San Francisco Bay University  
EE461 Verilog-HDL  
2024 Fall Midterm Exam  
Student Name: Naim Student ID: 20176

**Q1.**

**FIXED CODE:**

module example\_module;

reg a; // Changed 'time' to 'reg' for appropriate signal declaration

reg b, c;

// Procedural block instead of assign for reg variables

always @(\*) begin

a = (b) ? b : c;

end

endmodule

**Key Fixes:**

* Changed the module name from module module; to module example\_module;.
* Changed the type of a from time to reg.
* Used an always @(\*) block for the conditional assignment to a because a is declared as a reg type.
* Removed the incorrect use of assign and begin

**Q2.**

`timescale 100ps / 100ps

module sampleDesign (z, x1, x2);

input x1, x2;

output z;

// NOR gate with a delay of 3.1415 time units (314.15 ps)

nor #3.1415 (z, x1, x2);

endmodule

**Delay time:**

* nor #3.1415 (z, x1, x2); specifies a delay of **3.1415 time units** for the NOR gate.
* Since the time unit is **100 ps**, the actual delay can be calculated as:

Delay=3.1415×100 ps=314.15 ps.

**Q3.**

module case\_equality\_test;

reg [3:0] a, b;

wire equal, not\_equal;

// Case equality and inequality assignments

assign equal = (a === b);

assign not\_equal = (a !== b);

initial begin

// Initialize values

a = 4'b01xz;

b = 4'bxz10;

// Display initial values

$display("Time=%0t", $time);

$display("a = %b", a);

$display("b = %b", b);

// Display results

#1 $display("a === b is %b", equal);

$display("a !== b is %b", not\_equal);

// Test with equal values

#1;

a = 4'b01xz;

b = 4'b01xz;

$display("\nTesting with equal values:");

$display("a = %b", a);

$display("b = %b", b);

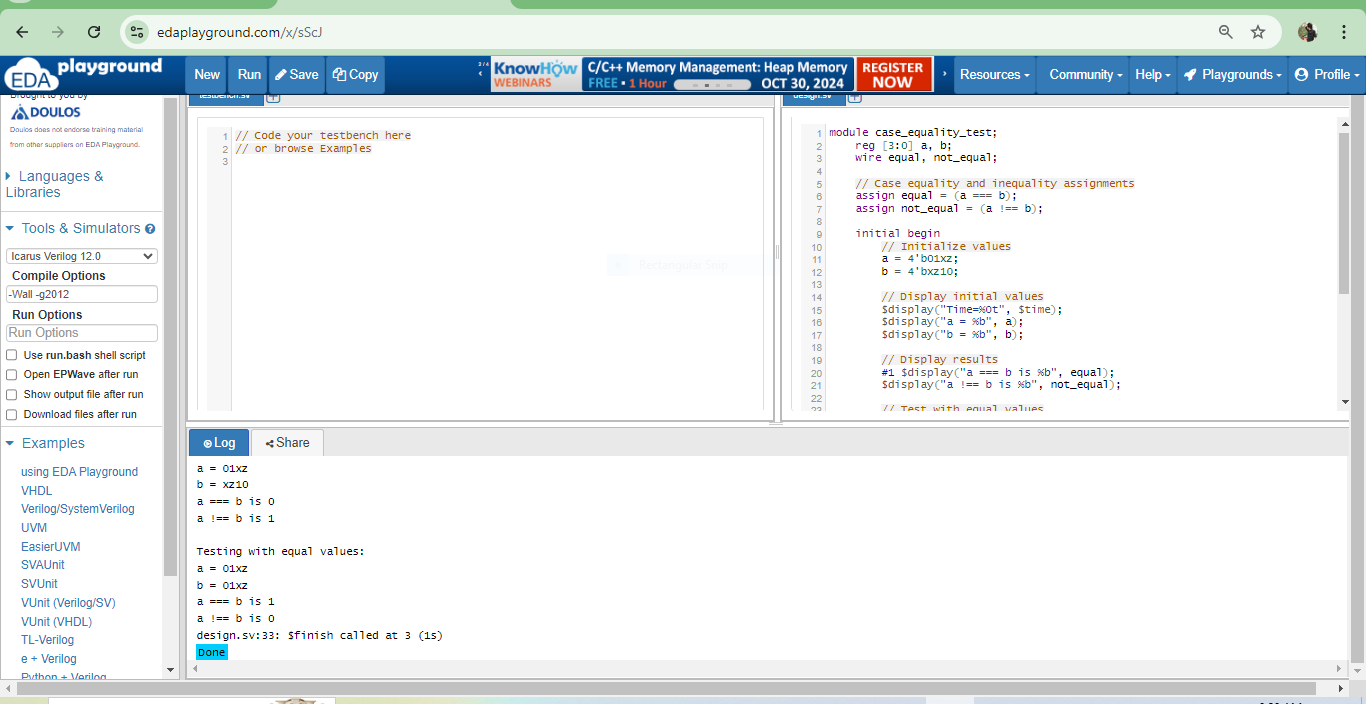
$display("a === b is %b", equal);

