

**VISVESVARAYA TECHNOLOGICAL UNIVERSITY, JNANASANGAMA,**

**BELAGAVI-590018**



**BLDEA'S V.P. Dr.P.G. HALAKATTI COLLEGE OF ENGINEERING AND TECHNOLOGY,  
VIJAYAPURA**



**DEPARTMENT OF  
ELECTRONICS AND COMMUNICATIONENGINEERING**

A Major Project report on

**“VLSI VERIFICATION OF FIFO REGISTER USING SYSTEM  
VERILOG”**

*Submitted in partial fulfillment for the award of degree of Bachelor of Engineering in Electronics and  
Communication Engineering*

**Submitted by**

SHANKAR R DEVAR  
SANKALP A PATIL  
SHRIVATSA S PARVATIKAR  
SAMARTH P BIRADAR

2BL21EC083  
2BL21EC080  
2BL21EC091  
2BL21EC076

**Under the Guidance of**

Prof.M.N. Deshmukh  
Prof. Vijaykumar Sajjanar

**2024-25**

**VISVESVARAYA TECHNOLOGICAL UNIVERSITY,  
BELAGAVI**

---



**B.L.D.E Association's**

**V.P Dr. P.G HALAKATTI COLLEGE OF  
ENGINEERING AND TECHNOLOGY, VIJAYAPURA**



**DEPARTMENT OF ELECTRONICS AND  
COMMUNICATION ENGINEERING**

**CERTIFICATE**

This is Certified that the Major project work entitled **“VLSI VERIFICATION OF FIFO REGISTER USING SYSTEM VERILOG”** carried out by **Mr.Shankar.R.Devar, Mr.Sankalp.A.Patil, Mr.Samarth.P.Biradar and Mr.Shrivatsa.S.Parvatikar** Bonafide students of **V.P Dr P.G Halakatti College of Engineering and Technology, Vijayapura** in partial fulfillment for the award of **Bachelor of Engineering in Electronics and Communication Engineering of the Visvesvaraya Technological University, Belagavi** during the year 2024-2025. It is certified that all corrections/suggestions indicated for External assessment have been incorporated in the report deposited in the departmental library. The Major project report has been approved as it satisfies the academic requirement in respect of Major project work prescribed for the said degree.

---

**Project Guide**  
**Prof.Vijaykumar Sajjanar**  
**Prof.M.N.Deshmukh**

---

**HOD**  
**Dr.U.D.Dixit**

---

**Principal**  
**Dr.V.G.Sangam**

**B.L.D.E. Association's  
VACHANA PITAMAHA Dr. P.G. HALAKATTI COLLEGE OF  
ENGINEERING & TECHNOLOGY, VIJAYAPURA**



DEPARTMENT OF ELECTRONICS & COMMUNICATION ENGINEERING

**DECLARATION**

We, students of Sixth semester B.E, at the department of Electronics & Communication Engineering, hereby declare that, the Major Project entitled “**VLSI VERIFICATION OF FIFO REGISTER USING SYSTEM VERILOG**”, embodies the report of our major project work, carried out by us under the guidance of Prof. **Vijaykumar Sajjanar & Prof. M.N.Deshmukh**, we also declare that, to the best of our knowledge and belief, the work reported here in does not form part of any other report or dissertation on the basis of which a degree or award was conferred on an earlier occasion on this by any student.

Place: -Vijayapura

Date: -

## **ACKNOWLEDGEMENT**

The satisfaction and euphoria that accompany the successful completion of any task would be incomplete without the mention of people who made it possible, whose consistent guidance and encouragement crowned our efforts with success. We consider it as our privilege to express the gratitude to all those who guided in the completion of our Major Project.

First and foremost, we wish to express our profound gratitude to our respected Principal **Dr. V.G. Sangam, B.L.D.E. Association's VACHANA PITAMAHA Dr. P.G. HALAKATTI COLLEGE OF ENGINEERING & TECHNOLOGY, Vijayapura**, for providing us with a congenial environment to work in.

We would like to express our sincere thanks to **Dr. U D Dixit**, the HOD of Electronics and Communication Engineering, **B.L.D.E. Association's VACHANA PITAMAHA Dr. P.G. HALAKATTI COLLEGE OF ENGINEERING & TECHNOLOGY, Vijayapura**, for his continuous support and encouragement.

We are greatly indebted to our guide **Prof. Vijaykumar Sajjanar & Prof. M.N. Deshmukh**, Department of Electronics and Communication Engineering, **B.L.D.E. Association's VACHANA PITAMAHA Dr. P.G. HALAKATTI COLLEGE OF ENGINEERING & TECHNOLOGY, Vijayapura**, who took great interest in our work. He motivated us and guided us throughout the accomplishment of this goal. We express our profound thanks for his meticulous guidance.

# ABSTRACT

A First-In-First-Out (FIFO) buffer is a critical component in digital systems, often used for temporary data storage and retrieval in communication interfaces, synchronization between different clock domains, and data buffering. Verifying the correct functionality and performance of a FIFO is essential to ensure system reliability. This project focuses on the functional verification of a FIFO design using System Verilog, leveraging its advanced features for constraint-driven stimulus generation, assertion-based checks, and functional coverage analysis. The FIFO design under verification includes configurable depth and width, synchronous or asynchronous modes, and support for key operations such as write, read, reset, and status flag generation (full, empty).

The verification process follows a systematic approach using a robust testbench developed in System Verilog. The testbench architecture includes essential components such as drivers, monitors, a scoreboard for result comparison, and a coverage model to evaluate the verification progress. Constrained-random testing ensures a broad range of input scenarios, while System Verilog assertions validate protocol compliance and key properties. Functional coverage metrics are collected to measure the completeness of the verification effort, with specific focus on boundary conditions, corner cases, and normal operations. Simulations are conducted using EDA PLAYGROUND tool and simulator used is Aldec Riviera Pro 2023.24.

This project aims to deliver a thoroughly verified FIFO design that meets functional specifications, a reusable and scalable testbench architecture for future verification tasks, and comprehensive coverage metrics to demonstrate the quality of the verification. The outcome highlights the capability of System Verilog as an effective language for verifying digital systems and ensures the FIFO design is ready for deployment in applications such as high-speed communication protocols, embedded systems, and data buffering between synchronous clock domains.

# INDEX

No	CONTENTS	PAGE NO
1	INTRODUCTION	1-5
2	LITERATURE SURVEY	6-8
3	METHODOLOGY	9-21
4	ADVANTAGES AND APPLICATIONS	22-27
5	RESULT	28-34
6	CONCLUSION	35-36
7	REFERENCE	37-38

# FIGURE LIST

Figure no	Figure
1.1	Block Diagram of Synchronous FIFO
1.2	Block Diagram of Asynchronous FIFO
3.1	System Verilog V/S UVM
3.2	Test Bench Architecture
5.1	Block Diagram of Write Control Logic
5.2	Block Diagram of Read Control Logic
5.3	System Verilog Waveform
5.4	Output Log I
5.5	Output Log II

# INTRODUCTION



## CHAPTER:1

# INTRODUCTION

### 1.1 Introduction to FIFO Verification Using System Verilog:

In modern digital systems, First-In-First-Out (FIFO) memory plays a crucial role in managing data flow between different clock domains or systems with varying processing speeds. The robustness and reliability of FIFO are critical in ensuring data integrity, making its verification an essential task in the design cycle.

SystemVerilog, with its powerful verification capabilities, offers a standardized and efficient approach to validate the functionality of FIFO designs. This project focuses on verifying a FIFO using SystemVerilog's advanced verification features, such as constrained random stimulus generation, functional coverage, and assertions.

### 1.2 What is FIFO?

FIFO stands for **First-In-First-Out**, a concept widely used in computer science, electronics, and digital system design. It refers to a data structure or buffer where the first piece of data that enters (is written into) the system is the first one to exit (be read out). This behavior mimics a queue, such as a line of people waiting where the person who enters first is served first.

### 1.3 Characteristics of FIFO:

- 1. Order Preservation:** The sequence of data output is the same as the sequence in which it was input.
- 2. Fixed Capacity:** FIFOs often have a predefined size or memory capacity.

#### Two Operations:

**Write (Enqueue):** Inserts data into the FIFO.

**Read (Dequeue):** Removes data from the FIFO.

### 1.4 Why is FIFO Important?

**Data Integrity:** Ensures data is processed in the correct sequence.

**Efficiency:** Handles asynchronous data streams or communication between different system components.

**Scalability:** Simplifies designs for systems requiring buffering, synchronization, or flow control.

## 1.5 First-in First-out (FIFO) Module

A First-In-First-Out (FIFO) module can be used for synchronization between different clock domains effectively solving the synchronization problem.

