Final Project
VERIFICATION TEST PLAN

Fundamentals of Pre-Silicon Validation || Spring -2024

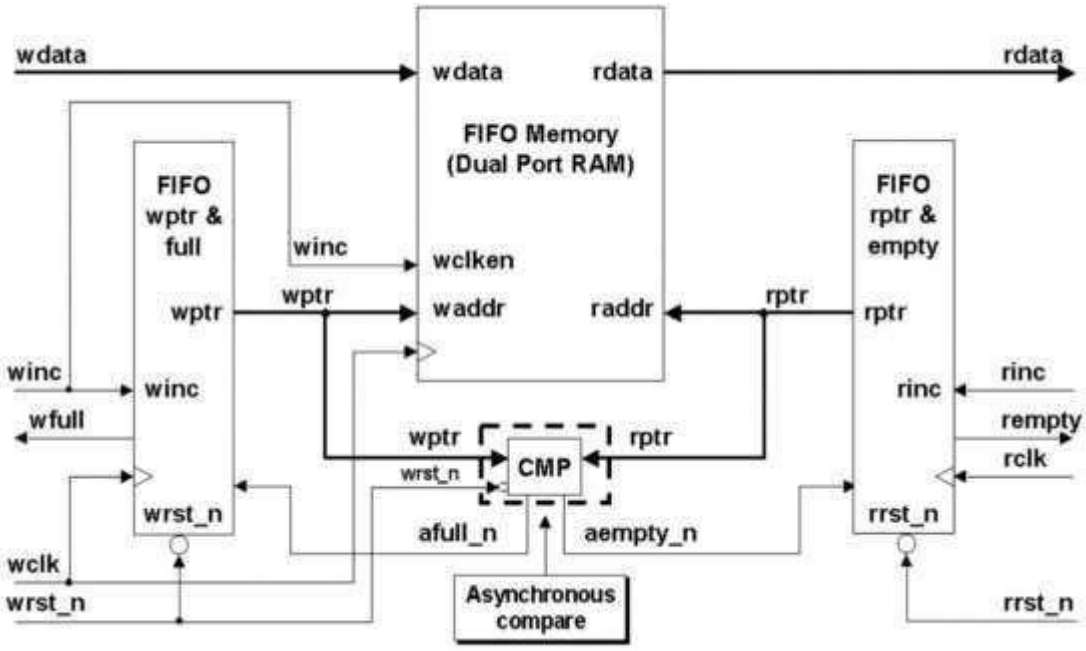
Project Name:
Asynchronous FIFO

Team-14

Sandeep Reddy Lingala	(988094614)
Sai Kumar Reddy Pulagam	(984110186)
Ganesh Kumar Sanke	(947941776)
Reshwanth Sri Sai Sesham	(964563316)

Implementation and Verification of Asynchronous FIFO 1.

Block Diagram



2. Design Description

To properly transfer data from one asynchronous clock domain to another, FIFO is frequently utilized. When data values are put into a FIFO buffer from one clock domain and read from the same FIFO buffer from another clock domain, an asynchronous FIFO design is used, meaning that the two clock domains operate independently of one another.

Always pointing to the next word to be written is the write pointer. The write pointer increases and the empty flag is cleared as soon as the first piece of data is written to the FIFO. There is always a current FIFO word to be read when the read pointer is pointed to.

When there are equal read and write pointers, the FIFO is empty. The FIFO is empty when the read and write pointers are both equal.

3. FIFO Depth Calculation:

Double the frequency of write and half of the read
Sender Clock Frequency = 250MHz
Number of idle cycles between two successive writes = 1
Receiver Clock Frequency = 100MHz
Number of idle cycles between two successive reads = 3
Write Burst = 200
Sender Clock Frequency > Receiver Clock Frequency
The number of idle cycles between two successive writes is 1 clock cycle, which means after writing one data, write module is waiting for 1 clock cycles to initiate the next write, meaning every **2 clock cycles** one data is written.

