Zewail City for Science and Technology
University of Science and Technology
Nanotechnology and Nanoelectronics Engineering

UVM-Based Functional Verification of PCIe PHY Layer

A Graduation Project
Submitted in Partial Fulfillment of
B.Sc. Degree Requirements in
Nanotechnology and Nanoelectronics Engineering

Prepared By:

Mahmoud Marzouk	202000258
Marwan Eid	201901246
Peter Gad	201901588
Alaa Taha	201900983
Almomenbelah Allam	201900102

Supervised By:

Dr. Rania Hassan
Dr. Hassan Mostafa
Eng. Sherif Hosny

2023 – 2024

Acknowledgements

We would like to express our deep appreciation to our supervisors, Dr. Rania Hassan and Dr. Hassan Mostafa for their invaluable guidance, encouragement, and support throughout the project. Their expertise and valuable insights were instrumental in shaping this work and ensuring its completion.

We extend our deepest and most sincere gratitude to Eng. Sherif Hosny, who dedicated countless hours and immense effort to guide and assist us with this project. His unwavering commitment and extraordinary contributions were crucial to its success. We are profoundly thankful for his exceptional dedication and the significant impact he has made. Without his invaluable assistance, this work would not have been possible. Eng. Sherif, your remarkable efforts and unwavering support are truly appreciated and will always be remembered.

Furthermore, we extend our genuine thanks to our families and friends for their support and encouragement during this challenging yet rewarding journey.
Lastly, we would like to express our gratitude to Dr. Ahmed Zewail, the staff, and faculty of The University of Science and Technology in Zewail City, whose dedication to academic excellence and a nurturing environment provided the foundation upon which this thesis was built.
This thesis would not have been possible without the guidance, support, and encouragement of the aforementioned individuals. We are truly grateful for their contributions and support.

Declaration

We declare that this thesis, entitled "UVM-Based Functional Verification of PCIe PHY Layer," has been composed solely by ourselves and that it has not been submitted, in whole or in part, in any previous application for a degree or qualification. Except where stated otherwise by reference or acknowledgment, the work presented is entirely our own.

Marwan Eid
Alaa Taha
Peter Gad
Almomenbelah Allam
Mahmoud Marzouk

June 2024

Abstract

The Thesis titled "UVM-Based Functional Verification of PCIe PHY Layer" explores the integral role of PCIe as a high-speed communication protocol in modern computing systems. The study begins by describing the evolution of PCIe from its inception, through subsequent generations, up to the latest Gen6 release. The focus then shifts to a detailed analysis of PCIe's underlying architecture and operation, including its layered protocol structure, PHY, DLL, and TL. Particular attention is given to the transmission and reception of data packets through the use of lanes and innovative solutions in the different generations. Furthermore, a thorough explanation of the PCIe Gen1 PHY layer is performed along with the project design, constraints, and alternatives. A comprehensive overview of the verification approach is presented, which involves a thorough comparison of the implemented RTL code against specifications and standards. The process includes identifying discrepancies, building a verification plan, creating a UVM-based verification environment on both block- and chip-level, and achieving robust code and functional coverage. Finally, bugs are reported and most bugs are resolved then the project is concluded and future work is discussed.

Contents

Acknowledgements

Declaration

Abstract

List of Figures

List of Acronyms/Abbreviations

1 Introduction **1**
1.1 General Introduction and Overview 1
1.2 Problem Definition and Objectives 6
1.3 Report Organization 6

2 Literature Review and Standards **8**
2.1 History of PCIe 8
2.2 PCI Express Base Specification Revision 1.0 9
2.2.1 PHY Overview and Function 9
2.2.2 PHY Sub-Blocks 10
2.3 Other Standards 26
2.3.1 PHY Interface for the PCI Express 3.0 Architecture Revision .9 . . 26
2.3.2 IEEE Standard for SystemVerilog— Unified Hardware Design, Specification, and Verification Language (IEEE Std 1800 ™ - 2017) 27
2.3.3 IEEE Standard for Universal Verification Methodology Language Reference Manual (IEEE Std 1800.2 ™ - 2020) 27
2.4 VLSI Verification 28
2.4.1 Challenges and Trends 28
2.4.2 Universal Verification Methodology 34
2.5 Literature on PCIe Verification 38

3 Project Design and Planning **39**
3.1 Description of The Selected Design 39
3.2 Project Purpose and Constraints 42
3.2.1 Project Purpose 42
3.2.2 Project Constraints 43
3.3 Project Tasks and Gantt Chart 43
3.4 Verification Approach 44

4 Block-Level Verification **45**
4.1 LTSSM Blocks 48
4.1.1 PIPE Operation Block 48
4.1.2 OS Decoder 54
4.1.3 Timer 66
4.1.4 OS Creator 71
4.1.5 State Machine 82
4.2 Transmitter Blocks 97
4.2.1 Tx Buffers Collection 97
4.2.2 Tx Muxes 106
4.2.3 OS Buffer 112
4.2.4 Encoder and Scrambler 117
4.2.5 Controller 124

5 Chip-Level Verification **129**
5.1 LTSSM System 129
5.1.1 Overview 129
5.1.2 Top Environment 129
5.1.3 Binding Interfaces with Modules 131
5.1.4 Evaluation 132
5.2 Tx System 134
5.2.1 Top Environment 134

6 Conclusion and Future Work **135**
6.1 Conclusion 135
6.2 Future Work 135

References **137**

Appendices **139**

List of Figures

1.1 Actual PCIe Bandwidth vs Required I/O Bandwidth 1
1.2 PCIe Generations and Bandwidth Evolution 2
1.3 PCIe Dual-Simplex Link 2
1.4 PCIe Lane and Differential Signaling 3
1.5 PCIe Layered Architecture 4

2.1 PCIe Modulation Evolution: NRZ vs PAM4 9
2.2 Logical and Electrical Sub-Blocks of The PHY 10
2.3 PHY Transmit Logic 12
2.4 Gen 1 Byte Striping 13
2.5 Gen1 Scrambler Algorithm 13
2.6 Gen1 Encoding Example 14
2.7 Gen1 Encoding and Serialization 15
2.8 PHY Receive Logic 16
2.9 Gen1 Receiver Logic Front-End 17
2.10 Gen1 Byte Un-Striping 17
2.11 Lane Reversal 19
2.12 Polarity Inversion 19
2.13 LTSSM States and Transitions 20
2.14 Detect State Machine 22
2.15 Polling State Machine 22
2.16 Summary of Substates in The Normal Flow of Link Training Process . . 24
2.17 Differential Common-Mode Noise Rejection 25
2.18 Transmission with De-Emphasis 26
2.19 PHY Specification according to The PIPE Standard 26
2.20 ASIC Verification Language Adoption 27
2.21 ASIC Verification Methodologies Adoption 28
2.22 VLSI Digital Design Flow 29
2.23 Directed Testing Progress (Coverage) Over Time 31
2.24 Random vs Directed Testing Progress (Coverage) Over Time 32
2.25 Coverage Convergence 33

2.26 Bug Rate Throughout a Typical Project 35
2.27 Typical UVM Testbench Architecture 36
2.28 UVM Agent 37

3.1 Design Architecture 39
3.2 LTSSM Architecture 40
3.3 Tx Architecture 41
3.4 Rx Architecture 41
3.5 Gantt Chart 44

4.1 Agent internal structure 45
4.2 Stimulus Coordination using Virtual Sequencer 46
4.3 Generic Checking Mechanism 47
4.4 LTSSM System - PIPE Block Highlighted 48
4.5 PIPE Operation Block Diagram 49
4.6 PIPE Block Functionality 52
4.7 PIPE Block Final Log 52
4.8 PIPE Block Code Coverage Details 53
4.9 PIPE Block Functional Coverage 53
4.10 LTSSM implementation - Decoder highlighted 54
4.11 OS_Decoder implementation 55
4.12 Decoder testbench architecture and hierarchy 58
4.13 Decoder Final Log 65
4.14 Decoder code coverage 65
4.15 Decoder Functional Coverage 66
4.16 LTSSM Implementation – Timer Highlighted 67
4.17 Timer Block Diagram 68
4.18 Timer Waveforms 69
4.19 Timer Waveforms (zoomed-in) 69
4.20 Timer Final Log 70
4.21 Timer Functional Coverage 70
4.22 LTSSM Implementation – Creator Highlighted 71
4.23 OS_Creator implementation 72
4.24 OS_Creator internal modules 75
4.25 OS_Creator 76
4.26 Creator Final Log 80
4.27 Creator Code Coverage 81
4.28 Creator functional coverage 82
4.29 LTSSM System - FSM Highlighted 83
4.30 FSM Block Diagram 84

4.31 FSM UVM Hierarchy 88
4.32 Downstream Device States Flow from Detect Quiet to L0 93
4.33 Time-out condition in each state 93
4.34 Random combinations of Ack and time-out signals 94
4.35 FSM Final Log 94
4.36 FSM Code Coverage Details 95
4.37 FSM Cross Bins Coverage 95
4.38 FSM Coverpoints 96
4.39 Position of the buffers collection modules 97
4.40 Buffers Collection Block Diagram 98
4.41 Buffers Collection testbench architecture 101
4.42 Buffers test waveforms 102
4.43 buffers functional coverage 104
4.44 buffers code coverage 105
4.45 Tx_Muxes interaction with other blocks. 106
4.46 Muxes Block Diagram 107
4.47 Cover group Tx_Muxes. 111
4.48 Tx System - OS Buffer Highlighted 112
4.49 OS Buffer Block Diagram 113
4.50 OS Buffer Testbench Architecture 114
4.51 OS Buffer Code Coverage 116
4.52 OS Buffer Functional Coverage 117
4.53 OS Buffer Read Empty Bug 117
4.54 The position of the modules in the transmitter architecture. 118
4.55 Block Diagram encoding 118
4.56 Architecture and Hierarchy Encoding Environment 121
4.57 Checking mechanism encoding. 122
4.58 Cover group encoding. 123
4.59 Code Coverage encoding. 124
4.60 Encoder Waveform Results 124
4.61 Controller test functional coverage 128
4.62 Controller Code Coverage 128

5.1 Top LTSSM Environment 130
5.2 Binding in Passive Agent 131
5.3 LTSSM Top Functional Coverage 132
5.4 LTSSM Top Errors Report 133
5.5 Top Tx Environment 134

List of Acronyms/Abbreviations

AGP	Accelerated Graphics Port
AI	Artificial Intelligence
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
ASPM	Active State Power Management
BCL	Base Class Library
CDR	Clock and Data Recovery
COM	Comma Character
DLL	Data Link Layer
DLLP	Data Link Layer Packet
DUT	Design Under Test
DV	Digital Verification
DVE	Discovery Visual Environment
EDA	Electronic Design Automation
EIEOS	Electrical Idle Exit Ordered Set
EIOS	End of Idle Ordered Set
FEC	Forward Error Correction
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
FTS	Fast Training Sequence

GHz Giga Hertz

GT Giga Transfer

GTPS Giga Transfers Per Second

HDL Hardware Description Language

IDL Idle Character

IEEE Institute of Electrical and Electronics Engineers

IEEE-SV IEEE Standard for SystemVerilog— Unified Hardware Design, Specification, and Verification Language

IEEE-UVM IEEE Standard for Universal Verification Methodology Language Reference Manual

IP Intellectual Property

ISE Integrated Synthesis Environment

ISI Inter-Symbol Interference

LCRC Link Cyclic Redundancy Code

LFSR Linear Feedback Shift Register

LSB Least Significant Bit

LTSSM Link Training and Status State Machine

MAC Media Access Layer

MHz Mega Hertz

ML Machine Learning

MSB Most Significant Bit

MUX Multiplexer

NRZ Non-Return-to-Zero

OOP Object-Oriented Programming

OVM Open Verification Methodology

PAD Pad Character

PAM4 4-level Pulse Amplitude Modulation

PCIe Peripheral Component Interconnect Express

PCIeBS PCI Express Base Specification Revision 1.0

PCI-SIG Peripheral Component Interconnect Special Interest Group

PCS Physical Coding Sublayer

PHY Physical Layer

PIPE PHY Interface for the PCI Express

PLL Phase Locked Loop

PM Power Management

PMA Physical Media Attachment Layer

RTL Register Transfer Level

SKP Skip Character

SOS SKIP Ordered Set

SRoT Secure Root of Trust

STR Secure Transaction Replay

TL Transaction Layer

TLM Transaction Level Modeling

TLP Transaction Layer Packet

TS Training Sequence

TS1 Training Sequence 1

TS2 Training Sequence 2

URM Universal Reuse Methodology

UVM Universal Verification Methodology

VCS Verilog Compiler Simulator

VLSI Very Large Scale Integration

VMM Verification Methodology Manual

Chapter 1

Introduction

1.1 General Introduction and Overview

The accelerated evolution of technology and the surge in the amount of data creation drive a demand for high-speed computing. These factors have stimulated the development of improved and more effective connections. Within this spectrum, the Peripheral Component Interconnect Express (PCIe) standard has risen as a prevailing interface within present-day computing systems [1]. According to Statista, an estimated 328.77 million terabytes of data are created every day [2]. This gigantic amount of data requires a fast and reliable transfer, and this is where PCIe becomes crucial. The bandwidth of the serial point-to-point interconnect has kept up with the required Input/Output (I/O) bandwidth that doubles every three years, as shown in figure 1.1 [3].

