The Design and Verification of a Synchronous First-In First-Out (FIFO) Module Using System Verilog Based Universal Verification Methodology (UVM)

**Table of Contents**

[Chapter 1: Design Overview 4](#_Toc406486740)

[Chapter 2: FIFO features 5](#_Toc406486741)

[Chapter 3: FIFO Verification Plan 6](#_Toc406486744)

[3.1 Feature Extraction 6](#_Toc406486745)

[3.2 Coverage Plan 6](#_Toc406486746)

[3.3 Checker Plan 6](#_Toc406486747)

[3.4 Verification Environment Development 7](#_Toc406486749)

[3.5 Test suite development 7](#_Toc406486750)

[3.5.1 Directed Testcases 7](#_Toc406486751)

[3.5.2 Random Testcases 7](#_Toc406486752)

[Chapter 4: FIFO Architecture 8](#_Toc406486753)

[4.1 FIFO Architecture Block Diagram 8](#_Toc406486754)

[4.2 Verification components 9](#_Toc406486755)

[4.2.1 Transaction(Data Items) 9](#_Toc406486756)

[4.2.2 Agent 10](#_Toc406486757)

[4.2.3 Agent Block Diagram 10](#_Toc406486758)

[4.2.4 Driver 11](#_Toc406486759)

[4.2.5 Driver Block Diagram 12](#_Toc406486760)

[4.2.6 Monitor 13](#_Toc406486761)

[4.2.7 Monitor Block Diagram 14](#_Toc406486761)

[4.2.8 Sequencer 15](#_Toc406486761)

[4.2.9 Environment 16](#_Toc406486761)

[Chapter 5: Test Suite 17](#_Toc406486771)

[Chapter 6: Running Simulation 18](#_Toc406486772)

**List of Figures**

Figure 1‑1: FIFO Architecture Block Diagram……………………………………………………………..8

Figure 1‑2: Active Agent Block Diagram…………………………………...............................................10

Figure 1‑3: Passive Agent Block Diagram…………………………………………………………….......10

Figure 1-4: Driver Block Diagram…………………………………………………………………………..12

Figure 1-5: Monitor Block Diagram…………………………………………………………………………14

# Chapter 1

# Design Overview

The building blocks of a synchronous FIFO include memory array and flag logic controlled by the read control logic and the write control logic. An array of flip-flops forms the memory array and width and depth expansion of the array can be achieved easily through parameterization.

The FIFO module can efficiently handle two operations :

1. Writing to the FIFO
2. Reading from the FIFO,

Which are operating at different speeds. In the case of a read operation, the *rd\_enb* (read enable) signal help in successful data read from the memory array. In the case of a write operation, the *wr\_enb* (write enable) signal helps in successful data write into the memory array.

There are two pointers, *wptr* (write pointer) and *rptr* (read pointer), which help in steering the data into and out of the memory array. They store the write and read address value associated with the memory array. After each successful data write and / or read, the corresponding pointer is incremented by one to point to the next address.

Those two address pointers are involved in flag logic. Flag logic uses the information in both the pointers to generate flags based on the comparison between the read address pointer and the write address pointer.

If the difference between the two pointers is zero, the empty flag is asserted denoting the FIFO empty status. If the difference between the two pointers is equal to the FIFO depth, the full flag is asserted denoting the FIFO full status.

Similarly, other flags such as almost full flag and almost empty flag are generated by comparing the offset value specified in the program with the word count in the memory array.

# Chapter 2

# FIFO Features

The key features of synchronous FIFO include –

* + **Clock** – To ensure sampling of an output at negedge and driving of an input

signals at posedge.

* + **Reset** – To switch the design from unknown state to known state.
  + **Write** – To store the data in sequential order into the memory array.
  + **Read** – To access or retrieve the already written data from the memory array.

# Chapter 3

# FIFO Verification Plan

Verification involves studying the relevant specifications, extracting features from it that are to be tested, devising a strategy as to how these features are to be tested, developing a verification environment based on the strategy, writing test cases to cover all the scenarios and achieving 100% functional coverage figures.

# 3.1 Feature Extraction

“Feature Extraction” involves listing out features to be tested from the specification. A feature list (spreadsheets) has been prepared using the fifo\_verification\_plan document.

The above spreadsheet lists out features to be tested in the corresponding specification, assigns a feature id to them and tells how the feature is to be tested (test case name or checker task name).

# 3.2 Coverage Plan

A functional coverage plan needs to be made based on the feature extraction document. This plan lists out the various combinations of stimuli that need to be generated for the proper verification of the FIFO. This has been included in the feature extraction spreadsheet itself.

# 3.3 Checker Plan

A checker plan needs to be made based on the feature extraction document. Implementations for the features marked as “checker” are elaborated here. We have included it in the feature extraction spreadsheet itself.

**3.4 Verification Environment Development**

Our environment is based on system verilog and UVM.

**3.5 Test suite Development**

**3.5.1 Directed Test Cases**

Test cases written to test specific areas of the FIFO or to generate a specific kind or sequence of transactions is known as a directed test case. These test cases are helpful in the initial and final stages of verification. In the initial stages, when neither the verification environment nor the FIFO is matured, these test cases help in checking and correcting specific pieces of code in both the FIFO and the verification environment. In the final stages, they are used to hit specific functional or code coverage areas.

