Asynchronous FIFO Verification Document

Table of Contents

CHAPTER 1: PROJECT OVERVIEW AND SPECIFICATIONS

1.1 Project Overview

An Asynchronous FIFO (First-In-First-Out) is a special type of memory used to transfer data between two different clock domains. In this system, the write and read operations use separate clocks that are not synchronized. This allows data to be written at one clock speed and read at another, ensuring safe and reliable data transfer between parts of a digital system working at different frequencies.

Asynchronous FIFOs are important for crossing clock domains without data loss or corruption. They are widely used in SoC designs, FPGAs, and communication systems. A common use case is connecting high-speed processors with slower peripherals to maintain smooth data flow even when their clocks differ. They are also used to link modules that run on independent clocks, such as CPUs, communication blocks, or external interfaces.

1.2 Verification objective

- Implement a modular testbench using different UVM components, including assertions and coverage mechanisms.
- Verify the design's behavior under reset conditions.
- Perform functional validation to ensure correct write and read operations across respective clock domains as per the specification.
- Achieve maximum code coverage and functional coverage for complete verification.
- Identify and analyze any potential timing issues.

1.3 DUT Interfaces

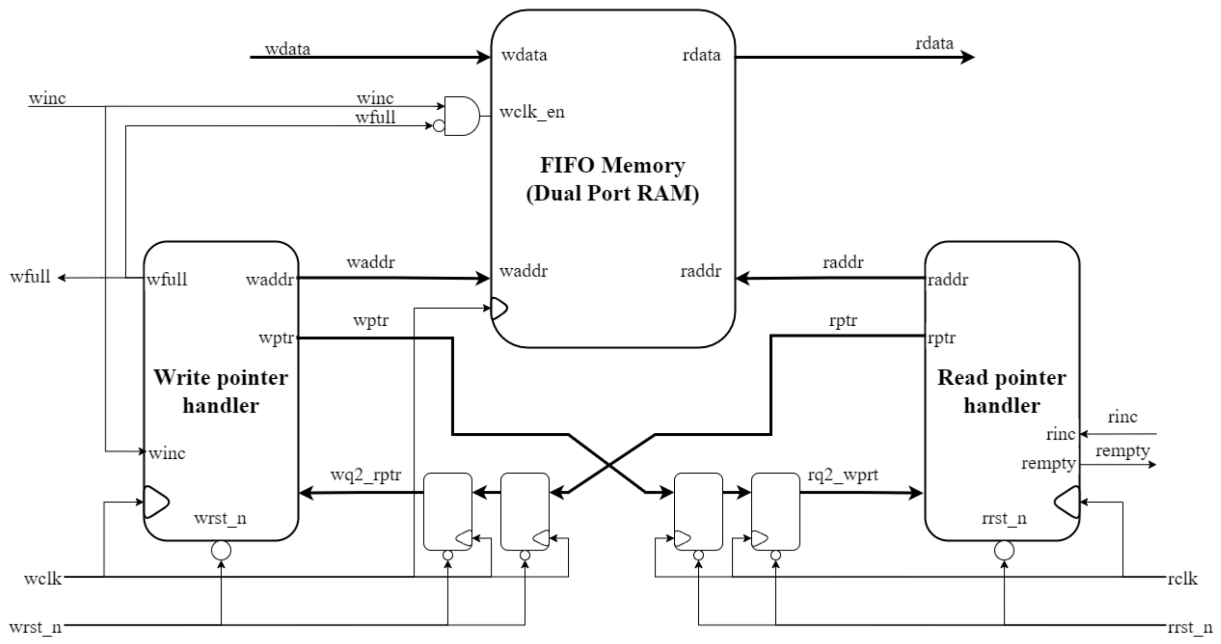


Figure 1: Asynchronous FIFO block diagram

Pin description

Signal	Size(bits)	Description
Input Ports		
WCLK	1	Write Clock Signal
RCLK	1	Read Clock Signal
WRST_N	1	Active-low Asynchronous Write Reset
RRST_N	1	Active-low Asynchronous Read Reset
WINC	1	Write Increment/Enable

RINC	1	Read Increment/Enable
WDATA	DATA_SIZE	Write Data
Output Ports		
REMPTY	1	Read Empty
WFULL	1	Write Full
RDATA	DATA_SIZE	Read Data