The number of idle cycles between two successive reads is 3 clock cycles, which means after reading one data, read module is waiting for 4 clock cycles to initiate next read, meaning every **4 clock cycles** on data is read.

Time required to write one data item = 2 * (1/250MHz) = 8 ns.
Time required to write the data in the burst = 1600ns.
Time required to read one data item = 4 * (1/100MHz) = 40ns
So, for every 40ns read module is going to read one data item in the burst.
No of data items can be read in a period of 1600ns = (1600/40) = 40 items. Remaining no

of bytes to be stored in the FIFO = 200 – 40 = 160

4. Verification Levels:

Designer-Level Verification: Designer-Level Verification involves testing of individual modules to ensure functionality aligns with design specifications, focusing on interfaces, functionalities, and boundary conditions. This process is typically conducted using simulation tools to verify module performance.

Unit-Level Verification: Unit-level verification tests the integration of modules within a subsystem, ensuring intended interactions and functionality. It validates communication protocols, data flow, and signal integrity to detect interface mismatches and ensure seamless integration into larger systems.

Sub-Functional Block Verification: Sub-functional block verification assesses larger blocks or subsystems, confirming their intended functionality and interactions within the system. It entails testing complex functionalities like state machines and data paths, ensuring seamless integration and cohesion among system components. This verification level detects integration issues and assures proper interaction between blocks, fostering system-wide functionality.

System-Level Verification: System-level verification evaluates the entire system or SoC against design requirements, validating functionalities like performance, power consumption, and reliability through real-world simulations. It also encompasses testing interactions with external components and interfaces, ensuring seamless integration and functionality across various scenarios.

Integration and Regression Testing: Integration testing verifies seamless integration of system components, while regression testing reassesses functionalities post-design changes. These tests detect integration issues and regressions, ensuring system stability and functionality amid design evolution.

5. Software and Hardware Tools

Questa sim.

Notepad++

EDA Play Ground. Git

hub

6. Verification Strategy:

white box testing: Design under verification is because it ensures all the parts of the design are tested but it is time consuming process and we know the inputs of the design.

Black Box Testing: It enables faster testing and better simulation of real world scenarios but it lacks the technical insights.

Gray Box Testing: It provides early detection of inputs and Improved accuracy but it is time consuming and has some security concerns.

7. Functions to be Verified.

- Data addition to the FIFO involves consecutive writes until the wr_ptr reaches the tail position.
- When wr_ptr reaches the tail, `full is asserted, indicating that the FIFO cannot accept additional data.
- The initial position of wr_ptr is at the head of the FIFO, and upon writing data, it increments and points to the next location.
- If wr_rst is asserted, wr_ptr resets to the head position, allowing whatever data present will be cleared and data to be written again from the initial position.
- The FIFO operates asynchronously, ensuring proper synchronization and seamless write and read operations without discrepancies.
- If rd_ptr is at the head position, empty is asserted, indicating that the FIFO is empty.
- Concurrent read and write operations are supported, maintaining data integrity during simultaneous access.
- Read operations are performed on previously written data, with rd_ptr pointing to the next data item to be read.
- If wr_rst is asserted, wr_ptr resets to the initial position by clearing the data in the FIFO, allowing new read operations.
- Description of each function and why it will not be verified.

