UVM VERIFICATION TEST PLAN

Fundamentals of Pre-Silicon Validation

Winter -2024

Project Name: Verification of Asynchronous FIFO

Members:

SAI SUKITHA PULI

DEEPTHI CHEVURU

MASAAKI ISHII

Date:

02-25-2024

Acknowledgement:

1 Table of Contents

2 Introduction:

The digital design under verification is an asynchronous FIFO (First-In-First-Out) memory. FIFO is designed to store data asynchronously, allowing data to be written and read concurrently. The verification objectives include ensuring that the FIFO operates correctly under various conditions, such as different read and write speeds, full and empty conditions, and boundary cases using Universal Verification Methodology (UVM). UVM is a widely accepted standard in the world of design verification, which is designed to handle increased complexities and to create a structured environment with components like agents, sequencers, scoreboard. This methodology also gives more flexibility and reusability. All the above ensures to test asynchronous FIFO thoroughly and efficiently.

2.1 Scope and Goals:

- Scope: The verification will cover functional correctness, performance, and robustness of the asynchronous FIFO design.
- Goals:
    1. Validate the functionality of the FIFO under normal operation.
    2. Verify the FIFO behavior under stress conditions, such as full and empty conditions, concurrent read and write operations, and boundary cases.
    3. Ensure proper handling of asynchronous operations and timing constraints.
    4. Achieve high coverage of functional scenarios and corner cases.

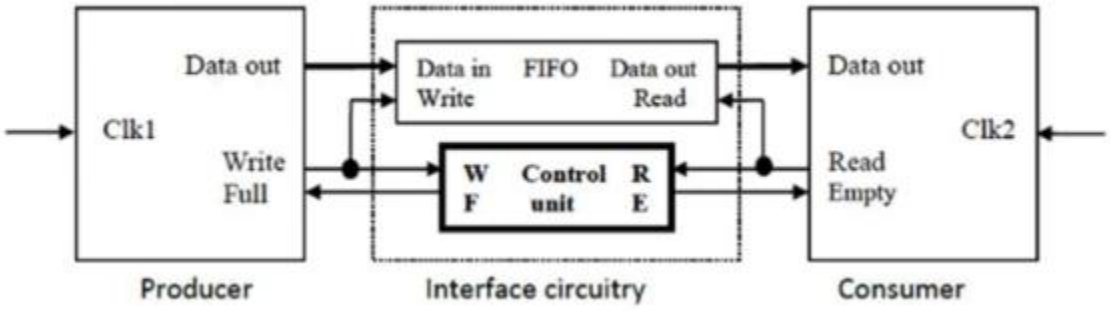
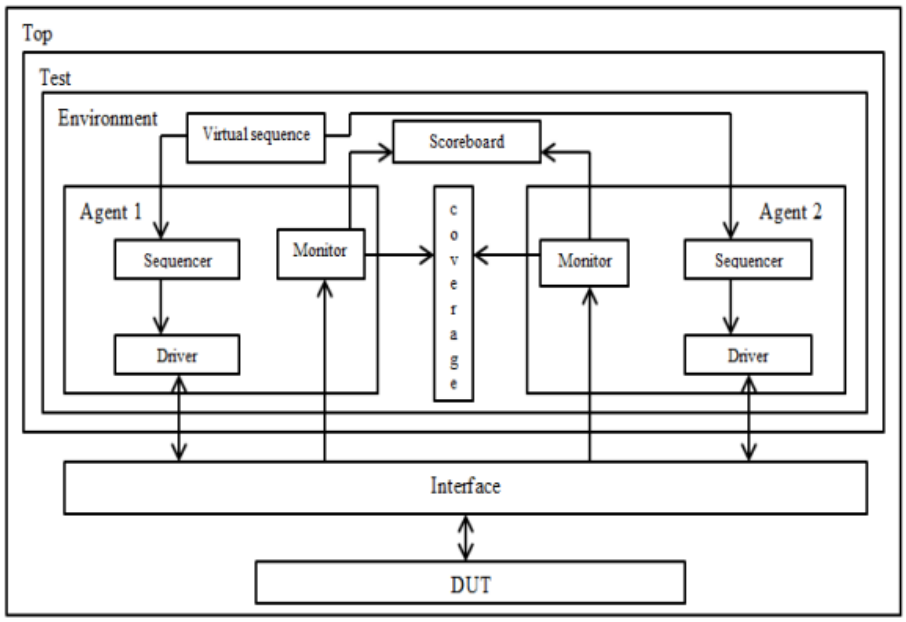


Fig: Asycnhronous FIFO

2.2 UVM based Verification structure:



3 Verification Methodology:

3.1 UVM Framework:

The verification will primarily utilize the Universal Verification Methodology (UVM) framework for building the testbench. Simulation will be the main verification tool, It provides highly reusable and modular verification environments with this approach, components can be easily replaced, updated and resued across various projects leading to significant time saving and consistency. The methodology will involve creating comprehensive test sequences, functional coverage models, and assertions to verify the design's behavior accurately.

