**Q1.**

module clock\_divider\_by\_5\_5 (

input clk\_in, // Input clock

input reset, // Reset signal

output reg clk\_out // Divided output clock

);

reg [3:0] counter; // 4-bit counter for fine-grained control

reg toggle;

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

if (reset) begin

counter <= 4'b0;

clk\_out <= 0;

toggle <= 0;

end else begin

if (counter == 4'd10) begin // Total cycles for 5.5 division (approximation)

counter <= 4'b0;

toggle <= ~toggle; // Toggle output every 5.5 input clocks

end else begin

counter <= counter + 1;

end

// Generate clk\_out with adjusted duty cycle

if (counter < 4'd5)

clk\_out <= toggle;

else

clk\_out <= ~toggle;

end

end

endmodule

module tb\_clock\_divider\_by\_5\_5;

reg clk\_in;

reg reset;

wire clk\_out;

// Instantiate the clock divider

clock\_divider\_by\_5\_5 uut (

.clk\_in(clk\_in),

.reset(reset),

.clk\_out(clk\_out)

);

// Generate clock: 10 ns period

initial begin

clk\_in = 0;

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

end

// Test procedure

initial begin

reset = 1;

#20; // Hold reset for 20 ns

reset = 0; // Release reset

#2000; // Run simulation for 2000 ns

$stop; // End simulation

end

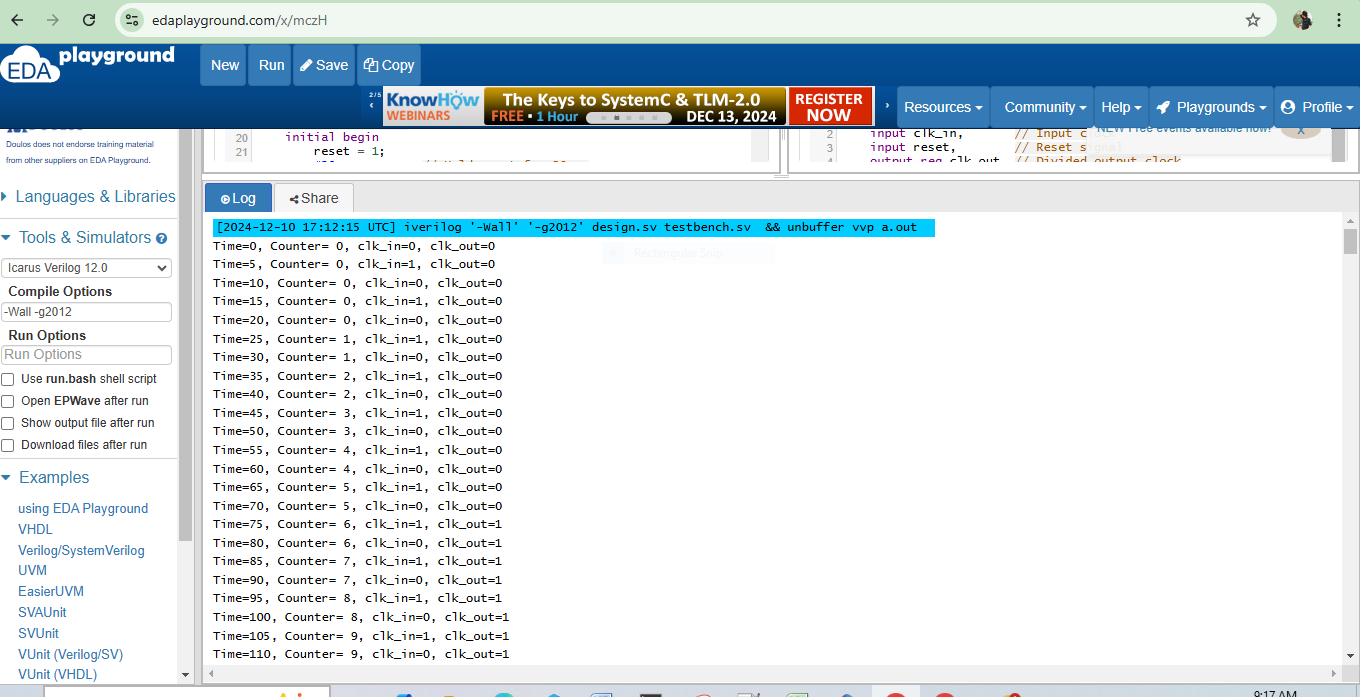
// Monitor for debug purposes

initial begin

$monitor("Time=%0d, Counter=%d, clk\_in=%b, clk\_out=%b",

$time, uut.counter, clk\_in, clk\_out);

end

endmodule  


**Q2.**

module ECL (

input clk, // Clock signal

input rst, // Reset signal

input button0, // Button 0 input

input button1, // Button 1 input

input button2, // Button 2 input

output reg open // Output signal

);

