Fundamentals of Pre-Silicon Validation Winter-2025

Design and Verification of Asynchronous FIFO using both Class based and UVM methodologies.

VERIFICATION TEST PLAN

Team -1

Bhargav Chunduri - bhargavc@pdx.edu

Dhushyanth Dharmavarapu – dharmava@pdx.edu

Venkata Krishna Kumar vedantam – vedantam@pdx.edu

CONTENTS **Page No**

1. Introduction

1.1 Purpose of the verification:

The purpose of the verification plan for the asynchronous FIFO design is to methodically assess its functionality, performance, and reliability. The key objectives include:

Functional Verification

- Ensuring the FIFO correctly follows the First-In-First-Out (FIFO) data movement principle
- Validating the proper functioning of FIFO full and FIFO empty conditions.
- Checking that the read and write pointers operate correctly and meet the required conditions.

Asynchronous Operation

- Ensuring data integrity between the write and read domains.
- Verifying that the design supports asynchronous data transfer, accommodating different clock frequencies between the read and write interfaces. Metastability Condition
- Assessing the design's ability to handle metastability issues caused by asynchronous inputs and confirming the implementation of proper synchronization techniques.

Cross-Clock Domain Signals

- Detecting and verifying signals that transition between clock domains, including handshaking signals and FIFO status flags..

The verification plan aims to achieve comprehensive coverage of the asynchronous FIFO's functionality, ensuring a high level of confidence in its accuracy, reliability, and performance across various usage scenarios.

1.2 Specifications for the design

Sender Clock Frequency = 500MHZ = fc

Write Idle Cycles = 2 (Data will be written on every consecutive clock cycle)

Write Burst Size = 1024

- Receive Clock Frequency = 225 MHz = fB

  Read Idle Cycles = 1 (A single data item is read every three clock cycles.)

  Since, fc > fB and

  Given Burst Length = 1024

- The number of idle cycles between two consecutive writes is 2 clock cycles. This means that after writing one data item, module A waits for two clock cycles before initiating the next write. Therefore, it can be understood that one data is written every three clock cycles.

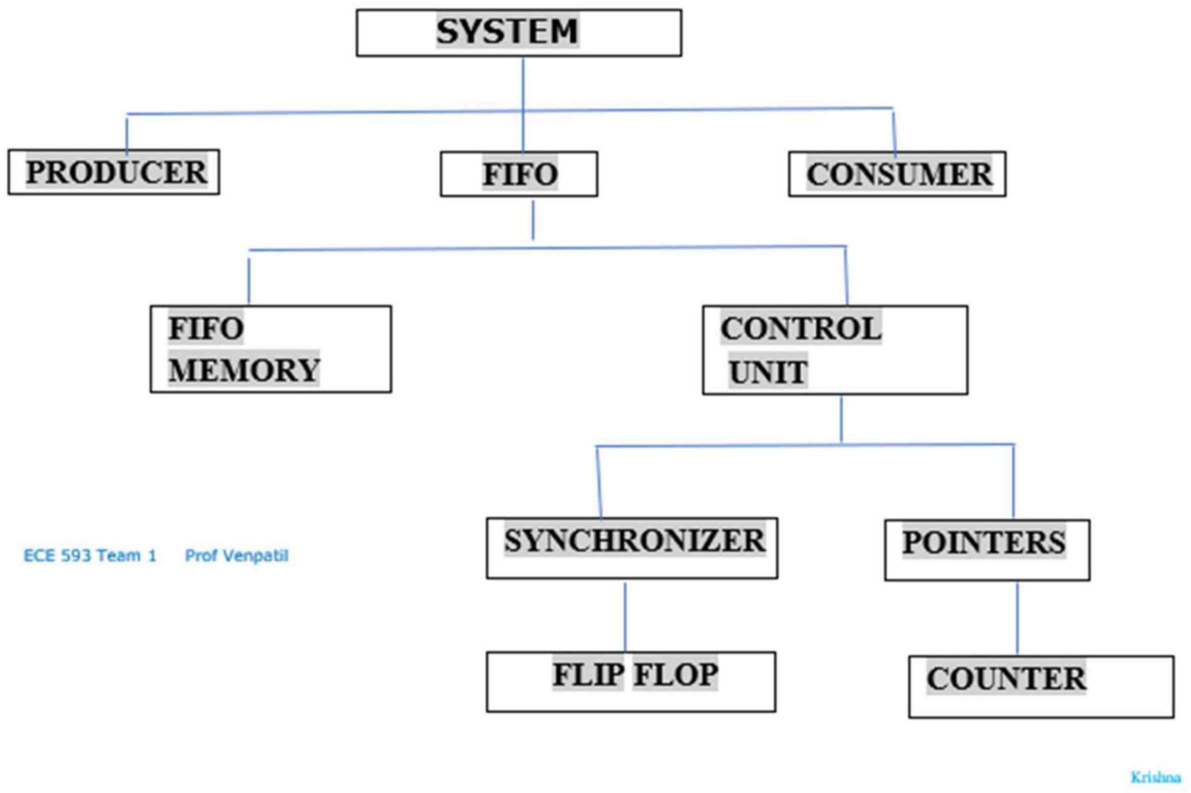
- The number of idle cycles between two consecutive reads is 1 clock cycle. This means that after reading one data item, module B waits for one clock cycle before initiating the next read. Therefore, it can be understood that one data is read every two clock cycles.

