VERIFICATION TEST PLAN

ECE-593: Fundamentals of Pre-Silicon Validation

Maseeh College of Engineering and Computer Science Winter 2025

Portland State UNIVERSITY

Github link: https://github.com/naidumaheshchowdary/Team_15_Async_FIFO

Team 15:

Alaina Anand Nekuri	(PSU ID: 959604917)
Mahesh Naidu	(PSU ID: 917048048)
Siddhartha Kaushik Gatta	(PSU ID: 946785223)
Venkata Sai Dhilli	(PSU ID: 980087156)

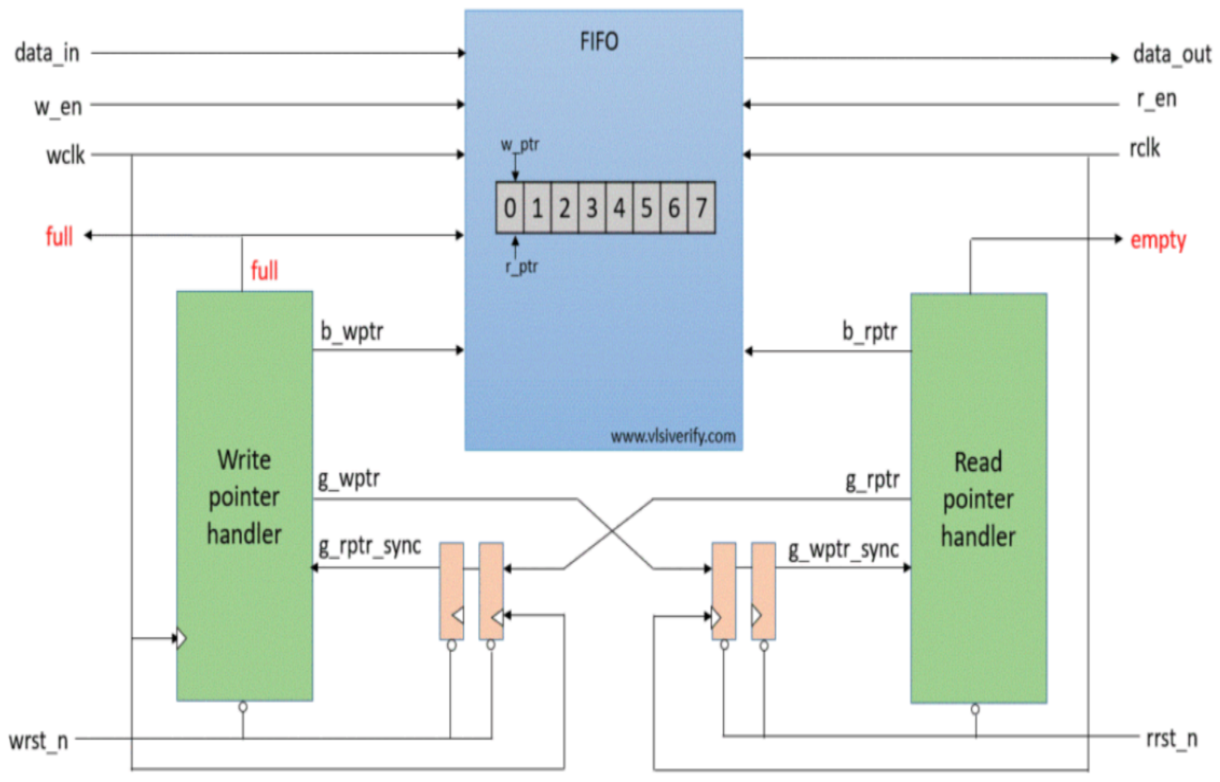
1 Table of Contents

2 Introduction:

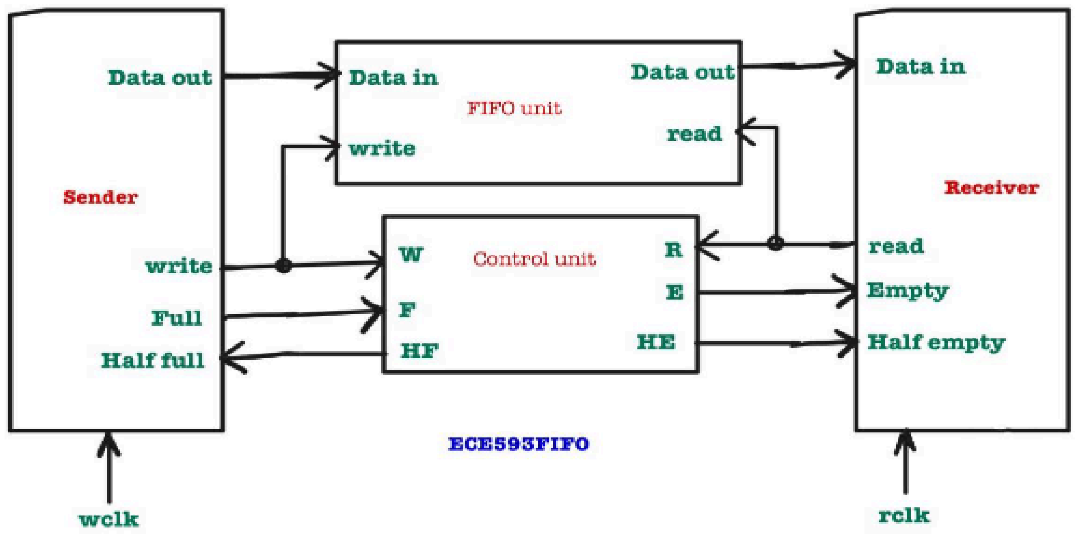
2.1 Objective of the verification plan

The objective of the verification plan is to ensure the Asynchronous FIFO correctly transfers data between independent write 120 MHz and read 50 MHz clock domains while maintaining data integrity. The testbench validates that data written into the FIFO is correctly read in a first-in, first-out (FIFO) manner. It checks that the full flag prevents writes when the FIFO is full and the empty flag blocks reads when empty. The design's ability to handle idle cycles (4 cycles between writes, 2 cycles between reads) is also verified. Random data is generated and stored in a queue expected_data to compare with actual FIFO output rd_data. The simulation runs through multiple iterations, ensuring FIFO functionality under different conditions before termination.

2.2 Top Level block diagram



Asynchronous FIFO



2.3 Specifications for the design

The design consists of a memory buffer, a write pointer, a read pointer, and control logic to handle data movement while preventing overflow and underflow conditions. The write pointer keeps track of the memory location where the next data item will be written, while the read pointer indicates the current location from which data should be read. Since the FIFO operates in two different clock domains i.e, Writing frequency of sender is 120MHz and Read Frequency of receiver is 50MHz, it is crucial to implement a synchronization mechanism to avoid metastability issues.

During initialization or reset, both the read and write pointers are set to zero, indicating that the FIFO is empty. Data is written to the location specified by the write pointer, and after every successful write operation, the pointer is incremented. Similarly, the read pointer moves forward after every read operation. The FIFO remains empty when both pointers are equal, and the empty flag is asserted to prevent unintended reads. When the write pointer wraps around and catches up with the read pointer, the FIFO is considered full, and the full flag is asserted to prevent additional writes.

