**CLASS BASED TEST BENCH AND UVM BASED TEST BENCH REPORT**

This project presents a class-based testbench built in SystemVerilog and also UVM to verify the functionality and correctness of an asynchronous FIFO.

**CLASS BASED TEST BENCH:**

Test bench Architecture:

Interface:

* Acts as the communication bridge between the DUT and testbench.
* Declares all relevant input/output signals (e.g., wr\_en, rd\_en, data\_in, data\_out).
* Defines clocking blocks for both write and read domains, which helps synchronize the signal sampling and driving based on the respective clocks.
* Supports modports to partition signal access rights (e.g., driver vs monitor).

Driver:

* Takes high-level transactions (generated by the generator) and converts them into low-level signal activity.
* Uses the interface to drive signals such as data, write enable, and read enable to the DUT.
* Mimics real-world usage patterns by applying input stimulus with proper timing.
* Operates based on the write clock, adhering to valid/ready handshaking or protocol constraints.

Generator:

* Generates randomized input stimulus (data values, read/write sequences).
* Creates corner cases (e.g., burst writes, simultaneous read/write, FIFO full/empty transitions).
* Sends abstract transactions to the driver for execution.
* Can be easily configured to generate a specific number or pattern of transactions.

Monitor:

**monitor\_in**:

* Watches input signals driven to the FIFO.
* Records the actual data and timing of write operations.

**monitor\_out**:

* Observes the data being read from the FIFO.
* Captures the output under read enable conditions.

Scoreboard:

* Acts as the checker or reference model of the testbench.
* Receives expected input values (from monitor\_in) and actual output values (from monitor\_out). Compares the sequences to validate correctness of FIFO operation:
* Ensures data integrity (written data matches read data).
* Verifies ordering (FIFO behavior: First-In, First-Out).
* Any mismatches or unexpected outputs are flagged as errors.

Environment:

* Top-level testbench controller that instantiates and connects all other modules:
* Generator
* Driver
* Monitors
* Scoreboard
* Ensures all components are properly initialized and ready to interact.
* Controls the execution flow of the simulation from setup to test completion.

Test:

* Contains the actual test scenario (number of transactions, patterns, delays, etc.).
* Creates the environment instance and calls its run task.
* May apply random seeds, reset sequences, or custom traffic profiles.

Package:

* Includes common class definitions, enumerations, and configuration parameters.
* Ensures all testbench modules share the same data structures and transaction formats.
* Helps reduce duplication and increase consistency across the environment.