ASYNCHRONOUS FIFO

VERIFICATION

DOCUMENT

Sumanth

Employee ID:6115

Mirafra Technologies Manipal

**Table of contents**

[CHAPTER 1:Project Overview 2](#_Toc28576)

[1.1 Project Overview: 2](#_Toc18558)

[1.2 Verification Objectives 3](#_Toc8432)

[1.3 DUT Interfaces 3](#_Toc21161)

[CHAPTER 2 :Testbench Architecture 6](#_Toc8132)

[2.1 ASYNCHRONOUS FIFO Testbench Architecture 6](#_Toc5968)

[2.2 Component details and Flowchart 7](#_Toc17152)

[ UVM Sequence Item 7](#_Toc15398)

[ UVM Sequence and Sequencer 7](#_Toc641)

[ UVM Driver 7](#_Toc26300)

[ UVM Monitor 8](#_Toc28178)

[ UVM Agent 8](#_Toc28203)

[ UVM Scoreboard 8](#_Toc21758)

[ UVM Subscriber 9](#_Toc11999)

[ UVM Environment 9](#_Toc27426)

[ UVM Test 9](#_Toc14806)

[ UVM Testbench Top 10](#_Toc24499)

[CHAPTER 3:Verification Results 11](#_Toc8923)

[3.1 Errors in the DUT 11](#_Toc15127)

[3.2 Coverage Report 12](#_Toc3451)

[ Input Coverage 12](#_Toc17524)

[ Output coverage 12](#_Toc22220)

[ Assertion coverage 13](#_Toc29933)

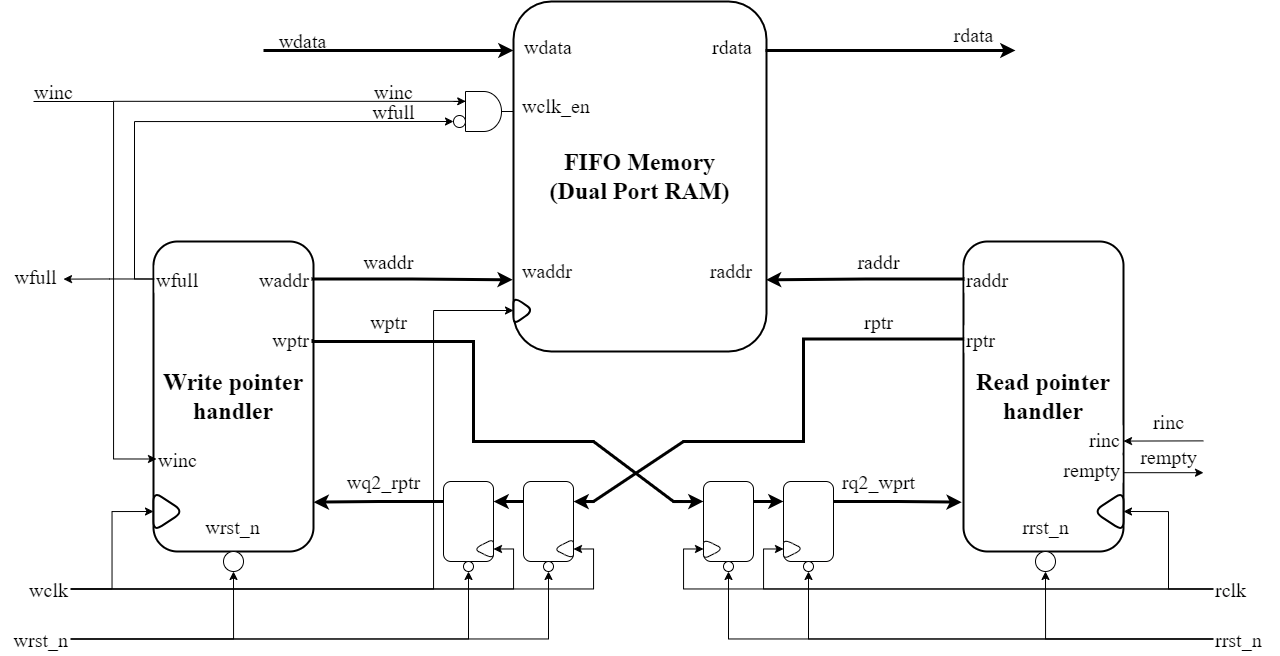
[ Overall coverage 13](#_Toc27196)

