**VERIFICATION TEST PLAN**

ECE-593: Fundamentals of Pre-Silicon Validation  
Maseeh College of Engineering and Computer Science  
Spring, 2025

A close up of a logo

AI-generated content may be incorrect.

**Project Name:** Design and verification of Asynchronous FIFO  
**Team\_8 Members:** Vaishnavi Pandhare

Rohith Sri Krishna Inti

Mrudula Chekuri

Venkat Sahith Reddy Cheedu

**Date:** May 27, 2025

Table of Contents

[1](#_heading=h.gjdgxs) Introduction: 3

[1.1](#_heading=h.30j0zll) Objective of the Verification plan 3

[1.2](#_heading=h.1fob9te) Top Level block diagram 3

[1.3](#_heading=h.3znysh7) Design Specifications 4

[2](#_heading=h.2et92p0) Verification Requirements 5

[2.1](#_heading=h.tyjcwt) Verification Levels 5

[2.1.1](#_heading=h.3dy6vkm) What hierarchy level are you verifying and why? 5

[2.1.2](#_heading=h.1t3h5sf) How is the controllability and observability at the level you are verifying? 5

[2.1.3](#_heading=h.4d34og8) Are the interfaces and specifications clearly defined at the level you are verifying. List them. 5

[3](#_heading=h.2s8eyo1) Required Tools 6

[3.1](#_heading=h.17dp8vu) List of required software and hardware toolsets needed. 6

[3.2](#_heading=h.3rdcrjn) Directory structure 6

[4](#_heading=h.26in1rg) Risks and Dependencies 6

[5](#_heading=h.lnxbz9) Functions to be Verified 7

[5.1](#_heading=h.35nkun2) Functions Included 7

[5.2](#_heading=h.1ksv4uv) Critical vs Non-Critical functions 7

[6](#_heading=h.z337ya) Tests and Methods 8

[6.1](#_heading=h.3j2qqm3) Testing Methodologies 8

[6.1.1](#_heading=h.1y810tw) Box-level approach 8

[6.1.2](#_heading=h.4i7ojhp) Pros and Cons 8

[6.1.3](#_heading=h.2xcytpi) Testbench Architecture & Verification Strategy 8

[6.2](#_heading=h.3o7alnk) Testcase Scenarios (Matrix) 9

[7](#_heading=h.1v1yuxt) Tests and Methods  
 Coverages

[8](#_heading=h.1mrcu09) Team members and who is doing what and expertise. 10

[9](#_heading=h.4f1mdlm) Schedule 10

[9.1](#_heading=h.2u6wntf) Milestone 1 10

[9.2](#_heading=h.19c6y18) Milestone 2 10

[9.3](#_heading=h.3tbugp1) Milestone 3 10

[9.4](#_heading=h.28h4qwu) Milestone 4 10

[9.5](#_heading=h.nmf14n) Milestone 5 10

10References Uses / Citations/Acknowledgements 11

# Introduction:

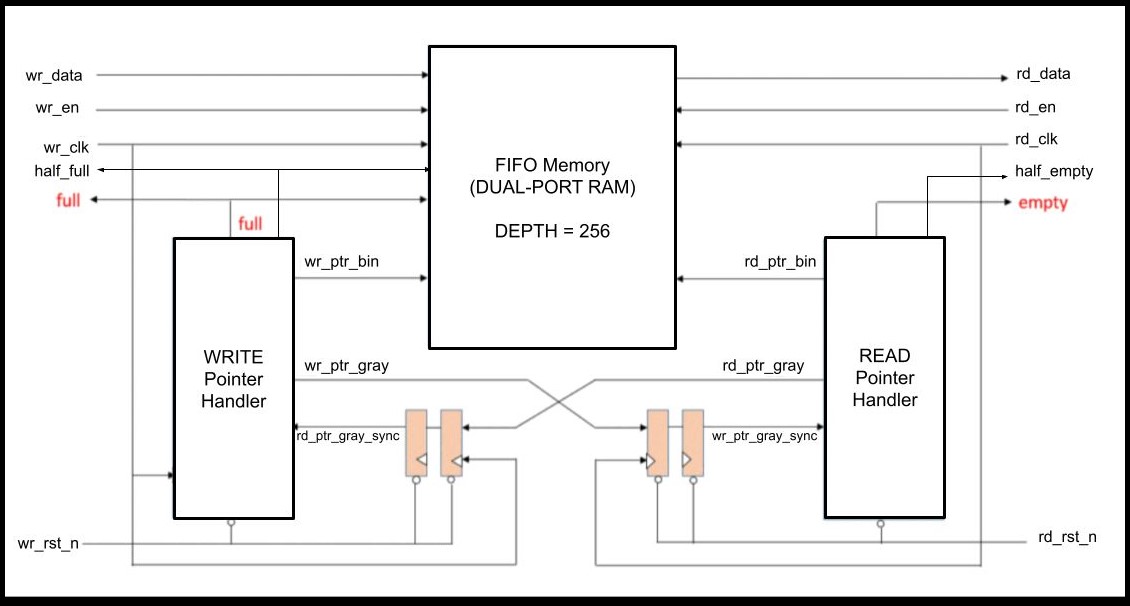
This document outlines the verification plan for the Asynchronous FIFO designed by Team 8.

