VERIFICATION TEST PLAN

ECE-593: Fundamentals of Pre-Silicon Validation
Maseeh College of Engineering and Computer Science
Winter, 2025

Project Name: Verification of Asynchronous FIFO

Members: Satyajit Samhaji Deokar,
Siddesh Dinesh Patil,
Aakash Siddharth Bhupal Hanumanthrao,
Sai Ganesh Reddy Charian

Date: 1/31/2025

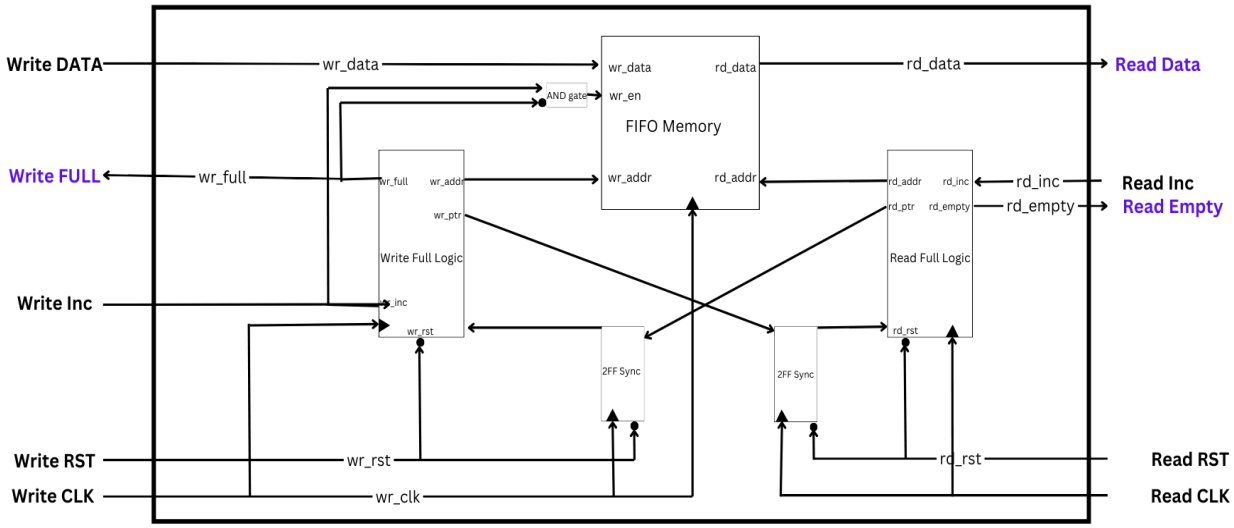
1 Table of Contents

2 Introduction:

2.1 Objective of the verification plan:

- Module Level Verification (FIFO functionality)
- System Level Verification (Integration with larger system)

2.2 Top Level block diagram



2.3 Specifications for the design

Design Features	
1	Theoretically unbounded, but in practice, limited by the simulator and hardware constraints supports data widths upto [2^64 -1] bits; Parameterized by 'DATASIZE'.
2	Theoretically unbounded, but in practice, limited by the simulator and hardware constraints supports memory depths upto [2^64 -1] bits; Parameterized by 'DEPTH'
3	Fully synchronous, and independent clock domains for the Read and Write ports
4	Supports FULL and EMPTY status flags
5	Two optional output signals for indicating Write-Full and Read-Empty of the memory port.

3 Verification Requirements

3.1 Verification Levels

- Module Level Verification (FIFO functionality)
- System Level Verification (Integration with larger system)

3.1.1 What hierarchy level are you verifying and why?

3.1.2 How is the controllability and observability at the level you are verifying?

- Write and read transactions will be controlled via UVM driver.
- FIFO status flags will be observed via UVM monitor and scoreboard.
- Coverage metrics will be collected using functional coverage and assertion-based verification.
- Error injection techniques will be employed to evaluate FIFO behavior under corner-case scenarios.

3.1.3 Are the interfaces and specifications clearly defined at the level you are verifying. List them.

- Write and Read Data Bus
- Clock and Reset signals
- Status flags (Full, Empty, Almost Full, Almost Empty)

4 Required Tools

4.1 List of required software and hardware tool sets needed.

- Simulation Tools: Mentor Questa
- Debugging Tools:Wave, Sim

4.2 Directory structure of your runs, what computer resources you will be using.

- Source Files Directory
- Testbench Directory
- Logs and Reports Directory

5 Risks and Dependencies

5.1 List all the critical threats or any known risks. List contingency and mitigation plans.

- Metastability concerns due to asynchronous clock domains
- Potential deadlocks or race conditions
- Incorrect flag generation causing data loss

6 Functions to be Verified.

6.1 Functions from specification and implementation

- Data write and read operations

- Handling of FIFO full and empty conditions
- Synchronous and asynchronous reset behavior

6.1.1 List of functions that will be verified. Description of each function

- Functions Verified: Carried out extensive checks on fundamental FIFO operations like read/write actions, flag management (empty/full), and boundary condition handling, ensuring reliability across different scenarios.
- Testing Approach: Performed in-depth testing to verify correct functionality under various conditions, including edge cases and concurrent read/write activities, employing detailed testbenches to cover all possible scenarios.