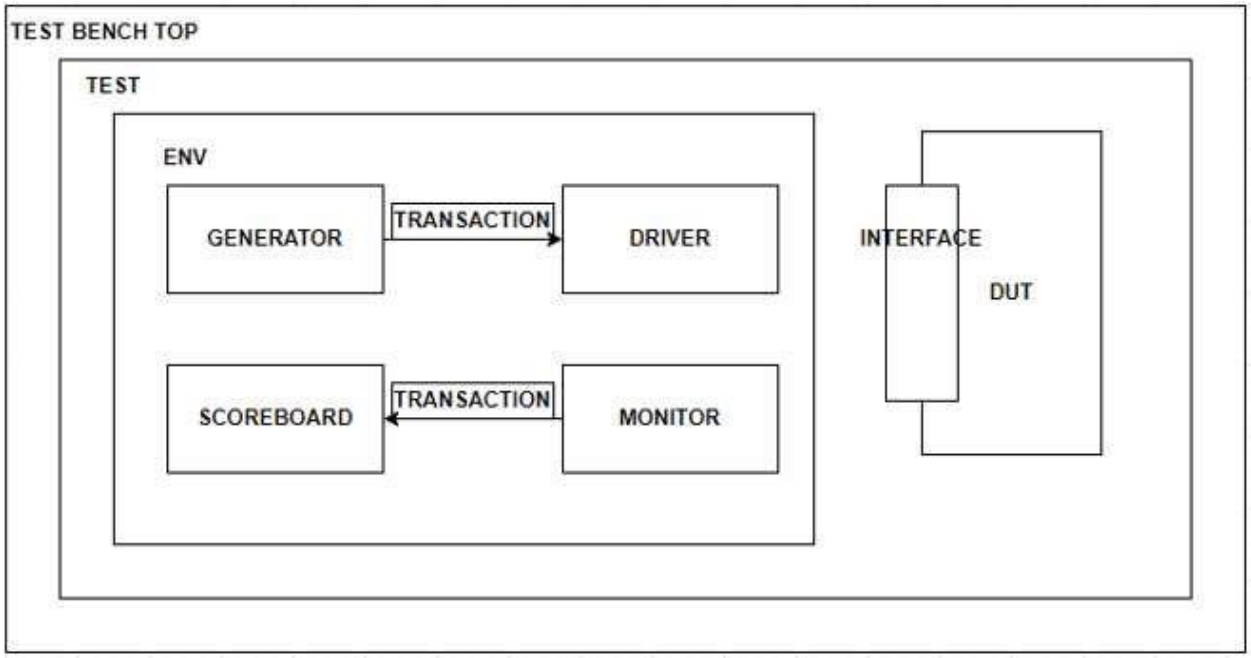
Critical functions:

Write clock domain crossing: Ensuring data integrity when transferring data between clock domains.

Data synchronization: Proper synchronization of data between the write and read sides.

Empty and full detection: Accurate detection of FIFO empty and full states.

8. Test Environment:



Scoreboard:

- The verification process compares expected and actual outputs to ensure the correctness of the Design Under Test (DUT) based on predefined scenarios.
- Utilizes data from the monitor to assess observed DUT outputs against expected values, generating pass/fail results.
- Tracks test case progress, manages results, and flags discrepancies for debugging to maintain test integrity.
-

Monitor

- During simulation, the observer monitors inputs and outputs of the DUT, capturing signals and data transactions.
- It focuses on specific signals or interfaces, logging input/output transactions for analysis and debugging purposes.

- The observer acts as a data collector for the scoreboard, providing real-time visibility into DUT behavior and interactions with the test environment.

Driver

- Converts high-level test scenarios into low-level signal-level details for the DUT.
- Manages the interface between the testbench environment and the DUT, ensuring proper communication.
- Drives stimulus to the DUT based on translated signals and manages protocol-specific details for efficient test case execution. **Generator**

- Develops test scenarios and sequences from defined test cases, ensuring comprehensive coverage of DUT functionality.
- Generates stimulus to test various aspects of the DUT, including operational modes and corner cases, enhancing test coverage.
- Customizes test sequences to stress specific functionalities or features, automating scenario creation to streamline testing and improve efficiency.

Coverage

- Monitors signal and state coverage during simulation, tracking their usage and transitions.
- Evaluates test suite effectiveness by generating coverage reports detailing exercised functionalities.
- Identifies uncovered areas in the DUT's design, guiding refinement of test scenarios to improve coverage and effectiveness.

