VERIFICATION TEST PLAN

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

Project Name: Asynchronous FIFO Design and Verification using UVM

Members: Uma Mahesh Bestha, Thanmai Neelampalli, Sathwik Bandi, Bhanu Sai Sidhardha Tummala

Date: 01/25/2025

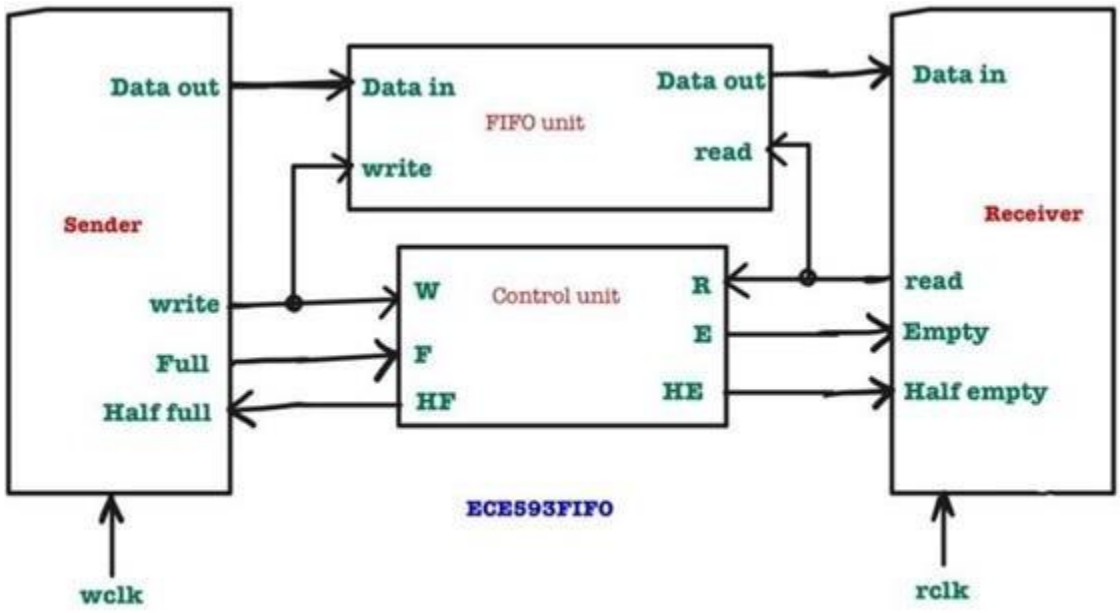
1 Table of Contents

2 Introduction:

2.1 Objective of the verification plan

The objective of the verification plan for the Asynchronous FIFO design is to ensure that the implemented system meets its specified functional and performance requirements. This involves a comprehensive testing strategy that includes functional, performance, corner case, and stress testing to validate the design under various conditions. The plan aims to identify and rectify any discrepancies or issues in the design before deployment, thereby enhancing the reliability and integrity of data transfer across different clock domains. Ultimately, the verification process is critical for confirming that the asynchronous FIFO operates correctly and efficiently within the larger system architecture, as outlined in the document.

2.2 Top Level block diagram



Level Block Diagram of the Design System

3 Verification Requirements

3.1 Verification Levels

Designer-Level Verification:

This involves verifying the functionality of individual modules or components designed by the engineers. It includes unit testing, where each module is tested in isolation using specific test cases to ensure compliance with design specifications. The focus is on functionalities, interfaces, and boundary conditions, typically using simulation tools and sometimes formal verification methods.

Unit-Level Verification:

This level tests the integration of multiple modules within a subsystem or block. It verifies the interactions between modules to ensure they work together as intended, focusing on communication protocols, data flow, and signal integrity.

Sub-Functional Block Verification:

This verification focuses on larger blocks or subsystems composed of multiple units or modules. It ensures that these blocks perform their intended functions and interact correctly with other blocks in the system, identifying integration issues.

System-Level Verification:

This involves testing the entire system or System-on-Chip (SoC) to ensure it meets overall design requirements. It verifies system-level functionalities such as performance, power consumption, and reliability, simulating real-world scenarios and interactions with external components.

Integration and Regression Testing:

Integration testing verifies the seamless operation of various system components, while regression testing involves retesting the system after changes to ensure existing functionalities remain unaffected. These tests help catch integration issues and regressions during the design evolution process.

4 Required Tools

4.1 Required Tools and Software

- Questa Sim: This is the primary simulation tool utilized for the verification process.
- High-Level Language Libraries: System Verilog is mentioned as the language used for the development and verification tasks.

4.2 Directory structure and Computer resources

The directory structure is implemented as follows

Team_11_Async_FIFO

Team_11_Async_FIFO/M1

Team_11_Async_FIFO/M1/CLASS

Team_11_Async_FIFO/M1/UVM

Team_11_Async_FIFO/M1/docs

Team_11_Async_FIFO/M2

Team_11_Async_FIFO/M2/CLASS

Team_11_Async_FIFO/M2/UVM

Team_11_Async_FIFO/M2/docs

Team_11_Async_FIFO/M3

Team_11_Async_FIFO/M3/CLASS

Team_11_Async_FIFO/M3/UVM

Team_11_Async_FIFO/M3/docs

Team_11_Async_FIFO/M4

Team_11_Async_FIFO/M4/CLASS

Team_11_Async_FIFO/M4/UVM

Team_11_Async_FIFO/M4/docs

Team_11_Async_FIFO/M5

Team_11_Async_FIFO/M5/CLASS

Team_11_Async_FIFO/M5/UVM

Team_11_Async_FIFO/M5/docs

5 Risks and Dependencies

5.1 Critical threats

Potential mismatches between sender and receiver specifications: This risk pertains to discrepancies in the expected data formats or protocols between the components communicating through the FIFO.

