1. What are some reusable coding practices for RTL design?
    - Modular Design: Break down the design into small, manageable modules with well-defined interfaces. This enhances reusability and testability.
    - Parameterization: Use parameters or generic types to create flexible modules that can be easily adapted for different uses.
    - Standard Naming Conventions: Follow consistent naming conventions for signals, modules, and parameters to improve readability and maintainability.
    - Document Code: Include comments and documentation for modules, detailing their functionality, inputs, and outputs.
    - Testbenches: Create reusable testbenches for modules to verify functionality and facilitate regression testing.
    - State Machine Encodings: Use consistent state machine encoding (like binary or one-hot) to simplify the design and debugging process.
    - Clock and Reset Handling: Use synchronous resets and clock-enable signals for consistency and reliability.
    - Avoid Hardcoding Values: Use parameters instead of hardcoded constants to allow easy modifications and enhance flexibility.
2. What are “snake” paths, and why should they be avoided?
   "Snake paths" refer to long combinatorial paths that zigzag across multiple logic levels, which can create long delays and complex timing issues.
   Avoidance Reason: They can lead to increased propagation delays, making timing closure difficult and increasing the chances of timing violations.
3. What are a few considerations while partitioning large designs?
    - Clock Domains: Identify and separate designs operating on different clock domains to simplify timing analysis and reduce complexity.
    - Signal Widths: Consider the width of the signals and bus interfaces. Group similar signals to minimize routing overhead.
    - Hierarchical Design: Use a hierarchical approach to manage complexity and facilitate easier debugging.
    - Module Interfaces: Define clear and minimal interfaces between partitions to reduce coupling and dependencies.
    - Resource Utilization: Keep an eye on resource usage, ensuring that partitions do not lead to excessive consumption of FPGA/ASIC resources.
4. How can I reliably convey control information across clock domains?
    - Asynchronous FIFOs: Use FIFOs to store data and control signals as they pass between clock domains, ensuring proper synchronization.
    - Handshaking Protocols: Implement handshaking (like ready/valid signals) to confirm that data has been received and processed correctly.
    - Dual-Clock Registers: Use dual-clock registers or synchronizers to transfer control signals safely between domains.
5. What is a safe strategy to transfer data of different bus widths and across different clock domains?
    - Data Packing/Unpacking: Use packing techniques to fit data from a wider bus into a narrower one, or vice versa, ensuring alignment and proper interpretation.
    - FIFOs for Buffers: Utilize FIFOs to handle differences in clock rates and widths, managing data flow efficiently.
    - Gray Code or Handshaking: Consider using Gray code for single-bit transfers or handshaking protocols to ensure data integrity.

6. What are a few considerations while using FIFOs for posted writes or prefetched reads that influence the speed of the design?
    - Depth and Width: The size of the FIFO can impact performance; deeper FIFOs can absorb bursty traffic but may introduce latency.
    - Timing: Ensure that the read and write timings are compatible to prevent data loss or corruption, especially during underflow or overflow conditions.
    - Control Signals: Use appropriate flags (full/empty) and control logic to manage data flow effectively, ensuring efficient operation.
    - Latency: Consider the added latency introduced by FIFO buffers and how it affects overall system performance.
7. What will be synthesized from a module with only inputs and no outputs?
   A module with only inputs and no outputs will typically synthesize into a no-operation or effectively do nothing in terms of logic. It might consume area for the inputs but won't contribute any functional logic to the design.
8. What are "combinatorial timing loops"? Why should they be avoided?
   Combinatorial timing loops occur when feedback paths exist in combinatorial logic, creating circular dependencies.
   Avoidance Reason: They can cause oscillations, and unpredictable behavior, and are difficult to analyze and debug, leading to functional failures.
9. How does the sensitivity list of a combinatorial always block affect pre and post-synthesis simulation? Is this still an issue lately?
   The sensitivity list defines which signals trigger the execution of the block.
   A correct sensitivity list is crucial for accurate simulation results, ensuring the design behaves as expected in both pre and post-synthesis simulations.
   In modern synthesis tools, using always @* (or always @(*)) can automatically include all inputs, reducing the likelihood of issues with missing signals.
10. Design an 8-bit counter by using a forever loop. The counter starts counting at count=5 and finishes at count=67. The count is incremented at the positive edge of the clock. The clock has a time period of 10. The counter counts through the loop only once and then is disabled.

```
module counter (
  input wire clk,
  input wire rst,
  output reg [7:0] count
);
  initial begin
    count = 8'd5; // Initialize count to 5
  end

  always @(posedge clk or posedge rst) begin
    if (rst) begin
      count <= 8'd5; // Reset count to 5
    end else if (count < 8'd67) begin
      count <= count + 1; // Increment count
    end else begin
      disable count; // Stop the counting process
    end
  end
endmodule
```