6.1.2 List of functions that will not be verified. Description of each function and why it will not be verified.

Functions Not Verified:

- Non-critical Operations: Debugging, logging, and extra performance features that don't influence the core FIFO functionality.
- Design-specific Features: Optional features or unused control signals that aren't required for the FIFO's operation.
- Reason for Non-verification: These functions don't affect the core FIFO operations and are outside the scope of this project, allowing the focus to remain on verifying the essential FIFO functionality.

6.1.3 List of critical functions and non-critical functions for tapeout

- Critical: Data integrity, flag behavior, metastability handling
- Non-Critical: Performance optimizations

7 Tests and Methods

7.1.1 Testing methods to be used: Black/White/Gray Box.

- Black Box: Validate FIFO as a standalone unit.
- White Box: Check internal register states.
- Gray Box: Combination of both approaches.

7.1.2 State the PROs and CONs for each and why you selected the method for this DUV.

Black Box Testing

- PROs: Assesses FIFO functionality based solely on inputs and outputs, simulating realistic scenarios without needing to understand the internal design.
- CONs: Limited insight into internal states may result in missing certain edge cases tied to the internal logic.
- Reason: Best for validating the FIFO's overall operation, such as ensuring proper read/write handling and correct flag behavior.

White Box Testing

- PROs: Provides visibility into the internal logic, verifying internal states, transitions, and boundary conditions to identify implementation flaws.
- CONs: Requires detailed access to the design and knowledge of its internals, which can be time-consuming.

- Reason: Essential for verifying the accuracy of internal states and flag management, including state transitions and register behavior.

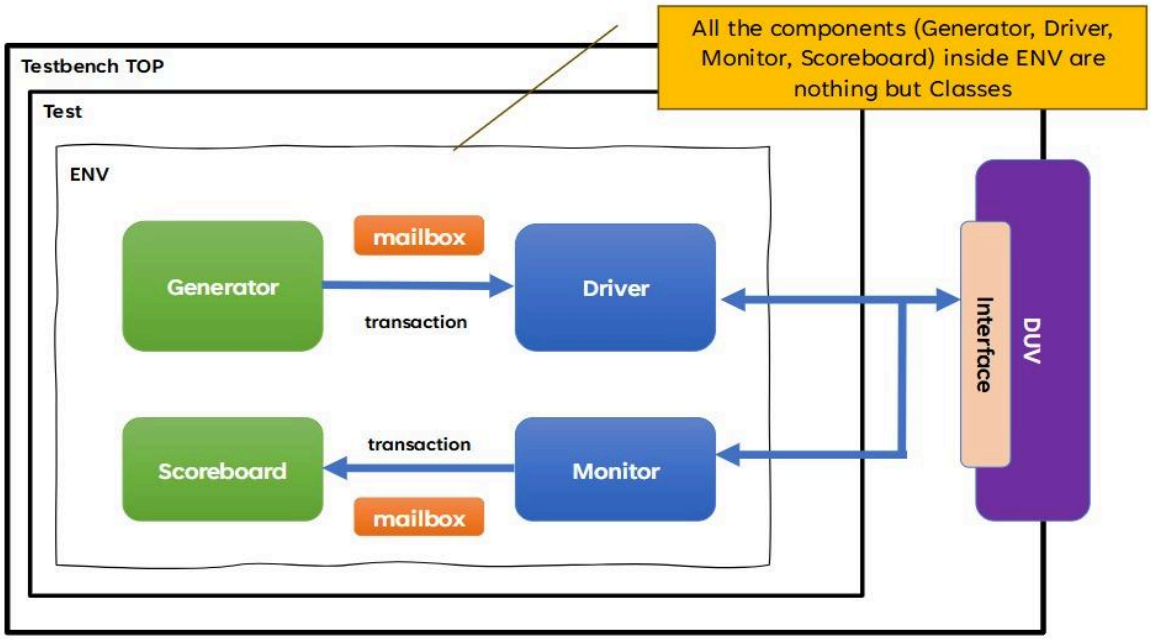
Gray Box Testing

- PROs: Merges the strengths of Black and White Box testing, offering a holistic view of both functionality and internal design.
- CONs: More complex and demanding, requiring a balance of knowledge of internal logic and external functionality.
- Reason: Provides thorough validation by combining both approaches, ensuring comprehensive testing of the FIFO design.