Clock domain crossing issues: This involves challenges related to data transfer between different clock domains, which can lead to timing issues and data corruption.

Data integrity concerns: Risks associated with ensuring that the data remains accurate and uncorrupted during transfer and processing.

5.2 Contingency and Mitigation plans.

Rigorous testing: Implementing comprehensive testing strategies to identify and rectify issues early in the design process.

Thorough clock domain crossing analysis: Conducting detailed analyses to understand and mitigate the effects of clock domain crossings on data integrity.

Functional verification: Ensuring that all functionalities are verified to meet the design specifications, thereby reducing the likelihood of errors in operation.

6 Functions to be Verified.

6.1 Functions from specification and implementation

The document describes several key functions of the asynchronous FIFO, including:

- **Transferring Data Between Clock Domains:** Ensuring safe data transfer across different clock domains.
- **Handling Full and Empty States:** Logic to manage FIFO states to prevent data loss.
- **Implementation of Gray Code Counters:** Using Gray code for reliable pointer management.
- **Safe Data Synchronization:** Maintaining data integrity during transfers.
- **Scalability and Flexibility:** Ensuring the FIFO can adapt to various configurations.

Functions Not to be Verified:

The rationale for not verifying specific functions typically includes:

- Limited resources or time constraints.
- Functions deemed non-essential for the core operation of the FIFO.

Critical Functions and Non-Critical Functions for Tapeout:

The following functions are critical:

- **Data Transfer Integrity:** Essential for the FIFO's primary purpose.
- **Full and Empty State Management:** Critical to prevent data corruption.
- Non-critical functions may include additional features that enhance performance but are not essential for the FIFO's basic operation.

7 Tests and Methods

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

For verifying the Asynchronous FIFO, the following testing methodologies are considered:

1. Black Box Testing

Definition: Treats the FIFO as a closed system where only inputs and outputs are considered. Internal logic is not examined.

Application for FIFO Verification:

- Functional verification: Checking if FIFO behaves as per specifications (e.g., read/write operations, full/empty conditions).
- Boundary Conditions: Checking FIFO operation near empty/full states.
- Stress Testing: Applying extreme conditions to observe system response.

2. White Box Testing

Definition: Examines the internal structure of the FIFO design, including RTL (Register Transfer Level) implementation.

Application for FIFO Verification:

- Verifies Gray Code Counters for pointer updates.
- Verifying FIFO control logic and state transitions.
- Checking metastability mitigation in synchronizers.

3. Gray Box Testing

Definition: A mix of Black Box and White Box testing, where the test environment has partial knowledge of internal design.

Application for FIFO Verification:

- Checking clock domain crossing (CDC) issues by monitoring synchronization stages.
- Ensures FIFO pointers are synchronized correctly between read and write clocks.
- Debugging unexpected behaviors in simulation while using partial design knowledge.

7.1.2 PROs and CONs of testing methods for DUV.

Method	**Pros**	**Cons**	**Chosen for FIFO Verification?**
Black Box	- Tests functional correctness - Simple to implement - Focuses on I/O behavior	- Cannot detect internal design flaws - Debugging failures is difficult	Yes, for functional validation
White Box	- Allows detection of design flaws in RTL - Verifies all internal states - Debugging is easier	- Time-consuming - Requires deep design knowledge	Yes, for control logic, synchronizers, and gray code verification
Gray Box	- Balances functional and structural testing - Detects hidden bugs related to CDC and pointer updates	- Requires a mix of testbench and RTL analysis	Yes, to verify CDC, pointer transitions, and deep debugging

7.1.3 Testbench Architecture and its Components

Testbench Components:

1. **Driver:**

- Generates random and directed transactions (write/read operations).
- Mimics actual FIFO usage scenarios

2. **Monitor:**

- Observes transactions on both input and output interfaces.
- Captures write/read operations and compares against expected behavior.

3. **Scoreboard:**

- Maintains expected FIFO state for comparison.
- Flags mismatches in expected vs. actual output data.

4. **Checker:**

- Implements assertions to verify FIFO properties (e.g., FIFO follows First-In-First-Out behavior).
- Ensures compliance with full and empty conditions.