- The Time required to write one data item = 3 * 1/500 MHz = 6nS
- The Time needed to write a complete burst of data.= 1024* 6 nS. = 6,144 nS
- The time needed to read a single data item is 2 * 1/225 MHz, which equals 9 ns. Therefore, every 9 ns, module B reads one data item from the burst.
- Within a duration of 6144 ns, 1024 data items can be written.
- The number of data items that can be read within 6144 ns is calculated as (6144 ns / 9 ns) = 682.
- The remaining number of bytes to be stored in the FIFO is 1024 - 682 = 342.
- Hence, the minimum required depth of the FIFO is 342.

2. Verification Requirements

2.1 Verification Levels

The verification is performed at the top level.



3 Required Tools

3.1 List of necessary software and hardware tools.

Software Tool sets:

QuestaSim

Windows OS

3.2 The organization of run directories and the computer resources utilized.

Source Code :

- Asynch_fifo_top.sv
- fifomem.sv
- writeptr_full.sv
- readptr_empty.sv
- syncread2w.sv
- syncwrite2r.sv
- interface.sv

Testbench code:

- fifo_uvm_top.sv
- fifo_test.sv
- fifo_env.sv
- agent.sv
- fifo_scoreboard.sv
- driver.sv
- sequencer.sv
- monitor.sv
- fifo_sequence_item.sv
- sequencer_fifo_wr.sv
- fifo_coverage_uvm.sv

Script :

- run.do

4. Tests and Methods

Types of testing methods to be applied: Black Box, White Box, and Gray Box.

Black Box Testing:

- Verifying asynchronous operation by performing concurrent read and write actions.

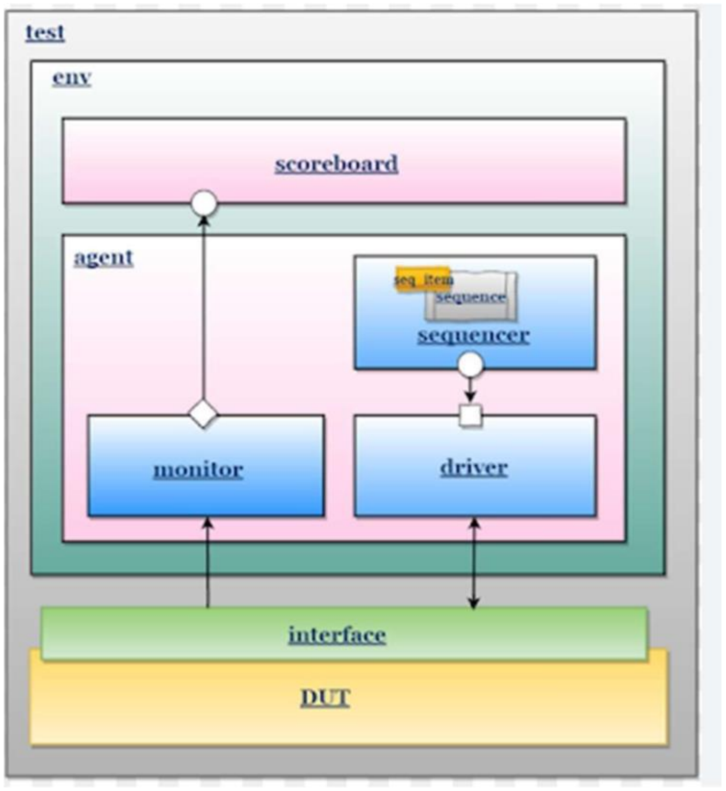
- Ensuring proper assertion of flags in the defined scenarios.
- Validating data integrity during read and write operations.
- Checking that the burst length fits within the FIFO as per specifications
- Confirming that the design functions at the specified frequency and accurately generates the expected waveforms.