One of the key challenges in asynchronous FIFO design is distinguishing between the full and empty states, as both occur when the read and write pointers are equal. To address this, an extra bit is added to each pointer, which toggles when the FIFO wraps around. If the extra bit of the write pointer differs from that of the read pointer, it indicates that the FIFO has wrapped around and is full. On the other hand, when both pointers, including the extra bit, are equal, the FIFO is empty.Another critical aspect of the design is the use of Gray-coded pointers for synchronization. When transferring data between different clock domains, binary counters can lead to metastability because multiple bits may change simultaneously. Using Gray code ensures that only one bit changes at a time, reducing the likelihood of metastability and improving the stability of pointer synchronization.

The design also includes logic to handle overflow and underflow conditions. When the FIFO is full, additional write operations are ignored to prevent data corruption. Likewise, when the FIFO is empty, read operations are blocked to avoid retrieving invalid data. The FIFO operates efficiently by ensuring that the read pointer always points to the next valid data, minimizing the number of clock cycles required for data retrieval.

Overall, the design of the Asynchronous FIFO focuses on efficient memory utilization, proper synchronization, and robust pointer management to provide reliable data transfer across different clock domains.

3 Verification Requirements

3.1 Verification Levels

3.1.1 What hierarchy level are you verifying and why?

At the module level, we are verifying the Asynchronous FIFO to ensure its fundamental functionality, including data integrity, pointer updates, flag assertions, and synchronization across clock domains. This level of verification is crucial because the FIFO operates with independent 120 MHz write and 50 MHz read clocks, making robust cross-clock synchronization essential to prevent metastability. The verification focuses on write and read transactions, ensuring that data is correctly stored and retrieved in a first-in, first-out manner while preventing overflow and underflow conditions. Additionally, the full and empty flag assertions are tested to ensure that writes are blocked when the FIFO is full and reads are blocked when empty. Another key aspect is the ability of the FIFO to handle concurrent read and write operations, ensuring smooth operation without data corruption. By verifying at the module level, we gain precise control over individual components, such as the write and read agents, drivers, monitors, scoreboard, and sequence items, ensuring that each part functions correctly before integrating into a larger system. This approach provides a structured and scalable verification methodology, allowing thorough validation of the FIFO's functionality before moving to system-level testing.

