DESIGN SPECIFICATION REPORT

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 & UVM

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)

Project Name	Design and Verification of Asynchronous FIFO using Class Based & UVM
Location	Portland
Start Date	25 January 2025
Estimated Finish Date	March 11, 2025

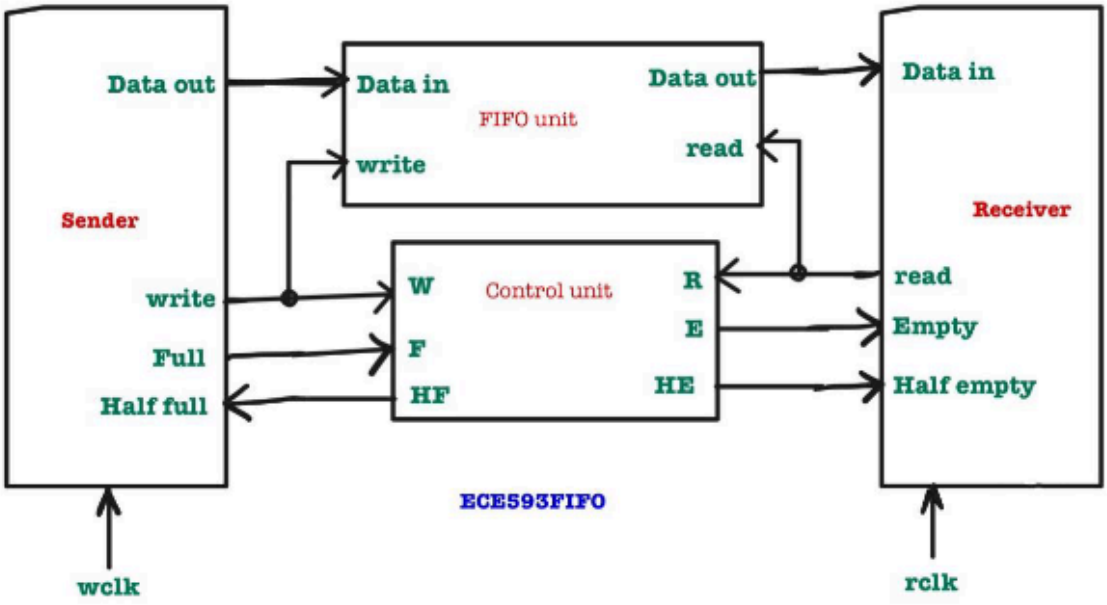
Prepared by: Team 15	
Prepared for: Prof. Venkatesh Patil	
Alaina Anand Nekuri	alainaa@pdx.edu
Mahesh Naidu	maheshn@pdx.edu
Siddhartha Kaushik Gatta	sgatta@pdx.edu
Venkata Sai Dhilli	dhilli@pdx.edu

INTRODUCTION

In digital systems, efficient data transfer between different clock domains is crucial. An Asynchronous FIFO (First-In, First-Out) buffer is designed to handle this scenario by allowing data to be read and written using separate independent clock signals. Unlike Synchronous FIFOs, which operate under a single clock domain, Asynchronous FIFOs provide a flexible way to transfer data across systems running at different speeds without data loss or corruption.

The key advantage of an Asynchronous FIFO is that it acts as a buffer, preventing overflow and ensuring smooth communication between sender and receiver modules. However, designing such a system presents challenges such as metastability handling, synchronization, and data integrity. Metastability occurs when signals cross between asynchronous clock domains, which can lead to undefined logic states. To mitigate this, techniques such as Gray-coded pointers, synchronizers, and proper clock domain crossing methodologies should be used.

