Design and Verification of Asynchronous FIFO Using Both Class Based and UVM methodologies

Presenting :
Mahesh Naidu
Alaina Anand Nekuri
Siddhartha Kaushik Gatta
Venkata Sai Dhilli

Table of Context

Introduction

An Asynchronous FIFO is Used to enable data transfer between two different clock frequencies Where the two devices are not synchronous. so it is referred as Asynchronous FIFO. Asynchronous FIFO helps to synchronize data flow between two systems working on different clocks. Where in synchronous FIFO both read and write operation happens at same clock. But in asynchronous FIFO they both are happening independently.

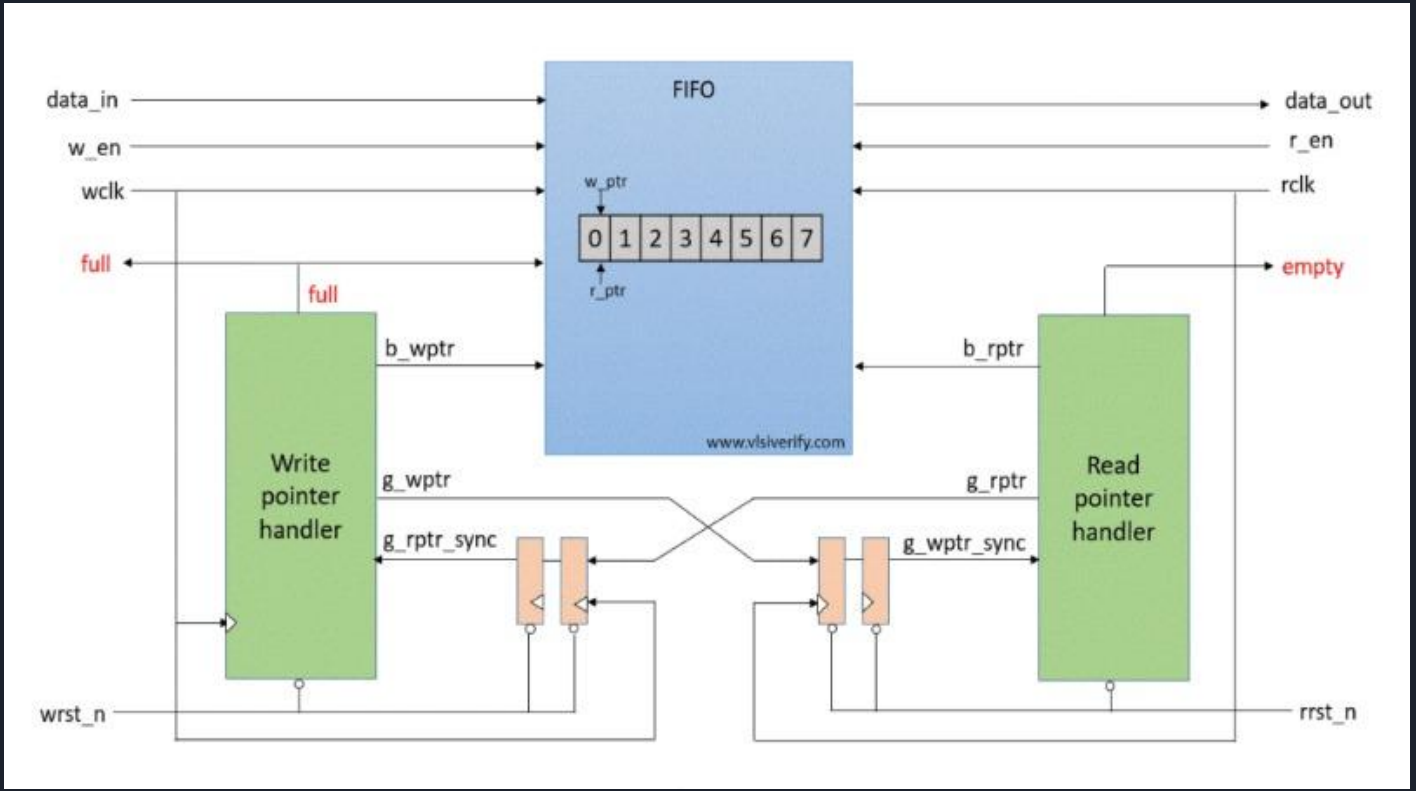
FIFO : is a memory structure where the first item written is the first item to be read out. (First In First Out)

Introduction

Advantages of using Asynchronous FIFO??