$display("a !== b is %b", not\_equal);

#1 $finish;

end

endmodule



* For the first case (different values):

a = = = b is 0

a!= = b is 1

* For the second case (equal values):

a = = = b is 1

a!= = b is 0

**Q4.**

module not\_operations\_test;

reg [3:0] a;

wire [3:0] bitwise\_not;

wire logical\_not;

assign bitwise\_not = ~a;

assign logical\_not = !a;

initial begin

a = 4'bxz01;

#1;

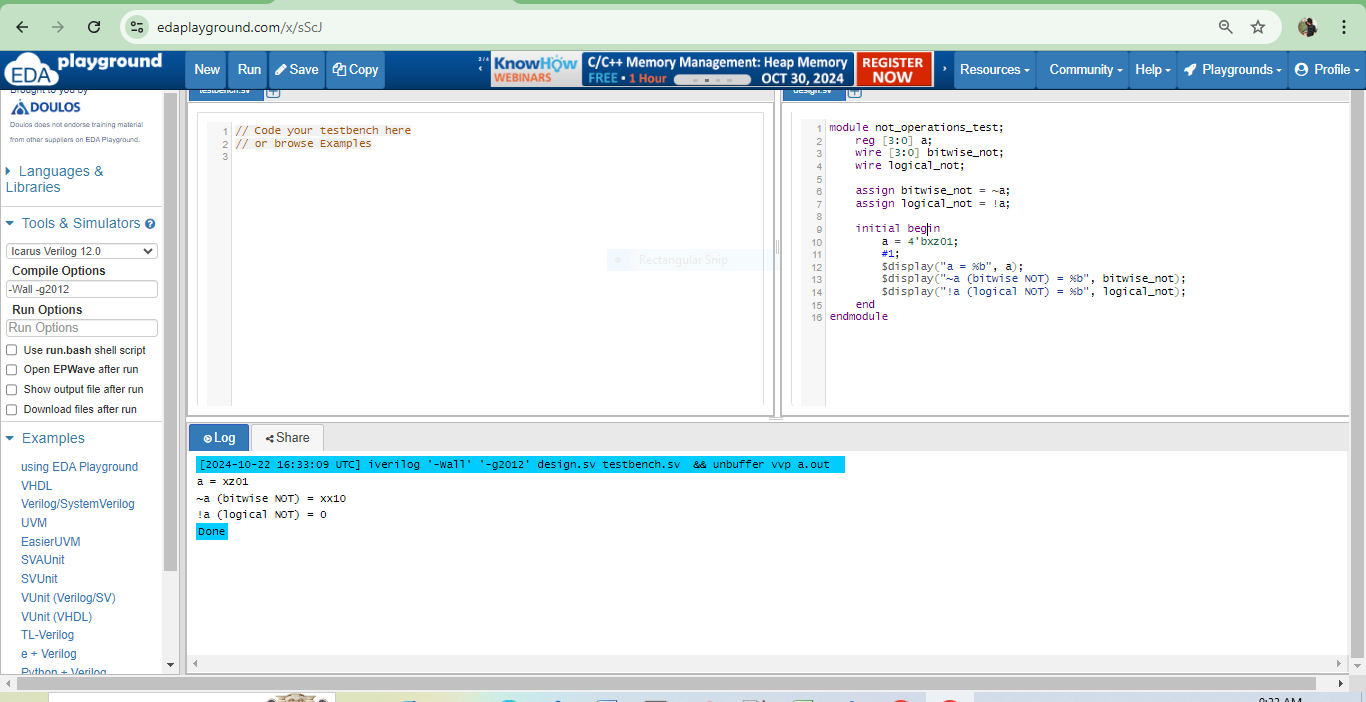
$display("a = %b", a);

