

---

# Verification Plan

Team 5 Asynchronous FIFO

May 2, 2025

**ECE 593**

Dr. Venkatesh Patil

**Team 5 Members:**

Kai Roy, Pavan Gaddam, SuryaTeja Purma, Gene Hu

# 1 Table of Contents

## [1 Table of Contents](#)

## [2 Introduction](#)

### [2.1 Team 5 Github Repository Link](#)

### [2.2 Objective of the verification plan](#)

### [2.3 Top Level block diagram](#)

### [2.4 Specifications for the design](#)

## [3 Verification Requirements](#)

### [3.1 Verification Levels](#)

#### [3.1.1 Hierarchy Level](#)

#### [3.1.2 Controllability and Observability](#)

#### [3.1.3 List of Interfaces and Specification](#)

## [4 UVM Verification](#)

### [4.1 UVM Architecture](#)

### [4.2 Verification Levels](#)

#### [4.2.1 Hierarchy Level](#)

#### [4.2.2 Controllability and Observability](#)

## [5 Required Tools](#)

### [5.1 List of required software and hardware toolsets needed.](#)

## [6 Risks and Dependencies](#)

### [6.1 Risks](#)

### [6.2 Mitigation](#)

## [7 Functions to be Verified.](#)

### [7.1 Functions from specification and implementation](#)

#### [7.1.1 List of functions that will be verified. Description of each function](#)

#### [7.1.2 List of functions that will not be verified / need to be implemented](#)

#### [7.1.3 List of critical functions and non-critical functions for tapeout](#)

## [8 Tests and Methods \(TEST CASES\)](#)

### [8.1.1 Testing methods to be used](#)

### [8.1.2 State the PROs and CONs for each and why you selected the method for this DUV.](#)

### [8.1.3 Testbench Architecture; Component used \(list and describe Drivers, Monitors, scoreboards, checkers etc.\)](#)

### [8.1.4 Verification Strategy](#)

### [8.1.5 What is your driving methodology?](#)

### [8.1.6 What will be your checking methodology?](#)

### [8.1.7 Testcase Scenarios \(Matrix\)](#)

#### [8.1.7.1 Basic Tests](#)

#### [8.1.7.2 Complex Tests](#)

#### [8.1.7.3 Regression Tests \(Do tests again on code updates\)](#)

[8.1.7.4 Any special or corner cases test cases](#)

[9 Coverage Requirements](#)

[9.1.1.1 Describe Code and Functional Coverage goals for the DUV](#)

[9.1.2 Covergroups](#)

[9.1.3 Assertions](#)

[10 Roles and Responsibilities](#)

[11 Schedule](#)

[12 References](#)