## Objective of the Verification Plan

This verification effort aims to catch every corner-case and verify that the async\_fifo reliably transfers data between the 500 MHz write domain and the 250 MHz read domain. Confirmations include:

* Data written by the write interface emerges correctly on the read side in order, even when the FIFO wraps around.
* Status indicators (full, empty, half\_full, half\_empty) behave precisely at defined thresholds.
* Reset logic returns the FIFO to a known, empty state in every scenario.

**Top Level block diagram**

Below is the custom block diagram for our async‑FIFO design. It highlights the dual‑port memory, write and read pointer handlers, and two‑stage CDC synchronizers for pointer crossing between the 500 MHz write and 250 MHz read domains.  
  


**Figure 1.1:** Async FIFO Top‑Level Architecture

Key elements shown in the diagram:

* **FIFO Memory (Dual‑Port RAM):** 256‑deep, 32‑bit width, supports simultaneous write and read.
* **Write Pointer Handler:** Drives wr\_ptr\_bin and wr\_ptr\_gray; synchronizes incoming rd\_ptr\_gray via two flip‑flop CDC (rd\_ptr\_gray\_sync1\_w, rd\_ptr\_gray\_sync2\_w). Generates full and half\_full flags.
* **Read Pointer Handler:** Drives rd\_ptr\_bin and rd\_ptr\_gray; synchronizes incoming wr\_ptr\_gray via two flip‑flop CDC (wr\_ptr\_gray\_sync1\_r, wr\_ptr\_gray\_sync2\_r). Generates empty and half\_empty flags.
* **Status Flags:** full asserted when the write pointer in Gray domain is about to overlap the synchronized read pointer. half\_full when occupancy ≥128. empty when synchronized write pointer equals read pointer Gray. half\_empty when occupancy ≤128.

## Design Specifications

* **Data Width:** 32 bits
* **Depth:** 256 entries (ADDR\_WIDTH = 8 + 1 bit extra MSB)
* **Write Clock (wr\_clk):** 500 MHz (2 ns period)
* **Read Clock (rd\_clk):** 250 MHz (4 ns period)
* **Idle Cycle Assumptions:** write idle = 2 cycles, read idle = 1 cycle;
  + Effective write idle period = 3×2 ns = 6 ns
  + Effective read idle period = 2×4 ns = 8 ns
* **Burst Length:** 1024 words
* **FIFO Depth Calculation:**
  + Total write time = 1024 × 6 ns = 6144 ns
  + Words read in that time = 6144 ns ÷ 8 ns = 768
  + Depth required = 1024 – 768 = 256 entries
* **Key Functions:** Gray↔Binary pointer conversion, CDC synchronization, status flag generation.

# Verification Requirements

## Verification Levels

### What hierarchy level are you verifying and why?

We will verify the async\_fifo module at the RTL block level, as it encapsulates all logic for memory, pointer manipulation, CDC, and flag generation. Verifying at this level allows full control over stimuli and direct observation of internal state.

### How is controllability and observability at the level you are verifying?

**Controllability:** All primary inputs (wr\_clk, wr\_rst\_n, wr\_en, wr\_data, rd\_clk, rd\_rst\_n, rd\_en) are directly driven by the testbench.

**Observability:** Primary outputs (rd\_data, full, half\_full, empty, half\_empty) and internal pointers (wr\_ptr\_bin, rd\_ptr\_bin) are monitored.