3.2 Testbench Architecture:

The UVM testbench architecture is structured and hierarchy consists of the following components:

- Agent: Contains the driver, monitor, sequencer, and scoreboard modules enhancing reusability.
- Interface: Provides the interface between the testbench and the design under test (DUT).

- Sequencers: Responsible for generating and randomizing the stimulus based on specific test scenarios or sequences.
- Drivers: They take the sequences from sequencers and drive them onto the DUT interfaces.
- Monitors: Continuously observe the DUT's interfaces, capturing its responses and behavior.
- Scoreboard: A critical component that compares the DUT's responses with expected results, flagging any discrepancies.
- UVM Tests: Define various test scenarios to be executed on the DUT.
- Environment: Orchestrates the test sequences and manages the testbench components.

3.3 Stimulus Generation:

Generating varied and random stimulus is significant of coverage-driven verification. Our sequences are designed to produce both targeted (for specific test scenarios) and randomized stimulus to ensure the DUT is tested exhaustively.

3.4 Configuration Database:

The UVM configuration database is a powerful tool that allows for hierarchical configuration and customization of UVM components without direct connections. In our testbench, the uvm_config_db is pivotal in passing the virtual interfaces, and other configurations, to the UVM components, ensuring they have the necessary context to interact with the DUT and each other.

4 Verification Components:

4.1 Top Module:

The top module serves as foundation for UVM-based verification of Asynchronous FIFO. Acting as primary module which integrates various UVM components, sequences, agents, environment ensuring a seamless interaction with DUT. It will be developed in the future steps of project verification.

To initiate the verification process, the fifo_sequence_item module is called, triggering the desired UVM test sequence. This method streamlines the entire verification process from generating transactions, interacting with the DUT, monitoring its outputs, and verifying its behavior against expectations.

Upon the next steps of Verification of Asynchronous FIFO, we are expecting to implement according to the architecture in separate modules and integrating them well for reusability.

4.2 FIFO Verification Environment:

The Environment class (fifo_environment) represents the core verification environment for the Asynchronous FIFO. This environment encapsulates all the essential verification components, including agents for both the producer and consumer modules and a scoreboard for results validation. The environment is crucial as it helps in setting up, connecting, and orchestrating the verification process.

4.2.1.1 Key Components:

1. Configuration Handle: This handle provides access to the environment's configuration, which dictates the behavior and structure of the environment. The configuration settings determine which agents or components are active during a simulation run.

2. WR and RD Agents: These agents are responsible for driving and monitoring transactions on the read and write interfaces, respectively. Each agent consists of a driver to send transactions and a monitor to observe the interactions on the FIFO.

3. Scoreboard: The scoreboard is tasked with checking and validating the behavior of the DUT. It receives transactions from the monitors and checks if the observed outputs match the expected results.

4.2.2 Key Methods:

1. Constructor (new): The constructor initializes the environment component, setting its name and parent.

2. Build Phase (build_phase): In this phase, the environment components are instantiated based on the configuration settings. The environment checks the settings to determine which agents and the scoreboard should be created. For instance, if the read agent is enabled in the configuration, the read agent is instantiated. The same logic applies to the write agent and the scoreboard.

3. Connect Phase (connect_phase): This phase is responsible for connecting the various components together. If both the write agent and the scoreboard are enabled, the write monitor's output port gets connected to the scoreboard's write analysis port, allowing the observed write transactions to be sent to the scoreboard for checking. Similarly, the read monitor's output port is connected to the scoreboard's read analysis port if both the read agent and the scoreboard are enabled.

4.2.3 Scoreboard (fifo_scoreboard):

The fifo_scoreboard class is instrumental in the verification flow as it serves the primary function of observing, predicting, and validating the data flow between the producer and consumer modules.

4.2.4 Core Components:

1. FIFOs: These First-In-First-Out structures store the transactions collected from the read and write monitors. They act as buffers, ensuring that the transactions are processed in the order they arrive.

2. Gray Pointers: Variables like wptr, rptr track the total number of spaces in the fifo memory and the number of correctly verified transactions.

3. Covergroup (fifoInterfaceSignalsCG): This captures the functional coverage metrics, giving insight into which parts of the design have been exercised and which have not. The coverage metrics include checking the reset, read, write operations, and transaction types.