The test will instantiate an environment object and configure it accordingly. The Testbench top module orchestrates the test environment, handling clock generation (using clocking blocks), reset generation (through driver class), and interface connectivity. Together, these components manage data exchange, randomize input generation for declared rand or randc inputs (e.g., in Asynchronous FIFO), facilitate data transfer, and analyze results.

Test Scenarios

- Generate transactions with different values for each bit of wr_data.
- Ensure that each bin for wrdata_bit0 through wrdata_bit7 is hit.
- Generate transactions with different values for wr_rstn.
- Ensure that each bin for wr_rstn is hit.
- Generate transactions with different values for rd_rstn.
- Ensure that each bin for rd_rstn is hit.
- Generate transactions to fill the FIFO.
- Ensure that both bins for wr_full_bin are hit.
- Generate transactions to empty the FIFO.
- Ensure that both bins for rd_empty_bin is hit.

9. Coverage Requirements

Code Coverage

Code coverage goals are specific targets set to measure the extent to which the source code of a software or hardware design has been exercised during testing. These goals help ensure thorough verification of the design and identify areas of the code that require additional testing or refinement. Here are key aspects to consider when defining code coverage goals:

1. Statement Coverage:

- Achieve 100% statement coverage.
- Ensure every line of code in the DUT is executed at least once during testing.
- Detect syntax errors, uninitialized variables, and dead code that could impact system behavior.

2. Branch Coverage:

- Target high branch coverage (>90%).
- Verify that all decision branches (e.g., if-else statements) are exercised.
- Ensure all possible outcomes of conditional statements are thoroughly tested.

3. Condition Coverage:

- Aim for high condition coverage (>90%).
- Evaluate all Boolean conditions within expressions (e.g., &&, ||) to ensure they evaluate
- to both true and false during testing.
- Identify issues related to logical operators and complex conditions.

4. Path Coverage:

- Achieve high path coverage (>90%).
- Exercise all possible paths through the code, including loops and nested conditionals.
- Provide a comprehensive understanding of the code's behavior and identify complex
- program flow issues.

5. Function and Interface Coverage:

- Validate all functions and interface transactions.
- Confirm that each function is called, and each interface is exercised as per
- specifications.
- Ensure correct behavior and interaction of different components within the DUT.

6. Error Handling Coverage:

- Achieve coverage of error-handling paths.
- Test how the DUT handles unexpected or error-prone scenarios.
- Validate robustness and reliability in error management.

7. Critical Code Path Testing:

- Focus testing efforts on critical code paths and functionalities.
- Mitigate risks and ensure system stability by thoroughly testing essential functionalities.

- Prioritize testing of code that significantly impacts system behavior and performance.

8. Continuous Improvement:

- Implement iterative improvements to achieve higher coverage goals.
- Regularly monitor coverage metrics and adjust testing strategies based on feedback and
- analysis.
- Enhance overall test coverage and validation effectiveness over time.

By setting clear and measurable code coverage goals aligned with project objectives and priorities, verification teams can effectively plan, execute, and assess testing efforts to ensure comprehensive validation of the design's functionality, reliability, and adherence to industry standards. Code coverage goals play a crucial role in improving software quality, reducing defects, and enhancing the reliability of software and hardware systems.

Functional Coverage

Functional coverage goals define specific objectives to ensure that critical functionalities and use-case scenarios of a software or hardware design are thoroughly tested during verification. Functional coverage aims to validate that the design meets functional requirements and behaves as expected under various conditions. Here are key aspects to consider when defining functional coverage goals:

1. Interface Coverage:

- Achieve coverage of all interface transactions and interactions.
- Verify that each interface is exercised with valid and boundary value transactions.
- Ensure correct data exchange and communication between different design
- components.

2. State Coverage (Finite State Machines):

- Achieve coverage of all states and state transitions in finite state machines (FSMs).
- Validate the behavior of the design under different states and transitions.
- Ensure proper functioning and synchronization of state-dependent operations.