// State Encoding

typedef enum reg [2:0] {

S0 = 3'b000, // Initial state

S1 = 3'b001, // Received '2'

S2 = 3'b010, // Received '20'

S3 = 3'b011, // Received '201'

S4 = 3'b100 // Sequence complete, open

} state\_t;

state\_t current\_state, next\_state;

// State Transition Block (Sequential Logic)

always @(posedge clk or posedge rst) begin

if (rst) begin

current\_state <= S0; // Reset to initial state

end else begin

current\_state <= next\_state;

end

end

// Next State Logic (Combinational Logic)

always @(\*) begin

// Default values

next\_state = current\_state;

open = 0;

case (current\_state)

S0: if (button2) next\_state = S1; // Input = 2

S1: if (button0) next\_state = S2; // Input = 0

S2: if (button1) next\_state = S3; // Input = 1

S3: if (button0) next\_state = S4; // Input = 0

S4: begin

open = 1; // Unlock the lock

next\_state = S0; // Reset to initial state

end

endcase

end

endmodule

module tb\_ECL;

reg clk, rst;

reg button0, button1, button2;

wire open;

// Instantiate the ECL module

ECL uut (

.clk(clk),

.rst(rst),

.button0(button0),

.button1(button1),

.button2(button2),

.open(open)

);

// Clock Generation

initial begin

clk = 0;

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

end

// Test Sequence

initial begin

// Initialize inputs

rst = 1; button0 = 0; button1 = 0; button2 = 0;

#10; rst = 0;

// Apply sequence: 2 -> 0 -> 1 -> 0

#10; button2 = 1; // Input '2'

#10; button2 = 0; button0 = 1; // Input '0'

#10; button0 = 0; button1 = 1; // Input '1'

#10; button1 = 0; button0 = 1; // Input '0'

#10; button0 = 0;

// Wait and stop simulation

#50;

$finish;