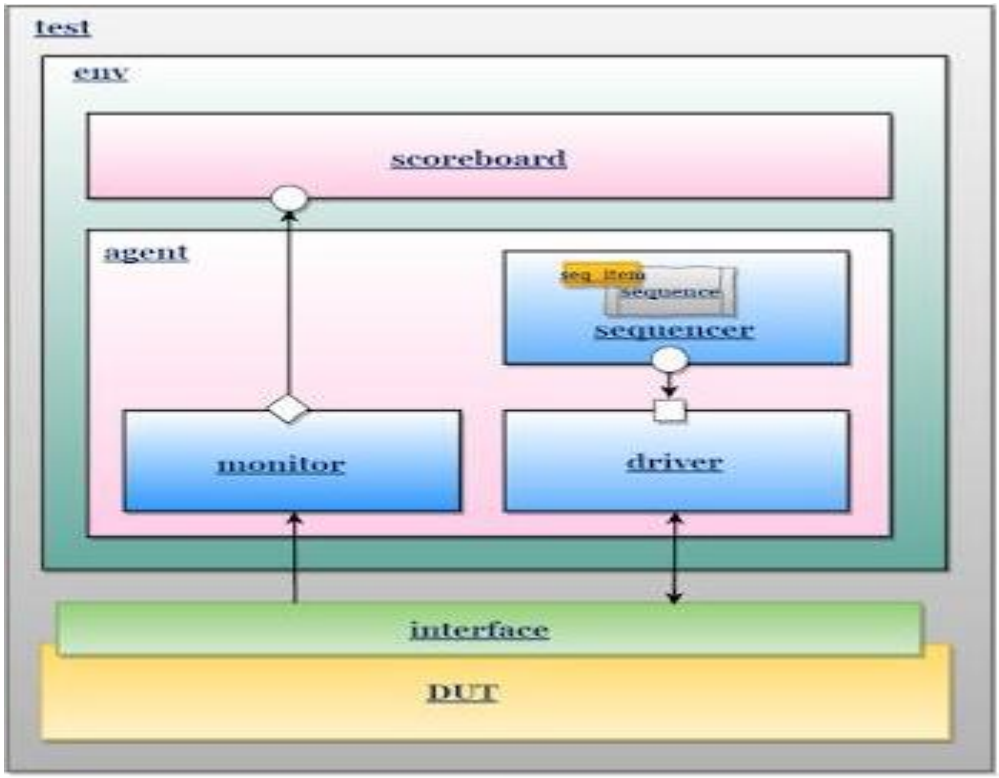
5. **Sequence & Sequencer:**

- Generates various test cases, including random and directed tests.
- Handles corner case testing (e.g., simultaneous read/write, overflow, underflow).

6. **Environment (env):**

- Integrates all UVM components into a structured verification setup.
- Manage multiple test scenarios and test case execution.

Testbench Architecture



7.1.4 Verification Strategy

For the Asynchronous FIFO Verification, the chosen verification strategy is Dynamic Simulation using SystemVerilog UVM. This approach ensures that the design functions correctly across various conditions, particularly handling Clock Domain Crossing (CDC), read/write synchronization, and metastability issues.

Selected Verification Strategy: Dynamic Simulation

Definition:

Dynamic simulation involves applying real-time stimuli to the Design Under Test (DUT) and observing its behavior over time using SystemVerilog and UVM.

Why Dynamic Simulation for FIFO?

- Ensures Functional Correctness
- Verifies FIFO read/write operations, full/empty conditions, and pointer synchronization.
- Detects Timing and Synchronization Issues
- Identifies issues caused by clock domain crossings (CDC).

- Tests Stress and Corner Cases
- Simulates extreme FIFO conditions like continuous write without read, simultaneous read/write operations, and underflow/overflow scenarios.
- Supports Random and Directed Testing
- Uses random stimulus generation to cover unexpected scenarios while allowing specific directed tests for critical cases.
- Debugging with Waveforms and Logs
- Allows visual debugging using waveform analysis (VCD, FSDB) and transaction logs

Key Testing Approaches Under Dynamic Simulation

1. Transaction-Level Modeling (TLM) in UVM

Abstracts away low-level details, making verification efficient.

2. Clock Domain Crossing (CDC) Verification

Ensures that FIFO pointers are synchronized correctly between write clock and read clock domains.

3. Coverage-Driven Verification (CDV)

Uses functional and code coverage to measure verification completeness.

4. Assertion-Based Verification (ABV)

Uses SystemVerilog Assertions (SVA) to check FIFO properties (e.g., FIFO is never read when empty).

7.1.5 Driving Methodology for FIFO Verification

Constrained Random Stimulus Generation

Using constrained random stimulus to generate write/read operations, data sequences, and clock variations dynamically. This helps in covering a wide range of FIFO conditions without manually specifying each scenario.

Directed Test Cases

To ensure coverage of all critical scenarios, the directed test cases are:

- FIFO Full and Empty conditions
- Simultaneous Read and Write operations
- Boundary conditions like FIFO overflow/underflow
- Clock domain crossing synchronization

UVM Sequences & Sequencer-Based Driving

Implementing UVM sequences that interact with the UVM driver to generate different test scenarios efficiently. The UVM sequencer controls transaction flow, ensuring dynamic verification.

Transaction-Level Modeling (TLM)

Using TLM for efficient data handling, reducing low-level signal manipulation, and improving simulation performance. This ensures modular and scalable verification.

7.1.5.1 Test Generation Methods Used

Directed Testing

Creating specific test cases to validate key FIFO functionalities and corner cases, including:

- Basic Read/Write operations – Ensuring data integrity.
- Full & Empty Conditions – Checking proper handling of FIFO full and empty scenarios.
- Boundary Conditions – Testing FIFO behavior when it is almost full or almost empty.
- Clock Domain Crossing (CDC) Scenarios – Ensuring correct synchronization between write and read domains.

Constrained Random Testing

Generating randomized stimuli for:

- Write/Read sequences with varying data lengths and patterns.

- Clock frequency variations between write and read domains.
- Random bursts of reads and writes to simulate unpredictable traffic.