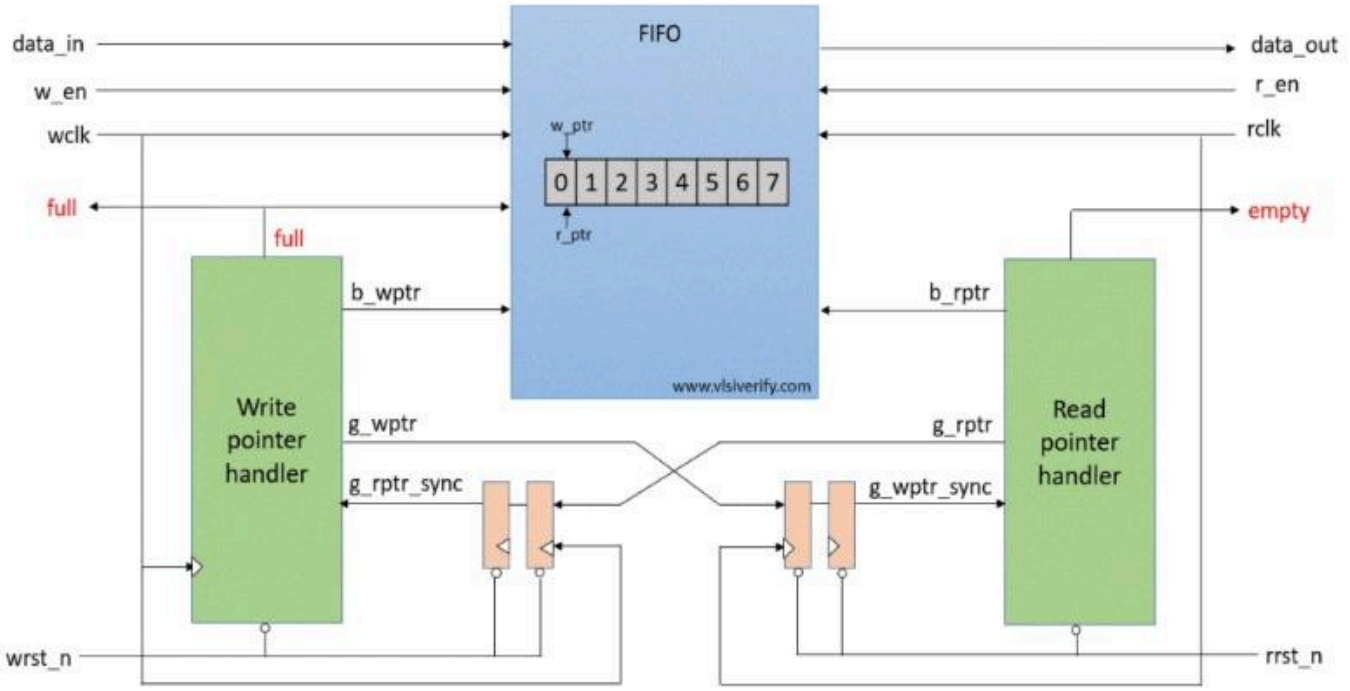
Key challenges in designing an asynchronous FIFO include Metastability Handling, pointer management, gray code encoding.



DESIGN SPECIFICATION

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.

FIFO CALCULATION

Writing frequency of sender = f(sender) = 120MHz.

Reading Frequency of receiver = f(receiver) = 50MHz.

Burst Length = No. of data items to be transferred = 1024.

No. of idle cycles between two successive writes is = 3.
No. of idle cycles between two successive reads is = 2.

The no. of idle cycles between two successive writes is 3 clock cycle. It means that, after writing one data,Sender is waiting for Three clock cycle, to initiate the next write. So, it can be understood that for every four clock cycles, one data is written.
The no. of idle cycles between two successive reads is 2 clock cycles. It means that, after reading one data, module B is waiting for 2 clock cycles, to initiate the next read. So, it can be understood that for every three clock cycles, one data is read.

Time required to write one data item = $4 * (1/120)$ = 33.33 nSec.
Time required to write all the data in the burst = 1024 * 33.33 nSec. = 34,129.92 nSec.
Time required to read one data item = 3*(1/50) = 60 nSec.

So, for every 60 nSec, the Receiver is going to read one data in the burst.
So, in a period of 34129.92 nSec, 1024 no. of data items can be written.

The no. of data items can be read in a period of 34129.29 nSec = 34,129.92/60 = 568.82 ≃ 569
The remaining no. of bytes to be stored in the FIFO = 1024 – 569 = 455.
So, the FIFO which has to be in this scenario must be capable of storing atleast 455 data items.
So, the minimum depth of the FIFO should be 455.