### Module Description

A FIFO module in a digital system helps in assisting with variable-rate data transfers or to hold/buffer data in the case of clock domain crossing to ensure no data loss in the system. Data loss would be a serious problem in any digital system and must be avoided.

- Shift register – FIFO associated with an invariable number of stored data words. It needs necessary synchronization between the read and the write operations. Also, a data word will be read each and every time a new data word is written.
- Exclusive read/write FIFO – FIFO with a variable number of stored data words. It also needs necessary synchronization between the read and the write operations (its internal structure - being the reason behind).
- Concurrent read/write FIFO – FIFO with a variable number of stored data words. It can support asynchronous read and write operations, hence giving rise to two sub types, Synchronous FIFO and Asynchronous FIFO. This FIFO design ensures synchronization between the source and the destination systems by utilizing the control signals for writing and reading. The two systems, source and destination, can operate at different frequencies.

## 1.6 Types of FIFOs :

### 1. Synchronous FIFO:

Both read and write operations are synchronized to the same clock.

Simpler design, suitable for systems where all operations use a single clock domain.

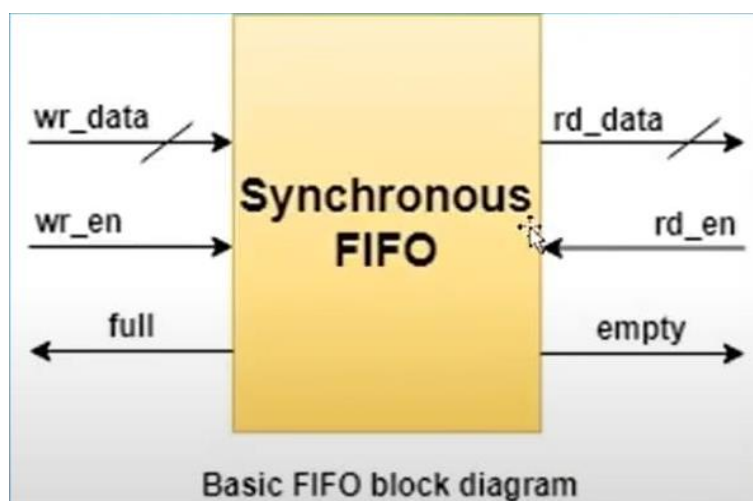


Figure1.1-Block diagram of Synchronous FIFO

## 2. Asynchronous FIFO:

Read and write operations occur in different clock domains.

Includes special techniques to handle clock domain crossing (e.g., Gray code pointers).

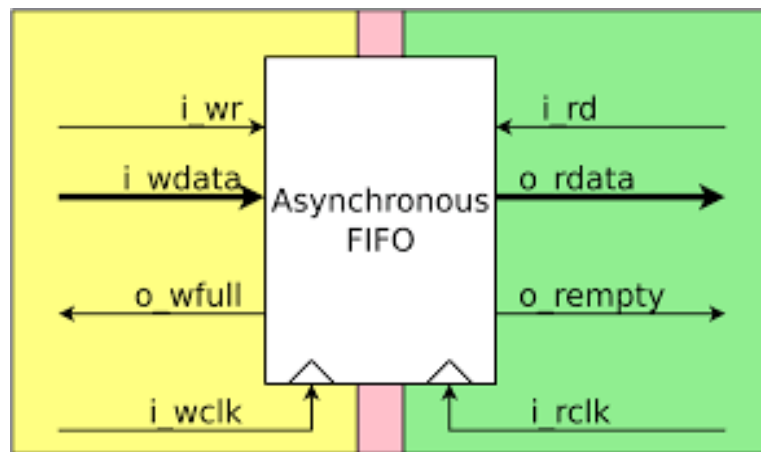


Figure 1.2-Block diagram of Asynchronous FIFO

## 1.7 Tools and Simulators

### EDA Playground

EDA Playground is an online platform for writing, simulating, and sharing hardware description language (HDL) code. It supports various design and verification tools, providing an easy-to-use interface for practicing and testing HDL-based designs.

### Simulation Tools Integration:

Includes support for major simulation tools, such as Aldec Riviera-PRO, Synopsys VCS, Cadence Xcelium, and ModelSim.

### Aldec Riviera-PRO 2023.24

Aldec Riviera-PRO is a high-performance, industry-standard HDL simulation and verification tool widely used for designing and verifying digital systems. Version 2023.24 is the latest release, equipped with advanced features to enhance simulation and debugging efficiency.

**Key Features of Aldec Riviera-PRO 2023.24:**

- Comprehensive Language Support: Fully supports Verilog, System Verilog, VHDL, PSL, and UPF standards.
- Advanced Debugging: Features waveforms, transaction viewers, and UVM-aware debug tools to simplify root cause analysis.
- Assertion-Based Verification: Supports SVA (System Verilog Assertions) and PSL (Property Specification Language) for formal and simulation-based verification.
- UVM and Functional Coverage: Seamlessly integrates with UVM (Universal Verification Methodology) and supports functional coverage collection and analysis.
- Multi-core Simulation: Utilizes parallelism to speed up simulation of large designs.
- Integration with EDA Tools: Works with synthesis and formal verification tools for end-to-end design verification.
- Enhancements in Version 2023.24: Improved runtime performance for mixed-language simulations.

# LITERATURE SURVEY

## CHAPTER: 2

# LITERATURE SURVEY

## 2.1 INTRODUCTION

The literature survey explores prior research, methodologies, tools, and techniques related to FIFO verification, focusing on the use of System Verilog. This section discusses existing work in the fields of FIFO design and verification, highlighting gaps addressed by the current project.

## 2.2 FIFO Design and its Importance

**FIFO Basics:** FIFO memory is widely used in digital systems to manage data transfer between components operating at different speeds or clock domains. Literature such as "Digital Design Principles and Practices" (Wakerly, 2006) outlines the core design principles, including write-read operations, pointers, and depth management.

**Challenges in FIFO Design:** Papers such as "Design and Challenges in FIFO-Based Systems" (IEEE, 2015) identify potential pitfalls like data loss during overflow/underflow and synchronization issues in asynchronous FIFOs.

**Application Areas:** FIFO is crucial in areas like SoC design, network routers, and multimedia systems, as highlighted in "A Survey on Hardware Data Buffers" (ACM, 2014).

## 2.3 Verification Techniques

**Traditional Verification Approaches:** Early verification methods relied on directed testing, which is labor-intensive and often fails to uncover corner cases.

**Modern Methodologies:** The introduction of hardware verification languages like SystemVerilog has revolutionized verification. The IEEE 1800 SystemVerilog Standard provides a foundation for designing reusable and robust verification environments.

**Constrained Random Testing:** As discussed in "Systematic Verification of Complex Digital Systems" (Springer, 2016), constrained random testing helps uncover edge cases efficiently compared to traditional methods.

**Assertions and Functional Coverage:** Papers like "Assertion-Based Verification: Benefits and Practices" (IEEE, 2013) emphasize the importance of assertions for validating temporal properties and functional coverage for ensuring test completeness.

## 2.4 Verification of FIFO

**Case Studies:** Research papers like "Functional Verification of FIFO Using SystemVerilog and UVM" (DesignCon, 2018) demonstrate practical applications of UVM in FIFO verification, highlighting challenges like pointer wrap-around and simultaneous read-write operations.

**Assertion-Based Verification:** The paper "Formal and Assertion-Based Techniques for FIFO Verification" (DVCon, 2019) underlines how SystemVerilog Assertions (SVA) can validate critical FIFO properties such as data integrity, correct sequence, and adherence to control signals.

**Coverage Metrics:** Studies such as "Achieving Verification Closure for FIFO Systems" (IEEE, 2020) stress the importance of functional and code coverage in determining verification completeness.

## 2.5. Tools and Industry Trends

**Automation in Verification:** Papers on automation, such as "Trends in Automated Testbench Generation" (IEEE, 2017), discuss how UVM automation can expedite verification tasks and reduce human error.

**Real-World Challenges:** Asynchronous FIFOs often require clock-domain crossing (CDC) verification, which introduces additional complexities, as described in "CDC Verification for Asynchronous FIFO Designs" (DAC, 2015).

## 2.6 Gaps Identified

Lack of unified approaches to handle both synchronous and asynchronous FIFO verification under a single framework.

Limited coverage on how to optimize constrained random generation for FIFO edge cases. Minimal emphasis on integrating functional coverage and assertions for seamless verification closure.

