Md. Maniruzzaman

ID: 155009

**EE461 – Mid Term Exam**

***Q1: Answer***

`timescale 1ns/1ps

module q1 (

output reg a,

input b, c

);

always @(\*) begin

a = (b) ? b : c;

end

endmodule

|  |  |
| --- | --- |
| **Problem was** | **Explanation** |
| module module; | Illegal name: keyword ‘module’ cannot be used as identifier. |
| time a; | time is a 64-bit simulation variable, not mean for logic signals. |
| assign begin ... end | Invalid - assign is for continuous assignments only, not procedural blocks. |
| Missing ports | No input/output defined - not synthesizable. |
| Missing sensitivity list | always @(\*) required for combinational logic. |

***Q2: Answer***

From the given statement: `timescale 100ps / 100ps

* 1 time unit = 100 ps

In module given nor #3.1415 (z, x1, x2);

* The delay is 3.1415 × time unit

Therefore,

* Delay = 3.1415 x 100 ps

= 314.15 ps

**≈** 314 ps

**Now,** Rounded to the nearest hundred picoseconds

* Delay **=** 300 ps (approx.)

***Q3: Answer***

module equality\_check;

reg [3:0] a, b;

initial begin

a = 4'b01xz;

b = 4'bxz10;

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

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

end

endmodule

**Output:**

A screenshot of a computer

AI-generated content may be incorrect.

***Q4: Answer***

module not\_operator\_check;

reg [3:0] a;

initial begin

a = 4'bxz01;

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

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

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

end

endmodule

**Output:**

A screenshot of a computer

AI-generated content may be incorrect.

***Q5: Answer***

module 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", a);

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

#1 $display("c = %b", c);

#10 $finish;

end

endmodule

**Output:**

A screenshot of a computer

AI-generated content may be incorrect.

**Here,**

Input value

* a = 4'b0010 = 2 (decimal)

Compute b = a << 2

* The operator << means left shift by 2 bits.

So, we shift a (0010) left by two positions:

* 0010 << 2 => 1000

That is equivalent to multiplying by 4:

* b = 2 × 4 = 8 => 4'b1000
* b = 4’b1000

Compute c = 2 << a

* Now we left shift the constant 2 (0010) by the value of a (which is 2 bits).

So:

* 0010 << 2 => 1000
* That’s also decimal 8.
* c = 4’b1000

Therefore,

Given: a = 4’b0010

Computation:

* b = a << 2 => 0010 << 2 = 1000 => b = 4’b1000 (8)
* c = 2 << a => 0010 << 2 = 1000 => c = 4’b1000 (8)

Explanation:

* The left-shift operator (<<) moves bits to the left and fills zeros on the right.
* Shifting left by 1 bit multiplies the number by 2; shifting left by 2 bits multiplies by 4.
* Therefore, both expressions produce the binary result 1000.

***Q6: Answer***

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

output going;

input std1;

input std2;

input std3;

// Here is your code

// Answer part:

table // std1 std2 std3 : going

     0 0 0 : 0;  // None agree => no minority => not go

     0 0 1 : 1;  // One agrees => minority => go

     0 1 0 : 1;  // One agrees => minority => go

     1 0 0 : 1;  // One agrees => minority => go

     0 1 1 : 0;  // Two agree => majority => not go

     1 0 1 : 0;  // Two agree => majority => not go

     1 1 0 : 0;  // Two agree => majority => not go

     1 1 1 : 0;  // All agree => majority => not go

  endtable

endprimitive

***Q7: Answer***

module test;

reg a;

reg [1:0] b;

reg [2:0] c;

reg [3:0] d;

always @(\*) begin

c = a + b;

d = b + c;

a = d[0];

end

endmodule

|  |  |
| --- | --- |
| **Problem Was** | **Explanation** |
| Mixing posedge and level-sensitive signals in one always block | You cannot mix posedge (edge-triggered) and level-sensitive signals like a or b in the same sensitivity list. It causes simulation ambiguity. |
| wire variables assigned inside always block | Variables assigned inside always must be **reg**, not wire. |
| Illegal assign inside always blocks | assign is only allowed for continuous assignments, not inside procedural blocks. |
| Width mismatch | Expression a + d mixes 1-bit with 4-bit signals - inconsistent data width. |
| No proper timing control | The code looks like combinational logic but mixes sequential behavior (posedge). |

