

VERIFICATION TEST PLAN
Fundamentals of Pre-Silicon Validation Spring
2024
Implementation and Verification of Asynchronous
FIFO using both Class based and UVM methodologies

Anurag Ranga, Naveen Kumar Reddy Thummala,
Maheshwar Kamalapuram, Yashaswi Katne.

Date:05-14-2024

Table of Contents

Contents

2	Introduction.....	2
2.1	Objective of the verification plan.....	2
2.2	Top Level block diagram.....	2
3	Verification Requirements	2
3.1	Verification Levels	2
4	Required Tools.....	5
4.1	Software and hardware tools.	5
4.2	Directory structure of your runs, what computer resources you will be using	5
5	Risks and Dependencies.....	6
6	Functions to be Verified.....	8
6.1	Functions from specification and implementation	8
7	Tests and Methods	4
8	Coverage Requirements	19
9	Resources requirements	22
9.1	Team members and who is doing what and expertise.	22

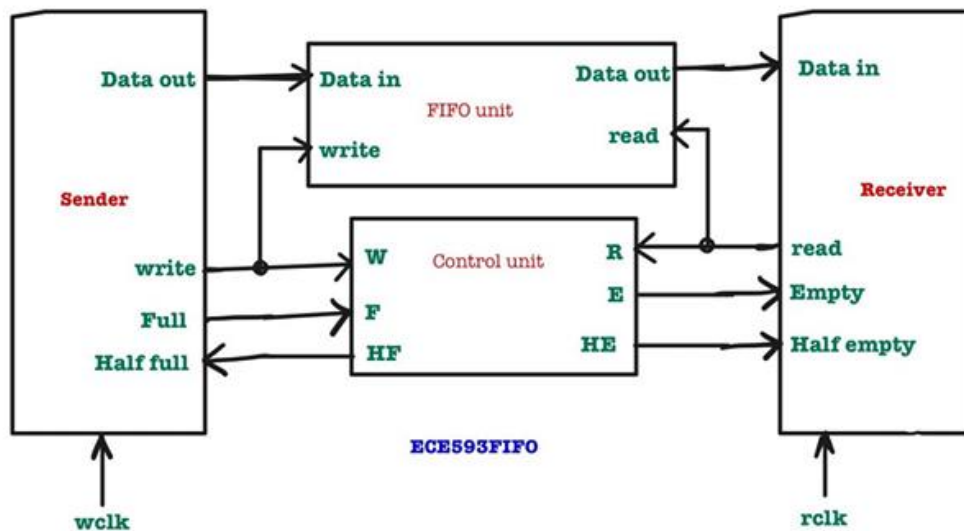
10	Schedule	23
11.	References Uses / Citations/Acknowledgements	24

Introduction:

2.1 Objective of the verification plan

The objective of this verification plan is to ensure the functional correctness and performance compliance of the Asynchronous FIFO design. Specifically, the plan aims to verify a Async FIFO with a minimum depth of 45, with successive reads and writes = 0.