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

At the module level, we have full controllability over the FIFO using write and read agents, which generate both constrained-random and directed stimulus to test different scenarios. The drivers apply these transactions, while the testbench allows us to control resets and clock variations. Observability is ensured through monitors that track FIFO operations and a scoreboard that compares expected vs. actual results, flagging mismatches. Additionally, coverage collection and assertions help verify pointer transitions, flag behavior, and clock synchronization. This setup ensures we can thoroughly test and debug the FIFO before system integration.

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

- Yes, the interfaces are clearly defined:

  Inputs:

  - wr_en, wr_clk, wr_reset, wr_data, fifo_full (Write interface).
  - rd_en, rd_clk, rd_reset,rd_data, fifo_emoty (Read interface).

  Outputs:

  - rd_data (read data).
  - rd_en (read enable)
  - fifo_full (full flag).
  - fifo_empty (empty flag).

4 Required Tools

4.1 List of required software and hardware toolsets needed.

The Siemens EDA Questa tool is utilized for SystemVerilog simulation in an object-oriented programming (OOP)-based verification environment.

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

<u>SV CLASS BASED TESTBENCH</u>

Design Files

- ➢ Async_fifo_memory.sv
- ➢ Fifo_top.sv
- ➢ rempty.sv
- ➢ sync_rtow_wtor.sv
- ➢ data_fields.sv
- ➢ Wfull.sv

Testbench Files

- ➢ Generator.sv
- ➢ packet.sv
- ➢ driver.sv
- ➢ Interface.sv
- ➢ monitor.sv
- ➢ Environment.sv
- ➢ Scoreboard.sv
- ➢ test.sv
- ➢ top.sv

Script Files

- ➢ run.do

<u>UVM BASED TESTBENCH</u>

Design Files

- ➢ FIFO_design.sv
- ➢ interface.sv

Testbench Files

- ➢ uvm_coverage.sv
- ➢ uvm_seq_1.sv
- ➢ uvm_driver.sv
- ➢ uvm_environment.sv
- ➢ uvm_test1.sv
- ➢ uvm_scoreboard.sv
- ➢ uvm_seq_item.sv

➢ uvm_sequencer.sv
➢ uvm_write_agent.sv
➢ uvm_write_monitor.sv

Script Files

➢ run.do

5 Risks and Dependencies

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

- Metastability in Clock Domain Crossing → Use Gray-coded pointers and synchronizers.
- FIFO Overflow/Underflow → Implement full/empty flag logic and boundary tests.
- Low Test Coverage → Improve UVM-based verification and functional/code coverage.

6 Functions to be Verified.

6.1 Functions from specification and implementation

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

- Basic Write/Read: Ensure data is written and read correctly.
- Full/Empty Flags: Verify FIFO full and empty conditions.
- Pointer Increment: Check write and read pointer updates.
- Reset Operation: Ensure FIFO initializes correctly.
- Cross-Clock Sync: Validate data transfer between clock domains.
- Concurrent R/W: Test simultaneous read and write operations.

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

- Error Handling: Overflow/underflow checks deferred.
- Power-On Reset: Advanced verification for later.
- Performance Metrics: Throughput/latency not tested yet.

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

<u>Critical Functions (Must be verified before tapeout)</u>

- Read/Write Operations – Ensure correct FIFO data transfer.
- Full/Empty Flag Logic – Prevent overflow/underflow errors.
- Pointer Synchronization – Avoid metastability in clock domains.
- Clock Domain Crossing (CDC) – Ensure reliable data transfer.

<u>Non-Critical Functions (Can be optimized post-tapeout)</u>

- Power Optimization – Low power techniques for efficiency.
- Performance Metrics – Throughput and latency analysis.
- Debug Features – Additional test hooks for post-silicon debug.
- Extended Error Handling – Advanced fault detection mechanisms.

7 Tests and Methods

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

Black Box Testing

Black box testing is used to verify the functional correctness of the FIFO without looking into its internal design. It focuses on input and output behavior, making it simple and effective for functional validation. However, it cannot detect internal design flaws or inefficiencies. This method is applied for testing read/write operations and flag behavior.