7.1.3 Testbench Architecture; Component used (list and describe Drivers, Monitors, scoreboards, checkers etc.)

- Driver: Generates write/read transactions.
- Monitor: Captures transactions for validation.
- Scoreboard: Compares expected vs. actual outputs.

Include general testbench architecture diagram and how it relates to your design



SystemVerillog testbench architecture

7.1.4 Verification Strategy: (Dynamic Simulation, Formal Simulation, Emulation etc.) Describe why you chose the strategy.

- Dynamic Simulation: Functional correctness verification.
- Formal Verification: Ensure no corner cases are missed.

7.1.5 What is your driving methodology?

- Robust Design Approach: Utilized a modular architecture to enhance FIFO reliability across asynchronous clock domains, focusing on minimizing metastability and ensuring seamless data transfer.
- Effective Synchronization: Employed Gray code pointers and dual-stage synchronizers to facilitate safe and efficient clock domain crossings.
- Test-Driven Verification: Adopted a verification-first strategy, developing SystemVerilog/UVM testbenches to validate functionality, performance, and corner cases early in the design phase.
- Continuous Optimization: Refined FIFO architecture iteratively through performance assessments, CDC verification, and timing analysis to maximize operating frequency and reduce latency.
- Handling Critical Scenarios: Anticipated and resolved key edge cases, including simultaneous read/write operations, near-full and near-empty states, and clock variations, to ensure system robustness.
- Comprehensive Documentation & Review: Maintain thorough design documentation and actively participate in design reviews to align with industry best practices and project timelines.

7.1.5.1 List the test generation methods (Directed test, constrained random)

- Directed Testing: Specific scenarios (reset conditions, flag checks).
- Constrained Random Testing: Randomized read/write sequences.

7.1.6 What will be your checking methodology?

- Synchronization Verification: Ensured the correctness of Gray code pointer transitions and dual-stage synchronizers for reliable clock domain crossings.
- Metastability Assessment: Conducted timing analysis, including setup and hold checks, to minimize the risk of metastability issues.
- Functional Evaluation: Designed SystemVerilog/UVM testbenches to rigorously test FIFO operations, such as read/write accuracy, flag transitions, and reset behaviors.
- Clock Domain Crossing (CDC) Validation: Utilized formal verification and static timing analysis to identify and resolve CDC-related challenges.
- Extreme Condition Testing – Simulated and examined FIFO responses under challenging scenarios like concurrent read/write operations, near-full or near-empty states, and fluctuating clock signals.
- Debugging and Design Review – Conducted comprehensive design reviews, waveform inspections, and debugging processes to enhance reliability and meet project specifications.

7.1.6.1 From specification, from implementation, from context, from architecture etc

- From Specification: Verified that the FIFO meets the required data width, depth, and latency constraints as defined in the project specifications. Ensured compliance with power, speed, and area limitations to align with design requirements.
- From Implementation: Conducted functional simulations using SystemVerilog/UVM to validate read/write operations, flag transitions, and reset behavior. Performed linting and synthesis checks to confirm that the design meets logical and structural requirements.
- From Context: Assessed FIFO behavior when integrated into the larger system, ensuring smooth interaction with upstream and downstream components. Verified clock domain crossing (CDC) mechanisms to prevent data corruption and ensure reliable operation across different clock domains.
- From Architecture: Analyzed the FIFO structure, buffer management, and pointer synchronization to ensure optimal performance and efficiency. Ensured that Gray code-based synchronization techniques were implemented correctly to prevent metastability issues.
- From Timing Analysis: Conducted setup and hold time verification to minimize metastability risks during asynchronous clock transitions. Performed static timing analysis (STA) to check for timing violations and optimize operating frequency.
- From Verification: Used formal verification and constrained-random testing to validate FIFO performance under various scenarios. Examined critical edge cases, including simultaneous read/write operations, near-full and near-empty states, and unexpected clock variations.
- From Debugging and Review: Analyzed simulation waveforms to debug functional issues and ensure correct FIFO operation. Conducted peer design reviews and static code analysis to enhance design robustness and maintain quality standards.