2.2 Top Level block diagram



2.3 Specifications for the design

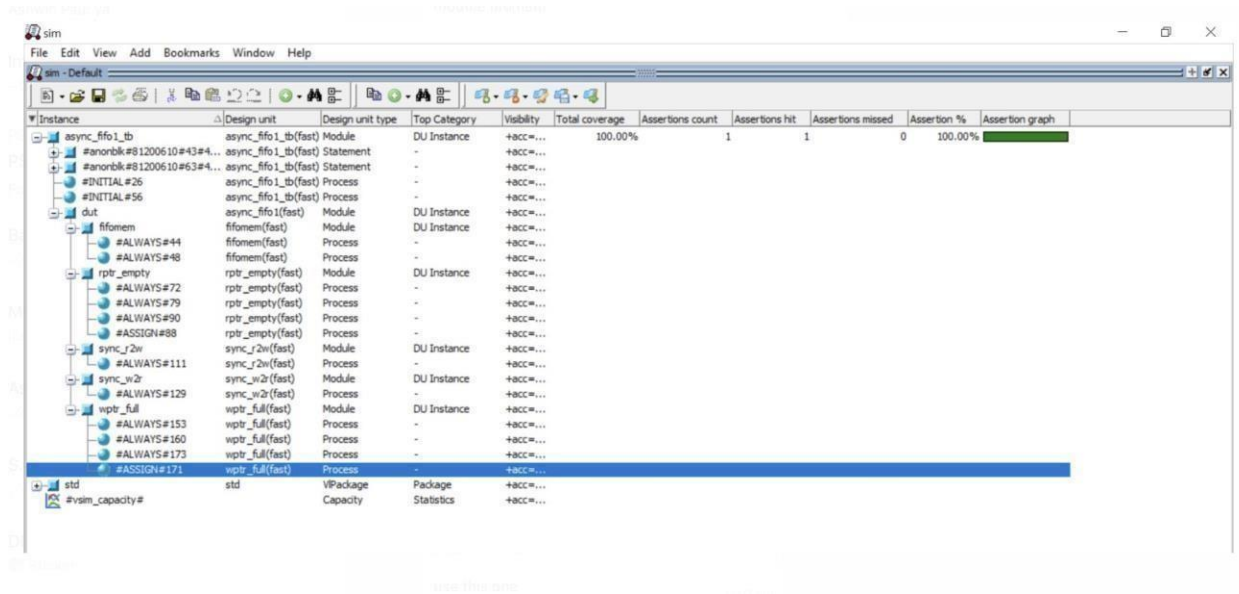
- There are no idle cycles in both reading and writing which means that, all the items in the burst will be written and read in consecutive clock cycles.
- Time required to write one data item = $1 \times (1/80\text{MHz}) = 12.5\text{ns}$
- Time required to write all the data in the burst = $120 \times 12.5\text{ns} = 1500\text{ns}$
- Time required to read one data item = $1 \times (1/50\text{MHz}) = 20\text{ns}$
- So, for every 20 ns, the Receiver is going to read one data item in the burst. In a period of 1500ns, 120 data items can be written.


3 Verification Requirements

3.1 Verification Levels

3.1.1 Module hierarchy

We are verifying the Async FIFO design at the block level as it encapsulates the complete functionality of the FIFO.



Instance	Design unit	Design unit type	Top Category	Visibility	Total coverage	Assertions count	Assertions hit	Assertions missed	Assertion %	Assertion graph
async_fifo_1_tb	async_fifo_1_tb(fast)	Module	DU Instance	+BCC=...	100.00%	1	1	0	100.00%	
#anonblk#81200610#43#4...	async_fifo_1_tb(fast)	Statement	-	+BCC=...						
#anonblk#81200610#63#4...	async_fifo_1_tb(fast)	Statement	-	+BCC=...						
#INITIAL#26	async_fifo_1_tb(fast)	Process	-	+BCC=...						
#INITIAL#56	async_fifo_1_tb(fast)	Process	-	+BCC=...						
dut	async_fifo_1(fast)	Module	DU Instance	+BCC=...						
ffomem	ffomem(fast)	Module	DU Instance	+BCC=...						
#ALWAYS#44	ffomem(fast)	Process	-	+BCC=...						
#ALWAYS#48	ffomem(fast)	Process	-	+BCC=...						
rptr_empty	rptr_empty(fast)	Module	DU Instance	+BCC=...						
#ALWAYS#72	rptr_empty(fast)	Process	-	+BCC=...						
#ALWAYS#79	rptr_empty(fast)	Process	-	+BCC=...						
#ALWAYS#90	rptr_empty(fast)	Process	-	+BCC=...						
#ASSIGN#88	rptr_empty(fast)	Process	-	+BCC=...						
sync_2w	sync_2w(fast)	Module	DU Instance	+BCC=...						
#ALWAYS#111	sync_2w(fast)	Process	-	+BCC=...						
sync_w2r	sync_w2r(fast)	Module	DU Instance	+BCC=...						
#ALWAYS#129	sync_w2r(fast)	Process	-	+BCC=...						
wptr_full	wptr_full(fast)	Module	DU Instance	+BCC=...						
#ALWAYS#153	wptr_full(fast)	Process	-	+BCC=...						
#ALWAYS#160	wptr_full(fast)	Process	-	+BCC=...						
#ALWAYS#173	wptr_full(fast)	Process	-	+BCC=...						
#ASSIGN#171	wptr_full(fast)	Process	-	+BCC=...						
std	std	VPackage	Package	+BCC=...						
#vsim_capacity#		Capacity	Statistics	+BCC=...						