White Box Testing

White box testing examines the internal structure of the design, ensuring correct logic implementation. It helps detect design flaws, verifies FSM transitions, and ensures proper pointer synchronization. Since it requires detailed RTL knowledge and a complex setup, it is mainly used for code coverage analysis, CDC verification, and FSM validation.

Gray Box Testing

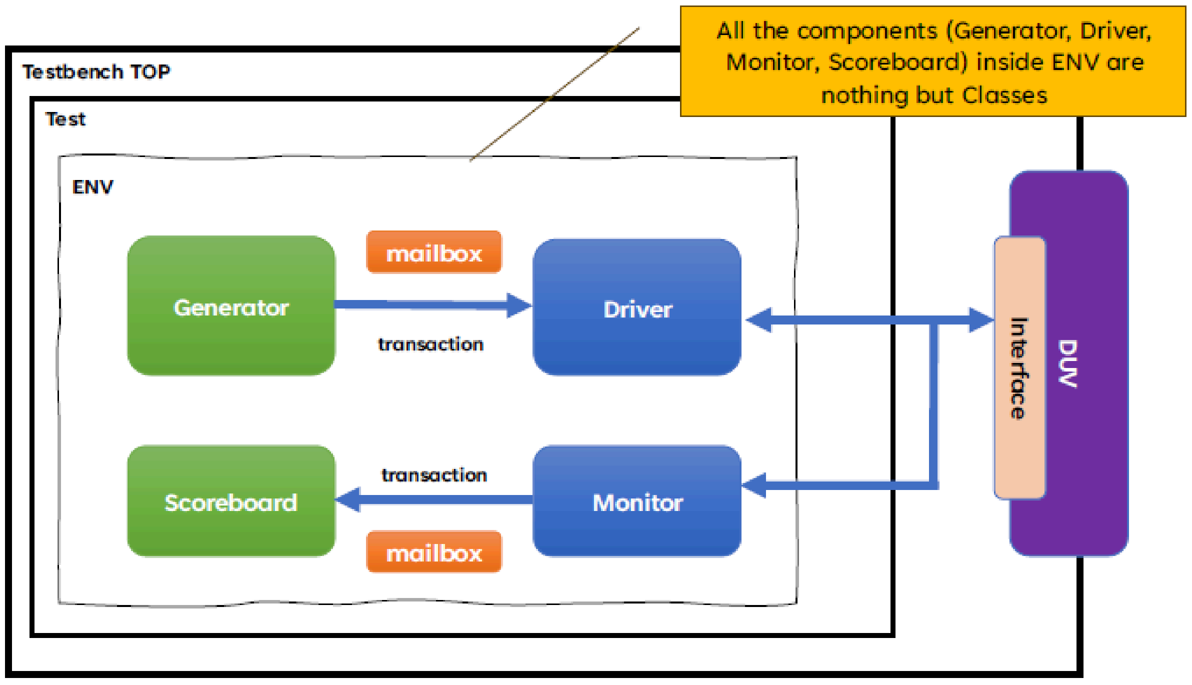
Gray box testing combines black and white box approaches, balancing functional and structural testing. It provides a good mix of validation but may still miss some low-level bugs. This method is implemented through UVM-based verification with assertions, functional coverage, and constrained random stimulus to ensure a comprehensive validation strategy.

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

Black box testing is simple and effective for functional validation as it does not require knowledge of the internal RTL design. However, it is limited in detecting internal design flaws and cannot verify pointer synchronization or FSM behavior. White box testing provides deep insight into the internal structure, allowing detection of FSM transitions, pointer synchronization issues, and CDC behavior. Despite its advantages, it requires extensive RTL knowledge and a complex setup, making it time-consuming.

Gray box testing combines the benefits of both methods by balancing functional and structural verification. It improves coverage while maintaining efficiency, but it may still miss some low-level corner-case bugs.

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



SystemVerillog testbench architecture

In digital design verification, a SystemVerilog testbench is a structured environment that simulates real-world conditions to test a Design Under Test (DUT). It follows a transaction-based verification approach, where stimulus is generated, applied to the DUT, and results are checked automatically. The testbench consists of several components, each performing a specific role. These components interact using mailboxes, which facilitate communication between different testbench blocks. The testbench is designed using Object-Oriented Programming (OOP) principles, making it modular, reusable, and scalable.

Testbench:

At the highest level, the Testbench is the top module that connects the Test and DUT using an Interface. It is responsible for generating the clock and initializing the test environment.

