Viva Questions for DDCO Lab

Basics of Verilog

1. What is Verilog?
Verilog is a Hardware Description Language (HDL) used to model electronic systems. It allows designers to describe the structure and behavior of digital logic circuits.

2. Why do we use Verilog instead of schematic entry?
For complex designs, Verilog is more efficient, portable, and easier to manage than graphical schematic entry. Text-based descriptions are easier to read, modify, and reuse.

3. What are the 4 levels of abstraction in Verilog?
The four levels are:

1. **Switch level:** The lowest level, modeling with transistors.
2. **Gate level:** Modeling with basic logic gates like `and`, `or`, `not`.
3. **Dataflow level:** Modeling how data flows between registers, primarily using continuous assignments (`assign`).
4. **Behavioral level:** The highest level, describing the circuit's algorithm or behavior using procedural blocks (`always`, `initial`).

4. Give examples of valid and invalid identifiers in Verilog.

- **Valid:** Identifiers must start with a letter or an underscore. Examples: `my_signal`, `_clock`, `BusA`.
- **Invalid:** Identifiers cannot start with a number or contain special characters (except _). Examples: `1input`, `my-signal`, `a&b`.

5. Is Verilog case sensitive? Give an example.
Yes, Verilog is case-sensitive. For example, the identifiers `my_signal` and `My_Signal` would be treated as two different variables.

6. What is a module in Verilog?
A `module` is the basic building block in Verilog. It encapsulates a design entity, defining its ports (inputs and outputs) and internal logic.

7. What keyword is used to end a module definition?
The `endmodule` keyword.

8. What is the difference between ports and signals?

- **Ports:** The external interface of a module, declared as `input`, `output`, or `inout`. They connect the module to the outside world.
- **Signals:** Internal connections within a module, typically declared as `wire` or `reg`. They are used to connect components inside the module.

9. Explain the rules for connecting input, output, and inout ports.

- **input:** An input port must always be a net type (e.g., `wire`). It can only be on the right-hand side of an assignment.
- **output:** An output port can be a net type (`wire`) or a register type (`reg`). It must be a `reg` if its value is assigned within a procedural block (`always` or `initial`).
- **inout:** An inout port must be a net type (`wire`). It can be used for both input and output.

10. What are vector data types in Verilog? Give an example.
A vector is a multi-bit data type that represents a bus or a group of signals.
Example: `wire [7:0] data_bus;` declares an 8-bit wire named `data_bus`.

Data Types

11. What are nets in Verilog?
Nets represent physical connections between hardware elements. They must be continuously driven by a source (like a gate output or an `assign` statement) and do not store values. The most common net type is `wire`.

12. What is the default type of a net?
If a net type is not specified, it defaults to `wire`.

13. What are registers (reg)?
A `reg` is a data type that stores a value. It holds its value until a new value is assigned to it within a procedural block (`always` or `initial`). It does not necessarily imply a physical hardware register (like a flip-flop), but it can be used to model one.

14. Difference between wire and reg.

- **wire:** Represents a physical connection. It cannot store a value and must be continuously driven. It is used on the left-hand side of `assign` statements.

- **reg:** Represents a data storage element. It holds a value between assignments. It is used on the left-hand side of assignments within procedural blocks (`always`, `initial`).

15. What is a memory in Verilog? Give an example declaration.
A memory is a two-dimensional array of registers used to model RAM or ROM.
Example: `reg [7:0] my_memory [0:255];` declares a memory with 256 locations, where each location can store an 8-bit value.

16. What is the difference between integers and reals in Verilog?

- **integer:** A 32-bit signed variable. It is synthesizable.
- **real:** A variable that can store floating-point (decimal) numbers. It is generally not synthesizable and is used for simulation purposes.

17. What is the default width of an integer in Verilog?
32 bits.

18. How is time represented in Verilog?
Time is represented using the `` `timescale `` compiler directive and delay specifications (`#delay`). The timescale specifies the time unit and precision for the simulation.

19. What are parameters in Verilog?
Parameters are constants defined within a module. They allow for creating configurable and reusable designs. Their values can be overridden during module instantiation.
Example: `parameter WIDTH = 8;`

20. What is an event variable used for?
An `event` is a data type used for synchronization between different procedural blocks in a simulation. An action can be triggered by the occurrence of a named event.