[ Output waveform 14](#_Toc28075)

# CHAPTER 1:Project Overview

## 1.1 Project Overview:

An asynchronous FIFO (First-In, First-Out) is a type of memory buffer used to transfer data safely between two parts of a system that operate on different, unrelated clocks. It stores data in the order it arrives, ensuring that the first data written is also the first to be read out. The FIFO consists of separate write and read sections, each controlled by its own clock domain. To prevent errors due to clock differences, special synchronization techniques are used—most commonly Gray code pointers and double-flop synchronizers—to track how full or empty the FIFO is without causing metastability. This design allows reliable communication between components running at different speeds, such as between a fast processor and a slower peripheral.



The asynchronous FIFO verification testbench thoroughly validates the FIFO design by testing all functional and timing aspects across different clock domains. It verifies correct data transfer between independent write and read clocks operating at various frequencies and phase relationships. The testbench ensures that data written into the FIFO is read out in the exact same order (First-In, First-Out) without corruption or loss. It checks proper generation of status flags, including wfull, rempty, under various conditions. The testbench also injects random write and read enable patterns to test corner cases like simultaneous write and read. Finally, it verifies correct operation across different FIFO depths and data widths, confirming that the parameterized design functions reliably for all supported configurations.

## 1.2 Verification Objectives

* Verify correct reset behaviour, ensuring both pointers are cleared and FIFO is empty.
* Validate proper data integrity during write and read operations.
* Ensure full and empty flags assert and de-assert correctly under boundary conditions, including pointer wrap-around.
* Check simultaneous read and write operations across clock domains without data corruption
* Achieve functional and cross coverage to confirm all states, transitions, and corner cases are exercised.

## 1.3 DUT Interfaces

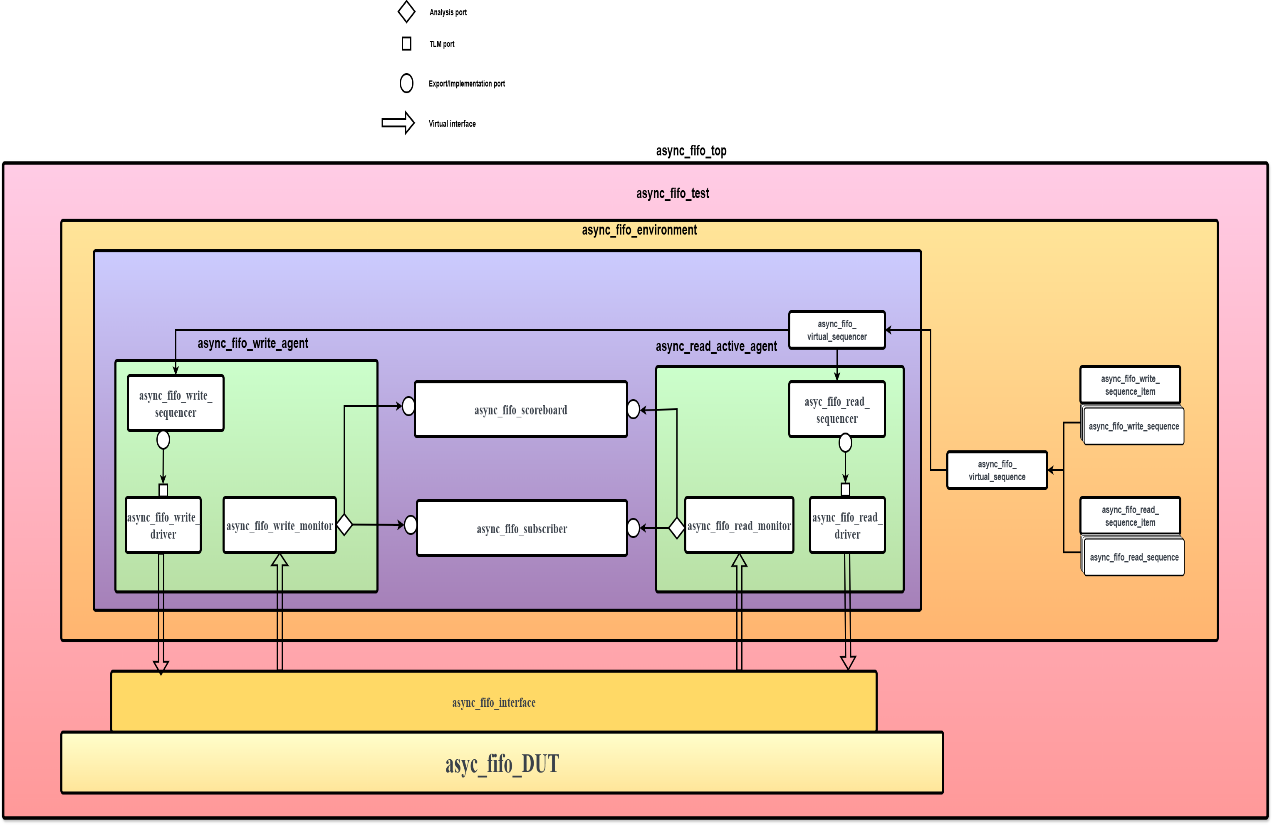
The Design Under Test (DUT) in Universal verification methodolgy is the hardware design or module being verified for correctness and functionality. It represents the actual RTL (Register Transfer Level) code that implements the desired digital circuit or system, written in hardware description languages like Verilog or SystemVerilog. The DUT sits at the center of the verification environment, receiving stimulus from testbenches, drivers, and verification components while producing responses that are monitored and checked by scoreboards and checkers.

