# IMPROVING THE VERIFICATION EFFICIENCY WITH VIRTUAL SEQUENCES USING UNIVERSAL VERIFICATION METHODOLOGY

**A PROJECT REPORT**

***Submitted by***


| | |
|---|---|
| **BALAJI S** | **410120106014** |
| **GOMATHI P** | **410120106024** |
| **JEEVA V** | **410120106029** |
| **SANGEETHA R** | **410120106063** |


***in partial fulfillment for the award of the degree-***

***of***

**BACHELOR OF ENGINEERING**

*in*

**ELECTRONICS AND COMMUNICATION ENGINEERING**



**ADHI COLLEGE OF ENGINEERING AND TECHNOLOGY**

**SANKARAPURAM, KANCHIPURAM 631 605**

**ANNA UNIVERSITY :: CHENNAI 600 025**


**MAY 2024**

# ANNA UNIVERSITY :: CHENNAI 600 025

## BONAFIDE CERTIFICATE

Certified that this project report "**IMPROVING THE VERIFICATION EFFICIENCY WITH VIRTUAL SEQUENCES USING UNIVERSAL VERIFICATION METHODOLOGY**" is the bonafide work of "**BALAJI S (410120106014), GOMATHI P (410120106024), JEEVA V (410120106029), SANGEETHA R (410120106063)**" who carried out the project work under my supervision.

**SIGNATURE**

**Dr. K. DINESH BABU, M.E, Ph.D.,**
**HEAD OF THE DEPARTMENT**
ASSOCIATE PROFESSOR

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING,
ADHI COLLEGE OF ENGINEERING & TECHNOLOGY,
SANKARAPURAM,
KANCHIPURAM 631 605

**SIGNATURE**

**Mrs. N RUPADEVI M.E.**
**SUPERVISOR**
ASSISTANT PROFESSOR

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING,
ADHI COLLEGE OF ENGINEERING & TECHNOLOGY,
SANKARAPURAM,
KANCHIPURAM 631 605

Certified that the candidates were examined by us in the project work viva-voce examination held on _____

**INTERNAL EXAMINER** **EXTERNAL EXAMINER**

## ACKNOWLEDGEMENT

At this moment we would like to add a few heartfelt thanks for the people who made this project successful.

We would like to take this pleasant moment to thank our management for creating an environment to carry our project. Very special thanks to our Honorable Chairman **Dr. SARAN RAJ S** and CEO **Dr. SUJATHA MARAN,** for motivating us to carry out project successfully.

With deep sense of gratitude, we thank our beloved principal **Dr. DEVARAJU A** who has always been a constant source of inspiration and has rendering his kind cooperation to complete our project.

With deep sense of gratitude to **Dr. DINESH BABU K,** Head of the department for his words of wisdom and also for encouraging us in right way from the beginning and also, we thank **Mrs. RUPADEVI N,** internal guide for constant guidance and also thanks to **TEACHING AND NON-TEACHING STAFFS** of our department who rendered their valuable guidance in making this project a reality.

We also express our gratitude to our **PARENTS** and **FRIENDS** who had support us and helped us to keep the enthusiasm in completing the project successfully.

## ABSTRACT

The verification of complex digital designs is further compounded when dealing with specific components like FIFO (First-In-First-Out) memory modules. These modules are integral to data processing and communication systems. Within the Universal Verification Methodology (UVM) framework, virtual sequences play a vital role in addressing these challenges. By explores the master-slave concept using virtual sequences for FIFO module verification. virtual sequences for FIFO module verification, we enable the creation of intricate test scenarios specifically to emulate the read and write operations of FIFO memory modules automating the verification process to enhance productivity, accuracy, and reduce time-to-market. Thus, while UVM provides a robust framework for system-level verification, the application of virtual sequences to FIFO modules is crucial for ensuring the reliability and functionality of digital designs.

# TABLE OF CONTENTS

| CHAPTER NO. | TITLE | PAGE NO. |
|---|---|---|

# LIST OF FIGURES

# LIST OF ABBREVIATIONS

| SYMBOLS | ABBREVIATIONS |
|---|---|
| UVM | Universal Verification Methodology |
| FIFO | First In First Out |
| OVM | Open Verification Methodology |
| SPI | Serial Peripheral Interface |
| FPGA | Field Programmable Gate Array |
| SOC | System On-Chip |
| DUT | Design Under Test |
| TLM | Transaction Level Modeling |
| ABV | Assertion Based Verification |
| CRV | Constrained Random Verification |
| HDL | Hardware Description Languages |

# CHAPTER 1

# INTRODUCTION

## 1.1. OVERVIEW

This project focuses on the development and implementation of a comprehensive verification environment utilizing the Universal Verification Methodology (UVM) framework. The primary objective is to verify complex digital designs, with a specific emphasis on the operation of FIFO (First-In-First-Out) memory modules. To address the challenges associated with intricate data storage and retrieval mechanisms inherent in FIFO memory modules, we employ a master-slave concept. The master component is responsible for write operations, while the slave component handles read operations. Leveraging virtual sequences within the UVM framework, to automate the verification process, enhancing productivity, accuracy, and reducing time-to-market. The master component generates sophisticated test scenarios tailored to emulate write operations, whereas the slave component focuses on creating diverse read scenarios. Together, these components ensure a holistic verification approach. Key components of the verification environment include the construction of systematic test scenarios. These scenarios are meticulously designed to cover both read and write operations of FIFO memory modules, ensuring thorough verification coverage and reliability.

## 1.2. OBJECTIVE

The primary objective of this project is to develop and implement a robust verification environment utilizing the Universal Verification Methodology (UVM) framework to verify the operation of FIFO (First-In-First-Out) memory modules in complex digital designs. In this pursuit, we integrate a master-slave concept to manage the read and write operations of the FIFO memory modules effectively. The master component will be responsible

for generating and managing write operations, while the slave component will handle read operations, ensuring seamless data flow and operation within the FIFO module. Specifically, the project aims to construct a comprehensive verification environment leveraging the UVM framework to ensure thorough verification of FIFO memory modules and their integration within digital designs. Implementing virtual sequences within the UVM environment will automate test scenario generation, improving verification efficiency and augmenting coverage metrics tailored to verify the read and write operations of FIFO memory modules. The project will focus on verifying critical aspects of FIFO memory modules, including data storage, retrieval mechanisms, timing constraints, and functionality, to ensure reliability and correctness in digital designs. Leveraging virtual sequences to automate verification tasks related to FIFO memory modules, in conjunction with the master-slave concept, will enhance productivity, accuracy, and ultimately reduce time-to-market for semiconductor products integrating FIFO memory modules. Through practical implementation and case studies, the project will showcase the effectiveness of UVM-based verification methodologies, master-slave architecture, and virtual sequences in addressing the challenges of verifying FIFO memory modules in complex digital designs

## 1.3 AIM OF THE PROJECT

The primary aim of this project is to devise a robust verification methodology tailored for FIFO (First-In-First-Out) memory modules using virtual sequences within a master-slave architecture. Utilizing this master-slave concept, the project will manage and validate the read and write operations to ensure seamless data integrity within the FIFO module and to enhance verification efficiency, accuracy.

# CHAPTER 2

# LITERATURE SURVEY

## LITERATURE SURVEY 1

Apoorva H M and Dr. Kiran Bailey 2020," UVM based Design Verification of FIFO" International Journal of Engineering Research & Technology (IJERT).

## Description:

In this paper FIFO consists of dual port RAM. This dual port RAM allows simultaneous access of read and write port synchronous FIFO architecture. The read and write process of FIFO is performed on a same clock. For every positive edge of the clock the data is written in to the FIFO, when the write enable is high and the FIFO is not full. When the read enable is high the data is read out for every positive edge of the clock and FIFO is not empty and all the output signals is set to zero when reset is given. Verification process is important stage in SOC's and FPGA. OVM was used for design verification but, it was replaced by UVM. As it offers advantages such as a sequential library that collects multiple sequences, command line processor and many more that make it a better methodology for design verification. .As the technology is leading towards nano new methodology's are coming up in field of verification. Universal Verification Methodology (UVM) is one of the methodology with advantages robust, scaling and reusable. In this work Synchronous FIFO is designed using Verilog and verified using UVM and simulation is carried out in Questa Sim tool.

## LITERATURE SURVEY 2

Bidisha Kashyap and V Ravi 2020,"Universal Verification Methodology Based Verification of UART Protocol" Journal of Physics: Conference Series.

