

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 $3.1415 \times \text{time unit}$

Therefore,

- Delay = $3.1415 \times 100 \text{ ps}$
 $= 314.15 \text{ ps}$
 $\approx 314 \text{ ps}$

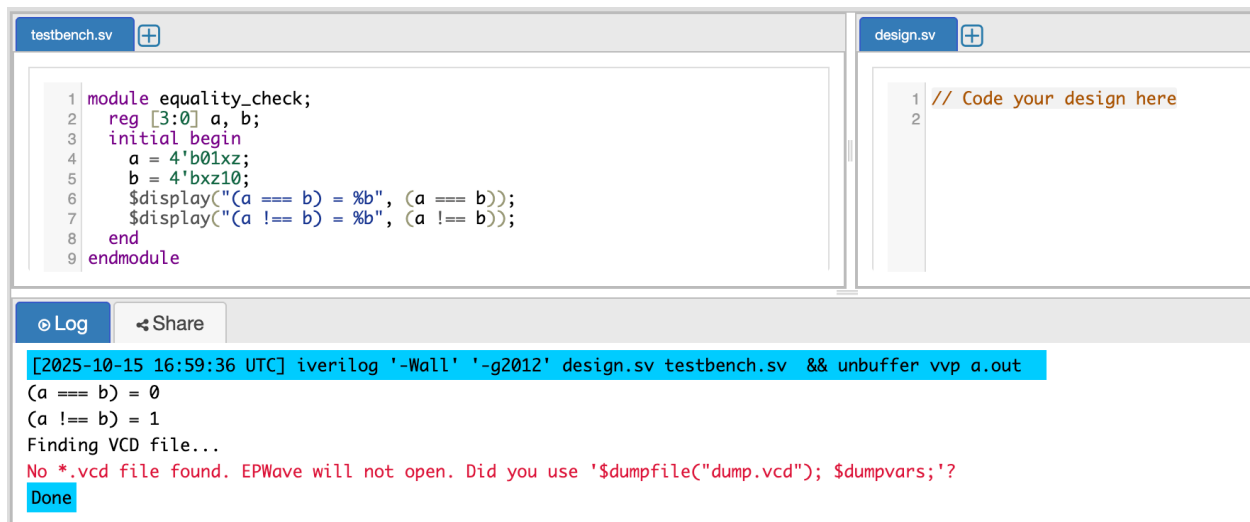
Now, Rounded to the nearest hundred picoseconds

- 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:



The screenshot shows a Verilog IDE with two files: `testbench.sv` and `design.sv`. The `testbench.sv` file contains the `equality_check` module definition. The `design.sv` file is empty, containing only a comment. Below the code editors, there is a Log window showing the command `iverilog '-Wall' '-g2012' design.sv testbench.sv && unbuffer vvp a.out` and its output. The output shows the results of the equality and inequality checks, followed by a message about the VCD file.

```
testbench.sv
1 module equality_check;
2   reg [3:0] a, b;
3   initial begin
4     a = 4'b01xz;
5     b = 4'bxz10;
6     $display("(a === b) = %b", (a === b));
7     $display("(a !== b) = %b", (a !== b));
8   end
9 endmodule

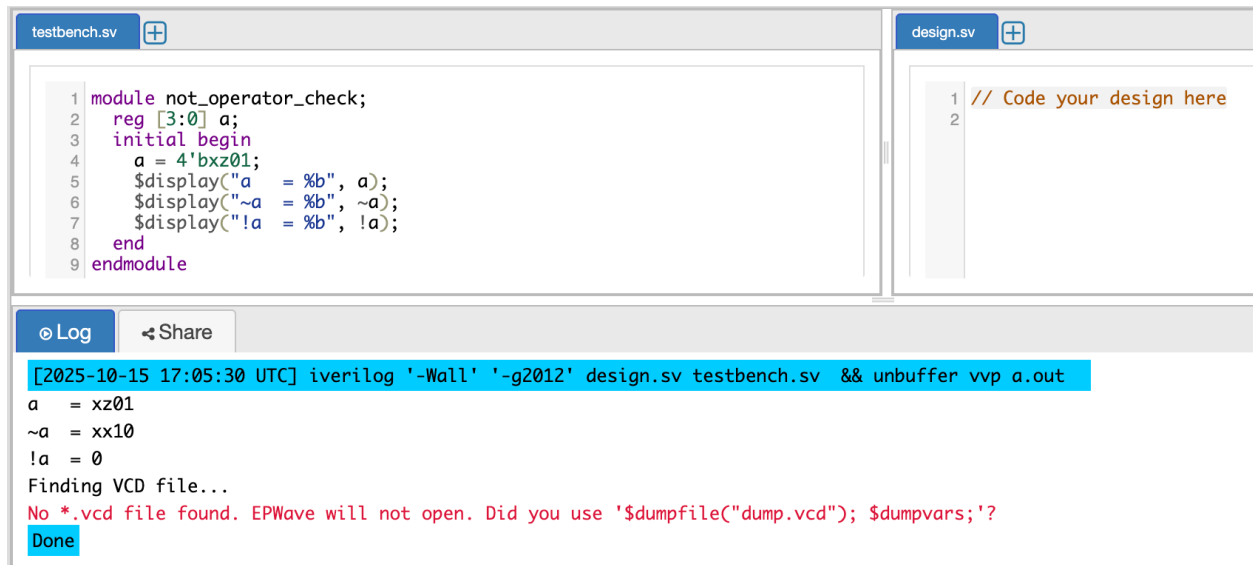
design.sv
1 // Code your design here
2

Log
[2025-10-15 16:59:36 UTC] iverilog '-Wall' '-g2012' design.sv testbench.sv && unbuffer vvp a.out
(a === b) = 0
(a !== b) = 1
Finding VCD file...
No *.vcd file found. EPWave will not open. Did you use '$dumpfile("dump.vcd"); $dumpvars;'?
Done
```