end

// Monitor for Debugging

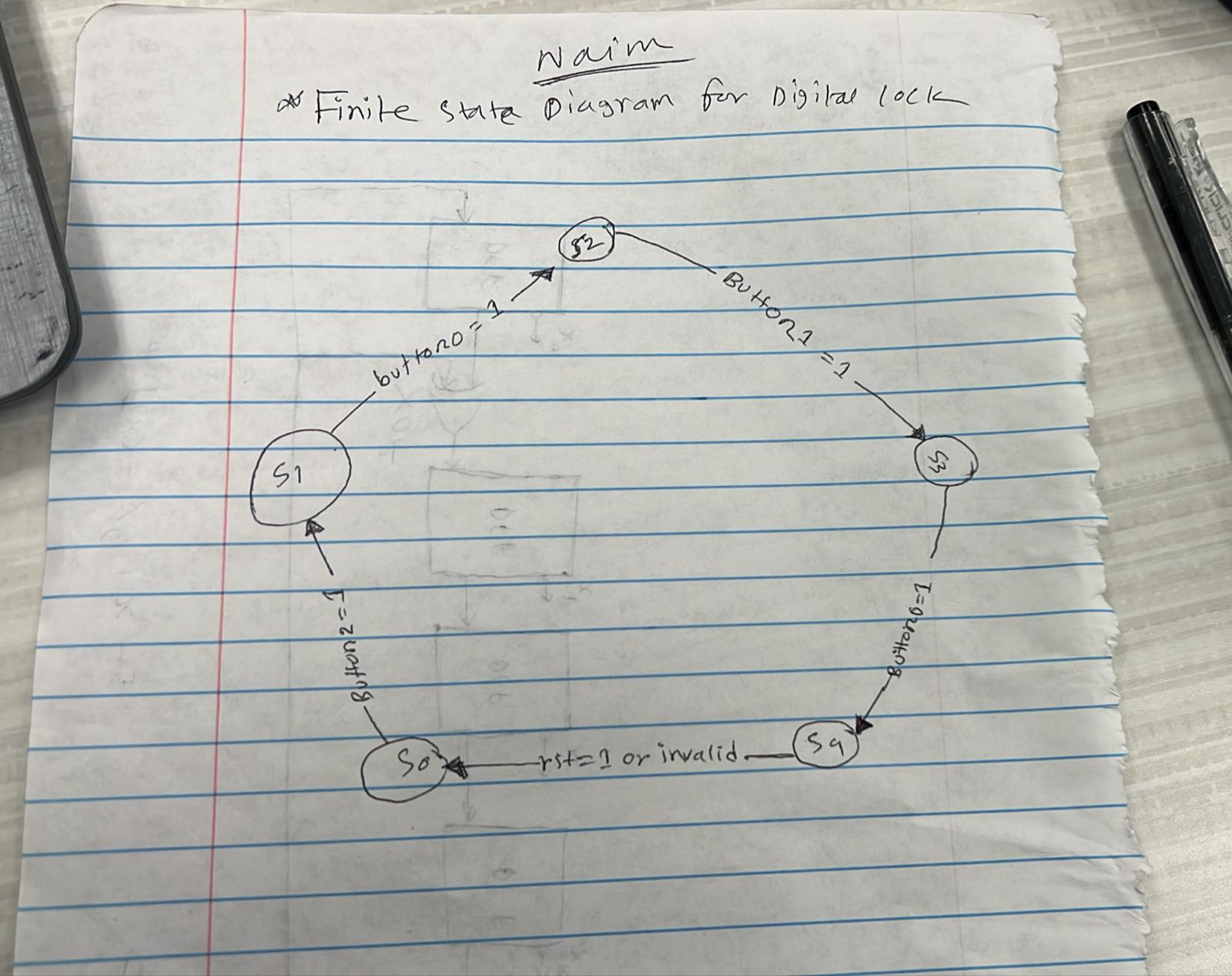
initial begin

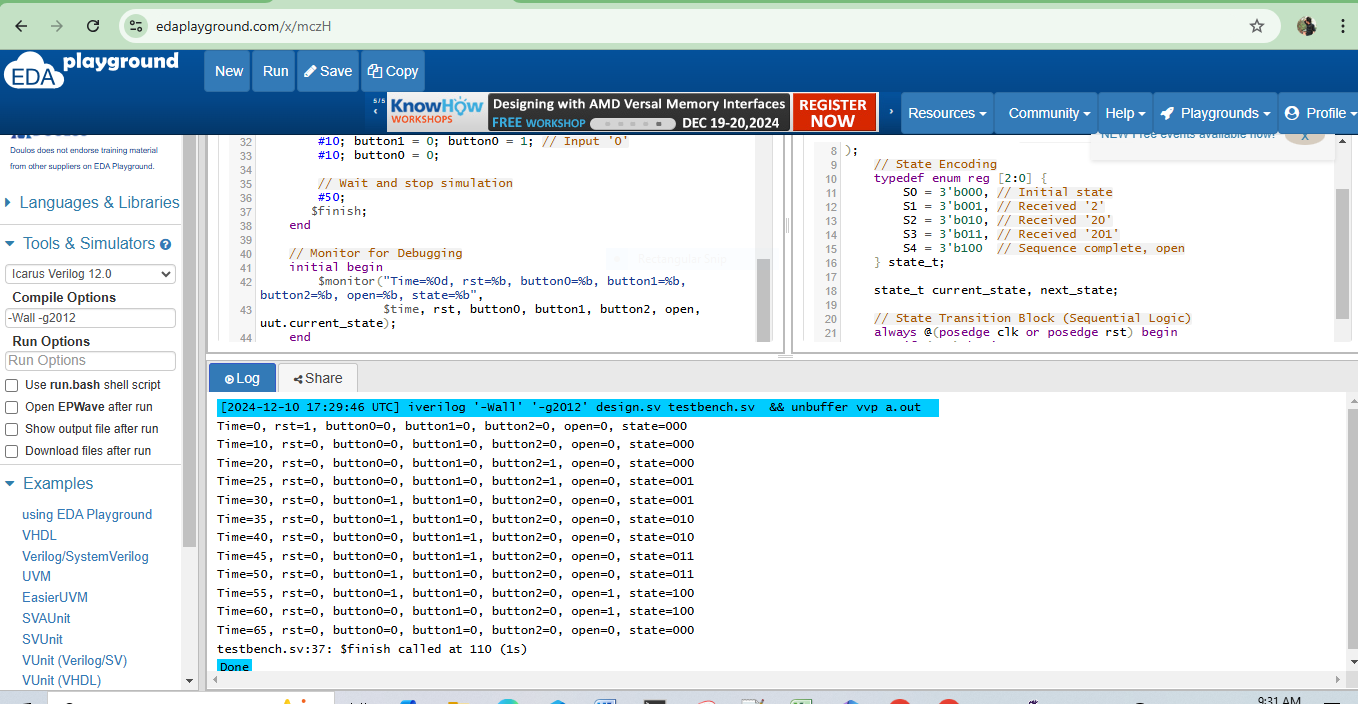
$monitor("Time=%0d, rst=%b, button0=%b, button1=%b, button2=%b, open=%b, state=%b",

$time, rst, button0, button1, button2, open, uut.current\_state);

end

endmodule





**Q3.**

module crc6\_encoder (

input clk, // Clock signal

input rst, // Reset signal (synchronous)

input serial\_in, // Serial input data

output reg [5:0] crc // CRC remainder (6 bits)

);

// Polynomial: x^6 + x + 1 (1000011)

wire feedback;

// Feedback Calculation

assign feedback = crc[5] ^ serial\_in;

// CRC Shift Register

always @(posedge clk or posedge rst) begin

if (rst) begin

crc <= 6'b0; // Reset CRC to 0

end else begin

// Shift and XOR based on feedback

crc <= {crc[4:0], feedback}; // Shift left and add feedback

crc[1] <= crc[1] ^ feedback; // Polynomial term for x^1

end

end

endmodule  
module tb\_crc6\_encoder;

reg clk; // Testbench clock

reg rst; // Testbench reset

reg serial\_in; // Testbench serial input

wire [5:0] crc; // CRC output

// Instantiate the CRC-6 Encoder

crc6\_encoder uut (

.clk(clk),

.rst(rst),

.serial\_in(serial\_in),

.crc(crc)

);

// Clock Generation

initial begin

clk = 0;

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

end

// Test Sequence

initial begin

// Initialize inputs

rst = 1; serial\_in = 0;

#10; rst = 0;

// Apply test data (example: 1011001)