## Description:

Verification today acts as a constriction of any complex VLSI design. Bringing out improved verification efficiency is a must. Most of the computers and microcontrollers contain a number of serial data ports. These data ports are used to connect with devices such as keyboards and printers which are basically serial input and output devices. Transmission and reception of serial data from an isolated location can be done with the help of a modem connected to the serial port. UART- Universal Asynchronous Receiver and transmitter is a hardware device which facilitates serial transmission and reception of data. In this work presented here, the UART has been designed with the use of the industry standard Verilog HDL code and the verification of the protocol has been done using system Verilog code in UVM environment. The UVM based verification methodology can significantly reduce the time needed for verification. Most UART is a computer hardware device that is used for serial communication. The device is used for data exchange between a computer and outer devices as it provides high reliability and capability of data transmitting to a long distance. It is used to control the process of converting parallel data into serial data. It consists of one transmitter and one receiver.

## LITERATURE SURVEY 3

Hyuk Je Kwon, Myeong-Hoon Oh, Won-ok Kwon 2021," Verification of Interconnect RTL Code for Memory-Centric Computing using UVM" International Conference on Electronic, Information and Communication (ICEIC).

## Description:

This document is about the verification of an interconnect (i.e. switch) RTL code that is based on Gen-Z protocol using Universal Verification Methodology (UVM). Ports in the switch for transmission packets are connected to virtual interfaces with UVM. The packets that are generated in the UVM environment are transmitted into the ports of the switch through the virtual interfaces. For verifying the switch logic, we designed sequence items and a virtual sequencer and simulated it. The many EDA companies have made tools for verification and reusable, and the tools have been used it now. The tools can make to generate some stimulus codes by several users simultaneously. Recently, tools are unified into one. And is to verify the interconnect (i.e. switch) RTL code (based Gen-Z 1.0 protocol) for memory-centric computing, using UVM. The switch has several ports that transmit or receive packets to or from other memory-centric components. To put the stimulus into switch ports, we designed sequence items of UVM and verified switch codes. In this paper, we show the designed sequence item and virtual sequencer. And we show the structure that is for verifying switch RTL.

## LITERATURE SURVEY 4

Deepika, Jayanthi K Murthy 2020," Interrupt Enabled Priority Based Master Slave Communication using SPI Protocol" International Journal of Innovative Technology and Exploring Engineering (IJITEE).

## Description:

It is a master – slave type protocol that provides a simple and low-cost interface between a microcontroller and its peripherals. This paper proposes the design of a priority-based master slave communication system using SPI Protocol that enables the system to operate using interrupts. The design mainly emphasizes on priority-based communication where the slaves will generate an interrupt over a newly defined interrupt pin when some data transfer needs to happen. When the master receives an interrupt from the slave it establishes communication with one slave at a time based on the priority and the priority to each slave is assigned by the arbiter or priority control block. In SPI protocol, usually the master always initiates the communication. The master generates clock signal which is of a frequency the slave devices support. The slave device with which the communication must be started is selected by pulling the slave select pin to low state. The slave devices which are not selected by the slave select signal will reject the clock signal. Devices operating using SPI can be in single master, single slave configuration which is the simplest form or can have single master multiple slave configuration. In the second type, they can be connected using daisy chain configuration or parallel configuration (independent slave configuration) in which each slave will have independent slave select line. To design a priority-based master slave communication and in order to generate interrupts from the slaves a new pin called interrupt pin can be defined.

## LITERATURE SURVEY 5

Abhishek Jain, Richa Gupta 2017," Expanding the UVM Register Model towards Automation and Simplicity of Use" International Journal of Advanced Research in Computer Science.

## Description:

The standard UVM register package contains built-in test sequences library which is used to perform most of the basic register and memory tests. These sequences are very useful at IP level verification but at SoC level verification, these sequences take very long time to run. Similarly, currently users require strong knowledge of System Verilog UVM language to use UVM_REG model. Some limitations in current UVM_REG package like no automatic data checking for memory accesses and limited support for memory burst operation were also seen. In this paper, we are describing how we addressed the above mentioned issues. We are accessing processor programmable registers and memories through a standard UVM_REG API. This API is aimed at writing simpler directed tests which require less or no System Verilog/UVM understanding. This API can be used to facilitate dumping register access for reuse from IP to SoC, or format outputs for use in ATE test vectors development etc. We also developed our own register/memory sequences to address the SoC level register and memory testing. Customized code is written to enhance the features of standard UVM_REG model. IP-XACT based tools are also developed to automatically generate all required verification environment files for using standard register model

# CHAPTER 3

# METHODOLOGY

## 3.1 EXISTING SYSTEM

- The existing system utilizes the UVM framework for FIFO memory module verification.
- Verification is conducted using regular sequences, which are predefined sequences of events or transactions.
- While regular sequences can support constrained random verification, they are limited in their flexibility and dynamic generation capabilities.
- Regular sequences offer limited flexibility in modeling higher-level scenarios based on constraints and specifications.
- Engineers manually define regular sequences, which can be time-consuming and less adaptable to changes in the design.
- A single sequencer manages both write and read operations, potentially leading to synchronization issues and reduced efficiency.

### 3.1.1 DRAWBACKS

- The existing system lacks the flexibility to dynamically generate scenarios based on constraints and specifications.
- Manual sequence definition may result in increased effort and potential errors.
- Limited flexibility and manual effort may pose challenges in scaling the verification environment to accommodate changes in the design.
- The rigidity of regular sequences may lead to gaps in verification coverage for certain scenarios.
- Single sequencer handling both write and read operations may lead to inefficiencies and synchronization issues.

## 3.2 PROPOSED SYSTEM

- The proposed system integrates virtual sequences and a master-slave concept into the UVM verification environment for FIFO memory module verification.
- Virtual sequences offer dynamic generation capabilities and flexibility in modeling higher-level scenarios based on constraints and specifications.
- Virtual sequences support constrained random verification, allowing for more comprehensive scenario coverage.
- A master sequencer manages write operations, while a slave sequencer handles read operations, ensuring synchronized and efficient operation
- Virtual sequences and the master-slave architecture automate scenario generation, reducing manual effort and increasing efficiency
- The proposed system provides enhanced flexibility in adapting to changes or updates in the design.
- Virtual sequences enable comprehensive scenario coverage, addressing potential verification gaps present in the existing system.

