Fundamentals of Pre-Silicon Validation
Winter -2025

Implementation and Verification of Asynchronous FIFO using both Class based and UVM methodologies.

Class Based TB Implementation

Team -1

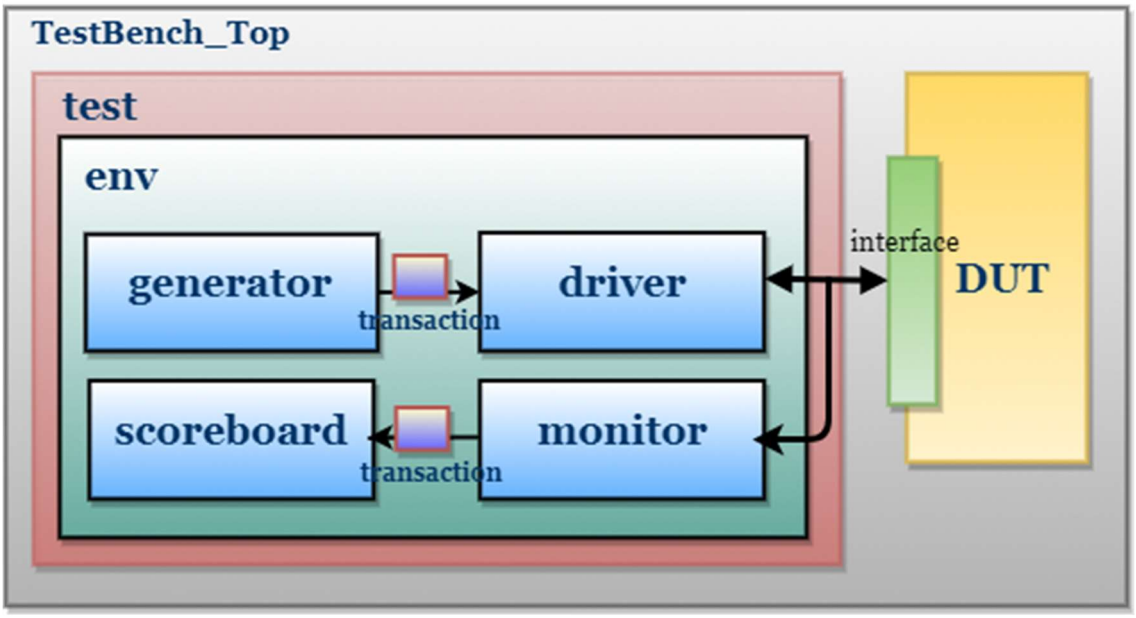
Bhargav Chunduri -- bhargavc@pdx.edu

Dhushyanth Dharmavarapu – dharmava@pdx.edu

Venkata Krishna Kumar vedantam – vedantam@pdx.edu

For Verifying this Asynchorous FiFO, we employed Proper SV test bench for architecture with following:

1. trans_fifo.sv
2. gen_fifo.sv
3. driv_fifo.sv
4. mon_fifo.sv
5. scb_fifo.sv
6. environment.sv
7. testbench.sv
8. test.sv
9. fifo_coverage.sv



1. **trans.fifo.sv** :
    a) The trans_fifo class defines a transaction model for FIFO operations, incorporating randomized control and data signals. It includes parameters for data width (d_width) and pointer width (p_width), ensuring scalability. The class generates random w_inc and r_inc signals to simulate write and read operations, along with randomized write data (wdata). Additionally, it

maintains read data (rdata), FIFO status flags (rempty, wfull), and address pointers (waddr, raddr).

b) To ensure valid address operations, constraints limit waddr and raddr within their respective ranges. Similarly, wdata and rdata are constrained within their valid data width range. The FIFO state probabilities are adjusted using distribution constraints, making wfull and rempty more likely to be 0, simulating normal FIFO operation. A burst_id variable is included to track transactions within a burst, facilitating better organization and analysis of FIFO transactions.

2. gen_fifo.sv:

a) The gen_fifo class is responsible for generating transactions and sending them to the driver for processing. It utilizes a mailbox (genrator2driv) for synchronized communication between the generator and driver. The class includes a constructor (new) that initializes the mailbox and an event (ended) to signal completion.

b) The main task generates a sequence of transactions based on the size and count parameters. It alternates between read (r_inc=1, w_inc=0) and write (w_inc=1, r_inc=0) operations, ensuring a balanced FIFO workload. Transactions are randomized with constraints to control their behavior, and each transaction is sent to the driver via the mailbox.

c) After completing the specified number of transactions, an end transaction (endtransx.temp_trans=1) is created and sent to indicate the termination of the generation phase. Throughout execution, debug messages provide insights into transaction details and the generator's progress.

3. driv_fifo.sv:

a) The driv_fifo class is responsible for driving transactions from the generator to the DUT using a virtual interface (intfc). It interacts with the FIFO design by asserting control signals for read and write operations.
b) The class includes a reset task that waits for the reset signals (w_rst and r_rst), ensuring that all control signals and data lines are properly initialized before starting transactions.
c) The drive task continuously retrieves transactions from the mailbox (genrator2driv) and applies them to the DUT. It ensures that write

operations occur only when the FIFO is not full and read operations happen only when it is not empty.

d) Each transaction is logged with a unique Transaction_ID, displaying relevant details such as w_inc, r_inc, waddr, and raddr. The task keeps track of the number of executed transactions and ensures synchronization with the DUT clock edges.
e) If a termination transaction (temp_trans) is received, it signals the end of transaction processing, preventing further driving operations.

