Asynchronous FIFO

Group Number

Date: 03/04/2025

Siddesh
Satyajit
Siddarth
Sai Ganesh

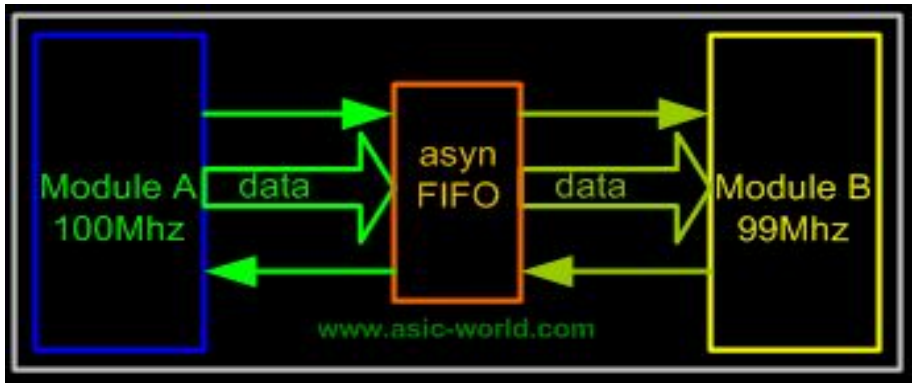
Portland State
UNIVERSITY

Contents

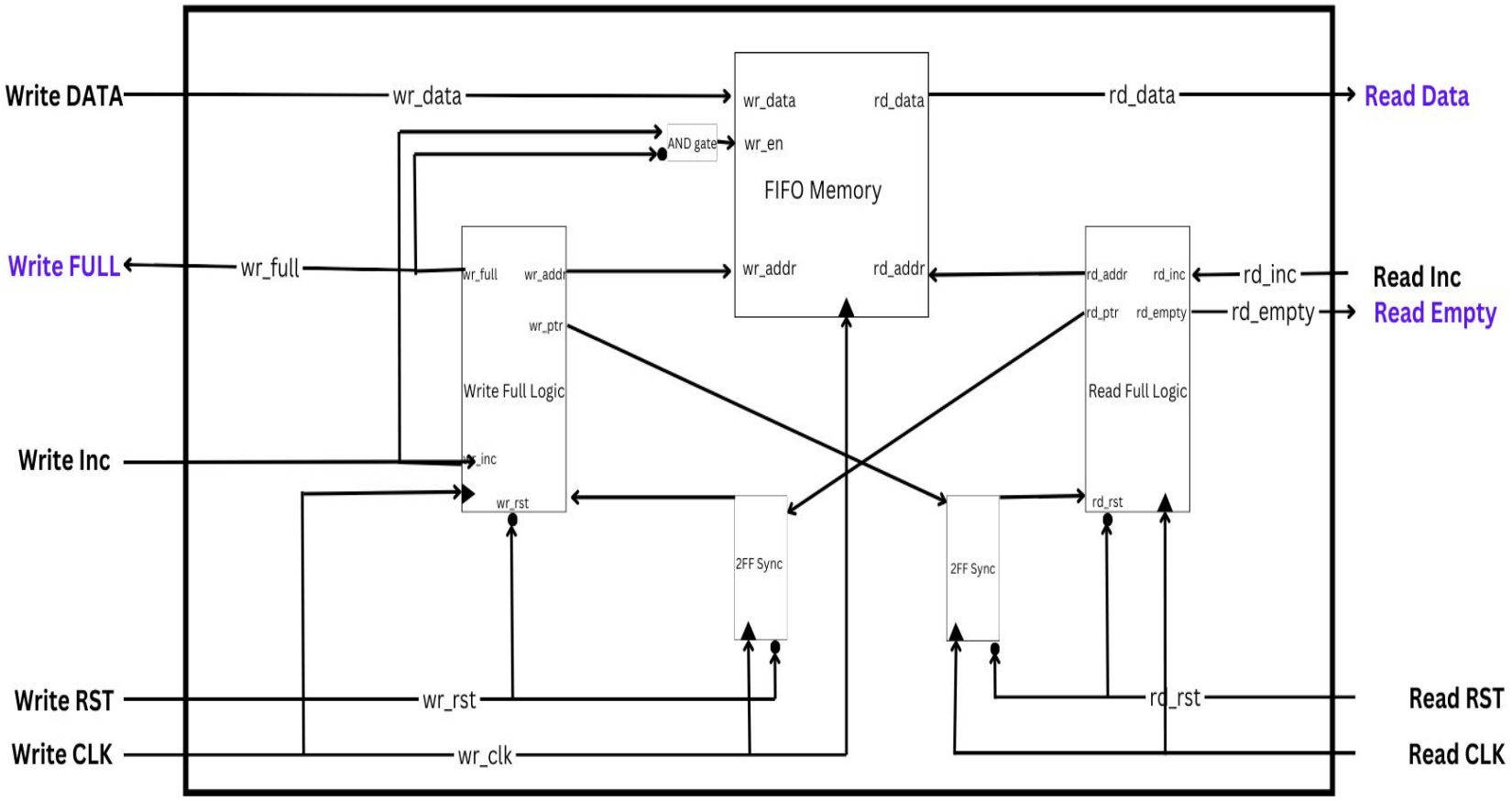
Introduction

What is an Asynchronous FIFO?

- Asynchronous FIFO is a memory buffer used to transfer data between two different clock domains.
- It ensures smooth data communication while handling different read and write clock speeds.
- Synchronization techniques, such as Gray code pointers, help prevent metastability issues.
- Status flags like Full, Empty, Almost Full, and Almost Empty control data flow and prevent overflow or underflow.
