**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.

**UVM BASED TEST BENCH:**

The testbench verifies a parameterized asynchronous FIFO module using the Universal Verification Methodology (UVM). The FIFO has distinct read and write clocks, and its condition is indicated by the status flags (full, half\_full, empty, and half\_empty). Standard UVM elements including agents, environment, sequences, drivers, monitors, and a scoreboard form the foundation of the UVM testbench.

DUT and Interface:

The testbench and the asynchronous FIFO DUT communicate via the interface fifo\_if. DATA\_WIDTH and DEPTH, which parameterize it, have default values of 32 and 256, respectively. The interface consists of:  
Write Side (wr\_clk): full, half\_full, wr\_en, wr\_data

Read Side (rd\_clk): empty, half\_empty, rd\_en, rd\_data

By lining up stimulus and sampling with clock edges, clocking blocks (cb\_wr, cb\_rd) are defined for synchronization in the driver and monitor, increasing simulation dependability. The same parameters are used by the DUT instantiation in tb\_top.sv, which connects the interface signals appropriately. Two distinct clock domains—500 MHz for write and 250 MHz for read—are provided to capture the asynchronous nature of the FIFO.

Testbench Top Module (tb\_top.sv): This is where the simulation starts. It specifies the instantiates, clocks, and resets:

-> The interface for FIFO (vif)

-> The async\_fifo DUT

-> Using UVM's run\_test("fifo\_test") method, the test environment

After a little delay, the asynchronous reset is deasserted, and the UVM test is then launched after a one-clock-cycle wait. For worldwide access, the virtual interface is sent to the UVM configuration database (uvm\_config\_db).

UVM Interface Integration:

The UVM components (driver and monitor) use uvm\_config\_db to access the virtual interface. This handle is retrieved by every UVM component throughout the build process. The driver uses clocking blocks (cb\_wr, cb\_rd) to apply transactions in a synchronized manner.

UVM Components and Environment:

Transaction (fifo\_item):

The fifo\_item is a sequence item class derived from uvm\_sequence\_item. It models the data transferred across the FIFO:

* Write enable (wr\_en)
* Read enable (rd\_en)
* Write data (wr\_data)
* Read data (rd\_data) — captured by the monitor

Constraints ensure:

* At most one operation per cycle (read or write)
* Write data is less than 256 (wr\_data < 256)

Sequence (fifo\_master\_seq):

This class defines the stimulus scenarios and drives the DUT through the driver. It has four main phases:

1. Fill to FULL**:** Writes DEPTH elements to the FIFO.
2. Overflow Writes**:** Additional writes to test overflow condition.
3. Drain to EMPTY**:** Reads DEPTH elements to empty the FIFO.
4. Random R/W Burst**:** Randomly generated read/write transactions with constraints.

Each transaction is created using start\_item() and finish\_item() calls to the driver.

Driver (fifo\_driver):

The driver applies stimulus to the DUT based on the received fifo\_item transactions:

* If wr\_en is high, a write pulse is generated using the cb\_wr clocking block.
* If rd\_en is high, a read pulse is generated using the cb\_rd clocking block.

After applying the transaction, it calls item\_done() to signal completion.

Monitor (fifo\_monitor):

The monitor samples signals from the DUT and sends observed transactions to the scoreboard:

* Observes wr\_en and wr\_data on write clock.
* Observes rd\_en and rd\_data on read clock.
* Covergroups (wr\_cov and rd\_cov) are defined to capture functional coverage over enable signals and FIFO status flags.

The monitor uses an analysis port (mon\_ap) to forward transactions to the scoreboard.

Agent (fifo\_agent):

Encapsulates the sequencer, driver, and monitor. In the connect\_phase, the sequencer is connected to the driver's seq\_item\_port.

Scoreboard (fifo\_scoreboard):

Implements reference model and checking logic:

* Maintains a queue (exp\_q) for expected write data.
* Enqueues data on every valid write transaction (unless FIFO is full).
* On read transactions, it compares the dequeued expected value with the actual rd\_data.
* Logs mismatches using uvm\_error and tracks total error count (err\_cnt).

The scoreboard receives observed transactions via an analysis export (sb\_exp), connected to the monitor’s analysis port.

Environment (fifo\_env):

Comprises the fifo\_agent and fifo\_scoreboard. In the connect\_phase, the monitor’s analysis port is connected to the scoreboard’s export for transaction forwarding.

UVM Test (fifo\_test):

The top-level test instantiates the environment and launches the sequence.

* In the build\_phase, the environment is created.
* In the run\_phase, the test raises an objection, starts the fifo\_master\_seq on the agent’s sequencer, and drops the objection once done.
* In the report\_phase, the test reports success if err\_cnt is zero, or fails with the number of mismatches.