Figure 1.1: Actual PCIe Bandwidth vs Required I/O Bandwidth

PCIe is a high-speed interconnect technology used to connect various components /devices within a computer system. This technology offers a standardized, point-to-point serial interface for data transmission, which ensures fast and reliable data transfer between connected devices. By shifting away from the traditional parallel bus topology, PCIe manages to support extremely high data transfer rates. Figure 1.2 shows the evolution of the interface and the bandwidth of every generation [4].

PCIe architecture	Raw bit rate	Interconnect bandwidth	Bandwidth per lane per direction	Total bandwidth for a x16 link
PCIe 1.x	2.5 GT/s	2.0 Gb/s	~250 MB/s	~8 GB/s
PCIe 2.x	5.0 GT/s	4.0 Gb/s	~500 MB/s	~16 GB/s
PCIe 3.x	8.0 GT/s	8.0 Gb/s	~1 GB/s	~32 GB/s
PCIe 4.0	16.0 GT/s	16.0 Gb/s	~2 GB/s	~64 GB/s
PCIe 5.0	32.0 GT/s	32.0 Gb/s	~4 GB/s	~128 GB/s
PCIe 6.0	64.0 GT/s	64.0 Gb/s	~8 GB/s	~256 GB/s

Figure 1.2: PCIe Generations and Bandwidth Evolution

The interconnect is primarily employed for interfacing expansion cards with the motherboard of a computer system, replacing former parallel bus designs such as Peripheral Component Interface (PCI) and Accelerated Graphics Port (AGP) with a more sophisticated and efficient serial interface. It features high data transfer rates, supporting rapid communication between devices [1]. Nevertheless, the primary drivers behind PCIe advancement are within the areas of cloud computing and storage, Artificial Intelligence (AI), and Machine Learning (ML) [3].

PCIe employs a point-to-point architecture, wherein each device is directly connected to the system's host controller via dedicated links, as illustrated in figure 1.3. This architecture eliminates the need for bus sharing and improves the overall system efficiency by allocating dedicated bandwidth to each device [1].

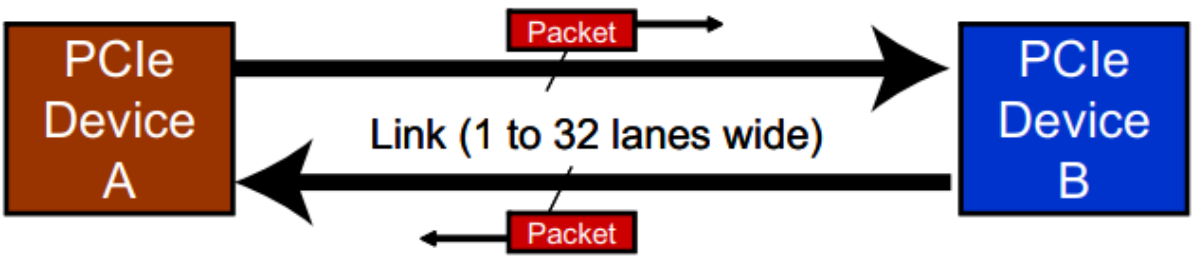


Figure 1.3: PCIe Dual-Simplex Link

For data transfer, PCIe utilizes lanes, each comprising two differential pairs facilitating bidirectional communication, as visualized in figure 1.4. The differential signaling

introduces a number of advantages, including improved noise immunity and reduced signal voltage. Because it is a dual-simplex connection, each interface has a simplex transmit path and a simplex receive path. The communication path between two devices is technically full duplex. The number of lanes varies based on PCIe generation and device requirements, with the allowed lane configurations being x1, x4, x8, x16, and x32 [5].

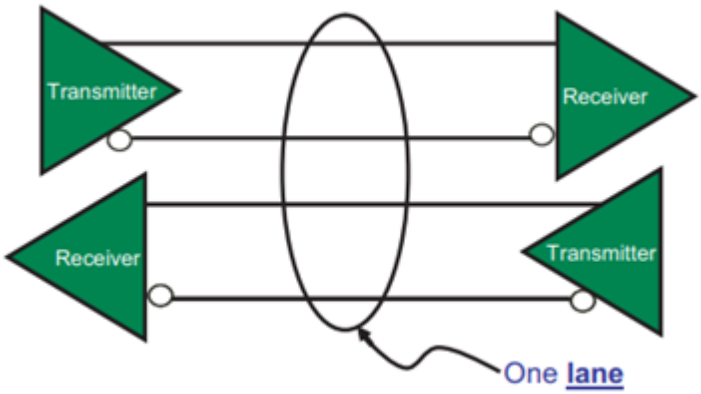


Figure 1.4: PCIe Lane and Differential Signaling

To avoid problems with clock skew and the propagation delay of signals, the PCIe system does not manage transactions using a common clock between the transmitter and receiver. Instead, the receiver extracts the clock from the received data stream by monitoring the time period between transitions. This is done using a Phase Locked Loop (PLL) that continuously produces an output clock that adapts to try to match the clock of the input stream. The encoding facilitates maintaining DC balance since PCIe uses an AC-coupled link and enhances error detection [5].

For the PLL to work properly, it needs to observe many voltage transitions, so the serial input data must not have long streams of only ones or only zeros. Gen1 and Gen2 use 8b/10b encoding to handle this problem. Each eight bits of data are modified into a ten-bit collection that increases the number of transitions. Gen3 and higher use a similar method called 128b/130b encoding. The encoding is done by the PHY of the transmitter and reversed in the PHY of the receiver [5].

PCIe is equipped with hot-plugging capability, enabling the connection or disconnection of devices while the system is powered on. This feature facilitates the straightforward installation and removal of expansion cards without the need to power down the system [1].

PCIe employs a layered protocol architecture, consisting of the Physical Layer (PHY),

Data Link Layer (DLL), and Transaction Layer (TL), as illustrated in figure 5. Each layer is responsible for specific functions and protocols to ensure reliable data transmission, accurate error detection and correction, flow control, and transaction management [1].

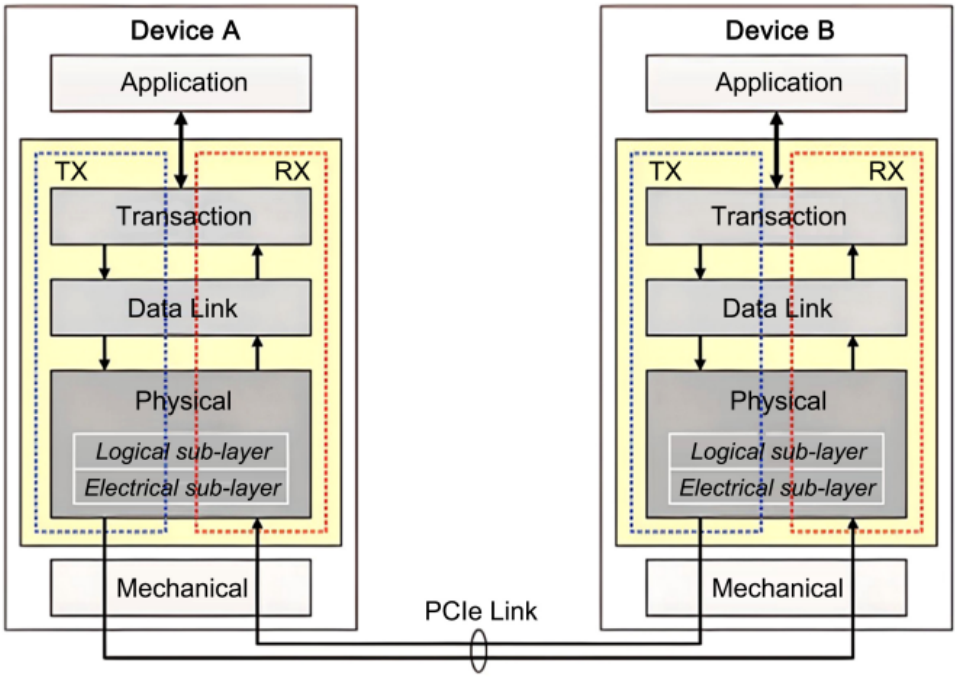


Figure 1.5: PCIe Layered Architecture

The TL is the uppermost PCIe architectural layer. It has direct interaction only with the application/software and its DLL. This layer starts the process of turning requests or data packets from the Device Core into PCIe transactions. The TL receives requests (such as "read from memory location hFEDCBA98") or completion packets ("here is the result of a certain read") from the software and it is responsible for turning that request/data into a TL Packet (TLP). A TLP is simply a packet that is sent from the TL of one device to that of the other device. The TLP uses a header to identify the type of transaction that it is (for example, I/O versus memory, read versus write, or request versus completion). It relies on the software to provide valid requests and completion data, and on its DLL to get that information from the TL on the other side of the link [5].

The DLL is the middle PCIe architectural layer. It has direct interaction only with its TL and its PHY. This layer continues the process of turning TLPs from the TL into PCIe transactions. It is responsible for error detection and correction and makes sure that each packet makes it across the link intact. This is done by checking the sequence number at the front of the packet and the Link Cyclic Redundancy Code (LCRC) error checker at the end. The sequence number makes sure that each packet makes it across the link and the LCRC makes sure that each packet makes it across intact. On the transmit side, the DLL accepts incoming packets from the TL and applies the sequence

number and LCRC. Then, the Data Link Layer Packet (DLLP) is submitted to the PHY. On the receive side, the DLL accepts incoming packets from the PHY and checks the sequence number and LCRC. If it is correct, it then passes it up to the receiver side of the TL. If an error occurs (either wrong sequence number or bad data), the packet is not passed on to the TL until the issue has been resolved [5].

The PHY is the lowest PCIe architectural layer. It has direct interaction only with its DLL and its physical PCIe link. This layer finishes the process of turning DLLPs from the DLL into PCIe transactions. It is responsible for actually sending and receiving all the data to be communicated across the PCIe link and contains all the circuitry for the interface operation and some logic functions. On the transmit side, the PHY takes information from the DLL and converts it into the proper serial format. It performs data scrambling, 8b/10b encoding, packet framing , parallel-to-serial conversion, etc. The packet is then sent out across the link using the appropriate transfer rate and allowed link width. On the receive side, the PHY takes the incoming serial stream from the link and turns it back into a parallel data to be passed along to its DLL. This procedure is basically the reverse of what would occur on the transmit side and includes clock extraction, serial-to-parallel conversion, packet framing removal, 8b/10b decoding, data descrambling, etc. [5].

Engineered with backward compatibility in mind, PCIe generations are backward compatible with prior versions, permitting the incorporation of newer devices into systems featuring older PCIe interfaces. However, the data transfer rate will be constrained to the capabilities of the lowest common generation [1].

Supporting a multitude of device types including graphics cards, network adapters, storage controllers, sound cards, and others, PCIe offers a versatile interface for linking a variety of peripherals to a computer system [1].

PCIe has emerged as the standard interconnect for high-speed data transfer in state-of-the-art computer systems, delivering scalability, high data transfer rates, low latency, and compatibility with a diverse selection of devices. Its continuous evolution, marked by each new generation offering enhanced performance and capabilities, ensures it meets the growing demands of bandwidth-intensive applications and fosters the creation of cutting-edge computing systems [1].

1.2 Problem Definition and Objectives

To verify the functionality of the implemented design, represented by the Register Transfer Level (RTL) code, which describes the Logical Sub-Block of the PCIe Gen1 PHY Layer. This involves comparing the behavior of the implemented design against the functionalities and rules required by the specifications and standards. Given the flexibility of the specifications in certain areas, any constraints defined by the design will also be taken into consideration.
The objectives of the projects are as listed:

1. Identify the mandatory functionalities and implementation rules from the specifications and standards.
2. Understand the implemented design and check for its completeness.
3. Compare the implemented design against the specifications and standards.
4. Develop a verification plan.
5. Build complete verification environments on block and chip levels.
6. Achieve a satisfactory code and functional coverage.
7. Report the detected bugs, and solve as many as possible.