### Contribution of This Project

This project builds on prior research by:

Combining UVM methodologies with assertion-based techniques to ensure comprehensive coverage of FIFO properties. Addressing real-world scenarios, such as boundary conditions, overflow/underflow, and reset behavior.

Demonstrating a reusable and modular testbench architecture to streamline FIFO verification for varying design configurations.

# METHODOLOGY



## CHAPTER: 3

# METHODOLOGY

### 3.1 System Verilog Advanced Features

#### 3.1.1 Constraint-Driven Stimulus Generation

Constraint-Driven Stimulus Generation for a FIFO involves using System Verilog Randomization techniques, where specific constraints define the allowed values for the inputs to a FIFO. This method ensures a wide variety of test scenarios are generated while adhering to design specifications, improving verification coverage.

#### 3.1.2 Assertion-Based Verification

Assertion-Based Verification (ABV) is a methodology that uses assertions to validate the functional and temporal correctness of a design. Assertions monitor the behavior of the design under test (DUT) and raise errors when specific conditions or sequences are violated. This approach significantly improves verification quality by automating and formalizing the process of checking design properties.

#### 3.1.3 Functional Coverage analysis

Once simulation is complete, the coverage tool generates a report summarizing which scenarios have been exercised and which have not. Below is an example format:

Coverage Point	Description	Coverage (%)
wr_en	Write enable toggling	100%
rd_en	Read enable toggling	95%
wdata	Data ranges (low/mid/high)	87%
full_empty_cross	Cross between full and empty states	92%
wr_rd_cross	Interaction of write and read enables	90%

SYSTEM VERILOG	UVM
<ul style="list-style-type: none"> <li>• LANGUAGE FOR WRITING AND SIMULATING DIGITAL DESIGNS.</li> <li>• CUSTOM-BUILT TESTBENCH, DESIGNED MANUALLY BY THE USER.</li> <li>• LIMITED REUSABILITY; TESTBENCHES ARE USUALLY PROJECT-SPECIFIC.</li> <li>• TESTCASES ARE DIRECTLY TIED TO THE TESTBENCH.</li> <li>• CLASS OVERRIDING IS LIMITED AND REQUIRES MANUAL UPDATES.</li> <li>• CONFIGURATIONS ARE OFTEN HARDCODED OR MANUALLY SET.</li> <li>• LESS STANDARDIZED, MAKING COLLABORATION HARDER.</li> </ul>	<ul style="list-style-type: none"> <li>• A METHODOLOGY/Framework FOR ORGANIZING VERIFICATION.</li> <li>• PROVIDES A STANDARD STRUCTURE WITH PRE-DEFINED COMPONENTS.</li> <li>• HIGHLY MODULAR AND REUSABLE ACROSS MULTIPLE PROJECTS.</li> <li>• TESTCASES ARE WRITTEN SEPARATELY AS SEQUENCES, IMPROVING FLEXIBILITY.</li> <li>• USES A FACTORY PATTERN FOR DYNAMIC CLASS REPLACEMENT.</li> <li>• CENTRALIZED CONFIGURATION DATABASE FOR EASY ACCESS.</li> <li>• INDUSTRY-STANDARD, MAKING COMPONENTS EASIER TO SHARE.</li> </ul>

Figure 3.1-SystemVerilog V/S UVM

### 3.1 Universal Verification Methodology (UVM)

A UVM test bench contains verification components that are reusable. A verification component is said to be an encapsulated, configurable, ready-to-use verification environment for a portion of or the entire design module under test. Each verification component has its own set of elements for stimulating, driving, monitoring and collecting coverage information for the DUT. All these verification components are well connected by making use of object-oriented programming concepts in System Verilog [23] and help in efficient verification of the DUT. Eventually, the UVM test bench made for one project could be re-used and configured for another project based on the verification plan, thereby, reducing human effort in creating a test bench from the scratch.

### 3.2 Verification Methodology:

- **UVM Framework:** Ensures modularity, reusability, and scalability of the testbench.
- **Constrained Random Testing:** Stimuli are generated randomly within constraints to cover diverse scenarios, including edge cases.
- **Coverage-Driven Verification:** Functional coverage and code coverage metrics ensure verification completeness.
- **Assertion-Based Verification:** SystemVerilog Assertions (SVAs) validate design properties like FIFO order and signal integrity.

### 3.2.1 Tools:

- Simulation tools like Synopsys VCS, Cadence Xcelium, or ModelSim.
- Waveform viewers and coverage analyzers for debugging and verification closure.
- Blocks in the Verification Environment
- The verification environment is divided into modular blocks, as per the UVM

### 3.2.2 Architecture:

#### Generator:

```
// Constructor
function new(mailbox #(transactor) gen2driv);
    this.gen2driv = gen2driv; // Initialize mailbox
endfunction

// Main task to generate transactions
task main();
    repeat(repeat_count) begin
        trans = new(); // Create a new transactor object
        trans.randomize(); // Randomize the transaction fields
        gen2driv.put(trans); // Send the transaction to the mailbox
    end
endtask
endclass
```

#### 1. Class Purpose

The generator class is designed to:

- Generate randomized transactions (instances of the transactor class).
- Send transactions to a driver or another component via a mailbox for processing.

#### 2. Code Walkthrough

Fields/Properties:

1. rand transactor trans;
  - Declares a transaction object of type transactor.
  - Marked as rand, indicating it can be randomized using SystemVerilog's constrained randomization.
2. mailbox #(transactor) gen2driv;
  - A parameterized mailbox that holds transactions (transactor type).
  - Used for communication between the generator and the driver.
3. int repeat\_count;
  - Determines the number of transactions to be generated.
  - The repeat construct in the task uses this variable.

4. `int count`;
  - A general-purpose counter (unused in this snippet but may be used for logging or iterations).
5. The **generator class** is instantiated in the testbench, and the `gen2driv` mailbox is connected to the driver.
6. The **main task** is called to generate `repeat_count` random transactions.
7. Each transaction is:
  - Created dynamically (`trans = new();`).
  - Randomized to provide varied test scenarios.
  - Sent to the mailbox (`gen2driv.put(trans);`).
8. The **driver** retrieves these transactions from the mailbox and processes them (e.g., applying signals to the DUT).

### Driver:

- Sends input transactions (write/read requests) to the DUT based on the stimulus generated.
- Interacts with the DUT by asserting control signals like `write_enable`, `read_enable`,

```

class driver;
  virtual fifo_if vif;
  mailbox #(transactor) gen2driv;

  function new (virtual fifo_if vif, mailbox #(transactor) gen2driv);
    this.vif = vif;
    this.gen2driv = gen2driv;
  endfunction

  task reset();
    vif.DRIVER.driver_cb.rst <= 1;
    repeat (40) @(posedge vif.DRIVER.clk);
    vif.DRIVER.driver_cb.rst <= 0;
  endtask : reset

  task main();
    fork : main
      forever begin
        transactor trans;
        trans = new();
        gen2driv.get(trans);
        @(posedge vif.DRIVER.clk);

        if (trans.wr_en || trans.rd_en) begin
          if (trans.wr_en) begin
            vif.DRIVER.driver_cb.wr_en <= trans.wr_en;
            vif.DRIVER.driver_cb.rd_en <= trans.rd_en;
            vif.DRIVER.driver_cb.wdata <= trans.wdata;
            @(posedge vif.DRIVER.clk);
          end
          else
            begin
              vif.DRIVER.driver_cb.wr_en <= trans.wr_en;
              vif.DRIVER.driver_cb.rd_en <= trans.rd_en;
              trans.rdata = vif.MONITOR.monitor.rdata;
              @(posedge vif.DRIVER.clk);
            end
          end
        end
      end
    join_none : main
  endtask

```

The driver class represents the driver in a UVM-like testbench for a FIFO. The driver interacts with the DUT (Device Under Test) through the FIFO interface (fifo\_if), applying stimulus as directed by a transaction object (transactor)

- **Receive Transaction:**
  - A transaction (trans) is received from the gen2driv mailbox, which contains details of the operation, such as whether it is a **read** or **write** operation (wr\_en, rd\_en), and associated data (wdata, rdata).
- **Stimulus Application:**
  - **Write Operation:**
    - If wr\_en is asserted, the wdata signal is applied to the DUT along with wr\_en.
    - The driver waits for the next clock cycle (@(posedge vif.DRIVER.clk)).
  - **Read Operation:**
    - If rd\_en is asserted, the driver captures the rdata value from the MONITOR signals.
- **Forever Loop:**
  - The main task runs in a forever loop inside a **fork-join\_none** construct, continuously receiving transactions from the generator and applying them to the DUT.