CHAPTER 2: TESTBENCH ARCHITECTURE AND METHODOLOGY

2.1 Testbench Architecture

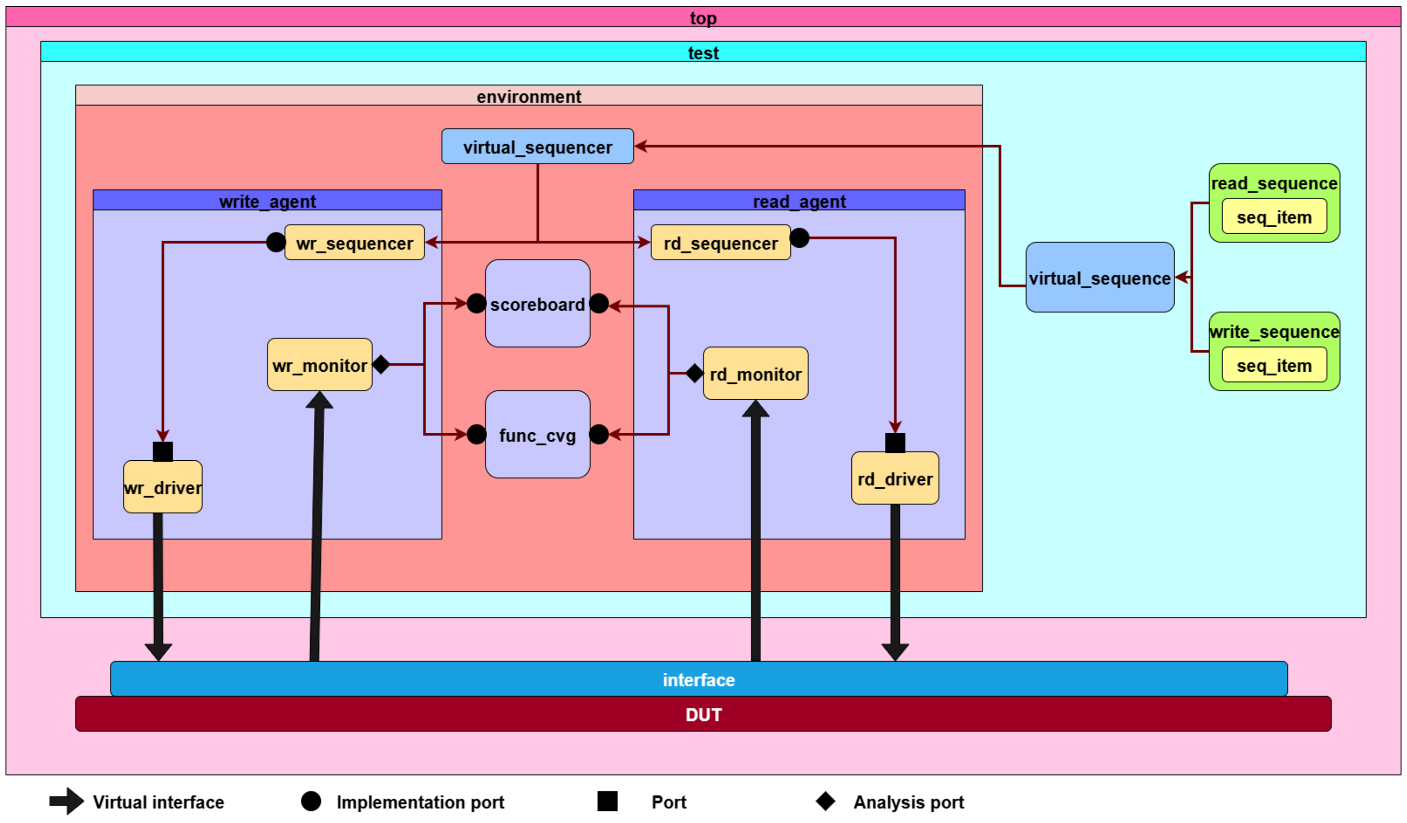


Figure 2: Testbench architecture

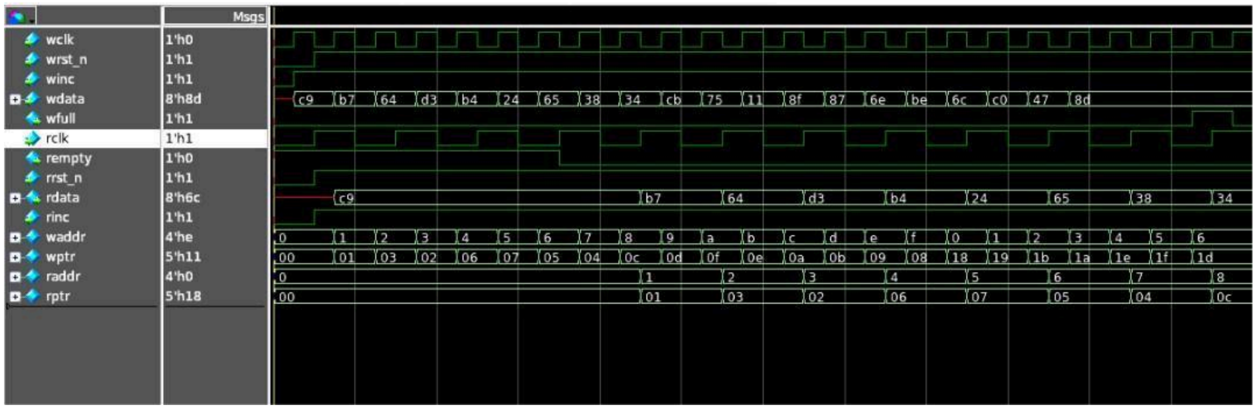
2.2 Component Details

- Sequence Item – A user-defined transaction object that holds the data fields for a single operation, such as input values and control signals.
- Sequence – Generates a series of sequence items to provide stimulus to the DUT and is initiated from the test.

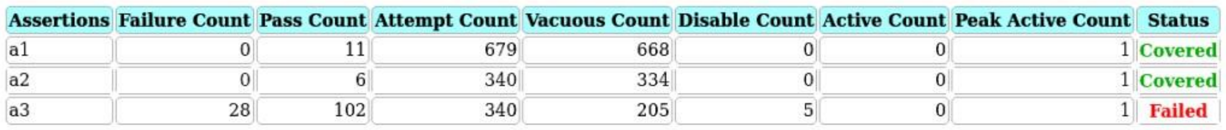
- Sequencer – Serves as a communication link between the sequence and the driver, transferring sequence items using TLM connections to ensure proper delivery of transactions.
- Driver – Receives transactions from the sequencer and converts them into pin-level signals to drive the DUT through the interface.
- Monitor – Passively observes the DUT’s signals through the virtual interface, converts them into transaction objects, and sends them to the scoreboard via a TLM analysis port. An active monitor captures DUT inputs, while a passive monitor captures DUT outputs.