1.3 Report Organization

The thesis is organized as follows:
Chapter 2: Literature Review and Standards
This chapter states the specifications and standards used in this project, providing in-depth description of the architecture and operation of the PCIe Gen1 PHY layer. Also, it depicts the goals and trends of Very Large Scale Integration (VLSI) functional verification and discusses - in detail - the verification techniques and methodology used to build the verification environment.
Chapter 3: Project Design and Planning
This chapter provides a comprehensive description of the project's selected RTL design, purpose, constraints, time schedule, and verification approach.
Chapter 4: Block-Level Verification
This chapter provides detailed information about the input and output ports, operation, verification plan, and evaluation of each block in the selected design.
Chapter 5: Chip-Level Verification
This chapter discusses the process and the results of the integration of block-level verification environments into chip-level environments to test the two systems (LTSSM

and Tx) of the design.

Chapter 6: Conclusion and Future Work

This chapter summarizes the work done in the project, and how it can be furtherly extended.

Chapter 2

Literature Review and Standards

2.1 History of PCIe

PCIe Gen1 was first introduced in 2003 as a backward compatible overhaul of PCI. In 2007, Gen2 was released, featuring double the bandwidth and an improved point-to-point communication in the electrical layer. In 2010, Gen3 was released, doubling the bandwidth again by replacing the 8b/10b encoding scheme with the 128b/130b encoding. The novel encoding also handled charge balance on the transmission lines, eliminating the need for scrambling. It was not until 2017 that Gen4 emerged, doubling the bandwidth once more. This generation took a relatively extended period to be released as it required many modifications in the electrical layer of the connections and the fabrication technology was still being developed to adapt with it. Gen5 was released in 2019 to double the bandwidth again and introduce flow control to manage congestion. Additionally, this generation added new equalization techniques to compensate for signal loss at high frequencies [1].
The latest generation of PCIe, Gen6, was introduced in 2021. It introduces several hardware security measures such as Secure Transaction Replay (STR), which enables detection and prevention of unauthorized transaction replays, and Secure Root of Trust (SRoT), providing a foundation for secure boot and execution environments [1].
To double the bandwidth again, Gen6 replaces the conventional Non-Return-to-Zero (NRZ) modulation with the more advanced 4-level Pulse Amplitude Modulation (PAM4), which encodes two bits of data into each transmitted signal using four different voltage levels as shown in figure 2.1. This increases the bandwidth but results in a larger error rate due to the reduced noise margins. Gen6 mitigates this problem by using Forward Error Correction (FEC), which sends redundant bits along the data stream to allow the receiver to detect and correct errors during transmission [6].

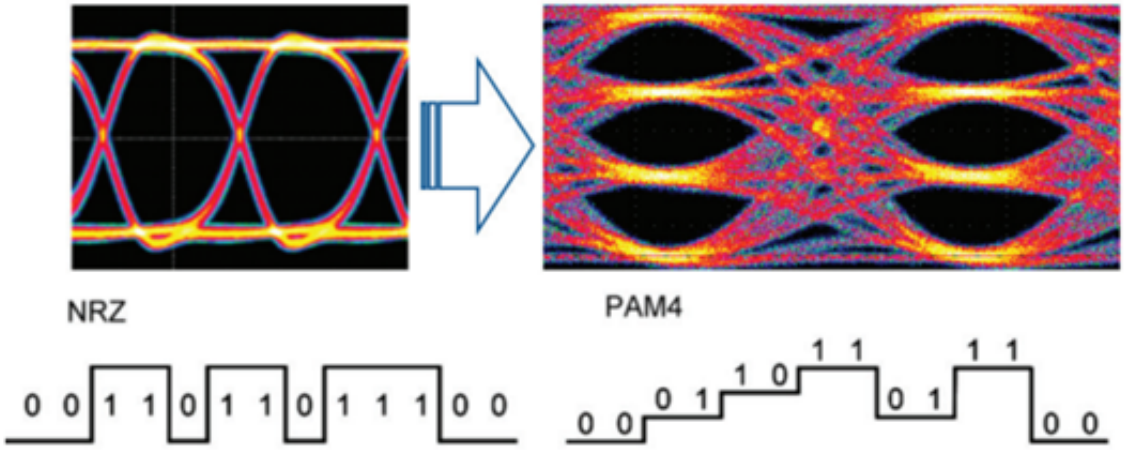


Figure 2.1: PCIe Modulation Evolution: NRZ vs PAM4

2.2 PCI Express Base Specification Revision 1.0

The PCI Express Base Specification Revision 1.0 (PCIeBS), developed by Peripheral Component Interconnect Special Interest Group (PCI-SIG) in 2002, provides a detailed documentation on the operation, architecture, interconnect attributes, and bus management of the PCIe protocol. Any system, device, or peripheral must support the mandatory functionalities and comply with the rules mentioned in the PCIeBS to be considered compliant with it and operate properly. Section 4.2 in the PCIeBS is of particular interest as it focuses on the Logical Sub-Block of the Physical Layer (PHY) of PCIe. This includes details on the Transmitter (Tx), Receiver (Rx), and Link Training and Status State Machine (LTSSM) [7].
This document is used as the main reference for understanding the PCIe protocol. In case discrepancies are detected between the design and the PCIeBS, they will be considered bugs that need to be resolved.

2.2.1 PHY Overview and Function

The PHY is the bottom layer of the architecture of PCIe devices, it has interfaces with DLL and external physical link. It contains all the need digital and analog circuitry to configure and operate the link. On the transmitter side of this layer, outbound packets are framed, encoded, scrambled, clocked, and serialized in each of the available lanes of the link. While on the receiver side, inbound bit streams are recovered into the original packets, which are further forwarded to the DLL. The PHY can be divided into two key sub-blocks: the logical one and the electrical one as shown in figure 2.2 [5].

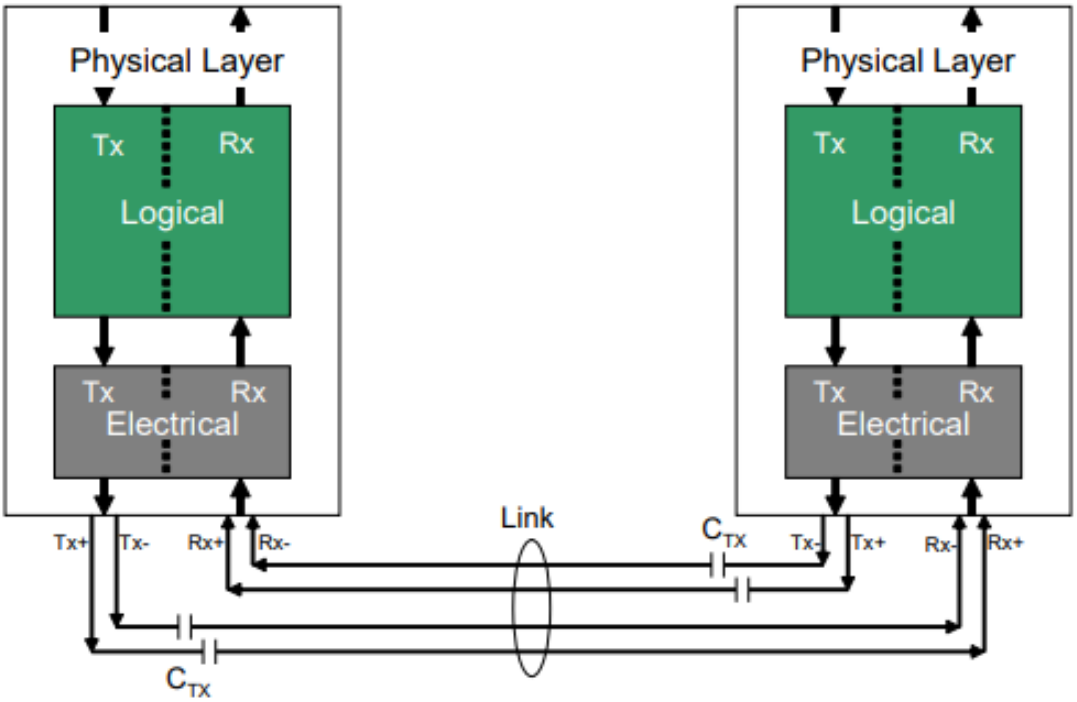


Figure 2.2: Logical and Electrical Sub-Blocks of The PHY

2.2.2 PHY Sub-Blocks

Logical Sub-Block

The operation and the state of the link controls the output characters of the multiplexer, to be byte-striped and sent on available lanes, which are chosen between four different items. The first item is the characters of outbound packets sent from DLL and stored in a buffer to allow flow delays when sending the other items. The second item is the token characters added to indicate the start and the end of each sent packet. The third item contains logical Idle data sent to maintain the physical link while informing the receiver that no actual data is being transmitted, allowing it to optimize its power consumption and avoid processing unnecessary information. Lastly, several types of ordered set packets (explained in the Appendix) are injected for link training and initialization, maintaining link integrity, and facilitating link recovery and debugging processes. Moreover, sent characters can be differentiated based on the type, they can be either data or control characters, which is indicated via the D/K# bit (Data or "Kontrol"). Control characters are used to mark the boundaries of packets, and some special characters of ordered sets [5].

The multiplexing and striping logic should be as fast as possible to allow simultaneous transmission on all lanes. On each lane, characters are first scrambled in a predictable and recoverable pattern. Then, they go through an 8b/10b encoding logic to produce 10-bit symbols that are serialized into a bit stream and forwarded to the electrical sub-

block of the PHY and transmitted to the other end of the link [5].
On the other side, starting from the bottom of the receiver side, serial encoded bit streams are forwarded from the electrical sub-block of the receive side. The first step of Clock and Data Recovery (CDR) is to achieve synchronization with the transmitter side by recovering the clock based on transitions in the incoming bit streams. This enables proper latching of the incoming bits into a deserializing buffer. Then, boundaries of symbols are detected, and the recovered clock is divided by ten to latch the 10-bit Symbols into the Elastic Buffer [5].
The Elastic Buffer's logic modifies the number of Skip (SKP) Symbols when a SKIP Ordered Set (SOS) is detected in order to compensate for small clock differences between the recovered clock and the receiver's local clock. All Symbols are finally moved out of the Elastic Buffer by the Receiver's local clock. Then, symbols are decoded, descrambled, and un-striped to recover the original byte streams, which are filled into the receiver buffer to be delivered to the DLL of the receiver [5].

Transmitter:

Figure 2.3 shows the transmitter architecture. At the top of the diagram, the Tx buffer receives TLPs, DLLPs, and control signals from the DLL. This buffer delays the flow of data at certain times when control characters or ordered sets need to be inserted. In case it is full, the buffer sets the throttle signal to stop the flow of data from the DLL [5].
Below the Tx buffer, a Multiplexer (MUX) is used to select one of four inputs (characters) at a time. Typically, the MUX selects the output of the Tx buffer (incoming packets from the DLL). In this case, the D/K# signal is asserted indicating that characters of type data is selected. The Control Characters are predefined symbols, referred to as Start and End characters, which are added to the start and end of every TLP and DLLP to facilitate the operation of the receiver. The D/K# signal is driven low when the Start and End characters are inserted. Communication across the link requires the initiation and maintenance of link operation, which is accomplished through the regular exchange of Ordered Sets. The details and function of ordered sets are explained in the LTSSM section. In case no packets or ordered sets need to be transmitted, the Logical Idle characters (h00) are sent to maintain the receiver's PLL in a locked state to the transmitter's frequency. The MUX is controlled by the LTSSM and this will be further explored later [5].