- Commonly used in clock domain crossing (CDC) circuits, it improves reliability in high-speed digital designs.



Design Implementation



Top Level block Diagram

Design Implementation

Use of Gray Code:

Employs Gray code for pointer synchronization, reducing metastability risks across different clock domains.

Pointer Structure:

- Write Pointer: Tracks the next write location, resets to zero on initialization, and increments with each data write.
- Read Pointer: Points to the current read location, resets to zero initially, increments after each write, and clears the empty flag while driving the first valid word to the output for immediate reading.

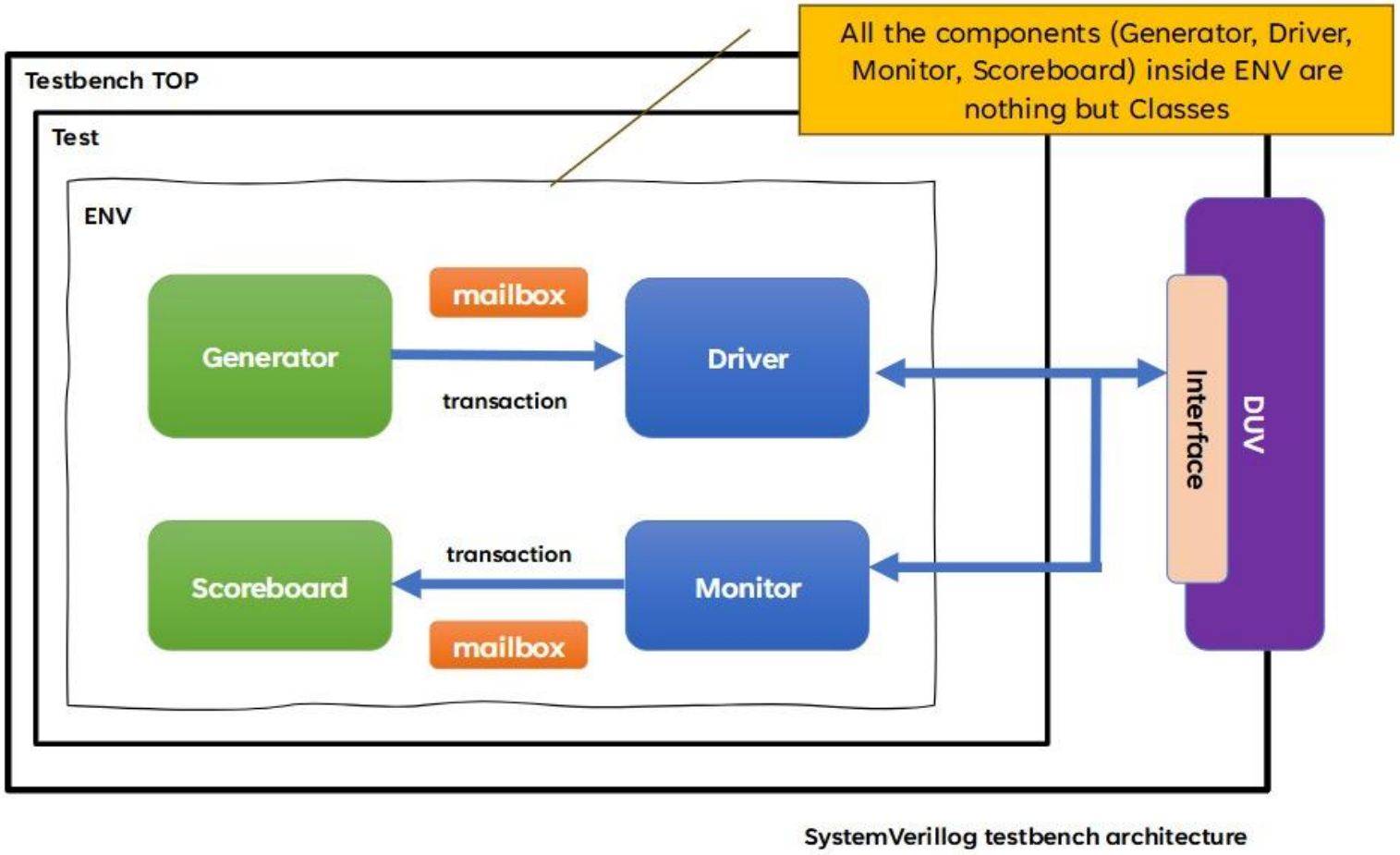
Empty and Full Conditions:

- The FIFO is empty when the write and read pointers are identical.
- The FIFO is full when the pointers match again after a complete cycle (wrapping around).

Distinguishing Between Empty and Full States:

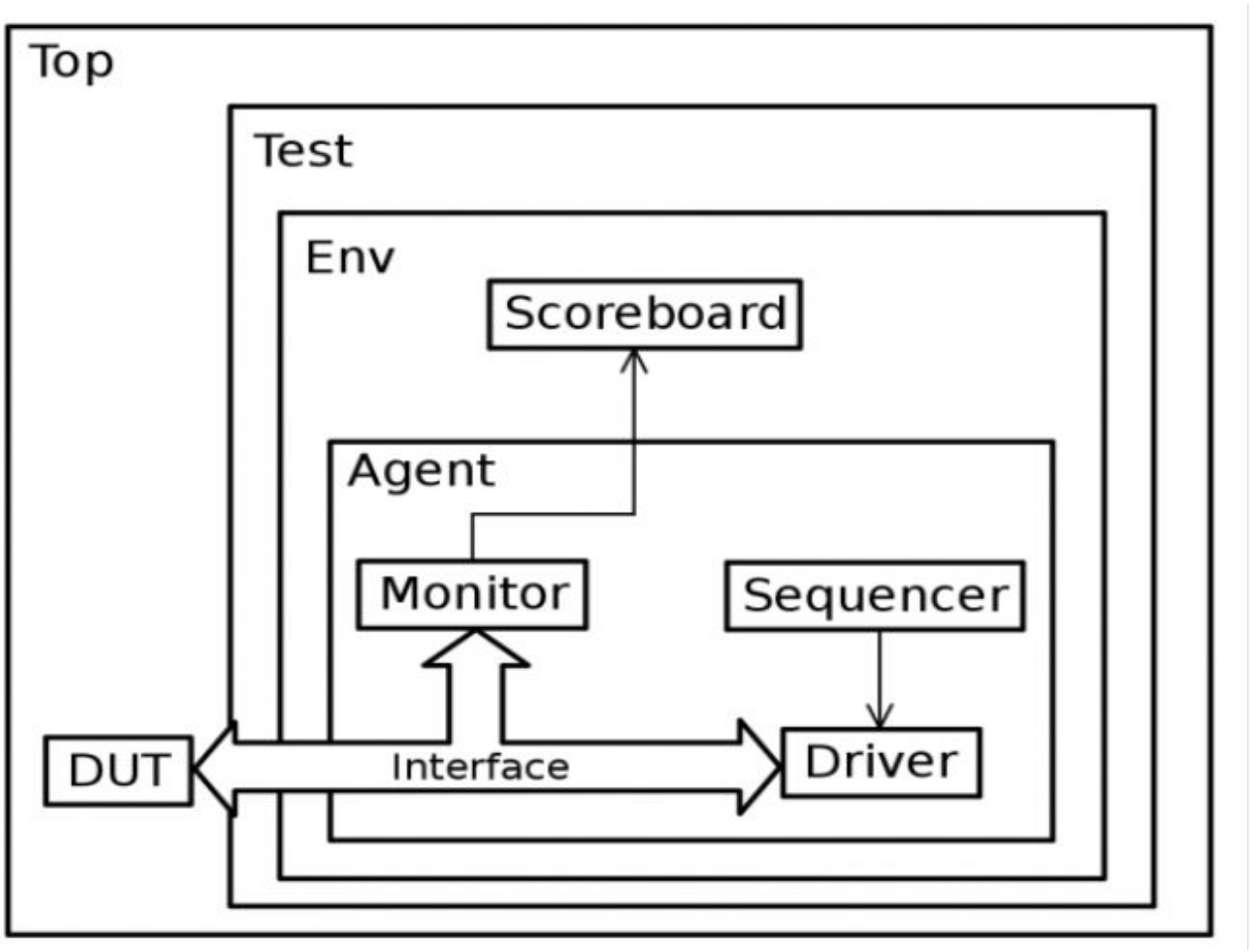
- An extra bit is included in each pointer for differentiation.
- Empty Condition: All pointer bits, including the most significant bit (MSB), are identical.
- Full Condition: All pointer bits match except for the MSB.