The goal is to find any bugs or problems in the DUT before the hardware is actually built, since fixing errors after manufacturing is extremely expensive.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Sl.No | Pin Name | Direction | No. Of bits | Function |
| 1 | wclk | Input | 1 | Write clk. |
| 2 | rclk | Input | 1 | Read clk |
| 3 | wrst\_n | Input | 1 | Active low asynchronous write reset |
| 4 | rrst\_n | Input | 1 | Active low asynchronous read reset |
| 5 | winc | Input | 1 | Write increment |
| 6 | rinc | Input | 1 | Read increment |
| 7 | wdata | Input | 8 | Write data |
| 8 | rdata | Output | 8 | Read data |
| 9 | wfull | Output | 1 | Write full flag |
| 10 | rempty | Output | 1 | Read empty flag |

# CHAPTER 2 :Testbench Architecture

## 2.1 ASYNCHRONOUS FIFO Testbench Architecture



UVM is a standardized verification methodology based on SystemVerilog, offering a structured, reusable, and scalable functional verification strategy. UVM adopts a hierarchical component-based architecture with standardized base classes, communication paradigms, and verification strategies to develop solid testbench environments.

A standard UVM testbench contains a number of major components in a hierarchical layering: sequence items and sequences used for stimulus generation, agents with drivers and monitors, scoreboards for checking results, and an overall environment that coordinates all the components.

## 2.2 Component details and Flowchart

### UVM Sequence Item

The sequence item inherits from uvm\_sequence\_item and contains all the input and output signals of the Design Under Test (DUT), apart from clock and reset signals that are processed at the interface level. The sequence item provides intrinsic constraint blocks based on SystemVerilog's constraint-random verification capabilities, automation macros such as uvm\_field\_int for printing and copying, and stimulus generation control methods.

### UVM Sequence and Sequencer

UVM sequences extends from uvm\_sequence and create random sequence items (transactions) for the DUT. The sequencer (uvm\_sequencer) serves as an arbiter between the driver and multiple sequences and controls the sequence item flow. Sequences employ the start\_item(), randomize(), and finish\_item() protocol to create and pass sequence items to the driver via the sequencer. The sequencer offers features such as sequence prioritization, sequence arbitration, and layering of sequences.

### UVM Driver

UVM driver is extended from uvm\_driver and translates high-level sequence items into pin-level signals on the DUT interface. It gets in touch with the sequencer through TLM ports to send sequence items with the seq\_item\_port. The driver executes the DUT's protocol timing and signal transitions via virtual interfaces, and runs in the run\_phase() where it keeps retrieving sequence items from the sequencer and driving them to the DUT.

### UVM Monitor

The UVM monitor inherits from uvm\_monitor and passively monitors DUT signals, translating pin-level activity back into sequence items. The monitor captures transactions on the inputs and outputs of DUT without interfering with the operation of the DUT. The monitor utilizes virtual interfaces to sample signals and reconstruction methods to form complete transactions and broadcasts the transactions to other modules via TLM analysis ports (uvm\_analysis\_port).