$display("~a (bitwise NOT) = %b", bitwise\_not);

$display("!a (logical NOT) = %b", logical\_not);

end

endmodule



1. **Given: a = 4'bxz01**
2. **For ~a (bitwise NOT):**

* Performs bitwise inversion
* x inverts to x
* z inverts to z
* 0 inverts to 1
* 1 inverts to 0

So ~a = 4'bxz10

1. **For !a (logical NOT):**

* If any bit is 1, result is 0
* If all bits are 0, result is 1
* If no bits are 1 and any bit is x or z, result is x
* In this case, we have a 1 bit in position 0

Therefore:

* ~a = 4'bxz10 (bitwise NOT)
* !a = 1'b0 (logical NOT)

**Q5.**

module shift\_test;

reg[3:0] a;

wire[3:0] b, c;

assign b = a << 2;

assign c = 2 << a;

initial begin

a = 4'b0010;

#1;

$display("a = %b (%d)", a, a);

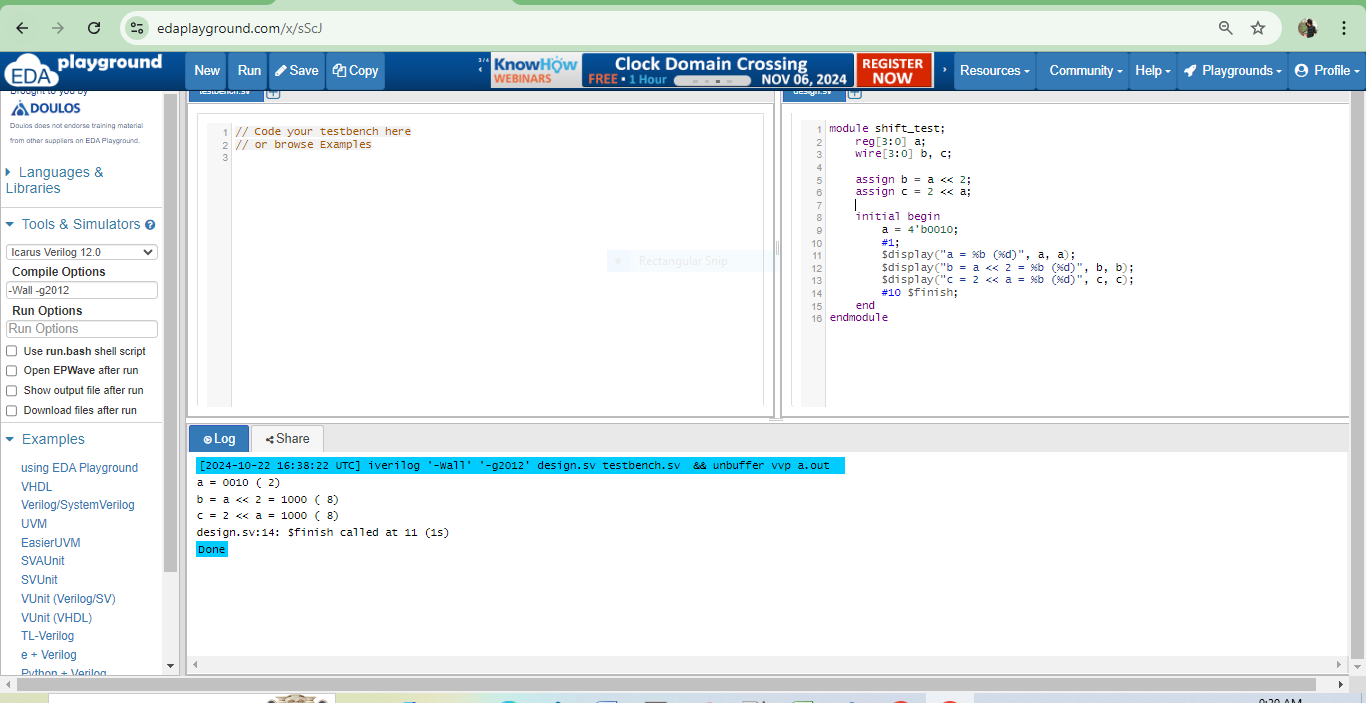
$display("b = a << 2 = %b (%d)", b, b);

$display("c = 2 << a = %b (%d)", c, c);

#10 $finish;

end

endmodule



**Final Values:**

* **b = 4'b1000** (8 in decimal)
* **c = 4'b1000** (8 in decimal)

Both b and c will have the same value of 4'b1000.