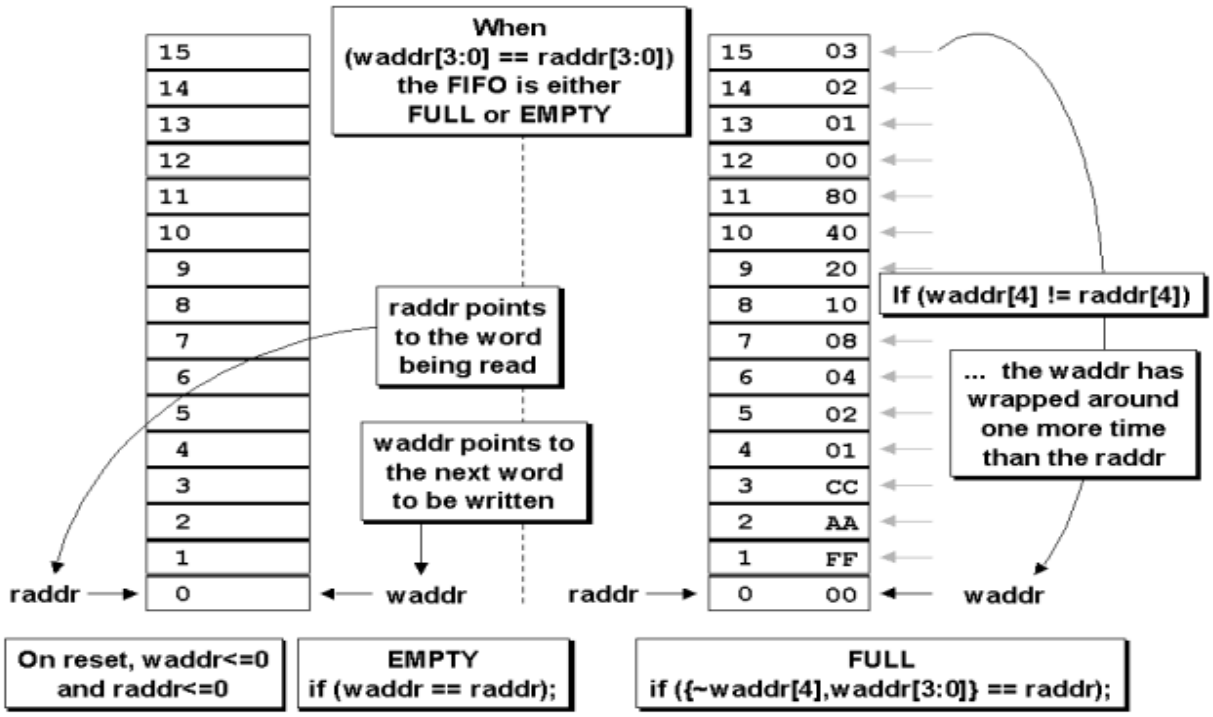
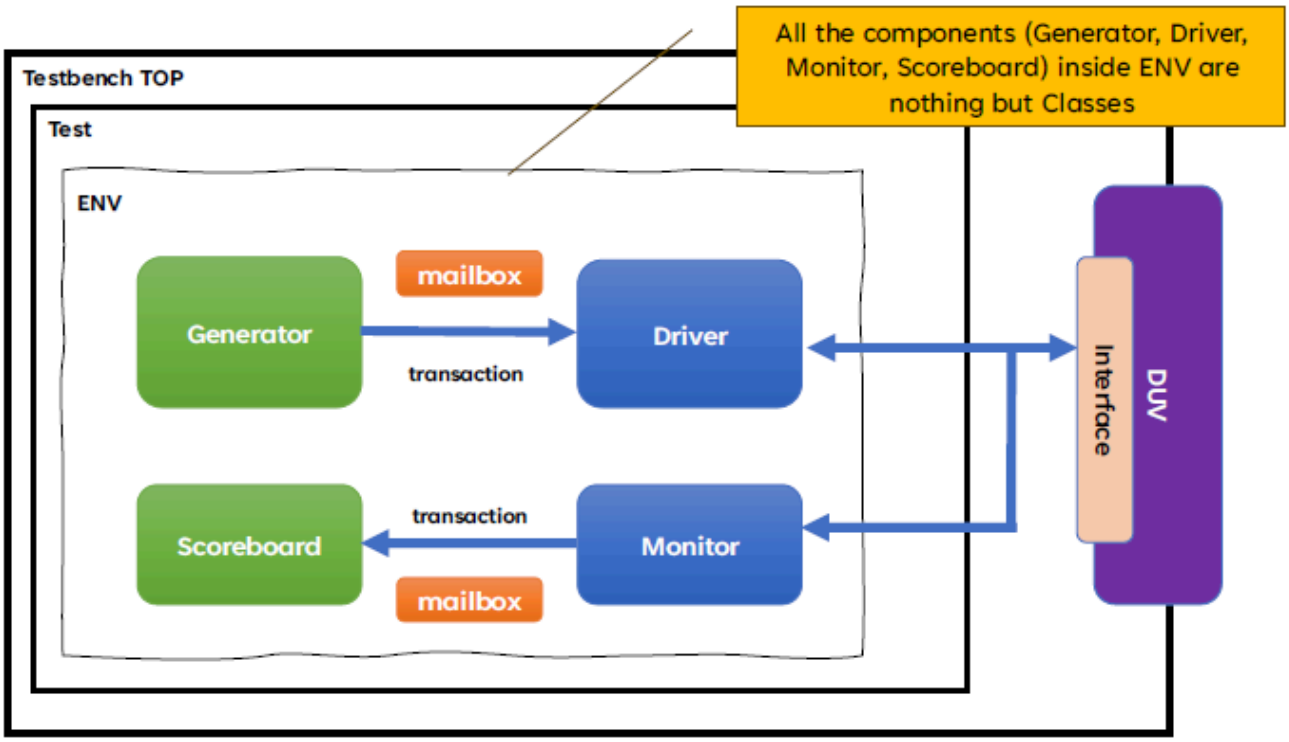