- It ensures proper data synchronization without data loss during data transfer between two different devices or two different systems operating at different frequencies (eg: Laptop and printer).
- It is simpler and more efficient process for transferring data across different clock domains without the need of complex synchronization techniques
- It is used to optimize memory usage to store and retrieve data efficiently.
- Using this we can transfer data between subsystems with different clock frequencies.
- In FIFO buffers the data, allowing it to be read and written at different rates without overflowing or underflowing.

Block Diagram oF Asynchronous FIFO



Design Specifications

- **Asynchronous FIFO** : It is Used to transfer data between different systems or devices operates at different clock frequencies.

- **Memory (FIFO):** This FIFO uses a memory to store the data and read it later. As the data transfer between happends two different frequencies so first data write to the memory in the write clock domain at 120Hz frequency and then read clock domain read the memory at 50Hz frequency.

- **Write Pointer**: it is a binary write pointer monitors the position where the next data will be written. This binary write pointer is then converted into a Gray code write pointer to ensure proper synchronization between different clock domains.

- **Read Pointer**: It is a binary read pointer tracks the location from which the next data will be read. This binary read pointer is then converted into a Gray code read pointer to facilitate synchronization across different clock domains.

Design Specifications

- **Write Pointer Synchronization:** The Gray code write pointer is synchronized into the read clock domain using a two-stage synchronization process. This technique helps mitigate metastability by allowing the pointer to stabilize before it is used in the read clock domain.
- **Read Pointer Synchronization:** Similarly, the Gray code read pointer is synchronized into the write clock domain using a two-stage synchronization method. This ensures that the read pointer is correctly aligned with the write clock, preventing metastability and ensuring reliable data transfer.
- **Full Condition:** it indicate the FIFO is full when the write pointer is equal to the read pointer -1 value.
- **Half Full Condition:** The FIFO is Half when the Full condition read point is at depth/2.

Design Specifications

- **Empty Condition:** it indicates the FIFO is empty when the read pointer is equal to the write pointer.
- **Half Empty Condition:** The FIFO is half when the Empty when write pointer is at depth/2.
- **Gray code :** Gray code is used for the read and write pointers because it guarantees that only one bit changes at a time. This reduces the likelihood of sampling errors and minimizes the risk of metastability during synchronization across clock domains.

Specification and calculations

Write Frequency :120MHz

Read Frequency: 50MHz

Burst Length: No of data bits to be transferred per cycle = 1024.

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

No. of Idle cycles between two successive reads is = 2.

Minimum depth of FIFO = 455

FIFO depth considered here = 1024.

Time required to write one data item = 4*(1/120) = 33.33ns.

Time required to write all the data in the burst = 1024 * 33.33nsec = 34,129.92nSec..

Time required to read one data item = 3*(1/50) = 60 ns.