4.2.5 Random Test Class (fifo_test):

The fifo_test class is derived from the base test class(uvm_test) and aims to introduce randomized traffic on interfaces. It helps in verifying how the DUT behaves under unpredictable scenarios, adding robustness to the verification process.

4.2.6 Sequencers Core Components:

1. Random Sequences (fifo_sequence): These handles are linked to sequences that generate randomized date on the write and read interfaces respectively.

Primary Methods:

1. Constructor (new): Defines the name and parent for the random test class instance.
2. Build Phase (build_phase):
    - Inherits the build phase of the base test by calling super.build_phase(phase).
    - Instantiates the random sequences for the interfaces.
3. Connect Phase (connect_phase):
    - connects the connect phase of the base test by calling super.uvm_phase(phase).
4. Run Phase(run_phase):
    - Raises an objection to keep the simulation running.
    - Executes the random sequences on their respective sequencers in parallel using the fork...join construct.
    - Drops the objection once the sequences have finished running.
    - Sets a drain time to allow any remaining transactions to be processed before the simulation ends.

4.3 UVM Imports and Macros:

- The Universal Verification Methodology (UVM) package is imported to utilize its standardized features.
- The UVM macros are included to aid in UVM component registration and message reporting.

Expecting to write tests and generate sequences for the below test scenarios in the future development of verification:

- Write Operation: Verify that data can be correctly written into the FIFO.

- Read Operation: Verify that data can be correctly read from the FIFO.
- Full Condition: Verify the behavior when the FIFO is full.
- Empty Condition: Verify the behavior when the FIFO is empty.
- Concurrent Read and Write: Verify the behavior when read and write operations occur simultaneously.
- Boundary Cases: Verify the behavior at the boundary conditions of the FIFO capacity.
- Asynchronous Timing: Verify proper functionality under varying read and write timing constraints.

5 Coverage:

In the field of verification, coverage is a crucial metric that serves as a measure for how thoroughly the design has been tested. It offers information on the aspects of the design that have been adequately validated as well as those that might need more work. As design complexity increases, it becomes more important than ever to make sure that every aspect of the design performs as intended.

5.1 Functional Coverage:

In the context of our UVM-based verification environment, functional coverage is used to measure and ensure that all possible scenarios, states, and transitions of the design have been exercised. The covergroup construct in SystemVerilog is employed to define specific coverage points and bins, which represent different values or ranges of values.

- Coverpoints: These are specific points in the design or the verification environment where we want to measure coverage. In our fifoInterfaceSignalsCG and zeros_or_ones_on_ops, we have defined multiple coverpoints like reset, fifoTBintf.rdata, fifoTBintf.wdata, etc.

- reset coverpoint captures the coverage of the reset signal (wrst_n, rrst_n).
  Specifically, it is interested in the scenario where the reset signal is '0' & '1'.

- wdata and rdata coverpoints focus on the write and read signal, capturing scenarios where write and read operations occur respectively.

- Bins: For every coverpoint, bins are used to partition the value space into subsets. Each bin captures a specific value or a range of values for the associated coverpoint.

- Cross Coverage: This will be implemented across various cover groups and allowing us to get multiple combinations of coverpoints.

5.2 Assertion Coverage:

Assertions are essential to the verification process because they allow particular behaviors or features of a design to be validated. When it comes to the read and write sequences, assertions are mainly used to make sure that particular characteristics of the sequence items are properly randomized according to the intended criteria.

6 Running the Verification Environment:

1. The Do file:

The provided do file is a tool to automate the compilation, simulation, and coverage report generation for the verification environment. Here's a breakdown of its functionality:

- Creates a work library to store all the results and deletes any previous work which is present when simulating other versions or any previous code and updates the work directory.

The run.do file is a script for running a design and UVM testbench called `tb_fifo_uvm` with Questasim.

1. vlib work and del: This command creates a library named "work" in which the compiled design units and testbench will be stored.And deletes any previous work which is present when simulating other versions or any previous code and updates the work directory.
2. The rest files will be compiled in the order of definitions, design, testbench files.
3. vopt +cover=bcesxf tb_fifo_uvm -o test_sm_opt: This command optimizes the compiled design and testbench (`tb_fifo_uvm`) for coverage analysis. It specifies various coverage options (`bcesxf`) to enable different types of coverage metrics such as branch, condition, expression, statement, functional, and toggle coverage. The optimized design is saved as `test_sm_opt`.
4. vsim -coverage test_sm_opt: This command starts Questasim with coverage analysis enabled (`-coverage` option) and loads the optimized design `test_sm_opt` for simulation.
5. add wave -r: This command adds all signals in the design hierarchy to the waveform window for viewing during simulation. The `-r` option recursively adds all signals.
6. run -all: This command initiates the simulation and runs it until completion (`-all` option), executing all test cases and sequences defined in the testbench. During simulation, coverage data is collected for analysis.