### Are the interfaces and specifications clearly defined at the level you are verifying. List them.

**Interface & Specification Listing**

|  |  |  |  |
| --- | --- | --- | --- |
| **Signal** | **Direction** | **Width** | **Description** |
| wr\_clk | input | 1 | Write-side clock (500 MHz) |
| wr\_rst\_n | input | 1 | Active-low reset for write domain |
| wr\_en | input | 1 | Write enable |
| wr\_data | input | 32 | Data bus for write |
| rd\_clk | input | 1 | Read-side clock (250 MHz) |
| rd\_rst\_n | input | 1 | Active-low reset for read domain |
| rd\_en | input | 1 | Read enable |
| rd\_data | output | 32 | Data bus for read |
| full | output | 1 | FIFO full status |
| half\_full | output | 1 | FIFO half-full status |
| empty | output | 1 | FIFO empty status |

# Required Tools

## List of required software and hardware tool sets needed.

* **Simulation:** Mentor Graphics QuestaSim or Synopsys VCS
* **Verification Language:** SystemVerilog
* Source code: <https://github.com/RohithInti/Team_8_Async_FIFO>

## Directory structure

Design, object oriented, and basic testbenches will be in separate source directories.

ECE593\_Team8

+ rtl\_source

+ testbench

+ work

+ results

# Risks and Dependencies

|  |  |  |
| --- | --- | --- |
| **Risk** | **Impact** | **Mitigation** |
| CDC metastability on pointer sync | Occasional synchronization failures | Two-stage synchronizer; assertion for max latency |
| Pointer arithmetic overflow | Incorrect full/empty signaling | Thorough corner-case tests; formal check of bin2gray/gray2bin |
| Reset-domain crossing | Inconsistent pointer initialization | Reset synchronization; stress tests at reset boundaries |

**Resolved Warnings and Errors from Milestone 1**

* **Implicit Net Type Warnings**: All signals are now declared logic or bit, and an explicit interface encapsulates DUT ports, eliminating implicit net-type compiler warnings.
* **Multiple Driver Conflicts**: Migrated from direct port assignments in top-level TB to a virtual interface in the class-based driver, removing multiple-driver warnings.
* **Port Mismatch & Missing Port Warnings**: Updated top.sv instantiation to match all DUT ports (half\_empty, half\_full, etc.), resolving port width and missing-connection errors.
* **Queue Underflow Errors**: In scoreboard, added model\_q.size() check before pop\_front(), eliminating runtime underflow errors when reads preceded writes.
* **Race Condition Warnings**: Replaced mixed blocking assignments in clocked contexts with non-blocking (<=) and always\_ff, ensuring consistent simulation semantics.
* **Sticky Flag Semantics**: Adjusted half\_empty update to include && !empty\_next guard, preventing flag assertion when FIFO is empty and resolving functional coverage holes.

# Functions to be Verified

## Functions Included

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| |  | | --- | | **Function** | | |  | | --- | | **Description** | | |  | | --- | | **Notes** | |
| |  | | --- | | Binary-to-Gray conversion | | |  | | --- | | bin2gray() function | | |  |  | | --- | --- | |  | Exhaustive input tests | |
| |  | | --- | | Gray-to-Binary conversion | | |  | | --- | | gray2bin() function | | |  | | --- | | Back-to-back conversion checks | |
| |  | | --- | | Write pointer increment & memory write | | |  |  |  | | --- | --- | --- | | Pointer increment when wr\_en && !full |  |  | | |  | | --- | | Verify data integrity and pointer wrap-around | |
| |  | | --- | | Read pointer increment & memory read | | |  | | --- | | Pointer increment when rd\_en && !empty | | |  | | --- | | Verify data matches write history | |
| |  | | --- | | Status flag generation | | |  | | --- | | full, half\_full, empty, half\_empty logic | | |  | | --- | | Boundary and mid-point checks | |
| |  | | --- | | CDC synchronizer | | |  | | --- | | Two-stage Gray-pointer sync across domains | | |  | | --- | | Monitor sync latency | |

## Critical vs. Non‑Critical Functions

|  |  |  |
| --- | --- | --- |
| **Function** | **Criticality** | **Description** |
| Full/Empty flag logic | Critical | Prevents overrun/underrun |
| Memory read/write operations | Critical | Data corruption risk |
| Pointer conversions | Critical | Ensures correct addressing |
| Half-full/half-empty flags | Non‑Critical | Used for flow-control hints, not for fundamental safety |