Environment:

The Test component creates the Environment and sets the number of transactions to be generated. The Environment is the heart of the testbench, containing all major verification components: Generator, Driver, Monitor, Scoreboard, and Transaction.

It facilitates the interaction between these components, ensuring proper synchronization and execution of transactions.

Generator:

The Generator is responsible for creating test stimulus by randomizing transaction data. It sends these transactions to the Driver using a Mailbox. Additionally, the Generator includes an event mechanism to signal the completion of packet generation using a triggering mechanism.

Driver:

The Driver receives transactions from the Generator, converts them into signal-level activity, and applies them to the DUT via the Interface. It ensures that the correct signals are driven and tracks the number of packets processed during the simulation.

Monitor:

The Monitor passively observes the DUT's response by sampling interface signals. It converts the captured signals into transaction-level data and forwards them to the Scoreboard via a Mailbox. The Monitor is a passive component and does not drive signals, ensuring a non-intrusive verification methodology.

Scoreboard:

The Scoreboard is responsible for checking the correctness of the DUT's output. It compares the actual response received from the Monitor with the expected results. If there is a mismatch, it reports an error. This component plays a critical role in ensuring functional correctness.

Interface:

The Interface acts as a bridge between the testbench and DUT. It groups multiple signals into a single structured entity, simplifying the connection. A virtual interface allows the same code to be reused for different DUT configurations, making the testbench more flexible and scalable.

Mailbox:

A Mailbox is a SystemVerilog built-in feature that facilitates communication between different testbench components. It temporarily stores transactions in system memory, enabling synchronization between components. This testbench uses two mailboxes:

1. Generator → Driver
2. Monitor → Scoreboard

Mailboxes ensure that transactions are transferred efficiently, maintaining the testbench's modularity

Transaction:

The Transaction class defines the structure of the data used in the testbench. It includes fields required to generate test stimulus, and the rand keyword is used to create randomized inputs. This randomization technique improves test coverage by allowing a variety of test cases to be generated dynamically.

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

The verification of the Asynchronous FIFO is primarily based on dynamic simulation with assertion-based checking. Dynamic simulation ensures functional correctness by applying real-time stimulus and analyzing FIFO behavior under different conditions such as read/write operations, full/empty conditions, and clock domain synchronization. The UVM-based testbench generates constrained-random stimulus, tracks transactions, and validates expected outputs, enabling thorough debugging and waveform analysis.

Assertion-based checking is used to verify pointer synchronization, flag correctness, and CDC stability. These assertions help detect functional violations but do not replace full formal verification.

7.1.5 What is your driving methodology?

The verification of the Asynchronous FIFO follows a UVM-based constrained-random approach combined with directed testing. The UVM sequencer and driver generate random transactions to improve functional coverage, while directed tests validate specific scenarios like full, empty, and corner cases. This methodology ensures comprehensive verification by covering both expected and edge-case behaviors, enhancing the reliability of FIFO operation.

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

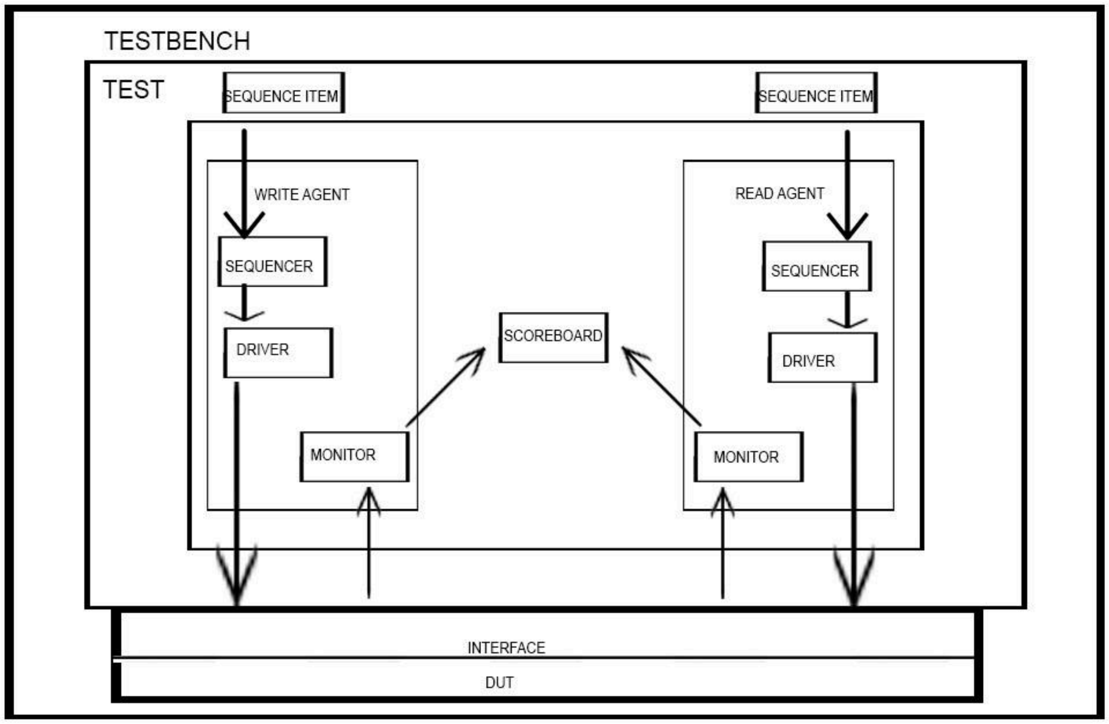
The Asynchronous FIFO verification uses directed testing and constrained-random testing to ensure comprehensive coverage.