## 2 Introduction

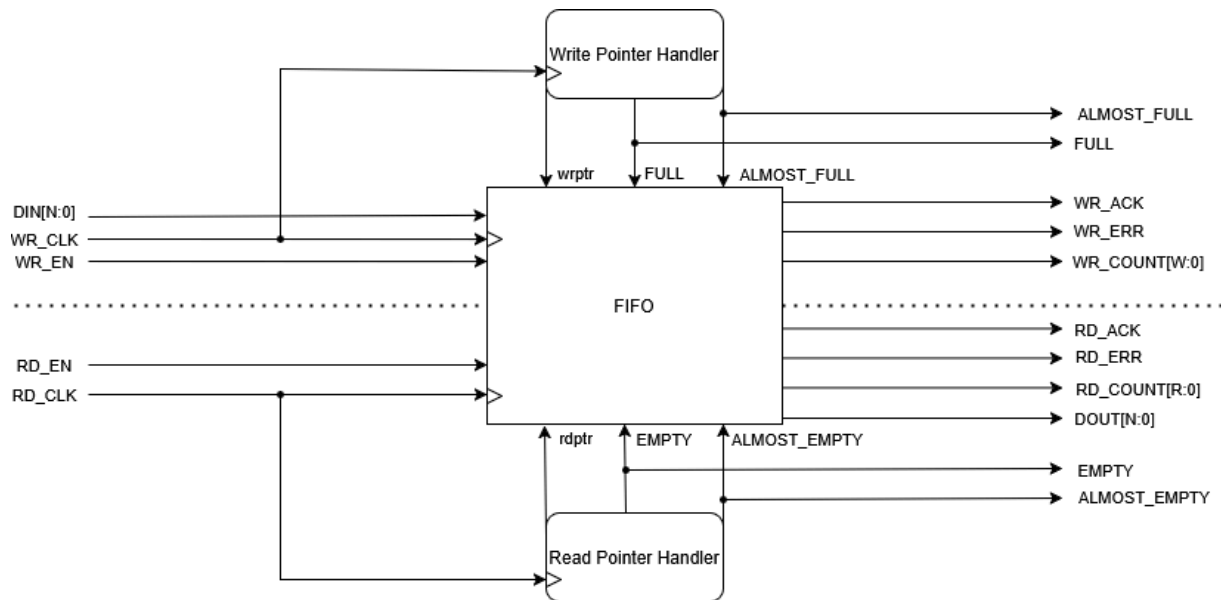
### 2.1 Team 5 Github Repository Link

[https://github.com/xbicai/ECE593\\_Project](https://github.com/xbicai/ECE593_Project)

### 2.2 Objective of the verification plan

The Purpose of this document is to provide members of the project a guideline for how the asynchronous FIFO should function. By having a plan to follow, each member will be able to put in work that will help reach the end goal.

### 2.3 Top Level block diagram



### 2.4 Specifications for the design

Data can be safely passed from one asynchronous clock domain to another using FIFO. When two clock domains are asynchronous to one another, an asynchronous FIFO design occurs when data values are written from one clock domain to a FIFO buffer and read from the same FIFO buffer from another clock domain.

The next word to be written is always shown by 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. The current FIFO word to be read is always shown by the read pointer. When the read and write pointers are equal, the FIFO is empty.

FIFO Depth Calculation:

Double the frequency of write and half of the read

120 MHz is the sender clock frequency.

Three idle cycles separate two consecutive writes, and the receiver clock frequency is 50 MHz.

There are two idle cycles in between two consecutive reads.

Given that the sender clock frequency is less than the receiver clock frequency, write burst = 1024.

Three clock cycles separate two consecutive writes, indicating that after writing one piece of data, the write module waits three clock cycles before starting the next write. This means that one piece of data is written every four clock cycles.

Writing one data item takes  $4 * (1/120\text{MHz}) = 33.33 \text{ ns}$ .

It took  $34,129.92 = 34,129 \text{ ns}$  to write the data in the burst.

One data item's reading time is equal to  $3 * (1/50\text{MHz}) = 60 \text{ ns}$ .

Thus, the read module will read one data item from the burst every 60 ns.

In 34,129 ns, the number of data pieces that can be read is  $34,129/60 = 568.81 = 568 \text{ items}$ .  $1024 - 568 = 456$  is the remaining number of bytes that must be kept in the FIFO.

## 3 Verification Requirements

### 3.1 Verification Levels

#### 3.1.1 Hierarchy Level

This project will be verified at the **block level** since we want to verify the entire asynchronous FIFO module. Such parts include data correctness, proper clock timing, and flag handling (empty/full).

#### 3.1.2 Controllability and Observability

Controllability will be good because the asynchronous FIFO's inputs will be directly controlled by the testbench. Observability will be good too, because the flags such as empty/full can be monitored directly by the testbench or QuestaSim.

#### 3.1.3 List of Interfaces and Specification

Interfaces	Description
din	Data write with parameterized width
dout	Data read with parameterized width
fifo_full	FIFO full flag
fifo_empty	FIFO empty flag
fifo_almost_full	FIFO almost full flag (one below full)
fifo_almost_empty	FIFO almost empty flag (one above empty)
wr_en	Write enable bit
wclk_i	Write clock
rd_en	Read enable bit
rclk_i	Read clock
ainit	Reset bit for the entire FIFO

**Specifications:**

- Separate read and write clocks to allow for a fully asynchronous FIFO.
- Reset is low enable.
- Data width is parameterizable with a default width of 8 bits.
- FIFO depth is parameterizable with default depth of 4 entries.
- FIFO behavior, first write in is the first write out.
- FULL flag is asserted when all entries are valid.
- EMPTY flag is asserted when all entries are invalid.
- Reset low will bring all four flags high.
  - The next write clock will bring FULL and ALMOST\_FULL flags low
  - The EMPTY and ALMOST\_EMPTY flags will fall low on the first and second valid write respectively
- Writes are blocked when FULL is high, and reads are blocked when EMPTY is high.