## Monitor:

- Observes the DUT's inputs and outputs, collecting data for analysis.
- Ensures no unintended modifications are made to the input transactions.

```
class monitor;
virtual fifo_if vif;
mailbox #(transactor) rcvr2sb;

function new(virtual fifo_if vif, mailbox #(transactor) rcvr2sb);
    this.vif = vif;
    if (rcvr2sb == null) begin
        $finish;
    end else begin
        this.rcvr2sb = rcvr2sb;
    end
endfunction : new

task start();
    fork
        forever begin
            transactor trans;
            trans = new();
            $display("=====Mode of operation=====");
            @(posedge vif.MONITOR.clk);
            wait(vif.MONITOR.monitor.rd_en || vif.MONITOR.monitor.wr_en);

            @(posedge vif.MONITOR.clk);
            if (vif.MONITOR.monitor.wr_en) begin
                trans.wr_en = vif.MONITOR.monitor.wr_en;
                trans.rd_en = vif.MONITOR.monitor.rd_en;
                trans.wdata = vif.MONITOR.monitor.wdata;
                trans.full = vif.MONITOR.monitor.full;
                trans.empty = vif.MONITOR.monitor.empty;
                if (trans.empty)
                    $display("\wr_en=%h Memory is Full", vif.MONITOR.monitor.wr_en);
                else
                    $display("\wr_en=%h \rdata=%h", vif.MONITOR.monitor.wr_en,
vif.MONITOR.monitor.rdata);
            end else begin
                trans.rd_en = vif.MONITOR.monitor.rd_en;
                trans.wr_en = vif.MONITOR.monitor.wr_en;
                trans.rdata = vif.MONITOR.monitor.wdata;
                trans.full = vif.MONITOR.monitor.full;
                trans.empty = vif.MONITOR.monitor.empty;
                if (trans.empty)
                    $display("\rd_en=%h Memory is Empty", vif.MONITOR.monitor.rd_en);
                else
                    $display("\rd_en=%h \rdata=%h", vif.MONITOR.monitor.rd_en,
vif.MONITOR.monitor.rdata);
            end
            rcvr2sb.put(trans);
        end
        join_none
    endtask : start
endclass
```

The monitor class is designed to monitor the signals of a FIFO (First In First Out) buffer and capture the transactions that occur. It interfaces with the Design Under Test (DUT) through a virtual interface (fifo\_if), which provides access to the DUT's signals. The class also communicates with other components of the testbench via a mailbox (for sending transactor objects).

The task checks whether **write enable (wr\_en)** or **read enable (rd\_en)** is asserted, and captures the corresponding transaction

- **If wr\_en is asserted** (write operation):
  - It captures the values of wr\_en, rd\_en, wdata, full, and empty signals from the DUT.
  - If the memory is **full**, it displays a message indicating the full status. If not, it shows the write data (wdata).
- **If rd\_en is asserted** (read operation):
  - It captures the values of rd\_en, wr\_en, rdata, full, and empty signals from the DUT.
  - If the memory is **empty**, it displays a message indicating the empty status. If not, it shows the read data (rdata).

### Scoreboard:

```
class scoreboard;
  mailbox #(transactor) gen2driv; // Use mailbox with correct type
  mailbox #(transactor) rcvr2sb; // Use mailbox with correct type
  integer compare;

  function new(mailbox #(transactor) gen2driv, mailbox #(transactor) rcvr2sb);
    this.gen2driv = gen2driv;
    this.rcvr2sb = rcvr2sb;
  endfunction : new

  task start();
    transactor trans_rcv, trans;
    trans_rcv = new();
    trans = new();
    fork
      forever begin
        rcvr2sb.get(trans_rcv); // Get transaction from receiver mailbox
        gen2driv.get(trans);    // Get transaction from sender mailbox
        $display("===== Scoreboard =====");
        compare = 1'b0;

        if (trans.wr_en == trans_rcv.wr_en && trans.rd_en == trans_rcv.rd_en)
          compare = 1'b1;

        if (trans_rcv.full || trans_rcv.empty) begin
          if (trans_rcv.full)
            $display("wr_en=%0h Memory is Full", trans_rcv.wr_en);
          else
            $display("wr_en=%0h wdata=%0h", trans_rcv.wr_en, trans_rcv.wdata);
        end

        if (trans_rcv.empty)
          $display("rd_en=%0h Memory is Empty", trans_rcv.rd_en);
        else
          $display("rd_en=%0h rdata=%0h", trans_rcv.rd_en, trans_rcv.rdata);

        if (compare == 1)
          $display("Yes");
        else
          $display("No");
      end
    endfork
  endtask
endclass
```

- A reference model that keeps track of expected FIFO behavior.
- Compares the actual DUT outputs against the expected results to identify mismatches.

This code defines a **SystemVerilog class called scoreboard** that is designed to compare the transactions received from two mailboxes and check whether they match, i.e., whether the expected behavior of the **Design Under Test (DUT)** matches the generated transaction values.

- **Creating Transactions:**
  - `trans_rcv` and `trans` are two instances of the transactor class. `trans_rcv` receives the transaction from the receiver mailbox, and `trans` holds the transaction for comparison.
- **Fork-Join:**
  - The fork block is used to create a parallel execution flow. The code inside fork runs concurrently with other parts of the testbench.
  - The forever loop ensures that the process continues indefinitely, receiving and comparing transactions as they arrive.
- **Fetching Transactions:**
  - `rcvr2sb.get(trans_rcv);` This line retrieves a transaction from the receiver's mailbox.
  - `gen2driv.get(trans);` This line retrieves a transaction from the driver's mailbox.
- **Comparison Logic:**
  - `trans.wr_en == trans_rcv.wr_en && trans.rd_en == trans_rcv.rd_en`: This compares the `wr_en` (write enable) and `rd_en` (read enable) fields from both the `trans` and `trans_rcv` transactions. If they match, `compare` is set to 1 (meaning the transactions are the same in terms of read and write operations).
- **Handling Full/Empty Memory States:**
  - If the receiver transaction indicates the memory is full or empty, it prints corresponding information:
  - `trans_rcv.full`: Indicates that the memory is full.
  - `trans_rcv.empty`: Indicates that the memory is empty.
- **Displaying Data:**
  - The `wr_en`, `wdata`, `rd_en`, and `rdata` fields of the transactions are displayed to show the current transaction and memory state.
  - If the comparison was successful (i.e., the transactions match), it prints "Yes". Otherwise, it prints "No".

**Assertions:**

- Encoded properties to check FIFO behavior, such as:
- Data is output in the same order it was input (FIFO ordering).
- Overflow/underflow conditions trigger appropriate error signals.
- Proper response to reset

**Design Under Test (DUT):**

- The DUT is a synchronous FIFO with configurable parameters (depth, width, etc.).
- Key functionalities include write and read operations, handling full/empty conditions, overflow/underflow management, and reset operations.

```
// Testbench Module
`include "test.sv"
`include "inf_fifo.sv"
module tb_top();
    bit clk;
    fifo_if inf(clk);    // Create FIFO interface
    test t1(inf);        // Instantiate testbench

    fifo dut (
        .wdata(inf.wdata),
        .rd_en(inf.rd_en),
        .wr_en(inf.wr_en),
        .full(inf.full),
        .empty(inf.empty),
        .rdata(inf.rdata),
        .clk(clk)
    );

    initial begin
        clk = 1;
    end

    always #5 clk = ~clk; // Clock generation

    initial begin
        $dumpfile("dump.vcd"); // Enable VCD dump for waveform viewing
        $dumpvars;
    end
endmodule
```

- Clock Generation: The clock (clk) is toggled every 5 time units to simulate a 10-time-unit period (100 MHz if time is in nanoseconds).
- FIFO Interface (fifo\_if): The fifo\_if interface connects the testbench signals to the DUT, allowing for easy signal management and communication.
- Test Class (test): This class will handle the stimulus generation, checking, and possibly other testbench logic using the FIFO interface.
- DUT (FIFO): The FIFO module is instantiated and connected to the interface signals.
- VCD Dump: The waveform data is dumped to a dump.vcd file for later inspection in a waveform viewer.

This testbench sets up the necessary environment to test a FIFO module and observe its behavior over time using signal waveforms.