7.1.7 Testcase Scenarios (Matrix)

7.1.7.1 Basic Tests

Test Name / Number	Test Description/ Features
1.1.1 Read	Check basic read operation
1.1.2 Write - Read	Ensure correct data transfer
1.1.3 Reset	Check reset behavior

7.1.7.2 Complex Tests

Test Name / Number	Test Description/ Features
1.2.1 Concurrent Read-Write and Boundary Condition Test	Concurrent events (R+W) Conditions: fifo_full/fifo_empty/always_full/always empty etc.
1.2.2 Simultaneous Read-Write Operation Test	Validate simultaneous read/write
1.2.3 Metastability	Cross-clock domain checks

7.1.7.3 Regression Tests (Must pass every time)

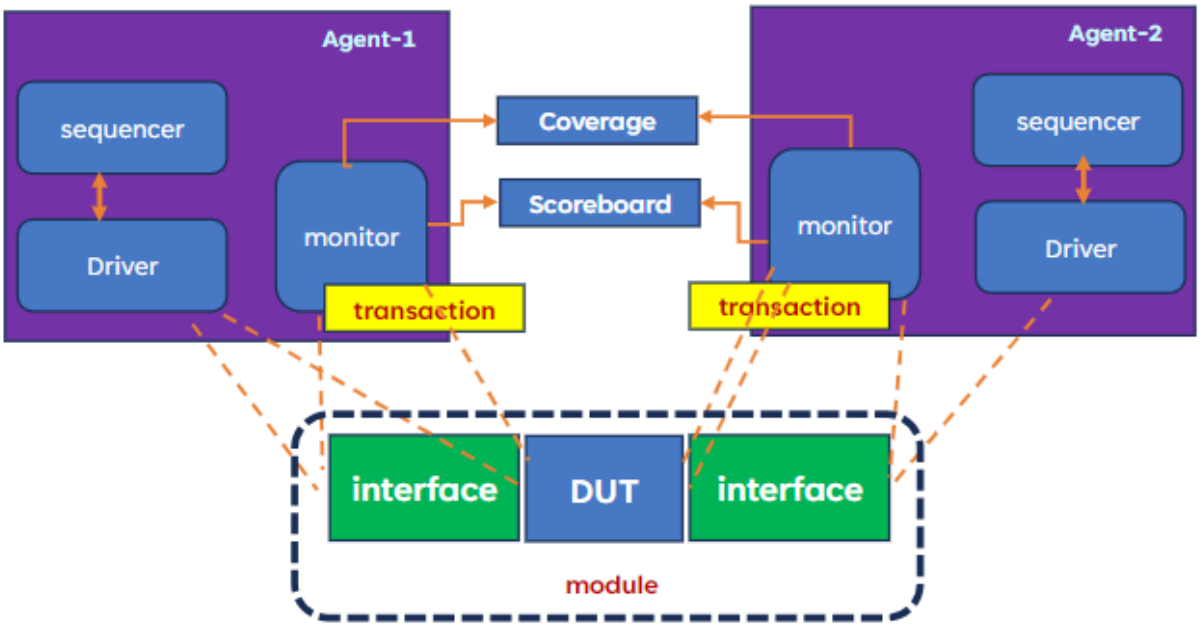
Test Name / Number	Test Description/Features
1.3.1 Baseline Functionality Test	Tests that should always pass
1.3.2 Full/Empty Handling	Ensure FIFO doesn't overflow or underflow

7.1.7.4 Any special or corner cases test cases

Test Name / Number	Test Description
1.4.1 Special Case and Edge Condition Test	Special Case testing tests and conditions
1.4.2 Bug Injection and Fault Tolerance Test	Bug injection and testing scenario