Directed Testing is used to validate specific scenarios such as FIFO full, empty, reset behavior, and boundary conditions. These tests help confirm expected functionality and flag behavior under controlled conditions. Constrained-Random Testing generates a variety of test cases by applying randomized input sequences within defined constraints. This method helps uncover corner cases, metastability issues, and unexpected design behaviors while improving functional and code coverage. Both methods are combined to achieve a balanced verification strategy, ensuring that the FIFO works reliably under different operating conditions.

7.1.6 What will be your checking methodology?

The checking methodology for the Asynchronous FIFO involves a UVM scoreboard, assertions, and functional coverage analysis. The scoreboard compares expected and actual FIFO outputs to ensure correctness. Assertions check critical properties such as pointer synchronization, flag behavior, and CDC stability. Functional coverage ensures all key scenarios, including full, empty, and corner cases, are exercised. This approach guarantees thorough validation and detects errors early in the verification process.

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



In my Asynchronous FIFO project, We initially implemented a SystemVerilog class-based testbench for functional verification. However, as the complexity of verification increased, we have transitioned the testbench to Universal Verification Methodology (UVM) to make it more scalable, reusable, and coverage-driven. The UVM provides better framework and provided a structured environment for constrained-random stimulus generation, automated checking mechanisms, and functional coverage collection, ensuring a robust verification strategy for the FIFO design.

The UVM testbench for the FIFO was designed to validate read and write operations, check FIFO full and empty conditions, test clock domain synchronization, and ensure metastability resilience. The architecture followed the standard UVM hierarchy, consisting of multiple layers working together to generate transactions, apply stimuli to the DUT, monitor the responses, and verify the expected behavior against actual results.

Test:

At the highest level, the UVM Test class controlled the overall verification flow. It was responsible for configuring the environment, instantiating sequences, and defining different test scenarios to ensure that the FIFO functioned correctly under various operating conditions. Within the test, the UVM Environment served as the main container, encapsulating multiple UVM Agents, the Scoreboard, and a Coverage Collector. The agents were responsible for handling transactions related to the read and write interfaces of the FIFO.

Agent:

Each UVM Agent contained a Sequencer, Driver, and Monitor. The Sequencer generated sequences of read/write transactions, which were sent to the Driver. The Driver then converted these high-level transactions into pin-level signals that were applied to the FIFO interface, ensuring correct write and read operations.

Monitor

Meanwhile, the Monitor passively observed signals from the DUT and captured transactions, forwarding them to the Scoreboard for validation and the Coverage Collector for functional coverage tracking.

Scoreboard:

The UVM Scoreboard played a critical role in verifying FIFO correctness. It compared expected vs. actual results to detect mismatches, ensuring that the FIFO properly buffered and retrieved data in a first-in, first-out manner. It also validated boundary conditions, such as FIFO full and empty states, ensuring that data was not lost due to incorrect flag assertions. The Coverage Collector measured the effectiveness of the verification by tracking whether all functional scenarios were exercised, ensuring comprehensive validation of the FIFO.

Interface:

The Interface connected the testbench to the FIFO DUT, defining the signals required for communication. The testbench also included a Transaction class, representing FIFO operations such as write, read, and control flag checks. Transactions acted as the basic unit of communication between the Sequencer, Driver, and Monitor, ensuring a structured flow of data within the UVM testbench.

With this UVM-based approach, the FIFO verification became fully automated, reusable, and scalable, achieving 83.33% functional coverage and 83% code coverage. This transition to UVM enabled randomized testing, protocol checking, and automated result validation, making the verification process more robust.