**Q6.**

primitive agree(going, std1, std2, std3);

output going;

input std1, std2, std3;

// Truth table

table

// std1 std2 std3 : going

0 0 0 : 0; // No one wants to go

0 0 1 : 1; // One person wants to go

0 1 0 : 1; // One person wants to go

0 1 1 : 1; // Majority wants to go

1 0 0 : 1; // One person wants to go

1 0 1 : 1; // Majority wants to go

1 1 0 : 1; // Majority wants to go

1 1 1 : 1; // Everyone wants to go

endtable

endprimitive

// Test module to verify the UDP

module test\_agree;

wire going;

reg std1, std2, std3;

// Instantiate the UDP

agree vote(going, std1, std2, std3);

initial begin

// Test all combinations

std1 = 0; std2 = 0; std3 = 0;

#5 $display("Case 000: going = %b", going);

std1 = 0; std2 = 0; std3 = 1;

#5 $display("Case 001: going = %b", going);

std1 = 0; std2 = 1; std3 = 0;

#5 $display("Case 010: going = %b", going);

std1 = 0; std2 = 1; std3 = 1;

#5 $display("Case 011: going = %b", going);

std1 = 1; std2 = 0; std3 = 0;

#5 $display("Case 100: going = %b", going);

std1 = 1; std2 = 0; std3 = 1;

#5 $display("Case 101: going = %b", going);

std1 = 1; std2 = 1; std3 = 0;

#5 $display("Case 110: going = %b", going);

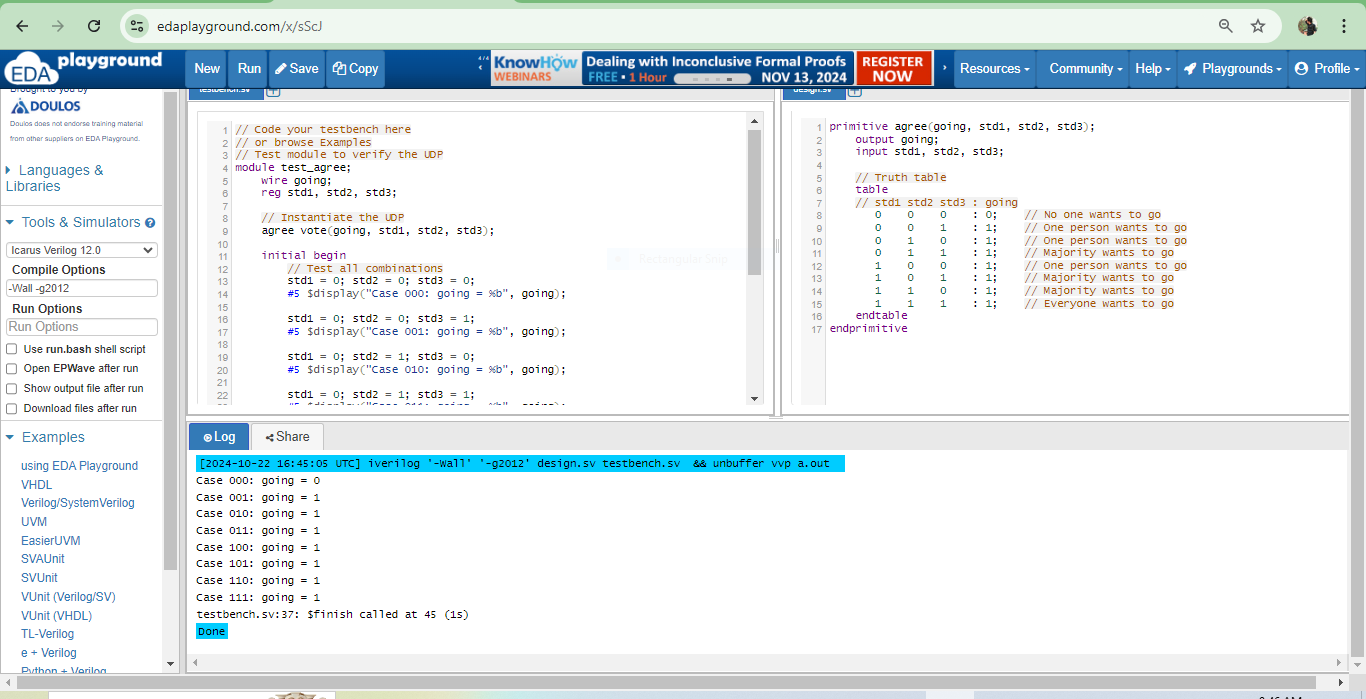
std1 = 1; std2 = 1; std3 = 1;