### UVM Agent

The UVM agent inherits from uvm\_agent and packages associated verification components (sequencer, driver, and monitor) for an interface. Agents may be made active (sequencer and driver) or passive (monitor only) using the is\_active config field. The agent offers a modular verification component structure and facilitates seamless reuse in various testbench environments.

### UVM Scoreboard

The UVM scoreboard inherits from uvm\_scoreboard and the checks the current operation. It accepts actual results from monitors via TLM analysis imports (uvm\_analysis\_imp) and checks them against the expected results from a reference model or predictor. The scoreboard holds transaction fifo’s, executes comparison algorithms, and exports detailed mismatch and verification status reporting.

### UVM Subscriber

The uvm\_subscriber is a specialized UVM component class that acts as a **listener or observer** in the verification environment. Subscribers are basically listeners of an analysis port. They subscribe to a broadcaster and receive objects whenever an item is broadcasted via the connected analysis port.A uvm\_component class does not have an in-built analysis port, while a uvm\_subscriber is an extended version with an integrated uvm\_analysis\_export that automatically handles TLM communication.

### UVM Environment

The UVM environment inherits from uvm\_env and is the highest-level container for all verification blocks. It creates and initializes agents, scoreboards and other verification IP. The environment manages component interconnections via TLM ports, manages configuration parameters via the configuration database (uvm\_config\_db), and controls the overall verification flow between phases.

### UVM Test

**The UVM test is an extension of uvm\_test and incorporates customized test scenarios through setup of the environment and identification of which sequences to execute. Tests initialize the verification environment setup, instantiate and set up the environment instance, and manage the sequence execution through direct sequence start on sequencers. Several test classes can be established to test various aspects of the DUT behavior.**

### UVM Testbench Top

The testbench top is a module that instantiates the DUT, interface, and executes the UVM test. It provides clock and reset signals, creates instances of the interface, connects the interface to the DUT, puts the interface into the configuration database with uvm\_config\_db::set(), and starts the execution of the UVM test with run\_test(). It acts as the gateway between the module-based design universe and class-based UVM verification environment.

# CHAPTER 3:Verification Results

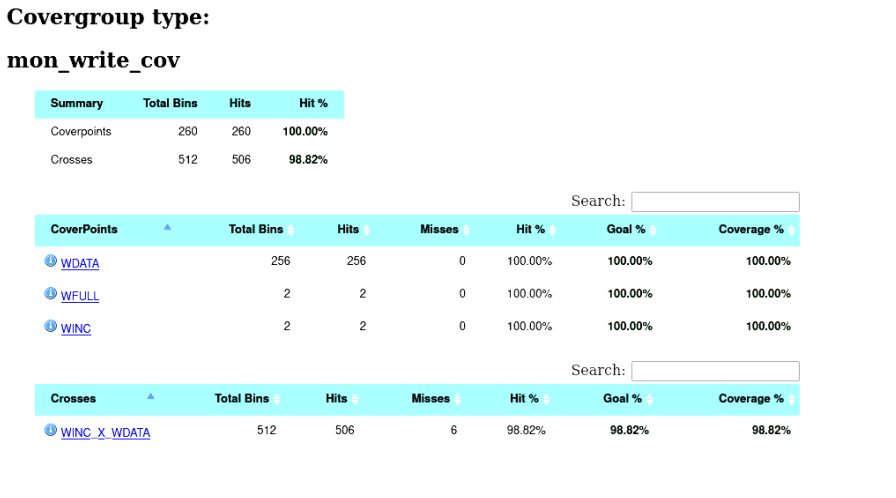
## 3.1 Errors in the DUT

1. When rrst is asserted rdata does not cleared to zero
2. Rempty is asserted initially even there is a data on the fifo
3. Targeted test employing a high write-to-read clock ratio (WCLK: 100 MHz, RCLK: 50 MHz)