- Agent – A reusable UVM component that groups the driver, sequencer, and monitor. An active agent includes all three components and captures inputs, while a passive agent contains only the monitor to observe DUT outputs.
- Scoreboard – Compares the DUT’s actual outputs with the expected results to verify correctness.
- Subscriber – Collects functional coverage information to ensure all test scenarios are exercised. It connects to active and passive monitors through TLM analysis ports.
- Environment – A container that instantiates and organizes agents, scoreboards, and subscribers into a complete verification setup.
- Test – The top-level UVM component responsible for building the environment, configuring components, and starting the stimulus generation.
- Top – Instantiates the DUT and interface, and initiates the UVM phasing mechanism.
- Interface – Provides the connection between the testbench and the DUT signals, enabling interaction between them.

CHAPTER 3: RESULTS

Waveform



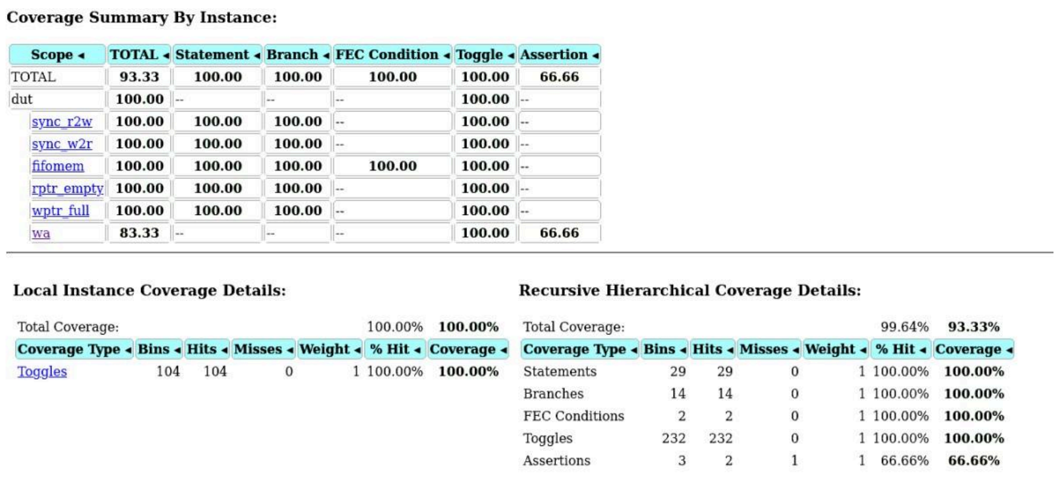
Assertion



Assertions	Failure Count	Pass Count	Attempt Count	Vacuous Count	Disable Count	Active Count	Peak Active Count	Status
a1	0	11	679	668	0	0	1	Covered
a2	0	6	340	334	0	0	1	Covered
a3	28	102	340	205	5	0	1	Failed