#5 $display("Case 111: going = %b", going);

#5 $finish;

end

endmodule



* Only when all students vote 0 (Case 000), the output is 0 (not going)
* In all other cases (7 combinations), the output is 1 (going)

**Q7.**

### Corrected RTL Module

module test;

reg a; // Change wire a to reg

reg [1:0] b; // Change wire b to reg

reg [2:0] c; // Change wire c to reg

reg [3:0] d; // Change wire d to reg

always @(posedge c or a or b) begin

c <= a + b; // Use non-blocking assignment

d <= b + c; // d gets the updated value of c

a <= a + d; // a gets the updated value of d

end

endmodule

**Changes Made:**

**Changed wire to reg**:

* a, b, c, and d are changed to reg types since they are assigned in an always block.

**Sensitivity List Update**:

* The sensitivity list now uses posedge c, which implies that the block is triggered on the rising edge of c, along with changes in a and b.

**Removed Continuous Assignment**:

* Removed the assign keyword since all assignments are now handled within the always block.

**Corrected Non-blocking Assignments**:

* Used non-blocking assignments (<=) for sequential logic.

**Q8**.

module CalculateOutput (

input wire [31:0] x, // 32-bit input x

input wire [31:0] y, // 32-bit input y

output reg [31:0] out // 32-bit output out

);

// Intermediate wires for shifted values (using addition)

wire [31:0] add\_2x; // 2x

wire [31:0] add\_4x; // 4x

wire [31:0] add\_8x; // 8x

wire [31:0] add\_16y; // 16y

// Calculate 2x, 4x, and 8x using adders

assign add\_2x = x + x; // 2x

assign add\_4x = add\_2x + add\_2x; // 4x

assign add\_8x = add\_4x + add\_4x; // 8x

// Calculate 14x = 8x + 4x + 2x using adders

wire [31:0] temp\_14x;

assign temp\_14x = add\_8x + add\_4x + add\_2x; // 14x

// Calculate 16y using a series of adders

assign add\_16y = y + y + y + y + y + y + y + y +

y + y + y + y + y + y + y + y; // 16y

// Final output calculation

always @(\*) begin

out = temp\_14x + add\_16y; // out = 14x + 16y

end

endmodule

module testbench;

reg [31:0] x; // Test input x

reg [31:0] y; // Test input y

wire [31:0] out; // Output from CalculateOutput

// Instantiate the CalculateOutput module

CalculateOutput uut (

.x(x),

.y(y),

.out(out)

);

initial begin

// Test case 1

x = 1; y = 1; #10; // Wait for some time

$display("x = %d, y = %d, out = %d", x, y, out); // Should print 30

// Test case 2

x = 2; y = 2; #10;

$display("x = %d, y = %d, out = %d", x, y, out); // Should print 60

// Test case 3

x = 0; y = 5; #10;

$display("x = %d, y = %d, out = %d", x, y, out); // Should print 80

// Test case 4

x = 3; y = 0; #10;

$display("x = %d, y = %d, out = %d", x, y, out); // Should print 42

// Test case 5

x = 4; y = 1; #10;

$display("x = %d, y = %d, out = %d", x, y, out); // Should print 72

// Test case 6

x = 10; y = 5; #10;

$display("x = %d, y = %d, out = %d", x, y, out); // Should print 220

// Test case 7

x = 5; y = 10; #10;