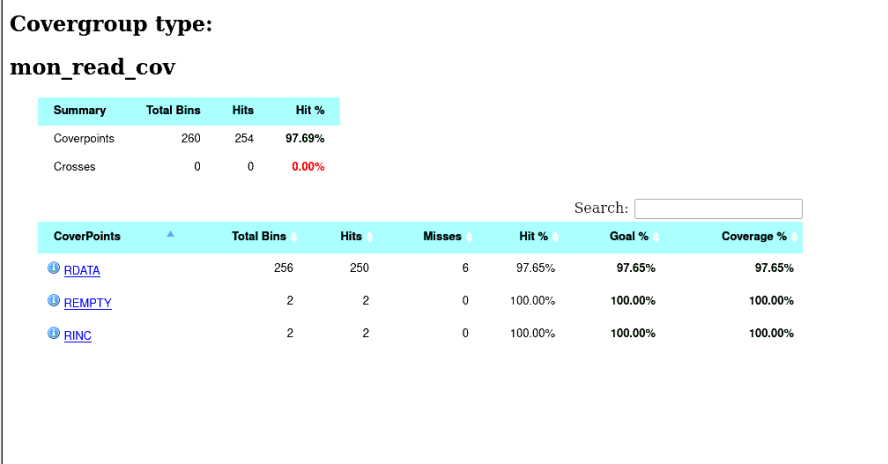
The FULL flag failed to assert when the FIFO depth reached its capacity limit (or near limit) during the burst write sequence immediately.