8 UVM Verification Plan

8.1 UVM Architecture



UVM Testbench

8.2 UVM Components

```
UVM_INFO @ 0: reporter [UVMTOP] UVM testbench topology:
----------------------------------------------------------
```

```
Name                Type                    Size  Value
------------------------------------------------------------
uvm_test_top        AFIFO_wr_rd             -     @470
  env               uvm_AFIFO_env           -     @477
    Rd_agent        uvm_AFIFO_Rd_agent      -     @488
    Rd_cov          uvm_AFIFO_Rd_cov        -     @649
    analysis_imp    uvm_analysis_imp        -     @656
    Rd_drvr         uvm_AFIFO_Rd_driver     -     @619
    rsp_port        uvm_analysis_port       -     @634
    seq_item_port   uvm_seq_item_pull_port  -     @626
    Rd_mon          uvm_AFIFO_Rd_monitor    -     @642
    mon_port_cov    uvm_analysis_port       -     @667
    Rd_sqr          uvm_sequencer           -     @510
    rsp_export      uvm_analysis_export     -     @517
    seq_item_export uvm_seq_item_pull_imp   -     @611
    arbitration_queue  array                0     -
    lock_queue      array                   0     -
    num_last_reqs   integral                32    'd1
    num_last_rsps   integral                32    'd1
    Wr_agent        uvm_AFIFO_Wr_agent      -     @495
    afifo_Wr_cov    uvm_AFIFO_Wr_cov        -     @823
    analysis_imp    uvm_analysis_imp        -     @830
    afifo_Wr_drv    uvm_AFIFO_Wr_driver     -     @793
    rsp_port        uvm_analysis_port       -     @808
    seq_item_port   uvm_seq_item_pull_port  -     @800
    afifo_Wr_mon    uvm_AFIFO_Wr_monitor    -     @816
    sb_export_mon   uvm_analysis_port       -     @840
    afifo_Wr_seqr   uvm_sequencer           -     @684
    rsp_export      uvm_analysis_export     -     @691
    seq_item_export uvm_seq_item_pull_imp   -     @785
    arbitration_queue  array                0     -
    lock_queue      array                   0     -
    num_last_reqs   integral                32    'd1
    num_last_rsps   integral                32    'd1
    scoreboard      uvm_AFIFO_scoreboard    -     @502
    imp_rd          uvm_analysis_imp_rd     -     @862
    imp_wr          uvm_analysis_imp_wr     -     @854
```

```
--------------------------------------------------------------
# ----------------------------------------------------------------
# Name                          Type                      Size  Value
# ----------------------------------------------------------------
# uvm_test_top                  AFIFO_wr_rd               -     @470
#   env                         uvm_AFIFO_env             -     @477
#     Rd_agent                  uvm_AFIFO_Rd_agent        -     @488
#       Rd_cov                  uvm_AFIFO_Rd_cov          -     @649
#         analysis_imp          uvm_analysis_imp          -     @656
#       Rd_drvr                 uvm_AFIFO_Rd_driver       -     @619
#         rsp_port              uvm_analysis_port         -     @634
#         seq_item_port         uvm_seq_item_pull_port    -     @626
#       Rd_mon                  uvm_AFIFO_Rd_monitor      -     @642
#         mon_port_cov          uvm_analysis_port         -     @667
#       Rd_sqr                  uvm_sequencer             -     @510
#         rsp_export            uvm_analysis_export       -     @517
#         seq_item_export       uvm_seq_item_pull_imp     -     @611
#         arbitration_queue     array                     0     -
#         lock_queue            array                     0     -
#         num_last_reqs         integral                  32    'd1
#         num_last_rsps         integral                  32    'd1
#     Wr_agent                  uvm_AFIFO_Wr_agent        -     @495
#       afifo_Wr_cov            uvm_AFIFO_Wr_cov          -     @823
#         analysis_imp          uvm_analysis_imp          -     @830
#       afifo_Wr_drv            uvm_AFIFO_Wr_driver       -     @793
#         rsp_port              uvm_analysis_port         -     @808
#         seq_item_port         uvm_seq_item_pull_port    -     @800
#       afifo_Wr_mon            uvm_AFIFO_Wr_monitor      -     @816
#         sb_export_mon         uvm_analysis_port         -     @840
#       afifo_Wr_seqr           uvm_sequencer             -     @684
#         rsp_export            uvm_analysis_export       -     @691
#         seq_item_export       uvm_seq_item_pull_imp     -     @785
#         arbitration_queue     array                     0     -
#         lock_queue            array                     0     -
#         num_last_reqs         integral                  32    'd1
#         num_last_rsps         integral                  32    'd1
#     scoreboard                uvm_AFIFO_scoreboard      -     @502
#       imp_rd                  uvm_analysis_imp_rd       -     @862
#       imp_wr                  uvm_analysis_imp_wr       -     @854
#       rd_cov                  uvm_AFIFO_Rd_cov          -     @870
#         analysis_imp          uvm_analysis_imp          -     @877
#       wr_cov                  uvm_AFIFO_Wr_cov          -     @885
#         analysis_imp          uvm_analysis_imp          -     @892
# ----------------------------------------------------------------
#
```

In UVM-based verification, components are fundamental parts of the testbench that perform specific functions to ensure thorough testing of the DUT (Design Under Test). These include drivers, monitors, sequencers, scoreboards, and agents, all of which interact to generate stimulus, observe responses, and verify the DUT's behavior under various conditions. Each component plays a unique role within the testbench structure: drivers translate high-level transactions into signals for the DUT, while monitors capture and record the DUT's outputs for validation. The sequencer manages the sequence of stimulus, and the scoreboard compares the DUT's output to expected values, ensuring its correctness.

For chip verification, these components are instantiated and interconnected within UVM's hierarchical structure. The test class sits at the top level, organizing and instantiating the necessary components and ensuring proper communication among them. The environment class contains the agents and other essential modules, such as the scoreboard and coverage components. In the build phase, components are created and configured using factory registration methods, while the connect phase establishes the necessary connections. During the run phase, the components collaborate to generate stimulus, monitor responses, and validate the DUT's functionality, ensuring a comprehensive verification of the chip's performance in a range of scenarios.

