**Q1.**

// RTL Level Design

module pattern\_detector\_rtl (

input [7:0] data,

output reg zeroflag,

output reg oneflag

);

always @(\*) begin

zeroflag = (data == 8'b00000000);

oneflag = (data == 8'b11111111);

end

endmodule

// Gate Level Design

module pattern\_detector\_gate (

input [7:0] data,

output zeroflag,

output oneflag

);

// For zeroflag: NOR all bits together

wire [6:0] nor\_stages;

nor(nor\_stages[0], data[0], data[1]);

nor(nor\_stages[1], data[2], data[3]);

nor(nor\_stages[2], data[4], data[5]);

nor(nor\_stages[3], data[6], data[7]);

and(nor\_stages[4], nor\_stages[0], nor\_stages[1]);

and(nor\_stages[5], nor\_stages[2], nor\_stages[3]);

and(zeroflag, nor\_stages[4], nor\_stages[5]);

// For oneflag: AND all bits together

wire [6:0] and\_stages;

and(and\_stages[0], data[0], data[1]);

and(and\_stages[1], data[2], data[3]);

and(and\_stages[2], data[4], data[5]);

and(and\_stages[3], data[6], data[7]);

and(and\_stages[4], and\_stages[0], and\_stages[1]);

and(and\_stages[5], and\_stages[2], and\_stages[3]);

and(oneflag, and\_stages[4], and\_stages[5]);

endmodule

**// Testbench**

module pattern\_detector\_tb;

reg [7:0] data;

wire zeroflag\_rtl, oneflag\_rtl;

wire zeroflag\_gate, oneflag\_gate;

// Instantiate both designs

pattern\_detector\_rtl rtl\_design (

.data(data),

.zeroflag(zeroflag\_rtl),

.oneflag(oneflag\_rtl)

);

pattern\_detector\_gate gate\_design (

.data(data),

.zeroflag(zeroflag\_gate),

.oneflag(oneflag\_gate)

);

// Test vector generation and checking

initial begin

$monitor("Time=%0t data=%b zeroflag\_rtl=%b oneflag\_rtl=%b zeroflag\_gate=%b oneflag\_gate=%b",

$time, data, zeroflag\_rtl, oneflag\_rtl, zeroflag\_gate, oneflag\_gate);

// Test case 1: All zeros

data = 8'b00000000;

#10;

// Test case 2: All ones

data = 8'b11111111;

#10;

// Test case 3: Mixed pattern 1

data = 8'b10101010;

#10;

// Test case 4: Mixed pattern 2

data = 8'b01010101;

#10;

// Test case 5: Single one

data = 8'b00000001;

#10;

// Test case 6: Single zero

data = 8'b11111110;

#10;

$finish;

end

// Check for mismatches between RTL and gate-level designs

always @(data) begin

#1; // Small delay to let outputs settle

if (zeroflag\_rtl !== zeroflag\_gate)

$display("Mismatch in zeroflag at time %0t: RTL=%b, Gate=%b",

$time, zeroflag\_rtl, zeroflag\_gate);