Class Based Testbench Architecture



SystemVerillog testbench architecture

UVM based Verification



Tests and Coverage

Tests covered:
Empty state
Basic Read
Basic Write
Full Condition Empty Condition
Random testing
Bug Injection

Total Coverage By Instance (filtered view): 77.89%

```
Covergroup Coverage:
    Covergroups                  2        na        na    77.89%
        Coverpoints/Crosses     10        na        na        na
            Covergroup Bins    524       212       312    40.45%
```

Challenges

Three major challenges we faced were

1. **TLM Transactions in UVM Architecture:** Understanding and implementing Transaction-Level Modeling (TLM) in the UVM architecture was initially challenging. Grasping the concepts of TLM interfaces, communication between components, and effective use of analysis ports required significant effort and debugging.
2. **Logging Files:** Efficiently handling and managing log files posed a challenge. Extracting meaningful insights from large volumes of simulation logs and structuring them for effective debugging demanded a systematic approach.
3. **Achieving High Functional Coverage:** Initially, achieving good functional coverage was difficult. Identifying missing scenarios, refining coverage points, and optimizing test cases required continuous iterations. Through systematic enhancements, I was able to significantly improve the coverage percentage.

Key Learnings:

- **UVM Reusability and Flexibility:** Overcoming UVM-related challenges provided valuable insights into its reusability and flexibility. Learning to design modular and reusable testbenches will be highly advantageous for future projects.
- **Industry Preparedness:** Our experience with UVM and coverage has equipped us with practical, industry-relevant skills. We now have a deeper understanding of professional workflows and project expectations.
- **Enhanced Problem-Solving Skills:** Addressing coverage challenges strengthened our problem-solving abilities in verification. We developed the capability to identify coverage gaps and implement effective strategies to close them.

Conclusion

- Designed and verified an asynchronous FIFO using SystemVerilog and UVM, ensuring accuracy and reliability.
- Developed a structured UVM-based testbench, improving reusability and scalability in the verification workflow.
- Applied functional coverage and thorough debugging strategies to achieve high verification confidence.
- Used a combination of directed tests for core functionality and random stimulus to identify edge cases, ensuring comprehensive validation.
- Verified that the FIFO design meets performance and reliability standards across various operating conditions.

References

- UVM Class notes
- UVM Primer by Ray Salemi
- https://vlsiverify.com/verilog/verilog-codes/asynchronous-fifo/
- http://www.sunburst-design.com/papers/CummingsSNUG2002SJ_FIFO1.pdf
- http://www.sunburst-design.com/papers/CummingsSNUG2002SJ_FIFO2.pdf
- FIFO Depth Calculation MadeEasy PDF
- https://ieeexplore.ieee.org/abstract/document/7237325
- https://github.com/raysalemi/uvmprimer