White Box Testing:

- In future testbenches, assertions will be utilized to access internal variable values for testing.

4.1. Testbench Architecture:

Components used, including Drivers, Monitors, Scoreboards, Checkers, etc.
Below Block Diagram is taken from internet source



5. Verification Strategy:

Simulation was selected for functionality testing due to the design's nature and the flexibility it offers for updating testbenches in future milestones.

5.1 Verification Plan:

5.1.1 Functional Verification:

5.2 Corner Case Testing:

5.2.1 Test Case Scenarios:

Test Full:

This test case involves filling the FIFO by writing data to all available locations until it reaches its capacity. The verification process includes tracking the FIFO's full flag or status to ensure it correctly indicates when the FIFO is full. Additionally, the FIFO's response to attempts to write data when full should be examined. It should either prevent further writes, trigger an error, or follow some other predefined behavior.

Test Empty:

this test case, the FIFO starts off empty, and data is read from it until it becomes empty once more. Similar to the "Test Full" scenario, the empty flag or status of the FIFO must be monitored to ensure it accurately reflects when the FIFO is empty. Additionally, the FIFO's response to attempts to read from it when empty should be observed, such as blocking reads, generating an error, or following a specific behavior.

Test Full Error:

This test case checks the FIFO's response when attempting to write data while it is already full. The expected outcome is that the FIFO should either trigger an error, assert a full flag, or use another method to indicate that the write operation cannot be completed because the FIFO is full.

Test Empty Error:
Similar to the "Test Full Error" case, this test case checks the FIFO's response when attempting to read data while it is empty. The expected behavior is that the FIFO should either trigger an error, assert an empty flag, or use another method to indicate that the read operation cannot be executed because the FIFO is empty.

6. Coverage Metrics:

- Functional coverage tracks how effectively the test scenarios exercise all intended behaviors of the FIFO design, ensuring that important use cases are covered.
- Code coverage measures the percentage of design code — including lines, branches, and conditions — that is executed during testing, helping confirm comprehensive design exploration.
- Assertion coverage evaluates how well the assertions capture critical design properties and detect violations.

7. Verification Environment:

Developing a class-based verification environment for an Asynchronous FIFO involves multiple steps to ensure comprehensive testing and validation of the design.

7.1 Testbench Architecture:

- Define the overall testbench structure using UVM-based verification, incorporating verification components like drivers, monitors, and scoreboards.
- Organize the testbench hierarchy to promote modularity, scalability, and reusability.
- Implement distinct transactions for the FIFO’s input and output interfaces, encapsulating the functionality of the driver and monitor for each interface.
- Configure the packets to interface with the FIFO design, managing data transactions, protocol validation, and synchronization between clock domains.

Uvm top:

- All verification components, interfaces and DUT are instantiated in a top level module called testbench. It is a static container to hold everything required to be simulated and becomes the root node in the hierarchy. This is usually named tb or tb_top although it can assume any other name.

Interface:

- An interface is used to group all the design ports together as a bundle, making all input and output signals accessible in one place
- A **virtual interface** is employed to link the dynamic testbench architecture with a static DUT.

UVM Test

- A testcase is a pattern to check and verify specific features and functionalities of a design. A verification plan lists all the features and other functional items that needs to be verified, and the tests neeeded to cover each of them.

UVM Environment

- A UVM environment contains multiple, reusable verification components and defines their default configuration as required by the application. For example, a UVM environment may have multiple agents for different interfaces, a common scoreboard, a functional coverage collector, and additional checkers.

- It may also contain other smaller environments that has been verified at block level and now integrated into a subsystem. This allows certain components and sequences used in block level verification to be reused in system level verification plan.