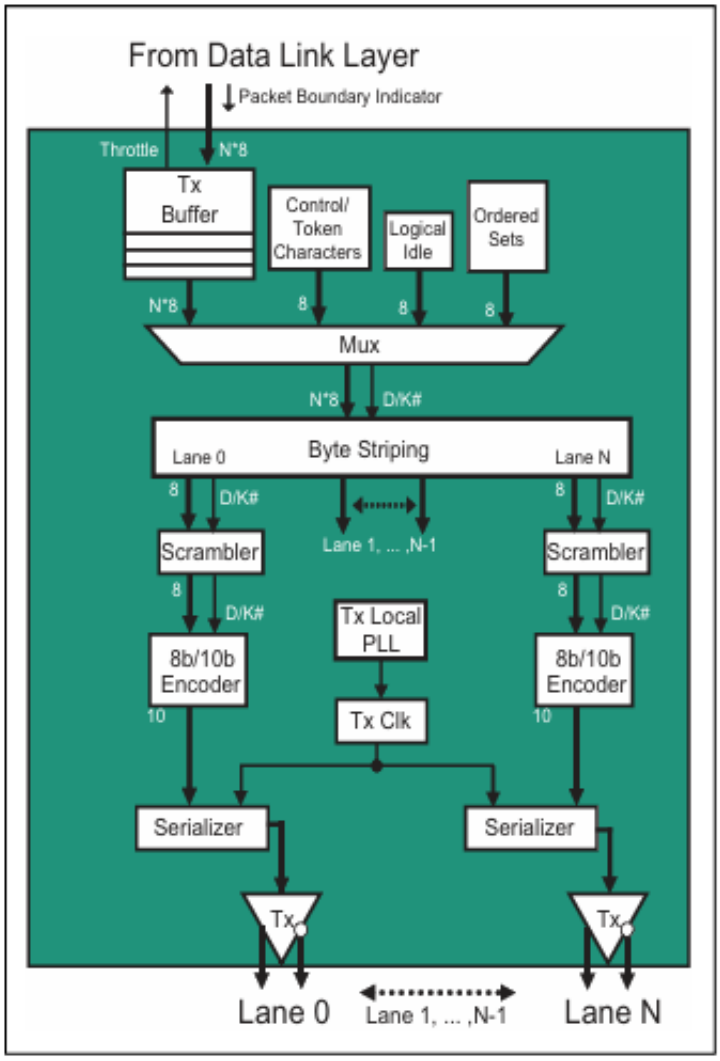


Figure 2.3: PHY Transmit Logic

The output of the MUX is then inserted into Byte Striping logic. Byte striping refers to a method of distributing data across link lanes. It enhances the overall data transfer rate by parallelizing the data across multiple lanes. This involves breaking down the packets into individual bytes and distributing them across the available lanes as shown in figure 2.4 [5].

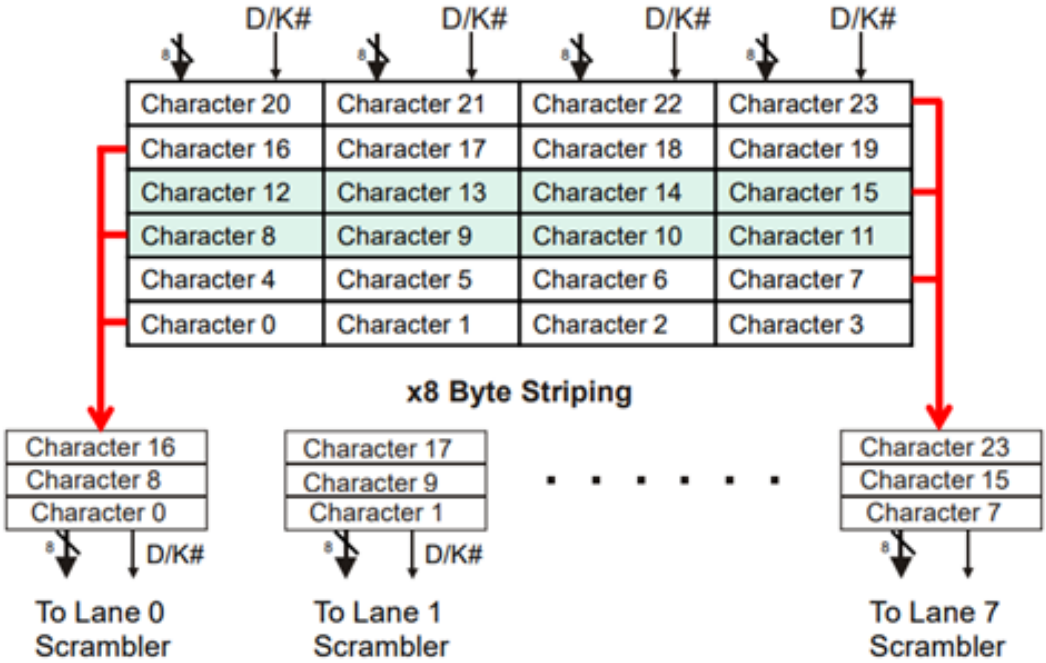


Figure 2.4: Gen 1 Byte Striping

Following striping, the data is scrambled to prevent repetitive patterns (achieve a better balance between 0s and 1s) in the transmitted bitstream. This aims to reduce the possibility of electrical resonances on the link. The scrambler uses a 16-bit Linear Feedback Shift Register (LFSR) to implement the following polynomial

$$G(x) = X^{16} + X^5 + X^4 + X^3 + 1$$

as depicted in figure 2.5 [5].

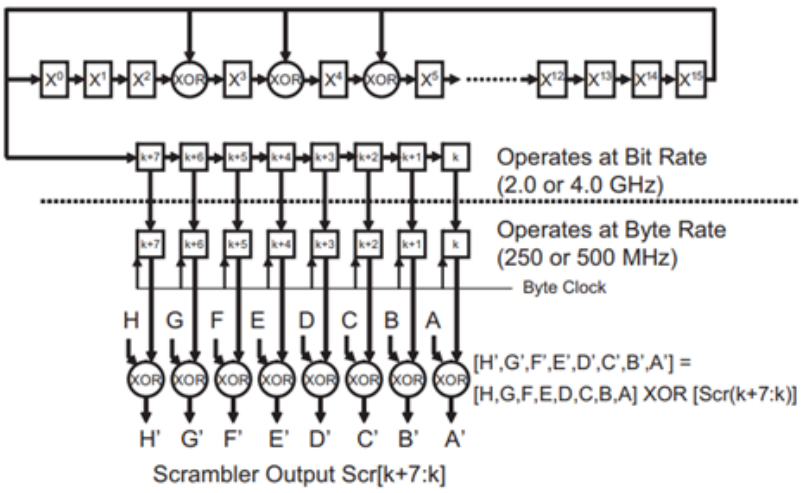


Figure 2.5: Gen1 Scrambler Algorithm

The key step in the transmitter logic is the encoding. PCIe Gen1 uses an 8b/10b encoding scheme. This means that each 8-bit character is translated into 10-bit symbols. Despite the overhead it introduces, encoding is needed for two main reasons. The primary purpose of 8b/10b encoding is to embed a clock signal into the data stream which renders external clock signals unnecessary. The dependency of a group of signals to a single clock source severely reduces setup and hold margins in a particular data transaction. Encoding ensures that the data stream has enough edges in it to recover a clock at the receiver. Since PCIe uses an AC-coupled link by inserting a capacitor for DC blocking, the 8b/10b is used in maintaining DC balance. An example of 8b/10b encoding is shown in figure 2.6.

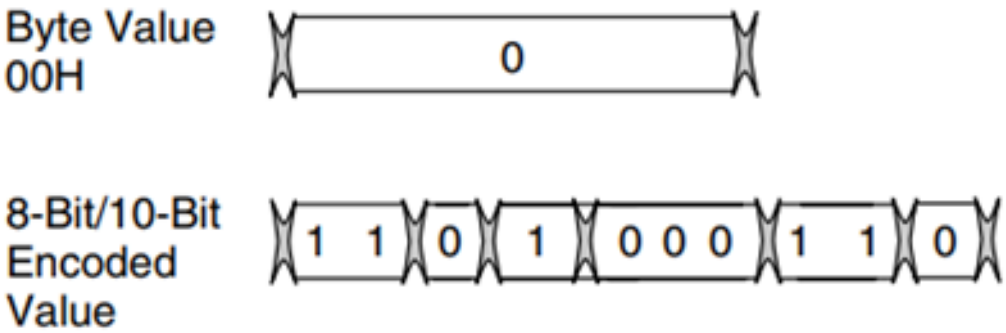


Figure 2.6: Gen1 Encoding Example

The 8b/10b encoder is represented in figure 2.7 below. The encoding is performed based on the input character, the D/K# signal, and the current running disparity (CRD). Disparity is the difference between the number of zeros and ones within each 10-bit data stream (symbol). A symbol with more zeros indicates a negative disparity, whereas a symbol with more ones indicates a positive disparity. The CRD indicates the current state of the link disparity. In order to maintain DC balance, the encoder uses the CRD to generate the next symbol with suitable disparity [5]. The last step is to serialize the data. The symbols are clocked out such that the Least Significant Bit (LSB) is shifted out first and the Most Significant Bit (MSB) last as seen in figure 2.7 [5].

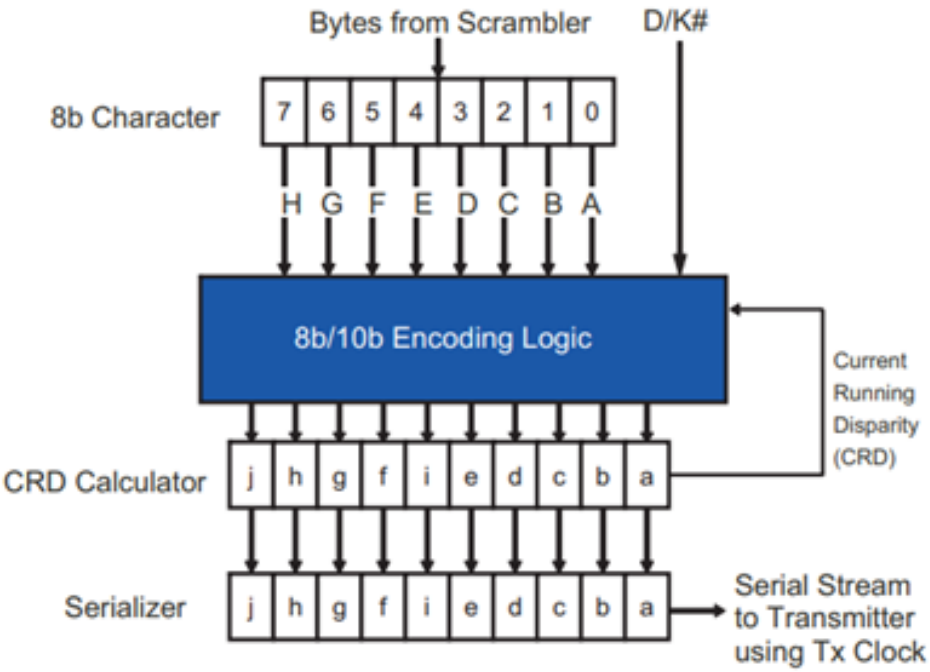


Figure 2.7: Gen1 Encoding and Serialization

Receiver:

Figure 2.8 shows the receiver architecture. At the bottom of the diagram, the Rx lanes receive bits from the data stream. They are first passed through a serial to parallel converter to group each ten bits together. The symbols then go through an elastic buffer that adds or removes extra SKP symbols based on the current clock disparity. Then, each ten bit symbol is converted back into eight bits of data using the 8b/10b decoder. Any errors detected from parity violations are reported back. The data is then passed through a De-Scrambler to reverse the randomization process and retrieve the original data. The incoming bytes from all lanes are then passed through the Un-Stripping logic to group then back together. Any Start/End/Idle/Pad characters are then removed by the PHY before being passed through the Rx buffer to be sent to the DLL [5].