Regression Testing

- Ensuring that all test cases pass consistently after any design changes.
- Running multiple tests with different seeds to improve coverage.

7.1.6 Checking Methodology for FIFO Verification

Scoreboard-Based Checking

- By using a UVM scoreboard to compare expected vs. actual FIFO output.
- The scoreboard tracks all transactions and flags mismatches when data integrity is violated.
- It helps ensure FIFO follows the First-In-First-Out principle.

Assertions-Based Verification (ABV)

- Implementing SystemVerilog Assertions (SVA) to verify:
- FIFO does not read when empty.
- FIFO does not write when full.
- Correct pointer increments and wraparounds.
- Clock Domain Crossing (CDC) synchronization correctness.

Reference Model Comparison

- Using a golden reference model to validate FIFO behavior against a known correct model.
- The testbench fetches outputs from both the FIFO DUT and the reference model, comparing results to detect functional mismatches.

Functional Coverage Analysis

- Tracking functional coverage metrics to ensure that:
- All FIFO states (full, empty, normal, almost full, almost empty) are tested.
- All read/write sequences and burst operations are exercised.

7.1.6.1 Sources for My Checking Methodology

From Specification

- Validating FIFO behavior against the design specification document to ensure:
- FIFO adheres to First-In-First-Out principles.
- Proper handling of full and empty conditions.
- Correct implementation of synchronization across clock domains.

- Proper reset behavior ensuring FIFO starts in a known state.

From Implementation (RTL Design)

- Checking FIFO behavior using the RTL (Register Transfer Level) implementation to confirm that:
- FIFO control signals behave as expected.
- Read/write operations correctly modify internal registers and pointers.
- Gray code pointer synchronization functions correctly to prevent metastability.

From Architecture Context

- Ensuring that the FIFO design integrates correctly within a larger system-on-chip (SoC) environment, checking:
- Compatibility with producer and consumer modules.
- Data integrity when transferring between different clock domains.
- Timing constraints and interaction with external memory subsystems.

From Verification Environment (Testbench & UVM Components)

- Using the UVM scoreboard, reference model, and functional coverage metrics, I am verifying:
- Data integrity across FIFO transactions.
- Detection of incorrect behaviors using assertions (SystemVerilog SVA).
- Coverage-driven validation ensures all functional scenarios are exercised.

7.1.7 Testcase Scenarios (Matrix)

7.1.7.1 Basic Tests

Test Name / Number	Test Description/ Features
1.1.1 Basic Write Operation	Writes a set of data into FIFO and checks if it is correctly stored.
1.1.2 Basic Read Operation	Reads data from FIFO and verifies correctness.
1.1.3 Write & Read Sequentially	Performs write followed by read and checks data integrity.
1.1.4 Reset Functionality	Checks if FIFO clears all data and resets pointers properly.

7.1.7.2 Complex Tests

Test Name / Number	Test Description/ Features
1.2.1 Concurrent Read & Write	Performs simultaneous read and write operations to check synchronization.
1.2.2 FIFO Full Condition Handling 1.2.3 FIFO Empty Condition Handling	Writes continuously until FIFO reaches full state and verifies correct full flag behavior. Reads continuously until FIFO is empty and ensures correct empty flag behavior.
1.2.4 Clock Domain Crossing	Tests FIFO behavior across different read and write clock frequencies.

7.1.7.3 Regression Tests (Must pass every time)

Test Name / Number	Test Description/Features
1.3.1 Basic Regression Test	Runs a combination of read/write operations and checks for any failures.
1.3.2 Randomized Read/Write Sequences	Uses constrained random stimulus to generate various FIFO usage scenarios.
1.3.3 Back-to-Back Read/Write Operations	Continuously writes and reads without pause, ensuring no glitches.

7.1.7.4 Any special or corner cases testcases

Test Name / Number	Test Description
1.4.1 Almost Full & Almost Empty Handling	Ensures FIFO correctly handles near-full and near-empty conditions.

1.4.2 Simultaneous Reset & Operation	Checks if a reset during a read/write operation properly resets FIFO without data corruption.
1.4.3 Bug Injection Test	Intentionally introduces errors (e.g., incorrect pointer updates) to verify error detection mechanisms.
1.4.4 Metastability Condition Testing	Tests FIFO operation under small clock phase differences to evaluate CDC robustness.

8 UVM Verification Plan

8.1 UVM Architecture

The Universal Verification Methodology (UVM) provides a standardized framework for building reusable and scalable verification environments. For the Asynchronous FIFO Verification, UVM will be leveraged to create a robust and structured testbench that ensures comprehensive functional coverage and automation of the verification process.

Key Aspects of UVM in FIFO Verification:

Layered Testbench Architecture: UVM separates stimulus generation, checking, and execution, making the testbench modular and reusable. The environment (uvm_env) encapsulates all necessary components, including agents, scoreboards, and coverage collectors.

Transaction-Based Verification (TLM - Transaction Level Modeling): Stimulus will be generated at the transaction level using uvm_sequence_item, which will represent FIFO read and write operations. The driver (uvm_driver) will convert these transactions into pin-level signals to interact with the FIFO DUT.

Automation & Reusability: Constrained-random stimulus generation will allow verification of a wide range of FIFO behaviors without manually creating every test case. Reusable test sequences will be created using uvm_sequence, making it easier to extend and modify tests.

