VERIFICATION TEST PLAN

ECE-593: Fundamentals of Pre-Silicon Validation

Maseeh College of Engineering and Computer Science

Winter, 2025

Design and Verification of Asynchronous FIFO using Class Based Verification & UVM

Team 12

Aravindh Nanjaiya Latha (aravindh@pdx.edu)

Rishi Gunda (rgunda@pdx.edu)

Rohit Bonigala (rohit@pdx.edu)

Shreya Umesh Shetty (shreyau@pdx.edu)

Date: May 29, 2025

GitHub Repository: https://github.com/aravindh-nanjaiya-latha/Team12_Asynchronous-FIFO_S25_ECE593

1 Table of Contents

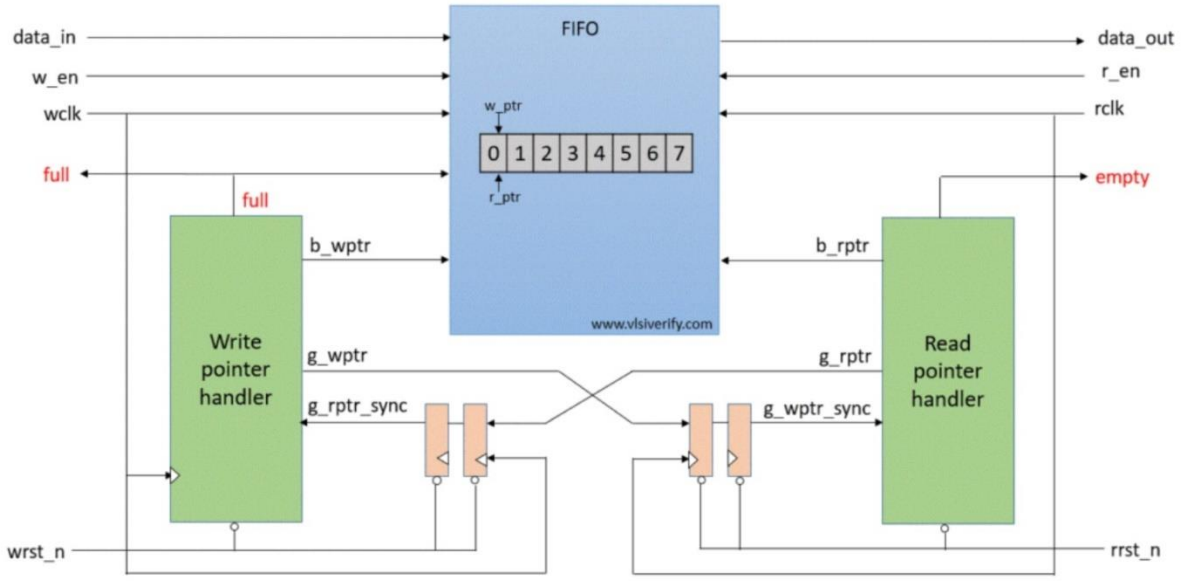
2 Introduction:

2.1 Objective of the verification plan

The objective is to ensure the FIFO meets all functional and performance requirements including:

- Verifying data integrity across clock domains.
- Ensuring accurate full, empty and half_full flag generation.
- Validating reset behavior and pointer synchronization.
- Testing concurrent read/write operations and edge cases.
- Achieving 100% code coverage (line, branch, toggle) and comprehensive functional coverage using UVM methodologies

2.2 Top Level block diagram



Asynchronous FIFO

2.3 Specifications for the design

- Data Width (DSIZE): 8 bits
- FIFO Depth: 128 entries (ASIZE=7)
- Clock Domains:
  - Write Clock (wclk): 500 MHz (2 ns period)
  - Read Clock (rclk): 450.45 MHz (2.22 ns period)
- Reset Signals:
  - wrst_n: Active-low, asynchronous reset for write domain
  - rrst_n: Active-low, asynchronous reset for read domain
- Control Signals:
  - w_en: Active-high, enables write when full=0
  - r_en: Active-high, enables read when empty=0

- Status Signals:
    - full: Asserted when FIFO is full
    - half_full: Asserted when FIFO is half full
    - empty: Asserted when FIFO is empty
- Data Ports:
    - data_in: 8-bit input
    - data_out: 8-bit output
- Synchronization: Gray code for pointer synchronization to minimize metastability

3 Verification Requirements

3.1 Verification Levels

3.1.1 What hierarchy level are you verifying and why?

- Level: Module-level
- Reason: The FIFO is a standalone module with well-defined interfaces, making module-level verification ideal for isolating internal logic (e.g., pointer synchronization flag generation) without system-level complexity.

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

- Controllability: High – Inputs (wclk, rclk, wrst_n, rrst_n, w_en, r_en, data_in) are fully controllable via the testbench, enabling precise stimulation of all scenarios.