## 3.2.1 PROPOSED ARCHITECTURE

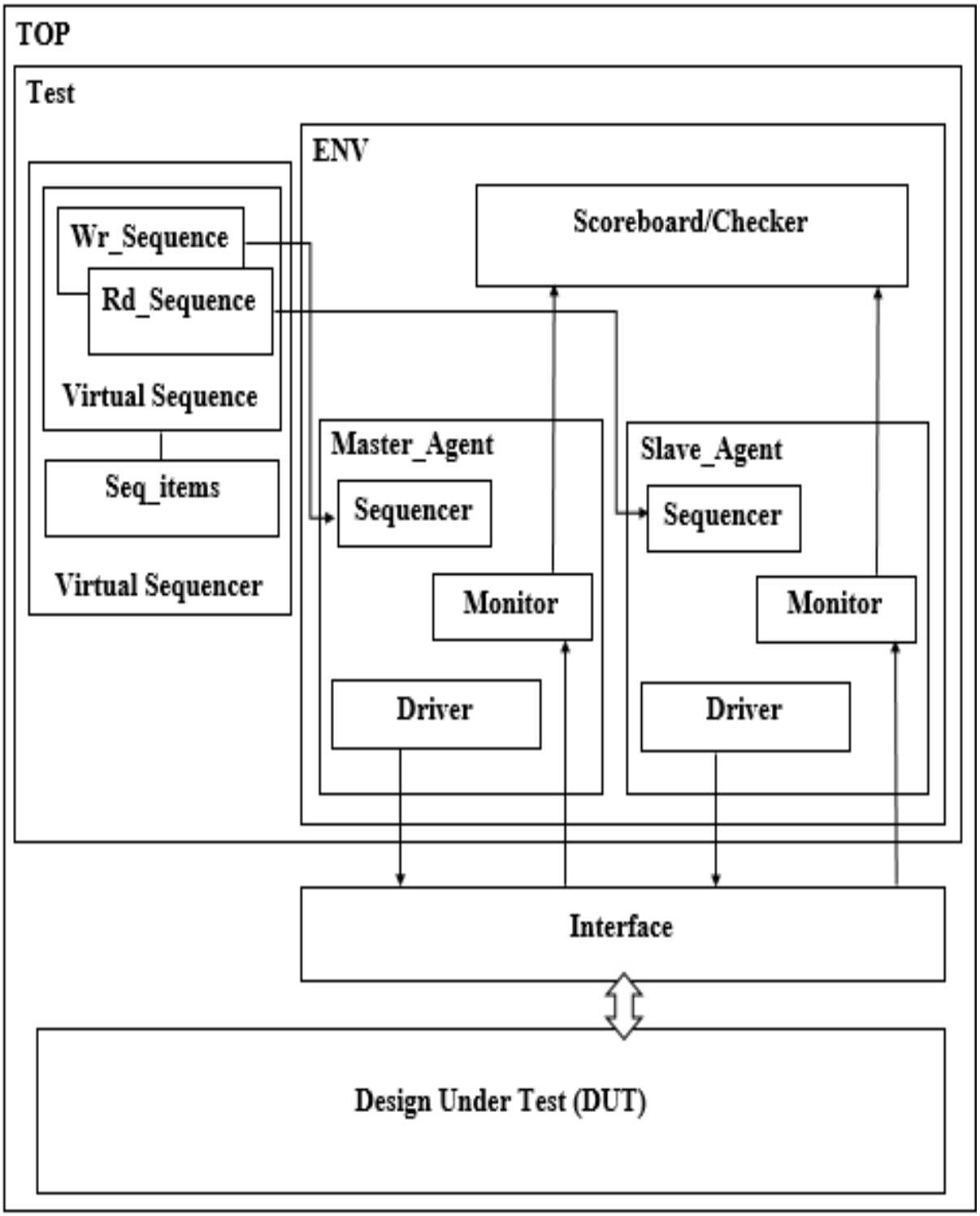


*Figure 3.2.1 Architecture Diagram*

## 3.2.2 ADVANTAGES

**Enhanced Flexibility:** The proposed system offers enhanced flexibility in scenario generation based on constraints and specifications. High data transmission rates of up to 10Gbps can be achieved.

**Automation:** Automated scenario generation reduces manual effort and increases verification efficiency.

**Scalability:** The proposed system is scalable and adaptable to changes in the design, improving overall flexibility.

**Comprehensive Verification:** Virtual sequences enable comprehensive scenario coverage, reducing the risk of verification gaps

**Improved Coverage:** The dynamic nature of virtual sequences ensures coverage of a wide range of scenarios, enhancing overall verification quality.

## 3.2.3 APPLICATIONS

The proposed system with virtual sequences is applicable to various digital design verification.

- Verification of FIFO memory modules
- System-on-chip (SoC) designs verification
- FPGA designs verification
- Digital Communication System.

# CHAPTER 4

# PROTOCOL DESCRIPTION

## 4.1 Synchronous FIFO Design

A synchronous FIFO (First-In-First-Out) is a fundamental building block in digital design, offering efficient data storage and retrieval mechanisms synchronized to a common clock signal. At its core, a synchronous FIFO comprises input and output registers, read and write pointers, control logic, and data storage elements. Unlike asynchronous FIFOs, which operate without a clock and can introduce timing uncertainties, synchronous FIFOs synchronize read and write operations to the rising edge of a clock signal, ensuring precise data transfer and coordination. When data is written into the FIFO, it is captured by the input register and transferred to the storage elements. Similarly, when data is read from the FIFO, it is retrieved from the storage elements and presented at the output register. The read and write pointers track the current locations within the FIFO, allowing for efficient data movement and management. Synchronous FIFOs find wide application across various digital systems, including communication interfaces, memory controllers, and data processing units, where they play a crucial role in buffering, synchronizing, and managing data flow.

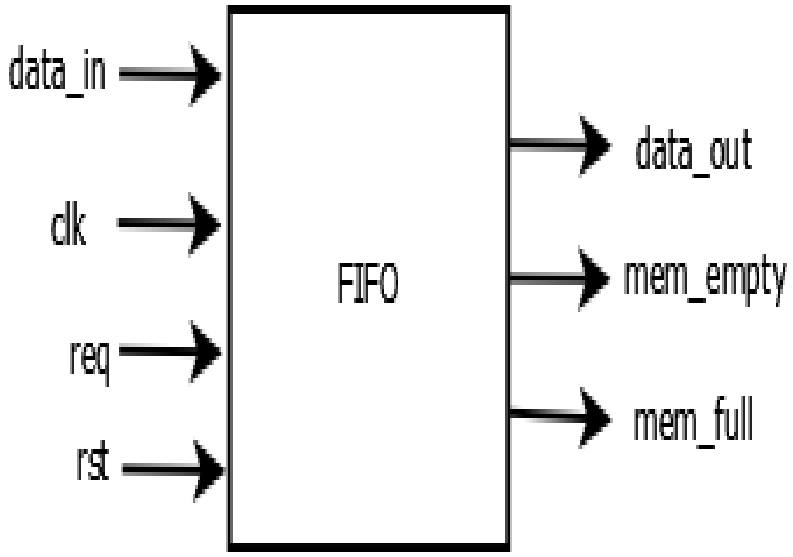


***Figure 4.1 Block Diagram of FIFO***

### 4.1.1 FIFO Operation

In your project, implementing the functionality of a FIFO (First-In-First-Out) memory module involves several key operations and considerations to ensure proper data management and integrity. Here's a description of the FIFO operation and its functionalities:

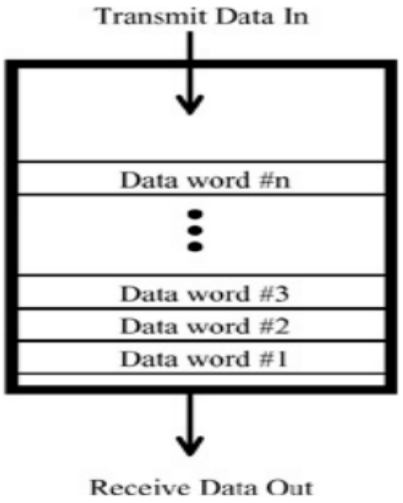


***Figure 4.1.1 FIFO Functional Diagram***

#### 4.1.1.1 Write Operation

- The write operation involves adding new data items into the FIFO for storage and subsequent retrieval.
- When performing a write operation, data is enqueued into the FIFO at the current write pointer location.
- If the FIFO is not full, the write pointer is advanced to the next available location, and the data is stored.
- However, if the FIFO is full, attempting to write new data may result in an overflow condition, which needs to be handled appropriately.

#### 4.1.1.2 Read Operation

- The read operation entails retrieving data items from the FIFO in the order they were enqueued.
- During a read operation, data is dequeued from the FIFO at the current read pointer location.
- If the FIFO is not empty, the read pointer is advanced to the next available location, and the data is returned for processing.
- However, if the FIFO is empty, attempting to read data may result in an underflow condition, which also requires appropriate handling.

#### 4.1.1.3 IsFull Functionality

- The isFull functionality checks whether the FIFO has reached its maximum capacity.
- It examines the current number of stored data items against the FIFO's depth to determine if there is available space for additional writes.
- If the FIFO is full, attempting to write new data may lead to overflow, indicating that no more data can be added until space becomes available.

#### 4.1.1.4 IsEmpty Functionality

- The isEmpty functionality determines whether the FIFO contains any data items for retrieval.
- It checks if the read and write pointers are at the same location, indicating that the FIFO is empty.
- If the FIFO is empty, attempting to read data may result in underflow, indicating that no data is available for retrieval.

#### 4.1.1.5 Simultaneous Read and Write

- In a synchronous FIFO, where read and write operations are synchronized to a common clock signal, simultaneous read and write operations may occur during adjacent clock cycles.
- When simultaneous read and write operations happen, it's essential to ensure proper synchronization and data integrity to avoid potential data corruption or race conditions.
- Proper arbitration mechanisms or control logic should be implemented to manage simultaneous read and write requests and ensure that data is accessed correctly without interference.

### 4.1.1.6 Overlapping Read and Write Pointers

- Another corner case arises when the read and write pointers overlap or reach the same location within the FIFO.
- This situation can occur when the FIFO is nearly full or nearly empty, and the read and write operations converge at the same location.
- Handling overlapping pointers requires careful consideration to prevent data loss or corruption. Proper pointer management and boundary checks are necessary to ensure that data is read and written correctly without overwriting existing data or accessing invalid memory locations.

### 4.1.1.7 FIFO Full and Empty Conditions

- Corner cases related to FIFO full and empty conditions also need to be addressed.
- When the FIFO is nearly full, attempting to write additional data may lead to overflow conditions if not handled properly.
- Similarly, when the FIFO is nearly empty, attempting to read data may result in underflow conditions if appropriate error handling mechanisms are not in place.
- Properly managing these corner cases involves implementing robust error detection and handling mechanisms to prevent data loss, corruption, or system instability.

## 4.2 Master-Slave Concept in Digital Design Verification

The Master-Slave concept is a fundamental architecture widely used in various applications, including digital design verification. In this architecture, the Master and Slave components work collaboratively to manage and execute tasks efficiently.