Overall, the UVM-based testbench significantly improved verification efficiency, enabling rigorous testing under diverse scenarios. The structured methodology ensured that the FIFO design was functionally correct, robust against metastability, and met all performance requirements.

7.1.7 Testcase Scenarios (Matrix)

7.1.7.1 Basic Tests

TESTCASE	DESCRIPTION	EXPECTED OUTCOME
Reset Functionality	Verify that when the reset signals are activated, all FIFO internal states, including pointers and status flags, are reset to their default values.	The FIFO should clear all stored data, reset both read and write pointers, and assert the empty flag while deasserting full and half-full indicators.
Basic Write and Read Operations	Ensure that data written to the FIFO is correctly stored and retrieved in the same order when read operations	Each read should return the correspondi -ng previously written data, maintaining the FIFO order. The empty and full

	occur.	flags should update accordingly during operations.
Write pointer	Validate that the write pointer incre -ments only when a write operation occurs and stops when the FIFO reaches full capacity.	The write pointer should advance with each valid write cycle and should halt once the FIFO is full, asserting the full flag.
Read Pointer	Check if the read pointer increments correctly when data is read from the FIFO and stops when no data remains.	The read pointer should advance with each valid read operation and should stop when the FIFO is empty, asserting the empty flag.

8 Coverage Requirements

8.1.1.1 Describe Code and Functional Coverage goals for the DUV

Coverage plays a vital role in ensuring that the Asynchronous FIFO operates correctly across all possible scenarios. It verifies that the design meets its intended specifications and behaves as expected under different conditions. The coverage model is implemented using SystemVerilog covergroups, where each functional aspect is represented as a coverpoint to ensure complete verification. The key functional areas covered are described below.

Functional Coverage

In our verification, the achieved functional coverage is 79.14%, ensuring that the FIFO meets the expected performance criteria. The following critical scenarios are covered:

- Read and Write Transactions: Validates that data is written and retrieved in the correct order, ensuring proper storage and retrieval functionality.
- Boundary Conditions: Tests FIFO behavior when it is full, empty, half-full, and half-empty to verify correct assertion of wfull and rempty flags.
- Clock Domain Crossing: Ensures seamless data transfer across asynchronous write and read clocks, preventing metastability issues.
- Control Signals: Confirms that wr_en and rd_en operate as expected and that status flags (wfull, rempty, half_wfull, half_rempty) activate correctly.
- Reset Behavior: Ensures that applying wrst or rrst resets all pointers and status flags to their default values.
- Back-to-Back Transactions: Verifies FIFO functionality under continuous high-speed read and write operations without data corruption.

By covering these scenarios, the verification strategy guarantees a robust and fully functional FIFO design.

Code Coverage

Initially, code coverage was 46.01%, but with the UVM-based environment, it has now improved to 79.05%, ensuring that the RTL is exercised thoroughly. The breakdown of coverage metrics is as follows:

- Statement Coverage: Ensured 94.44% of all executable lines in the RTL were exercised at least once.

- Branch Coverage: Verified all decision points (if-else, case statements) were tested under both true and false conditions, achieving 94.44% coverage.
- Expressions Coverage: Ensured that all conditional expressions were evaluated, achieving 66.67% coverage.
- Toggle Coverage: Verified that all flip-flops, registers, and storage elements toggled between 0 and 1, achieving 89.94% coverage.

To achieve this, we have implemented constrained-random stimulus generation, introducing variability in test scenarios and covering a broader range of operational conditions. Moving forward, we will further refine our randomization constraints, introduce additional corner-case scenarios, and optimize our test plan to improve coverage in missed expressions and toggle conditions. This approach will help close the remaining coverage gap while maintaining an efficient and scalable verification environment.

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

To achieve high code and functional coverage, a combination of directed and constrained-random testing, assertion-based verification, and coverage-driven stimulus refinement is used.

Code Coverage Approach

- Statement & Branch Coverage: Ensured by running exhaustive test scenarios, covering all RTL conditions.
- FSM Coverage: Verified by triggering all state transitions through directed and random tests.
- Toggle Coverage: Achieved by applying varied stimulus, ensuring all signals toggle at least once.

Functional Coverage Approach

Functional coverage is implemented using **covergroups and coverpoints** in SystemVerilog to track critical design behaviors.