3.1.2 Controllability and Observability

Controllability and observability are achieved through dedicated input and output ports of the FIFO module, allowing effective stimulus application and result monitoring.

3.1.3 Interface signals

Parameters:

- Data Size (DSIZE) = 8
- Address Size (ASIZE) = 4

Testbench Signals:

- wdata: Data to be written
- winc: Write enable
- welk: Write clock
- wrst_n: Write reset
- rinc: Read enable
- rclk: Read clock
- rrst_n: Read reset
- rdata: Data read from the FIFO
- wfull: Write FIFO full indicator
- rempty: Read FIFO empty indicator

Clock Periods:

- Write Clock Period (CLK_PERIOD_WR) = 12.5 ns
- ReadClock Period (CLK_PERIOD_RD) = 20 ns

Test Sequence:

1. Initialize all signals.
2. Reset the write and read processes.
3. Write 120 items into the FIFO.
4. Wait for some cycles.
5. Read 120 items from the FIFO.
6. Finish simulation.

Verification Environment:

- Testbench instantiates the fifo2 module.
- Clocks wclk and rclk are generated with their respective periods.
- Test sequence writes and reads data to/from the FIFO.
- Monitor displays key signals

FIFO Module Interface:**Definition:****Memory Interface (FIFO mem):**

Definition: The interface between the FIFO module and the memory subsystem (FIFO mem).

Specifications:

winc: Write enable signal.

wfull: Input signal indicating FIFO full.

wclk: Write clock signal.

waddr: Write address bus.

raddr: Read address bus.

wdata: Input data bus for write operations.

rdata: Output data bus for read operations.

Read Pointer Empty Module Interface (rp_ptr_empty):

Definition: The interface of the module handling read pointer and empty condition.

Specifications:

rinc: Read enable signal.

rclk: Read clock signal

rrst_n: Active-low read reset signal.

rq2_wptr: Input read pointer with write side.

rempty : Output signal indicating FIFO empty condition.

raddr: Output read address.

Write Pointer Full Module Interface (wp_ptr_full):

Definition: The interface of the module handling write pointer and full condition.

Specifications:

winc: Write enable signal.

wfull: Output signal indicating FIFO full condition.

wclk: Write clock signal.

wq2_rptr: Input read pointer with write side.

waddr: Output write address.

wptr: Output write pointer.

Synchronization Modules (sync_r2w, sync_w2r):

Definition: Modules ensuring synchronization between read and write

Specifications:

wclk, wrst_n: Write clock and reset signals.

rptr: Read pointer.

rq2_wptr, wq2_rptr: Synchronized read and write pointers.

4. Required Tools

3.2 Software and hardware tools.

Mentor Questa sim.

3.3 Directory structure of your runs, what computer resources you will be using.

The directory structure for the verification runs will be organized for clarity and ease of access. Below is a proposed directory structure:

verification_plan

/testbench

- async_fifo1_tb.sv

/modules

- async_fifo1.sv - fifomem.sv

- rptr_empty.sv
- wptr_full.sv
- sync_r2w.sv
- sync_w2r.sv

6. Risks and Dependencies

Incomplete Specifications:

Risk: Incomplete or ambiguous specifications may lead to misinterpretations during verification.

Contingency/Mitigation: Regularly communicate with design and specification teams to clarify any ambiguities. Document assumptions made during verification.

Concurrency Issues:

Risk: Concurrency-related problems may arise during read and write operations.

Contingency/Mitigation: Thoroughly test scenarios involving concurrent read and write operations to identify and address any concurrency issues.

Performance Bottlenecks:

Risk: Performance issues, such as slow read or write operations, may impact the overall functionality.

Contingency/Mitigation: Implement performance-related tests to ensure compliance with specifications. Optimize design if performance bottlenecks are identified.

Integration Challenges:

Risk: Integration issues may arise when connecting the FIFO design with other system components.