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:



The screenshot shows the IVerilog web IDE interface. On the left, the 'testbench.sv' file is open, containing a Verilog testbench for a 4-bit register. The testbench initializes 'a' to 4'bxx01 and displays its value, its complement, and its logical NOT. On the right, the 'design.sv' file is open, showing a placeholder comment '// Code your design here'. Below the code editors, there is a 'Log' tab showing the execution output. The output includes the command used to run the simulation, the initial value of 'a' (xx01), its complement (xx10), and the result of the logical NOT operation (0). It also shows a message about finding a VCD file and a warning about EPWave not opening due to a missing VCD file. The log ends with a 'Done' status.

```
1 module not_operator_check;
2   reg [3:0] a;
3   initial begin
4     a = 4'bxx01;
5     $display("a = %b", a);
6     $display("~a = %b", ~a);
7     $display("!a = %b", !a);
8   end
9 endmodule
```

```
1 // Code your design here
2
```

Log

[2025-10-15 17:05:30 UTC] iverilog '-Wall' '-g2012' design.sv testbench.sv && unbuffer vvp a.out

a = xx01

~a = xx10

!a = 0

Finding VCD file...

No *.vcd file found. EPWave will not open. Did you use '\$dumpfile("dump.vcd"); \$dumpvars;'?

Done

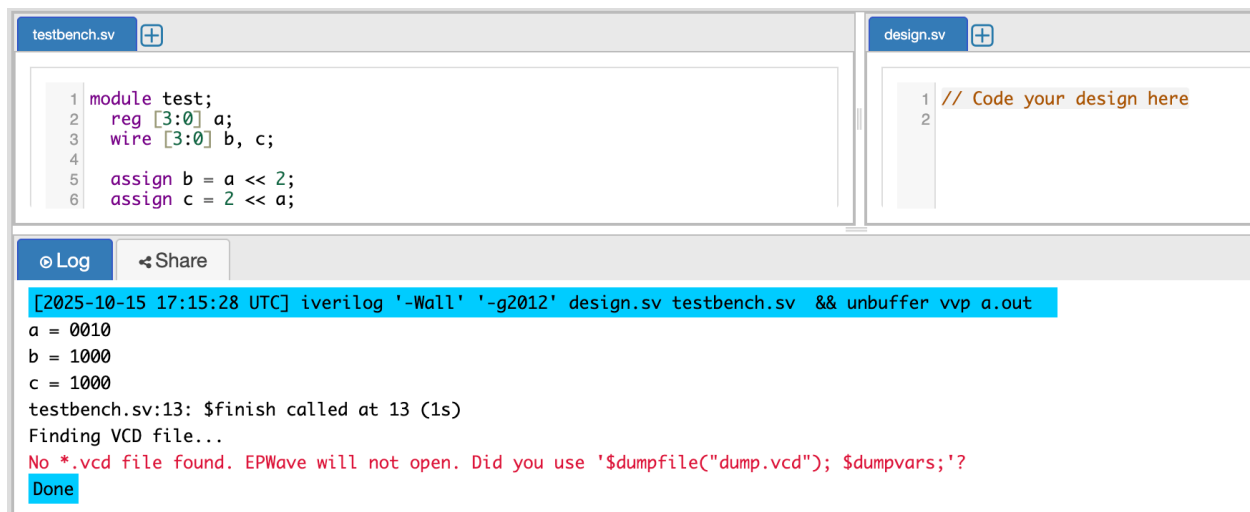
Q5: Answer

```
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:



The screenshot shows a Verilog simulation environment with two tabs: 'testbench.sv' and 'design.sv'. The 'testbench.sv' tab contains the following code:

```
1 module test;
2   reg [3:0] a;
3   wire [3:0] b, c;
4
5   assign b = a << 2;
6   assign c = 2 << a;
```

The 'design.sv' tab contains the following code:

```
1 // Code your design here
2
```

Below the code tabs is a 'Log' button and a 'Share' button. The 'Log' button is selected, and the terminal window displays the following output:

```
[2025-10-15 17:15:28 UTC] iverilog '-Wall' '-g2012' design.sv testbench.sv && unbuffer vvp a.out
a = 0010
b = 1000
c = 1000
testbench.sv:13: $finish called at 13 (1s)
Finding VCD file...
No *.vcd file found. EPWave will not open. Did you use '$dumpfile("dump.vcd"); $dumpvars;'?
Done
```

Here,

Input value

- $a = 4'b0010 = 2$ (decimal)

Compute $b = a \ll 2$

- The operator \ll means left shift by 2 bits.

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

- $0010 \ll 2 \Rightarrow 1000$

That is equivalent to multiplying by 4:

- $b = 2 \times 4 = 8 \Rightarrow 4'b1000$
- $b = 4'b1000$

Compute $c = 2 \ll a$

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

So:

- $0010 \ll 2 \Rightarrow 1000$
- That's also decimal 8.
- $c = 4'b1000$

Therefore,

Given: $a = 4'b0010$

Computation:

- $b = a \ll 2 \Rightarrow 0010 \ll 2 = 1000 \Rightarrow b = 4'b1000$ (8)
- $c = 2 \ll a \Rightarrow 0010 \ll 2 = 1000 \Rightarrow c = 4'b1000$ (8)

Explanation:

- The left-shift operator (\ll) 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
<pre>// 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</pre>	<pre>// 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</pre>

Output:

testbench.sv

```

1 // Q8
2 `timescale 1ns/1ps
3 module tb_MultiplyAdder;
```

design.sv

```

1 // Q8
2 `timescale 1ns/1ps
3
```

Log
Share

[2025-10-15 17:32:15 UTC] iverilog '-Wall' '-g2012' design.sv testbench.sv && unbuffer vvp a.out

VCD info: dumpfile MultiplyAdder.vcd opened for output.

Time	x	y	out (14x + 16y)

5000	3	9	186
10000	5	10	230
15000	1	1	30

testbench.sv:30: \$finish called at 25000 (1ps)
Finding VCD file...
./MultiplyAdder.vcd

[2025-10-15 17:32:15 UTC] Opening EPWave...

Done

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
<pre>`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</pre>	<pre>`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</pre>

Output:

The screenshot shows the Icarus Verilog IDE with two source files: `testbench.v` and `design.v`. Below the code editors is a log window showing the simulation output.

testbench.v

```
1 `timescale 1ns/1ps
2 module tb_gray_counter;
3     reg clk, reset;
4     wire [3:0] gray, binary;
5     integer i;
```

design.v

```
1 `timescale 1ns/1ps
2 module gray_counter (
3     input wire clk,
4     input wire reset,
5     output reg [3:0] gray,
```

Log Output:

```
[2025-10-15 17:40:19 UTC] iverilog '-Wall' '-g2012' design.v testbench.v && unbuffer vvp a.out
 | Gray | Bin
-----
0 | 0000 | 0000
1 | 0001 | 0001
2 | 0011 | 0010
3 | 0010 | 0011
4 | 0110 | 0100
5 | 0111 | 0101
testbench.v:24: $finish called at 55000 (1ps)
Finding VCD file...
No *.vcd file found. EPWave will not open. Did you use '$dumpfile("dump.vcd"); $dumpvars;'?
Done
```


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.