#10; serial\_in = 1; // Bit 1

#10; serial\_in = 0; // Bit 2

#10; serial\_in = 1; // Bit 3

#10; serial\_in = 1; // Bit 4

#10; serial\_in = 0; // Bit 5

#10; serial\_in = 0; // Bit 6

#10; serial\_in = 1; // Bit 7

// End simulation

#50;

$finish;

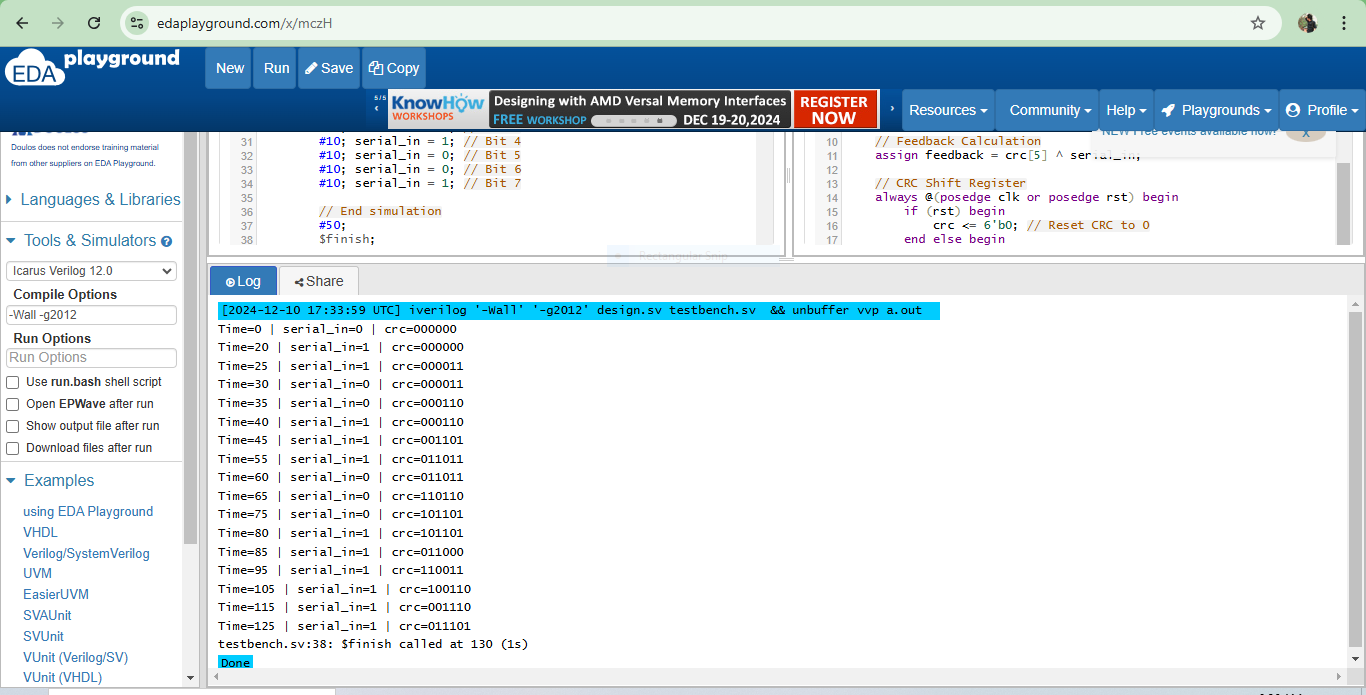
end

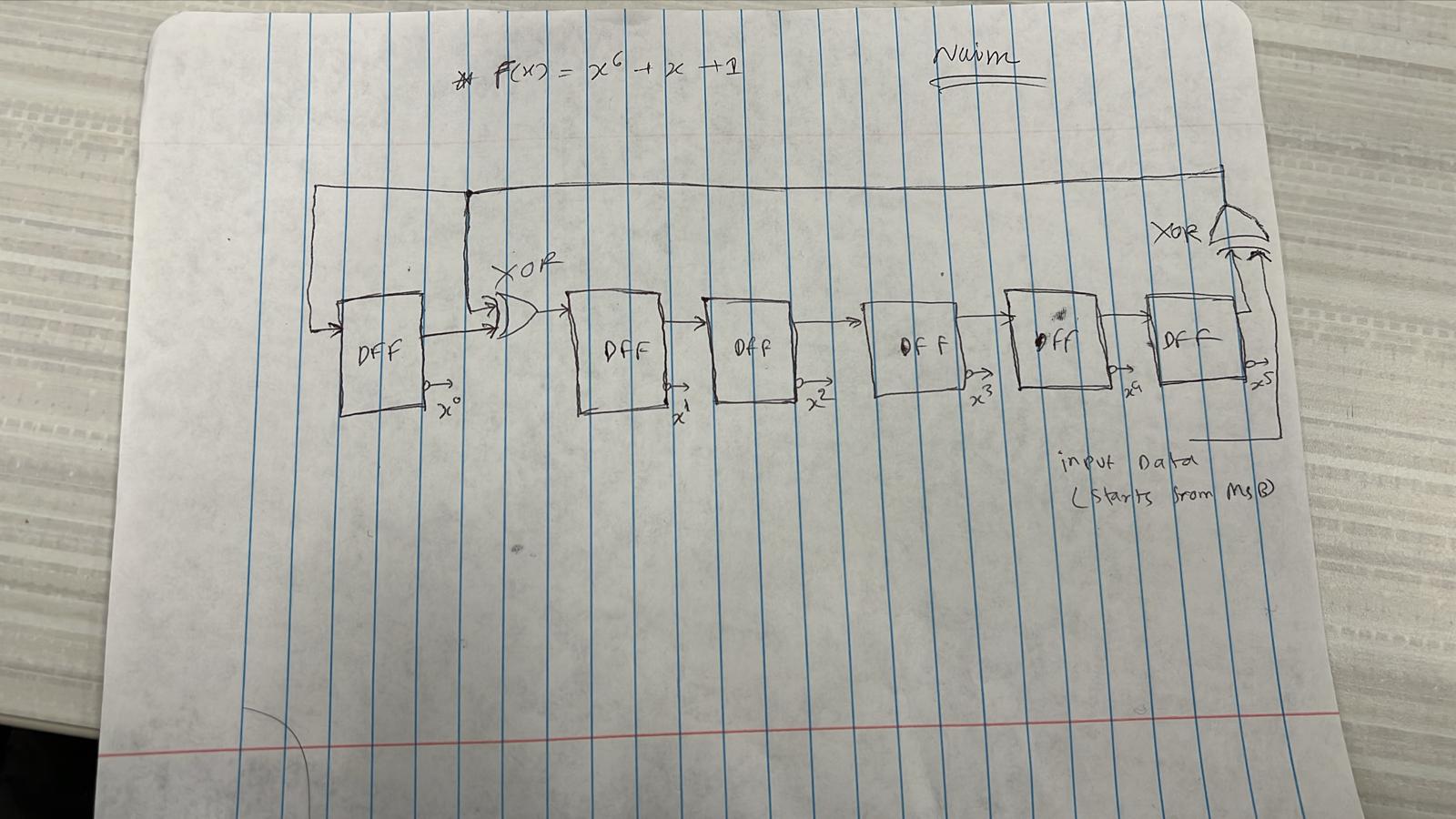
// Monitor CRC for debugging

initial begin

$monitor("Time=%0d | serial\_in=%b | crc=%b", $time, serial\_in, crc);

end

endmodule  




**Q4.**

module dual\_clock\_fifo (

input wire clk\_tx, // Transmitter clock

input wire clk\_rx, // Receiver clock

input wire rst, // Reset signal (active high)