Number System and Logic

21. How do you represent numbers in binary, octal, decimal, and hexadecimal in Verilog?
The format is `<size>'<base><value>`.

- **Binary:** `'b` or `'B` (e.g., `4'b1011`)
- **Octal:** `'o` or `'O` (e.g., `4'o13`)
- **Decimal:** `'d` or `'D` (e.g., `8'd255`)
- **Hexadecimal:** `'h` or `'H` (e.g., `8'hFF`)

22. Write the Verilog representation of the decimal number 13 in binary, octal, and hexadecimal.

- **Binary:** `4'b1101`
- **Octal:** `4'o15`
- **Hexadecimal:** `4'hD`

23. What are the 4 possible logic values in Verilog?

- **0:** Logic low or false.
- **1:** Logic high or true.
- **X (or x):** Unknown logic value.
- **Z (or z):** High-impedance state (floating).

24. What does Z represent?
Z represents a high-impedance state. This is like a disconnected wire; the output is not driving a low (0) or high (1) value. It is commonly found in tri-state buffers.

25. What does X represent?
X represents an unknown or uninitialized value. This can occur if a signal has not been assigned a value yet, or if there is a conflict where a wire is being driven to both 0 and 1 simultaneously.

26. What is the truth table of AND gate when one input is Z?
If one input is Z, the output is X unless the other input is a definite 0.

- `0 & Z = 0`
- `1 & Z = X`
- `X & Z = X`
- `Z & Z = X`

27. Write the reduction operator equivalent for "all bits ANDed together".
The reduction AND operator is `&`. For a vector `a`, `&a` performs a bitwise AND on all bits of `a`.

Operators

28. List all arithmetic operators in Verilog.
`+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), `%` (modulus).

29. List all logical operators in Verilog.
`&&` (logical AND), `||` (logical OR), `!` (logical NOT).

30. What is the difference between bitwise & and logical &&?

- **Bitwise (&):** Performs an AND operation on each corresponding bit of the operands. The result has the same width as the operands.
- **Logical (&&):** Treats entire operands as boolean values (0 is false, non-zero is true). It returns a single-bit result: 1 (true), 0 (false), or X (unknown).

31. What does the concatenation operator {} do? Give an example.
The concatenation operator {} joins multiple signals or vectors into a single larger vector.
Example: If `a = 2'b10` and `b = 2'b11`, then `c = {a, b}` results in `c = 4'b1011`.

32. Write the Verilog code to reverse the bits of a 4-bit bus a.
```
assign reversed_a = {a[0], a[1], a[2], a[3]};
```

33. Write the ternary operator equivalent of a NOT gate
```
assign out = (in == 1'b1) ?  1'b0 :  1'b1;
```

34. What is the difference between blocking (=) and non-blocking (¡=) assignments?

- **Blocking (=):** Executes sequentially. The simulator waits for the assignment to complete before executing the next statement in the procedural block. It is typically used for modeling combinational logic.
- **Non-blocking (¡=):** Executes in parallel. The simulator evaluates all right-hand sides in the current time step and schedules the assignments to occur at the end of the time step. It is used for modeling sequential logic (e.g., flip-flops) to avoid race conditions.

System Tasks

35. What is $display used for?
It is a system task used in simulation to print formatted strings and variable values to the console. It automatically appends a newline character to the output.

36. Difference between $display and $write.
`$display` automatically adds a newline character at the end of its output, moving the

cursor to the next line. `$write` does not, leaving the cursor at the end of the printed text.

37. What is $strobe used for?
`$strobe` is similar to `$display`, but it prints its output at the very end of the current simulation time step, after all other assignments for that time step have been completed. This is useful for viewing the final, stable values of signals.

38. How is $monitor different from $display?
`$display` executes only when the simulator encounters it. `$monitor` is set up once and then automatically executes whenever one of its argument variables changes value. Only one `$monitor` can be active at a time in a simulation.

39. Write a $display command to print a variable in binary and decimal format.
Assuming a variable `reg [3:0] data;`, the command is:
`$display("Data is %b (binary) and %d (decimal).", data, data);`

40. What are $fopen and $fclose used for?
They are system tasks for file operations during simulation. `$fopen` opens a specified file and returns a multi-channel descriptor (MCD) handle for it. `$fclose` closes the file associated with a given MCD.