Scoreboarding & Self-Checking Mechanism: A scoreboard (uvm_scoreboard) will compare expected vs. actual FIFO outputs to verify correctness. Assertions and functional coverage will be used to validate FIFO properties such as First-In-First-Out (FIFO) order, full/empty conditions, and clock domain synchronization.

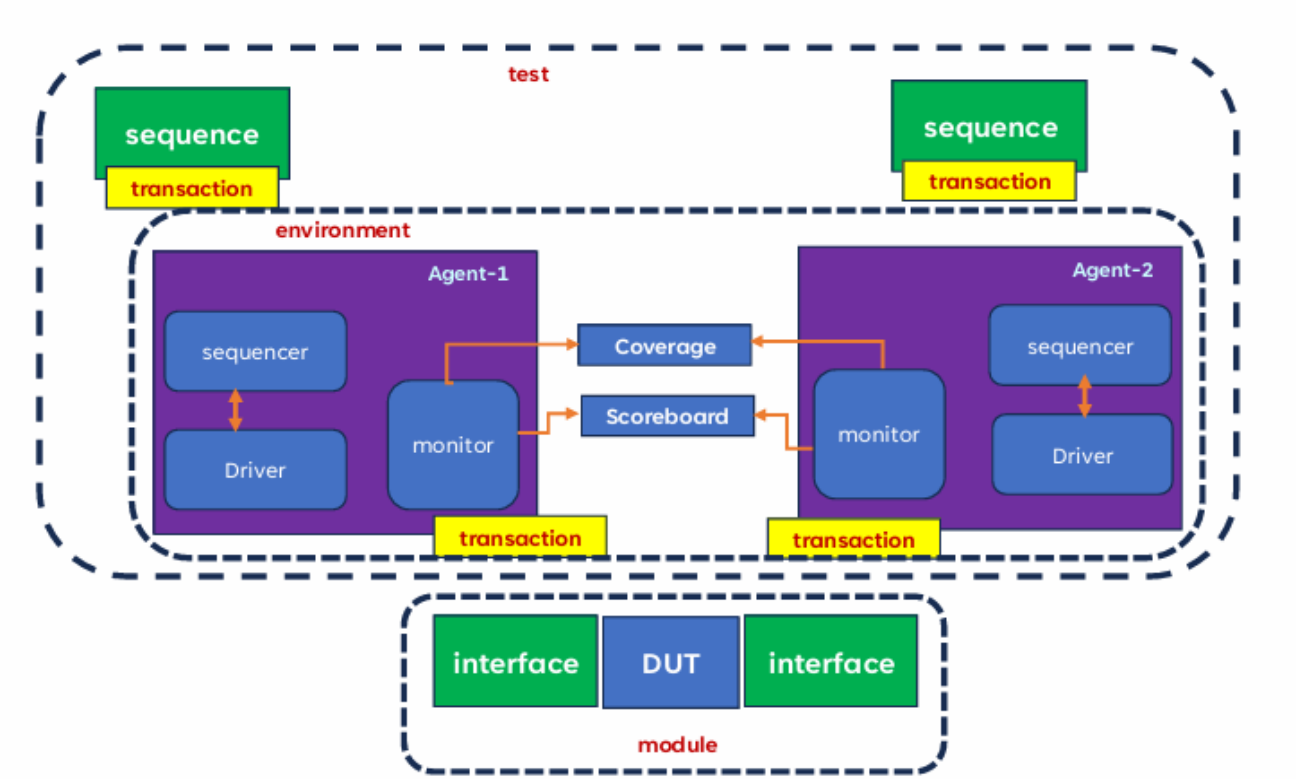
Coverage-Driven Verification: Functional coverage will track whether all FIFO scenarios (full, empty, underflow, overflow, clock domain crossing) have been exercised. Code coverage will ensure all branches of RTL code are tested.

Debugging with UVM Messaging & Logging: uvm_report_* macros will be used to log information, warnings, and errors for debugging test failures efficiently. Detailed logs will help in identifying timing or synchronization issues in the FIFO design.

Regression Testing & Scalability: The UVM environment will be structured to support regression testing, ensuring that any design modifications do not introduce new bugs. The testbench can be reused for different FIFO configurations with minimal changes.

By implementing UVM, FIFO verification will be more efficient, automated, and scalable, reducing manual effort while increasing test coverage and reliability.

8.2 UVM Components



UVM Hierarchy

Description of the hierarchical structure of the UVM testbench, including:

- uvm_test
- uvm_env
- uvm_agent
- uvm_driver
- uvm_monitor
- uvm_scoreboard
- uvm_sequencer

1. UVM Test (uvm_test)

Acts as the top-level test control mechanism. Instantiates the UVM environment (uvm_env). It defines different test scenarios (e.g., normal operation, boundary conditions, stress tests).

Responsibilities:

- Configures the FIFO testbench with various parameters (FIFO depth, clock frequencies).
- Calls sequences to generate constrained-random and directed test scenarios.
- Controls verbosity levels and logging for debugging.

2. UVM Environment (uvm_env)

Acts as the container for all verification components. Instantiates and connects agents, scoreboards, functional coverage, and monitors.

Responsibilities:

- Provides a structured framework for data flow between components.
- Ensures reusability by separating test logic from stimulus generation.
- Configures agents and passes parameters throughout the testbench.

