INTERVIEW QUESTIONS | VERILOG

PART 4

JAIRAJ MIRASHI
DESIGN VERIFICATION ENGINEER

1. **What is the difference between $finish and $stop?**

$finish is a system task that simply makes the simulator exit and passes control back to the host operating system.

$stop is a system task that causes the simulation to be suspended.

2. **What hardware structure is inferred by both case and if statements in Verilog?**

Both case and if statements are typically synthesized into hardware multiplexers or combinational logic circuits in Verilog.

3. **How could you change a case statement in order that its implementation does not result in a priority structure?**

You can change a case statement into a "don't care" priority structure by using the default clause without specifying any other condition inside the case. This will prevent the synthesis tool from inferring a priority structure.

Example:

```
module PriorityExample(input [1:0] sel, output reg [3:0] out);
    always @(*)
    begin
        case (sel)
            2'b00: out = 4'b0001; // Condition 1
            2'b01: out = 4'b0010; // Condition 2
            2'b10: out = 4'b0100; // Condition 3
            default: out = 4'b1000; // Default or "don't care" condition
        endcase
    end
endmodule
```

4. **If you are not using a synthesis attribute "full case," how can you assure coverage of all conditions for a case statement?**

You can ensure coverage of all conditions for a case statement by adding a default clause (default) at the end of the case block. This way, any condition not explicitly covered by other cases will be handled by the default clause.

5. How do you infer tristate gates for synthesis?

Tristate gates can be inferred in Verilog by using the tri data type and assigning values using the assign statement with conditional logic to enable or disable the tristate condition based on specific control signals.

6. Can a task be synthesized?

No, tasks are not synthesized. Tasks are used for specifying simulation-level behavior and are not part of the hardware description that gets synthesized into hardware circuits.

7. What is the difference between ! and ~?

! is the logical NOT operator, which inverts the value of a Boolean expression. For example, !A will be true if A is false, and vice versa.

~ is the bitwise NOT operator, which inverts each bit of a binary number.

8. What is the difference between the two statements? Do a and b values have to be equal?

a = edata[0] || edata[1];

b = |edata;

a and b are not necessarily equal. a performs a logical OR operation between edata[0] and edata[1], while b performs a bitwise OR operation on all bits of edata.

9. What is the difference between the two programs?

a)

```
initial
  #10 a = 0;

always @(a)
  a <= ~a;
```

b)

```
initial
  #10 a = 0;

always @(a)
  a = ~a;
```

The key difference between the two programs is the use of blocking (=) and non-blocking (<=) assignments:

In the first program, a is assigned the value of ~a using non-blocking assignment (<=). This means that a is updated at the end of the current time step, and the assignment does not affect the order of execution within the always block.

In the second program, a is assigned the value of ~a using blocking assignment (=). This means that the assignment occurs immediately, potentially affecting the order of execution within the always block.

10. Why is it recommended not to mix blocking and non-blocking assignments in the same block?

Mixing blocking and non-blocking assignments in the same block can lead to simulation errors and unexpected behavior. It can create race conditions and make the code difficult to understand and maintain. Keeping them separate helps ensure predictable and reliable behavior in your Verilog code.

11. Declare parameters for representing state machine states using one-hot encoding.

```
module OneHotStateMachine (
  input wire [2:0] state,
  output wire stateA,
  output wire stateB,
  output wire stateC
);

  // Define parameters for one-hot encoded states
  parameter STATE_A = 3'b001; // State A
  parameter STATE_B = 3'b010; // State B
  parameter STATE_C = 3'b100; // State C

  // Output assignments based on the current state
  assign stateA = (state == STATE_A);
  assign stateB = (state == STATE_B);
  assign stateC = (state == STATE_C);

endmodule
```

12. What does a function synthesize into?

A function in Verilog typically synthesizes into combinational logic. It is used to describe a Boolean function, and the synthesis tool will map it to gates and wires that implement the desired logic.

13. How to change the value of width to 3 in the following code?

`define width 7

To change the value of width to 3 in the Verilog code using a define macro, you can do the following:

```
`define width 3
```

This will define width as 3, and you can use it in your code as width wherever needed.

14. What is the functionality of $input?

The $input system task allows command input text to come from a named file instead of from the terminal. At the end of the command file, the input is switched back to the terminal.