Delays & Timescale

41. What does #20 mean in Verilog?
It represents a delay of 20 time units. The actual duration of a time unit is defined by the `‘timescale` compiler directive.

42. What does timescale 1ns/1ps mean?
This directive sets the simulation’s time reference.

- `1ns`: This is the **time unit**. A delay like `#1` corresponds to 1 nanosecond.
- `1ps`: This is the **time precision**. It is the smallest time step the simulator can resolve.

43. Give an example of specifying rise and fall delays in a gate.
You can specify different delays for a gate’s output rising to 1 and falling to 0.
Example: `and #(5, 6) a1 (y, a, b);`
Here, the rise delay (0 to 1) is 5 time units, and the fall delay (1 to 0) is 6 time units.

44. Differentiate between rise, fall, and turn-off delay.

- **Rise delay:** The time taken for a signal to transition from another value (0, X, Z) to 1.

- **Fall delay:** The time taken for a signal to transition from another value (1, X, Z) to 0.
- **Turn-off delay:** The time taken for a signal to transition from any value to the high-impedance state (Z).

Modeling Styles

45. Define Gate-level modeling.
This style describes a circuit by instantiating and connecting predefined logic gates and primitives (e.g., `and`, `or`, `not`). It is a structural description of the hardware.

46. Define Data-flow modeling.
This style describes how data flows through a circuit. It is primarily done using continuous assignment statements (`assign`) and operators. It focuses on the data flow rather than the specific gate structure.

47. Define Behavioral modeling.
This is the highest level of abstraction, where the circuit is described by its behavior or algorithm using procedural blocks like `always` and `initial`. It describes what the circuit does, not how it is built with gates.

48. Write a gate-level model of a 2-input AND gate.

```
module and_gate_gl(output y, input a, b);
  and(y, a, b);
endmodule
```

49. Write a data-flow model of a 2-input AND gate.

```
module and_gate_df(output y, input a, b);
  assign y = a & b;
endmodule
```

50. Write a behavioral model of a 2-input AND gate.

```
module and_gate_bh(output reg y, input a, b);
  always @(a or b) begin
    y = a & b;
  end
endmodule
```

51. Why is a behavioural output declared as reg?
In behavioral modeling, assignments to outputs are made inside procedural blocks (`always` or `initial`). Any variable that is the target of an assignment within a procedural block must be declared as type `reg` because it needs to hold or store its value between trigger events.

52. Write the RTL code for a 2:1 MUX using assign.

```
module mux_2_to_1_df(
    output y,
    input d0, d1,
    input sel
);
  assign y = sel ? d1 : d0;
endmodule
```

53. Write a behavioral code for a 2:1 MUX using if-else.

```
module mux_2_to_1_bh_if(
    output reg y,
    input d0, d1,
    input sel
);
  always @(d0 or d1 or sel) begin
    if (sel == 1'b1)
      y = d1;
    else
      y = d0;
  end
endmodule
```

54. Write a behavioral code for a 2:1 MUX using case.

```
module mux_2_to_1_bh_case(
    output reg y,
    input d0, d1,
    input sel
);
  always @(d0 or d1 or sel) begin
    case(sel)
      1'b0: y = d0;
      1'b1: y = d1;
    endcase
  end
endmodule
```

Circuits

55. Write the data-flow code for a NOT gate.

```
module not_gate_df(output y, input a);
  assign y = ~a;
endmodule
```

56. Write the behavioral code for a NOT gate.

```
module not_gate_bh(output reg y, input a);
  always @(a) begin
    y = ~a;
  end
endmodule
```

57. Write a gate-level code for a half-adder.

```
module half_adder_gl(
    output sum, cout,
    input a, b
);
  xor(sum, a, b);
  and(cout, a, b);
endmodule
```

58. Write data-flow code for a half-adder.

```
module half_adder_df(
    output sum, cout,
    input a, b
);
  assign sum = a ^ b;
  assign cout = a & b;
endmodule
```

59. Write behavioral code for a half-adder.

```
module half_adder_bh(
    output reg sum, cout,
    input a, b
);
  always @(a or b) begin
    {cout, sum} = a + b;
  end
endmodule
```