### 4.2.1 Master

The Master component serves as the primary controller or initiator in the Master-Slave architecture. It takes on the responsibility of initiating tasks or operations, directing the Slave components to execute specific sequences or transactions. Additionally, the Master manages the overall flow of operations, ensuring synchronization and coordination among multiple Slave components. It also handles data transfer, storage, and retrieval, directing the Slave components to perform read and write operations as required. Furthermore, the Master monitors the execution of tasks and manages error handling, ensuring that any issues or discrepancies are addressed promptly.

### 4.2.2 Slave

The Slave component acts as the executor or subordinate in the Master-Slave architecture. It executes the tasks or operations initiated by the Master, performing the required sequences or transactions. The Slave component processes data as directed by the Master, performing read and write operations, data manipulation, or other specified tasks. It reports the results of task execution back to the Master, providing status updates, data verification, or error reports as required. Additionally, the Slave component maintains synchronization with the Master, ensuring timely and accurate execution of tasks in coordination with other Slave components.

## 4.3 The UVM Class Library

The UVM Class Library provides the building blocks needed to develop well-constructed and reusable VCs and test environments. The library consists of base classes and infrastructure facilities. Base classes in the UVM hierarchy largely fall into two distinct categories: components and data. The component class hierarchy derives from uvm component and is intended to model permanent structural parts of the test-bench such as monitors and drivers. Data classes are derived from uvm sequence item and are intended to model stimulus and transactions. The infrastructure facilities provide various utilities to simplify the development, management and use of verification environments, such as phasing and execution control, configuration methods, factory convenience methods, interconnection of components facilities and hierarchical reporting control

- Compile uvm_pkg.sv file.
- Import uvm_pkg into the desired scope.
- Include file that contains uvm macros.

### 4.3.1 UVM Component Class

All the infrastructure components in a UVM environment derive from the uvm component class and build a hierarchy which includes: sequencers, drivers, monitors, coverage collectors, scoreboards, environments and tests. In the following subsections the components that conform a UVM

#### 4.3.1.1 Test

- A test in UVM represents a specific scenario or testcase within the verification environment.
- It defines the stimulus, constraints, and expected behavior to verify a particular aspect of the design.
- Tests are executed by the testbench to verify the functionality of the Device Under Test (DUT).

#### 4.3.1.2 Environment

- The environment represents the top-level structure of the verification environment.
- It contains multiple components such as testbenches, agents, and other verification IP.
- The environment coordinates the activities of its constituent components and manages overall testbench functionality.

#### 4.3.1.3 Agent

- An agent represents a functional block within the testbench responsible for interfacing with the DUT.
- It typically includes components such as drivers, monitors, sequencers, and virtual sequences.
- Agents abstract the communication protocol and provide a standardized interface for interacting with the DUT.

## Master Agent

The Master Agent serves as the primary controller and coordinator, primarily responsible for initiating and managing write operations in the verification environment. It orchestrates the generation and sending of write transactions to the Design Under Test (DUT) based on predefined or dynamically generated sequences. The components present in the Master Agent include:

i. Master_Sequencer
ii. Master_Driver
iii. Master_Monitor

## Slave Agent

The Slave Agent is responsible for executing read operations within the verification environment. It interacts with the DUT to perform read operations, retrieve data, and validate the correctness of the DUT's responses. The components present in the Slave Agent include:

i. Slave_Sequencer
ii. Slave_Driver
iii. Slave_Monitor

### 4.3.1.4 Scoreboard

- A scoreboard compares the expected behavior of the DUT with its actual behavior.
- It analyzes transaction-level data from monitors and verifies that the DUT produces the correct outputs in response to the provided inputs.
- Scoreboards detect errors, discrepancies, or violations of protocol specifications, generating alerts or reports as needed.

### 4.3.1.5 Driver

- A driver is responsible for driving stimulus into the DUT's interface.
- It converts transactions generated by sequencers into signals or transactions compatible with the DUT's interface protocol.
- Drivers handle timing, sequencing, and synchronization of stimulus delivery to the DUT.

## Master Driver

- Generates write transactions based on the test scenarios defined in the virtual sequences.
- Drives the write transactions into the DUT (Device Under Test) through the interface.
- Ensures proper timing and synchronization of write operations with the DUT's clock and other interfaces.

## Slave Driver

- Extracts the read data from the DUT through the interface and prepares it for further verification.
- Ensures proper timing and synchronization of read operations with the DUT's clock and other interfaces.
- Compares the read data with the expected data to verify the correctness and data integrity of the read operations.

### 4.3.1.6 Monitor

- A monitor observes the activity on the interface between the DUT and the testbench.
- It captures input and output transactions, extracts relevant information, and forwards it to other components for analysis.
- Monitors play a crucial role in protocol checking, coverage collection, and debugging during verification

## Master Monitor

- Monitors write transactions sent by the Master Driver to ensure they are correctly initiated and synchronized with the DUT (Device Under Test).
- Monitors the interface signals and handshakes between the Master Driver and the DUT to ensure proper communication and synchronization.
- Provides real-time status updates on the write operations, including successful writes, errors, and any other relevant information.

## Slave Monitor

- Monitors read transactions from the Slave Driver to ensure they are correctly initiated and synchronized with the DUT.
- Validates the read data captured by the Slave Driver to ensure it matches the expected data and adheres to the protocol.
- Monitors the interface signals and handshakes between the Slave Driver and the DUT to ensure proper communication and synchronization.

#### 4.3.1.7 Virtual Interface

- A virtual interface provides a mechanism for connecting UVM components to external interfaces or models.
- It abstracts the interface details and allows components to interact with external models or simulators in a standardized manner.
- Virtual interfaces enable seamless integration of UVM test-benches with external models or simulators, facilitating mixed-language and mixed-platform verification.

### 4.3.2 UVM Transaction Level Modeling

- UVM transactions are commonly used in transaction-level modeling (TLM) to model communication between different components in a verification environment.
- TLM allows for abstraction of communication details and facilitates modeling of system-level interactions.

#### 4.3.2.1 Sequences

- In TLM, sequences define high-level scenarios or sequences of transactions that are to be executed by the testbench.
- Sequences encapsulate stimulus generation, constraints application, and transaction sequencing logic.
- They provide a convenient way to model complex interactions between components at a higher level of abstraction.

### 4.3.2.2 Sequencers

- Sequencers are responsible for coordinating the execution of sequences and managing transaction flow between the testbench and the Device Under Test (DUT).
- They receive requests from sequences, generate transactions based on these requests, and deliver them to the appropriate interface or agent for execution.
- Sequencers ensure proper synchronization and sequencing of transactions, facilitating efficient communication between components.

### Master Sequencers

- Controls the initiation and sequencing of write transactions to be sent by the Master Driver to the DUT (Device Under Test).
- Manages the flow and ordering of write sequences based on predefined sequences or dynamic sequence generation to ensure proper protocol adherence.
- Synchronizes with the Master Monitor to ensure proper transaction initiation, data validation, and error detection during write operations.

### Slave Sequencers

- Controls the initiation and sequencing of read transactions to be executed by the Slave Driver to fetch data from the DUT.
- Manages the flow and ordering of read sequences based on predefined sequences or dynamic sequence generation to ensure proper protocol adherence.
- Synchronizes with the Slave Monitor to ensure proper transaction initiation, data validation, and error detection during read operations.

### 4.3.2.3 Virtual Sequences

- Virtual sequences are sequences that are not tied to a specific interface or agent but instead represent higher-level scenarios or use cases in the verification environment.
- They provide a flexible and scalable approach to modeling complex test scenarios that may involve multiple interfaces or agents.
- Virtual sequences orchestrate the execution of sequences across different components, enabling comprehensive verification of system-level behavior.

### 4.3.2.4 Virtual Sequencer

- Virtual sequencers coordinate the execution of virtual sequences and manage the flow of transactions between different parts of the testbench.
- They act as intermediaries between virtual sequences and sequencers associated with specific interfaces or agents.
- Virtual sequencers enable the modularization and reuse of testbench components, promoting a hierarchical and scalable verification environment.