15. What is the difference between blocking and non-blocking?

Blocking Assignments	Non-Blocking Assignments
Sequential Execution: Blocking assignments (=) are executed sequentially in the order they appear within a procedural block.	Parallel Execution: Non-blocking assignments (<=) are executed in parallel within the same simulation time step, regardless of their order in the code.
Immediate Update: A blocking assignment causes the value of the variable or signal on the left-hand side to be updated immediately within the current simulation time step.	Deferred Update: A non-blocking assignment schedules the update of the variable or signal on the left-hand side to occur at the end of the current simulation time step.
Typically used for describing combinational logic or concurrent assignments where order of execution matters.	Typically used for modeling clocked sequential logic, like flip-flops and state machines, to ensure proper simulation behavior and avoid race conditions.

Race Conditions: Can potentially lead to race conditions if multiple assignments to the same variable occur within the same block.	Preventing Race Conditions: Non-blocking assignments inherently prevent race conditions within the same block because all assignments are scheduled to happen simultaneously at the end of the time step.
Example: `a = b; // Value of 'a' is immediately updated to 'b'`	Example: `a <= b; // Value of 'a' is scheduled to be updated to 'b' at the end of the time step`

16. What are the two types of race conditions?

Data Race Condition: It occurs when two or more processes or threads access shared data concurrently, and at least one of them modifies the data. Data races can lead to unpredictable behavior and incorrect results.

Control Race Condition: It occurs when the control flow or sequence of execution in a program or system is not deterministic due to multiple possible paths. Control race conditions can lead to unexpected outcomes and are often associated with asynchronous events.

17. How can race conditions between the DUT and the testbench be avoided?

To avoid race conditions between the DUT (Design Under Test) and the testbench, you can follow these practices:

- Synchronize clock domains properly if multiple clocks are involved.
- Use non-blocking assignments (<=) for signal updates in testbench processes.
- Ensure that the testbench generates stimuli and checks results in a controlled and synchronized manner.
- Use proper synchronization constructs like wait statements or events to coordinate testbench and DUT activities.
- Avoid creating race conditions by ensuring that signals are sampled or modified at the appropriate times based on clock edges.

18. How can race conditions between the DUT and the testbench be avoided?

Guidelines to avoid race conditions:

- Use non-blocking (<=) assignments for signal updates in synchronous processes.
- Properly synchronize signals when crossing clock domains.
- Use synchronization constructs like wait statements to coordinate activities in testbenches.
- Be aware of signal setup and hold times in synchronous designs.
- Avoid multiple drivers for the same signal.
- Ensure that signals are sampled and modified at the right clock edges.

19. Identify the bug in the following code:

always @(posedge clk)

a = b;

always @(posedge clk)

b = a;

The bug in the code you provided is that it results in multiple drivers for signals a and b, which can lead to a race condition. To fix it, you should use a single always block for both assignments:

```
always @(posedge clk) begin
  a <= b;
  b <= a;
end
```

This code will ensure that both a and b are updated synchronously on the rising edge of the clock, preventing any race condition.

20. Identify the bug in the following code:

if (a = b)

match = 1;

else

match = 0;

The bug in the code is that you are using a single equals sign (=) for comparison instead of a double equals sign (==).

Here's the corrected code:

```
if (a == b)
  match = 1;
else
  match = 0;
```

In Verilog, == is used for equality comparison, while = is used for signal assignment.

21. Find the bug in the following code.

```
for (.....);
begin
.....
end
```

Misplaced semicolons in for-loops.

22. Find the bug in the following code.

```
automatic itasko entry_assign();
begin
a <= b; // Comment
end
```

Intra-assignment non-blocking statements are not allowed in automatic tasks.

23. Find the bug in the following code.

```
always @(in)
if (ena)
out = in;
else
out = 1'b1;
```

Simulation mismatch might occur. To assure the simulation will match the synthesized logic, re-add "ena" to the event list so the event list reads: always @ (in or ena)

24. Find the bug in the following code.

```
always @(in1 or in2 or sel)
begin
out = in1;
if (sel)
out <= in2;
end
```

Not supported; cannot mix blocking and non-blocking assignments in an always statement.

25. Find the bug in the following code.

```
reg [1:0] select;
always @(select)
begin
case (select)
00: yo = 1;
01: yo = 2;
10: yo = 3;
```

11: yo = 4;

endcase

end

Branches 01 and 11 are considered as integers, and they will never be selected.

26. Fill the ????

fd = $fopen("filename", "r");

if (????)