Full and Empty Conditions

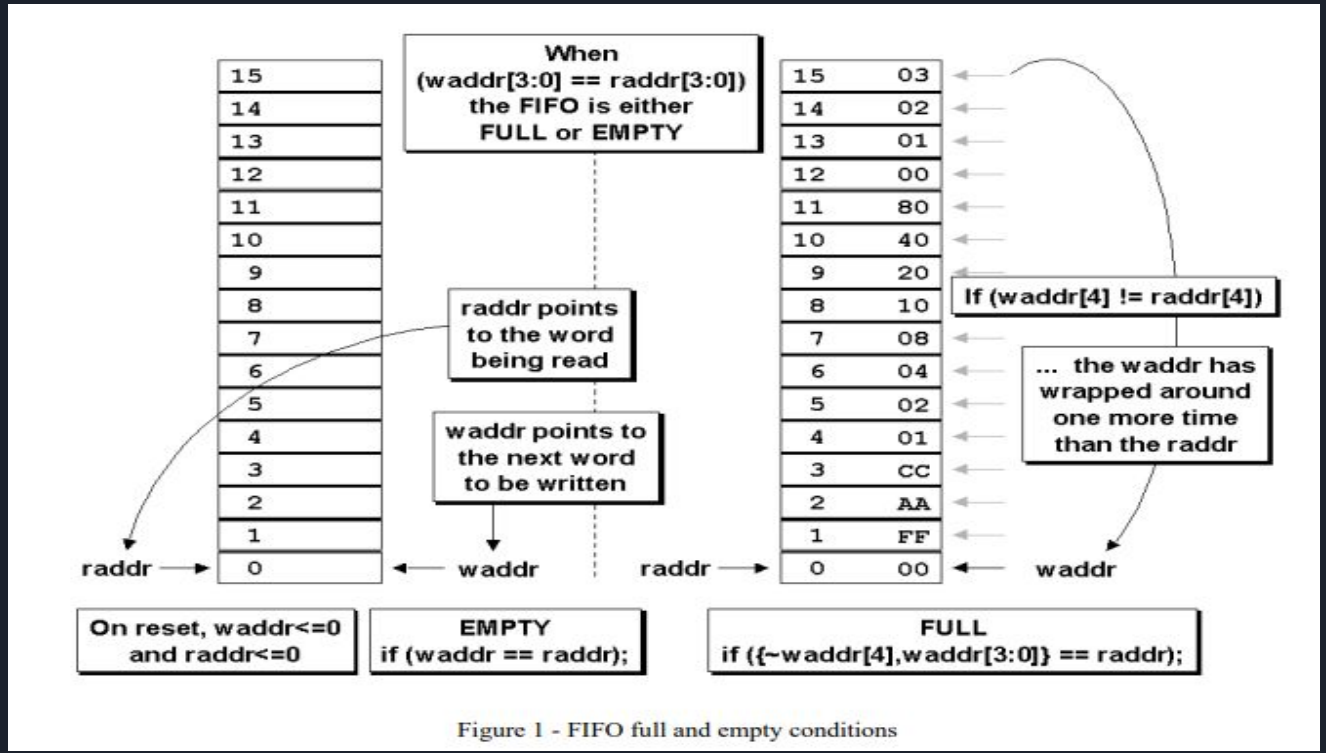
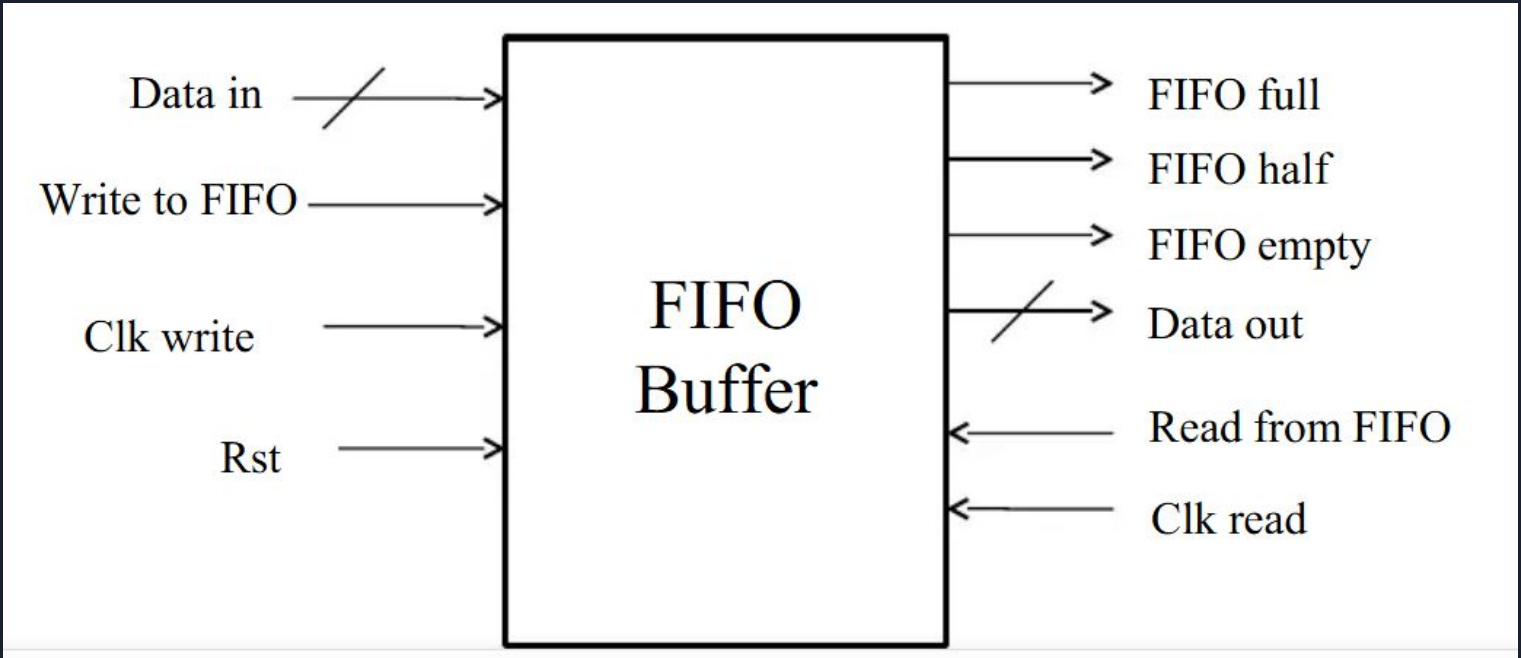