UVM Agent:

- An agent encapsulates a Sequencer, Driver and Monitor into a single entity by instantiating and connecting the components together via TLM interfaces. Since UVM is all about configurability, an agent can also have configuration options like the type of UVM agent (active/passive), knobs to turn on features such as functional coverage, and other similar parameters.

UVM sequence (Transaction Class):

- UVM sequences are made up of several data items which can be put together in different ways to create interesting scenarios. They are executed by an assigned sequencer which then sends data items to the driver. Hence, sequences make up the core stimuli of any verification plan.

UVM Sequencer:

- A sequencer generates data transactions as class objects and sends it to the Driver for execution. It is recommended to extend uvm_sequencer base class since it contains all of the functionality required to allow a sequence to communicate with a driver. The base class is parameterized by the request and response item types that can be handled by the sequencer.

UVM Driver:

- UVM driver is an active entity that has knowledge on how to drive signals to a particular interface of the design. For example, in order to drive a bus protocol like APB, UVM driver defines how the signals should be timed so that the target protocol becomes valid. All driver classes should be extended from uvm_driver, either directly or indirectly.
- Transaction level objects are obtained from the Sequencer and the UVM driver drives them to the design via an interface handle.

UVM Monitor

A UVM monitor is responsible for capturing signal activity from the design interface and translate it into transaction level data objects that can be sent to other components.

In order to do so, it requires the following:

- A virtual interface handle to the actual interface that this monitor is trying to monitor
- TLM Analysis Port declarations to broadcast captured data to others.

UVM Scoreboard:

- UVM scoreboard is a verification component that contains checkers and verifies the functionality of a design. It usually receives transaction level objects captured from the interfaces of a DUT via TLM Analysis Ports.

UVM Coverage:

- UVM Coverage refers to the process of measuring and analyzing how much of the design functionality has been exercised by the testbench. It helps to determine whether the test scenarios are sufficient and whether critical features of the DUT (Design Under Test) have been thoroughly verified.

Stimulus Generation:

Inputs for testing the FIFO are generated using randomization and constraints

8. Bug Injected for Milestone 5:

For Milestone 5 we inject bug in writeptr_full.sv file. And the change is like this:

```
//injected bug as per Milestone-5
`ifdef BUG_INJECTED_MILESTONE5
assign next_waddr = waddr + (w_inc & wfull);
assign next_wptr = (next_waddr >> 1) & next_waddr;
assign bin_rptr_sync = gray_to_bin(rptr_sync);
`else
assign next_waddr = waddr + (w_inc & !wfull);
assign next_wptr = (next_waddr >> 1) ^ next_waddr;
assign bin_rptr_sync = gray_to_bin(rptr_sync);
`endif
```

9. Resources requirements

Team members and tasks allocation:

- **Bhargav Chunduri:** He will be focusing on implementing the core functionality of the asynchronous FIFO design. He will refine the write functionality, ensuring it works with the specified idle cycles. Bhargav will also design and test the FIFO memory, as well as ensure the correct operation of the generator and monitor modules. In addition, he will contribute to drafting the verification plan document to ensure that all write-related features are adequately tested.

- **Dhushyanth Dharmavarapu :** He will be in charge of implementing and verifying the read functionality of the asynchronous FIFO design, ensuring it meets the requirement of one read idle cycle. He will thoroughly test the design with various test cases, write the Design Specification document, and work on the driver and scoreboard functionalities to ensure the read domain operates as expected.

- **Venkata Krishna Kumar Vedantam :** He will lead the integration efforts, implementing the design through interfaces and ensuring the correct connectivity with the top module. He will calculate and document the FIFO depth, write the scoreboard, and verify the percentage thoroughly. Venkata will also create the overall test environment, develop tests, and set up the testbench top module. Throughout the process, he will collaborate with the other team members to ensure smooth integration and functionality.

10. REFERENCES:

- http://www.sunburst-design.com/papers/CummingsSNUG2002SJ_FIFO1.pdf
- The UVM Primer
- https://github.com/teekamkhandelwal/asynchronous_fifo/blob/main/r_pointer_epty.v
- https://verificationguide.com/uvm/uvm-testbench-architecture/
- https://www.chipverify.com/uvm/uvm-scoreboard