**3.5.2 Random Test Cases**

Random test cases are written to test the FIFO. Random scenarios are generated based on constraints provided in the test case. These test cases are run several times with different seed numbers to generate different scenarios to achieve more functional coverage figure.

# Chapter 4

# FIFO Architecture

**4.1 FIFO Architecture Block diagram**

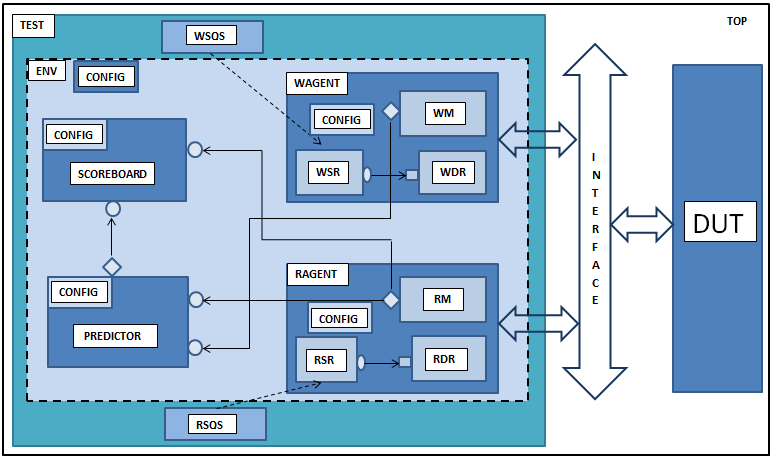


Figure 1-1 **FIFO Architecture Block Diagram**

**4.2 Verification Components**

This section deals with a short description of each verification component.

**4.2.1 Transaction (Data Items)**

This component is responsible for creating different meaningful tests for the DUT. These tests are basically the inputs given to the DUT. The inputs to the DUT are called as data items. Maximum test coverage is attained by intelligently randomizing the fields of the data items using SystemVerilog constraints.

**4.2.2 Agent**

As part of the UVM specification, it is recommended that test bench creators make a more abstract container called an agent which encapsulates a sequencer, a driver and a monitor. And all these components can be called through agent. Verification environments can have more than one agent. Some agents can be master agents involved in initiating transactions to the DUT, while other agents can be slave agents that react to transaction requests. Agents have to be configured to act as either an active or a passive agent. Active agents are responsible for driving transactions, while passive agents are involved only in monitoring the DUT behaviour.

**4.2.3 Agent Block Diagram**

**CONFIG**

**CONFIG**

**MONITOR**

**MONITOR**

**DRIVER**

**SEQUENCER**

Fig 1-2 **Active Agent** Fig 1-3 **Passive Agent**

**4.2.4 Driver**

This component is responsible for driving signals to the DUT. It receives data items repeatedly from the Sequencer and subsequently drives signals to the DUT. Whatever transaction it gets, it samples them and finally issues them into the DUT. For example, in the case of the Synchronous FIFO module, this component repeatedly receives write address, write data from the sequence class where data items are randomized and feeds these inputs to the DUT. Using UVM predefined ports, these randomized sequences can be collected and sent to some other component in the verification environment.

**4.2.5 Driver Block Diagram**



Fig 1-4 **Driver Block Diagram**

**4.2.6 Monitor**

This component is responsible for sampling the DUT signals without driving them. It collects coverage details and does checking. Apart from collecting coverage and performing checks, it has other functionalities as given below.

* The Monitor collects data items from the DUT and translates it into a transaction making it available to other verification components and to the test writer.
* It notifies other components about the availability of transaction through an event emission.
* It also captures status information that are made available to other components and to the test writer.
* Trace information can also be printed using a monitor.

Monitors can be of two types, namely bus monitor and agent monitor. All bus signals and bus related transactions are handled by a bus monitor. Signals and transaction related to a specific agent are handled by an agent monitor. It is always recommended to create a monitor that doesn’t depend on driver for information.

**4.2.7 Monitor Block Diagram**



Fig 1-5 **Monitor Block Diagram**

**4.2.8 Sequencer**

A Sequencer happens to be the stimulus generator that controls the data items fed to the Driver. Basically, constrained randomization is done by this component. A Sequencer and a Driver operate based on a request-acknowledgment protocol. The driver requests the next sequence from the Sequencer. To control the randomization of data items, this component adds constraints to the extended UVM\_sequence\_item class.

It also has its built-in capabilities, including:

* Reaction to the DUT’s current state for every sequence generated.
* Knows the order between data items, thereby, forming a meaningful and structured stimulus pattern.
* Controls multiple interfaces and allows synchronization.
* Enables time modeling.

**4.2.9 Environment**

The environment happens to be the top-level component in a verification test bench. It contains all the verification components. It also contains configuration properties that allow customization of topology and behavior, making it reusable. Any verification component can have an environment-level monitor that performs checking and collection of coverage details not related to a single agent. Pre-existing uvm\_env class can be extended and configured according to the project specifications. Environment class is responsible for modeling the behavior of the DUT.

# Chapter 6

# Test Suite

# Chapter 7

# Running Simulation