### 3.2.3 Testbench Architecture :

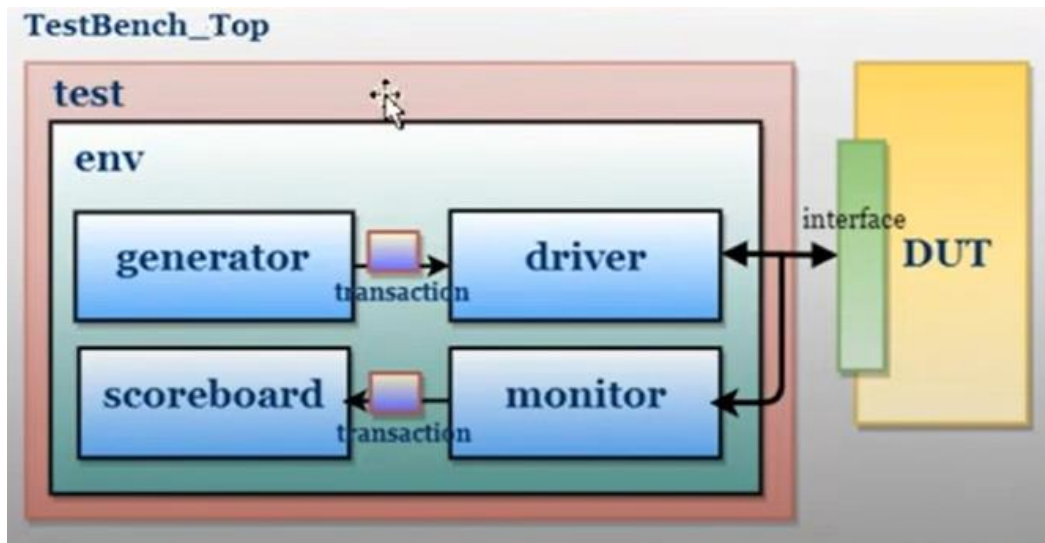


Figure 3.2-Test Bench Architecture

#### Testcases:

**Directed Tests:** Validate specific FIFO functionalities, such as:

- Write/read operations.
- Full and empty conditions.

**Random Tests:** Generate random sequences of writes and reads to test edge cases and uncover unexpected behaviors.

**Corner-Case Tests:** Focus on scenarios like simultaneous write and read operations or pointer wrap-around.

#### Functional Coverage:

- Coverage points are added to track test progress and ensure all possible scenarios are verified, including:
- FIFO depth utilization.
- Different states: full, empty, partial.
- Reset functionality.

#### Testbench Top:

- Instantiates all components (driver, monitor, scoreboard, DUT).
- Integrates and coordinates the flow of transactions.
- Verification Workflow

#### Plan:

- Identify key verification objectives.
- Develop a verification plan outlining scenarios, corner cases, and coverage goals.

**Develop:**

- Build the UVM-based testbench with reusable and parameterized components.

**Simulate:**

- Run the testbench in an EDA simulation environment.
- Use both directed and random tests.

**Debug:**

- Analyze waveform dumps and logs to identify and resolve mismatches or failures.

**Analyze Coverage:**

- Review functional and code coverage reports to ensure completeness.
- Refine and add tests for uncovered scenarios.

**Report:**

- Document results, highlighting any bugs found and fixed.
- Demonstrate that the FIFO meets all functional requirement.

## Interview questions

\* Write a System Verilog class with random constraints for a packet object. The packet should have fields addr, data, and len, with constraints ensuring addr is 8-bit, data is 16-bit, and len is between 1 and 10.

```
➤ class packet;
    rand bit [7:0] addr;
    rand bit [15:0] data;
    rand int len;
    constraint c_len { len inside {[1:10]}; }
endclass
```

\* Implement a UVM scoreboard that receives transactions from the monitor and compares them against expected values.

```
➤ class my_scoreboard extends uvm_scoreboard;
    uvm_component_utils (my_scoreboard)
    uvm_analysis_imp #(packet, my_scoreboard) analysis_export;
    function new(string name, uvm_component parent);
        super.new(name, parent);
        analysis_export = new("analysis_export", this);
    endfunction
    virtual function void write (packet trans);
        if (trans.data !== expected_data)
            uvm_error("SCOREBOARD", $sformatf ("Data mismatch: expected
            %0h, got %0h", expected_data, trans.data));
    endfunction
endclass
```

\* Write a clocking block in SystemVerilog for a testbench that has a 10ns clock period.

```
➤ clocking tb_clk @(posedge clk);
    default input #1ns output #1ns;
    output logic data_out;
    input logic data_in;
endclocking
```

\* Write a UVM sequence that accepts a parameter for the number of transactions it should generate and can adapt its behavior based on the parameter value.

```
➤ class param_sequence extends uvm_sequence #(packet); `uvm_object_utils (param_sequence)
    int num_trans;
    function new(string name = "param_sequence"); super.new(name);
    endfunction
    virtual task body;
    i++) begin
    for (int i = 0; i < num_trans; req packet::type_id::create("req");
    req.randomize();
    start_item (req);
    finish_item(req);
    end
    endtask
endclass
```

\* Write a parameterized UVM driver that can support both 8-bit and 16-bit data transactions.

```
➤ class param_driver #(int DATA_WIDTH = 8) extends uvm_driver # (packet);
    `uvm_component_param_utils (param_driver #(DATA_WIDTH))
    virtual task run_phase (uvm_phase phase);
    forever begin
    seq_item_port.get_next_item(req);
    // send data with parameterized width
    uvm_info("DRIVER", $sformatf ("Sending data: %0h",
    req.data), UVM_MEDIUM)
    seq_item_port.item_done();
    end
    endtask
endclass
```

\* Create a constrained random scenario where a packet's address is randomized between 0x10 and 0x20, and data is between 0x0 and 0xFF.

```
➤ class packet;
    rand bit [7:0] addr;
    rand bit [7:0] data;
    constraint addr_c { addr inside {[8'h10: 8'h20]}; }
    constraint data_c { data inside {[8'h00: 8'hFF]}; }
endclass
```

\* Write a class that uses the pre\_randomize and post\_randomize hooks to adjust the data field based on a calculated value.

```
➤ class packet;
    rand int data;
    int calc_value;
    function void pre_randomize();
    calc_value = some_function;
    endfunction
    function void post_randomize;
    if (data calc_value)
    data + calc_value;
    endfunction
endclass
```

\* Write a parameterized UVM sequence that can generate packets with varying data widths, constrained to certain ranges depending on the width (e.g., if width is 8 bits, data range is [0:255]; if width is 16 bits, data range is [0:65535]).

```
➤ class data_width_sequence #(int WIDTH = 8) extends uvm_sequence #(packet);
    uvm_object_param_utils (data_width_sequence #(WIDTH))
    virtual task body;
    req packet::type_id::create("req");
    assert(req.randomize with { data inside {[0: ((1 << WIDTH) - 1)]}; });
    start_item(req);
    finish_item(req);
    endtask
endclass
```

# ADVANTAGES AND APPLICATIONS

## CHAPTER: 4

### 4.1 Advantages of FIFO Verification Using System Verilog

The chosen methodologies and tools for FIFO verification offer several key advantages, enhancing both the effectiveness and efficiency of the process.

#### 4.1.1 Comprehensive Verification

##### **Constrained Random Testing:**

- Enables the generation of diverse test scenarios, including edge cases and unexpected input sequences, improving bug discovery.
- Helps uncover issues that directed tests might miss.

##### **Coverage-Driven Verification:**

- Functional coverage ensures all key features of the FIFO (e.g., full, empty, reset conditions) are tested.
- Code coverage helps identify untested areas in the design, ensuring no functionality is overlooked.

#### 4.1.2 Enhanced Debugging and Issue Detection

##### **Assertion-Based Verification:**

- Real-time checks on FIFO properties (e.g., ordering, overflow, and underflow) help detect errors early in the simulation.
- Provides precise information about the conditions under which a failure occurs.

##### **Scoreboard and Reference Model:**

- Validates output against expected results, quickly identifying mismatches and facilitating debugging.

#### 4.1.3 Modularity and Reusability

##### **UVM Framework:**

- The agent-based architecture (driver, monitor, sequencer, scoreboard) is modular and reusable.
- Components can be easily adapted for similar designs, saving time in future verification projects.

**Parameterized Testbench:**

- The testbench can handle different FIFO configurations (e.g., varying depths and widths) without significant changes.

**4.1.4 Scalability**

- The UVM-based environment is scalable to larger and more complex FIFO designs or systems incorporating multiple FIFOs.
- Easily integrates with additional verification components, such as coverage collection and clock-domain crossing (CDC) verification.

**4.1.5 Efficiency****Automation:**

- Automated test generation and reusable components reduce the manual effort required for test creation and maintenance.
- Faster simulation cycles through constrained random testing and automated assertion checks.

**Debugging Tools:**

- Integration with industry-standard tools (e.g., Synopsys VCS, Cadence Xcelium) provides advanced debugging features like waveform analysis and detailed logs.

**4.1.6 Verification Completeness****Functional Coverage:**

- Ensures verification closure by tracking scenarios tested, improving confidence in design correctness.