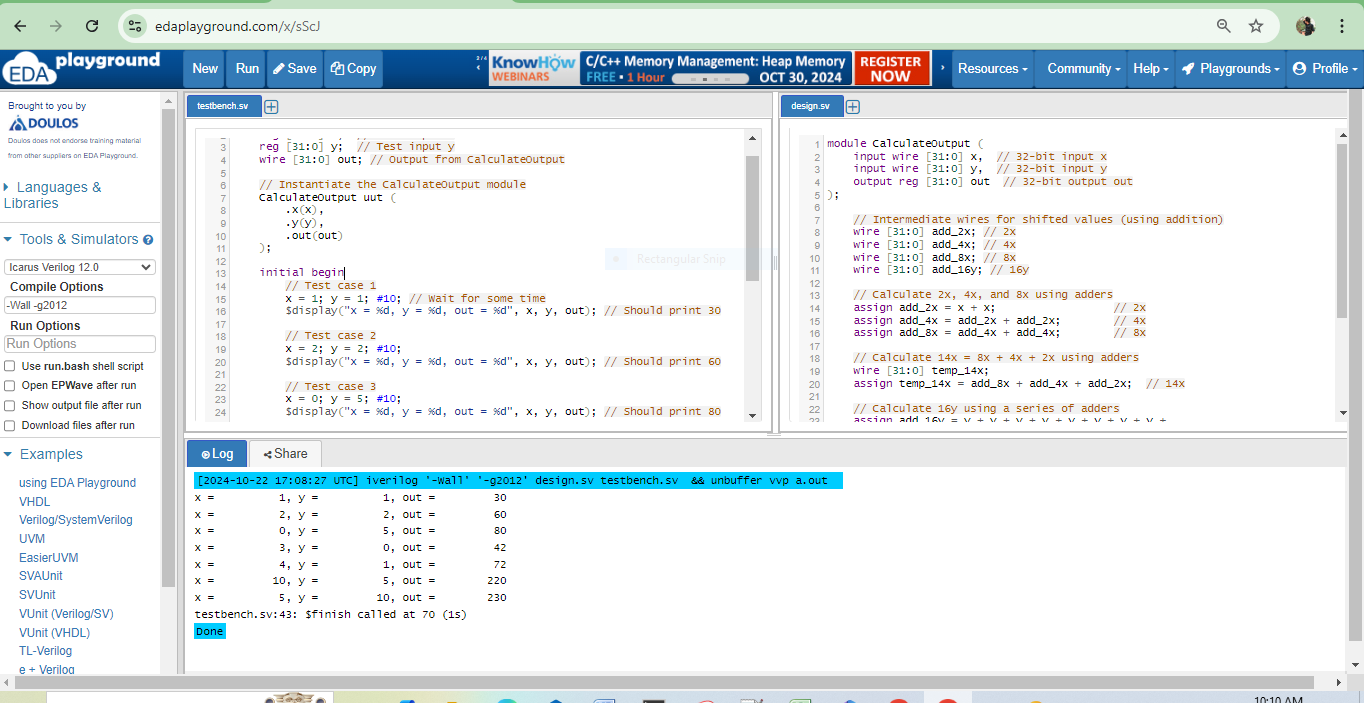
$display("x = %d, y = %d, out = %d", x, y, out); // Should print 230

// Finish simulation

$finish;

end

endmodule



**Q9.**

Given code,

module test1;

wire a;

reg b, c, d;

assign a = (b) ? c : d; // Continuous assignment

endmodule

module test;

reg a;

reg b, c, d;

always @(\*) begin // Always block

if (b)

a = c;

else

a = d;

end

endmodule

### Key Differences

| **Feature** | **Continuous Assignment** | **Always Block** |
| --- | --- | --- |
| **Output Type** | Output a is a wire | Output a is a reg |
| **Driving Mechanism** | Continuously drives a based on inputs (b, c, d) | Drives a based on the conditions evaluated in the block |
| **Sensitivity** | Implicitly sensitive to changes in b, c, and d | Explicitly sensitive to all inputs used inside the block (using @(\*)) |
| **Behavior** | Immediate update of a whenever b, c, or d changes | Update of a occurs only when entering the always block after changes to b, c, or d |
| **Latch Potential** | No potential for latches, as a is always driven | Potential for latches if not all paths in the always block assign a value to a |
| **Use Case** | Suitable for simple conditional assignments | Suitable for more complex conditions and procedural logic |

**Q10.**

module gray\_counter (

input wire clk, // Clock signal

input wire reset, // Asynchronous reset signal

output reg [3:0] gray // 4-bit output for Gray code

);

// Internal state variable for counting (3 bits are enough for 0 to 6)

reg [2:0] state;

// State machine to handle counting

always @(posedge clk or posedge reset) begin

if (reset) begin

state <= 3'b000; // Reset state to 0

gray <= 4'b0000; // Reset Gray output to 0

end else begin

// Increment state but limit to 6

if (state < 3'd6) begin

state <= state + 1; // Increment state

end

// Convert state to corresponding Gray code

case (state)