## 3.2 Coverage Report

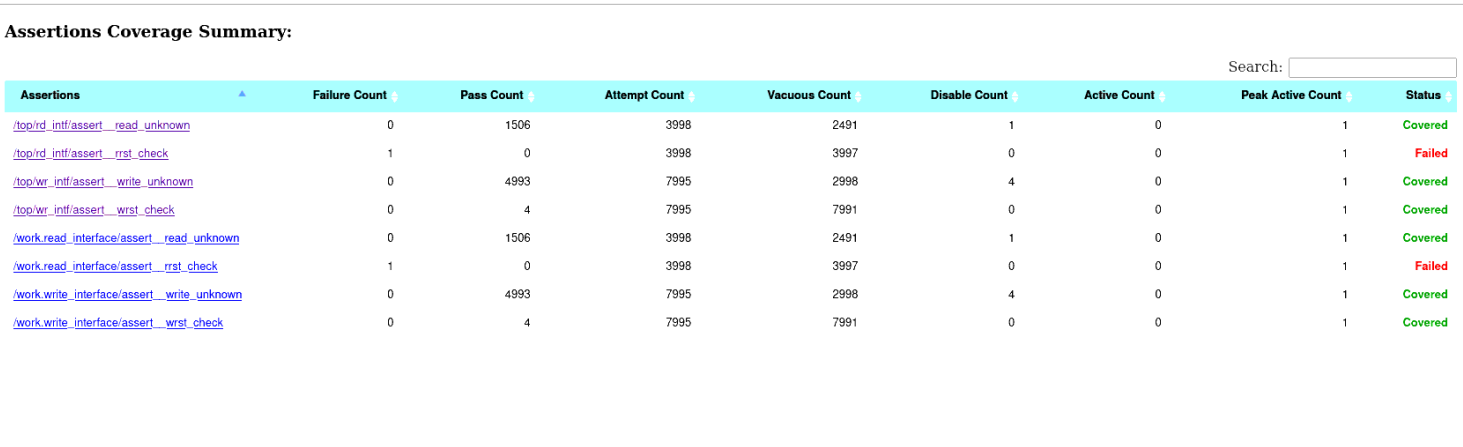
### Write transaction coverage



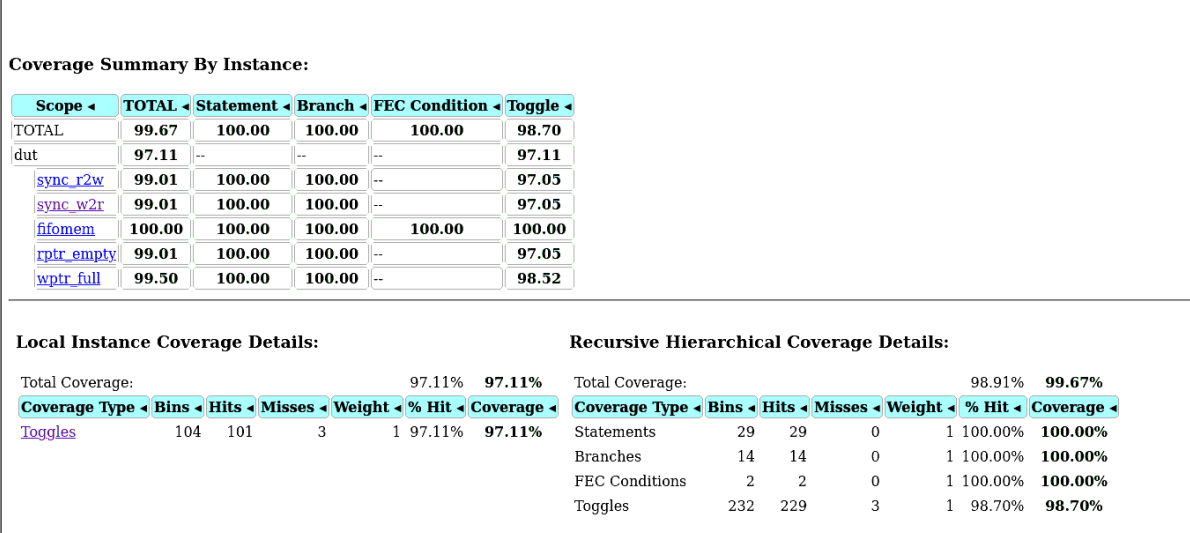
### Read transaction coverage



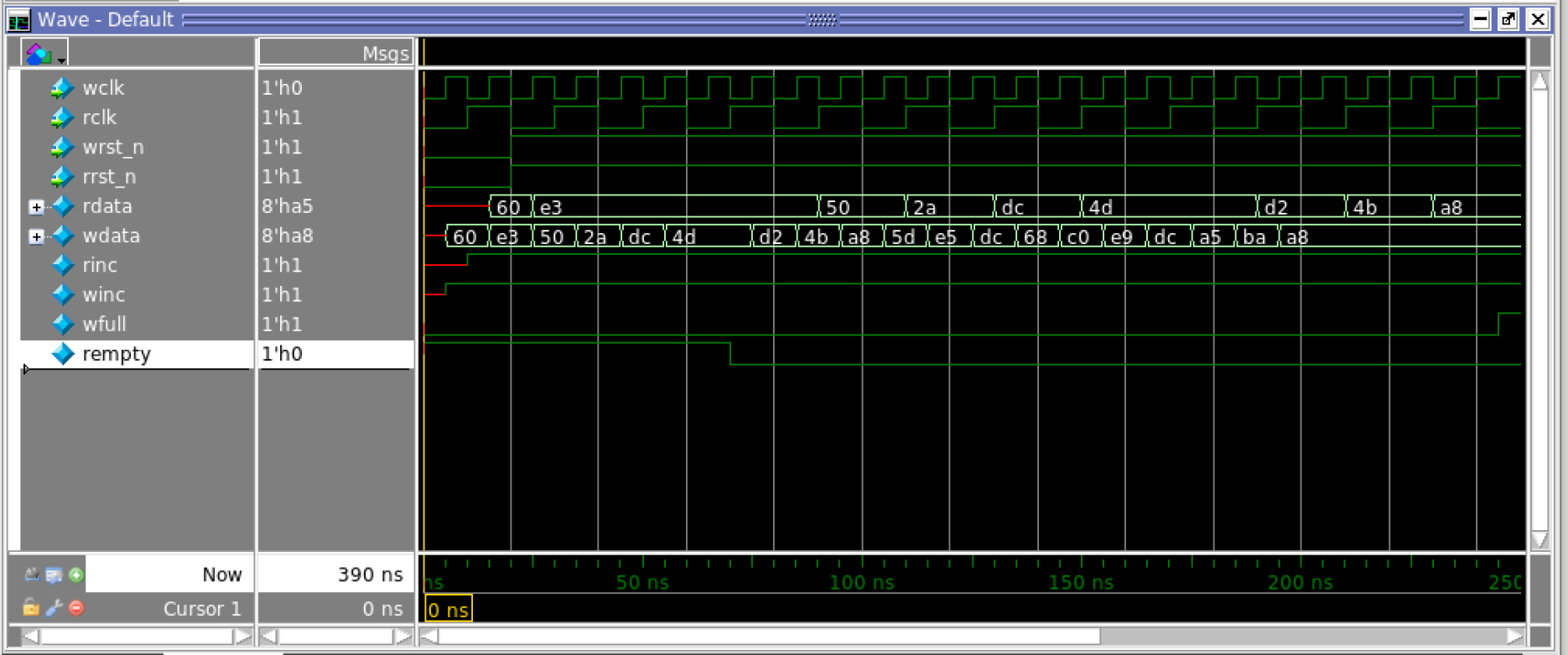
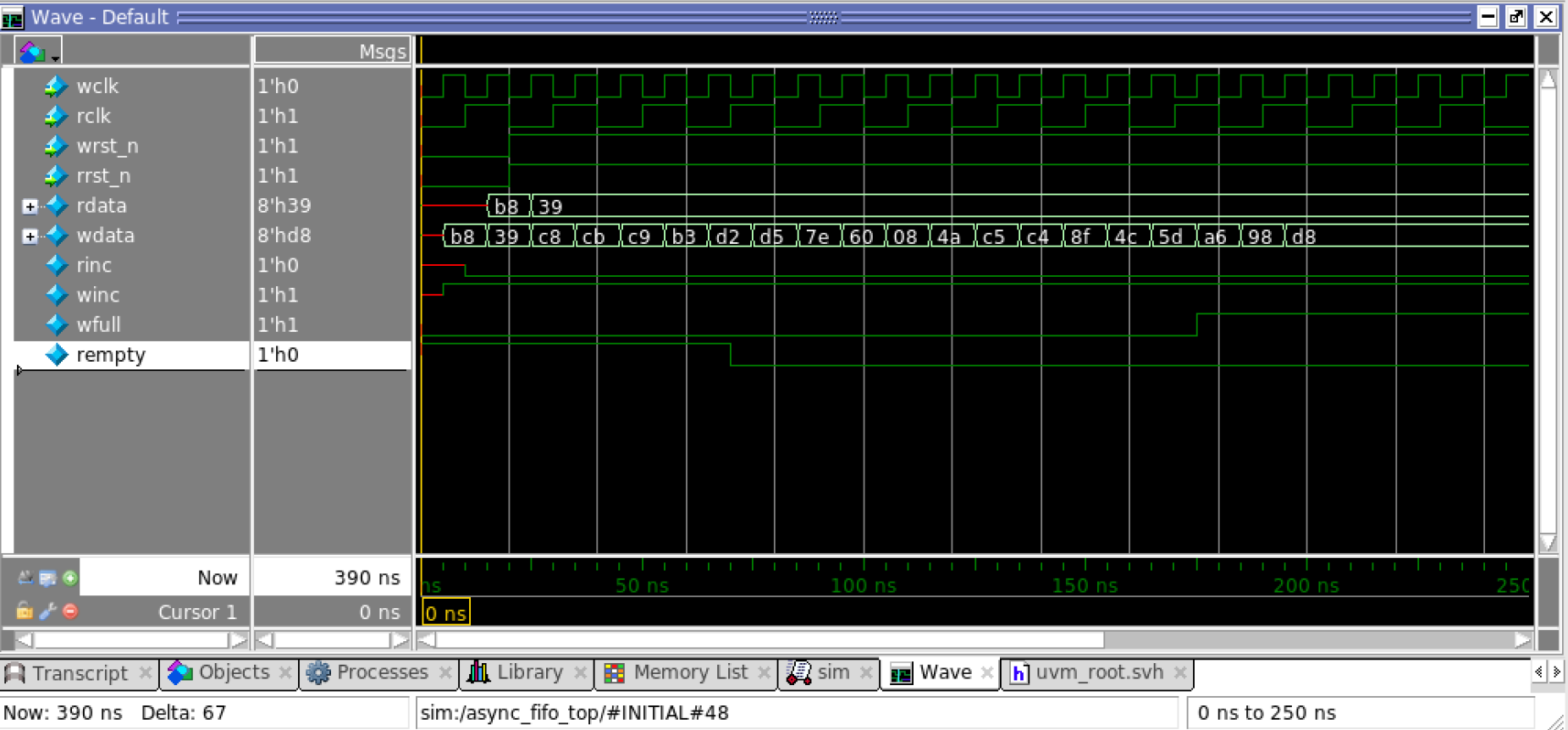
### Assertion coverage



### Overall coverage



### Output waveform

1. **Continous read and write with winc=1 and rinc=1**
2. **Continous write but no read with winc=1 and rinc=0**
3. **Continous read but no write with winc=0 and rinc=1**

