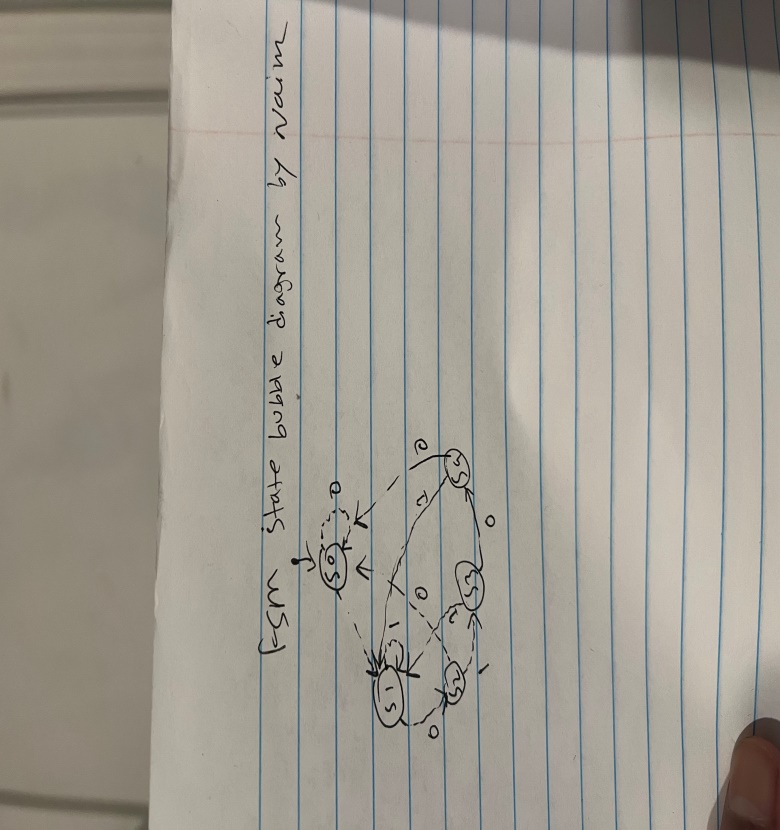
**Q1.**

****

// Pattern Detector Module with fixed output logic

module pattern\_detector(

input wire clk,

input wire rst,

input wire data\_in,

output reg pattern\_detected

);

// State encoding

parameter S0 = 3'b000;

parameter S1 = 3'b001;

parameter S2 = 3'b010;

parameter S3 = 3'b011;

parameter S4 = 3'b100;

reg [2:0] current\_state, next\_state;

// Sequential logic for state transition

always @(posedge clk or posedge rst) begin

if (rst)

current\_state <= S0;

else

current\_state <= next\_state;

end

// Combinational logic for next state

always @(\*) begin

case (current\_state)

S0: next\_state = (data\_in) ? S1 : S0;

S1: next\_state = (data\_in) ? S1 : S2;

S2: next\_state = (data\_in) ? S3 : S0;

S3: next\_state = (data\_in) ? S1 : S4;

S4: next\_state = (data\_in) ? S1 : S0;

default: next\_state = S0;

endcase

end

// output logic - purely combinational

always @(\*) begin

pattern\_detected = (current\_state == S3 && data\_in == 0);

end

endmodule

// Testbench Module

module pattern\_detector\_tb;

reg clk;

reg rst;

reg data\_in;

wire pattern\_detected;

// Instantiate the pattern detector

pattern\_detector uut (

.clk(clk),

.rst(rst),

.data\_in(data\_in),

.pattern\_detected(pattern\_detected)

);

// Clock generation

initial begin

clk = 0;

forever #5 clk = ~clk;

end

// Test stimulus

initial begin

// Initialize signals

rst = 1;

data\_in = 0;

// Wait for 100 ns

#100;

// Release reset

rst = 0;

// Test case 1: Pattern "1010"

#10 data\_in = 1;

#10 data\_in = 0;

#10 data\_in = 1;

#10 data\_in = 0; // Should detect pattern here

// Test case 2: Pattern "11010"

#10 data\_in = 1;

#10 data\_in = 1;

#10 data\_in = 0;

#10 data\_in = 1;

#10 data\_in = 0; // Should detect pattern here

// Test case 3: Pattern "10101010"

#10 data\_in = 1;

#10 data\_in = 0;

#10 data\_in = 1;

#10 data\_in = 0; // Should detect pattern here

#10 data\_in = 1;