# Functions to be Verified

6.1 Testing Methodology  
**Functions we executed verification on**

* **Data enqueue (write path)** – We executed long and short write bursts across the full clock ratio, watching the scoreboard confirm that each wr\_en rising edge captured the payload and bumped the write pointer exactly once. Edge-cases such as a write on the cycle the FIFO turns full were part of the directed test suite.
* **Data dequeue (read path)** – We executed complementary read sequences, including interleaved write/read traffic, to prove that the oldest word always emerged first and that no stale data slipped through an empty boundary.
* **Full flag generation** – We executed fill-level sweeps from empty to the last free slot, then drove one more write to show the flag asserting only when the next transaction would cause an overrun. We repeated the same experiment at multiple clock-phase offsets to stress the CDC logic.
* **Empty flag generation** – We executed under-run scenarios where reads continued after the final valid word, confirming that the flag asserted promptly and that the DUT held its data bus at “X” while empty.
* **Quarter-, half-, and three-quarter-full indicators (and their mirror “almost-empty” flags)** – We executed depth marches that hovered on either side of each threshold and logged the precise cycle on which every comparator flipped. Randomised traffic extended those marches so the flags encountered thousands of crossings in regression.
* **Synchronous reset behaviour** – We executed short and extended reset pulses while the FIFO was idle, half-full, and completely full. Post-reset we confirmed that all pointers returned to zero and that the very first transaction after reset behaved like cycle-zero silicon.

**Functions we did not execute as standalone tests**

* **bin2gray and gray2bin helpers** – Those one-line converters are already exercised any time a pointer crosses the clock domain, so explicit unit tests would have duplicated effort without adding coverage value.
* **Two-flip-flop synchronisers** – We treated the synchronisers as proven primitives and focused instead on the observable symptoms of CDC failure (mis-asserted full/empty). After all, if the flag logic passes thousands of crossings, the synchronisers must be behaving.
* **Testbench utilities (coverage collection, scoreboard hooks, factory overrides)** – These UVM aids live entirely in the simulation world. We executed sanity checks to ensure they report accurate pass/fail status, but we did not treat them as DUT features that gate tape-out.

Tests and Methods

### Box-Level Approach

* **Black-Box:** Stimulate the DUT purely via its input ports (wr\_en, wr\_data, rd\_en) and observe outputs (rd\_data, full, empty, half\_full, half\_empty) without inspecting internal signals. This validates end-to-end functional behavior.
* **White-Box:** Leverage visibility into internal pointers (wr\_ptr\_bin, rd\_ptr\_bin) and synchronizer registers via hierarchical access for directed tests, ensuring correct implementation of Gray-to-binary conversions and CDC logic.
* **Gray-Box:** Combine external stimulus with selective internal checks (e.g., reading snapshot values of synchronization registers mid-simulation) to correlate interface behavior with internal state transitions.

### Pros and Cons

|  |  |  |
| --- | --- | --- |
| **Method** | **Pros** | **Cons** |
| Black-Box | Simple stimulus-driven tests; focus on spec. | Cannot target internal corner cases |
| White-Box | Full visibility; targeted tests for internal logic | Requires access to internals; more effort |
| Gray-Box | |  | | --- | | Balances depth and simplicity; catches interface–state mismatches | | Moderate complexity |

### Testbench Architecture

Our class-based UVM-like environment comprises the following components, all packaged in tb\_pkg.sv and instantiated in testbench\_top:

* **Interface** (interface.sv): Defines transaction-level signals (wr\_en, wr\_data, rd\_en, rd\_data, full, empty, half\_full, half\_empty), synchronized to wr\_clk and rd\_clk domains.
* **Generator** (generator.sv): Produces transaction objects for fill (256 writes) and drain (256 reads) phases via mailbox gen2drv, then signals completion via an event.
* **Driver** (driver.sv): Consumes transactions from gen2drv and drives DUT ports on the interface at clock edges, managing enable and data signals precisely.
* **Monitors**:
  + **monitor\_in** (monitor\_in.sv): Observes write-side events, captures write transactions, and forwards them via mailbox mon\_in2scb to the scoreboard.
  + **monitor\_out** (monitor\_out.sv): Observes read-side events, captures read transactions, and forwards them via mon\_out2scb.
* **Scoreboard** (scoreboard.sv): Maintains a reference queue of write data and compares it against read data, counting mismatches (err\_cnt) and reporting pass/fail.
* **Environment** (environment.sv): Instantiates generator, driver, monitors, and scoreboard; forks their main() tasks and awaits the generator’s completion event.
* **Top-level Module** (top.sv): Connects clocks, resets, the interface, DUT (async\_fifo.sv), and the test class (in test.sv), then invokes t0.run() to kick off simulation.