Overall, this run file compiles the design and testbench files, optimizes them for coverage analysis, starts the simulation with coverage enabled, adds signals to the waveform viewer, and runs the simulation to collect coverage data.

7 Results and Discussion:

7.1 Debugging the Transaction Print Method:

One of the primary challenges faced during the verification process was the visualization and tracking of transaction information. Despite it being error-free, it didn't provide the desired output. And of giving the transaction's details is done.

Coverage of 80.92% is achieved with basic cover groups but targeting for 100% coverage functional and code coverages under all stress conditions to make sure of the correctness of the design functionality and working.

7.2 Enhancing Transaction Visibility:

With the implementation, combined with the **uvm_info** macro across the testbench components, we achieved a significant enhancement in transaction visibility. This combination allowed us to:

- Display which packet/transaction was generated.
- Monitor the packet's output from the Device Under Test (DUT).
- Track the packet's progression through the verification environment.

This approach provided a dynamic way to observe and analyze transaction behavior in real-time. It also ensured that any irregularities or unexpected behaviors could be promptly identified and addressed.

7.3 Scoreboard Insights:

The increased transaction visibility was very beneficial to the scoreboard, which is a crucial part of our verification environment. The scoreboard could more effectively monitor and validate the data flow between the modules and the fifo with the use of the data and uvm_info.

7.4 Debugging:

This is most time consumed task where Transitioning from a SystemVerilog testbench (from Milestone 3) to the Universal Verification Methodology (UVM).

Our initial approach was to directly translate the SV TB to UVM. However, the complexities arose when tried to incorporate essential UVM features. We encountered various errors in the past week such as compilation errors, simulation errors, and UVM usage errors.

Each error became a learning opportunity. Through rigorous debugging sessions, hours of scrutinizing the UVM manual, and leveraging online communities for insights, we were able to systematically address and rectify each issue. And the verification of the FIFO improves in the coming weeks and makes sure to achieve our targeted correctness using UVM.

8 Conclusion:

This UVM verification plan outlines the objectives, methodology, testbench architecture, test cases, coverage metrics, and sign-off criteria for verifying the asynchronous FIFO design. By following this plan, we aim to ensure the reliability and functionality of FIFO under various operating conditions. This verification plan keeps updating accordingly with the verification of the FIFO and test scenarios.

9 References Uses / Citations/Acknowledgements.

[1] Clifford E. Cummings, “Synthesis and Scripting Techniques for Designing Multi-Asynchronous Clock Designs,” SNUG 2001 (Synopsys Users Group Conference, San Jose, CA, 2001) User Papers, March 2001, Section MC1, 3rd paper.

[2] Clifford E. Cummings and Peter Alfke, “Simulation and Synthesis Techniques for Asynchronous FIFO Design with Asynchronous Pointer Comparisons,” SNUG 2002 (Synopsys Users Group Conference, San Jose, CA, 2002) User Papers, March 2002, Section TB2, 3rd paper

[3] “FIFO (Computing and Electronics).” Wikipedia, Wikimedia Foundation, 2 Sept. 2023, en.wikipedia.org/wiki/FIFO_(computing_and_electronics).

[4] Vlsiverify. (2022, December 26). Asynchronous FIFO. VLSI Verify.https://vlsiverify.com/verilog/verilog-codes/asynchronous-fifo/

[5] https://research.ijcaonline.org/volume86/number11/pxc3893347.pdf

[6] https://hardwaregeeksblog.files.wordpress.com/2016/12/fifodepthcalculationmadeeasy2. pdf

[7] Ramesh, G. (n.d.). Asynchronous FIFO Design with Gray code Pointer for High Speed AMBA AHB Compliant Memory controller. IOSR Journal of VLSI and Signal Processing (IOSR-JVSP), 1(3), 32–37. https://doi.org/www.iosrjournals.org

[8] Chaney, Thomas J. "My Work on All Things Metastable OR Me and My Glitch" (PDF). Archived from the original (PDF) on 2015-12-08. Retrieved 2015-11-05.

[9] https://research.ijcaonline.org/volume86/number11/pxc3893347.pdf

[10] https://www.geeksforgeeks.org/difference-between-black-box-vs-white-vs-grey-box-testing/

[11] https://www.chipverify.com/verification/verification-plan

[12] https://github.com/Ghonimo/Pre_Silicon-AHB-to_APB-Verification

[13] https://cds.cern.ch/record/2232660/files/pdf.pdf

[14] https://github.com/akzare/Async_FIFO_Verification/