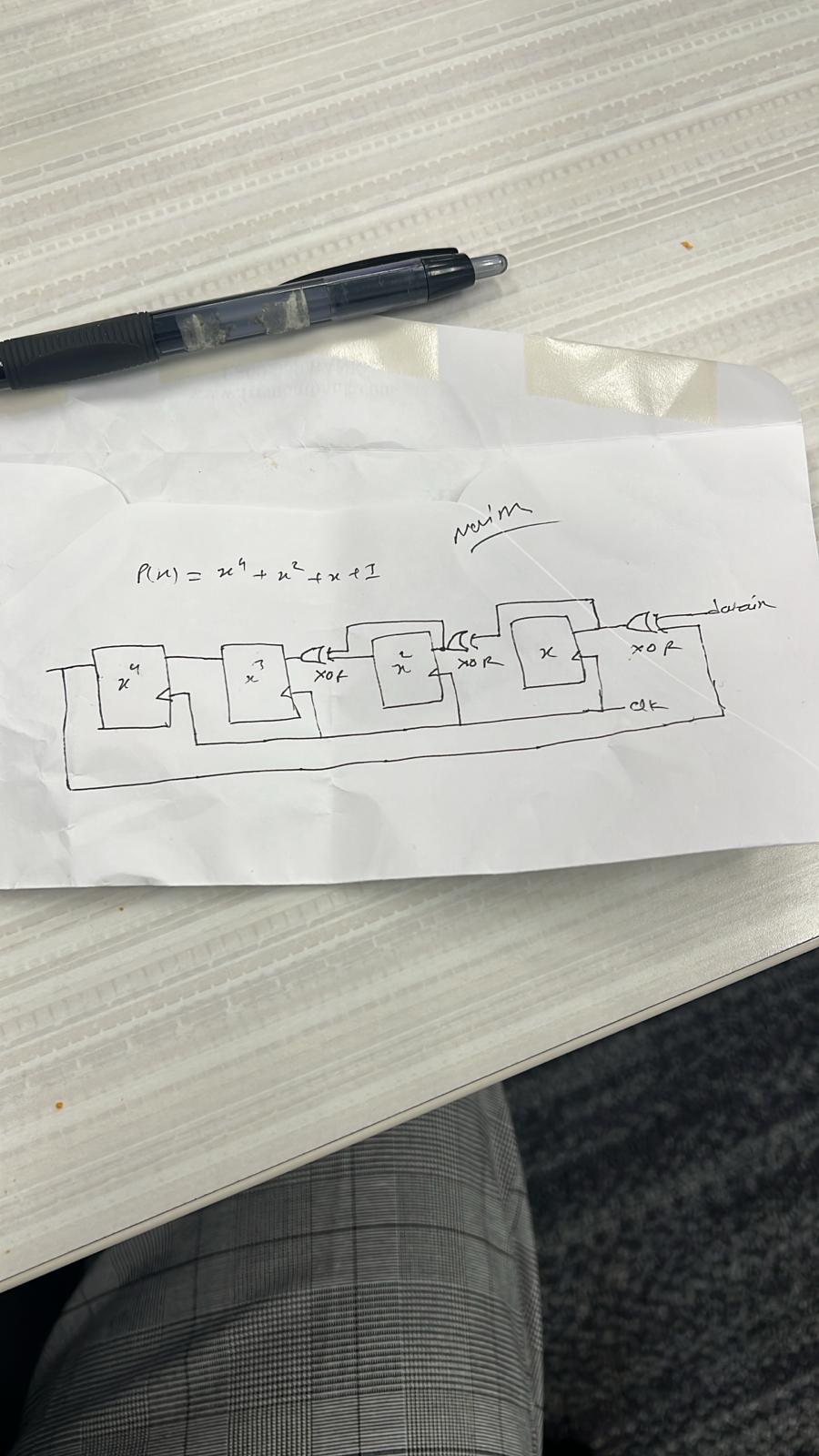
$display("Test Case 3:");

$display("Input: %b, Data Out: %b, CRC Error: %b", data\_in, data\_out, crc\_error);

$finish;

end

endmodule

`