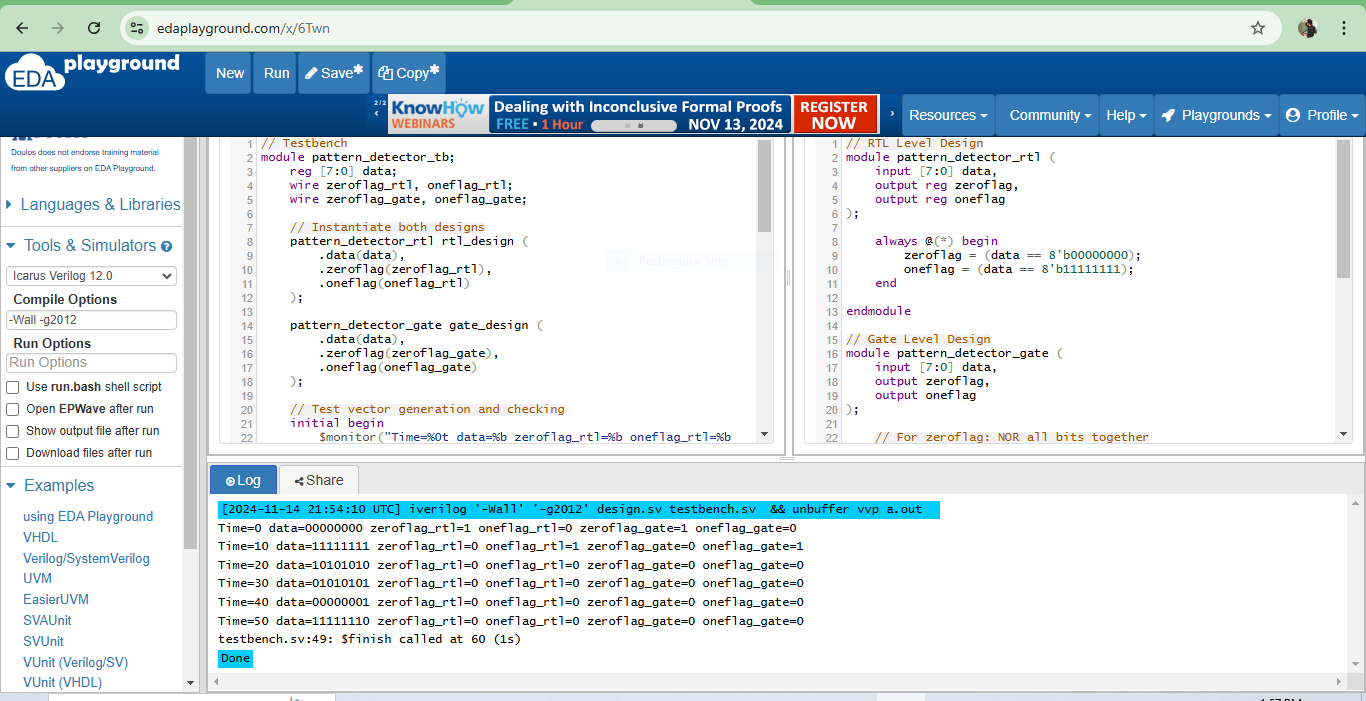
if (oneflag\_rtl !== oneflag\_gate)

$display("Mismatch in oneflag at time %0t: RTL=%b, Gate=%b",

$time, oneflag\_rtl, oneflag\_gate);

end

endmodule



**Q2.**

`timescale 1ns / 1ps

// 4-bit 2's Complement Number Converter

module TwosComplement4Bit(

input [3:0] binary\_in, // 4-bit binary input

output [3:0] complement\_out // 4-bit 2's complement output

);

wire [3:0] inverted; // Wire to hold inverted bits

wire [3:0] carry; // Wires to hold carry bits for addition

wire sum1, carry1\_temp1, carry1\_temp2;

wire sum2, carry2\_temp1, carry2\_temp2;

wire sum3, carry3\_temp1, carry3\_temp2;

// Step 1: Invert the input bits

not (inverted[0], binary\_in[0]);

not (inverted[1], binary\_in[1]);

not (inverted[2], binary\_in[2]);

not (inverted[3], binary\_in[3]);

// Step 2: Add 1 to the inverted bits to compute 2's complement

// Using a 4-bit ripple-carry adder structure

// Full adder for bit 0

xor (complement\_out[0], inverted[0], 1'b1);

and (carry[0], inverted[0], 1'b1);

// Full adder for bit 1

xor (sum1, inverted[1], carry[0]);

xor (complement\_out[1], sum1, 1'b0);

and (carry1\_temp1, inverted[1], carry[0]);

and (carry1\_temp2, sum1, 1'b0);

or (carry[1], carry1\_temp1, carry1\_temp2);

// Full adder for bit 2

xor (sum2, inverted[2], carry[1]);

xor (complement\_out[2], sum2, 1'b0);

and (carry2\_temp1, inverted[2], carry[1]);

and (carry2\_temp2, sum2, 1'b0);

or (carry[2], carry2\_temp1, carry2\_temp2);

// Full adder for bit 3

xor (sum3, inverted[3], carry[2]);

xor (complement\_out[3], sum3, 1'b0);

and (carry3\_temp1, inverted[3], carry[2]);

and (carry3\_temp2, sum3, 1'b0);

or (carry[3], carry3\_temp1, carry3\_temp2);

endmodule

**//tb**

`timescale 1ns / 1ps

module TwosComplement4Bit\_tb;

reg [3:0] binary\_in; // Input to the module

wire [3:0] complement\_out; // Output from the module

// Instantiate the module

TwosComplement4Bit uut (

.binary\_in(binary\_in),

.complement\_out(complement\_out)