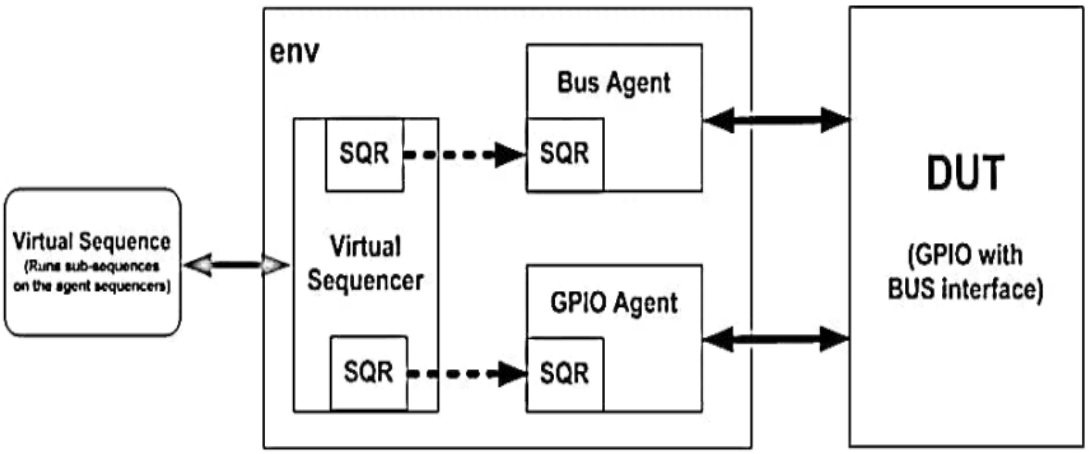


***Figure 4.3.2.4 Virtual Sequencer***

### 4.3.3 UVM Phases

In the Universal Verification Methodology (UVM), verification activities are organized into different phases to ensure systematic and coordinated execution of tasks across the testbench components.

#### 4.3.3.1 Build Phase

- During the build phase, testbench components are created and initialized.
- Components allocate resources, set up configuration parameters, and establish hierarchical relationships.
- This phase is primarily focused on constructing the testbench hierarchy and preparing for simulation.

#### 4.3.3.2 Connect Phase

- In the connect phase, inter-component connections are established.
- Components connect to their respective interfaces, agents, and virtual interfaces.
- Communication paths between components are set up to enable data exchange during simulation.

#### 4.3.3.3 End of Elaboration Phase

- The end of elaboration phase marks the completion of design elaboration and testbench setup.
- All design units and testbench components are elaborated, and their interconnections are finalized.
- The design hierarchy and testbench configuration are fully established.

#### 4.3.3.4 Start of Simulation Phase

- The start of simulation phase initializes simulation-specific settings and prepares the testbench for execution.
- Testbenches may perform additional setup tasks or configuration adjustments before simulation begins.
- This phase serves as a transition from testbench setup to actual simulation execution.

#### 4.3.3.5 Run Phase

- During the run phase, the main simulation activities take place.
- Test sequences are executed, stimulus is applied to the DUT, and responses are monitored and checked.
- Verification activities, such as coverage collection and error detection, occur throughout the run phase.

#### 4.3.3.6 Extract Phase

- The extract phase captures relevant information and data generated during simulation.
- It may involve extracting coverage data, transaction logs, or other relevant metrics for analysis and reporting.

#### 4.3.3.7 Check Phase

- The check phase verifies the correctness of simulation results and performs assertions or checks on generated data.
- It ensures that the DUT behaves according to expected specifications and protocol requirements.

### 4.3.3.8 Report Phase

- In the report phase, simulation results, coverage reports, and other metrics are collected and summarized.
- Reports may include pass/fail status, coverage metrics, performance analysis, and debugging information

### 4.3.3.9 Finalize Phase

- The finalize phase concludes simulation activities and performs cleanup tasks.
- Resources are deallocated, simulation logs are closed, and final reports are generated.
- Testbenches prepare for termination and exit simulation gracefully.

### 4.3.3.10 Execution of UVM Phases

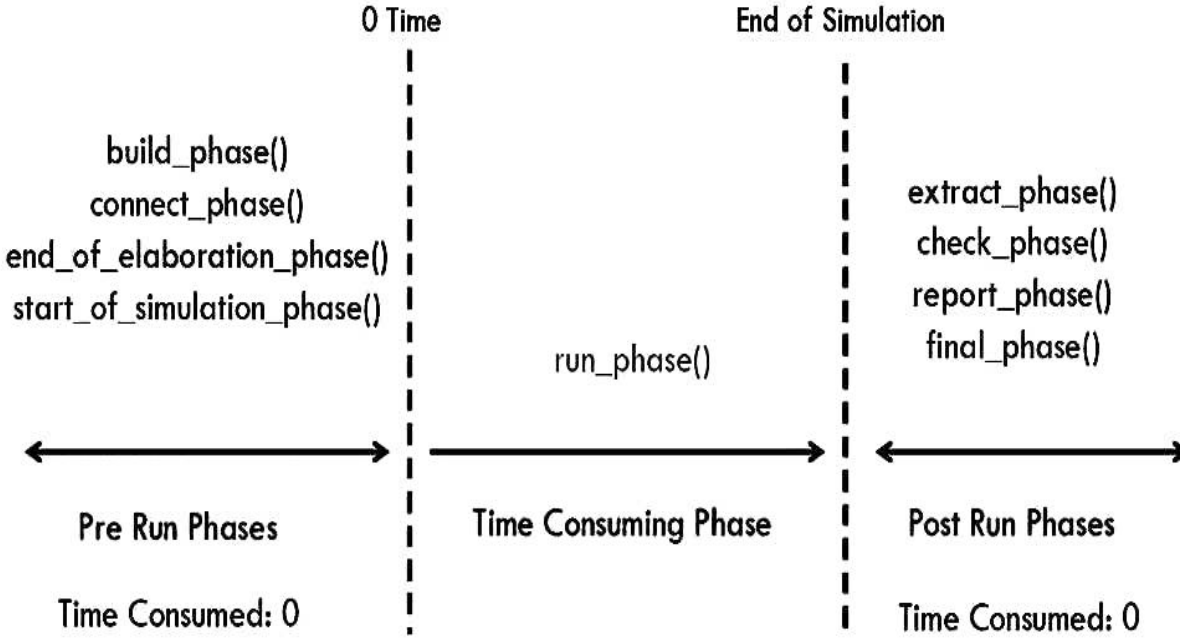


***Figure 4.2.3.10 Execution of UVM phases***

### 4.3.4 Simplified UVM Inheritance

In UVM, inheritance simplifies the creation and organization of verification components by allowing custom components to inherit properties and methods from base classes. At the core of UVM's inheritance hierarchy are base classes like '**uvm_component'**, '**uvm_sequence_item'**, '**uvm_sequence'**, '**uvm_driver'**, '**uvm_monitor'**, and '**uvm_sequencer'**. For instance, a custom '**master_driver'** class can inherit from the '**uvm_driver'**, enabling it to leverage and extend the driver's standard functionalities specific to the master agent. This inheritance approach streamlines the development process, promotes code reusability, and ensures consistency across different verification components within the UVM environment.

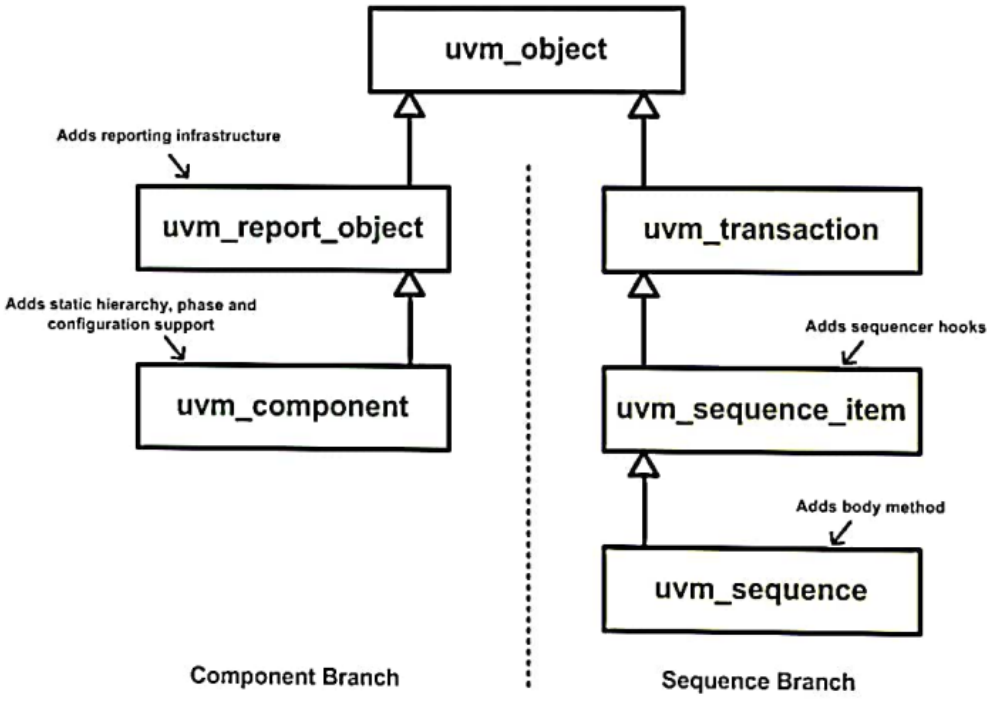


***Figure 4.3.4 Simplified UVM Inheritance diagram***

### 4.3.5 Approaches to Implementing Virtual Sequence

#### 4.3.5.1 Approach 1: Virtual Sequences with Agent's Sequencers

The Virtual Sequence directly contains the handles of the Agent's Sequencers, which manage the execution of the Sub-Sequences. This method allows the Virtual Sequence to control and orchestrate the Sub-Sequences by directly interacting with the Sequencers of the respective Agents (Master and Slave).