3'd0: gray <= 4'b0000; // Gray 0

3'd1: gray <= 4'b0001; // Gray 1

3'd2: gray <= 4'b0011; // Gray 2

3'd3: gray <= 4'b0010; // Gray 3

3'd4: gray <= 4'b0110; // Gray 4

3'd5: gray <= 4'b0111; // Gray 5

3'd6: gray <= 4'b0101; // Gray 6

default: gray <= 4'b0000; // Default case (not needed)

endcase

end

end

endmodule

module tb\_gray\_counter;

reg clk;

reg reset;

wire [3:0] gray;

// Instantiate the Gray counter

gray\_counter gc (

.clk(clk),

.reset(reset),

.gray(gray)

);

// Clock generation

initial begin

clk = 0;

forever #5 clk = ~clk; // Generate a clock with a 10-time unit period

end

// Test sequence

initial begin

reset = 1; // Start with reset

#10 reset = 0; // Release reset

#100; // Run for some time (adjust as needed)

$finish; // End the simulation

end

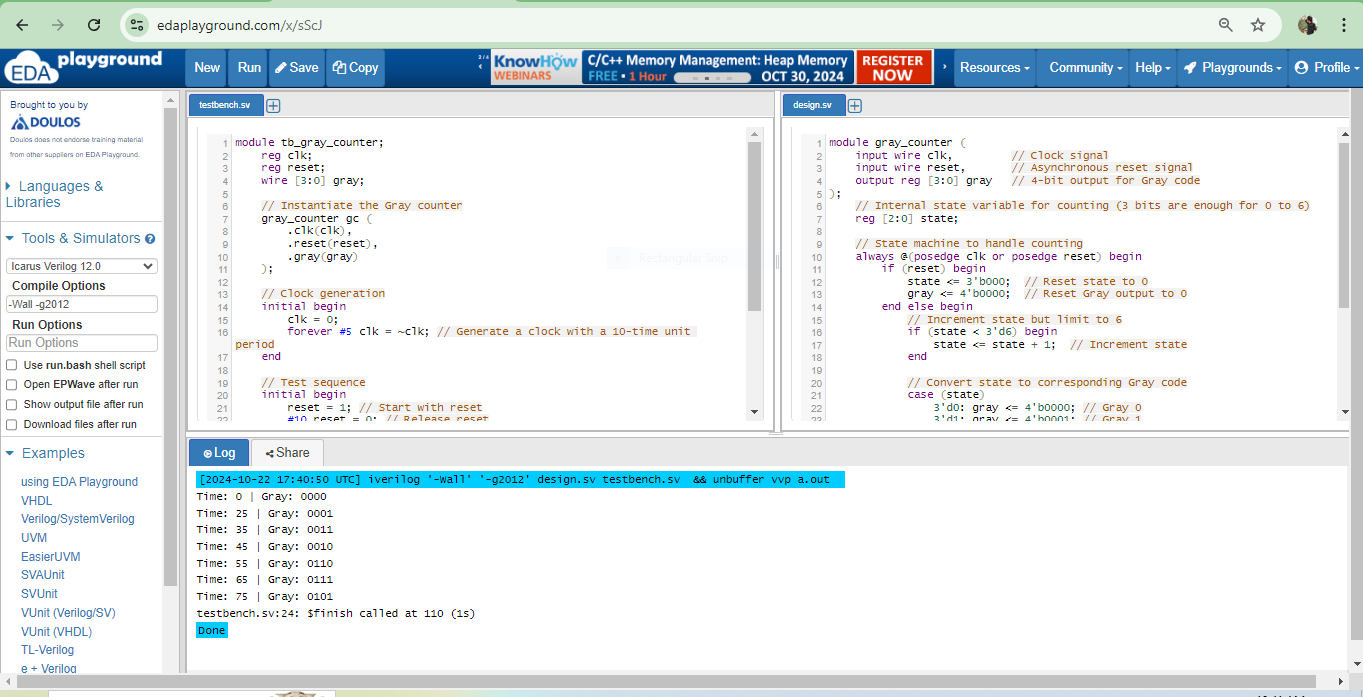
// Monitor output

initial begin

$monitor("Time: %0t | Gray: %b", $time, gray);

end

endmodule



**Q11.**

### Corrected bad design Module

module test (

input reg a,

input reg b,

input reg en,

output reg c

);

always @(\*) begin // Using a combinational always block

if (en) begin

c = a | b; // Perform bitwise OR when enabled

end else begin

c = 1'b0; // Assign a default value when not enabled

end

end

endmodule