Figure 1 - FIFO full and empty conditions

SV 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.



SystemVerilog testbench architecture

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.

TEST PLAN:

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 occur.	Each read should return the correspondi -ng previously written data, maintaining the FIFO order. The empty and full 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.

FUNCTIONAL COVERAGE

Functional coverage plays a vital role in ensuring that the 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.

- Read and Write Transactions: Validates that data is written into the FIFO and retrieved in the same 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. In our verification, the achieved functional coverage is 83.33%, ensuring that the FIFO meets the expected performance criteria.

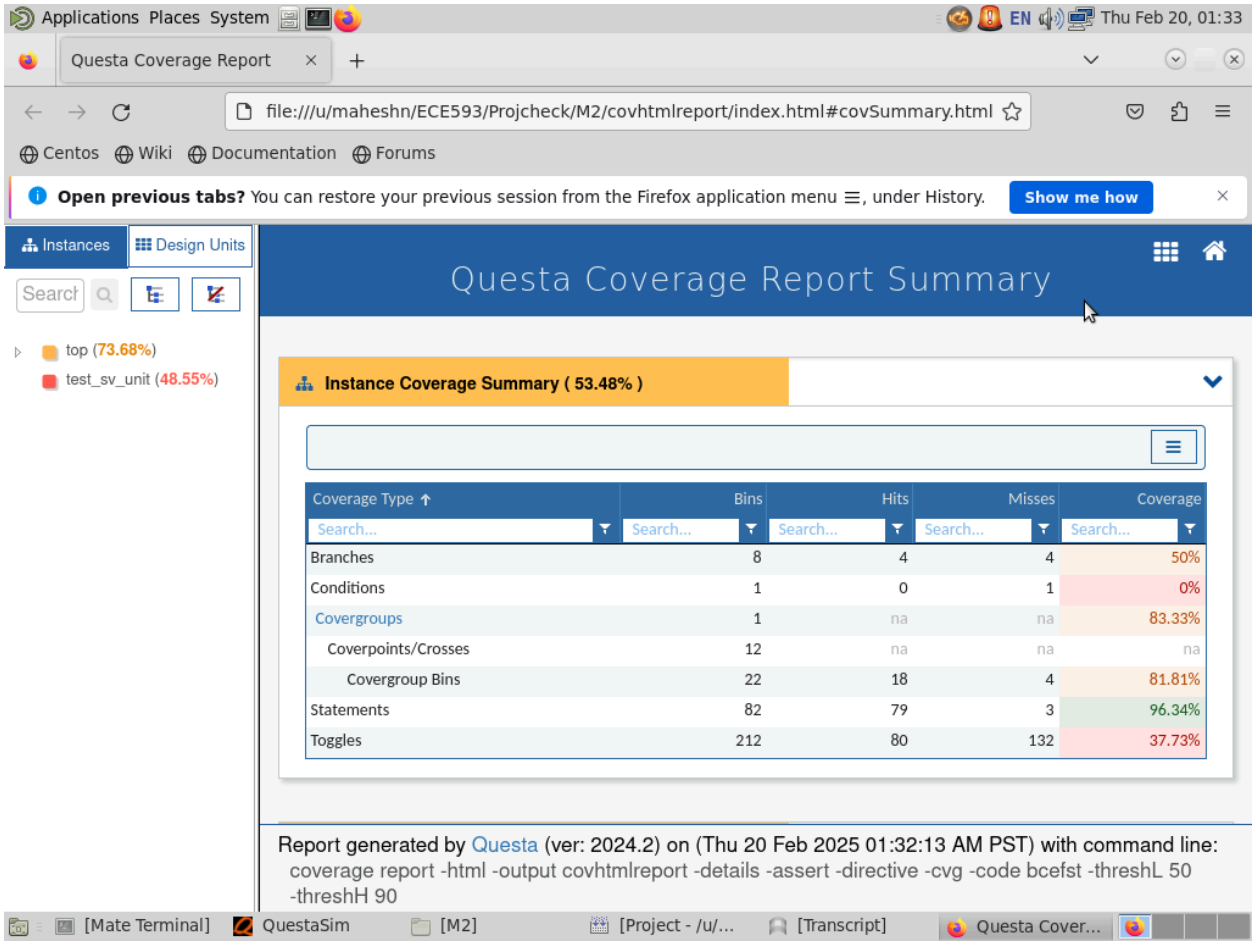
CODE COVERAGE:

Code coverage analysis was conducted to evaluate how effectively the FIFO RTL design was exercised during verification. The current achieved total coverage stands at 46.01%, indicating a significant portion of the design has been tested. However, our target is to achieve at least **53.48%** coverage to ensure a more rigorous and comprehensive validation of the design.

- Statement Coverage: Ensured 96.34% 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 50% coverage.
- FSM Coverage: Ensured all FIFO state transitions were visited and exercised.
- Toggle Coverage: Verified that all flip-flops, registers, and storage elements toggled between 0 and 1 at least once, achieving 37.73% coverage.

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

Results Obtained:



CONTRIBUTION:

- Mahesh Naidu: Led the development of the top-level module, ensuring proper parameterization and integration of internal components. Designed and implemented the clock generation and reset logic, guaranteeing correct synchronization and initialization. Integrated the FIFO module with the testbench, streamlining simulation and verification efforts.

- Venkata Sai Dhilli: Focused on functional verification, creating and executing test cases to validate data flow, flag operations, and reset behavior. Analyzed coverage reports to identify untested scenarios and enhance test completeness. Debugged issues encountered during simulation to ensure FIFO operates reliably under all conditions.

- Alaina Anand Nekuri: Developed a coverage-driven verification approach, implementing checks for statement, branch, and expression coverage. Worked closely with the team to

optimize test strategies, ensuring comprehensive scenario validation. Compiled verification documentation, detailing test methodologies and outcomes.

- Siddhartha Kaushik Gatta: Studied the FIFO design specifications and created the scoreboard class to validate read and write operations. Implemented logic for handling half-full, half-empty, full, and empty states, ensuring accurate condition monitoring. Designed the interface for the testbench, managing signal connections and clock synchronization. Led efforts in functional coverage modeling, ensuring all critical behaviors were thoroughly tested.

REFERENCES:

[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