Figure 1 - FIFO full and empty conditions

Block diagram FIFO Buffer or Memory array



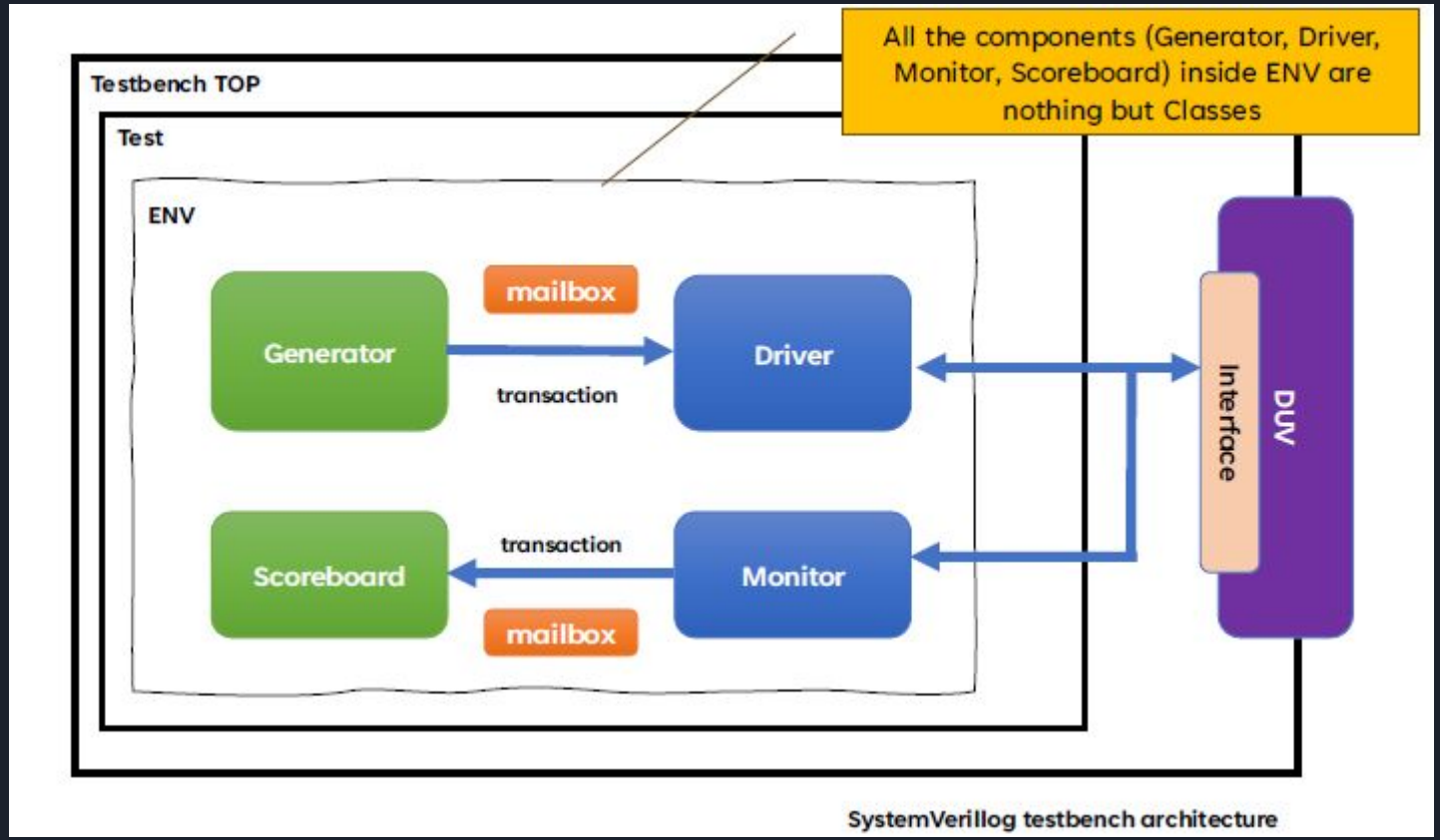
Verification Methods

Class Based Verification and UVM based Verification.

Class Based Verification:

UVM Based Verification:

SV Testbench Architecture or Class based Verification



Class Based Verification

- **Top Module:** The **Testbench** is the top module connecting the Test and DUT using an Interface. It is responsible for generating the clock and initializing the test environment.

- **Environment:** **Test Component** creates the **Environment** and sets the number of transactions to generate. The **Environment** is the core