Covergroups and Coverpoints:

- FIFO Full & Empty Conditions: Coverpoints for full, almost full, empty, and almost empty states.
- Pointer Synchronization: Coverpoints for correct read/write pointer alignment across clock domains.
- Reset Behavior: Ensures reset initializes FIFO properly before operations start.
- Concurrent Read/Write: Tracks conditions where read and write operations happen simultaneously.
- Metastability Handling: Covers different timing scenarios of asynchronous clocks.

Selection of Bins:

- Bins are defined for different flag conditions (full, empty, half-full) to ensure complete flag verification.
- Read and write operations are binned based on data depth levels to check FIFO efficiency.
- Bins for corner cases ensure FIFO behaves correctly in extreme conditions.

By covering these scenarios, the verification strategy guarantees a robust and fully functional FIFO design. In our verification, the **achieved functional coverage is 83.33% through UVM Based testbench architecture**, ensuring that the FIFO meets the expected performance criteria.

Code coverage analysis was conducted to evaluate how effectively the FIFO RTL design was exercised during verification. The previous achieved total coverage stands at 46.01%, indicating a significant portion of the design has been tested and now when we updated it to **UVM based environment** we are able to improve the **total code coverage** to **79.05%** and now the code is exercised properly.

8.1.2 Assertions

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.

Assertions are used to validate critical properties of the Asynchronous FIFO, ensuring correct behavior and improving functional coverage. Key assertions focus on pointer synchronization, flag behavior, and protocol checks, helping detect violations early in the verification process.

Assertions ensure that the full flag is asserted only when the FIFO is completely filled, preventing overflow, while the empty flag is asserted only when all data has been read, avoiding underflow. Pointer synchronization assertions check that the read and write pointers update correctly across clock domains, ensuring metastability handling. Additional assertions validate that writes are ignored when FIFO is full and reads are blocked when FIFO is empty, enforcing correct control logic.

By incorporating assertions in the UVM-based verification environment, functional correctness is continuously monitored throughout simulation. This approach improves coverage by automatically detecting violations, reducing debugging time, and ensuring that all critical scenarios are exercised, leading to a more robust and verified FIFO design.

9 Resources requirements

9.1 Team members and who is doing what and expertise.

Mahesh Naidu:led the implementation of the top-level module, focusing on parameterization and the instantiation of internal components. He developed the clock and reset logic to ensure proper FIFO initialization and synchronization. Additionally, he integrated the FIFO module into the UVM-based testbench environment, enabling seamless interaction with other verification components.

Venkata Sai Dhilli: reviewed the design specifications and developed the scoreboard class, which handled read and write operations while monitoring half-full, half-empty, full, and empty conditions. She also designed an interface that supported both class-based and UVM testbenches, ensuring smooth communication between components. Additionally, she implemented clocking mechanisms for the driver and monitor and led the development of functional coverage models for the asynchronous FIFO, capturing critical features and scenarios to enable thorough verification.

Alaina Anand Nekuri: played a key role in the functional verification of the FIFO module. He designed and executed various test cases to validate data transfer, flag signaling, and reset behavior. He also analyzed code and functional coverage reports to evaluate verification completeness and identify potential coverage gaps in UVM components. Furthermore, he debugged and resolved simulation issues, ensuring the overall correctness and reliability of the FIFO design.

Siddhartha Kaushik Gatta:Contributed to the verification process by implementing coverage-driven verification strategies, ensuring comprehensive testing of the FIFO module. She focused on achieving statement, branch, and expression coverage, collaborating closely with team members to coordinate verification efforts and resolve challenges. Additionally, she documented the verification plan, UVM architecture, test cases, and results, compiling detailed reports that tracked verification progress and key findings and also conducted Bug injection method.

10 Schedule

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

	Tasks	**Timeline**
1	Design specs, Verification Plan, RTL implementation, basic testbench.	Week 1-4
2	Class-based TB, transactions, generator, driver, randomized data tests.	Week 4-7
3	Finalize RTL, complete class-based TB, coverage reports.	Week 8-10
4	Develop UVM TB, add UVM plan, implement logging.	Week 11-13
5	Complete UVM TB, final coverage, bug injection, final docs, presentation.	Week 14-16

11 References Uses / Citations/Acknowledgements

[1] https://hardwaregeeksblog.wordpress.com/wp-content/uploads/2016/12/fifodepthcalculationmadeeasy2.pdf

[2] Slides from Professor Venkatesh Patil

[3] https://vlsiverify.com/verilog/verilog-codes/asynchronous-fifo/

[4] https://ieeexplore.ieee.org/document/10090696

[5] http://www.sunburst-design.com/papers/CummingsSNUG2002SJ_FIFO1.pdf