#10 data\_in = 0; // Should detect pattern here

// End simulation after some time

#100 $finish;

end

// Monitor changes

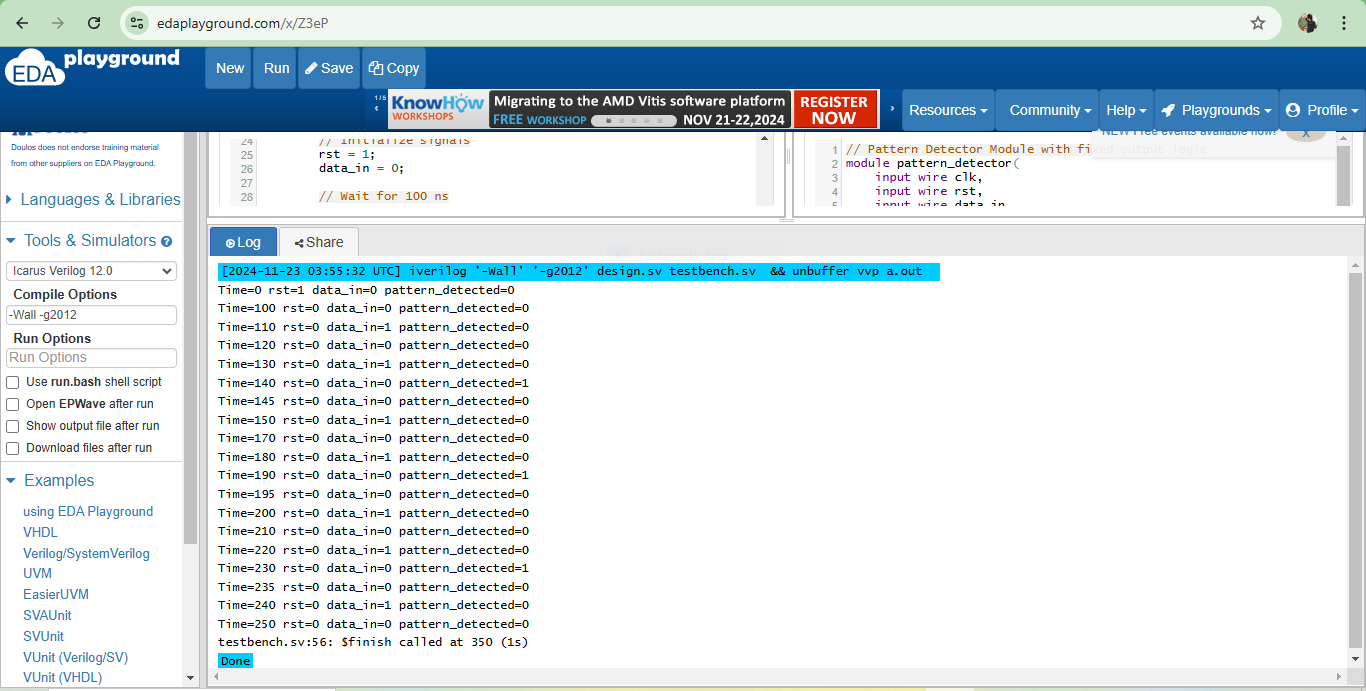
initial begin

$monitor("Time=%0t rst=%b data\_in=%b pattern\_detected=%b",

$time, rst, data\_in, pattern\_detected);

end

endmodule



**Q2.**

module fsm\_divisible\_by\_7 (

input clk,

input rst,

input bit\_in,

output reg flag

);

// State encoding for remainders 0 to 6

localparam S0 = 3'b000; // Remainder 0 (divisible by 7)

localparam S1 = 3'b001; // Remainder 1

localparam S2 = 3'b010; // Remainder 2

localparam S3 = 3'b011; // Remainder 3

localparam S4 = 3'b100; // Remainder 4

localparam S5 = 3'b101; // Remainder 5

localparam S6 = 3'b110; // Remainder 6

reg [2:0] current\_state, next\_state; // 3-bit states for remainders

// State transitions

always @(posedge clk or posedge rst) begin

if (rst)

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

else

current\_state <= next\_state; // Transition to next state

end

// Next state logic

always @(\*) begin

case (current\_state)