Contingency/Mitigation: Conduct integration testing with other modules early in the verification process. Collaborate with other teams to address interface compatibility.

Assertion Failures:

Risk: Assertions may fail to capture specific corner cases or may trigger false positives.

Contingency/Mitigation: Regularly review and update assertions. Use multiple assertion types to cover various aspects of the design.

Incomplete Test Coverage:

Risk: Insufficient test coverage may lead to undetected bugs or corner cases.

Contingency/Mitigation: Develop a comprehensive test plan with a variety of scenarios to achieve highest coverage. Utilize code coverage tools to identify areas that need additional testing.

Misalignment with Design Changes:

Risk: Changes in the design may not be accurately reflected in the testbench or test cases.

Contingency/Mitigation: Establish a robust change management process. Regularly synchronize the testbench and test cases with the latest design specifications.

Resource Constraints:

Risk: Insufficient computing resources may impact the efficiency of simulations.

Contingency/Mitigation: Optimize testbenches for efficiency. Use parallel simulation options if available. Upgrade hardware resources if necessary.

Unrealistic Test Scenarios:

Risk: Test scenarios may not accurately represent real-world usage.

Contingency/Mitigation: Collaborate with domain experts to create realistic test scenarios. Incorporate feedback from system architects.

Human Error:

Risk: Mistakes in testbench development, test case creation, or result analysis.

Contingency/Mitigation: Implement thorough code reviews, use automation where possible, and encourage a culture of careful verification practices.

6. Functions to be Verified.

3.4 Functions from specification and implementation

6.1.1 List of functions that to be verified.

Write Operation:

Function: Verify that data can be successfully written into the FIFO.

Description: Ensure that the FIFO accepts data when the write enable signal is asserted. Check if the data is correctly stored in the memory.

Read Operation:

Function: Verify that data can be successfully read from the FIFO.

Description: Ensure that the FIFO provides valid data when the read enable signal is asserted. Check if the data read matches the expected values.

Write and Read Concurrency:

Function: Verify simultaneous write and read operations.

Description: Test scenarios where write and read operations occur concurrently. Ensure that the FIFO handles simultaneous read and write requests without data corruption.

FIFO Full Condition:

Function: Verify the FIFO full condition.

Description: Write data into the FIFO until it reaches its maximum depth. Verify that the FIFO signals a full condition correctly.

FIFO Empty Condition:

Function: Verify the FIFO empty condition.

Description: Read data from the FIFO until it becomes empty. Verify that the FIFO signals an empty condition correctly.

Idle Write Cycles:

Function: Verify the behavior during idle write cycles.

Description: Confirm that the FIFO remains stable during idle write cycles (when no data is being written). Check for any unintended side effects during idle write periods.

Idle Read Cycles:

Function: Verify the behavior during idle read cycles.

Description: Confirm that the FIFO remains stable during idle read cycles (when no data is being read). Check for any unintended side effects during idle read periods.

Asynchronous Write and Read:

Function: Verify asynchronous write and read operations.

Description: Introduce variations in write and read timing to ensure that the FIFO can handle asynchronous operations.

Tests and Methods

7.1.1 Testing methods to be used: Black/White/Gray Box.

Black Box Testing: Functional verification based on specifications.

White Box Testing: Code coverage analysis and assertion-based verification.

Gray Box Testing: Scenario-based testing for corner cases.

7.1.2 PROs and CONs.

Black Box Testing: PRO - High-level coverage. CON - Limited visibility into internals.

White Box Testing: PRO - In-depth analysis. CON - May miss system-level issues.

Gray Box Testing: PRO - Comprehensive testing. CON - Increased simulation time.

7.1.3 Testbench Architecture; Component used (list and describe Drivers, Monitors, scoreboards, checkers etc.)

The testbench module **async_fifo1_tb** initializes and drives signals to the DUT and checks the output against expected results. To relate this code to the image, we could map the code components to the diagram as follows:

- **Sequencer:** In the testbench (**async_fifo1_tb**), the part where random data is generated (**\$urandom**) and controlled by **winc** can be seen as the sequencer.
- **Driver:** The signals **winc**, **wdata**, **wclk**, and **wrst_n** driven from the testbench to the DUT represent the driver.
- **Monitor:** The part of the testbench that reads the output **rdata** and compares it with the expected data (**verif_wdata**) acts as the monitor.
- **Scoreboard:** The array **verif_data_q** and the assertions checking the DUT's output (**rdata**) against the expected output (**verif_wdata**) can be considered as the scoreboard mechanism.