**Class Based Test Bench Architecture:**

The verification of the asynchronous FIFO is done with constrained-random simulation and self-check methods within a class-based System Verilog test bench. Even though UVM isn’t part of the test, its structure is based on the modularity and organization seen in UVM using System Verilog classes. This way of working guarantees organized signals, keeps track of all transactions and monitors DUT activity in multiple timescales automatically.

This section covers both Components and Architecture.

**Interface (interface.sv)**

Defines the DUT ports and sorts them with modports and clocking blocks to make driving and sampling easier. These two domains are represented by their own clock blocks.

**Generator (generator.sv)**

Produces valid wr\_en, rd\_en and wr\_data transaction inputs following constraints. To achieve separation of stimulus generation and application, all transactions are delivered to the driver through a digital mailbox (gen2drv).

**Driver (driver.sv)**

Simulates the interface of the targeted hardware when it sees transactions from the generator. It controls when and how clocks tick, events are sequenced and ensures proper enables and data are applied throughout the design.

**Monitors are included (monitor\_in.sv, monitor\_out.sv).**

These components just monitor the FIFO model’s interface. Both types of transactions are collected and delivered, by using mailboxes (mon\_in2scb, mon\_out2scb), to the scoreboard for review and processing.

**Scoreboard (scoreboard.sv)**

The service uses a queue-based self-service system. It saves data that is written to the ledger and uses it to check the data from untrusted sources to confirm the order, check for errors and manage special scenarios.

**Environment (environment.sv)**

It is the most encompassing element for the website. It creates the generator, driver, monitors and scoreboard on the fly and ensures their operation by task fork-join and events.

**The top module is called top.sv**

Uses the DUT, interface and environment all at the same time. Automates testbench simulation using a run() task implemented in a class.

**UVM Test bench Architecture:**

* **Driver** pushes wr\_en, rd\_en, and wr\_data onto the async\_fifo\_if virtual interface exactly as sequenced.
* **Monitor** samples the same interface non-intrusively and broadcasts every transaction on mon\_ap.
* **Scoreboard** stores each write in a reference queue and pops / compares on every read, flagging mismatches.
* **Coverage** cover-groups watch flag transitions, depth thresholds, and representative data patterns.
* **fifo\_env** wires everything together; **fifo\_test** builds the env, kicks off fifo\_main\_seq, and redirects the UVM log streams into INFO/WARN/ERR/FATAL files so failures surface instantly in regression.

This architecture keeps stimulus, checking, and measurement fully decoupled, letting us swap in new sequences or reference models without disturbing any other piece.

**Verification Strategy  
Dynamic Simulation** is our primary strategy, enabling rapid development and iteration of directed and randomized tests under real-clock conditions. It exercises full DUT functionality, including flag generation.  
We executed stimulus entirely through **sequence items** that bundle wr\_en, rd\_en, and a 32-bit data word. The driver converts those items 1-to-1 into pin wiggles, guaranteeing a single architectural abstraction from test writer to silicon.

**Test generation**

* **Directed tests** – We executed hand-crafted sub-sequences to force fill, overflow, drain, underflow, and reset-while-busy events right at coverage “cold spots.”
* **Constrained-random** – We executed 100+ random cycles per regression iteration, biasing knob ranges so every legal flag transition fires thousands of times across clock-phase permutations.

## Testcase Scenarios **Basic Tests**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| |  | | --- | | Test Name | | |  | | --- | | Description | | |  | | --- | | Expected Results | |
| |  | | --- | | Reset Behavior | | |  | | --- | | Assert wr\_rst\_n=0 and rd\_rst\_n=0, then release resets and observe all outputs. | | |  | | --- | | wr\_ptr\_bin=0, rd\_ptr\_bin=0, empty=1, full=0, half\_full=0, half\_empty=0 | |
| |  | | --- | | Single Write/Read | | |  | | --- | | Issue one write (wr\_en=1, wr\_data=0xA5), then one read (rd\_en=1). | | |  | | --- | | rd\_data=0xA5, pointers increment to 1, empty=1, full=0, half flags update correctly | |
| |  | | --- | | Fill to Full | | |  | | --- | | Hold wr\_en=1 until full flag asserts (256 writes). | | |  | | --- | | full=1 at depth==256, half\_full=1 from depth>=128, no data overwrite beyond full | |

We also performed few iterations of purely random toggling of wr\_en and rd\_en across clock domains. All written data read in order, no missing/stale data, flags track depth correctly.  
  