input wire write\_enable, // Write enable signal

input wire read\_enable, // Read enable signal

input wire [7:0] data\_in, // 8-bit data input

output reg [7:0] data\_out, // 8-bit data output

output reg full, // FIFO full flag

output reg empty // FIFO empty flag

);

parameter DEPTH = 16; // Depth of the FIFO

parameter ADDR\_WIDTH = 4; // Address width for FIFO indexing

reg [7:0] fifo\_mem [DEPTH-1:0]; // FIFO memory array

reg [ADDR\_WIDTH-1:0] write\_ptr; // Write pointer

reg [ADDR\_WIDTH-1:0] read\_ptr; // Read pointer

reg [ADDR\_WIDTH:0] fifo\_count; // Counter for number of elements in FIFO

// Reset and Initialization

always @(posedge clk\_tx or posedge clk\_rx or posedge rst) begin

if (rst) begin

write\_ptr <= 0;

read\_ptr <= 0;

fifo\_count <= 0;

empty <= 1;

full <= 0;

end

end

// Write Logic

always @(posedge clk\_tx) begin

if (write\_enable && !full) begin

fifo\_mem[write\_ptr] <= data\_in; // Write data to FIFO

write\_ptr <= write\_ptr + 1;

fifo\_count <= fifo\_count + 1;

empty <= 0;

full <= (fifo\_count == DEPTH - 1); // Set full flag

end

end

// Read Logic

always @(posedge clk\_rx) begin

if (read\_enable && !empty) begin

data\_out <= fifo\_mem[read\_ptr]; // Read data from FIFO

read\_ptr <= read\_ptr + 1;

fifo\_count <= fifo\_count - 1;

full <= 0;

empty <= (fifo\_count == 1); // Set empty flag

end

end

// Full and Empty Flags

always @(posedge clk\_tx or posedge clk\_rx) begin

full <= (fifo\_count == DEPTH);

empty <= (fifo\_count == 0);

end

endmodule  
module tb\_dual\_clock\_fifo;

reg clk\_tx; // Transmitter clock

reg clk\_rx; // Receiver clock

reg rst; // Reset signal

reg write\_enable; // Write enable

reg read\_enable; // Read enable

reg [7:0] data\_in; // Data input to FIFO

wire [7:0] data\_out; // Data output from FIFO

wire full; // FIFO full flag

wire empty; // FIFO empty flag

// Instantiate the Dual-Clock FIFO

dual\_clock\_fifo uut (

.clk\_tx(clk\_tx),

.clk\_rx(clk\_rx),

.rst(rst),

.write\_enable(write\_enable),

.read\_enable(read\_enable),

.data\_in(data\_in),

.data\_out(data\_out),

.full(full),

.empty(empty)

);

// Generate Transmitter Clock: 50 MHz (20 ns period)

initial begin

clk\_tx = 0;

forever #10 clk\_tx = ~clk\_tx;

end

// Generate Receiver Clock: 80 MHz (12.5 ns period)

initial begin

clk\_rx = 0;

forever #6.25 clk\_rx = ~clk\_rx;

end

// Test Sequence

initial begin

// Initialize signals

rst = 1;

write\_enable = 0;

read\_enable = 0;

data\_in = 0;

#50; rst = 0; // Release reset after 50 ns

// Write 20 data words to FIFO

repeat (20) begin

@(posedge clk\_tx);

write\_enable = 1;

data\_in = $random % 256; // Random 8-bit data

@(posedge clk\_tx);

write\_enable = 0;

end

// Read 20 data words from FIFO

repeat (20) begin

@(posedge clk\_rx);

read\_enable = 1;

@(posedge clk\_rx);

read\_enable = 0;

end

// End simulation

#200;

$finish;

end

// Monitor Outputs

initial begin

$monitor("Time=%0d | data\_in=%b | data\_out=%b | full=%b | empty=%b",

$time, data\_in, data\_out, full, empty);

end

endmodule  