8.2.1 Sequence Item

A sequence item is a defined collection of input signals extracted from the DUT, which are randomized according to design specifications to generate diverse test cases. These inputs are first specified within the sequence item and then undergo randomization in the sequence to enhance verification coverage. After randomization, the sequence creates transactions and passes them to the sequencer, which serves as a bridge between the sequence and the driver. The sequencer then transfers these transactions to the driver, which converts them into low-level signals for the DUT, ensuring a thorough and effective test stimulus.

8.2.2 Sequence

A sequence in UVM is a structured collection of sequence items and control logic that defines a specific verification scenario. It is responsible for generating stimulus, which is sent to the DUT, and monitoring its response to ensure the design functions correctly. Sequences are designed to be hierarchical and modular, making them reusable and scalable for different test scenarios. By defining different sequences, we can create various verification conditions and improve test coverage.

In our UVM-based testbench, we implemented two distinct sequences to handle different operations based on read and write enable signals. The fifo_sequence_wr manages write operations, while the sequence_fifo_rd handles read operations, ensuring a comprehensive test of FIFO functionality. These sequences help validate the design by covering multiple scenarios, verifying data integrity, and detecting potential corner cases.

By leveraging modular sequences, we improve the efficiency of the verification environment, making it adaptable to future modifications. The structured approach of using separate sequences for different operations allows for better debugging, increased coverage analysis, and more effective functional verification, ultimately leading to a more reliable and robust DUT.

8.2.3 Sequencer

The sequencer in UVM is responsible for managing the generation and sequencing of stimulus transactions. It acts as an intermediary between the testbench and the driver, ensuring that the appropriate sequences are selected and executed based on the test requirements. The sequencer plays a crucial role in coordinating stimulus generation, ensuring that the test scenarios are executed in a controlled and systematic manner. It facilitates transaction-based communication, allowing for better control over randomized and constrained stimulus generation, improving verification coverage and efficiency.

The sequencer communicates with the driver, transmitting generated stimulus to the DUT while receiving responses from the monitor for further processing. To establish this communication, UVM phases such as the build phase and connect phase are utilized. In the build phase, the sequencer is instantiated and configured, while in the connect phase, it establishes a link with the driver to ensure seamless transaction flow. This structured approach ensures that test sequences are properly executed and that the DUT receives the required stimulus efficiently, contributing to a well-organized and robust verification environment.

8.2.4 Driver

The driver in UVM is tasked with converting transaction-level stimulus generated by the sequencer into specific signals or transactions that can be transmitted to the DUT. It acts as the bridge between high-level transaction generation and the low-level signal driving required for DUT stimulation. The driver ensures that the correct interface signals are driven to the DUT, facilitating the accurate testing of the design.

The driver's operation is structured across different UVM phases: the Build Phase, Connect Phase, and Run Phase. During the Build Phase, the driver is configured and prepared, establishing connections with the DUT. In the Connect Phase, it ensures the proper communication links are set up. Finally, in the Run Phase, the driver takes control by actively driving the interface signals, transmitting the stimulus to the DUT for functional verification. This phase-driven approach ensures seamless integration and effective operation of the driver within the testbench.

8.2.5 Agent

An agent in UVM represents a logical entity responsible for interacting with a specific part of the DUT. It typically comprises a driver, monitor, sequencer, and sometimes a scoreboard. Agents abstract the complexities of the interface protocol, offering a clean interface for generating stimulus and monitoring responses, which helps streamline the verification process.

In the Build Phase, a constructor is defined for the agent class that calls the create() method for the sequencer, driver, and monitor, utilizing the factory registration methods from the uvm_object class. During the Connect Phase, the agent establishes the necessary connections, such as linking the driver's port to the sequencer export, ensuring proper data flow. The Run Phase then activates the agent's components to generate stimulus and monitor the DUT's responses. This structured approach ensures modularity and reusability, ultimately making the testbench more efficient and flexible for different verification scenarios.

8.2.6 Monitor

The monitor in UVM is responsible for observing the activity on the DUT's interfaces or any signals of interest. It records the events or transactions that occur on designated interfaces, acting as a bridge between the DUT and the testbench environment. By continuously gathering input and output data from the DUT, the monitor transforms this information into transaction-level data, which can then be used by other components in the testbench for further processing and verification. This ensures accurate and efficient monitoring of the DUT’s behavior during testing.