Coverage

Code coverage



Coverage Summary By Instance:

Scope	TOTAL	Statement	Branch	FEC Condition	Toggle	Assertion
TOTAL	93.33	100.00	100.00	100.00	100.00	66.66
dut	100.00	--	--	--	100.00	--
sync_r2w	100.00	100.00	100.00	--	100.00	--
sync_w2r	100.00	100.00	100.00	--	100.00	--
fifomem	100.00	100.00	100.00	100.00	100.00	--
rptr_empty	100.00	100.00	100.00	--	100.00	--
wptr_full	100.00	100.00	100.00	--	100.00	--
wa	83.33	--	--	--	100.00	66.66

Local Instance Coverage Details:

Total Coverage: 100.00% **100.00%**

Coverage Type	Bins	Hits	Misses	Weight	% Hit	Coverage
Toggles	104	104	0	1	100.00%	100.00%

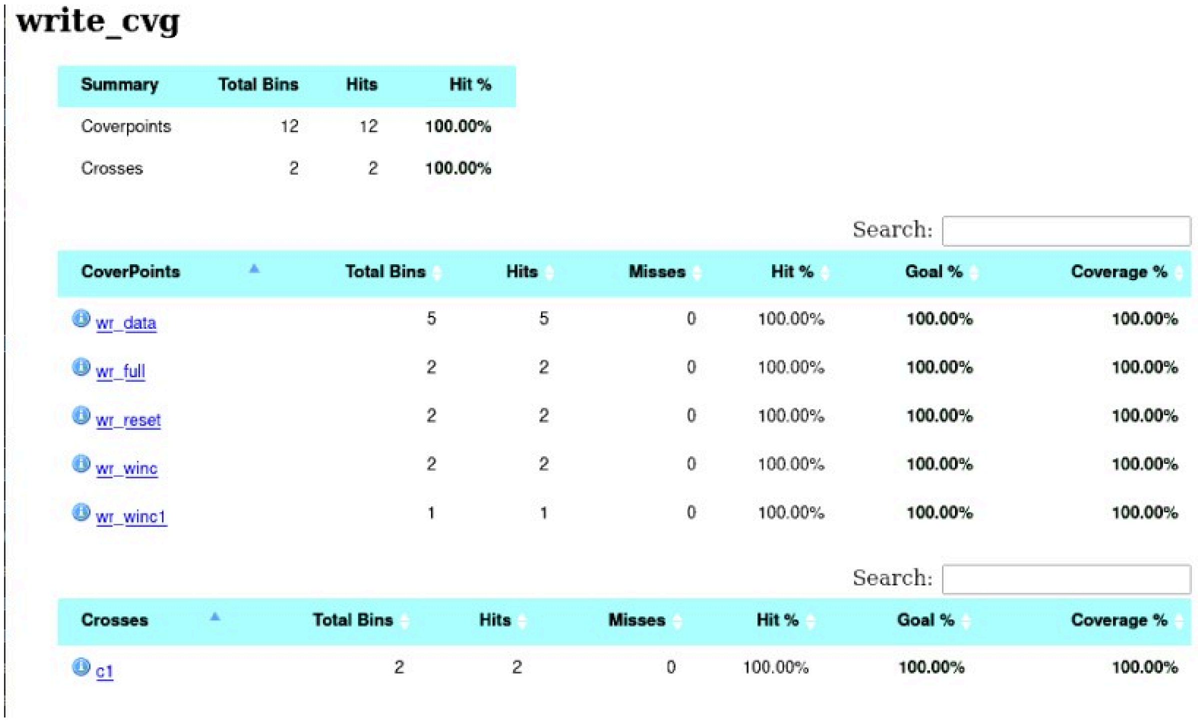
Recursive Hierarchical Coverage Details:

Total Coverage: 99.64% **93.33%**

Coverage Type	Bins	Hits	Misses	Weight	% Hit	Coverage
Statements	29	29	0	1	100.00%	100.00%
Branches	14	14	0	1	100.00%	100.00%
FEC Conditions	2	2	0	1	100.00%	100.00%
Toggles	232	232	0	1	100.00%	100.00%
Assertions	3	2	1	1	66.66%	66.66%