of the testbench, containing major verification components:**Generator, Driver, Monitor, Scoreboard, and Transaction**. It ensures proper synchronization and execution of transactions between components.

- **Generator:** Creates test stimulus by **randomizing transaction data** and Sends transactions to the **Driver** via a **Mailbox**.

Class Based Verification

- **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.

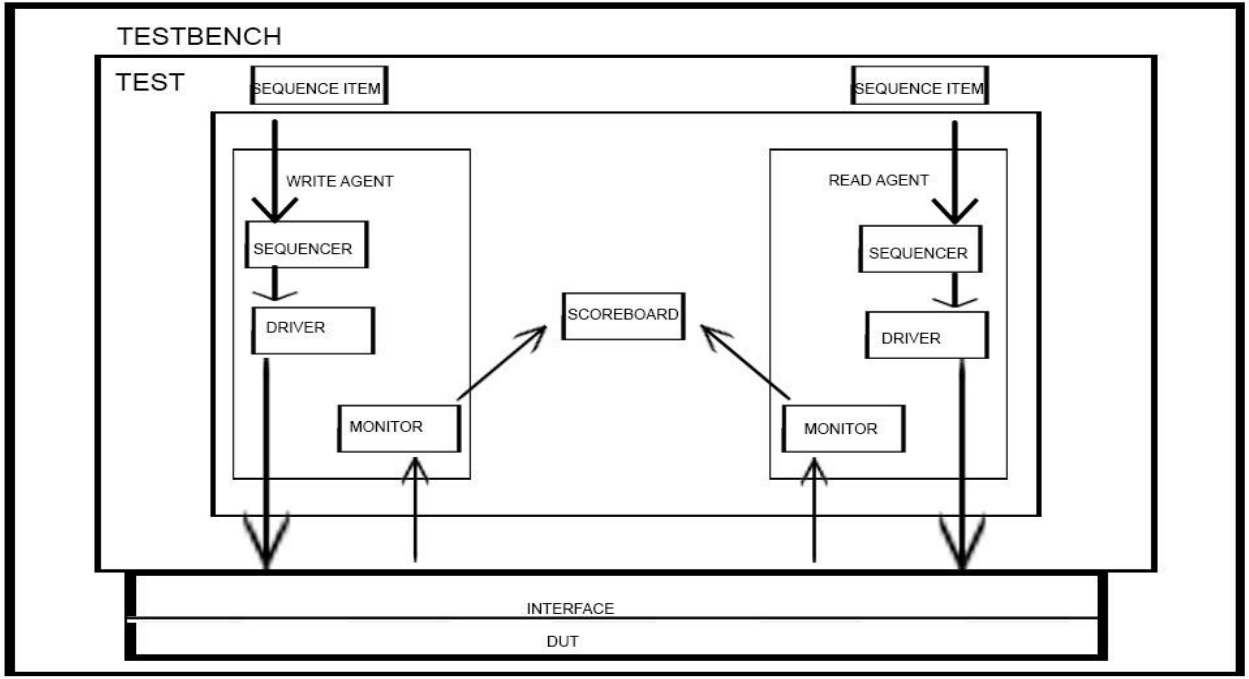
- **Monitor:** Passively observes the **DUT's** response by sampling interface signals. Converts captured signals into **transaction-level data**. Forwards the data to the **Scoreboard** via a **Mailbox**.