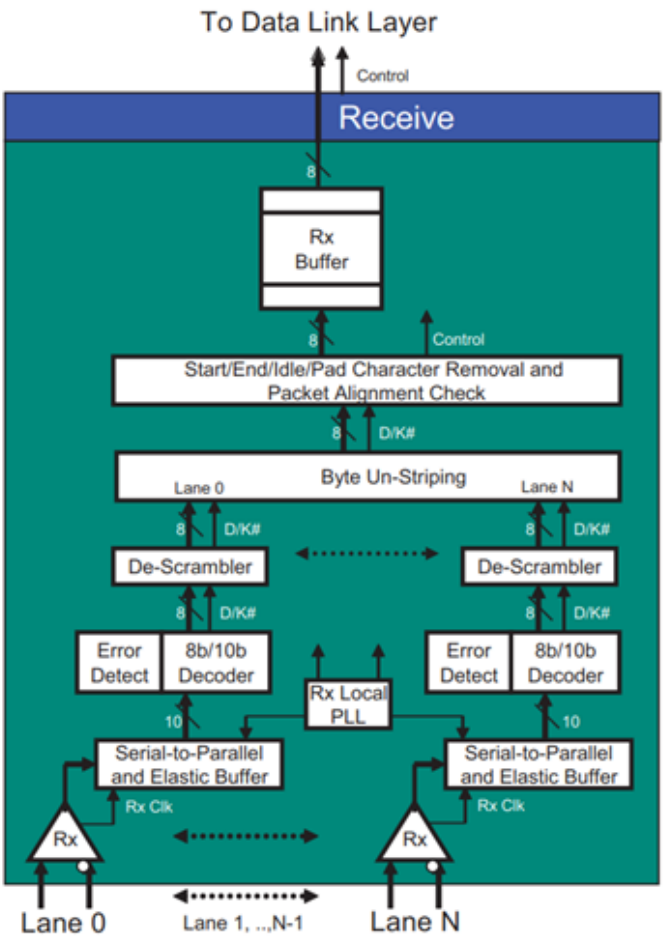


Figure 2.8: PHY Receive Logic

Figure 2.9 shows an in-depth view of the Serial-to-Parallel and Elastic Buffer logic. The incoming bit stream is analyzed by the PLL to infer the clock of the data transmission. This clock rate is used to control the sampling rate of symbols and the uptake of symbols into the Elastic buffer. The symbols are also passed into a Comma (COM) signal detection logic to be able to detect any incoming SKP symbols. The data is then clocked into the elastic buffer using the transmission clock rate. The rest of the receiver logic then sample symbols from the elastic buffer using the receiver's own clock rate. To compensate for any discrepancy between the two clocks, the elastic buffer adds or removes extra SKP symbols to balance the two data rates. The data is then passed into the lane De-Skew circuit to delay some bytes in order to match the delay of the slowest lane [5].

The symbols are then passed into the 8b/10b decoder to reverse the encoding process. The decoder consists of a simple look-up table that checks the corresponding 8b byte from each 10b symbol. The decoder also checks whether the byte was a data (D) or control (K) signal and reports that to the next logic. The decoder also checks for any parity violations during the transmission and reports any detected errors [5].

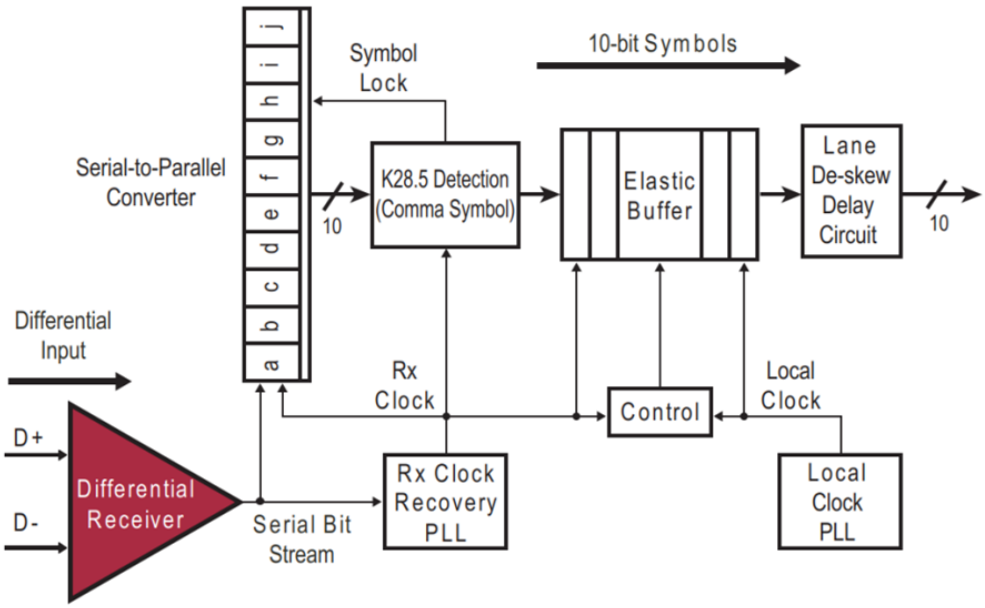


Figure 2.9: Gen1 Receiver Logic Front-End

The data bits are then descrambled to reverse the randomization process and passed through the de-stripping logic to assemble the data from all the lanes together as is shown in figure 2.10 [5].

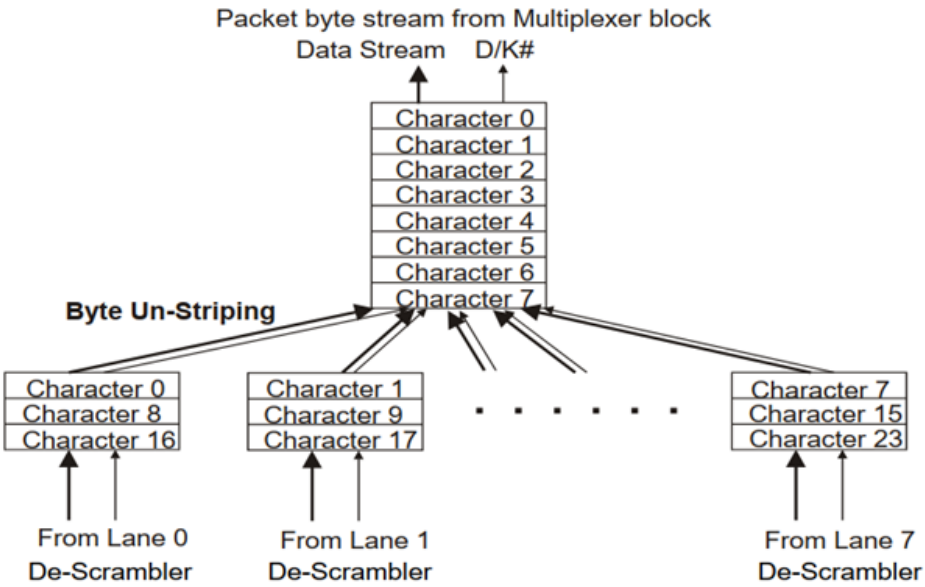


Figure 2.10: Gen1 Byte Un-Striping

LTSSM:

Link initialization and training is a hardware-based process controlled by the PHY. The process configures and initializes a device's link and port so that normal packet traffic proceeds on the link. The full training process is automatically initiated by hardware after a reset and is managed by the LTSSM. At the heart of the PHY, the LTSSM stands out as the driving force responsible for initiating and upholding PCIe connections. It guides through the complex PCIe environment, guaranteeing the dependable and secure transfer of data by setting up the interface and managing classification complexity. The following conditions are achieved throughout the process of link training [5]:

- Bit Lock: when Link training begins, the Receiver's clock is not yet synchronized with the transmit clock of the incoming signal and is unable to reliably sample incoming bits. During Link training, the Receiver CDR logic recreates the Transmitter's clock by using the incoming bit stream as a clock reference. Once the clock has been recovered from the stream, the Receiver is said to have acquired Bit Lock and is then able to sample the incoming bits.

- Symbol Lock: this is a similar problem in that the receiver can now see individual bits but does not know where the boundaries of the 10-bit Symbols are found. As TS1s and TS2s are exchanged, Receivers search for a recognizable pattern in the bit stream. A simple one to use for this is the COM Symbol. Its unique encoding makes it easy to recognize and its arrival shows the boundary of both the Symbol and the Ordered Set since a TS1 or TS2 must be in progress.

- Link Width: The PHY of both devices tests the Link and sets the width to the highest common value of available lane in the two devices.

- Lane Reversal (optional): The Lanes on a multi-Lane device's port are numbered sequentially starting with Lane 0. Normally, Lane 0 of one device's port connects to Lane 0 of the neighbor's port, Lane 1 to Lane 1, and so on. However, sometimes it is desirable to be able to logically reverse the Lane numbers to simplify routing and allow the Lanes to be wired directly without having to crisscross. The situation is detected during Link training and one device must internally reverse its Lane numbering. Nonetheless, implementing lane reversal feature adds complexity to the design and manufacturing of a device. The additional circuitry needed for lane reversal can also increase power consumption. That is why it is an optional feature in PCIe devices to allow manufacturers to concentrate on core functionality and optimize their devices for specific use cases, without burdening them with unnecessary features in simpler applications.

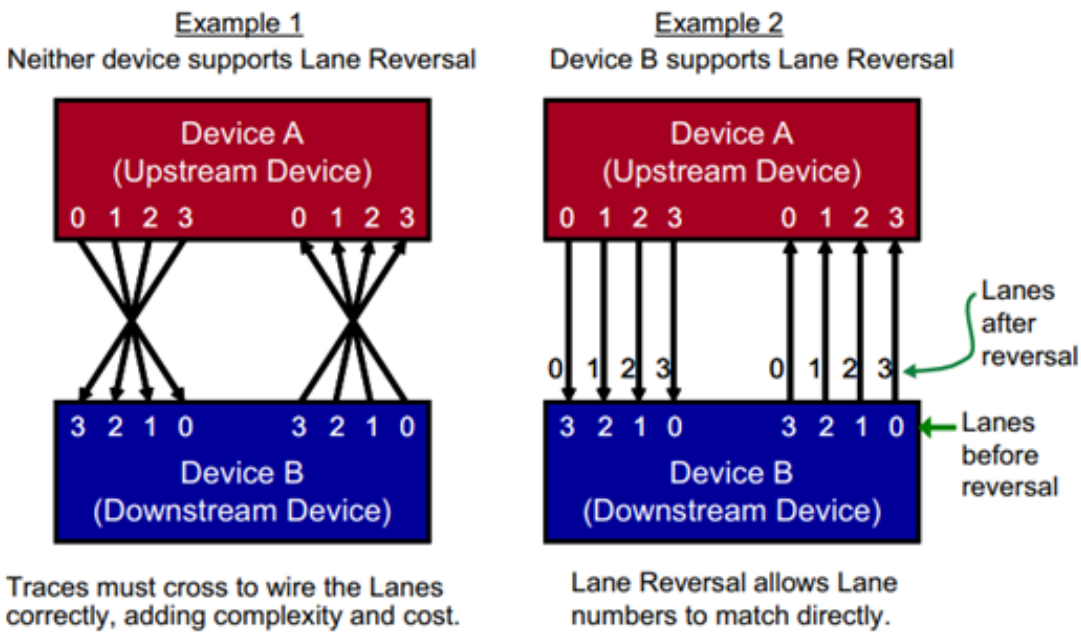


Figure 2.11: Lane Reversal

- Polarity Inversion: the D+ and D- differential pair terminals for two devices may also be reversed as needed to make board layout and routing easier. Unlike Lane reversal, support for this feature is mandatory. That is due to the fact that it only deals with reversing the electrical signal itself, not the physical location of the lanes. This is a simpler technical challenge and requires less complex circuitry compared to lane reversal.

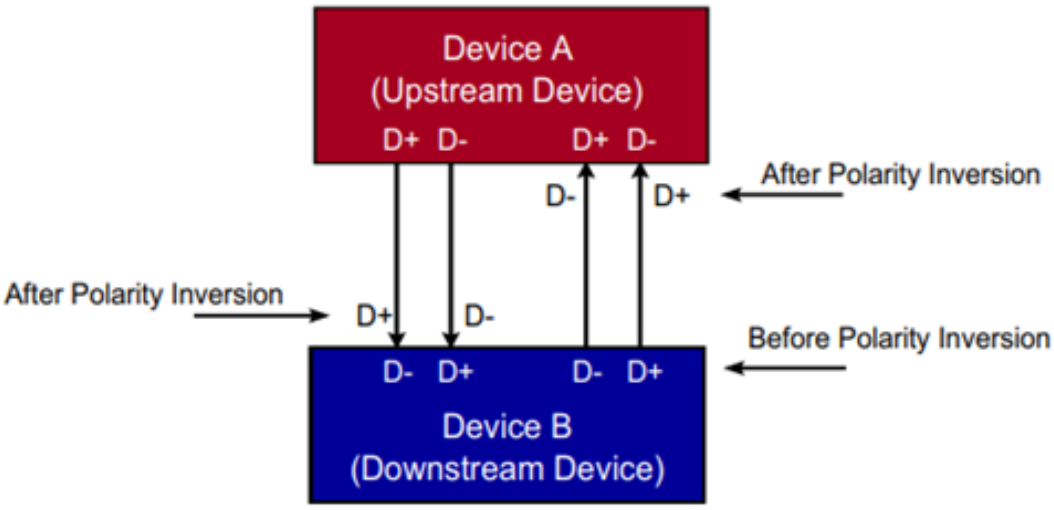


Figure 2.12: Polarity Inversion

- Link Data Rate: After a reset, Link initialization and training will always use the default 2.5 Gb/s data rate for backward compatibility. If higher data rates

are available, they are advertised during this process and, when the training is completed, devices will automatically go through a quick re-training to change to the highest commonly supported rate.