- **Interface:** This is implicit in the testbench, as the signals driven to the DUT and the signals read from it act as the interface.

7.1.4 Verification Strategy: (Dynamic Simulation, Formal Simulation, Emulation etc.) Describe why you chose the strategy.

Verification Strategy: Dynamic Simulation.

Reasoning: Dynamic simulation balances accuracy and efficiency for FIFO verification.

8. Coverage Requirements

8.1.1.1 Describe Code and Functional Coverage goals for the DUT.

1. **Bit Coverage of wdata:**
 - Verify that each bit of **bus_tb.wdata** toggles between 0 and 1.
 - Ensure that each bit is exercised individually to catch any potential stuck-at faults or other issues.
2. **Control Signals:**
 - **Read and Write Increment Signals (rinc, winc):**
 - Verify that these signals toggle correctly during read and write operations.
 - **Reset Signals (wrst_n, rrst_n):**
 - Ensure that the reset signals transition properly from active to inactive states and viceversa.
 - Confirm that the DUT resets and initializes as expected.
3. **FIFO Full and Empty Signals:**
 - **wfull Signal:**
 - Validate that the FIFO full signal toggles correctly when the FIFO reaches its maximum capacity.
 - **rempty Signal:**
 - Ensure that the FIFO empty signal toggles correctly when the FIFO is empty and ready to accept new data.
 - Test scenarios where the FIFO transitions between empty, partially full, and full states.
4. **Specific Data Patterns in wdata:**
 - **One-Hot Values (wdata_onehot):**
 - Confirm that each one-hot value in the **wdata** bus is exercised during simulation.
 - **Border Values (wdata_border):**
 - Validate the behavior of the FIFO when border values are encountered, such as all 0s or all 1s.
 - Ensure that data patterns are correctly recognized and handled by the DUT.
5. **Cross Coverage:**
 - **wfull and rempty Cross Coverage:**
 - Create cross coverage between **wfull** and **rempty** to ensure correct behavior when the FIFO is transitioning between full and empty states.

- Explore interactions between control signals (e.g., **wrst_n**, **rrst_n**) and specific datapatterns (e.g., **wdata_onehot**) to verify robustness across various scenarios.

6. Functional Coverage:

- **Read and Write Operations:**
 - Ensure that all read and write operations are exercised, including edge cases and corner scenarios.
- **Data Integrity and Error Handling:**
 - Validate that data integrity is maintained throughout read and write operations.
 - Test error handling mechanisms, such as overflow and underflow conditions.

Coverage Requirements

Coverage analysis will mainly focus on writing covergroups and coverpoints to verify all the testcases that are being proposed. By doing so, we are just re-affirming that all the necessary checks are not being missed. Further, it also helps to debug if any of the test cases fail, or the verification effort is stuck at some phase. Also, coverage will be written to verify corner/interesting cases that come up once we start working on this project.

To assess the verification closure, a coverage model will be implemented to determine that the DUV has been exposed to a satisfactory variety of stimulus.

Code Coverage

In this model, we will come up with the code coverage collection for the various coverage types as mentioned below with the support of automated code coverage report from the Mentor Questa EDA tool.

- Statement Coverage
- Branch Coverage
- Conditional Coverage
- Toggle Coverage

Functional Coverage

Functional coverage specifies some values to observe at certain times in a design or testbench and counts how many times those values occur. Using functional coverage requires a detailed verification plan and much time creating the cover groups, analyzing the results, and modifying tests to create the proper stimulus.

We are planning to achieve a functional coverage goal of 98% for each level of hierarchies of verification mentioned in the verification strategy and finally achieving overall coverage of 100% at the end of the project. (Currently we are stuck at 82%, we are rushing against deadline to at least push it above 90%). Update: We have reached 99% code coverage now.