$display("File cannot be opened");

```
// Attempt to open the file
fd = $fopen("filename", "r");

// Check if the file opened successfully
if (!fd)
    $display("File could not be opened");
```

27. How to model a perfect buffer of 10 units of delay?

a) #10 a = b;

b) a = #10 b;

c) #10 a <= b;

d) a <= #10 b;

c) #10 a <= b;

This line of Verilog code correctly uses non-blocking assignment (<=) and introduces a 10-unit delay specified by #10 between signal b and signal a.

28. Write code for a clock generator.

```
module clk1();
  parameter clk_period = 10;
  reg clk;

  initial begin
    clk = 0;
  end

  always #(clk_period/2) clk = ~clk;
endmodule
```

29. Write code for a clock generator that can generate a clock frequency of 156MHz.

```
module clk_gen;

  `timescale 1ns/1ps

  bit a, b;
  bit clk = 0;

  initial begin
    #100 $finish();
  end

  initial begin
    //forever #5 ns clk = !clk; // 100MHz
    forever #1.6 ns clk = !clk; // 156MHz
    //forever #2 ns clk = !clk; // 200MHz
  end

endmodule
```

30. Write a Verilog code to generate a 40MHz clock with a 50% duty cycle.

```
//DESIGN
`timescale 1ns/1ps

module clock_gen (  input      enable,
                    output reg clk);

  parameter FREQ = 40000;  // in kHZ
  parameter PHASE = 0;      // in degrees
  parameter DUTY = 50;      // in percentage

  real clk_pd       = 1.0/(FREQ * 1e3) * 1e9;   // convert to ns
  real clk_on       = DUTY/100.0 * clk_pd;
  real clk_off      = (100.0 - DUTY)/100.0 * clk_pd;
  real quarter      = clk_pd/4;
  real start_dly     = quarter * PHASE/90;

  reg start_clk;
```

```
  initial begin
    $display("FREQ       = %0d kHz", FREQ);
    $display("PHASE      = %0d deg", PHASE);
    $display("DUTY       = %0d %%",  DUTY);

    $display("PERIOD     = %0.3f ns", clk_pd);
    $display("CLK_ON     = %0.3f ns", clk_on);
    $display("CLK_OFF    = %0.3f ns", clk_off);
    $display("QUARTER    = %0.3f ns", quarter);
    $display("START_DLY = %0.3f ns", start_dly);
  end

  initial begin
    clk <= 0;
    start_clk <= 0;
  end

  always @ (posedge enable or negedge enable) begin
    if (enable) begin
      #(start_dly) start_clk = 1;
    end else begin
      #(start_dly) start_clk = 0;
    end
  end

  always @(posedge start_clk) begin
    if (start_clk) begin
        clk = 1;

        while (start_clk) begin
            #(clk_on)  clk = 0;
            #(clk_off) clk = 1;
        end

        clk = 0;
    end
  end
endmodule


//TB
module tb;
  wire clk1;
  wire clk2;
  wire clk3;
  wire clk4;
  reg  enable;
  reg [7:0] dly;

  clock_gen u0(enable, clk1);

  initial begin
    enable <= 0;

    for (int i = 0; i < 10; i= i+1) begin
      dly = $random;
      #(dly) enable <= ~enable;
```

```
      $display("i=%0d dly=%0d", i, dly);
      #50;
    end

    #50 $finish;
  end

  initial begin
    $dumpvars;
    $dumpfile("dump.vcd");
  end
endmodule
```

31. What are the >>> and <<< operators?

The >>> and <<< operators in Verilog are used for logical right shift and logical left shift operations, respectively. They are typically used for bitwise shifting of values.

Example:

result = value >>> 2; // Right shift "value" by 2 positions, filling with zeros

result = value <<< 3; // Left shift "value" by 3 positions, filling with zeros

32. What does the following code mean?

reg [22:0] sig;

always @(|sig) begin

// ...

end

The Verilog code snippet declares a 23-bit wide register named sig and sets up an always block that triggers whenever any of the bits in the sig register change. Inside this always block, you would typically include logic that operates based on changes in the sig register, making it suitable for implementing combinational logic based on the register's value changes.

33. What is the function of the force and release?

The force and release statements are used to override assignments on both registers and nets. They are typically used in the interactive debugging process, where certain registers or nets are forced to a value, and the effect on other registers and nets is noted. They should occur only in simulation blocks.

34. What is the purpose of declaring tasks or functions as automatic?

Declaring tasks and functions as automatic will create dynamic storage for each task or function call.

35. What is Synthesis?

Synthesis is the stage in the design flow concerned with translating your Verilog code into gates. It involves converting your Verilog design into a netlist representing the chip that can be fabricated through an ASIC or FPGA vendor.