- Lane-to-Lane De-skew: Trace length variations and other factors cause the parallel bit streams of a multi-Lane Link to arrive at the receivers at different times, a problem referred to as signal skew. De-skew logic at receivers is used to detect the arrival of SOSs and will align across the lanes by adding and subtracting one or more SKP symbols from the stream to keep the lanes aligned to within tight constraints to minimize the amount of compensation buffering required.

LTSSM progresses through a sequence of states, each consists of a number of substate, organized into five primary categories [5]:

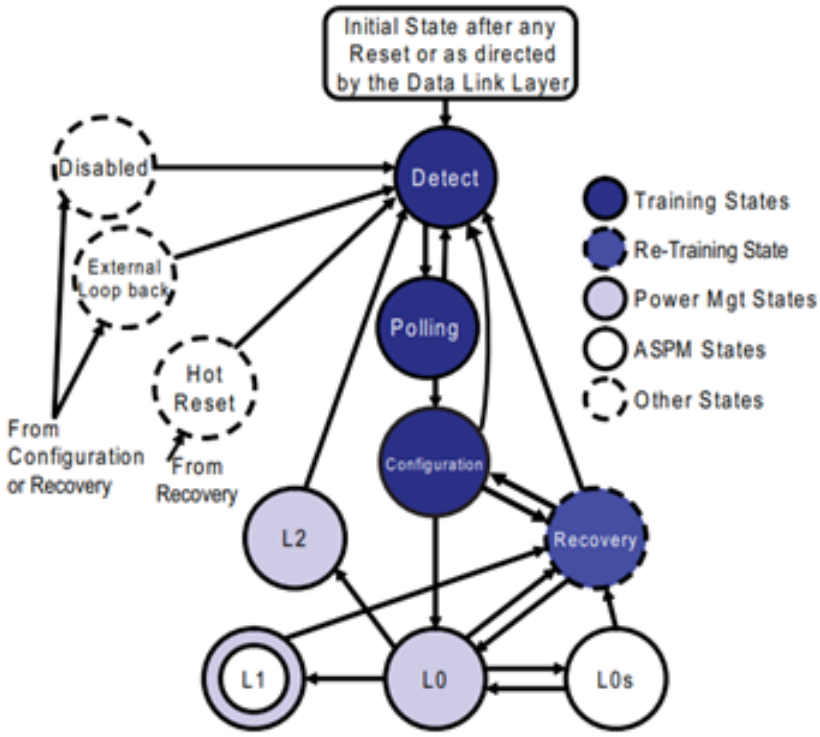


Figure 2.13: LTSSM States and Transitions

- Training States: This group includes the Detect, Polling, Configuration, and L0 states, which initialize and configure the interface initially.
- Re-training State: Here, situations requiring retraining, like speed upgrades, Link Retrain, and Link Width adjustments, are managed.
- Power Management (PM) States: Consisting of L1 and L2 states, this phase aims to optimize power consumption while ensuring connectivity.

- Active State Power Management (ASPM): The L0s state focuses on optimizing power usage while maintaining link availability.
- Other States: This category involves Loopback, Hot Reset, and Disable states, each fulfilling specific functions within the LTSSM framework.

The normal flow of link training is Detect → Polling → Configuration → L0. The Link Re-training, also called Recovery, state is entered for a variety of reasons, such as changing back from a low-power Link state, like L1, or changing the Link bandwidth (through speed or width changes). Power management software may also place a device into a low-power device state and that will force the Link into a lower Power Management Link state (L1 or L2). If there are no packets to send for a time, ASPM hardware may be allowed to automatically transition the Link into low power ASPM states (L0s or ASPM L1). In addition, software can direct a Link to enter some other special states: Disabled, Loopback, or Hot Reset [5].

LTSSM States Details:

In every state, both devices exchange Training Sequences (TS1 and TS2) until a specific number or time condition is met, signaling a transition to the next state. The LTSSM process begins in the Detect state, where the system assesses the existence of a receiver at the link's end by transmitting electrical signals [5].

1. Detect: Has two substates:
    - Detect.Quiet: Finishes after a 12 ms timeout or when any Lane exits Electrical Idle.
    - Detect.Active: The Transmitter tests whether a Receiver is connected on each Lane by setting a DC common mode voltage of any value in the legal range and then changing it. The detection logic observes the rate of change, which is much longer if a Receiver is attached. The motivation for this unusual approach in PCIe is to provide an automatic hardware assist in a test environment. If the proper load is detected, but the Link partner refuses to send TS1s and participates in Link Training, the component will assume that it must be in a test environment and will begin sending the Compliance Pattern to facilitate testing. The same Detect logic must be included in every design regardless of its intended operating speed, as Detect is only used for the 2.5 GT/s rate.
2. Polling: The primary purpose of this state is for the two devices to understand what each other is saying. To do so, both ends start transmitting TS ordered sets and wait to receive a certain number of ordered sets from the other link. Polarity inversion, bit lock, and symbol lock are done in this state. It has two substates:

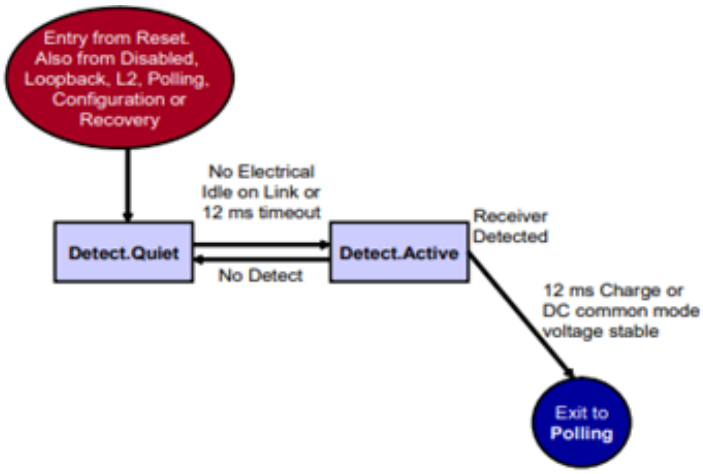


Figure 2.14: Detect State Machine

- Polling.Active: Transmitters send a minimum of 1024 consecutive TS1s (#Lane = #Link = PAD), on all detected Lanes (takes about 64µs). Receivers use the incoming TS1s to acquire bit Lock and symbol lock. The next state is Polling.Configuration if, after sending at least 1024 TS1s all detected Lanes receive eight consecutive training sequences.
- Polling.Configuration: Transmitters start sending TS2 to advertise to the link partner that this device is ready to proceed to the next state in the state machine. Each Lane's receiver must independently invert the polarity of its differential input pair if necessary. To do this, the Receiver looks at Symbols 6 to 15 of the incoming TS1s or TS2s. If a D21.5 is received instead of a D10.2 in a TS1, or a D26.5 instead of the D5.2 expected for a TS2, then the polarity of that lane is inverted and must be corrected. The substate ends after transmitting at least 16 TS2s (after receiving one) and receiving at least eight to move to Configuration State. Otherwise, returns to Detect State after 48 ms timeout.

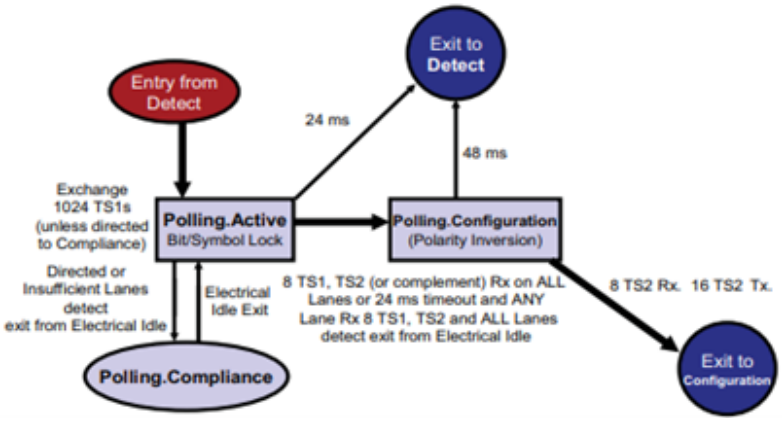


Figure 2.15: Polling State Machine

3. Configuration: it includes assigning a link number (or numbers if link split between different devices) and the lane numbers, with lane reversal if supported. In addition, scrambling may be disabled. De-skewing will be completed by the end of this state and the link will now be 'up' and the state enters L0, the normal link-up state (assuming not directed to Loopback or Disable states). It has six substates:

   - In Config.LinkWidth.Start state, configuration of the link and assigning link numbers is done in this state. The upstream link (such as the endpoint) resumes transmitting TS1s with both the link and lane set to PAD. Simultaneously, the downstream link (from the root complex) begins transmitting TS1s with a specified link number, while setting the lane number to PAD. Upon receiving at least two TS1s with a link number, the upstream link replies with the TS1 bearing that link value and transitions to Config.LinkWidth.Accept.
   - In Config.LinkWidth.Accept state, the downstream link will begin transmitting TS1s with designated lane numbers, while initially, the upstream link will maintain its TS1 transmissions with the lanes set to PAD. However, it will promptly adjust its TS1 transmissions to match the lane numbers received (or possibly reversed), subsequently transitioning to Config.Lanenum .Wait.
   - Upon receiving TS1s with non-PAD lanes, the downstream link enters this state. Its purpose is to provide a window of up to 1ms to address errors or de-skew issues that may lead to inaccurate link width configurations. When two consecutive TS1s are detected, the downstream initiates transmission of TS2s, prompting response from the upstream lanes upon receiving two consecutive TS2s. This marks the transition to the Config.Complete state, where the downstream proceeds to Config.Idle after receiving and transmitting eight consecutive TS2s. Subsequently, the lanes commence sending Idle (IDL) symbols, progressing to state L0 (LinkUp=1) after receiving eight IDL symbols and transmitting at least sixteen following the initial IDL detection.

4. L0 (normal operation): this is the normal, fully active state of a Link during which TLPs, DLLPs and Ordered Sets can be exchanged. The PHY also notifies the upper layers that the Link is ready for operation, by setting the LinkUp = 1.

The remaining states are auxiliary and were not implemented in the design.

Detect.Quiet	LinkUp=0 Wait for Electrical Idle Broken at Rx
Detect.Active	Perform Receiver Detect
Polling.Active	Transmit TS1/PAD/PAD Wait for Rx of 8 TSx/PAD/PAD or inverse, and transmission of 1024 TS1s
Polling.Config	Invert polarity where necessary Transmit TS2/PAD/PAD Wait for Rx of 8 TS2/PAD/PAD and transmitted 16 TS2s after first Rx TS2
Config.LinkWidth.Start	Transmit TS1/PAD/PAD Wait for Rx of 2 TS1/Link#/PAD Transmit TS1/Link#/PAD
Config.LinkWidth.Accept	Wait for Rx of 2 TS1/Link#/Lane# Tx TS1/Link#/Lane#
Config.Lanenum.Wait	Wait for Rx of 2 TS2/-/-
Config.Lanenum.Accept	Wait for Rx of 2 TS2/Link#/Lane#
Config.Complete	Tx TS2/Link#/Lane# Wait for Rx of 8 TS2/Link#/Lane# and Transmitted 16 TS2s after first Rx TS2 Note N_FTS. Disable scrambling if necessary. Deskewing must be complete.
Config.Idle	Tx Idle data (I.e. transmit nothing) Wait for Rx of 8 idle data and transmission of 16 Idles. LinkUp=1
L0	LinkUp, start flow control initialisation

Figure 2.16: Summary of Substates in The Normal Flow of Link Training Process

Electrical Sub-Block

The electrical sub-block in each lane is the lowest part in the architecture of the PHY. It implements the following processes [5]:

- Serial/Parallel Conversion: it involves the conversion of encoded/packetized data from a logical sub-block into a serial format within the transmit buffer of the electrical sub-block. After serialization, the data is directed across the link, and upon reception, the receivers deserialize the data and pass it back to the logical sub-block for additional processing.

- Differential Signaling: the electrical sub-block contains differential drivers and receivers which convert (back and forth) the bit stream into two single-ended electrical signals with opposite polarity, to improve noise immunity and reduce voltage level. The drivers apply the differential voltage linked to a logical 1 and a logical 0 while keeping the proper DC common mode voltage when the Link is in the L0 normal operation state. This voltage is perceived by receivers as the input stream; however, if it falls below a threshold, it is interpreted as the Electrical Idle Link condition. Since both differential signals are on nearby pins and have extremely close traces, any electrical noise that affects one will also influence the other. As both signals are affected, as figure 2.17 illustrates, their difference does not alter significantly and is thus invisible to the receiver.