Instances

Design Units

Search...

async_fifo_top (95.68%)

DUT (99.87%)

fifo_mem_inst

synchronizer_r2w_inst

synchronizer_w2r_inst

rprr_handler_inst

wptr_handler_inst

Questa Coverage Report Summary

Instance Coverage Summary (95.68%)

Coverage Type ↑

Bins

Hits

Misses

Cover

Search...

Search...

Search...

Search...

Search...

Branches

18

18

0

10

Conditions

2

2

0

10

Covergroups

1

na

na

78.9

Coverpoints/Crosses

17

na

na

Covergroup Bins

71

49

22

69.0

Statements

45

45

0

10

Toggles

392

390

2

99.4

Design Units Coverage Summary (95.67%)

Coverage Type ↑

Bins

Hits

Misses

Cover

Search...

Search...

Search...

Search...

Search...

Branches

16

16

0

10

Conditions

2

2

0

10

Covergroups

1

na

na

78.9

Coverpoints/Crosses

17

na

na

Covergroup Bins

71

49

22

69.0

Statements

40

40

0

10

Toggles

364

362

2

99.4

Instances

Design Units

Search...

async_fifo_top (95.68%)

DUT (99.87%)

fifo_mem_inst

synchronizer_r2w_inst

synchronizer_w2r_inst

rprr_handler_inst

wptr_handler_inst

Language

Verilog

Source File

async_fifo_top.sv

Coverage Summary By Instance (99.87%)

Instance ↑

Branches

Conditions

Statements

Toggles

Total

Search...

Search...

Search...

Search...

Search...

Total

100%

100%

100%

99.48%

99.87%

DUT

-

-

-

98.52%

98.52%

fifo_mem_inst

100%

100%

100%

100%

100%

rprr_handler_inst

100%

-

100%

100%

100%

synchronizer_r2w_inst

100%

-

100%

100%

100%

synchronizer_w2r_inst

100%

-

100%

100%

100%

wptr_handler_inst

100%

-

100%

100%

100%

Local Instance Coverage Details (98.52%)

Coverage Type ↑

Bins

Hits

Misses

Cover

Search...

Search...

Search...

Search...

Search...

Toggles

136

134

2

98.4

Recursive Hierarchical Coverage Details (99.87%)

Coverage Type ↑

Bins

Hits

Misses

Cover

Search...

Search...

Search...

Search...

Search...

Branches

18

18

0

10

Conditions

2

2

0

10

Statements

45

45

0

10

Toggles

392

390

2

99.4

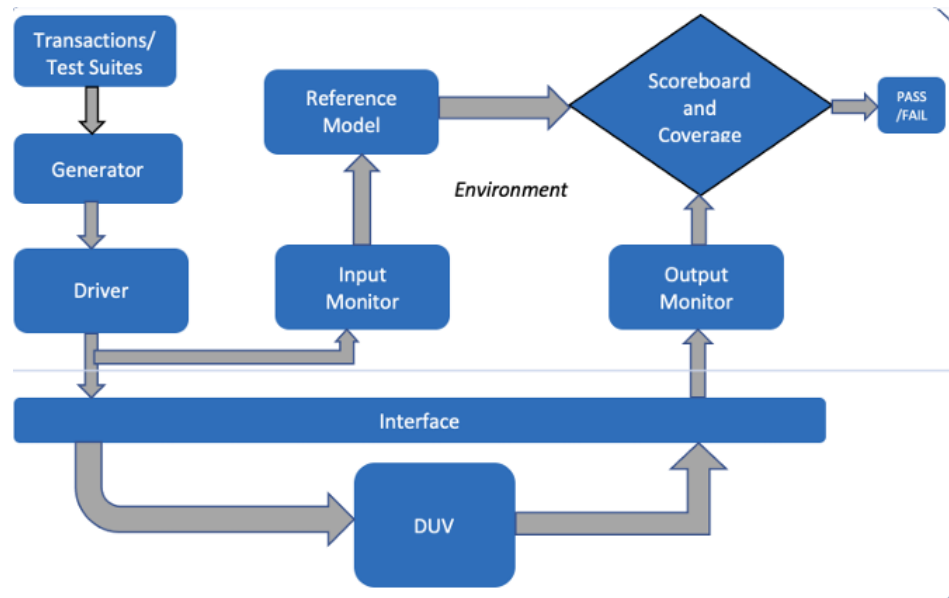
This coverage model creates and implements a following key features at the core-level verification and block-level verification as described below:

- Verify that every instruction reads or writes to all its legal operand.
- The total number of instructions being executed on the DUV.
- The type of instructions being executed on the DUV
- The coverage of 10-bit address space through the program counter (PC) register.
- Cross Coverage for back-to-back consecutive instructions in the pipeline to observe the dependencies between the instructions.
- Permutations for the various opcodes.
- Cross coverage for an opcode with 'funct' field.
- Cross coverage for an opcode with 'shamt' field.

Coverpoints and covergroups are added accordingly throughout the project for improvisation.

Verification Environment:

For this project we are approaching SystemVerilog class-based constrained randomized reusable and reconfigurable testbench environment for the verification of different levels of hierarchies in the microarchitecture.



Interface: The interface provides the port-level connection to the DUV and the testbench. We are planning to include the signals into interface from the core-level, block-level, and unit-level. The internal signals from the block-level and unit-level will be used for the monitoring purpose. Transactions: This class creates the stimulus for the signals that need to be constrained and randomized, along with post-randomization methods if there is a necessary case that need to monitor the overriding of a randomized stimuli. We will develop multiple data-items/transactions for verifying multiple cases at multiple blocks and core-level of verification of CPU.

Generator: The test transactions are provided to the Driver component through the generator, where in generator creates the object for a data-item and will be transferred to driver through mailbox.

Driver: The data items that are got from mailbox, are driven to Design-under-verification through the physical interface. Hence, before driving the stimulus to DUV, the signals that are encapsulated in the object need to be converted to pin-level and then drove to DUV.

Input Monitor: This component observes the signal changes happening at the DUV and are taken as inputs to the reference model.

Output Monitor: This component collects the output from the DUV and transfers the object into the scoreboard for comparison with the output from the reference model.

Reference Model: This component duplicates the functionality of DUV in the higher level of abstraction, and the signal changes happening at the DUV and are taken as inputs for this component.

Scoreboard: This component collects the observed output from the output monitor and expected output from the reference model and compares the results to achieve and verify the progress in the functional verification of the

DUV. I. II. III.

The testbench will be architected in such a way that, the components will be reusable and configurable for a lower-level verification in the hierarchy, i.e, unit/block level verification, and the same testbench would be reusable for the core-level of verification.

For this development, we are planning to reuse the components like generator, input and output monitors, reference model and scoreboard. Hence, only the driver which can't be reused will be reconstructed for various levels and its corresponding units. We are planning to implement and utilize the polymorphism, abstract classes and callback features of OOP, whereby we inherit the multiple drivers for the various blocks as well as for core-level from the base driver class and develop a functionality in accordance with the particular block.

Additional Considerations:

- **Randomization:** Ensure that the test sequences incorporate randomization to cover a widerange of scenarios.
- **Functional Coverage:** Include bins that cover functional aspects of the DUT's behavior, such asread and write operations, error handling, and boundary conditions.
- **Corner Cases:** Define bins that cover corner cases and extreme values to validate the DUT'sbehavior under adverse condition.

8.1.2 Assertions

8.1.2.1 Describe the assertions that you are planning to use and how it will help you improve the overallcoverage and functional aspects of the design.

Division of Tasks among Team Members:

This division of tasks between team members is tentative only. In reality everyone gets involved in the project in some way or the other. The responsibilities are likely to change as the project progresses and tasks are given.

Milestone-1:

1a. Anurag Ranga : Complete design specifications document with calculations for DUT.

1b.Maheshwar Kamalapuram: Verification Plan document in place with initial information. Task division and schedule included.

1c.Yashaswi Katne: Implementation of design and successful compilation.

1d.Naveen Kumar Reddy . T: Develop a simple conventional testbench to check basic functionality.

Milestone-2:

2a. Yashaswi Katne: Develop Class-Based testbench. All interfaces completed and tested.

2b.Naveen Kumar Reddy . T: Complete transactions, Generator, Drivers, and other components.

2c. Maheshwar Kamalapuram: Verify at least 20-50 randomized bursts of data.