3. UVM Agent (uvm_agent)

Groups Driver, Monitor, and Sequencer together. Represents a verification interface that interacts with the FIFO DUT.

At least two agents are required in FIFO verification: Write Agent (drives write operations), Read Agent (reads FIFO output).

Responsibilities:

- Coordinates transaction flow between sequencer, driver, and monitor.
- Enables reuse by supporting multiple configurations of FIFO testing.
- Can be set as active (drives transactions) or passive (only monitors transactions).

4. UVM Driver (uvm_driver)

Converts high-level transactions (from the sequencer) into pin-level signal interactions for the DUT.

Drives signals such as data_in, write_en, read_en, and clk.

Responsibilities:

- Generates stimulus for write and read operations in FIFO.
- Ensures transactions are correctly applied to DUT inputs.
- Handles protocol timings to prevent race conditions in CDC scenarios.

5. UVM Sequencer (uvm_sequencer)

Controls the flow of transactions between the test and driver. Generates random and directed test sequences.

Responsibilities:

- Sends sequences of write and read transactions to the driver.
- Supports constrained-random testing for coverage improvement.
- Allows easy modification of test scenarios by defining different sequences.

6. UVM Monitor (uvm_monitor)

Passively observes DUT input and output signals without modifying behavior. Captures all transactions happening in the FIFO.

Responsibilities:

- Collects transaction data and forwards it to the scoreboard for comparison.
- Ensures FIFO operations follow First-In-First-Out (FIFO) order.
- Helps debug unexpected behavior by logging observed transactions.

7. UVM Scoreboard (uvm_scoreboard)

Validates DUT behavior by comparing actual vs. expected results. Detects data corruption, ordering errors, overflow, and underflow conditions.

Responsibilities:

- Maintains a reference model of expected FIFO behavior.
- Compares read data from FIFO output with expected values.
- Logs mismatches and flags errors in FIFO operations.

8. UVM Transaction (uvm_sequence_item)

Represents a single FIFO operation (e.g., a write or read transaction). Defines the structure of test data being applied to the FIFO.

Responsibilities:

- Contains fields like data, write_enable, read_enable, and clk.
- Allows randomization of data patterns for enhanced verification coverage.

9. UVM Functional Coverage (uvm_subscriber)

Tracks whether all FIFO states (full, empty, almost full, almost empty) have been tested. Ensures verification completeness by collecting coverage metrics.

Responsibilities:

- Uses covergroups to track various FIFO conditions.
- Reports missing scenarios to enhance test plan effectiveness.
- Helps achieve 100% functional coverage of the FIFO.

10. UVM Messaging & Logging (uvm_report)

Provides debugging mechanisms using log messages. Helps identify failures and track transaction execution.

Responsibilities:

- Uses different verbosity levels (UVM_LOW, UVM_MEDIUM, UVM_HIGH) to control debug output.
- Logs test progress, unexpected conditions, and error reports.

UVM Architecture for FIFO Verification

- Modular & Reusable: Components can be reused for different FIFO designs and configurations.
- Scalable: Supports various FIFO depths, clocking schemes, and transaction patterns.
- Automated & Efficient: Randomization, self-checking, and coverage collection improve test quality.
- Thorough Debugging: Logging and assertions help detect errors early.

8.3 UVM Messaging and Logging

A structured messaging and logging approach is critical in **FIFO verification using UVM**. UVM provides a built-in reporting mechanism that helps track test execution, debug failures, and analyze verification progress. This section details how **UVM messaging** will be implemented in our FIFO verification plan and outlines an efficient **debugging strategy** using verbosity levels.

8.3.1 Logging Verification Progress with uvm_report_* Macros

UVM provides the uvm_report_* macros, which help log test progress, identify warnings, and report failures. These macros categorize messages based on severity, ensuring an efficient debugging process.

Categories of UVM Messages Used in FIFO Verification

Macro	Purpose	Example Usage in FIFO Verification
uvm_report_info	Logs normal test execution and progress updates.	"FIFO Test Started" "Write Transaction Initiated"
uvm_report_warning	Reports potential issues that do not halt simulation.	"Read Attempt on Empty FIFO" "Clock Frequency Mismatch"
uvm_report_error	Highlights functional mismatches or unexpected behaviors.	"FIFO Data Mismatch in Scoreboard"
uvm_report_fatal	Stops simulation on critical failures.	"FIFO Pointer Corruption Detected - Stopping Test!"

8.3.4 Practical Debugging Flow in FIFO Testbench

A structured debugging flow is crucial for efficiently identifying and resolving issues in the FIFO verification process. Since the FIFO operates under various conditions, including clock domain crossing, overflow, underflow, and data integrity checks, an effective debugging strategy ensures that failures are captured, analyzed, and resolved systematically. This section explains how UVM messaging and verbosity levels are used at different stages of the testbench to ensure smooth debugging and verification completeness.

Step 1: Logging Test Initialization and Configuration

Before executing transactions, it is essential to verify that the FIFO testbench environment is set up correctly. The initialization phase includes:

- Setting up the FIFO depth, clock frequencies, and reset conditions.
- Ensuring all UVM components (agents, monitors, drivers, and scoreboard) are instantiated correctly.
- Confirming that testbench parameters match the design specification to avoid unexpected failures later.

Without proper initialization logging, misconfigured parameters (e.g., incorrect FIFO depth or clock settings) may go unnoticed, leading to false failures in the simulation. Proper logging at this stage ensures that any setup issues are detected before running test sequences.