60. Write the gate-level code for a full-adder.

```
module full_adder_gl(
    output sum, cout,
    input a, b, cin
);
  wire w1, w2, w3;

  xor(w1, a, b);
  xor(sum, w1, cin);
  and(w2, a, b);
  and(w3, w1, cin);
  or(cout, w2, w3);
endmodule
```

61. Write data-flow code for a full-adder.

```
module full_adder_df(
    output sum, cout,
    input a, b, cin
);
  assign {cout, sum} = a + b + cin;
endmodule
```

62. Write behavioral code for a full-adder.

```
module full_adder_bh(
    output reg sum, cout,
    input a, b, cin
);
  always @(a or b or cin) begin
    {cout, sum} = a + b + cin;
  end
endmodule
```

63. Why is it called a ripple carry adder?
It is called a ripple carry adder because the carry-out from each full adder "ripples" to become the carry-in of the next full adder in the chain. The final sum is not valid until the carry has propagated through all the adders.

64. Write code for a 4-bit ripple carry adder.

```
// Assuming a full_adder_df module exists as defined above

module ripple_carry_adder_4bit(
    output [3:0] sum,
    output cout,
    input [3:0] a, b,
    input cin
);
  wire c1, c2, c3;

  full_adder_df fa0(sum[0], c1, a[0], b[0], cin);
  full_adder_df fa1(sum[1], c2, a[1], b[1], c1);
  full_adder_df fa2(sum[2], c3, a[2], b[2], c2);
  full_adder_df fa3(sum[3], cout, a[3], b[3], c3);
endmodule
```

65. Write code for a 4:1 multiplexer using gate-level modeling.

```
module mux_4_to_1_gl(
    output y,
    input [3:0] d,
    input [1:0] s
);
  wire s0_n, s1_n;
  wire w0, w1, w2, w3;

  not(s0_n, s[0]);
  not(s1_n, s[1]);

  and(w0, d[0], s1_n, s0_n);
  and(w1, d[1], s1_n, s[0]);
  and(w2, d[2], s[1], s0_n);
  and(w3, d[3], s[1], s[0]);

  or(y, w0, w1, w2, w3);
endmodule
```

66. Write data-flow code for a 4:1 multiplexer.

```
module mux_4_to_1_df(
    output y,
    input [3:0] d,
    input [1:0] s
);
  assign y = s[1] ? (s[0] ? d[3] : d[2]) : (s[0] ? d[1] : d[0]);
endmodule
```

67. Write behavioral code for a 4:1 multiplexer.

```
module mux_4_to_1_bh(
    output reg y,
    input [3:0] d,
    input [1:0] s
);
  always @(d or s) begin
    case(s)
      2'b00: y = d[0];
      2'b01: y = d[1];
      2'b10: y = d[2];
      2'b11: y = d[3];
      default: y = 1'bx;
    endcase
  end
endmodule
```

68. Write code for a simple digital fan controller where Fan = T & P — O.

```
module fan_controller(
    output Fan,
    input T, P, O // T=Temp, P=Power, O=Override
);
  assign Fan = (T & P) | O;
endmodule
```

Test Benches

69. What is a test bench?
A test bench is a Verilog module created to verify the functional correctness of a design (often called the Design Under Test or DUT). It generates input stimuli, applies them to the DUT, and sometimes checks if the DUT's outputs are correct.

70. Can a test bench have input/output ports? Why/why not?
No, a test bench typically does not have any input or output ports. It is a self-contained, top-level module that simulates the environment for the DUT. It generates all stimuli internally (`reg`) and monitors the outputs (`wire`).

71. Write a test bench for a NOT gate.

```
// Assuming a NOT gate module: module not_gate(y, a);

‘timescale 1ns/1ps

module tb_not_gate;
  reg a_tb;
  wire y_tb;

  // Instantiate the Design Under Test (DUT)
  not_gate dut (.y(y_tb), .a(a_tb));

  initial begin
    $monitor("Time=%0t | Input a=%b | Output y=%b", $time, a_tb, y_tb);
    a_tb = 1’b0; #10;
    a_tb = 1’b1; #10;
    $finish;
  end
endmodule
```

