Fundamentals of Pre-Silicon Validation
Winter -2025

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

UVM Based TB Implementation

Team -1

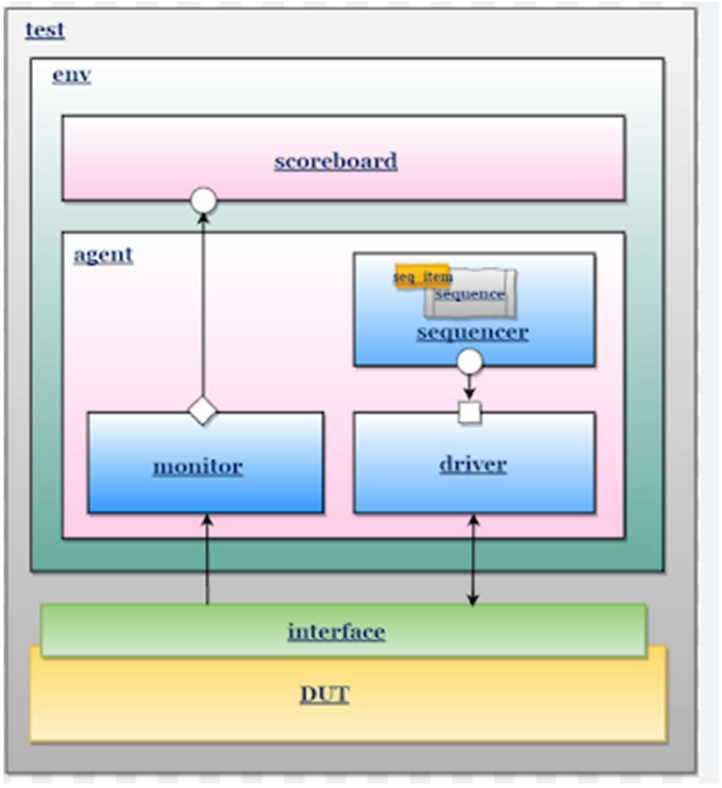
Bhargav Chunduri -- bhargavc@pdx.edu

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:

- fifo_uvm_top.sv
- fifo_test.sv
- fifo_env.sv
- agent.sv
- fifo_scoreboard.sv
- driver.sv
- sequencer.sv
- monitor.sv
- fifo_sequence_item.sv
- sequencer_fifo_wr.sv



1. **fifo_uvm_tb_top.sv**:
    - The testbench includes multiple UVM components by importing the uvm_pkg and relevant SystemVerilog files, ensuring a structured verification environment.
    - A SystemVerilog interface (intfc) is instantiated to handle signal communication between the testbench and the DUT, simplifying connectivity.
    - The UVM test is initiated using run_test("uvmtest");, which triggers the

execution of the predefined test sequences and environment setup.

- Clock signals for asynchronous operation are generated, with r_clk toggling every 4ns and w_clk toggling every 2ns to mimic real-world FIFO timing constraints.
- Reset signals are asserted at the start and deasserted sequentially after 15ns to ensure a proper initialization sequence before the test begins.

2. fifo_test.sv:

- The test class uvmtest extends uvm_test and is registered using \uvm_component_utils(uvmtest)`, allowing it to be factory-created within the UVM framework.
- The build_phase creates an instance of fifo_env, which serves as the top-level environment containing the agent, scoreboard, and other verification components.
- The run_phase raises an objection to keep the simulation active and sequentially starts reset, write, and read sequences using seq_rst, seq_wr, and seq_rd.
- The connect_phase is included but does not perform any explicit connections, allowing for future expansion if needed.
- The end_of_elaboration_phase prints the UVM testbench topology, providing a structured view of instantiated components for debugging and verification.

3. fifo_env.sv:

- The fifo_env class extends uvm_env and is registered with \uvm_component_utils(fifo_env)`, enabling factory-based creation within the UVM environment.
- The build_phase instantiates the agent (agnt) and fifo_scoreboard (scb), ensuring that all necessary verification components are available for simulation.
- The connect_phase links the monitor's analysis port to the scoreboard's analysis port using agnt.mon.monitor_port.connect(scb.scoreboard_port), enabling data flow for checking expected vs. actual results.
- The run_phase executes but does not perform additional operations, allowing for future extensions if required.
- Throughout the environment, UVM messages are logged at different phases to assist in debugging and tracking the flow of execution.