Functional coverage

Write covergroup



write_cvg

Summary	Total Bins	Hits	Hit %
Coverpoints	12	12	100.00%
Crosses	2	2	100.00%

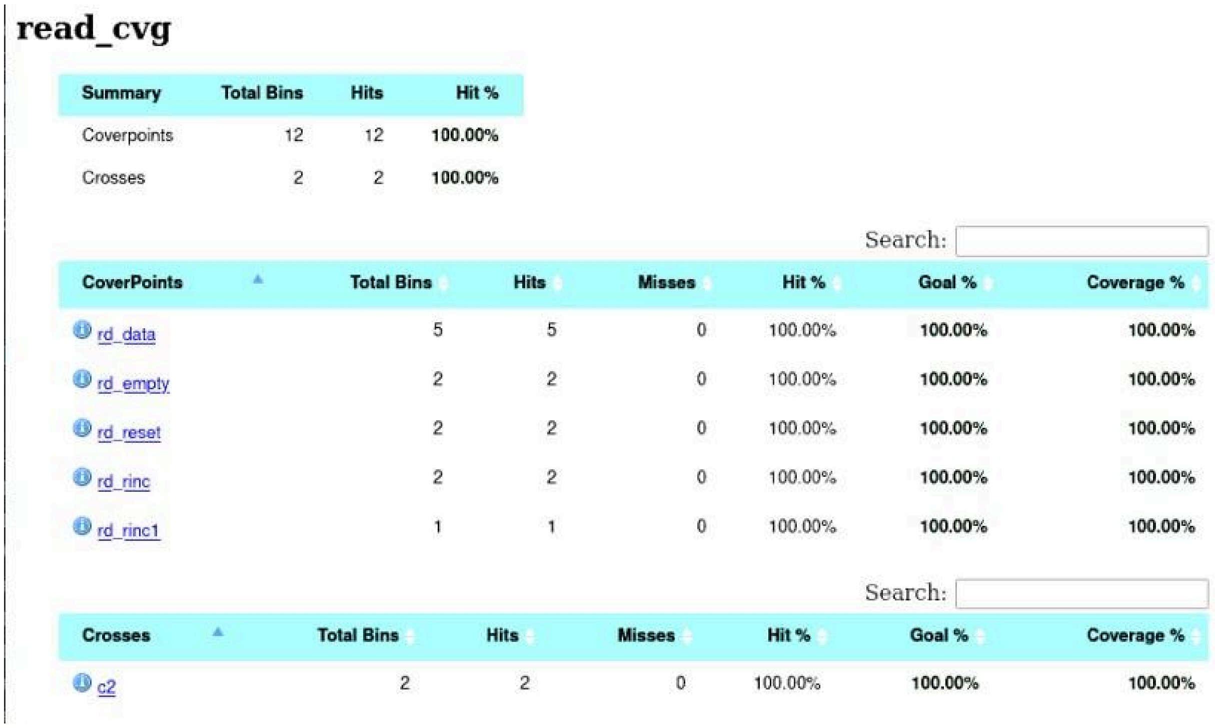
Search:

CoverPoints	Total Bins	Hits	Misses	Hit %	Goal %	Coverage %
wr_data	5	5	0	100.00%	100.00%	100.00%
wr_full	2	2	0	100.00%	100.00%	100.00%
wr_reset	2	2	0	100.00%	100.00%	100.00%
wr_winc	2	2	0	100.00%	100.00%	100.00%
wr_winc1	1	1	0	100.00%	100.00%	100.00%

Search:

Crosses	Total Bins	Hits	Misses	Hit %	Goal %	Coverage %
c1	2	2	0	100.00%	100.00%	100.00%

Read covergroup



read_cvg

Summary	Total Bins	Hits	Hit %
Coverpoints	12	12	100.00%
Crosses	2	2	100.00%

Search:

CoverPoints	Total Bins	Hits	Misses	Hit %	Goal %	Coverage %
rd_data	5	5	0	100.00%	100.00%	100.00%
rd_empty	2	2	0	100.00%	100.00%	100.00%
rd_reset	2	2	0	100.00%	100.00%	100.00%
rd_rinc	2	2	0	100.00%	100.00%	100.00%
rd_rinc1	1	1	0	100.00%	100.00%	100.00%

Search:

Crosses	Total Bins	Hits	Misses	Hit %	Goal %	Coverage %
c2	2	2	0	100.00%	100.00%	100.00%