);

initial begin

// Monitor values

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

// Apply test cases

binary\_in = 4'b0000; #10; // Input: 0

binary\_in = 4'b0001; #10; // Input: 1

binary\_in = 4'b0010; #10; // Input: 2

binary\_in = 4'b0111; #10; // Input: 7

binary\_in = 4'b1000; #10; // Input: -8

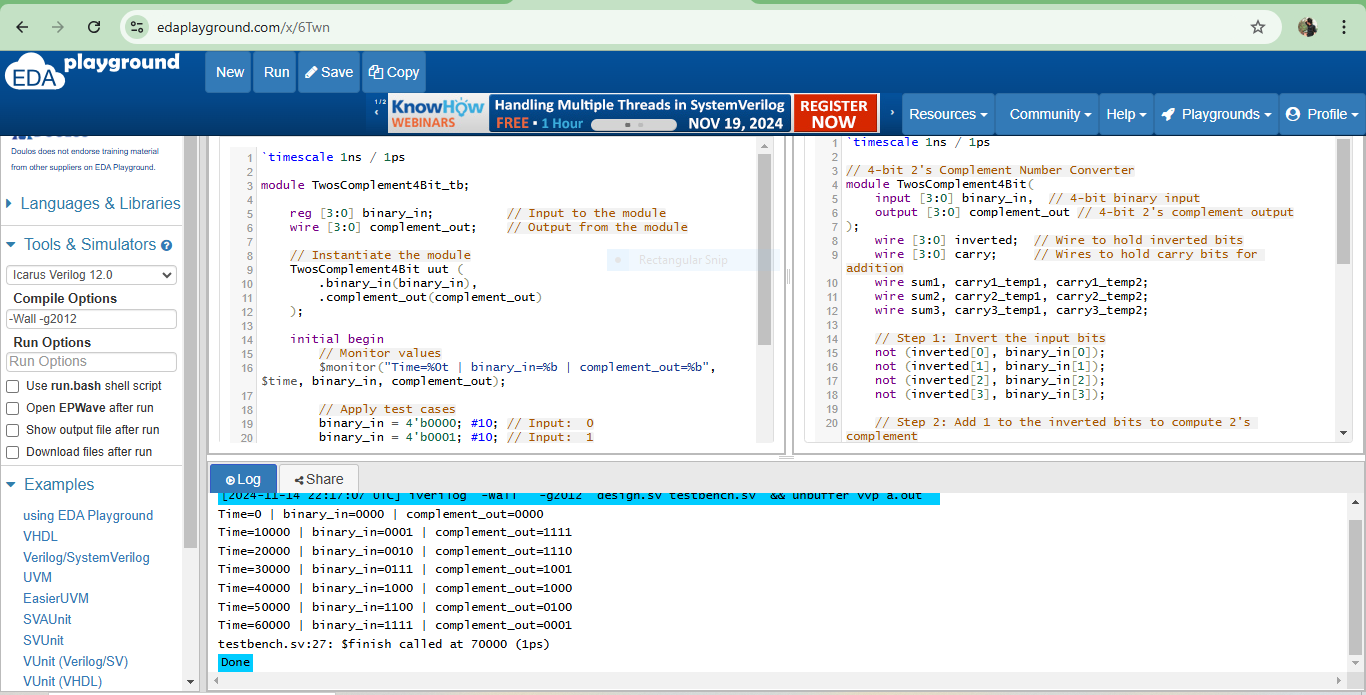
binary\_in = 4'b1100; #10; // Input: -4

binary\_in = 4'b1111; #10; // Input: -1

$finish; // End simulation

end

endmodule



**Q3.**

// 4-to-1 Multiplexer in continuous assignment

module Mux4to1(

input [3:0] data\_in, // 4-bit input data lines

input [1:0] sel, // 2-bit select line

output out // Output of the mux

);

assign out = (sel == 2'b00) ? data\_in[0] :

(sel == 2'b01) ? data\_in[1] :

(sel == 2'b10) ? data\_in[2] :

data\_in[3];

endmodule

**// Testbench for 4-to-1 Multiplexer**

module Mux4to1\_tb;

reg [3:0] data\_in; // 4-bit input data lines

reg [1:0] sel; // 2-bit select line

wire out; // Output of the mux

// Instantiate the Mux4to1 module

Mux4to1 uut (

.data\_in(data\_in),

.sel(sel),

.out(out)

