Nang Thiri Wutyi
20113
10/22/2024

Verilog Midterm Exam

1. **Bugs**
    - The module name “module” is a reserved keyword and not valid. It should be a proper name.
    - The assign block is incorrectly structured. The keyword assign is used for continuous assignments, and begin and end are not allowed inside an assign block.
    - The time keyword is used for simulation time representation and should not be used for regular logic or variables. It should be a wire or reg if the intent is to use a as a signal.
    - The variables b and c are declared as reg, but should be driven in a proper way. The conditional expression (b) ? b : c seems like it wants to assign to a based on logic, which implies a should be wire, not time.
    - The assign statement requires a valid expression but doesn't need begin-end blocks.

    Fixed code

    ```
    module my_module;
       reg b, c;          // Declare b and c as reg (for inputs, testbench control, etc.)
       wire a;            // Declare a as wire, because it is assigned in a continuous
    assignment

       // Corrected continuous assignment statement
       assign a = (b) ? b : c;  // Ternary operator to assign a based on the value of b
    and c

    endmodule
    ```

2. According to the code, the propagation delay of NOR gate is 3.1415 units,
   Time unit = 100ps
   Actual Delay = delay of NOR x time unit = 3.1415 x 100 = 314.15 ps

3. “===” is a case equality operator which compares both values bit by bit, and returns a value 1 if all bits are identical.
   In this case, a = 4’b01xz
   b = 4’bxz10
   Since the bits are not identical, **a===b is 0** (false)

“!==” is a case inequality operator which checks if there is any difference between the two values. Since the values of a and b are not the same, **a !== b** is true and it will return **1**.

4. a = 4b’xz01
   ~a (bitwise NOT a)
   x → x
   z → z
   1 → 0
   0 → 1
   ~a = 4b’xz10

   !a (logical NOT a)
   If all bits are 0, the result is 1.
   If any bit is 1, x, or z, the result is 0 or x (for unknown cases).
   If any part is non-zero, it's considered false.
   Therefore !a = 0

5. b = a << 2
   a = 4’b0010
   The value of a is shifted left by two bits
   b = 4’b1000

   c = 2<<a
   2 << a shifts the value 2 left by the number of positions specified by the value of a
   a = 4'b0010 (2 in decimal)
   2=0010 (binary for decimal 2)
   0010<<2=1000
   c = 4'b1000

6.

```
primitive agree(going, std1, std2, std3);
  output going;
  input std1, std2, std3;

  // Truth table for the minority rule
  table
    // std1  std2  std3 : going
        0     0     0   : 0;  // No one agrees, don't go
```

```
    0    0    1   : 1;  // Minority (1 person agrees), go
    0    1    0   : 1;  // Minority (1 person agrees), go
    1    0    0   : 1;  // Minority (1 person agrees), go
    0    1    1   : 1;  // Majority agrees, go
    1    0    1   : 1;  // Majority agrees, go
    1    1    0   : 1;  // Majority agrees, go
    1    1    1   : 1;  // All agree, go
  endtable
endprimitive
```

7. **Bugs**
    - Assigning Values in an always Block: The assign statement is used for continuous assignments, which should not be mixed with procedural assignments (<= or =) inside an always block.
    - Sensitivity List: The sensitivity list of an always block is incorrectly specified. If you're using posedge c, it should typically be in a clocked process, which should not also include other signals like a or b.
    - Data Types for Outputs: Ensure that all variables that are assigned new values inside an always block are declared as reg type.
    - Undefined Behavior: The way signals are assigned can lead to undefined behavior due to the simultaneous assignment of a.

Fixed RTL module

```
module test;
  reg a;              // Change wire to reg for procedural assignment
  reg [1:0] b;        // Change wire to reg for procedural assignment
  reg [2:0] c;        // Change wire to reg for procedural assignment
  reg [3:0] d;        // Change wire to reg for procedural assignment

  // Clock signal for simulation purposes
  reg clk;

  // Clock generation (for simulation purposes)
  initial begin
    clk = 0;
    forever #5 clk = ~clk; // Toggle clock every 5 time units
  end

  always @(posedge clk) begin
    c <= a + b;       // Assign values using non-blocking assignment
```

```
  d <= b + c;      // Assign values using non-blocking assignment
  a <= a + d;      // Correctly use a non-blocking assignment
 end

endmodule
```

