1. What things should you consider while implementing messaging in a model?
    - Message Levels: Implement message verbosity (e.g., error, warning, info, debug) to control the amount of information displayed.
    - Message Consistency: Ensure uniform message formatting for readability.
    - Message Filtering: Allow filtering based on severity levels.
    - Time Stamping: Include time stamps for debugging and tracking.
    - Performance: Ensure messages don't cause performance bottlenecks.
2. How are message levels implemented in a BFM?
   BFM uses different levels of messaging, such as:
    - Errors: Critical issues that stop the simulation.
    - Warnings: Non-fatal issues that may affect functionality.
    - Info: General information about operations.
    - Debug: Detailed messages for debugging.
3. What is a BUS Functional Model (BFM)?
   A BFM is a simulation model that mimics the behaviour of a bus or protocol without internal logic. It generates stimuli or responds to transactions based on bus operations and helps in testing design interfaces.
4. What are a few considerations that go into designing a BFM?
    - Protocol Compliance: Ensure the BFM accurately follows the bus protocol.
    - Transaction Speed: Balance between timing accuracy and performance.
    - Configurability: Design for flexible transaction generation (address, data, delays).
    - Error Injection: Support for testing corner cases and error conditions.
5. What is a typical flow in designing a BFM?
    - Specification Study: Understand the protocol or bus thoroughly.
    - Basic Transaction Design: Implement basic read/write and handshaking logic.
    - Timing Modeling: Introduce accurate timing for transaction events.
    - Error Handling: Implement error injection and handling scenarios.
    - Testing: Verify with various test cases and protocols.
6. How can BFMs be used to inject intentional errors in the stimulus?
   BFMs can inject errors by:
    - Sending incorrect addresses or data.
    - Violating timing constraints like setup or hold times.
    - Misaligning handshake signals.
7. What are the main responsibilities of a bus monitor?
    - Transaction Monitoring: Capture all bus transactions.
    - Error Detection: Detect protocol violations or incorrect data/addresses.
    - Logging: Log transactions and errors for debugging.
    - Coverage Collection: Monitor transactions to assess test coverage
8. Design of bus monitor and explain it's working.
    - Signal Capture: Monitor bus signals (e.g., data, address, control).
    - State Machine: Use a state machine to track the bus states and transactions.
    - Error Detection Logic: Compare observed transactions against protocol rules to identify errors.
    - Log Transactions: Capture all valid transactions with relevant details (address, data, timestamps).
9. What are the considerations in designing a Monitor?
    - Protocol Adherence: Must accurately reflect bus protocol timing and rules.
    - Latency: Ensure minimal delay in monitoring to avoid affecting performance.
    - Configurability: This should be adaptable to different test environments and buses.
    - Error Reporting: Efficiently report errors without overwhelming the user.

10. A 1-bit full subtractor has 3 inputs x,y and z(previous borrow) and 2 outputs D(difference) and B(borrow). The logic equations for D and B are as follows:
D = x’y’z+x’yz’+xy’z’+xyz
B=x’y+x’z+yz

a. Write the Verilog RTL description for the full subtractor.
```
module full_subtractor (
    input x, y, z,       // Inputs: x, y, z (previous borrow)
    output D, B          // Outputs: D (difference), B (borrow)
);
    assign D = (~x & ~y & z) | (~x & y & ~z) | (x & ~y & ~z) | (x & y & z);
    assign B = (~x & y) | (~x & z) | (y & z);
endmodule
```
b. Optimize for fastest timing.
```
module full_subtractor_optimized (
    input x, y, z,    // Inputs: x, y, z (previous borrow)
    output D, B       // Outputs: D (difference), B (borrow)
);
    // Optimized equations for Difference (D) and Borrow (B)
    assign D = x ^ y ^ z;                // XOR gates are fast
    assign B = (~x & (y | z)) | (y & z);  // Optimized borrow logic with minimal depth
endmodule
```
c. Apply stimulus to verify that the design function properly.
```
module test_full_subtractor;
    reg x, y, z;
    wire D, B;

    full_subtractor uut (.x(x), .y(y), .z(z), .D(D), .B(B));

    initial begin
        // Test all input combinations
        $display("x y z | D B");
        $monitor("%b %b %b | %b %b", x, y, z, D, B);

        // Apply test cases
        x = 0; y = 0; z = 0; #10;
        x = 0; y = 0; z = 1; #10;
        x = 0; y = 1; z = 0; #10;
        x = 0; y = 1; z = 1; #10;
        x = 1; y = 0; z = 0; #10;
        x = 1; y = 0; z = 1; #10;
        x = 1; y = 1; z = 0; #10;
        x = 1; y = 1; z = 1; #10;

        // Finish simulation
        $finish;
    end
endmodule
```
d. Print error messages whenever necessary.

```
initial begin
   // Apply test cases
   x = 0; y = 0; z = 0; #10; if (D !== 0 || B !== 0) $display("Error: x=0, y=0, z=0");
   x = 0; y = 0; z = 1; #10; if (D !== 1 || B !== 1) $display("Error: x=0, y=0, z=1");
end
```