1. What is the purpose of the ‘initial’ and ‘always’ blocks in Verilog?
   initial block
   - Runs **once** at the beginning of simulation.
   - Used to set up initial conditions, such as resetting signals or applying test stimulus.

   always block
   - Runs **continuously** as long as the simulation is active.
   - Used to model hardware behaviour, like combinational or sequential logic.
2. Do ‘initial’ and ‘always’ block synthesize in Verilog?
   initial blocks are not synthesizable.
3. Explain the ‘final’ block with an example.
   The **final block** in Verilog is used for executing code at the **end** of a simulation. It is similar to the initial block but runs only once when the simulation finishes. It's commonly used for reporting results or cleanup actions. This block is purely for simulation purposes and is not synthesized into hardware.

```
module test;
  reg clk, reset;

  // Initial block to generate a clock signal and reset
  initial begin
    clk = 0;
    reset = 1;
    #5 reset = 0;
  end

  // Generate clock signal
  always #5 clk = ~clk;

  // Final block to print a message at the end of the simulation
  final begin
    $display("Simulation completed.");
  end
endmodule
```

4. How can a ‘forever’ loop be stopped in Verilog?
   A forever loop in Verilog runs indefinitely until it is explicitly stopped. To stop a forever loop, you can use control mechanisms like:
   - **Conditional statements (e.g., if or while)** to exit the loop.
   - **disable statement** to terminate the process.
   - **Using break** to exit the loop.
5. Can ‘initial’ and ‘always’ blocks be used inside a ‘task’?
   No, **initial** and **always** blocks **cannot** be used inside a **task** in Verilog.

   - **initial and always blocks** are meant to describe behavior that occurs independently and concurrently in a module. They define when code is executed during simulation, either at the start (initial) or continuously (always), and they exist at the module level.
   - **tasks**, on the other hand, are procedural blocks that encapsulate a sequence of operations. They are designed to be called and executed when invoked, and their execution is controlled by the calling process.
6. What happens if there is no sensitivity list?

In Verilog, the **sensitivity list** in an always block defines the events (like changes in signals) that trigger the execution of the block. If there is **no sensitivity list** specified, the behaviour depends on the simulator, but typically, the always block will behave as if it is continuously executing, which can cause issues.

7. What are blocking and non-blocking statements in Verilog?
   Blocking statement
   - **Syntax**: = (single equals sign)
   - Execution is **sequential**: A blocking statement must finish executing before the next statement starts.
   - It behaves like a traditional assignment in a programming language (e.g., C).
   - Typically used in **combinational logic**.

   Non-blocking statement
   - **Syntax**: <= (non-blocking assignment)
   - Execution is **parallel**: All non-blocking statements within a block are scheduled at the same time but are updated only at the end of the simulation time step.
   - Used in **sequential logic** (e.g., for flip-flops or registers).
   - Ensures **correct simulation of concurrent** hardware behaviour, such as clocked processes.
8. Explain continuous assignment with an example.
   In Verilog, **continuous assignment** refers to the use of the **assign** statement to drive values onto **nets** (typically wire types) continuously. The assignment is updated automatically whenever any of the right-hand side (RHS) operands change.
9. Design and execute the following:
   a. 8:3 priority encoder using ‘case’ statement.

```
module priority_encoder_8to3 (
  input [7:0] in,
  output reg [2:0] out
);
  always @(*) begin
    casez (in)
      8'b10000000: out = 3'b111; // Highest priority
      8'b?1000000: out = 3'b110;
      8'b??100000: out = 3'b101;
      8'b???10000: out = 3'b100;
      8'b????1000: out = 3'b011;
      8'b?????100: out = 3'b010;
      8'b??????10: out = 3'b001;
      8'b???????1: out = 3'b000; // Lowest priority
      default: out = 3'bxxx;     // If no input is active
    endcase
  end
endmodule
```

   b. D latch.

```
module d_latch (
  input D,
  input En,
  output reg Q
);
  always @(*) begin
```

```
    if (En)
      Q = D;  // When enable is high, Q follows D
    // Otherwise, Q retains its previous value
  end
endmodule
```

c. Decade counter.

```
module decade_counter (
  input clk,       // Clock signal
  input reset,     // Reset signal (active high)
  output reg [3:0] count  // 4-bit counter
);

  always @(posedge clk or posedge reset) begin
    if (reset)
      count <= 4'b0000;  // Reset count to 0
    else if (count == 4'b1001)
      count <= 4'b0000;  // Reset to 0 after reaching 9
    else
      count <= count + 1;  // Increment the count
  end
endmodule
```