8. **RTL module**

```
module compute_out(out, x, y);
 input [31:0] x, y;  // 32-bit inputs
 output [31:0] out;  // 32-bit output
 wire [31:0] term1, term2, term3, term4;

 // Compute 14x using shifts and adders
 assign term1 = x << 3;  // 8x
 assign term2 = x << 2;  // 4x
 assign term3 = x << 1;  // 2x
 wire [31:0] sum_14x;
 assign sum_14x = term1 + term2 + term3;  // 14x = 8x + 4x + 2x

 // Compute 16y using a single shift
 assign term4 = y << 4;  // 16y

 // Final result: out = 14x + 16y
 assign out = sum_14x + term4;

endmodule
```

Testbench

```
module testbench;
 reg [31:0] x, y;  // 32-bit inputs
 wire [31:0] out;  // 32-bit output

 // Instantiate the compute_out module
 compute_out uut (
  .out(out),
  .x(x),
  .y(y)
 );

 initial begin
```

```
    // Display output headers
    $display("x\t\t y\t\t out\t\t\t (14x + 16y)");

    // Test case 1
    x = 32'd1;  // x = 1
    y = 32'd1;  // y = 1
    #10;
    $display("%d\t %d\t %d", x, y, out); // Expected out = 14*1 + 16*1 = 30

    // Test case 2
    x = 32'd2;  // x = 2
    y = 32'd2;  // y = 2
    #10;
    $display("%d\t %d\t %d", x, y, out); // Expected out = 14*2 + 16*2 = 60

    // Test case 3
    x = 32'd0;  // x = 0
    y = 32'd5;  // y = 5
    #10;
    $display("%d\t %d\t %d", x, y, out); // Expected out = 14*0 + 16*5 = 80

    // Test case 4
    x = 32'd3;  // x = 3
    y = 32'd4;  // y = 4
    #10;
    $display("%d\t %d\t %d", x, y, out); // Expected out = 14*3 + 16*4 = 118

    // Test case 5
    x = 32'd10; // x = 10
    y = 32'd1;  // y = 1
    #10;
    $display("%d\t %d\t %d", x, y, out); // Expected out = 14*10 + 16*1 = 156

    // End the simulation
    $finish;
  end
endmodule
```

Results

```
[2024-10-22 17:07:15 UTC] iverilog '-Wall' '-g2012' design.sv testbench.sv  &&
unbuffer vvp a.out
```

```
x             y            out                 (14x + 16y)
        1             1            30
        2             2            60
        0             5            80
        3             4           106
       10             1           156
testbench.sv:47: $finish called at 50 (1s)
Done
```

9. **Continuous Assignment**:
    - Wire Type: In the first code snippet, a is declared as a wire. Continuous assignments can only drive wire types. This means that a is always reflecting the value of c or d based on the value of b. If b changes, a will immediately update to either c or d without any delay. This is suitable for simple combinational logic where a must always mirror the result of the conditional expression.
    - Immediate Evaluation: The assignment (b) ? c : d is evaluated continuously. If b changes from 0 to 1 or vice versa, the value of a changes immediately to match the new condition, without needing to wait for a clock edge or any other triggering event.

    Always Block:

    - Reg Type: In the second code snippet, a is declared as a reg, allowing it to be assigned values in an always block. The value of a will be updated based on the condition of b during the execution of the always block, which is triggered whenever any input within the sensitivity list changes. In this case, the * means it responds to changes in b, c, or d.
    - Conditional Execution: The if statement inside the always block allows for more complex logic. It explicitly defines how a gets its value based on the state of b. If b is true, a takes the value of c; otherwise, it takes the value of d. The execution occurs at the time of any change in b, c, or d, making it more flexible than the continuous assignment.

    Key differences

Continuous Assignment

- Type: Uses wire type for output (a).
- Syntax: Utilizes the assign statement.
- Immediate Update: Reflects changes immediately when b changes.
- Logic Simplicity: Best for straightforward conditional logic (ternary operator).
- Sensitivity: Continuously evaluates the condition without needing an event.

Always Block

- Type: Uses reg type for output (a).
- Syntax: Encapsulated within an always block.
- Triggered Update: Updates when any input (like b, c, or d) changes.
- Logic Flexibility: Allows for complex conditional statements (e.g., if-else).
- Sensitivity: Executes on changes to specified inputs in the sensitivity list.

10.