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

Signal	Direction	Description
wclk	Input	Write clock
rclk	Input	Read clock
wrst_n	Input	Active-low write reset
rrst_n	Input	Active-low read reset
w_en	Input	Write enable control
r_en	Input	Read enable control

data_in[7:0]	Input	8-bit write data
data_out[7:0]	Output	8-bit read data
full	Output	FIFO full flag
half_full	Output	FIFO half full flag
empty	Output	FIFO empty flag

4 Required Tools

4.1 List of required software and hardware toolsets needed.

Software:

- QuestaSim 2024.31: For simulation, debugging, and coverage analysis
- System Verilog: For testbench development
- UVM Libraries: For advanced verification components
- VCS: Optional for cross-verification

Hardware:

- Linux workstation (16GB RAM, multi-core CPU)

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

- /design/: async_fifo.sv
- /testbench/: async_fifo_tb.sv, run.do
- /sim/: Logs (qrun.log), coverage reports (async_fifo_coverage.ucdb), waveforms
- /docs/: Verification plan, reports

5 Risks and Dependencies

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

Critical Threats

• Licensing Issues:

– Description: Simulation failures due to incorrect SALT_LICENSE_SERVER setup, as seen in logs (Read failure in vlm process).

– Mitigation: Set SALT_LICENSE_SERVER=1717@license-server-hostname; verify with lmstat; contact Siemens EDA Support if issues persist.

• Asynchronous Bugs:

– Description: Potential metastability or synchronization errors in pointer logic.

– Mitigation: Use UVM random tests, formal verification, and assertions to detect violations.

• Incomplete Coverage:

– Description: Missing corner cases (e.g., simultaneous read/write).

– Mitigation: Directed tests for edge cases; monitor coverage reports.

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

Function	Description	Verification Method
Write Operation	Data written when w_en=1, full=0	UVM sequences for writes
Read Operation	Data read when r_en=1, empty=0	UVM sequences for reads
Full Flag	full asserted when FIFO full	Check during write tests
Empty Flag	empty asserted when FIFO empty	Check during read tests
Half Full Flag	half_full asserted when FIFO is half full	Checking during the write tests
Reset	Pointers reset; flags set correctly	Directed reset tests
Synchronization	Gray code pointer updates	Formal checks; random tests

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

- Advanced error handling: Deferred to future milestones.
- Power-on reset: Handled at system level.

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

- Critical: Write/read operations, flag generation, reset, synchronization
- Non-Critical: None

7 Tests and Methods

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

- Method: White-Box testing

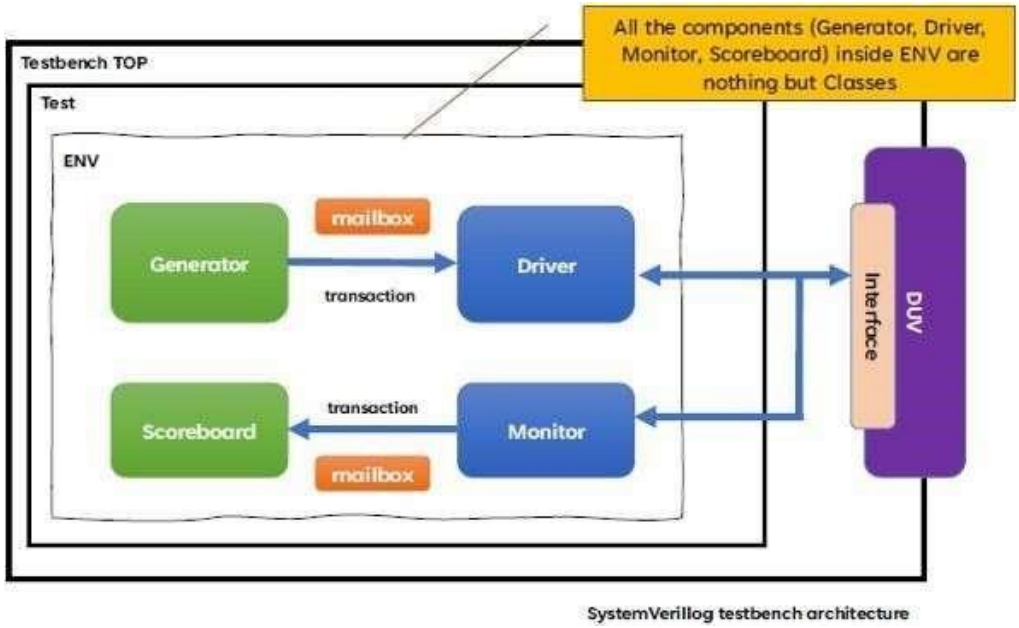
7.1.2 State the PROs and CONs for each and why you selected the method for this DUV.

- Pros: Balances internal design knowledge with external behavior testing.
- Cons: Requires design insight.
- Reason: Ideal for verifying synchronization and functionality.

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

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

- Generator: Creates transactions using UVM sequences.
- Driver: Drives w_en, r_en, data_in via clocking blocks.
- Monitor: Captures data_out, full, empty.
- Scoreboard: Compares expected vs. actual data.
- Coverage Collector: Tracks functional coverage.