);

initial begin

$monitor("Time=%0t | data\_in=%b | sel=%b | out=%b", $time, data\_in, sel, out);

// Test case 1: sel = 2'b00

data\_in = 4'b1010; sel = 2'b00; #10;

// Test case 2: sel = 2'b01

data\_in = 4'b1010; sel = 2'b01; #10;

// Test case 3: sel = 2'b10

data\_in = 4'b1010; sel = 2'b10; #10;

// Test case 4: sel = 2'b11

data\_in = 4'b1010; sel = 2'b11; #10;

// Test case with x and z in select bits

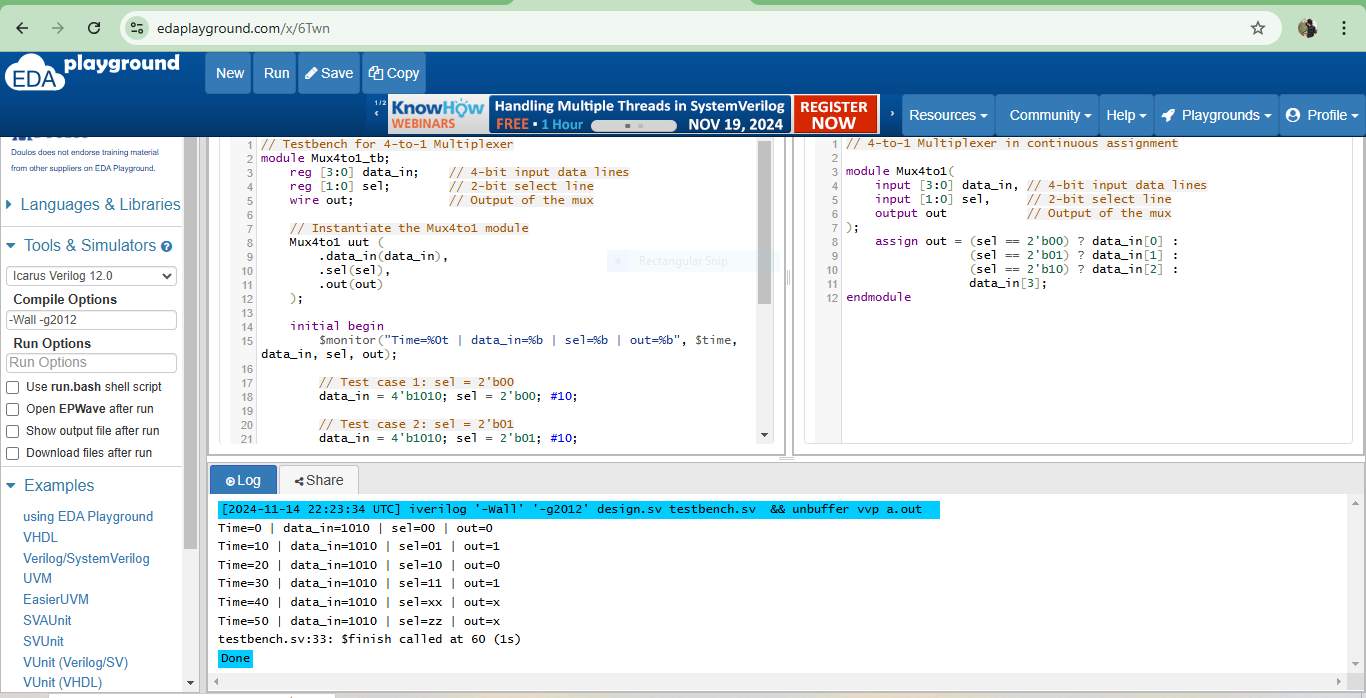
sel = 2'bx; #10;

sel = 2'bz; #10;