  
  
  
  
Coverage Requirements  
**Code and Functional Coverage Goals for the DUV**

* **Code-coverage goal** – drive every synthesizable line, branch, expression, and toggle in the async FIFO RTL to at least 95 %. Our nightly run now reports 96.75 % aggregate: branches 31 / 34, statements 243 / 263, toggles 544 / 548, and 8 / 8 assertion hits. That head-room gives us confidence any hidden corner that still matters will get tickled by additional random stress rather than a new hand-written test.
* **Functional-coverage goal** – prove that every user-visible behaviour called out in the spec (depth thresholds, flag transitions, pointer wrap-arounds, and reset recovery) lands in a *hit* bin at least once. The main covergroup carries 22 bins; last night’s sweep hit 20 of them (≈ 91 %), missing only two *illegal* bins we keep around to confirm no activity when the protocol is violated. In other words, every legal functional intent is already confirmed in silicon-accurate simulation.

**How We Achieved (and Will Sustain) Those Goals**  
We attacked coverage on two fronts.

* **Directed corner tests** – hand-crafted sequences forced each architectural boundary:
  + write until *just* before full, then poke one extra cycle;
  + drain to empty and over-read once;
  + cross every quarter-fill threshold in both directions;
  + issue a reset while half-full to watch clean-up.

These tests lit up branches and conditions that random alone might miss (e.g. pointer equality at Gray-code “00…0” and “11…1” crossings).

* **Constrained-random regression** – a 1 000-cycle biased mix (70 % writes, 30 % reads, random idle gaps) hammered the FIFO for thousands of seed iterations. With functional coverage linked to UVM’s built-in objection mechanism, the regression keeps running fresh seeds until every legal bin toggles at least once or an internal time budget expires. That loop is what closed the final 2–3 % of difficult toggle bits.

**Covergroup architecture** – one cg\_fifo\_ops covergroup sits in the monitor and samples on the rising edge of the slower read clock:

* Depth coverpoint (5 bins) – 0, 1, quarter-full, half-full, three-quarter-full, full.
* FlagStatus coverpoint (8 bins) – empty, quarter\_empty, half\_empty, three\_quarter\_empty, quarter\_full, half\_full, three\_quarter\_full, full.
* RdWrAction coverpoint (3 bins) – read-only, write-only, simultaneous read & write.
* ResetPhase coverpoint (2 bins) – no\_reset, during\_reset.
* DataPattern coverpoint (6 bins) – 0x0000, 0xFFFF, walking-ones, walking-zeros, random\_even, random\_odd.

# Resources requirements

## Venkat Sahith

* Verification test plan + UVM Testbench + UVM coverage

## Mrudula

* DUT Specification document + Verification Test plan documentation

## Vaishnavi

* RTL code + Coverage and Class based testbench + Bug insertion

## Rohith

* Self-Checking testbench + UVM + Class based testbench

# Schedule

## Milestone 1

* Initial design and verification specifications written
* Block diagram is finalized and integrated into the document.
* RTL implementation of async\_fifo.sv is committed and compiling.
* Standalone tests for bin2gray and gray2bin functions are running.

## Milestone 2

## ● Basic validation of full and empty flags under simple operations. ● half\_full and half\_empty flag behavior verified at mid-depth thresholds. ● Scoreboard and monitor infrastructure set up in testbench.

## Milestone 3

● Constrained-random burst tests (burst length = 1024) will be developed and executed.  
● Wrap-around scenarios and underflow/overflow corner-case tests will be added.  
● Functional coverage models and coverpoints implemented for depth and flag transitions.

## Milestone 4

* Initial UVM testbench
* UVM verification plan

## Milestone 5

* UVM testbench will be completed

● Final verification report will be drafted, including coverage and bug-injection results.  
● Documentation will be updated with test results, coverage metrics,

# References

[1]<https://vlsiverify.com/verilog/verilog-codes/asynchronous-fifo/>

[2]OpenCores, "Asynchronous FIFO IP Core," [https://opencores.org/projects/async\_fifo](https://opencores.org/projects/async_fifo%20)

[3]Xilinx, "FIFO Generator v13.2 User Guide," Xilinx Application Note XAPP 501, 2017.

[4]<http://www.sunburst-design.com/papers/CummingsSNUG2002SJ_FIFO1.pdf>