Step 2: Capturing Transaction-Level Activity

Once the test is running, tracking write and read operations is crucial. The driver initiates stimulus generation, and the monitor observes the actual DUT response. Key debugging actions in this phase include:

- Logging each write transaction to confirm that data is correctly applied to FIFO inputs.
- Capturing read transactions to verify that FIFO outputs data in the correct order.
- Ensuring that the write and read operations respect FIFO's full and empty conditions.

Without tracking transaction activity, debugging issues such as data corruption, missing transactions, or incorrect FIFO ordering becomes challenging. A structured log at this level allows for quick identification of where the issue originated—either at the input (driver) or the output (monitor).

Step 3: Detecting Errors and Unexpected Behavior

While normal operation logs provide visibility into test execution, functional errors and warnings are more critical in debugging failures. The scoreboard plays an essential role in:

- Detecting data mismatches between expected and actual FIFO outputs.
- Flagging unexpected read attempts when FIFO is empty.
- Reporting write violations if data is written when FIFO is full.

For example, a FIFO underflow error occurs when a read operation is attempted on an empty FIFO, leading to unpredictable output. Similarly, an overflow error happens if data is written when FIFO is full, causing data loss. Without structured logging, diagnosing such issues becomes difficult, as errors may only be noticeable in waveform analysis rather than in simulation logs.

Step 4: Debugging Timing and Synchronization Issues

Since the FIFO operates across multiple clock domains, clock domain crossing (CDC) errors are one of the most challenging aspects to debug. These errors can cause metastability, data corruption, or incorrect pointer updates. A structured debugging approach involves:

- Logging write and read clock frequencies to ensure they match testbench configurations.
- Tracking FIFO pointers (write and read) to confirm that they increment correctly.
- Verifying that data synchronization occurs without glitches when transferring between clock domains.

If CDC issues go unnoticed, the FIFO might appear to function correctly under certain conditions but fail intermittently, making it difficult to reproduce errors. Logging pointer values and synchronization stages at high verbosity levels helps detect such hidden failures early.

Step 5: Logging Test Completion and Final Results

After executing test sequences, a final report should summarize whether the FIFO operated as expected. A structured summary ensures:

- Successful test completion is logged when all operations match expected results.
- Fail conditions are clearly recorded with detailed reasons (e.g., FIFO output mismatch, unexpected read operations, incorrect reset behavior).
- Any fatal errors trigger a testbench halt, preventing unnecessary simulation runtime when a critical failure occurs.

Without a final summary, debugging across multiple test iterations becomes time consuming, as engineers must manually sift through large logs. A well-structured test completion log ensures that failures are identified immediately, reducing debugging effort.

Debugging strategy using UVM messaging and verbosity levels.

Verbosity Level	Purpose	Recommended Usage
UVM_NONE	Disables logging.	Used for silent execution.
UVM_LOW	Logs essential messages like test start/end.	Regression runs.
UVM_MEDIUM	Displays key debug information like transactions.	Functional validation.
UVM_HIGH	Provides detailed logs of each component interaction.	Debugging failing tests.
UVM_FULL	Prints all internal UVM details, useful for deep debugging.	Rarely used, exhaustive debugging.

8.4 UVM Test Scenarios

The UVM testbench will support both directed and constrained-random testcases to ensure thorough verification of the FIFO design. Below is a categorized list of UVM-based testcases, aligned with the test scenarios that were written previously.

1. **Basic Functionality Tests**

   These tests validate fundamental FIFO operations.

Test Name	**Description**
Basic Write Test	Writes a set of data into the FIFO and verifies if it is stored correctly.
Basic Read Test	Reads data from FIFO and checks if it matches expected values.
Write & Read Test	Writes and then reads data to verify FIFO maintains order (FIFO principle).
Reset Functionality Test	Applies reset in various scenarios (idle, active operation) and verifies that FIFO clears correctly.

2. **FIFO Boundary Condition Tests**

   These tests verify how the FIFO handles full and empty conditions.

Test Name	**Description**
FIFO Full Test	Continuously writes data into FIFO until full and verifies the full condition handling.
FIFO Empty Test	Reads data until FIFO is empty and ensures correct behavior when reading from an empty FIFO.
Almost Full & Almost Empty Test	Verifies FIFO behavior when it is nearly full or nearly empty.

3. **Clock Domain Crossing (CDC) Tests**

   These tests validate FIFO functionality across asynchronous clock domains.

Test Name	**Description**
Variable Write & Read Clock Test	Tests FIFO operation with different write and read clock frequencies.
Metastability Test	Introduces small phase differences in clocks to check CDC robustness.

4. **Stress and Corner Case Tests**

   These tests evaluate FIFO under extreme or unusual conditions.

Test Name	**Description**
Continuous Write Test	Writes indefinitely without reads to check FIFO overflow behavior.
Continuous Read Test	Reads indefinitely without writes to verify empty condition handling.

Back-to-Back Read/Write Test	Performs consecutive read/write transactions to check stability.
Random Burst Write/Read Test	Generates random-sized bursts of writes and reads.
Simultaneous Read & Write Test	Reads and writes in parallel to test FIFO synchronization.

5. Error Handling & Debugging Tests

These tests check how well the FIFO handles unexpected conditions.