During the Build Phase, the monitor establishes a connection with the DUT's interface using uvm_config_db, ensuring proper configuration. If the connection fails, it prompts an error to alert the user. In the Connect Phase, the monitor connects to other components of the testbench, such as the sequencer or driver, to ensure proper interaction. During the Run Phase, it continuously collects signals from the DUT via the interface and converts them into transactions, which are then passed on to other components for further analysis. This multi-phase approach allows the monitor to effectively track and record the DUT's activity throughout the verification process.

8.2.7 ScoreBoard

The scoreboard in UVM is responsible for ensuring the DUT behaves correctly by comparing its output with the expected results generated by the testbench. It serves as a reference model to validate the DUT’s performance. The monitor sends transaction-level data to the scoreboard, which then checks this data against the predicted results to confirm the DUT's correctness.

The scoreboard uses a reference FIFO queue, named trans, to store the incoming data. The data_in from the seq_item is pushed into the queue and later popped into a temporary variable called temp_data. This temporary data is then compared with the actual data_out from the DUT. A write method adds values to the queue, while the read method retrieves and compares the data with the actual output from the DUT.

An analysis port in the scoreboard is connected to different components within the UVM testbench hierarchy. During the Build Phase, the scoreboard object is created, and in the Connect Phase, it links to other components. The Run Phase focuses on comparing the expected values with the actual results during read and write operations, ensuring the DUT operates as intended. This process provides critical feedback, supporting effective debugging and verification throughout the testbench.

8.2.8 Environment

The environment in UVM acts as the main container for organizing and managing all the verification components. It houses essential modules like agents, drivers, monitors, sequencers, scoreboards, and other necessary elements. The environment is responsible for creating, configuring, and ensuring proper communication between these components to facilitate effective verification.

Similar to the agent, the environment follows a structured approach but goes further by calling the create method for the agent, scoreboard, and coverage components using the factory registration methods in the Build Phase. In the Connect Phase, it establishes connections by linking the monitor's analysis port to both the scoreboard and the coverage analysis port through hierarchical instantiation. This process ensures seamless interaction between components and strengthens the testbench's ability to thoroughly verify the DUT's functionality. The environment's systematic setup and configuration play a crucial role in ensuring accurate and efficient verification across all stages of the process.

8.2.9 Test

The testbench serves as the central entity that encapsulates the entire verification environment and coordinates the execution of tests. It manages the setup, execution, and teardown of tests, while facilitating communication between various components within the testbench. Typically, the testbench is implemented as a subclass of uvm_environment or uvm_component to organize the verification components effectively.

In this scenario, two distinct sequences are used to generate tests for read and write operations separately. The actual process of raising and dropping objections is defined in the Run Phase, marking the beginning of the test execution in the UVM testbench hierarchy. The uvm_test component calls the create() function using the factory override method in the Build Phase, ensuring that the necessary components are instantiated correctly. This structured approach is crucial for verifying the chip's functionality under various conditions, ensuring that both read and write operations are validated effectively, and any issues in the DUT are identified and addressed.

8.2.10 Testbench Top

The test class sits at the highest level of the UVM hierarchy, representing a specific verification scenario or test case designed to validate particular functionalities or features of the DUT. It is typically derived from the uvm_test base class, providing the structure for the overall testbench. This class is responsible for setting up the necessary components and triggering the execution of the verification sequences.

As the top-level component in the UVM testbench hierarchy, the test class does not define detailed phases like other classes. Instead, it focuses on generating reset signals and clocks, as well as instantiating the DUT. The primary function of the test class is to initialize the environment and create the framework for running tests, while delegating the task of stimulus generation and response monitoring to lower-level components. This ensures an organized approach for thorough DUT verification under different conditions.

9 Coverage Requirements

9.1.1.1 Describe Code and Functional Coverage goals for the DUV

- Code Coverage Goals: Ensure that every line of code is executed at least once (statement coverage) and all decision branches are tested (branch coverage).
  - Achieve path coverage by testing all possible execution paths in the design to ensure no paths are left untested.
  - Track signal transitions (toggle coverage) to ensure that signals change states correctly during testing.
- Functional Coverage Goals: Verify that the FIFO design behaves as expected in key scenarios, including read/write operations and flag behaviors such as empty/full conditions.
  - Cover corner cases like simultaneous read/write operations, FIFO overflow/underflow, and ensure that the FIFO operates within performance limits like latency and throughput.

9.1.1.2 Formulate conditions of how you will achieve the goals. Explain the Covergroups and Coverpoints and your selection of bins.

- Directed testing will be used to validate specific edge cases, like full/empty conditions and boundary states, by focusing on targeted functionality.
- Random testing will help achieve broader coverage by simulating unexpected conditions and scenarios to ensure robustness.
- Covergroups will be defined to track critical signals such as write-enable, read-enable, and FIFO full/empty flags during test execution.
- Bins will be used within the covergroups to monitor specific signal values and states, like write attempts during full FIFO conditions and near-empty/near-full states.