	Binary	**Gray**
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0100

Design gray counter module

```
`timescale 1ns / 1ps


module gray_counter (
    input wire clk,
    input wire reset,
    output reg [3:0] gray
);


    always @(posedge clk or posedge reset) begin
```

```
        if (reset) begin
            gray <= 4'b0000;  // Reset to 0 in Gray code
        end else begin
            case (gray)
                4'b0000: gray <= 4'b0001; // 0 -> 1
                4'b0001: gray <= 4'b0011; // 1 -> 2
                4'b0011: gray <= 4'b0010; // 2 -> 3
                4'b0010: gray <= 4'b0110; // 3 -> 4
                4'b0110: gray <= 4'b0111; // 4 -> 5
                4'b0111: gray <= 4'b0100; // 5 -> 6
                4'b0100: gray <= 4'b0000; // 6 -> 0 (wrap around)
                default: gray <= 4'b0000; // Default case (safety)
            endcase
        end
    end
endmodule
```

Testbench module

```
`timescale 10ns / 1ns

module testbench;
    reg clk;
    reg reset;
    wire [3:0] gray;
```

```
// Instantiate the gray_counter
gray_counter uut (
    .clk(clk),
    .reset(reset),
    .gray(gray)
);

// Clock generation
initial begin
    clk = 0;
    forever #5 clk = ~clk;  // Toggle clock every 5 ns
end

// Test sequence
initial begin
    reset = 1;              // Assert reset
    #10 reset = 0;          // Deassert reset
    #100;                   // Wait for a while to observe the counter

    reset = 1;              // Assert reset again
    #10 reset = 0;          // Deassert reset
    #100;                   // Wait for a while to observe the counter
```

```
    $finish;                // End simulation
  end

  // Monitor output
  initial begin
    $monitor("Time: %0t | Reset: %b | Gray Code: %b", $time, reset, gray);
  end
endmodule
```

Results

```
[2024-10-22 17:33:38 UTC] iverilog '-Wall' '-g2012' design.sv testbench.sv  && unbuffer vvp a.out

Time: 0 | Reset: 1 | Gray Code: 0000

Time: 100000 | Reset: 0 | Gray Code: 0000

Time: 150000 | Reset: 0 | Gray Code: 0001

Time: 250000 | Reset: 0 | Gray Code: 0011

Time: 350000 | Reset: 0 | Gray Code: 0010

Time: 450000 | Reset: 0 | Gray Code: 0110

Time: 550000 | Reset: 0 | Gray Code: 0111

Time: 650000 | Reset: 0 | Gray Code: 0100

Time: 750000 | Reset: 0 | Gray Code: 0000

Time: 850000 | Reset: 0 | Gray Code: 0001

Time: 950000 | Reset: 0 | Gray Code: 0011

Time: 1050000 | Reset: 0 | Gray Code: 0010

Time: 1100000 | Reset: 1 | Gray Code: 0000

Time: 1200000 | Reset: 0 | Gray Code: 0000
```

```
Time: 1250000 | Reset: 0 | Gray Code: 0001

Time: 1350000 | Reset: 0 | Gray Code: 0011

Time: 1450000 | Reset: 0 | Gray Code: 0010

Time: 1550000 | Reset: 0 | Gray Code: 0110

Time: 1650000 | Reset: 0 | Gray Code: 0111

Time: 1750000 | Reset: 0 | Gray Code: 0100

Time: 1850000 | Reset: 0 | Gray Code: 0000

Time: 1950000 | Reset: 0 | Gray Code: 0001

Time: 2050000 | Reset: 0 | Gray Code: 0011

Time: 2150000 | Reset: 0 | Gray Code: 0010

testbench.sv:31: $finish called at 2200000 (1ps)

Done
```

11. Corrected module

```
module test;

  reg a, b, c, en;



  always @(a or b or en) begin

    if (en)

      c = a | b; // Logical OR operation when en is high

    else

      c = 0; // Clear the output when en is low (optional, depends on intended behavior)

  end

endmodule
```

Changes

- Changed the sensitivity list from @(a or en) to @(a or b or en) to ensure that the always block is triggered whenever a, b, or en changes.
- If only a and en are monitored, changes to b would not update c, potentially leading to incorrect behavior.
- Added an else clause to set c to 0 when en is low. When en is high, c takes the logical OR of a and b.
- When en is low, c is set to 0 (or retains its previous value if that's the desired behavior).