Test Name	**Description**
Bug Injection Test	Intentionally introduces errors (e.g., incorrect pointer updates) to test error detection.
Unexpected Reset Test	Applies reset while FIFO is actively operating to check reset recovery.
FIFO Data Corruption Test	Introduces bit-flips in FIFO storage to verify error detection.

6. Coverage & Regression Tests

These tests ensure all functionality is covered and that changes don't break existing features.

Test Name	**Description**
Functional Coverage Test	Measures if all states (full, empty, read, write) are exercised.
Code Coverage Test	Ensures all RTL branches and conditions are executed.
Regression Test Suite	Runs all testcases with multiple seeds to verify consistent behavior.

9 Coverage Requirements

9.1 Code and Functional Coverage goals for the DUV

In a verification plan, code coverage aims to ensure that all lines of code in the Design Under Verification (DUV) are executed during testing. Functional coverage, on the other hand, focuses on verifying that all specified functionalities of the DUV are tested. Goals may include achieving a certain percentage of code coverage (e.g., 100%-line coverage) and ensuring that all functional scenarios, including corner cases, are validated.

9.2 Conditions to achieve goals

To achieve these coverage goals, the following conditions can be formulated:

Covergroups and Coverpoints: Covergroups are used to define specific conditions or scenarios that need to be monitored during simulation. Coverpoints within covergroups specify the variables or expressions to be tracked. The selection of bins within coverpoints allows for categorizing the outcomes of these variables, enabling detailed analysis of coverage.

Selection of Bins: Bins can be selected based on critical operational states of the FIFO, such as full, empty, and various data lengths. This ensures that all relevant scenarios are tested and accounted for in the coverage metrics.

9.3 Assertions

Assertions are statements that check whether certain conditions hold true during simulation. They can be used to validate assumptions about the design and ensure that it behaves as expected. In the context of the Async-FIFO design, assertions may include:

Data Integrity Assertions: To ensure that data written to FIFO is correctly read back without loss or corruption.

State Assertions: To verify that the FIFO correctly transitions between full, empty, and operational states.

Timing Assertions: To check that data is read and written within the expected timing constraints, especially across different clock domains.

Using assertions can significantly improve overall coverage and functional aspects of the design by providing immediate feedback on design behavior, helping to identify issues early in the verification process. This proactive approach enhances the reliability and robustness of the Async-FIFO design.

10 Division of tasks among team members and Schedule

This division of tasks between team members is tentative only. Everyone gets involved in the project in some way or the other. The responsibilities are likely to change as the project progresses and tasks are given.

Milestone-1:

1a. Uma Mahesh Bestha: Complete design specifications document with calculations for DUT.

1b. Thanmai Neelampalli: Verification Plan document in place with initial information. Task division and schedule included.

1c. Sathwik Bandi: Implementation of design and successful compilation.

1d. Bhanu Sai Sidhardha Tummala: Develop a simple conventional testbench to check basic functionality.

Milestone-2:

2a. Sathwik Bandi: Develop Class-Based testbench. All interfaces completed and tested.

2b. Bhanu Sai Sidhardha Tummala: Complete transactions, Generator, Drivers, and other components.

2c. Thanmai Neelampalli: Verify at least 20-50 randomized bursts of data.

2d. Uma Mahesh Bestha: Update Verification Plan with more detailed updates.

Milestone-3:

3a. Uma Mahesh Bestha: Finalize any changes in RTL for any updates needed.

3b. Sathwik Bandi and Uma Mahesh: Complete the class-based verification. All components must be defined and working (Transaction, Generator, Driver, Monitors, Scoreboard, and Coverage).

3c. Thanmai Neelampalli: Include both code-coverage and functional coverage reports.

3d. Bhanu Sai Sidhardha Tummala: Update Verification Plan with more detailed test cases if any more are added.

Milestone-4:

4a. Uma Mahesh Bestha: Develop UVM testbench, starting with UVM TB architecture.

4b. Sathwik Bandi: Add a section for UVM verification Plan and add details on UVMarchitecture, UVM hierarchy, UVM components (sequence, sequencer, driver, monitor, scoreboard, interfaces with DUT, number of agents planned, etc.).

4c. Bhanu Sai Sidhardha Tummala and Thanmai Neelampalli: Utilize UVM_MESSAGING, UVM_LOGGINGmechanisms to create and log the reports and data.

Milestone-5 (Final Deliverables + Presentations):

5a. Sathwik Bandi: Complete the UVM architecture, UVM environment, and UVMtestbench.

5b. Bhanu Sai Sidhardha Tummala: Complete all test cases and reflect them in the coverage reports.

5c. Thanmai Neelampalli: Create scenarios of bug injection and verify.
Show your work.

5d. Uma Mahesh Bestha: Finalize the documents, paper, and presentations. Update Design Specification document, Verification Plan documents with all the updated and the latest data.

11 References Uses / Citations/Acknowledgement

- Prof. Venkatesh Patil lecture slides of ECE 571: Introduction to SystemVerilog
- Prof. Venkatesh Patil lecture slides of ECE-593: Fundamentals of Pre-Silicon Validation
- https://vlsiverify.com/verilog/verilog-codes/asynchronous-fifo/
- XILINX Asynchronous FIFO V3.0 Design specification document

12 Git Hub Link:

https://github.com/Thanmai-02/Pre_Silicon_AsynchronousFIFO_Verification