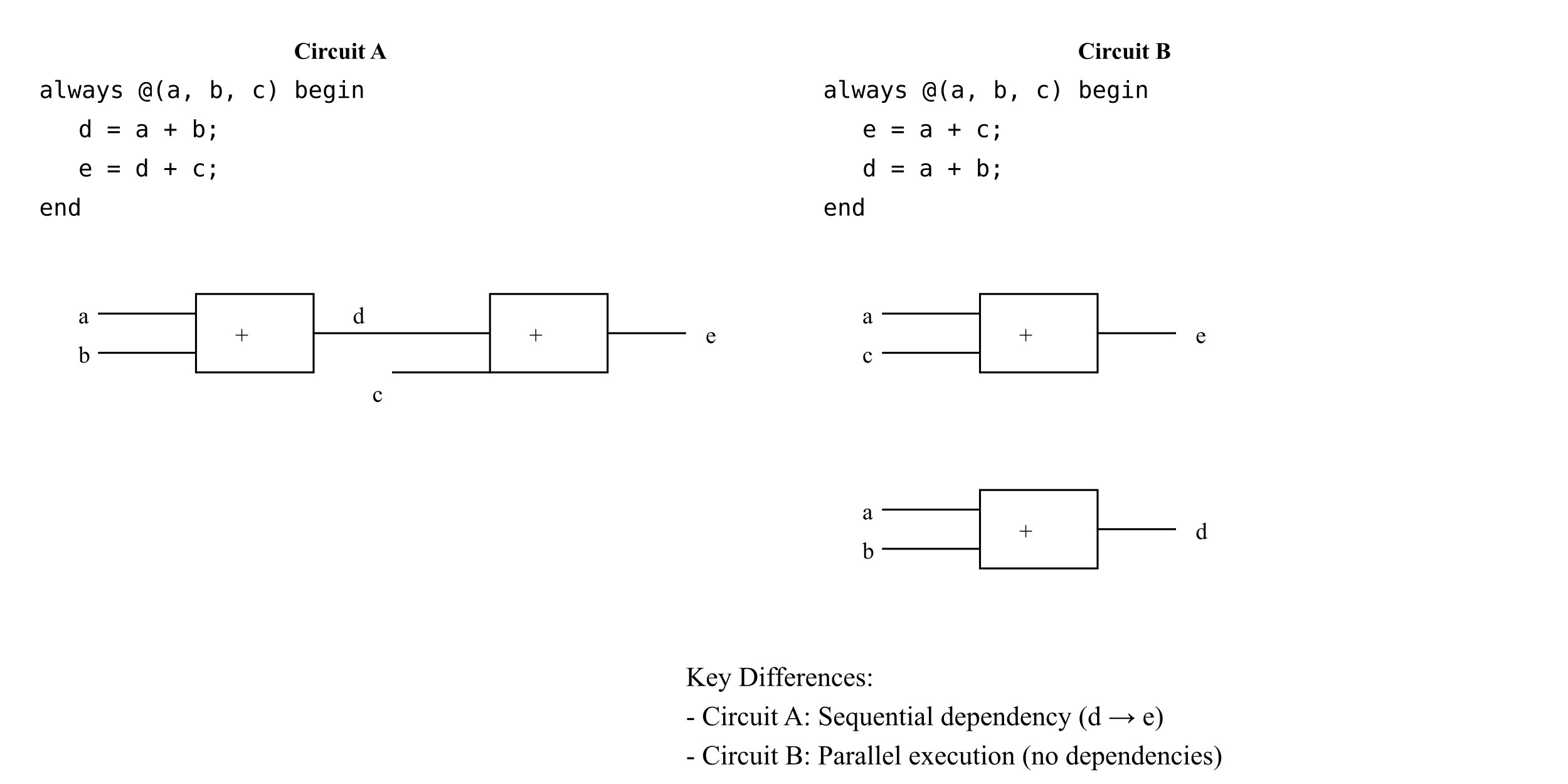
$finish;

end

endmodule



**Q4.**



**Q5.**

**Fixed code that prevents latches:**

module mux(a,b,c,d,e,sel,out);

input a,b,c,d,e;

input [2:0] sel;

output out;

reg out;

always @(\*) begin

case(sel)

3'b000: out=a;

3'b001: out=b;

3'b010: out=c;

3'b011: out=d;

3'b100: out=e;

default: out=1'b0; // Cover all remaining cases

endcase

end

endmodule

**Hardware schematic comparison:**

1. **Original design:**

* Will contain latches because output is not defined for all input conditions
* Latches will hold previous value when sel is 3'b101, 3'b110, or 3'b111
* More complex hardware due to additional latches

1. **Fixed design:**

* Pure combinational logic
* Only multiplexer circuitry without latches
* Simpler and more efficient hardware implementation
* Defined output for all possible input conditions