***Q8: Answer***

|  |  |
| --- | --- |
| **testbench** | **design** |
| // Q8  `timescale 1ns/1ps  module tb\_MultiplyAdder;  reg [31:0] x\_r;  reg [31:0] y\_r;  wire [31:0] out\_w;  MultiplyAdder uut (  .x(x\_r),  .y(y\_r),  .out(out\_w)  );  initial begin  $dumpfile("MultiplyAdder.vcd");  $dumpvars(0, tb\_MultiplyAdder);  $display("Time | x | y | out (14x + 16y)");  $display("------------------------------------------");  x\_r = 3; y\_r = 9; #5;  $display("%4t | %3d | %3d | %3d", $time, x\_r, y\_r, out\_w);  x\_r = 5; y\_r = 10; #5;  $display("%4t | %3d | %3d | %3d", $time, x\_r, y\_r, out\_w);  x\_r = 1; y\_r = 1; #5;  $display("%4t | %3d | %3d | %3d", $time, x\_r, y\_r, out\_w);  #10 $finish;  end  endmodule | // Q8  `timescale 1ns/1ps  module MultiplyAdder (  input wire [31:0] x,  input wire [31:0] y,  output wire [31:0] out  );  wire [31:0] term1, term2;  assign term1 = (x << 3) + (x << 2) + (x << 1);  assign term2 = (y << 4);  assign out = term1 + term2;  endmodule |

**Output:**

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

***Q9: Answer***

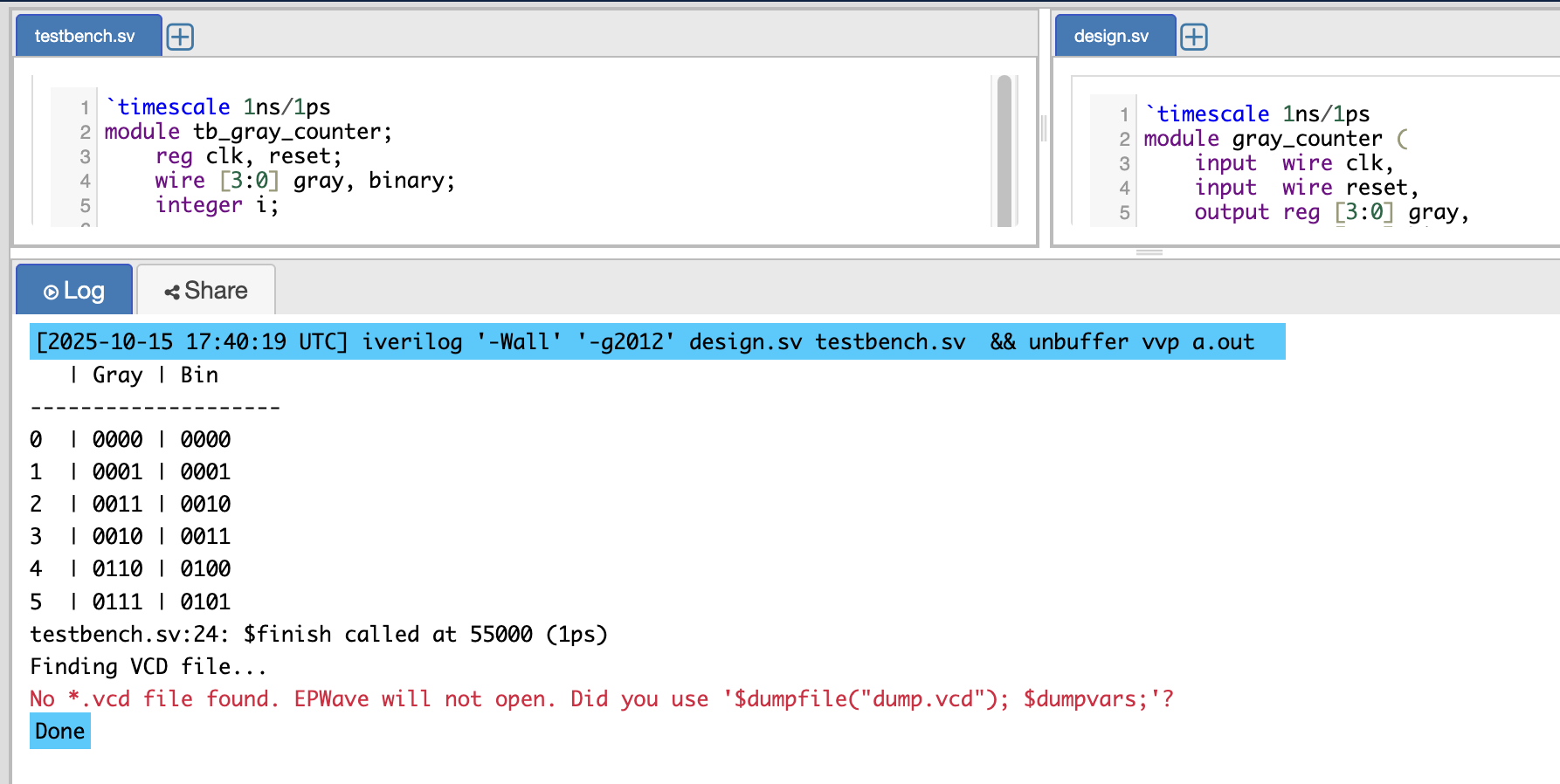
The difference between continuous assignment and always block in the given code:

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Continuous Assignment (assign)** | **Always Block (always @(\*))** |
| Signal Type | Used with wire | Used with reg or logic |
| Execution Style | Evaluated continuously whenever RHS changes | Executes procedurally when signals in the sensitivity list change |
| Coding Style | Dataflow / RTL modeling | Behavioral / procedural modeling |
| Hardware Representation | Synthesizes to combinational logic (such as 2-to-1 MUX) | Also synthesizes to combinational logic (such as 2-to-1 MUX) |
| Example Syntax | assign a = (b) ? c : d; | always @(\*) if (b) a = c; else a = d; |
| When b = 1 | a = c | a = c |
| When b = 0 | a = d | a = d |
| When b = x (unknown) | a = x  => because the ternary operator uses 4-state logic and cannot resolve x | a = d  => because if(b) treats x as false, so else executes |
| Key Behavioral Difference | Produces unknown (‘x’) when control signal is undefined | Produces deterministic output (assigns to d) when control is undefined |
| Conclusion | Both implement same hardware, but differ during simulation when control (b) is unknown | Behavior differs only for undefined (‘x’) conditions |

***Q10: Answer***

|  |  |
| --- | --- |
| **testbench** | **design** |
| `timescale 1ns/1ps  module tb\_gray\_counter;  reg clk, reset;  wire [3:0] gray, binary;  integer i;  gray\_counter uut (.clk(clk), .reset(reset), .gray(gray), .binary(binary));  always #5 clk = ~clk;  initial begin  clk = 0;  reset = 1;  #5 reset = 0;  $display(" | Gray | Bin");  $display("--------------------");  $display("0 | %b | %b", gray, binary);  for (i = 1; i < 6; i = i + 1) begin  #10 $display("%0d | %b | %b", i, gray, binary);  end  $finish;  end  endmodule | `timescale 1ns/1ps  module gray\_counter (  input wire clk,  input wire reset,  output reg [3:0] gray,  output reg [3:0] binary  );  always @(posedge clk or posedge reset) begin  if (reset)  binary <= 4'b0000;  else if (binary < 4'd5)  binary <= binary + 1;  else  binary <= 4'b0000;  end  always @(\*) begin  gray = binary ^ (binary >> 1);  end  endmodule |

**Output:**



***Q11: Answer***

module test (

input wire a, // input signal a

input wire b, // input signal b

input wire en, // enable signal

output reg c // output signal

);

// Combinational always block

always @(\*) begin

if (en)

c = a | b; // when enabled, perform OR operation

else

c = 1'b0; // otherwise output 0

end

endmodule

|  |  |
| --- | --- |
| **Problem was** | **Explanation** |
| No port list / not parameterized | The module has no input or output ports — all variables are declared internally, which makes it non-synthesizable and unusable for testing. |
| reg used incorrectly for inputs | a, b, and en are used like input signals but declared as reg, which is illegal in RTL design unless explicitly assigned in an always block inside the same module. |
| Incomplete sensitivity list (@(a or en)) | The signal b affects the output c, but it’s missing from the sensitivity list. This leads to mismatched simulation and synthesis behavior. |
| No default or else assignment | If en is 0, c is not assigned any value - causing an unintended latch (storage element), which violates combinational logic design rules. |
| Not parameterized as combinational module | The code implies combinational logic but doesn’t use always @(\*), the correct syntax for automatic sensitivity. |
| Non-synthesizable structure | Because of missing ports and improper declarations, the design cannot be connected to a testbench or synthesized properly. |