#### 4.3.5.2 Approach 2: Virtual Sequence on Virtual Sequencer

The Virtual Sequence runs on a Virtual Sequencer of type **‘uvm_sequencer’**. The Virtual Sequencer acts as an intermediary between the Virtual Sequence and the Agent's Sequencers. It manages the execution of the Virtual Sequence and delegates the execution of Sub-Sequences to the respective Agent's Sequencers.

#### 4.3.5.3 Selected Approach for the Project

- The Virtual Sequence runs on a Virtual Sequencer of type ‘**uvm_sequencer**’.
- The Virtual Sequence defines high-level scenarios and orchestrates the execution of sub-sequences.
- The Virtual Sequencer acts as a container or manager for the Virtual Sequence.
- The Virtual Sequencer receives sequence requests and dispatches them to the appropriate Agent's Sequencers.

- Each Agent's Sequencer manages the execution of sequences specific to its Agent.
- The Sequencers receive sequence items from the Virtual Sequencer.
- Sequencers forward sequence items to the respective Drivers for transaction execution.
- Drivers are responsible for driving write and read operations on the DUT.
- Drivers verify the data against expected values.
- Monitors capture transactions and responses from the DUT.
- Monitors ensure the correctness of the design under test.

### Advantages of Using Approach 2

**Simplicity**: Reduces the complexity of the Virtual Sequence by delegating the execution to the Virtual Sequencer.

**Modularity**: Enhances modularity and separation of concerns by using a Virtual Sequencer to manage the execution of Sub-Sequences.

**Flexibility**: Provides flexibility in adapting to changes or updates in the design through the abstraction layer of the Virtual Sequencer.

## 4.3.5.4 Hybrid Approach

Integrating both regular and virtual sequences, along with their respective sequencers, in your verification setup enables you to harness the unique advantages of each method. Regular sequences excel in executing deterministic tests, offering meticulous control over test scenarios. In contrast, virtual sequences excel in autonomously generating random scenarios, thereby improving test coverage and revealing elusive bugs. Adopting this hybrid strategy furnishes your verification environment with adaptability, adeptly catering to a diverse range of test scenarios

### 4.3.6 Messaging and Reporting Mechanism

In UVM (Universal Verification Methodology), various messaging and reporting mechanisms are provided to assist in debugging, reporting errors, and providing informational messages during the verification process.

#### 4.3.6.1 uvm_fatal

This macro is used to report critical errors that halt the simulation immediately. When triggered, it indicates a severe issue that renders the simulation irrecoverable.

#### 4.3.6.2 uvm_error

This macro is used to report non-recoverable errors. It logs the error but allows the simulation to continue.

#### 4.3.6.3 uvm_warning

This macro is used to report issues that are not critical but should be addressed. It provides a warning message without halting the simulation.

#### 4.3.6.4 uvm_info

This macro is used to provide informational messages. It displays non-critical information that can be useful for debugging or understanding the simulation flow.

### 4.3.7 Verbosity Levels

In UVM (Universal Verification Methodology), verbosity levels are used to control the amount of informational output printed during simulation. UVM provides several predefined verbosity levels, allowing you to adjust the level of detail in the simulation logs.

#### 4.3.7.1 UVM_NONE

No messages are printed. It silences all messages.

#### 4.3.7.2 UVM_LOW

Only critical and high-level messages are printed. Typically, this level includes only error and warning messages.

#### 4.3.7.3 UVM_MEDIUM

In addition to the messages printed at '**UVM_LOW**', this level includes informational messages. It provides a balance between verbosity and the amount of information displayed.

#### 4.3.7.4 UVM_HIGH

This level includes all the messages from '**UVM_MEDIUM**' plus additional debug messages. It's useful for debugging and understanding the flow of the simulation.

#### 4.3.7.5 UVM_FULL

The most verbose level, which includes all the messages from the previous levels plus detailed trace messages. This level provides a complete view of the simulation's activities but can generate a large amount of output.

### 4.3.8 TLM Ports and Exports

Ports and exports are used to establish connections between components. Ports are usually defined in the components that need to communicate with other components, while exports are defined in the components that provide services or capabilities.

#### 4.3.8.1 Ports

Ports in the Universal Verification Methodology (UVM) act as communication interfaces that facilitate the exchange of data and control signals between different UVM components. Ports can be categorized based on their primary function and directionality. For instance, an analysis port primarily serves as an output from an agent or a driver to transmit data transactions to other components in the testbench. On the other hand, blocking put and get ports are typically bidirectional, allowing a component to both send and receive data transactions. The directionality and type of a port determine its role in the testbench architecture and how it interacts with other components.

#### 4.3.8.2 Exports

Exports in UVM complement the functionality of ports by serving as their counterparts for receiving data transactions or signals. Unlike ports, which are generally outputs, exports act as inputs to UVM components. An analysis export, for instance, is commonly used as an input for a scoreboard component, receiving data transactions from an analysis port. Exports are crucial for establishing the communication link between different components in the testbench, ensuring that data flows seamlessly and efficiently.

### 4.3.8.3 Connect in UVM

The connect() method in UVM is the mechanism used to bind an export to a port, establishing the communication pathway between two UVM components. This binding enables data transactions and control signals to flow between the connected components, facilitating coordinated and synchronized operations within the testbench. The connect() method plays a pivotal role in setting up the testbench architecture, allowing for modular and scalable designs. By connecting ports to exports, UVM ensures a structured and organized approach to verification, enhancing the robustness and reliability of the testbench.

## 4.3.9 Configuration Database in UVM

The Configuration Database (config_db) is a powerful feature in the Universal Verification Methodology (UVM) that provides a centralized and flexible mechanism for storing and retrieving configuration settings, parameters, and objects across different UVM components. It serves as a repository where data can be stored and later accessed by any component in the testbench, facilitating modularity, reusability, and configurability.

### 4.3.9.1 Key Features and Benefits

- **‘config_db’** allows for the centralized storage of configuration data, reducing redundancy and ensuring consistency across the testbench.
- Components can access configuration data based on a hierarchical naming scheme, allowing for structured and organized data retrieval.
- The ‘**config_db’** ensures type safety during data retrieval, preventing type mismatches and potential runtime errors.
- Components can dynamically update their configuration settings during runtime, enabling adaptive and flexible testbenches.

### 4.3.9.2 Usage and Implementation

- **Registering Data**: Components register their configuration data using the '**uvm_config_db#(T)::set'** method, where **T** is the data type. This associates the data with a specific hierarchical name.
- **Retrieving Data**: Components retrieve configuration data using the '**uvm_config_db#(T)::get'** method. The data is retrieved based on the hierarchical name and type.
- **Hierarchical Naming**: Hierarchical names are used to scope configuration data. Wildcards can be used for flexible and broad matching.

  *****: Matches any single level

  ******: Matches any number of levels

## 4.3.10 Verification Methodology

Verification methodology refers to the systematic approach and set of techniques used to verify the correctness, functionality, and performance of digital designs. It encompasses various processes, tools, and best practices aimed at ensuring that the design meets its specifications and requirements

### 4.3.10.1 Assertion-Based Verification (ABV)

- ABV involves the use of assertions, which are statements or properties that describe expected behavior or conditions within the design.
- Assertions are embedded directly into the RTL (Register Transfer Level) code or written in a separate assertion language like SystemVerilog Assertions (SVA) or Property Specification Language (PSL).
- During simulation, assertions are continuously evaluated, and if any assertion fails (i.e., the specified condition is violated), an error message is generated.

- ABV helps detect design bugs, protocol violations, and corner-case scenarios by systematically checking for desired properties throughout the simulation.
- Assertions serve as executable specifications that clarify design intent and provide documentation for design behavior.

## Advantages of Assertion-Based Verification

- Highlight the benefits of using assertions as executable specifications to capture design intent and verify correct behavior.
- Discuss how assertions help detect design bugs, protocol violations, and corner-case scenarios early in the verification process

## 4.3.10.2 Constrained-Random Verification (CRV)

- CRV involves generating random stimuli for the DUT (Device Under Test) based on constraints defined by the verification engineer.
- Constraints restrict the randomness of the stimulus to ensure that it adheres to realistic scenarios and design specifications.
- Random stimulus generation helps achieve higher coverage by exploring a wide range of possible input scenarios, including both expected and corner-case behaviors.
- Coverage metrics are used to measure the effectiveness of CRV, ensuring that the verification environment exercises all relevant parts of the design.