72. Test bench for 2:1 MUX?
The stimulus part of the test bench would apply all possible combinations to the inputs.

```
initial begin
  sel = 1’b0; d0 = 1’b0; d1 = 1’b1; #10;
  sel = 1'b1; d0 = 1'b0; d1 = 1'b1; #10;
  sel = 1'b0; d0 = 1'b1; d1 = 1'b0; #10;
  sel = 1'b1; d0 = 1'b1; d1 = 1'b0; #10;
  $finish;
end
```

73. Test bench for half-adder?
Stimulus for a half-adder test bench.

```
initial begin
  a = 1'b0; b = 1'b0; #10;
  a = 1'b0; b = 1'b1; #10;
  a = 1'b1; b = 1'b0; #10;
  a = 1'b1; b = 1'b1; #10;
  $finish;
end
```

74. Test bench for full-adder?
Stimulus for a full-adder test bench.

```
initial begin
  a=0; b=0; cin=0; #10;
  a=0; b=0; cin=1; #10;
  a=0; b=1; cin=0; #10;
  a=0; b=1; cin=1; #10;
  a=1; b=0; cin=0; #10;
  a=1; b=0; cin=1; #10;
  a=1; b=1; cin=0; #10;
  a=1; b=1; cin=1; #10;
  $finish;
end
```

75. Test bench for RCA? Apply multiple 4-bit combinations.
Stimulus for a 4-bit Ripple Carry Adder.

```
initial begin
  a = 4'b0000; b = 4'b0000; cin = 0; #10;
  a = 4'b0001; b = 4'b0010; cin = 0; #10;
  a = 4'b0011; b = 4'b0100; cin = 1; #10;
  a = 4'b1111; b = 4'b0001; cin = 0; #10;
  a = 4'b1010; b = 4'b0101; cin = 1; #10;
  $finish;
end
```

76. Write a test bench for a 4:1 multiplexer.

```
'timescale 1ns/1ps

module tb_mux_4_to_1;
  reg [3:0] d_tb;
  reg [1:0] s_tb;
  wire y_tb;

  mux_4_to_1_bh dut(.y(y_tb), .d(d_tb), .s(s_tb));

  initial begin
    $monitor("Time=%0t | Sel=%b | Data=%b | Out=%b", $time, s_tb, d_tb, y_tb);
    d_tb = 4'b1011;
    s_tb = 2'b00; #10;
    s_tb = 2'b01; #10;
    s_tb = 2'b10; #10;
    s_tb = 2'b11; #10;
    $finish;
  end
endmodule
```

77. What is $dumpfile used for in a test bench?
`$dumpfile("filename.vcd")` is a system task that specifies the name of the Value Change Dump (VCD) file. This file will store the simulation waveform data.

78. What is $dumpvars used for?
`$dumpvars` is a system task that tells the simulator which variables' values should be "dumped" or recorded into the VCD file specified by `$dumpfile`. For example, `$dumpvars(0, tb_module)` will dump all variables within the `tb_module` and any modules instantiated within it.

Simulation & Tools

79. What is Icarus Verilog?
Icarus Verilog is a popular open-source Verilog compiler and simulator. It translates Verilog source code into an executable format that can be run to simulate the design.

80. What is GTKWave used for?
GTKWave is an open-source waveform viewer. It is used to open and display waveform files (like `.vcd` files) generated by simulators, allowing designers to visually analyze and debug their circuits' behavior over time.

81. What commands are used to compile and run a Verilog file in Icarus Verilog?

1. **Compile:** `iverilog -o output_file_name source_file.v`
2. **Run:** `vvp output_file_name`

82. Write the command to run iverilog on two files: comparator.v and stimulus.v.

```
iverilog -o comparator_test comparator.v stimulus.v
```

83. What is a .vcd file?
A `.vcd` (Value Change Dump) file is a standard ASCII text file that contains information about the value changes of signals in a Verilog simulation over time. It is generated by the simulator and used by waveform viewers like GTKWave.

84. How do you view a .vcd file?
You can view a `.vcd` file using a waveform viewer program. A common open-source viewer is GTKWave. The command would be: `gtkwave filename.vcd`

Advanced / Conceptual

85. What are User Defined Primitives (UDP)?
UDPs are a way to define custom combinational or sequential logic primitives in Verilog using a truth table format. They are limited to single-output functions and provide a way to model simple logic blocks without using modules.

86. Can UDPs have multiple outputs? Why/why not?
No, UDPs can only have a single output. This is a fundamental limitation of their definition in the Verilog language.