3. Protocol Compliance:

- Validate compliance with communication protocols and standards (e.g., USB, Ethernet).
- Ensure that the design conforms to specified protocol behaviors and requirements.
- Verify interoperability and compatibility with other systems or devices.

4. Scenario Coverage:

- Cover specific use-case scenarios and functional behaviors.
- Test critical paths, error conditions, and boundary cases within defined scenarios.
- Validate the design's response and behavior in real-world operational scenarios.

5. Error Handling Coverage:

- Achieve coverage of error-handling paths and exceptional conditions.

- Test how the design handles unexpected or error-prone situations (e.g., timeouts, data
- corruption).
- Validate robustness, reliability, and fault tolerance of error recovery mechanisms.

6. Performance and Load Coverage:

- Validate performance under varying load and stress conditions.
- Test scalability, responsiveness, and resource utilization of the design.
- Identify performance bottlenecks and optimize system efficiency.

7. Functional Safety Coverage:

- Ensure compliance with functional safety standards (e.g., ISO 26262).
- Verify safety-critical functionalities and fault tolerance mechanisms.
- Ensure the design meets safety requirements and mitigates risks of hazards.

8. Compliance with Specifications:

- Validate adherence to functional specifications and requirements.
- Confirm that the design meets specified functional behaviors and performance criteria.
- Ensure product quality, reliability, and customer satisfaction.

9. Edge Case Coverage:

- Test edge cases, boundary conditions, and corner scenarios.
- Validate behavior under extreme or unexpected inputs or conditions.
- Identify and address potential failure modes and unexpected behaviors.

By defining clear and measurable functional coverage goals aligned with project objectives and requirements, verification teams can systematically validate critical aspects of the design and ensure that it meets functional specifications and performance expectations. Functional coverage goals contribute to enhancing software quality, reliability, and compliance with industry standards and customer expectations. Regular monitoring, adaptation, and refinement of functional coverage objectives are essential for achieving thorough verification and validation of the design's functionality.

10. Roles & Responsibilities

S.No.	Task	Responsible person	Comments
1	High Level Design Specification (HLDS)	Sandeep, Sai Kumar, Ganesh, Reshwanth	Design spec documentation

2	Design spec calculation and plan	Sandeep, Sai Kumar, Ganesh, Reshwanth	Depth calculation modules understanding and plan has been made for the design and modules has been splitted
3	FiFo Memory module	Sandeep	System verilog code has been implemented for FIFO mem module
4	Read pointer Empty module	Sai Kumar	System Verilog code has been implemented & uploaded to github.
5	Synchronous Write to Read module	Ganesh	System Verilog code has been implemented & uploaded to github.
6	Synchronous Read to Write module	Reshwanth	System Verilog code has been implemented & uploaded to github.
7	Write pointer Full module	Sandeep, Sai Kumar, Ganesh, Reshwanth	System Verilog code has been implemented & uploaded to github.
8	Basic Testbench	Sandeep, Sai Kumar, Ganesh, Reshwanth	System Verilog code has been implemented & uploaded to github.

Schedule

- **Week 1**: Design has been implemented and compiled successfully. Basic Testbench has been implemented.
- **Week 2:** Generating randomized stimulus, create a class-based testbench, and modify the verification plan to new findings.
- **Week 3:** Complete code coverage and function coverage.
- **Week 4:** Develop UVM testbench.
- **Week 5:** Complete the UVM architecture, UVM environment, and UVM testbench. Finalize documents.

References

1. http://www.sunburst-design.com/papers/CummingsSNUG2002SJ_FIFO1.pdf

2. http://www.sunburst-design.com/papers/CummingsSNUG2002SJ_FIFO2.pdf
3. https://zipcpu.com/blog/2018/07/06/afifo.html
4. https://hardwaregeeksblog.wordpress.com/wpcontent/uploads/2016/12/fifodepthcalculationmadeeasy2.pdf
5. ChatGPT-AI

GITHUB

https://github.com/rsesham18/Asynchronous-FIFO