**Directed and Random Testing:**

- Combines the thoroughness of directed tests with the unpredictability of random tests to achieve maximum test coverage.

**4.1.7 Early Bug Detection**

- Early verification using SystemVerilog Assertions (SVAs) minimizes costly design changes in later stages of the development cycle.

**4.1.8 Industry Standard and Tool Compatibility**

- Using SystemVerilog and UVM aligns with industry standards, ensuring compatibility with popular EDA tools and fostering collaboration among verification teams.
- By leveraging the above advantages, this approach significantly enhances the robustness, reliability, and efficiency of FIFO verification, ensuring the design meets all functional requirements while minimizing development time and costs.

## 4.2 Applications of FIFO Verification Using System Verilog

FIFO (First-In-First-Out) memory is widely used in digital systems to manage data transfer and buffering. Ensuring its correctness through verification has direct implications for several application areas. Below are the primary applications of FIFO and the significance of its verification:

### 4.2.1 Data Buffering in Communication Systems

**Application:**

- Used in network routers, switches, and communication protocols to buffer data packets between devices operating at different speeds.
- Essential in handling data traffic in real-time systems like Ethernet, USB, or UART.

**Verification Impact:**

- Ensures no data loss during overflow or underflow conditions.
- Validates that data is transmitted in the correct order.

### 4.2.2 Clock Domain Crossing (CDC)

**Application:**

- Facilitates data transfer between components running on different clock domains, such as in asynchronous FIFOs.
- Common in multi-clock domain SoCs and FPGA designs.

**Verification Impact:**

- Ensures reliable synchronization and integrity of data across clock boundaries.
- Prevents metastability and timing violations

### 4.2.3 Audio and Video Processing

**Application:**

- Used to buffer streaming data in audio codecs, video decoders, and multimedia systems to manage varying input and output rates.
- Essential in devices like smart TVs, video players, and gaming consoles.

**Verification Impact:**

- Guarantees smooth playback without glitches due to buffer underruns or overruns.
- Ensures proper handling of data bursts.



#### 4.2.4 Embedded Systems

**Application:**

- Buffers sensor data in real-time embedded systems, such as IoT devices, automotive ECUs, and industrial controllers.
- Acts as a bridge between processors and peripherals like ADCs and DACs.

**Verification Impact:**

- Validates timely and ordered delivery of sensor data for accurate processing.
- Ensures reliability under extreme operating conditions (e.g., overflow/underflow).

#### 4.2.5 Processor and Memory Interfaces

**Application:**

- Acts as a queue between processors, caches, and memory units for tasks like instruction fetching and data transfer.
- Used in DMA controllers for bulk data movement.

**Verification Impact:**

- Ensures high-speed and error-free communication between memory and processor units.
- Prevents stalls or data corruption during burst transfers.

#### 4.2.6 Real-Time Systems

**Application:**

- Employed in applications requiring precise timing, such as robotics, avionics, and medical devices.

**Verification Impact:**

- Validates deterministic behavior under stringent timing constraints.
- Ensures no data is lost or delayed beyond acceptable limits.

#### 4.2.7 Consumer Electronics

**Application:**

- Used in devices like smartphones, tablets, and smart appliances to buffer user input, network data, and multimedia streams.

**Verification Impact:**

- Ensures seamless user experience with efficient data handling.
- Avoids performance degradation due to buffer mismanagement.

**4.2.8 Storage Systems****Application:**

- Buffers data in hard drives, SSDs, and cloud storage systems during read/write operations.

**Verification Impact:**

- Ensures correct data sequencing and integrity during storage and retrieval.
- Prevents loss or corruption of critical data.

**4.2.9 High-Performance Computing (HPC)****Application:**

- Manages data streams in parallel computing systems and GPUs to synchronize tasks and improve throughput.

**Verification Impact:**

- Ensures efficient utilization of computational resources by eliminating bottlenecks in data flow.

**4.2.10 Test and Measurement Systems****Application:**

- Buffers data in oscilloscopes, logic analyzers, and other measurement instruments to handle high-speed sampling.

**Verification Impact:**

- Guarantees accurate representation of real-world signals without losing critical data.
- Importance of Verification for Applications

# RESULTS

## CHAPTER: 5

# RESULTS

### 5.1 FIFO Verification Results

- Design Overview
- FIFO Type: Synchronous FIFO
- Width: 8 bits
- Depth: 32 entries
- Key Features:
  - Full and empty flag generation
  - Synchronous read/write operations controlled by wr\_en and rd\_en.
- Simulation Observations
  - Reset Functionality:
    - On asserting the rst signal:
      1. Write and read pointers (wr\_ptr, rd\_ptr) are correctly reset to 0.  
Memory is cleared (initialized to 0).  
empty flag is correctly asserted, indicating the FIFO is empty.  
rdata remains undefined (xx) until a valid read occurs.
      2. Write Operation:
 

When wr\_en is asserted:

Data is correctly written into the FIFO at the address indicated by wr\_ptr.

Write pointer (wr\_ptr) increments after each write cycle.

Full flag remains low during write operations until the FIFO is completely filled.

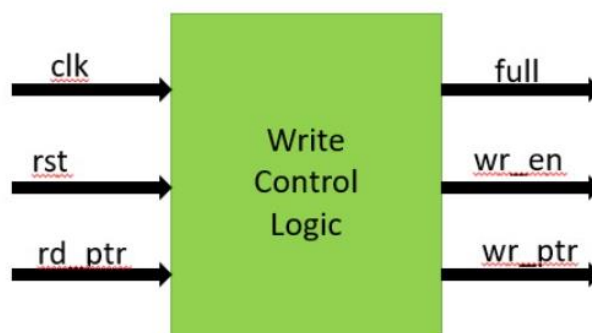


Figure 5.1-Block diagram of Write Control Logic

### 3. Read Operation:

When `rd_en` is asserted:

Data is correctly read from the FIFO at the address indicated by `rd_ptr`.

Read pointer (`rd_ptr`) increments after each read cycle.

`empty` flag remains low during read operations until the FIFO is completely emptied.

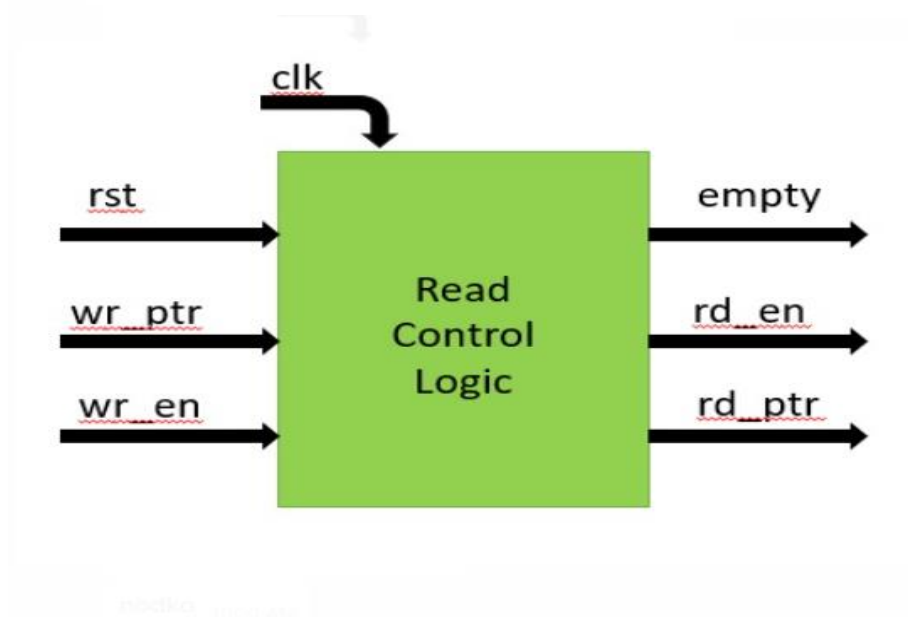


Figure 5.2-Block diagram of Read Control Logic

4.

### 5. Full and Empty Conditions:

Full Condition:

The full flag is asserted when the FIFO is completely filled (write pointer matches read pointer with `wr_ptr[5] != rd_ptr[5]`).

Additional write attempts are ignored, ensuring data integrity.

### 6. Empty Condition:

The empty flag is asserted when the FIFO is completely empty (`wr_ptr == rd_ptr`).

Additional read attempts produce no valid data, ensuring correctness.

### 7. Corner Cases:

Write operations are gracefully blocked when the FIFO is full.

Read operations are gracefully blocked when the FIFO is empty.