- AC Coupling: the link connecting two PCIe devices is AC-coupled by adding a capacitor in-line. This serves to decouple the DC part of the signal between the Link partners and means they do not have to share a common power supply or ground return path, as when the devices are connected over a cable.

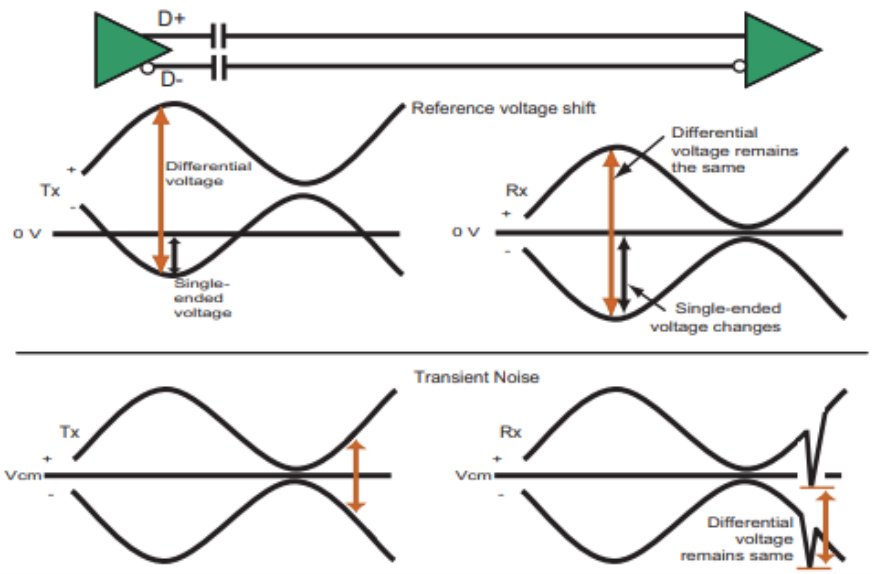


Figure 2.17: Differential Common-Mode Noise Rejection

- De-emphasis: to reduce the effects of signal distortion along the link transmission line, PCIe 1.0 transmitters use a simple equalization method called de-emphasis. The problem is that high frequency signaling has, by definition, a small bit time making it nearly inevitable to have the value in one bit time affect the value in another bit time, which is referred to as Inter-Symbol Interference (ISI). In a bit stream, de-emphasis lowers the voltage for repeated bits. Lowering the transmitter voltage in these situations can significantly enhance signal quality, despite initially seeming counterintuitive because it lessens the signal swing and the amount of energy that reaches the receiver. The de-emphasized voltage is reduced by 3.5 dB from normal for 2.5 GT/s, which translates to about a one-third reduction in voltage.

- PLL Circuit: it generates a clock that can be used as the internal timing reference for a PCI Express device. Each device typically receives a 100-megahertz differential pair clock. This clock signal can be input into a PLL circuit, which increases its frequency twenty-fivefold to achieve a 2.5 Giga Hertz (GHz) output.

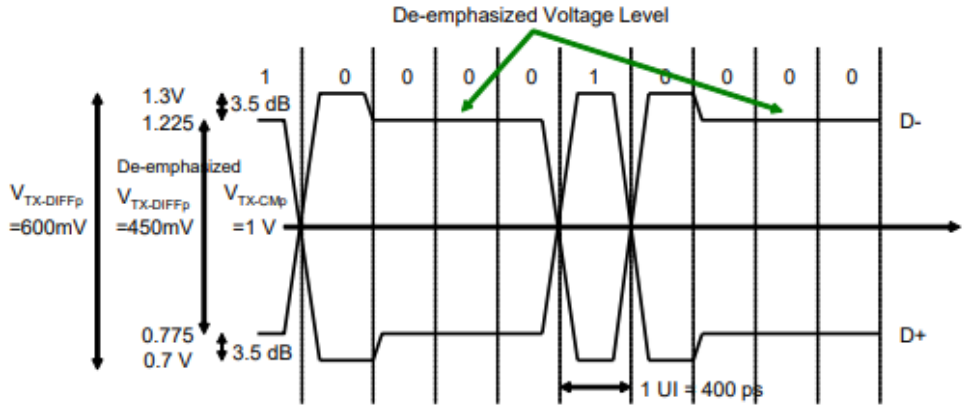


Figure 2.18: Transmission with De-Emphasis

2.3 Other Standards

2.3.1 PHY Interface for the PCI Express 3.0 Architecture Revision .9

The PHY Interface for the PCI Express 3.0 Architecture Revision .9 (PIPE) specification was developed by Intel in 2010. As shown in figure 2.19, the PIPE standard divides the Logical Sub-Block into Media Access Layer (MAC) and Physical Coding Sublayer (PCS) with a PHY/MAC interface linking them. The Physical Sub-Block is referred to as the Physical Media Attachment Layer (PMA). PIPE defines the interface and communication between the MAC and PCS of the PHY's Logical Sub-Block [8].
The PHY/MAC interface defines the data and logical signals between the MAC and PCS. The PIPE specification builds on the PCIeBS and allows for various configurations that support different signaling rates [8].

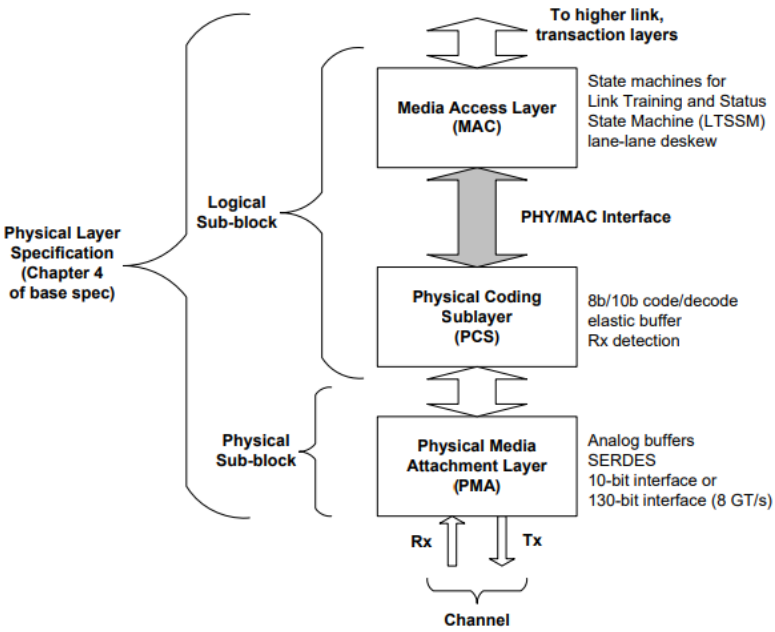


Figure 2.19: PHY Specification according to The PIPE Standard

2.3.2 IEEE Standard for SystemVerilog— Unified Hardware Design, Specification, and Verification Language (IEEE Std 1800 ™ - 2017)

The IEEE Standard for SystemVerilog— Unified Hardware Design, Specification, and Verification Language (IEEE-SV) is developed and maintained by IEEE to meet the increasing needs for a language that tackles the growing complexity of modern hardware design, specification, and verification [9].
The standard includes support for various hardware description levels such as behavioral, RTL, and gate-level, along with additional hardware verification constructs— testbench, coverage, assertion, object-oriented programming, and constrained random constructs. Furthermore, it offers Application Programming Interfaces (APIs) enabling interaction with foreign programming languages [9].
The wide adoption of SystemVerilog can be clearly seen in figure 2.20, which shows the ASIC verification language adoption for 2014, 2018, and 2022 according to The 2022 Wilson Research Group Functional Verification Study [10]. For years, SystemVerilog has been firmly established as the top preference in hardware verification for industry projects, rendering it the most suitable option for this particular project.

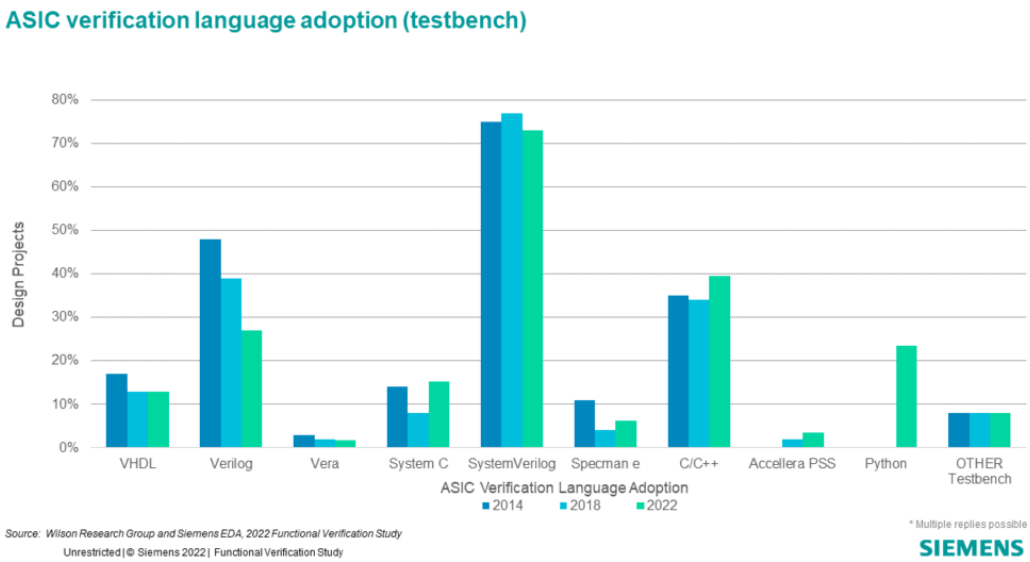


Figure 2.20: ASIC Verification Language Adoption

2.3.3 IEEE Standard for Universal Verification Methodology Language Reference Manual (IEEE Std 1800.2 ™ - 2020)

The IEEE Standard for Universal Verification Methodology Language Reference Manual (IEEE-UVM) is overseen and updated by IEEE. The IEEE-UVM establishes a series of APIs that define a Base Class Library (BCL). This BCL is instrumental in developing

modular, scalable, and reusable components for functional verification environments. Standardization of the Universal Verification Methodology (UVM) aims to enhance interoperability, diminish the need for repurchasing and rewriting of Intellectual Property (IP) for every project or Electronic Design Automation (EDA) tool, and simplify the reuse of verification components [11].
UVM is used to develop an environment that encapsulates the Design Under Test (DUT). This environment creates and manages transactions, defines and drives both randomized and directed inputs, monitors outputs, records functional coverage, and verifies the design against a reference model. The created testbench is built with levels of abstraction that allow for easier and faster development and debugging [11].
According to The 2022 Wilson Research Group Functional Verification Study, UVM is predominant ASIC verification methodology, as evident in figure 2.21 [11]. The UVM APIs and BCL are based on IEEE-SV, thereby ensuring compatibility between the two and solidifying their status as the leading standards in hardware verification [10].

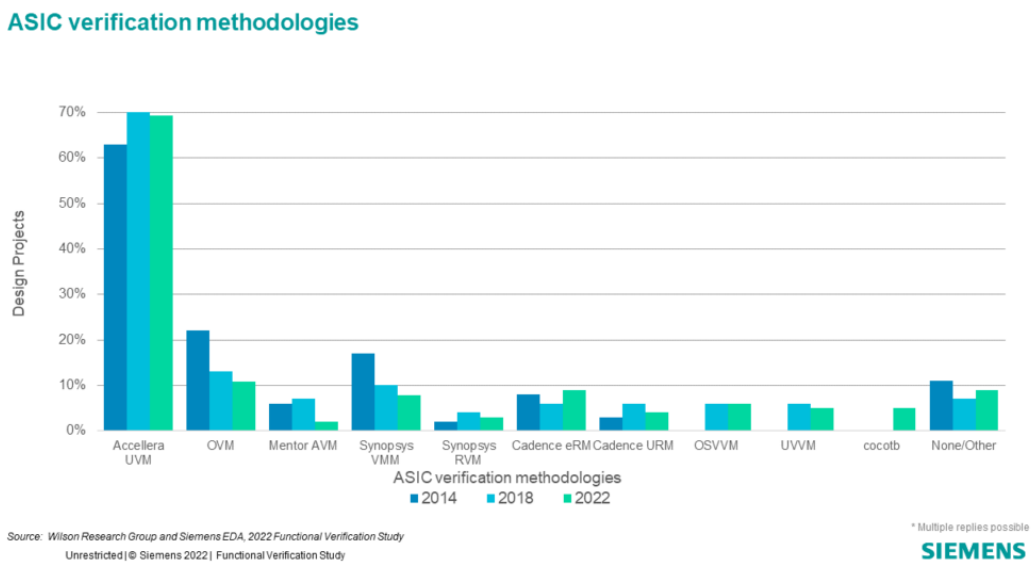


Figure 2.21: ASIC Verification Methodologies Adoption

2.4 VLSI Verification

2.4.1 Challenges and Trends

Silicon chip technology is the driving force behind the innovations that shape our modern world. From portable devices seamlessly blending cell phone, calculator, and digital camera functionalities to high-performance servers facilitating online transactions, these chips are ever-present in our daily lives. As technology advances, chip complex-

ity increases exponentially, presenting both opportunities and challenges for engineers [12].