SystemVerilog testbench architecture

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

- Strategy: Dynamic simulation with UVM, constrained random testing
- Reason: Covers diverse asynchronous scenarios.

7.1.5 What is your driving methodology?

- Transaction-based using UVM mailboxes.

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

- Directed tests for basic functionality and corner cases.
- Directed tests for basic functionality and corner cases.

7.1.6 What will be your checking methodology?

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

- Scoreboard-based comparison of reference queue vs. DUT output.

7.1.7 Testcase Scenarios (Matrix)

7.1.7.1 Basic Tests

Test Name / Number	Test Description/ Features
1.1.1	Reset: Verify wrst_n, rrst_n set full=0, empty=1
1.1.2	Write to empty FIFO: Check data storage, full behavior
1.1.3	Read from non-empty FIFO: Ensure data retrieval, empty updates
1.1.4	Pointer increment: Validate wptr, rptr updates
1.1.5	Single write/read cycle: Test basic data transfer

7.1.7.2 Complex Tests

Test Name / Number	Test Description/ Features
1.2.1	Concurrent read/write: Simultaneous operations with 500 MHz wclk, 450.45 MHz rclk
1.2.2	Full condition: Write 512 entries, verify full
1.2.3	Empty condition: Read until empty, verify empty
1.2.4	Half Full Condition: Write 256 entries, verify half_full
1.2.5	Clock skew: Test with max skew (1 ns)
1.2.6	Random transaction bursts: Mixed read/write patterns

7.1.7.3 Regression Tests (Must pass every time)

Test Name / Number	Test Description/Features
1.3.1	Reset consistency: Multiple reset cycles
1.3.2	Basic FIFO operations: Standard write/read sequence

7.1.7.4 Any special or corner cases testcases

Test Name / Number	Test Description
1.4.1	Back-to-back operations: Rapid write/read sequences
1.4.2	Max clock skew: Extreme timing differences
1.4.3	Data boundaries: Max/min data values (0x00, 0xFF)
1.4.4	Reset during operation: Reset during active read/write

8 Coverage Requirements

8.1.1.1 Describe Code and Functional Coverage goals for the DUV

- Code Coverage: 100% line, branch, toggle coverage using QuestaSim's -cover bcesf.
- Functional Coverage: Cover all scenarios: Write operations (empty, almost full, full)

  Read operations (empty, almost empty, full)

  Concurrent read/write

  Reset conditions

  Pointer synchronization

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

- Covergroups:

  w_en x full: Bins for valid writes (w_en=1, full=0), writes when full (w_en=1, full=1) r_en x empty:

  Bins for valid reads (r_en=1, empty=0), reads when empty (r_en=1, empty=1) w_en x r_en: Bins

  for simultaneous read/write wrst_n, rrst_n: Bins for reset states

- Achievement: Directed tests for basic scenarios; random tests for coverage closure; regular coverage report analysis.

9 Resources requirements

9.1 Team members and who is doing what and expertise.

Task Group	Member(s)	Responsibilities	Key Deliverables
Spec & Architecture	All	Define precise FIFO specifications; Create top-level block diagrams; Define interfaces; Identify verification levels; Set controllability/observability targets.	Specification document; Block diagram; Interface definition document; Verification level rationale.
CDC Analysis & Core RTL	Aravindh Nanjaiya Latha	Implement clock domain crossing circuits; Develop write/read pointer logic; Design FIFO memory interface; Address any synthesis concerns.	UVM agent, transactions and coverage classes. RTL write and read pointers, increment logic implementation.
UVM Infrastructure	Rishi Gunda	Develop UVM testbench structure; Implement base classes; Create transaction-level models; Design sequencer-driver architecture; Build reusable utilities.	UVM driver, sequencer and sequence item coding. RTL code modification. UVM reporting and bug injection.
Coverage & Assertions	Rohit Bonigala	Define functional coverage points; Implement cover groups; Develop System Verilog Assertions (SVA); Analyse coverage results; Refine test sequences to close coverage holes.	Functional coverage models; SVA properties; Coverage reports; Test sequence updates.
Test Development & Debug	Shreya Umesh Shetty	Create directed tests; Develop constrained-random tests; Implement error injection mechanisms; Debug test failures; Verify reset behaviour.	UVM monitor, environment and scoreboard development. RTL code binary to gray code implementation

10 References Uses / Citations/Acknowledgements

- S. Cummings, "FIFOs: Fast, predictable, and deep (Part II)," in Proceedings of SNUG, 2002.
  - http://www.sunburst-design.com/papers/CummingsSNUG2002SJ_FIFO2.pdf
- ASIC-World FIFO Examples
  - http://www.asic-world.com/examples/verilog/fifo.html
- VLSI Verify
  - https://vlsiverify.com/verilog/verilog-codes/asynchronous-fifo/