S0: next\_state = (bit\_in) ? S1 : S0; // 2\*0 + bit\_in

S1: next\_state = (bit\_in) ? S3 : S2; // 2\*1 + bit\_in

S2: next\_state = (bit\_in) ? S5 : S4; // 2\*2 + bit\_in

S3: next\_state = (bit\_in) ? S0 : S6; // 2\*3 + bit\_in

S4: next\_state = (bit\_in) ? S2 : S1; // 2\*4 + bit\_in

S5: next\_state = (bit\_in) ? S4 : S3; // 2\*5 + bit\_in

S6: next\_state = (bit\_in) ? S6 : S5; // 2\*6 + bit\_in

default: next\_state = S0; // Default case for safety

endcase

end

// Output logic

always @(posedge clk or posedge rst) begin

if (rst)

flag <= 0; // Reset flag on reset

else if (current\_state != S0 && next\_state == S0)

flag <= 1; // Assert flag during valid transition into S0

else

flag <= 0; // Clear flag otherwise

end

endmodule

module tb\_fsm\_divisible\_by\_7;

reg clk, rst, bit\_in;

wire flag;

// Instantiate the FSM

fsm\_divisible\_by\_7 uut (

.clk(clk),

.rst(rst),

.bit\_in(bit\_in),

.flag(flag)

);

// Clock generation

initial begin

clk = 0;

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

end

// Test stimulus

initial begin

rst = 1;

bit\_in = 0;

#10 rst = 0;

// Input bit stream: 111 (7 in decimal, divisible by 7)

#10 bit\_in = 1;

#10 bit\_in = 1;

#10 bit\_in = 1;

// Input bit stream: 1001 (9 in decimal, not divisible by 7)

#10 bit\_in = 1;

#10 bit\_in = 0;

#10 bit\_in = 0;

#10 bit\_in = 1;

// Input bit stream: 1110 (14 in decimal, divisible by 7)

#10 bit\_in = 1;

#10 bit\_in = 1;

#10 bit\_in = 1;

#10 bit\_in = 0;