4. mon_fifo.sv:

a) The mon_fifo class is responsible for observing transactions on the FIFO interface and forwarding them to the scoreboard for verification. Below are its key functionalities:
b) **Virtual Interface Connection**: The monitor interacts with the DUT through a virtual interface (vir_inf), ensuring flexibility and modularity in the testbench.
c) **Mailbox-Based Communication**: Transactions captured by the monitor are sent to the scoreboard using a mailbox (monitor2scb), enabling synchronization and result comparison.
d) **Continuous Monitoring**: The main task runs indefinitely, detecting write (w_inc) and read (r_inc) operations and capturing relevant signals, including addresses, data, and FIFO status flags (wfull and rempty).
e) **Transaction Logging**: Each transaction is assigned a unique transaction ID (mon_transfer), and detailed information about the operation is displayed for debugging and analysis.
f) **Seamless Integration with Scoreboard**: Once a transaction is recorded, it is forwarded to the scoreboard for validation, ensuring correctness in FIFO behavior and protocol compliance.

5. scb_fifo.sv :

a) The scb_fifo class serves as the scoreboard, validating FIFO transactions and tracking test results. Below are its key functionalities:

b) **Transaction Monitoring**: It receives transactions from the monitor via a mailbox (monitor2scb) and processes them to check correctness.

c) **Write Data Storage:** When a write (w_inc) operation occurs without FIFO being full (wfull), the data is stored in a queue (data_queue) for later validation.

d) **Read Data Verification**: During a read (r_inc) operation, the received data is compared with the expected data, ensuring FIFO correctness.

e) **Pass/Fail Tracking:** The scoreboard maintains counters (passcount and failcount) to track the number of successful and failed transactions, aiding in test analysis.

f) **Detailed Logging**: It provides debug messages with transaction details and final test results, helping identify errors and verify FIFO functionality.

6. environment.sv

a) The fifo_environment class integrates all testbench components, including the generator, driver, monitor, and scoreboard, to create a cohesive verification environment.

b) **Component Initialization:** It instantiates and connects the generator, driver, monitor, and scoreboard using mailboxes for communication and an event for synchronization.

c) **Test Execution Flow**: The environment provides structured tasks (bef_test, test, and aft_test) to handle reset, transaction execution, and post-test verification.

d) **Parallel Execution**: The test task uses a fork-join mechanism to execute generator, driver, monitor, and scoreboard operations concurrently.

e) **Simulation Control:** The run task orchestrates the test sequence and terminates the simulation upon completion.

7. testbench.sv :

a) The fifo_testbench module sets up the test environment for verifying the asynchronous FIFO design.

b) **Clock and Reset Generation**: It defines and toggles read (r_clk) and write (w_clk) clocks and controls reset signals to initialize the FIFO.

c) **Interface Connection**: The intfc instance connects the DUT (Asynch_fifo_top) with the testbench, facilitating signal interaction.

d) **Test and Coverage Integration**: It instantiates the test module to execute the verification process and fifo_coverage to collect functional coverage metrics.

8. test.sv:

a) The test program instantiates the fifo_environment class, which contains the generator, driver, monitor, and scoreboard for the FIFO verification.

b) **Environment Initialization:** It initializes the fifo_environment instance with the provided virtual interface vir_inf.

c) **Configuration:** The program sets the burst count to 1024 and the burst size to 20 for the FIFO generator.

d) **Test Execution:** It calls the run() task to execute the testbench, which triggers the simulation and verification of the FIFO design.

9. fifo_coverage.sv:

a. The fifo_coverage module provides functional coverage for the FIFO design by tracking key operations such as write and read actions, and FIFO state transitions.

b. **Signal Tracking**: The module tracks write and read enables, reset conditions, and FIFO full/empty statuses through logic signals.

c. **Covergroups:** Two covergroups (tewrite and tread) are defined, each triggered by the write and read clocks respectively. These covergroups cover various aspects of the FIFO, including write enable, read enable, and specific data values.

d. **Coverage Initialization**: The coverage objects are instantiated and initialized in the initial block, enabling coverage collection during the simulation.

e. **Write and Read Operations**: The module defines specific coverage points for both write and read operations, such as tracking write enable (w_inc), write data (wdata), read enable (r_inc), and read data (rdata), including edge cases for specific data values like 8'hFF, 8'h00, 8'hAA, and 8'h55.

f. **FIFO Full/Empty States**: Coverage is added for FIFO states, such as when the FIFO is full or empty, as well as near-full and near-empty conditions, ensuring these scenarios are covered during simulation.

g. **Reset Conditions**: The coverage groups also track the reset behavior (w_rst and r_rst), including assertions, deassertions, and transitions, to ensure proper reset handling is observed throughout the simulation.

h. **Toggle Tracking**: It tracks toggling behavior for key control signals like w_inc, r_inc, and reset conditions to capture all potential transitions and their impact on FIFO operation.

i. **Coverage for Data Values**: Data values are checked across the entire range ([0:255]), with special bins for edge data values, enabling comprehensive coverage of the data-handling functionality.

j. **Overall Test Coverage**: The module aims to ensure that all key operational conditions of the FIFO are tested by providing broad and detailed coverage for both read and write processes, ensuring robustness in the design verification.