4. agent.sv:

- The agent class extends uvm_agent and is registered with \uvm_component_utils(agent)`, enabling factory creation within the UVM framework.
- The build_phase instantiates key verification components, including the sequencer (seq), driver (driv), and monitor (mon), ensuring proper communication between stimulus generation, DUT interaction, and observation.

- The connect_phase establishes a connection between the driver and sequencer using driv.seq_item_port.connect(seq.seq_item_export);, allowing transactions to flow from the sequencer to the driver.
- The run_phase executes but does not perform additional operations, providing flexibility for future enhancements if needed.
- Throughout the agent, UVM messages are logged at different phases, assisting in debugging and tracking the execution flow.sent to the scoreboard using a mailbox (monitor2scb), enabling synchronization and result comparison.

5. fifo_scoreboard.sv :

- The fifo_scoreboard class extends uvm_test instead of uvm_scoreboard, which is unconventional but still functions as a scoreboard by tracking and comparing expected versus actual FIFO transactions.
- The build_phase initializes the scoreboard_port using uvm_analysis_imp, enabling it to receive transaction data from the monitor for analysis.
- The write function stores incoming transactions in a queue (trans) and pushes written data into trans_data if the write operation (w_inc) is valid and the FIFO is not full.
- The read task retrieves expected data from trans_data, compares it with the actual read value, and logs either a success or an error message if there is a mismatch.
- The run_phase continuously waits for transactions to be available, pops them from the queue, and calls the read function for validation, ensuring real-time monitoring of FIFO operations.

- The fifo_scoreboard class serves as the scoreboard, validating FIFO transactions and tracking test results.
- The fifo_scoreboard utilizes a dynamic array trans to store transaction items (fifo_seq_item), which allows for flexible management of FIFO data during simulation.
- The write function pushes data to the front of trans_data when a valid write transaction is detected (w_inc), ensuring that the most recent data is processed first, adhering to FIFO principles.
- The read task ensures synchronization by blocking until there is data to process (wait(trans.size != 0)), preventing race conditions and ensuring proper transaction handling in the scoreboard.
- The scoreboard provides real-time feedback through uvm_info and uvm_error messages, making it easier to debug FIFO operations by tracking both successful and failed read/write comparisons.

6. driver.sv

- The driver class extends uvm_driver and is responsible for generating and driving transactions (fifo_seq_item) to the DUT. It retrieves its virtual interface (vif) from the UVM configuration database during the build_phase.
- In the connect_phase, the class doesn't have explicit connections, but it is prepared for any necessary setup or future expansion.
- The run_phase continuously fetches transactions from the sequencer using seq_item_port.get_next_item(drv_pkt), and it calls the drive task to apply the transaction to the DUT.
- The drive task checks the type of transaction (write or read) and drives the appropriate signals to the virtual interface (vif), ensuring correct timing with respect to the w_clk or r_clk. It also provides logging information using uvm_info for both write and read operations.
- The write and read operations are separated, with the write operation controlled by w_inc and r_inc flags, and the task ensures that the correct data and control signals are applied to the DUT for each operation.
- The driver class ensures robust error handling by checking the successful retrieval of the virtual interface (vif) from the UVM configuration database in the build_phase. If the interface is not found, an error message is logged using uvm_error.
- The run_phase operates in a continuous loop (forever), meaning that it consistently processes transactions as they arrive from the sequencer, driving the values to the DUT, and ensuring that transactions are not skipped or missed.
- The drive task separately handles write and read operations. For write operations, it drives data (wdata) to the DUT and waits for a clock cycle (@(posedge vif.w_clk)), ensuring data is latched correctly. Similarly, for read operations, it drives the read signal (r_inc) and waits for the appropriate clock (@(posedge vif.r_clk)), ensuring synchronized data retrieval.
- The driver class also provides detailed transaction information during both write and read operations, helping the testbench monitor and verify correct signal behavior, including the status of flags such as wfull (write full) and rempty (read empty).
- The use of uvm_info within the drive task offers real-time insights into the ongoing operations, making it easier to debug and trace the flow of transactions, and providing visibility into the behavior of the FIFO in different operational states.