9.1.2 Assertions

- State and Flag Validation: Assertions ensure proper FIFO operation by validating state transitions, such as moving from full to empty only after a read operation. They also check that full/empty flags are asserted correctly—full when the FIFO reaches capacity, and empty when it contains no data.
- Error Detection and Consistency: These assertions monitor FIFO behavior during read/write operations, flagging errors when the state or flags are inconsistent. This includes detecting issues like an incorrect empty flag being set when there's still data in the FIFO or unexpected state transitions, improving the reliability of the design.

9.1.2.1 Describe the assertions that you are planning to use and how it will help you improve the overall coverage and functional aspects of the design.

- Functional Assertions: Monitor FIFO empty/full flags to ensure they are correctly asserted when the FIFO is empty or full. Verify that read and write operations happen only under valid conditions (e.g., no read from empty FIFO, no write to full FIFO).
- Boundary Condition Assertions: Detect FIFO overflow or underflow by asserting when invalid operations occur, such as reading when empty or writing when full. Ensure data integrity by asserting that data written to the FIFO is accurately read out.
- Increased Coverage: Assertions enhance both functional and code coverage, ensuring critical conditions like simultaneous operations and boundary states are properly checked during simulation.
- Early Bug Detection: Assertions immediately flag design violations, reducing debugging time by quickly identifying errors during simulation and ensuring design specifications are met.
- Improved Functional Verification: Assertions automate the enforcement of design constraints and specifications, ensuring reliable FIFO operation under all conditions, such as correct flag handling and operational correctness.

Total Coverage By Instance (CC+ FC, filtered view): 77.89%

```
[SCOREBOARD] Functional Coverage at End: 100.00%
[SCOREBOARD] Read Increment Coverage: 100.00%
[SCOREBOARD] FIFO Empty Coverage: 100.00%
[SCOREBOARD] Reset Coverage: 100.00%
[SCOREBOARD] Read Increment vs FIFO Empty Cross Coverage: 100.00%
[SCOREBOARD] Write Increment Coverage: 100.00%
[SCOREBOARD] FIFO Full Coverage: 100.00%
[SCOREBOARD] Reset Coverage: 100.00%
```

UVM REPORT SUMMARY

```
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO :  715
# UVM_WARNING :    0
# UVM_ERROR :    4
# UVM_FATAL :    0
# ** Report counts by id
# [Questa UVM]     2
# [READ_DRIVER]    96
# [READ_MONITOR]    48
# [READ_SEQUENCE]   106
# [RNTST]     1
# [Read Driver]     1
# [Read Monitor]     1
# [Read_Seq]     4
# [SCOREBOARD]   209
# [TEST_DONE]     1
# [UVMTOP]     1
# [WRITE_DRIVER]    20
# [WRITE_SEQUENCE]    20
# [Write_Monitor]   209
```

10 Bug Injection

10.1. Added Bug in FIFO_rd_empty.sv as Bug1_FIFO_rd_empty.sv

a. rd addr is taking 1 address less while accessing mem

```
23      assign rd_addr = rd_bin[ADDR_SIZE-1 : 1]; //give address for accessing mem
```

10.2 Added Bug in FIFO_2FF_Sync.sv as Bug2_FIFO_2FF_Sync.sv.

a. made 2 flip flop synchronizer to 1 flip flop synchronizer

```
15                          {dq1} <= {din};
```

10.2. **Results**

Values were randomized, but still only a single '1d' value was taken as input and output. Which is a bug in the design.

11 Resources requirements

11.1 Team members and who is doing what and expertise.

- Aakash Siddharth - Design debugging and Verification
- Satyajit Deokar - RTL Design and verification
- Siddesh Patil - RTL Design and verification
- Ganesh Reddy - Synchronising in design and Assertions in Verification

12 Schedule

12.1 Create a table with a plan of completion. You can use milestones as a guide to fill this.

Milestone	Expected Completion
Design and Conventional Testbench	Week 0
Class based Verification using SV	Week 1
Running Random Test Cases	Week 2
Adding Functional Coverage	Week 3
Adding Assertions and Running Final Results	Week 4
Class based Verification using UVM	Week 5

13 References Uses / Citations/Acknowledgements

- http://www.sunburst-design.com/papers/CummingsSNUG2002SJ_FIFO1.pdf
- https://vlsiverify.com/verilog/verilog-codes/asynchronous-fifo/

14 Steps to RUN

Enter into **RUN** Directory

1. run.do

   vsim -do run.do
2. Buggy Codes

   vsim -do run_buggyRTL.do