## 4 UVM Verification

### 4.1 UVM Architecture

This project will follow the traditional UVM testbench architecture and component hierarchy. The testbench top will include the UVM test class, the virtual interface, and the DUT. The UVM test class will include the environment, the sequence items, and the test sequence classes. The environment class is where the coverage, scoreboard, and agent classes are instantiated, with the agent consisting of the sequencer, driver, and monitor. The sequencer will connect to the static sequences and send that information dynamically to the driver. The driver will interact with the virtual interface to provide the input stimulus to the DUT. The monitor will observe the virtual interface for any input stimulus from the driver and output response from the DUT and send that information to the scoreboard. The scoreboard is where that data is compiled and measured for correctness. An extended UVM subscriber is used to take functional coverage during the simulation.

### 4.2 Verification Levels

#### 4.2.1 Hierarchy Level

This project will be verified at the **block level** since we want to verify the entire asynchronous FIFO module. Such parts include data correctness, proper clock timing, and flag handling (empty/full).

#### 4.2.2 Controllability and Observability

The sequence classes will allow us to make a variety of random and directed test stimulus that will be sent to the DUT via the other UVM components allowing us a good level of controllability of the DUT. The scoreboard is what will provide us the observability of the DUTs actions allowing us to both via the input stimulus and the corresponding output stimulus from the DUT for comparison. Since we are taking a more of a gray box approach to the testbench, we are limited by the output ports of the DUT for observing functionality and not any internal values. Since the DUT is parameterized for data size and buffer depth, we do know those aspects of the DUT's characteristics.

## 5 Required Tools

### 5.1 List of required software and hardware toolsets needed.

- QuestaSim (ran on PSU servers)
- MobaXterm (or some other server remote access method) for PSU Server access.
- UVM SV library

## 6 Risks and Dependencies

### 6.1 Risks

- Asynchronous Clock Synchronization: Metastability in pointer synchronization across clock domains.
- Coverage Gaps: Missing corner cases (e.g., simultaneous read/write with maximum depth)

### 6.2 Mitigation

- Use constrained-random tests to stress clock domain crossings.
- Implement comprehensive coverage metrics and review reports weekly.
- Schedule tool access and optimize testbench for simulation efficiency.

## 7 Functions to be Verified.

### 7.1 Functions from specification and implementation

#### 7.1.1 List of functions that will be verified. Description of each function

Function	Description
Write Operation	Data is being written in correctly when <b>wr_en</b> high and <b>fifo_full</b> low
Read Operation	Data is being read out correctly when <b>rd_en</b> high and <b>fifo_empty</b> low
Reset	When <b>reset</b> is asserted, all four flags will go high, and memory is flushed
Data Integrity	FIFO correctly outputs the same data in the same order it was put in
Full / Empty Flags	<b>fifo_full</b> is asserted when fifo is full <b>fifo_empty</b> is asserted when fifo is empty
Data width and memory depth	Supports data widths up to 256 bits Supports memory depths of up to 65,535 locations



Almost full / empty flags

When the FIFO has one empty entry left, the almost full flag will be asserted, and when the FIFO has only one valid entry, the almost empty flag will be asserted

Read / write requests are rejected without affecting FIFO state

When the FIFO is empty or full then the respective reads or writes will be rejected without affecting the values currently stored in the FIFO

## 7.1.2 List of functions that will not be verified / need to be implemented

Function	Why
WR_ACK, WR_ERR, RD_ACK, RD_ERR, WR_COUNT, RD_COUNT.	Not critical to functionality
Optional count vector(s) provide visibility into number of data words currently in the FIFO, synchronized to either clock domain	Not critical to functionality

## 7.1.3 List of critical functions and non-critical functions for tapeout

- Fixing the FULL/EMPTY flag bug. The read and write pointers prematurely wrap back to the beginning instead of the last entry in the FIFO. [FIXED]

# 8 Tests and Methods (TEST CASES)

## 8.1.1 Testing methods to be used

The team will be using gray box testing over black or white box testing.