Proper wrap-around behavior is observed for the pointers (`wr_ptr` and `rd_ptr`).

## System Verilog Waveform:

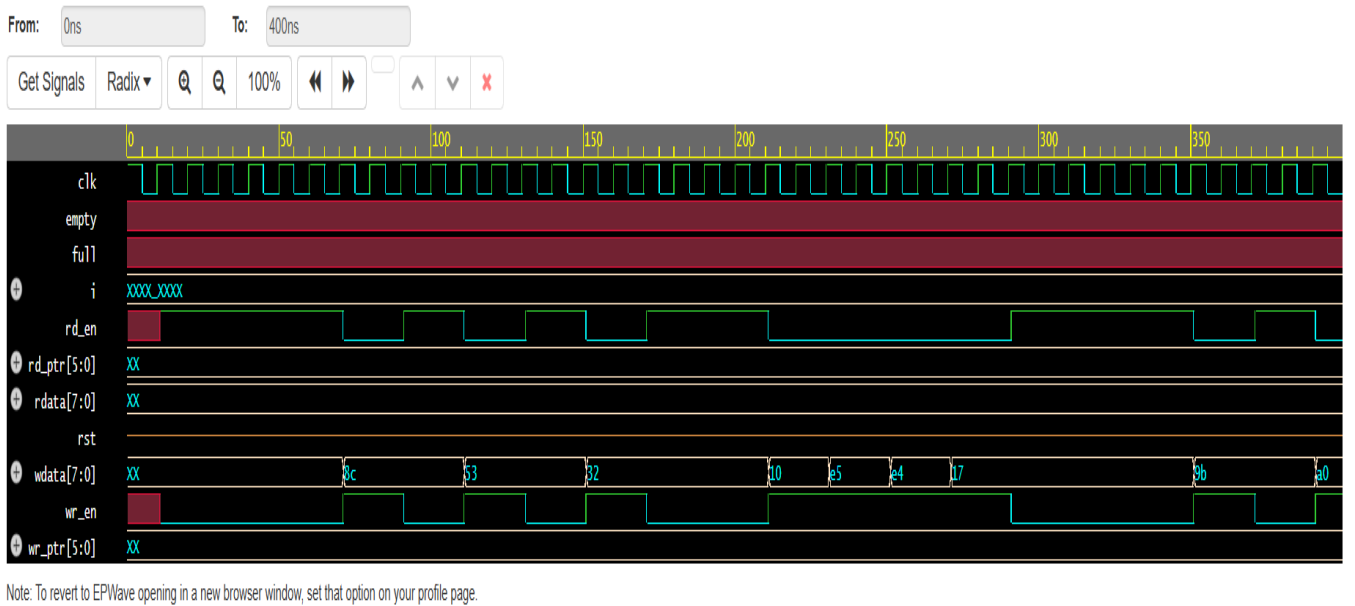


Figure 5.3-System Verilog waveform

This waveform represents the simulation of a FIFO (First-In-First-Out) buffer, showing the behavior of key signals over time. Below is a detailed explanation of the signals:

### Signals in the Waveform

#### 1. **clk (Clock):**

- This is the primary clock signal driving the entire design.
- All operations (write and read) occur on the rising edge of this signal.

#### 2. **empty:**

- Indicates if the FIFO is empty.
- High (1) when there is no data to read from the FIFO.
- Transitions to low (0) as soon as the FIFO receives data.

#### 3. **full:**

- Indicates if the FIFO is full.
- High (1) when the FIFO cannot accept any more data.
- Remains low (0) until the FIFO reaches its capacity.

#### 4. **rst (Reset):**

- Active at the start of the simulation (high), resetting all pointers (wr\_ptr and rd\_ptr) and clearing memory.
- Asserts that the FIFO starts in a known state.

#### 5. **wr\_en (Write Enable):**

- Controls whether data is written into the FIFO.
- When high (1), wdata is written into the memory at the wr\_ptr location, provided the FIFO is not full.

**6. rd\_en (Read Enable):**

- Controls whether data is read from the FIFO.
- When high (1), the data at the rd\_ptr location is output as rdata, provided the FIFO is not empty.

**7. wr\_ptr (Write Pointer):**

- Tracks the location in memory where the next write will occur.
- Increments on every write operation (wr\_en = 1).

**8. rd\_ptr (Read Pointer):**

- Tracks the location in memory where the next read will occur.
- Increments on every read operation (rd\_en = 1).

**9. wdata (Write Data):**

- Input data being written into the FIFO.
- Values such as 3c, 53, etc., indicate the hexadecimal values being written.

**10. rdata (Read Data):**

- Output data read from the FIFO.
- Matches previously written wdata values when rd\_en is asserted.

**Waveform Analysis****1. Reset Phase (0 to ~30ns):**

- rst is high, ensuring the FIFO starts in a cleared state.
- Both empty and full are high (empty = 1 and full = 0).
- wr\_ptr and rd\_ptr are set to 0.

**2. Write Operation (~50ns onwards):**

- wr\_en is asserted (high), and data (wdata) is written into the FIFO.
- wr\_ptr increments with each clock cycle, indicating the write pointer's movement.
- empty transitions to low as data is added to the FIFO.
- full remains low, as the FIFO is not at capacity.

**3. Read Operation (~200ns onwards):**

- rd\_en is asserted (high), and data is read out (rdata matches previously written wdata).
- rd\_ptr increments with each read, tracking the location being read.
- empty toggles back to high when all data is read out, indicating the FIFO is empty again.

**4. Mixed Operations (Write and Read Together):**

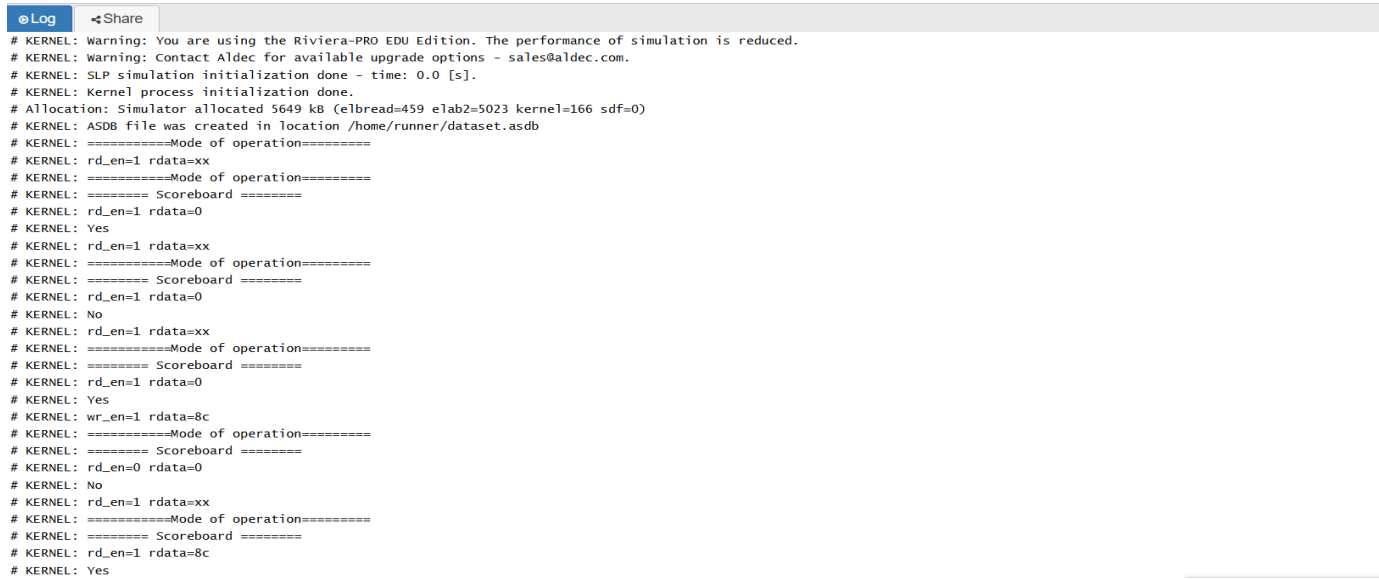
- In some cycles, both wr\_en and rd\_en are asserted, leading to simultaneous writes and reads.
- This ensures the FIFO handles dynamic operations correctly without overwriting unread data.

**5. End State:**

- wr\_ptr and rd\_ptr reflect their final positions.
- If the FIFO is emptied, empty returns high.