#20 $finish; // Stop the simulation

end

// Monitor output

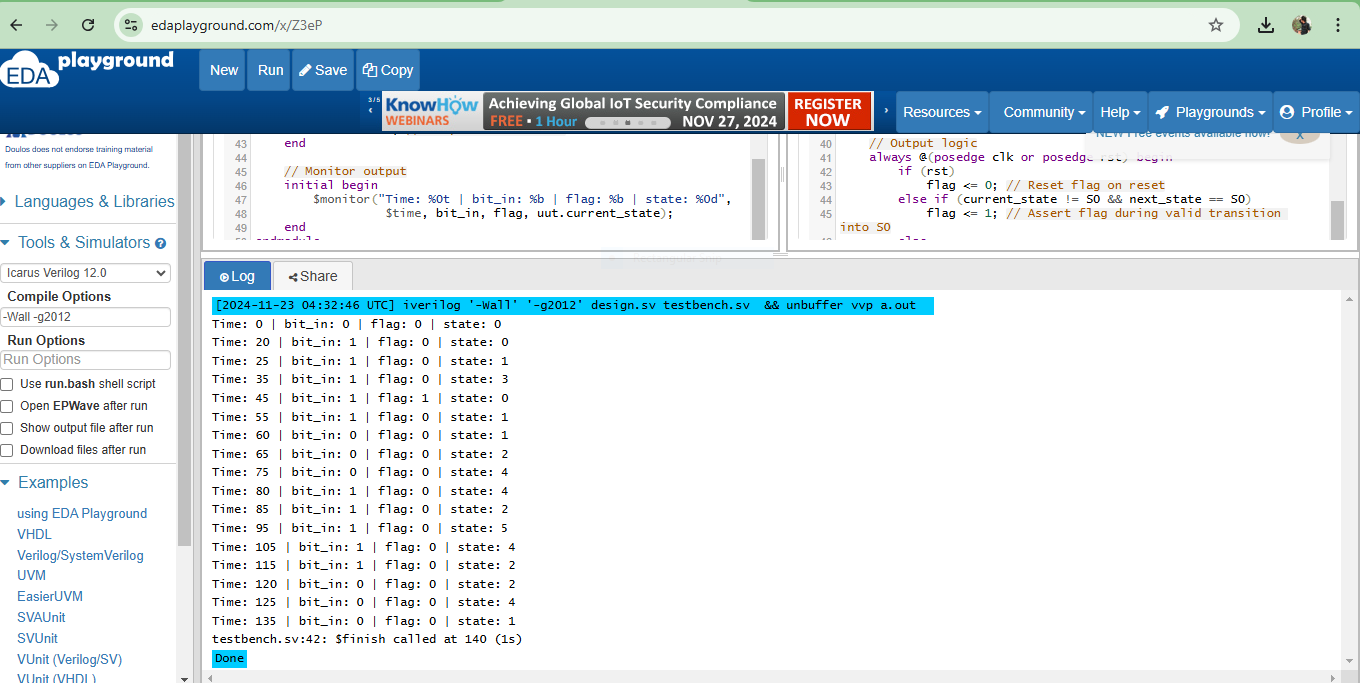
initial begin

$monitor("Time: %0t | bit\_in: %b | flag: %b | state: %0d",

$time, bit\_in, flag, uut.current\_state);

end

endmodule



**Q3.**

`timescale 1ns / 1ps

module duty\_cycle\_calculator (

input wire pwm\_signal, // Input PWM signal

input wire clk, // High-frequency clock

input wire reset, // Reset signal

output reg [7:0] duty\_cycle // Duty cycle as percentage (0-100)

);

reg [31:0] total\_cycles = 0; // Counts the total cycles in one period

reg [31:0] high\_cycles = 0; // Counts the high cycles in one period

reg pwm\_signal\_last = 0; // Tracks the last state of the PWM signal

always @(posedge clk or posedge reset) begin

if (reset) begin

// Reset all counters and output

total\_cycles <= 0;

high\_cycles <= 0;

duty\_cycle <= 0;

pwm\_signal\_last <= 0;

end else begin

// Increment total cycle counter

total\_cycles <= total\_cycles + 1;

// Increment high cycle counter if PWM signal is high

if (pwm\_signal) begin

high\_cycles <= high\_cycles + 1;

end

// Detect rising edge of the PWM signal (end of a period)

if (pwm\_signal && !pwm\_signal\_last) begin

// Calculate duty cycle after a full period

if (total\_cycles > 0) begin

duty\_cycle <= ((high\_cycles \* 100) + total\_cycles - 1) / total\_cycles; // Proper rounding logic

end else begin

duty\_cycle <= 0; // Default to 0% if no valid period

end

// Reset counters for the next period

total\_cycles <= 0;

high\_cycles <= 0;

end

// Update the last state of the PWM signal

pwm\_signal\_last <= pwm\_signal;

end

end

endmodule

`timescale 1ns / 1ps

module tb\_duty\_cycle\_calculator;

reg pwm\_signal; // Input PWM signal

reg clk; // High-frequency clock

reg reset; // Reset signal

wire [7:0] duty\_cycle; // Output duty cycle

// Instantiate the DUT (Device Under Test)

duty\_cycle\_calculator dut (

.pwm\_signal(pwm\_signal),

.clk(clk),

.reset(reset),

.duty\_cycle(duty\_cycle)

);

// Clock generation (500 MHz clock)

initial clk = 0;

always #1 clk = ~clk; // Clock period of 2 ns

// PWM signal generation variables

integer i;

// PWM signal generation

initial begin

// Initialize signals

pwm\_signal = 0;

reset = 1;

#20 reset = 0; // De-assert reset after 20 ns

// \*\*Simulate 10% duty cycle\*\*

$display("\nStarting 10%% duty cycle test");

for (i = 0; i < 10; i = i + 1) begin

pwm\_signal = 1; #20; // High for 20 ns

pwm\_signal = 0; #180; // Low for 180 ns

end

#2000; // Wait time to allow calculation

$display("10%% Duty Cycle: %0d%%", duty\_cycle);

// \*\*Simulate 50% duty cycle\*\*

$display("\nStarting 50%% duty cycle test");

for (i = 0; i < 10; i = i + 1) begin

pwm\_signal = 1; #100; // High for 100 ns

pwm\_signal = 0; #100; // Low for 100 ns

end

#2000; // Wait time to allow calculation

$display("50%% Duty Cycle: %0d%%", duty\_cycle);

// \*\*Simulate 90% duty cycle\*\*

$display("\nStarting 90%% duty cycle test");

for (i = 0; i < 10; i = i + 1) begin

pwm\_signal = 1; #180; // High for 180 ns

pwm\_signal = 0; #20; // Low for 20 ns

end

#2000; // Wait time to allow calculation

$display("90%% Duty Cycle: %0d%%", duty\_cycle);

$finish; // End simulation

end

endmodule