- **Scoreboard**: Checks the **correctness** of the DUT's output. Compares the **actual** response from the **Monitor** with the **expected** results.

Class Based Verification

- **Interface:** Acts as a bridge between the **Testbench** and the **DUT** and Groups multiple signals into a single structured entity, simplifying the connection.
- **Mailbox**: Facilitates **communication** between testbench components. Temporarily stores transactions in system memory, enabling synchronization between components.
- **Transaction:** The Transaction class defines the data structure used in the testbench. Includes fields for generating test stimulus.

UVM Based Verification Block Diagram



UVM Based verification

- **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.

- **Agent (Write Agent and Read 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.

- **Sequencer** : The Sequencer is used to collect the sequence of items in a order and send it to driver.

UVM Based Verification

- **Monitor:** 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**: 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.

- **Interface**: The Interface connected the testbench to the FIFO DUT, defining the signals required for communication.

Challenges and Learnings

Coverage Closure: While achieving 100% code coverage can be relatively straightforward, obtaining functional coverage to a satisfactory level is more challenging. This requires creating numerous bins to ensure that the values are adequately covered during verification. Additionally, analyzing the interactions between different values implemented in the coverage posed some difficulties for us.

Clock Domain: Asynchronous FIFOs naturally involve multiple clock domains for read and write operations. During the design, as well as when creating the monitor and driver classes, we encountered challenges in determining where and which clock domains should be specified for the read and write operations.

Connecting Modules: We faced difficulties while performing UVM verification, particularly in connecting the modules and recalling the built-in class names and method names. This challenge was resolved through guidance from lectures and additional resources available online.

DEMO

- Presenting the project in the action
- Executed within UVM verification Environment
- Running in Questa sim for demonstration
- Bug-injected scenario ready to show

Test Scenarios and Coverage

Code Coverage Metrics = 79.04%

Functional Coverage Metrics = 79.14

Conclusion

- Successful designed and Implemented the Asynchronous FIFO using System verilog.
- Successfully verified the Asynchronous FIFO using Universal Verification methodology (UVM).
- The techniques such as code coverage, functional coverage and thoughtful design implementation helped ensure the correctness of the verification process.
- Universal Verification Methodology (UVM) enhance the efficiency of verification by supporting reusability and scalability.
- Deterministic testing focused on few critical cases and key scenarios whereas random testing focused on identified edge cases, ensuring thorough coverage.
- After using all these techniques we can make sure that the design we satisfies the requirements and it performs reliably across different scenarios.

References

- E. Xie and J. Zhou, "Analysis and Comparison of Asynchronous FIFO and Synchronous FIFO," 2023 IEEE 2nd International Conference on Electrical Engineering, Big Data and Algorithms (EEBDA), Changchun, China, 2023, pp. 260-264, doi: 10.1109/EEBDA56825.2023.10090586. keywords: {Integrated circuits;Electrical engineering;Big Data;Hardware;Registers;Clocks;Comparison;Asynchronous FIFO;Synchronous FIFO},
- https://vlsiverify.com/verilog/verilog-codes/asynchronous-fifo/#google_vignette
- https://electronics.stackexchange.com/questions/650995/verification-of-asynchronous-fifo

Thank you!