87. What is the difference between combinational and sequential UDP?

- **Combinational UDP:** The output depends only on the current value of the inputs. Its behavior is defined by a simple truth table.
- **Sequential UDP:** The output depends on the current inputs and the previous state. Its truth table includes an extra field for the current state to determine the next state and output. It can be level-sensitive (latches) or edge-sensitive (flip-flops).

88. What are events used for in Verilog?
Events are used for synchronization between concurrent processes (procedural blocks) in a test bench. One process can trigger an event using `-> event_name;`, and another process can wait for that event using `@(event_name)`.

89. Differentiate between initial and always blocks.

- **initial block:** Executes only once, starting at simulation time 0. It is typically used in test benches to provide stimulus and control the simulation flow.
- **always block:** Executes repeatedly. Its execution is triggered by events specified in its sensitivity list (e.g., `always @(posedge clk)` or `always @(*)`). It is used to model logic that is continuously active.

90. Can an initial block be synthesized in hardware? Why not?
No, an `initial` block is generally not synthesizable. Synthesis tools are designed to create hardware that runs continuously. An `initial` block's behavior of "run once at the start" does not have a direct physical hardware equivalent for general logic. (Note: Some FPGAs can use `initial` blocks to set the initial state of registers at power-up).

91. Explain the difference between blocking and non-blocking assignments with an example.
Blocking (=): Completes the assignment before moving to the next statement.
Non-blocking (¡=): Schedules the assignment to happen at the end of the time step.
Consider swapping two registers, `a` and `b`, on a clock edge.
Incorrect (Blocking):

```
always @(posedge clk) begin
  a = b;  // a gets the value of b
  b = a;  // b gets the NEW value of a (which is old b).
end
// Result: Both a and b end up with the original value of b.
```

Correct (Non-blocking):

```
always @(posedge clk) begin
  a <= b; // a is scheduled to get the old value of b.
  b <= a; // b is scheduled to get the old value of a.
end
// At the end of the time step, the assignments happen.
// Result: The values of a and b are correctly swapped.
```

92. What is sensitivity list in an always block?
The sensitivity list, found after the `@` symbol, is a list of signals or events that trigger the execution of the `always` block. For example, `always @(a or b)` will execute whenever signal `a` or signal `b` changes value. `always @(posedge clk)` executes on the positive edge of the clock.

93. What is the use of case statement in Verilog?
The `case` statement is a multi-way conditional statement. It checks an expression against a series of case items and executes the statement associated with the first matching item. It is often used to model multiplexers and state machines in behavioral Verilog.

94. What is a synthesizable construct? Give examples.
A synthesizable construct is a piece of Verilog code that a synthesis tool can translate into a physical hardware implementation (logic gates, flip-flops, etc.).
Examples: `assign`, `always @(posedge clk)`, `if-else`, `case`, arithmetic operators (`+`,`-`,`*`).

95. What is a non-synthesizable construct? Give examples.
A non-synthesizable construct is one that is used for simulation and verification purposes only and cannot be converted into hardware by a synthesis tool.
Examples: `initial`, `#delay`, system tasks like `$display`, `$monitor`, `$finish`.

Code-Based (Value Combinations)

96. Write a test bench for a 4-bit ripple carry adder to test values A=1010, B=0101, Cin=1.

```
‘timescale 1ns/1ps

module tb_rca_specific;
  reg [3:0] a_tb, b_tb;
  reg cin_tb;
  wire [3:0] sum_tb;
  wire cout_tb;

  ripple_carry_adder_4bit dut (
    .sum(sum_tb), .cout(cout_tb),
    .a(a_tb), .b(b_tb), .cin(cin_tb)
  );

  initial begin
    // Test the specific combination
    // A=1010 (10), B=0101 (5), Cin=1
    // Expected: 10 + 5 + 1 = 16 -> Sum = 0000, Cout = 1
    a_tb   = 4’b1010;
    b_tb   = 4’b0101;
    cin_tb = 1’b1;

    #10; // Wait for the result to propagate

    $display("Test: A=%b, B=%b, Cin=%b", a_tb, b_tb, cin_tb);
    $display("Result: Cout=%b, Sum=%b", cout_tb, sum_tb);

    $finish;
  end
endmodule
```