## Observations

- **FIFO Functionality Validation:**
  - Writes occur when `wr_en = 1` and the FIFO is not full.
  - Reads occur when `rd_en = 1` and the FIFO is not empty.
  - The FIFO correctly handles full and empty conditions without errors.
- **Signal Synchronization:**
  - All signals operate in synchronization with the `clk`.



```

Log Share
# KERNEL: Warning: You are using the Riviera-PRO EDU Edition. The performance of simulation is reduced.
# KERNEL: Warning: Contact Aldec for available upgrade options - sales@aldec.com.
# KERNEL: SLP simulation initialization done - time: 0.0 [s].
# KERNEL: Kernel process initialization done.
# Allocation: Simulator allocated 5649 kB (elbread=459 elab2=5023 kernel=166 sdf=0)
# KERNEL: ASDB file was created in location /home/runner/dataset.asdb
# KERNEL: =====Mode of operation=====
# KERNEL: rd_en=1 rdata=xx
# KERNEL: =====Mode of operation=====
# KERNEL: ===== Scoreboard =====
# KERNEL: rd_en=1 rdata=0
# KERNEL: Yes
# KERNEL: rd_en=1 rdata=xx
# KERNEL: =====Mode of operation=====
# KERNEL: ===== Scoreboard =====
# KERNEL: rd_en=1 rdata=0
# KERNEL: No
# KERNEL: rd_en=1 rdata=xx
# KERNEL: =====Mode of operation=====
# KERNEL: ===== Scoreboard =====
# KERNEL: rd_en=1 rdata=0
# KERNEL: Yes
# KERNEL: wr_en=1 rdata=8c
# KERNEL: =====Mode of operation=====
# KERNEL: ===== Scoreboard =====
# KERNEL: rd_en=0 rdata=0
# KERNEL: No
# KERNEL: rd_en=1 rdata=xx
# KERNEL: =====Mode of operation=====
# KERNEL: ===== Scoreboard =====
# KERNEL: rd_en=1 rdata=8c
# KERNEL: Yes

```

Figure 5.4-Output Log I

This log output is generated from the simulation of a FIFO design using **Aldec Riviera-PRO**.

### 1. Warnings and Initialization:

- **Warning:** `csharp`

We are using the Riviera-PRO EDU Edition. The performance of simulation is reduced.

This indicates that the simulation is running in the educational version of Riviera-PRO, which may have some performance limitations.

- **Simulation Initialization:** `less`

SLP simulation initialization done - time: 0.0 [s].

This confirms that the simulator has completed the setup process, and the simulation is ready to execute.

- **Simulator Allocation:** `yaml`

Simulator allocated 5649 kB.

Memory allocation for the simulation environment is reported.




## 2.Transaction Checks (Scoreboard Results):

- The scoreboard compares data generated during write operations and received during read operations to verify FIFO correctness.

**Mode of Operation:** diff

=====Mode of operation=====

This marks the start of a new set of read (rd\_en) and write (wr\_en) operations.



```

@Log  Share
# KERNEL: Warning: You are using the Riviera-PRO EDU Edition. The performance of simulation is reduced.
# KERNEL: Warning: Contact Aldec for available upgrade options - sales@aldec.com.
# KERNEL: SLP simulation initialization done - time: 0.0 [s].
# KERNEL: Kernel process initialization done.
# Allocation: Simulator allocated 5649 kB (elbread=459 elab2=5023 kernel=166 sdf=0)
# KERNEL: ASDB file was created in location /home/runner/dataset.asdb
# KERNEL: =====Mode of operation=====
# KERNEL: rd_en=1 rdata=xx
# KERNEL: =====Mode of operation=====
# KERNEL: ===== Scoreboard =====
# KERNEL: rd_en=1 rdata=0
# KERNEL: Yes
# KERNEL: rd_en=1 rdata=xx
# KERNEL: =====Mode of operation=====
# KERNEL: ===== Scoreboard =====
# KERNEL: rd_en=1 rdata=0
# KERNEL: No
# KERNEL: rd_en=1 rdata=xx
# KERNEL: =====Mode of operation=====
# KERNEL: ===== Scoreboard =====
# KERNEL: rd_en=1 rdata=0
# KERNEL: Yes
# KERNEL: wr_en=1 rdata=8c
# KERNEL: =====Mode of operation=====
# KERNEL: ===== Scoreboard =====
# KERNEL: rd_en=0 rdata=0
# KERNEL: No
# KERNEL: rd_en=1 rdata=xx
# KERNEL: =====Mode of operation=====
# KERNEL: ===== Scoreboard =====
# KERNEL: rd_en=1 rdata=8c
# KERNEL: Yes

```

Figure 5.5-Output Log II

### ➤ Scoreboard Section:

The rd\_en (read enable) and rdata (read data) fields suggest this is related to memory or register access operations.

### ➤ Values:

- rd\_en=0 indicates that read operations are not enabled.
- rd\_en=1 indicates read operations are enabled.
- rdata represents the data being read, with specific values (e.g., e4, 17, xx, etc.).

### ➤ Mode of Operation Section:

This might indicate a transition or check for specific modes in the system.

The word "Yes" or "No" seems to indicate conditions or outcomes of certain checks.

### ➤ Repeating Patterns:

The logs follow a repeated structure, suggesting this might be a test or debugging process cycling through multiple states or operations.

The xx in rdata=xx could signify invalid or unknown data during specific read operations.

# CONCLUSION

## CHAPTER: 6

### CONCLUSION

The verification of a FIFO design using System Verilog and Universal Verification Methodology (UVM) provides a structured and efficient approach to ensure design correctness and robustness. Through constrained random testing, functional coverage, and assertion-based verification, the project validates key FIFO functionalities, including data integrity, proper sequencing, boundary conditions, and reset behavior. The modular and reusable testbench architecture, built using UVM, simplifies the verification process and facilitates scalability for different FIFO configurations.

The results demonstrate that the verification environment is capable of uncovering design bugs and ensuring the FIFO meets its functional requirements under diverse scenarios, including edge cases. Functional coverage metrics confirm the thoroughness of the verification, and the integration of industry-standard tools ensures reliability and compliance with best practices.

This project not only verifies the FIFO design effectively but also establishes a reusable framework that can be adapted for verifying other hardware components, contributing to the development of robust and high-performance digital systems across various application domains

#### **Verification of FIFO designs ensures:**

**Data Integrity:** Critical in applications like medical devices and avionics where data correctness is non-negotiable.

**System Reliability:** Prevents catastrophic failures in safety-critical systems like automotive or aerospace applications.

**Performance Optimization:** Ensures maximum throughput and efficient resource utilization in HPC and communication systems.

**Compliance with Standards:** Guarantees compliance with industry protocols and specifications, essential in networking and consumer electronics.

By thoroughly verifying FIFO designs, their performance and reliability in real-world applications can be guaranteed, contributing to robust and efficient systems across indus

# REFERENCES

**CHAPTER: 7****REFERENCES**

1. Miro Panadas and A. Greiner. Bi-synchronous FIFO for synchronous circuit communication well suited for network-on-chip in GALS architectures. In First International Symposium on Networks-on-Chip (NOCS'07), pages 83–94, May 2007.  
doi:10.1109/NOCS.2007.14.
2. P. Coussy, A. Baganne, and E. Martin. A design methodology for integrating IP into SOC systems. Proceedings of the IEEE 2002 Custom Integrated Circuits Conference (Cat. No.02CH37285), pages 307–310, May 2002. doi:10.1109/CICC.2002.1012825.
3. Allen E. Sjogren and Chris J. Myers. Interfacing synchronous and asynchronous modules within a high-speed pipeline. IEEE Transactions on Very Large-Scale Integration (VLSI) Systems, 8(5):573–583, October 2000. doi:10.1109/92.894162.
4. <https://repository.rit.edu/cgi/viewcontent.cgi?article=11135&context=theses>
5. <https://youtu.be/YNJigOWfC-E?si=ZJqfLN4mSIXDBHTM>
6. [https://www.linkedin.com/posts/mohamed-hussein-274337231\\_fifo-sv-verification-activity-7248676719514013696-Wjr2/?utm\\_source=share&utm\\_medium=member\\_desktop](https://www.linkedin.com/posts/mohamed-hussein-274337231_fifo-sv-verification-activity-7248676719514013696-Wjr2/?utm_source=share&utm_medium=member_desktop)
7. M. A. Khan and A. Q. Ansari. n-Bit multiple read and write FIFO memory model for network-on-chip. In 2011 World Congress on Information and Communication Technologies, pages 1326–1331, December 2011. doi:10.1109/WICT.2011.6141440.
8. M. E. S. Elrabaa. A new FIFO design enabling fully-synchronous on-chip data communication network. 2011 Saudi International Electronics, Communications and Photonics Conference (SIECPC), pages 1–6, April 2011. doi:10.1109/SIECPC.2011.5877006.