VERILOG ASSIGNMENT 3

A. Behavioral Modeling Basics (Initial & Always Blocks)

1. Write a module that toggles a 1-bit signal using `initial` and `always` blocks.
2. Simulate a 16-bit register being initialized and incremented inside an `always` block.
3. Use multiple `always` blocks in a module and simulate how they execute in parallel.
4. Explain the functional difference between `initial` and `always` with simulation output.
5. Design a clock generator using `initial` and `forever` loop.
6. Implement a module that uses `initial` for setup and `always` for functional updates.
7. What happens if you use multiple `initial` blocks in the same module? Demonstrate.
8. Simulate two `always` blocks modifying different signals—observe timing.

B. Procedural Assignments: Blocking vs Non-Blocking

9. Design a module that uses blocking assignments to assign two values in order.
10. Modify the above to use non-blocking assignments and compare the outputs.
11. Explain how = and <= affect simulation order with examples.
12. Write a testbench for a blocking-assigned flip-flop—verify its output.
13. Create a simple ALU module using only blocking assignments.
14. Create a pipelined register using non-blocking assignments.
15. Demonstrate how non-blocking assignments allow concurrent updates.
16. Write a code where incorrect usage of blocking assignment causes simulation mismatch.
17. Modify a race-prone code using = into a race-free one using <=.
18. Mix = and <= in one block and explain simulation output.

C. Race Conditions & Scheduling

19. Demonstrate a race condition using two always blocks assigning to the same variable.
20. Show how race condition is resolved using non-blocking assignments.
21. Explain the three-step process of non-blocking execution: Read, Evaluate, Schedule.
22. Create a race condition intentionally using blocking statements—analyze results.
23. Use `$display` to show wrong outputs due to race and fix it using proper constructs.
24. Simulate two flip-flops exchanging values with blocking and non-blocking assignments.

D. Looping Constructs

25. Write a `forever` loop to generate a square wave on a signal.
26. Design a counter using a `repeat` loop and simulate 8 cycles.
27. Use a `while` loop to count the number of 1s in a 4-bit vector.
28. Implement the above logic using a `for` loop instead of `while`.
29. Initialize a 16-byte memory with `repeat` loop.
30. Simulate a buffer that stores data every positive clock edge using `repeat`.
31. Write a module using `forever` loop and disable it after a time limit.
32. Use a `while` loop to shift a register left until MSB is 1.
33. Use `for` loop to compute parity of an 8-bit signal.
34. Count down from 10 to 0 using a `while` loop and show each value with `$display`.
35. Show a waveform that differentiates `for`, `repeat`, and `forever` behaviors.
36. Explain why `forever` loops are not synthesizable and how to work around them.

E. Conditional Statements

37. Implement a 2:1 multiplexer using `if` statement.
38. Extend the above to 4:1 mux using nested `if…else`.
39. Design a 4-bit counter with synchronous reset using `if…else`.
40. Demonstrate conditional logic that checks 3 flags and updates output accordingly.
41. Use nested `if` statements to implement an encoder.
42. Simulate a logic block using `if-else if-else` for priority-based selection.
43. Write a testbench that verifies `if` logic using multiple input patterns.
44. What is the result of omitting the final `else` in a priority selector?

F. Case, Casex, Casez Statements

45. Implement a 4:1 mux using `case` statement.
46. Modify the design to use `casex` to ignore undefined (x) bits.
47. Use `casez` to implement a pattern-matching decoder.
48. Create a testbench to show difference between `case`, `casex`, and `casez`.
49. Simulate a bus decoder using `case` with default statement.
50. Show how improper use of `casex` can lead to incorrect matching.

G. Generate Block & Structural Coding

51. Use `generate` and `genvar` to initialize an 8-bit register array.
52. Design a 4-bit ripple-carry adder using `generate` and `full_adder` module.
53. Convert Gray code to Binary using `generate` loop structure.
54. Implement an n-bit data bus multiplexer using `generate`.

H. Guidelines & Best Practices

55. Create a module that mixes `=` and `<=` in the same always block—fix the violation.
56. Separate combinational and sequential logic using different blocks with correct assignment types.
57. Explain why mixing blocking and non-blocking in the same block is discouraged—illustrate.
58. Write a module with multiple `always` blocks assigning to the same variable—explain why it's wrong and fix it.