## 8.1.2 State the PROs and CONs for each and why you selected the method for this DUV.

	Pros	Cons
Black	<ul style="list-style-type: none"><li>Quickly validate functionality</li><li>No need to understand internal signals</li></ul>	<ul style="list-style-type: none"><li>Will miss internal bugs</li><li>Low controllability/observability of internal signals</li></ul>
Gray	<ul style="list-style-type: none"><li>Have some knowledge of internal signals<ul style="list-style-type: none"><li>Leads to better detection of internal bugs</li></ul></li></ul>	<ul style="list-style-type: none"><li>Still don't see every internal signal or structure, which means some bugs might be missed</li></ul>

White	<ul style="list-style-type: none"> <li>• Highest controllability/observability</li> </ul>	<ul style="list-style-type: none"> <li>• High costs for maintenance (if one internal signal is changed, test might need a reflecting change)</li> </ul>
-------	---	---

The team decided on gray box testing because it provides the most balanced approach when it comes to controllability and observability. The most important signals and protocols can be tested along with their functionalities.

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

- Generator: Generates DIN, WR\_EN, RD\_EN signals with randomized or specified values depending on operation mode.
- Driver: Drivers the DIN, WR\_EN, RD\_EN signals synchronized to WR\_CLK and RD\_CLK to the DUT via the virtual interface. Uses random DIN values.
- Monitor: Observes DOUT, FULL, EMPTY, ALMOST\_FULL, ALMOST\_EMPTY.
- Scoreboard: Tracks written data and compares with read data to verify integrity. Maintains expected FIFO state (e.g., occupancy). Automates comparison of actual vs. expected outputs using a reference, behavioral fifo buffer.

### 8.1.4 Verification Strategy

The team decided on **dynamic simulation** since it's suitable for verifying asynchronous behavior, flag updates, and handshake signals. Formal verification may be added for pointer synchronization.

### 8.1.5 What is your driving methodology?

The driving methodology will be a mix of directed tests and constrained random testing depending on the need of the tests.

- Directed Tests: Reset, basic write/read, fill/empty scenarios.
- Constrained Random: Random DIN, random delays between read/write operations.

### 8.1.6 What will be your checking methodology?

- From specification: Verify flags, handshake signals, and count vectors.
- Automated checkers compare actual vs. expected outputs.

### 8.1.7 Testcase Scenarios (Matrix)

#### 8.1.7.1 Basic Tests

Number	Name	Test Description/ Features	Status
1.1.1	Standard write/read	Write 01234567 in Read 01234567 out	done
1.1.2	Random write/read stress test	Write many random bursts of data in, then confirm FIFO correctly reads these bursts out. Do this over a long period (write a lot, read a lot, write a lot...)	done
1.1.3	Fill	Write into FIFO until it's FULL, <b>fifo_full</b> should be asserted	done

1.1.5	Write when full	Write into FIFO when <b>fifo_full</b> is asserted; FIFO should block write	done
1.1.4	Drain	Read from FIFO until it's EMPTY, <b>fifo_empty</b> should be asserted	done
1.1.6	Read when empty	Write into the FIFO when <b>fifo_empty</b> is asserted; FIFO should block read	done
1.1.7	Reset	Reset when empty Reset when partially full Reset when full Write w/ reset on Read w/ reset on Make sure correct signals are reset in any instance	done

#### 8.1.7.2 Complex Tests

Number	Name	Test Description/ Features	Status
1.2.1	Concurrent testing 1	Concurrent read and write Condition: <b>fifo_full + rclk=wclk</b> What should happen: Read is allowed, write is blocked, <b>fifo_full</b> is low after clock	done
1.2.2	Concurrent testing 2	Concurrent read and write Condition: <b>fifo_empty + rclk=wclk</b> What should happen: Read is blocked, write is allowed, <b>fifo_empty</b> is low after clock	done
1.2.3	Concurrent testing 3	Concurrent read and write Condition: <b>fifo_empty</b> and <b>fifo_full + rclk=wclk</b> What should happen: Read is blocked, write is blocked, <b>fifo_empty/fifo_full</b> is low after clock  Note: this is possible after a reset since all flags are high	done
1.2.4	Concurrent testing 4	Concurrent read and write after inserting a single test data into fifo Condition: no flags asserted + <b>rclk=wclk</b> What should happen: Read out the test data, write in new data	done

#### 8.1.7.3 Regression Tests (Do tests again on code updates)

Test Name / Number	Test Description/Features	Status
1.3.1	Added more function bins 2/27	done
1.3.2		