## Advantages of Constrained-Random Verification

- Outline the advantages of CRV in exploring a wide range of input scenarios and achieving high verification coverage.
- Explain how constraints guide the generation of realistic stimulus, ensuring thorough verification of the design under test

# CHAPTER 5

# SOFTWARE DESCRIPTION

## 5.1 EDA Playground

EDA Playground is an online platform that allows users to compile and simulate System Verilog, Verilog, VHDL, and other hardware description languages (HDLs) in a web browser. The specific software specifications used in EDA Playground may include: Synthesis and Simulation Tools: EDA Playground typically utilizes industry-standard tools for synthesis and simulation, such as Synopsys Design Compiler, Cadence Incisive, Mentor Graphics ModelSim, or open-source tools like Icarus Verilog and GHDL.


***Figure 5.1 EDA Playground***

### 5.1.1 Simulation Engine

EDA Playground employs simulation engines to execute the compiled hardware description language code and generate waveforms. This could be based on tools like ModelSim or open-source simulators.

### 5.1.2 Waveform Viewer

A waveform viewer is essential for visualizing simulation results. EDA Playground may use its own waveform viewer or integrate with third-party viewers like GTK Wave.

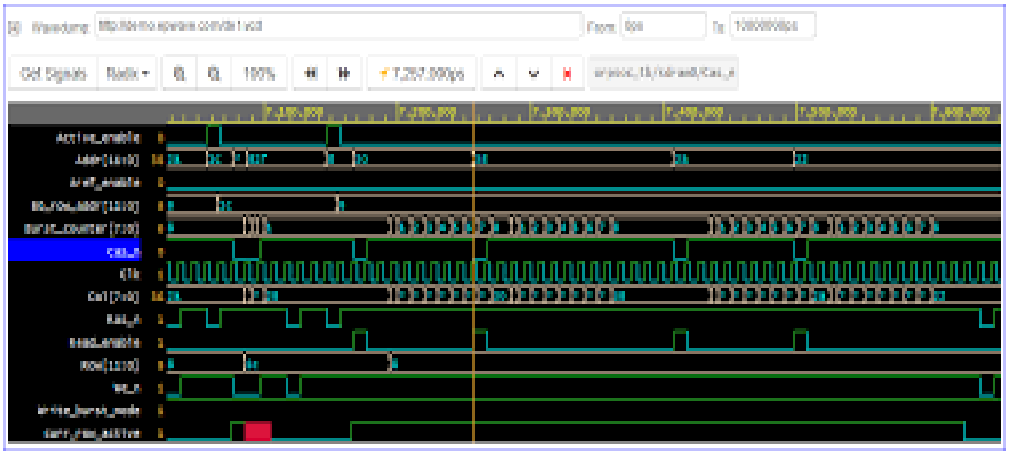

***Figure 5.1.2 Waveform Viewer***

## 5.2 Synopsis VCS 2020.09

Synopsis VCS 2020.09 is a leading Verilog simulator developed by Synopsys, designed to meet the demands of modern digital design and verification. It offers a comprehensive set of features and enhancements to streamline the simulation process and improve verification efficiency.

### 5.2.1 Features

- Improved support for SystemVerilog features, including interfaces, classes, and assertions, enabling engineers to develop and verify complex designs with ease
- Utilization of parallel processing and multi-threading techniques for faster compilation and simulation of large-scale designs, enhancing productivity and reducing time-to-market.

- Comprehensive debugging tools, including trace and assertion-based debugging, waveform analysis, and hierarchical visualization, facilitating efficient bug identification and resolution
- Seamless integration with popular development environments and third-party tools, ensuring compatibility with existing design and verification methodologies.
- Support for industry-standard file formats and protocols, enabling easy integration into existing workflows and environments.
- Built-in support for industry-standard verification methodologies, such as UVM (Universal Verification Methodology) and OVM (Open Verification Methodology), for scalable and efficient verification of complex designs.
- Implementation of robust security measures to protect intellectual property and sensitive data during simulation and verification activities.
- Scalable architecture to support simulation of designs ranging from small IP blocks to full-system-on-chip (SoC) implementations, with efficient resource management techniques to optimize performance and scalability.

# CHAPTER 6
# RESULT AND ANALYSIS

## 6.1 Analysis

The results of the assertion-based verification phase revealed critical insights into the functionality and correctness of the FIFO memory module. Through rigorous testing, data integrity was meticulously scrutinized, with assertions confirming whether the data read from the FIFO matched the expected values. Instances of mismatches were diligently logged and analyzed, shedding light on potential design flaws or anomalies within the module's operation. Additionally, the verification process generated detailed reports encapsulating the outcomes of each test case, offering comprehensive documentation of verification results and aiding in post-simulation analysis. Furthermore, waveform data was generated to visually inspect the behavior of signals and transactions within the FIFO, providing valuable insights into the module's operation and facilitating debugging efforts.

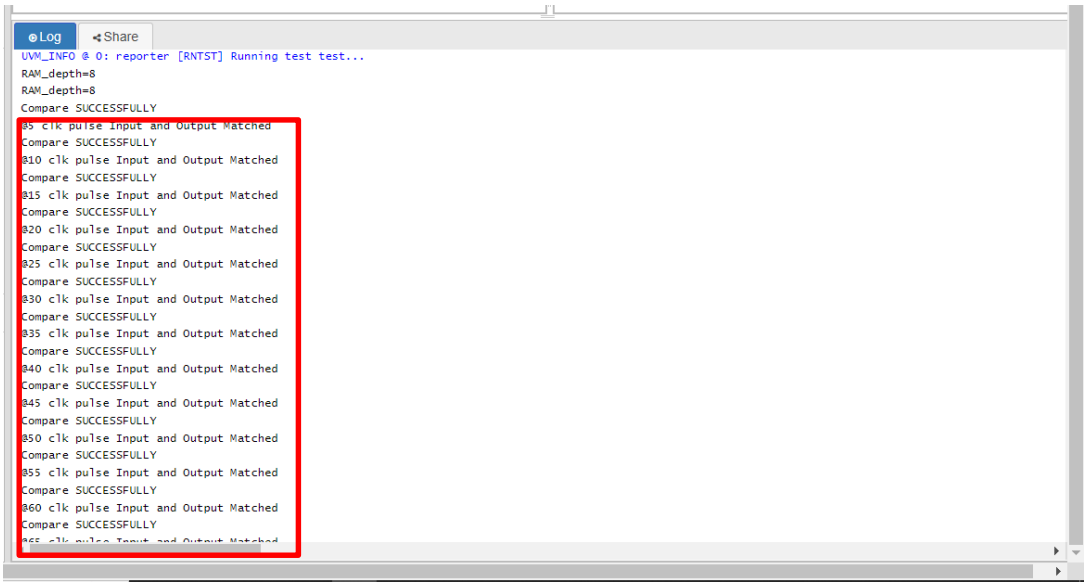


***Figure 6.1.1 Simulation Result***

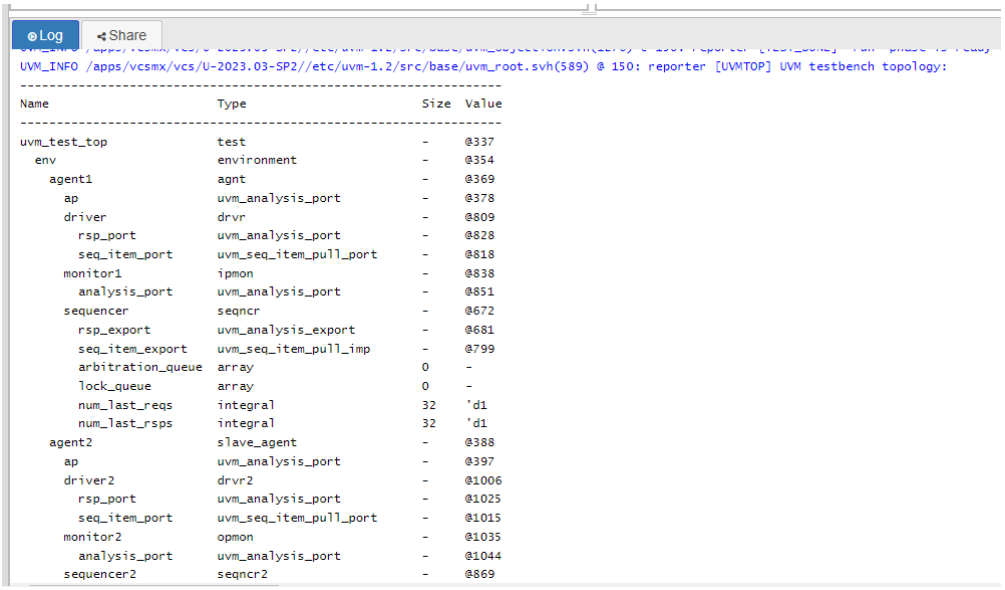


***Figure 6.1.2 Log Report***


***Figure 6.1.3 Report Summary***

## 6.2 Waveform output

**Empty and Full Conditions** indicating the status of the FIFO's empty and full conditions. These signals could be represented as boolean variables or binary signals, where a high (1) value indicates that the FIFO is either empty or full, and a low (0) value indicates that it is not.