7. sequencer.sv :

- The sequencer class extends uvm_sequencer and is designed to generate and sequence fifo_seq_item transactions in a UVM-based testbench. It is registered with \uvm_component_utils(sequencer)` to enable factory creation and manage component lifecycle.
- In the new function, the class logs an informational message using uvm_info to indicate when the class constructor is called.
- The build_phase calls the parent class's build_phase and logs another message, providing visibility into the phase execution. This is where the sequencer would

typically create any internal components or configurations, though no specific setup is done in this implementation.

- The connect_phase also logs a message and ensures that any necessary connections between the sequencer and other components are established. This phase is often used to connect the sequencer to the driver or other verification components.
- The class does not currently contain a sequence to generate transactions, but it is set up to manage and control the sequencing of transactions, allowing it to be expanded in future development phases.

8. sequence_fifo_wr.sv:

- The fifo_sequence class extends uvm_sequence and is used to generate FIFO sequence items (fifo_seq_item) for the simulation. It contains a constructor that logs an informational message using uvm_info and a task body that generates a sequence of transactions with randomized w_rst and r_rst values. The sequence item is then completed with finish_item.
- The fifo_sequence_wr class, also extending uvm_sequence, generates a sequence of write transactions. In the task body, it loops 16 times, randomizing each fifo_pkt_wr item with specific values (w_inc == 1 and r_inc == 0) to simulate write operations. It logs the item generation details and displays the process before finishing each item. The total number of items generated is also logged at the end.
- The sequence_fifo_rd class follows a similar structure to fifo_sequence_wr, but instead generates read transactions. The fifo_rd_pkt item is randomized with w_inc == 0 and r_inc == 1, simulating read operations. Like the previous class, it loops 16 times and logs the transaction generation process for each iteration.
- In all three sequence classes, UVM's randomize function is used to generate random values for the sequence item fields. Each transaction is generated, logged, and completed using start_item and finish_item.
- These sequence classes enable testing of both write and read operations in a FIFO interface, providing flexibility in controlling the transaction flow for functional verification in UVM-based testbenches.

9. monitor.sv:

- The monitor class extends uvm_monitor and is responsible for observing the behavior of the FIFO interface during simulation. It monitors signals like w_inc, r_inc, wdata, rdata, and others, creating transaction objects (fifo_seq_item) to capture the observed activity.
- In the new function, the class is initialized, and an info message is logged. The build_phase creates the monitor_port as a new uvm_analysis_port to send the monitored data to other components. It also retrieves the virtual interface (vif) using uvm_config_db to access the FIFO interface signals.
- The run_phase task continuously monitors the FIFO signals in a forever loop. It waits for the reset signals (w_rst and r_rst) to be inactive before checking the w_inc and r_inc flags. If w_inc is set and r_inc is not, it monitors a write

operation. If r_inc is set and w_inc is not, it monitors a read operation. Each monitored transaction is logged with detailed signal information and written to the monitor_port.

- The monitored signals are used to create a transaction (mon_pkt), which holds information such as wdata, rdata, and FIFO flags like wfull and rempty. This transaction is then written to the monitor_port, allowing downstream components (like scoreboards) to process and verify the transactions.
- The class uses standard UVM phases (build, connect, run) and communication mechanisms like uvm_analysis_port to send data. This integration ensures smooth interaction with other UVM components such as the scoreboard, which can verify the monitored data for functional precision.

10. fifo_seq_item.sv:

- The fifo_seq_item class extends uvm_sequence_item and defines the structure for FIFO sequence transactions. It includes various random variables (w_inc, r_inc, w_rst, r_rst, wdata, etc.) to represent the behavior and data of FIFO operations. These fields define the read and write operations, reset states, data being written and read, and the FIFO's status signals (rempty, wfull).
- The class includes several constraints to model valid FIFO behavior. These constraints ensure that the read and write operations are controlled and meet specific conditions. The FIFO cannot perform both read and write operations simultaneously when reset is active.Write operations are only allowed when the FIFO is not full.Read operations are only allowed when the FIFO is not empty.The data written (wdata) must fall within the valid range defined by the DATASIZE parameter.
- The no_rst constraint ensures that both the w_rst (write reset) and r_rst (read reset) signals are inactive for normal operation. This prevents any conflict when reset is applied.

- The new function is the constructor of the sequence item, which calls the base class constructor (super.new(name)) to initialize the sequence item with a given name. The class is registered with the UVM factory via the uvm_object_utils macro, enabling it to be created dynamically during simulation.