2d. Anurag Ranga: Update Verification Plan with more detailed updates.

Milestone-3:

3a.Anurag: Finalize any changes in RTL for any updates needed.

3b. Yashaswi Katne and Anurag: Complete the class-based verification. All components must be defined and working (Transaction, Generator, Driver, Monitors, Scoreboard, and Coverage).

3c.Maheshwar Kamalapuram: Include both code-coverage and functional coverage reports.

3d. Naveen Kumar Reddy T: Update Verification Plan with more detailed test cases if any more

are added.

Milestone-4:

4a.Anurag: Develop UVM testbench, starting with UVM TB architecture.

4b.Yashasvi Katne: Add a section for UVM verification Plan and add details on UVM architecture, UVM hierarchy, UVM components (sequence, sequencer, driver, monitor, scoreboard, interfaces with DUT, number of agents planned, etc.).

4c.Naveen Kumar Reddy T and Maheshwar : Utilize UVM_MESSAGING, UVM_LOGGING mechanisms to create and log the reports and data

Milestone-5 (Final Deliverables + Presentations):

5a.Yashasvi Katne: Complete the UVM architecture, UVM environment, and UVMtestbench.

5b.Naveen Kumar Reddy T: Complete all test cases and reflect them in the coverage reports.

5c.Maheshwar Kamalapuram: Create scenarios of bug injection and verify. Show your work.

5d.Anurag Ranga: Finalize the documents, paper, and presentations. Update Design Specification document, Verification Plan documents with all the updated and latest data

Future Scope: Until now, our team has done the calculations, DUT and a conventional Testbench. Now we are trying an automated self-checking testbench. We will prepare for Milestone-3 from now-on and improve our approach based on the feedback given after Milestone-1 and 2.

4 Schedule

- Weeks 1-2 (Current Date to April 22, 2024): Focus on developing the HLDS. This includes finalizing specifications and planning out the design and verification strategy.
- Weeks 3-5 (May 2 to May 10, 2024): Begin the implementation of the Async FIFO in System Verilog. Ensure that the design meets the requirements set out in the HLDS.

- Weeks 6-7 (May 11 to May 20, 2024): Develop the initial conventional testbench and begin basic testing of the design.
- Weeks 8-9 (May 21 to May 28, 2024): Expand into functional verification using UVM. This includes developing a more sophisticated UVM testbench and achieving the desired coverage goals.
- Week 10 (May 28 to June 2, 2024): Perform final testing and validation of the design. Address any bugs or issues found during testing.

GitHub Link : [sashakatne/Team3_AsyncFIFO_S24_ECE593 \(github.com\)](https://github.com/sashakatne/Team3_AsyncFIFO_S24_ECE593)

11. References Uses / Citations/Acknowledgements

1. S. Cummings, "FIFOs: Fast, predictable, and deep," in Proceedings of SNUG, 2002. [Online]. Available: http://www.sunburst-design.com/papers/CummingsSNUG2002SJ_FIFO1.pdf.
2. S. Cummings, "FIFOs: Fast, predictable, and deep (Part II)," in Proceedings of SNUG, 2002. [Online]. Available: http://www.sunburst-design.com/papers/CummingsSNUG2002SJ_FIFO2.pdf.
3. Putta Satish, "FIFO Depth Calculation Made Easy," [Online]. Available: <https://hardwaregeeksblog.files.wordpress.com/2016/12/fifodepthcalculationmadeeasy2.pdf>.
4. A. Author et al., "Title of the Paper," in Proceedings of the Conference, 2015, pp. 123-456. [Online]. Available: <https://ieeexplore.ieee.org/abstract/document/7237325>.
5. M. Last Name et al., "Designing Asynchronous FIFO," [Online]. Available: <https://d1wqtxts1xzle7.cloudfront.net/56108360/EC109-libre.pdf>.
6. A. Author et al., "Title of the Paper," in Proceedings of the Conference, 2011, pp. 789-012. [Online]. Available: <https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6041338>
7. Author, "Title of the Video," [Online]. Available: <https://www.youtube.com/watch?v=UNoCDY3pFh0>
8. Open AI, "Chat GPT," [Online].