#### 8.1.7.4 Any special or corner cases test cases

Number	Name	Test Description/ Features	Status
1.4.1	WR	Check input output of W->R->W->R many times Condition: fifo_empty <b>fifo_empty</b> should be asserted after every read	done
1.4.2	WWR	Check input output of W->W->R->W->W->R many times <b>fifo_full</b> should be asserted eventually Condition: fifo_empty	done
1.4.3	RRW	Check input output of R->R->W... many times First two reads will be blocked, then it will start allowing the first read and blocking the second, empty flag is asserted correctly	done

## 9 Coverage Requirements

### 9.1.1 Describe Code and Functional Coverage goals for the DUV

The goal for code coverage in this project is 100%, and 90% coverage for the Class-based testbench for Milestone 3 and 100% coverage for the UVM testbench. This will give us time to improve our set of test cases between the Class-based stage and the UVM stage of the verification project and adjust for any changes to the DUT that occur between those stages.

### 9.1.2 Covergroups

- 9.1.2.1 Coverpoint for AINIT and transition bins for 0->1 and 1->0
- 9.1.2.2 Coverpoint for Req\_WR with bins for it being 0, 1, and it being 1 for concurrent cycles
- 9.1.2.3 Coverpoint for Req\_RD with bins for it being 0, 1, and it being 1 for concurrent cycles
- 9.1.2.4 Coverpoint for data (Both in & out) where the data is all 0s, Low values, and high values, and all 1s (max value).
- 9.1.2.5 Coverpoint for Full, Almost Full, Empty, and Almost Empty Flags with bins for them being 0, 1, 0->1, 1->0.

### 9.1.3 Assertions

- 9.1.3.1 If Full flag is high, Almost Full is also high
- 9.1.3.2 If Empty flag is high, Almost Empty is also high
- 9.1.3.3 If Almost Full flag is low, Full is also low
- 9.1.3.4 If Almost Empty flag is low, Empty is also low

## 10 Roles and Responsibilities

Role	Main	Helper
RTL coding	Pavan	Suryateja, Kai

Testbench coding	Kai, Gene	Suryateja, Pavan
Design specification	Gene	
Verification plan	Gene, Suryateja	Kai
Presentation	Suryateja	

## 11 Schedule

Objective	Description	Date
Milestone 1	<ul style="list-style-type: none"> <li>• Complete design specification</li> <li>• Start initial verification plan</li> <li>• Initial implementation of design</li> <li>• Simple initial testbench</li> </ul>	4/20/25
Milestone 2	<ul style="list-style-type: none"> <li>• Develop class based testbench with completed interfaces <ul style="list-style-type: none"> <li>◦ Has components: generator, driver, monitor, scoreboard, mailbox</li> <li>◦ Verify 20-50 random bursts of data</li> </ul> </li> <li>• Update verification plan</li> </ul>	5/2/25
Milestone 3	<ul style="list-style-type: none"> <li>• Finalize changes to RTL</li> <li>• Complete class based testbench <ul style="list-style-type: none"> <li>◦ Finish coverage here as well</li> </ul> </li> <li>• Add reports for code and functional coverage</li> <li>• Update verification plan</li> </ul>	5/22/25
Milestone 4	<ul style="list-style-type: none"> <li>• Develop UVM TB</li> <li>• Update verification plan to include UVM plan</li> <li>• Utilize UVM_MESSAGE and UVM_LOGGING to create logs and reports about the data</li> </ul>	5/22/25
Milestone 5	<ul style="list-style-type: none"> <li>• Complete UVM architecture, environment, and testbench</li> <li>• All test cases completed with coverage reflected</li> <li>• Create RTL bug as a bug injection and verify</li> <li>• Finalize all documents and presentation</li> </ul>	5/27/25
Milestone 6	<ul style="list-style-type: none"> <li>• Fix critical bug (flags)</li> <li>• Create RTL as bug injection</li> <li>• Maybe add macro for turning on/off specific test cases</li> </ul>	6/03/25

## 12 References

- Xilinx Asynchronous FIFO V3.0 Design Specification.
- SystemVerilog IEEE 1800-2017 Standard.
- ECE-593 Course Materials, Winter 2025.
- Questa® SIM Command Reference Manual