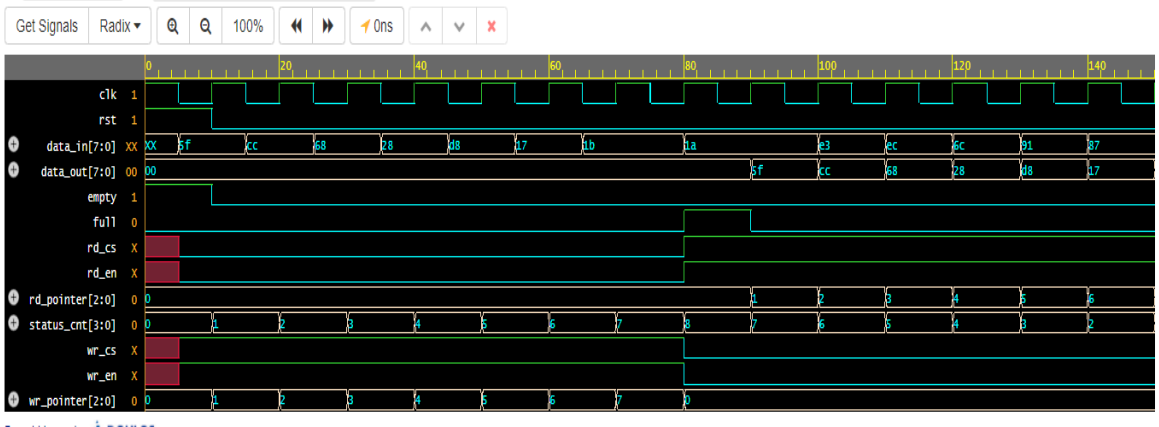


*Figure 6.2.1* Empty and Full Condition Waveform

**Write Operation** waveform corresponding to data being written into the FIFO. This might include transitions on signals representing the data inputs to the FIFO, as well as control signals indicating the start and completion of the write operation.

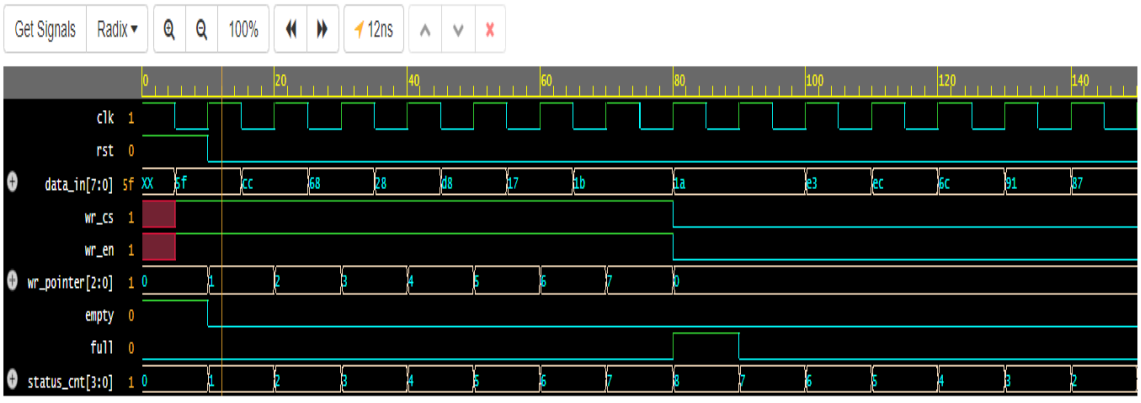


*Figure 6.2.2* Write Operation Waveform

**Read Operation** the waveform indicating the retrieval of data from the FIFO. This could involve transitions on signals representing the read request, as well as the data outputs from the FIFO.


*Figure 6.2.3* Read Operation Waveform

# CHAPTER 7

## 7.1 Conclusion

The project successfully implemented and verified a FIFO memory module using advanced verification methodologies within the Universal Verification Methodology (UVM) framework, integrating a master-slave concept to partition responsibilities between agents. By leveraging techniques such as virtual sequences and assertion-based verification, combined with the master-slave architecture, the project showcased robust functionality and correctness of the FIFO module in a multi-agent environment. This approach not only enhanced the scalability and efficiency of the verification process but also facilitated parallel and coordinated verification, allowing for more complex and realistic test scenarios. Through rigorous testing and analysis, various scenarios were explored to ensure comprehensive coverage and reliability. Overall, the adoption of modern verification techniques and the master-slave concept underscored the project's commitment to ensuring the integrity and reliability of digital designs, paving the way for more efficient, scalable, and successful semiconductor development in the future.

## 7.2 Future Scope

- Extend the project's verification environment to accommodate larger memory spaces, leveraging the master-slave architecture to manage and validate increased data sizes effectively.
- Adapt verification components to handle the complexities of larger memory capacities and diverse data volumes within the master-slave framework, ensuring seamless communication and coordination between agents.

- Integrate corner cases, edge scenarios, and advanced features such as dynamic resizing and multi-port access into the verification strategy to enhance the module's robustness, reliability, and functionality under various conditions.

- Develop specialized test cases and scenarios to validate the FIFO memory module's behavior under diverse and challenging conditions within the master-slave environment, ensuring comprehensive verification coverage.

- Continuously refine and optimize the master-slave verification environment to improve efficiency, effectiveness, and scalability.

- Leverage the insights gained from ongoing verification activities, particularly within the master-slave context, to iteratively enhance the verification environment and processes.

- Ensure that the verification infrastructure, designed with the master-slave architecture in mind, can seamlessly handle future iterations of the FIFO memory module and other digital designs.

- Foster a culture of continuous improvement and innovation within the verification team, encouraging the exploration of new techniques, methodologies, and tools, including those tailored for master-slave verification.

- Engage in collaborative research and development efforts with industry partners and academic institutions to push the boundaries of digital design verification.

# REFERENCES


[1] Apoorva H M and Dr. Kiran Bailey, "UVM based Design Verification of FIFO", *International Journal of Engineering Research & Technology (IJERT),* Vol. 9, pp 774-776, 2020.

[2] Agrawal, Navaid Z, Niraj, Rizvi , Rajat Arora, "Implementation and Verification of Synchronous FIFO using System Verilog Verification Methodology"**,** *Journal of Communications Technology*, Vol 8, pp 18-23, ISSN 2457-905X, 2015.

[3] Abhishek Jain and Richa Gupta "Expanding the UVM Register Model towards Automation and Simplicity of Use", *International Journal of Advanced Research in Computer Science*, Volume 8, No. 3, pp 471-480 2017.

[4] Clifford E. Cummings and Janick Bergeron "Using UVM Virtual Sequencers & Virtual Sequences" *World Class System Verilog & UVM Training,* DVCon, Vol 8, pp 1-26, 2016

[5] Josep Sans I Prats, "Verification of a microprocessor's memory pipeline with UVM" *Final Master Thesis Master in Innovation and Research in Informatics,*Vol 9, pp 1-34, 2022

[6] Bidisha Kashyap and Ravi V "Universal Verification Methodology Based Verification of UART Protocol" *National Science, Engineering and Technology Conference (NCSET)*, pp 1-6, 2020

[7] Khaled Fathy and Khaled Salah "An Efficient Scenario Based Testing Methodology Using UVM", *17th International Workshop on Microprocessor and SOC Test and Verification,* Vol 1, pp 57-60, *2016.*

[8] Agustin Rodriguez, Juan Francesconi J., Pedro M. Julian "UVM Based Testbench Architecture for Unit Verification" *Argentine School of Micro-Nanoelectronics, Technology and Applications*, Vol 9, pp 89-94, *2014.*

[9] Deepika, Jayanthi K Murthy, "Interrupt Enabled Priority Based Master Slave Communication using SPI Protocol", *International Journal of Innovative Technology and Exploring Engineering (IJITEE),* ISSN: 2278-3075 (Online), Volume-9 pp-9-13, July 2020.

[10] CHETAN N, R KRISHNA, "Verification of SPI protocol Single Master Multiple Slaves using Systemverilog and Universal Verification Methodology (UVM)" , *International Journal of Engineering Research and Applications(IJERA),* Vol. 11, Issue 7, (Series-VI), pp. 01-08, July 2021.