The expectations placed on these silicon chips are continually on the rise, demanding flawless performance despite escalating complexities. Whether embedded in a microprocessor, memory device, or a comprehensive system-on-a-chip (SoC), each chip undergoes a rigorous design process to overcome multifaceted challenges. The design flow, as depicted in figure 2.22, starts with customer requirements including functions, chip size, power consumption, and processing speed [12].

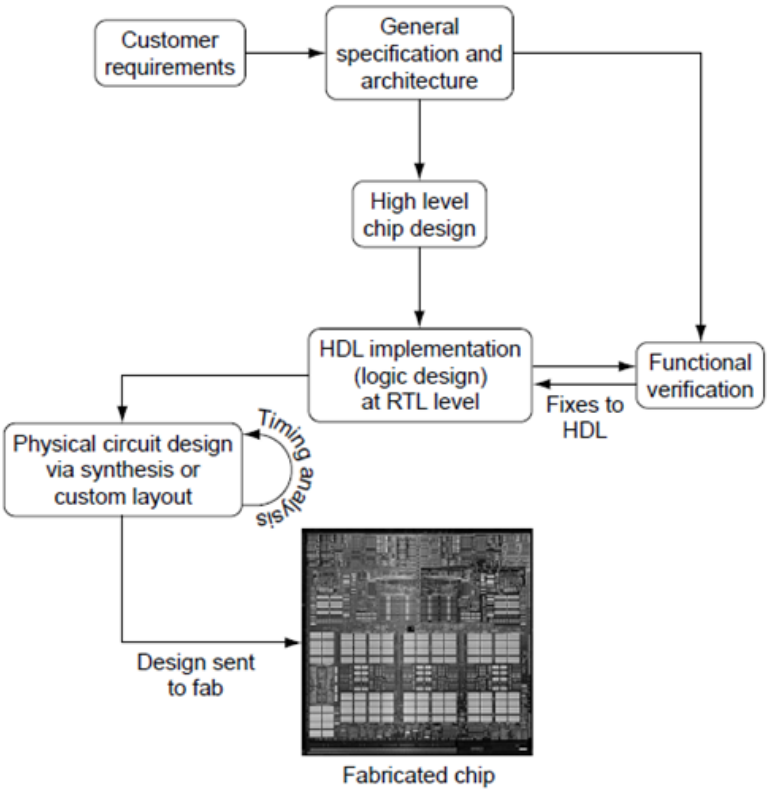


Figure 2.22: VLSI Digital Design Flow

As engineers progress through the chip design process, they tackle the delicate task of balancing conflicting goals and tradeoffs. Elevated chip performance may necessitate faster cycle times, invariably increasing power consumption. Conversely, faster cycle times could mandate the inclusion of more logic stages, amplifying chip complexity and area. EDA tools play a pivotal role by aiding engineers in predicting chip characteristics, including area, timing, and power consumption [12].

However, two critical questions persist throughout this intricate process: What if the initial Hardware Description Language (HDL) did not accurately express the desired function? What if the designer overlooked a critical corner condition? The answers lie in the domain of functional verification [12].

Functional verification is the cornerstone ensuring that a design aligns with the in-

tended tasks outlined by the system architecture. Unlike circuit-level validation, verification engineers operate at the RTL, utilizing RTL languages (HDLs) such as Verilog and VHDL [12].
While modern chip designs offer remarkable capabilities, they also present daunting challenges for verification engineers. Imagine scrutinizing hundreds of thousands of code lines, hunting for elusive flaws that could lead to catastrophic failures. This section delves into the major roadblocks facing verification engineers.
Today, in the era of multi-million gate ASICs and Field Programmable Gate Arrays (FPGAs), reusable IPs, and system-on-a-chip (SoC) designs, verification consumes about 70% of the design effort. Design teams, properly staffed to address the verification challenge, include engineers dedicated to verification. The number of verification engineers can be up to twice the number of RTL designers, highlighting the immense effort required to navigate this challenge [13].
Another challenge is detecting the violations in the design implementation. With all the possible states and transitions, the Digital Verification (DV) engineer must be able to decide whether the design is acting correctly or not. The solution to this challenge is validating the logic at the transaction level so that the focus is shifted to the behavior of the design. Briefly, the verification challenges come down to driving the input and state transitions scenarios and flagging the incorrect behavior of the design implementation. To address these challenges, digital verification engineers use plenty of tools and technologies that increase the level of abstraction and automation and enable parallelism [12].
Traditional directed testing has served as a cornerstone of verification for many years. By focusing on specific design features and meticulously crafting targeted test vectors, engineers can systematically verify individual functionalities. This approach offers several advantages [14]:

- Incremental Progress: Directed testing provides a clear path towards verification completion, allowing managers to track progress and monitor project milestones such as shown in figure 2.23.
- Immediate Results: With minimal infrastructure, directed testing quickly yields tangible results, providing early insights into design functionality.
- Resource-Friendly: For designs of moderate complexity, directed testing can be efficiently implemented using existing resources.

However, directed testing also faces limitations [14]:

- Scalability Challenges: As design complexity increases, the number of necessary test cases explodes, requiring exponentially more time and resources to achieve

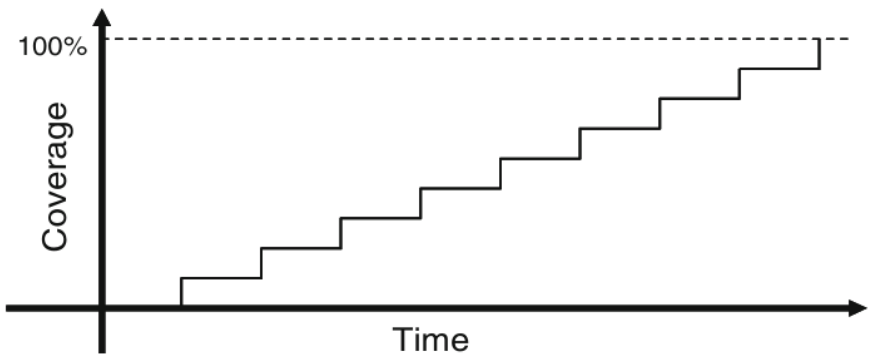


Figure 2.23: Directed Testing Progress (Coverage) Over Time

comprehensive coverage. Consider a seemingly simple design like a 32-bit adder. To exhaustively test every possible combination of inputs and outputs, simulations would run for years, making this approach impractical for more complex designs.

- Limited Scope: Directed tests focus on predetermined scenarios, potentially leaving hidden bugs and corner cases lurking outside the targeted areas. Imagine focusing on testing only specific corner cases of addition (e.g., adding large positive numbers), neglecting to test overflow possibilities or negative number interactions. This could lead to undetected errors in real-world scenarios.

- Time Constraints: In tight deadlines, relying solely on directed testing may not be feasible to achieve desired coverage levels within project timelines.

Therefore, while directed testing remains a valuable tool in the verification toolbox, it is crucial to recognize its limitations and explore complementary approaches to ensure efficient and comprehensive verification, especially for complex designs facing time constraints.

In the following sections, we will delve into other verification trends like randomization and constraints, exploring how they address the limitations of directed testing and contribute to a robust verification strategy.

Random stimulus plays a pivotal role in the verification of complex designs. While directed tests target expected bugs within a design, random tests have the potential to uncover unforeseen issues. When employing random stimuli, functional coverage becomes essential for measuring verification progress. Additionally, automated result prediction, typically through a scoreboard or reference model, becomes necessary when using automatically generated stimuli. Establishing the testbench infrastructure, inclusive of self-prediction mechanisms, entails a significant amount of effort. A layered testbench architecture aids in managing complexity by breaking down the problem into manageable components [14].

With proper planning, a shared testbench infrastructure can be developed, reducing the need for continual modification. This infrastructure should include "hooks" allowing tests to perform specific actions such as shaping stimulus and injecting disturbances. Conversely, code specific to a single test should remain separate from the testbench to prevent unnecessary complication of the infrastructure [14].
Constructing this style of testbench demands more time compared to traditional directed testbenches, especially for the self-checking components. Consequently, there may be a significant delay before the first test can be executed. This delay underscores the importance of incorporating this effort into the project schedule. Figure 2.24 illustrates the initial delay before the first random test runs [14].

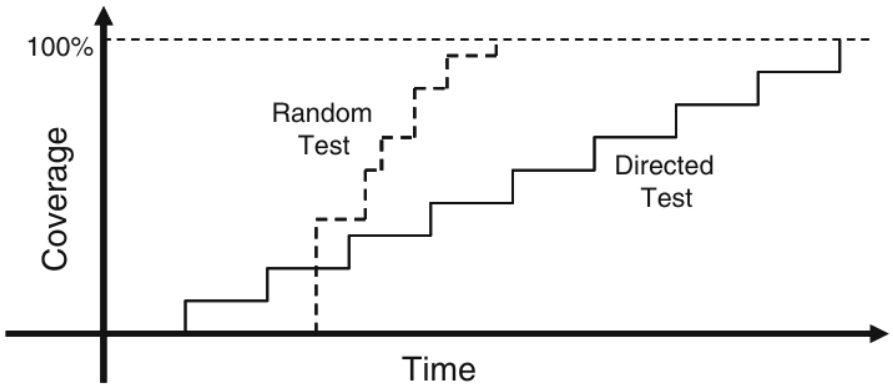


Figure 2.24: Random vs Directed Testing Progress (Coverage) Over Time

While the upfront investment in creating a constrained-random testbench may seem daunting, the rewards are substantial. Unlike directed tests, where each is written from scratch, every random test shares a common testbench. Each random test comprises only a few dozen lines of code to constrain the stimulus in a specific direction and induce desired exceptions, such as creating a protocol violation. Consequently, a single constrained-random testbench is capable of identifying bugs faster than many directed testbenches [14].
As the rate of bug discovery begins to diminish, new random constraints can be introduced to explore new areas. While the last few bugs may only be found with directed tests, the vast majority are uncovered by random tests. While a random testbench can be constrained to create directed tests, the reverse is not true for a directed testbench [14].
Though the simulator generates the stimulus, completely random values are undesirable. SystemVerilog language describes the stimulus format, specifying constraints that the simulator adheres to when selecting values. These values are then sent into the design and a high-level model that predicts the expected output [14].
Finally, figure 2.25 outlines the paths to achieve complete coverage. Starting at the

upper left with basic constrained-random tests, they are run with various seeds. Functional coverage reports identify gaps in coverage, prompting minimal code changes to address these gaps, such as introducing new constraints or injecting errors or delays into the DUT. The majority of the effort is spent in this outer loop, with directed tests written only for features unlikely to be reached by random tests [14].

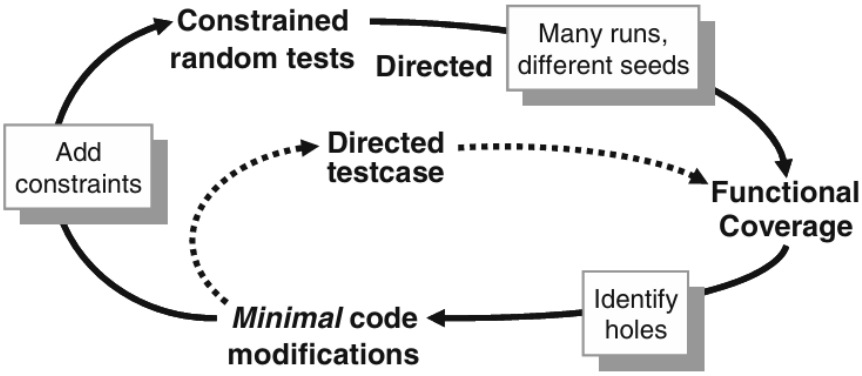


Figure 2.25: Coverage Convergence

Coverage serves as a fundamental metric for gauging the progress of design verification. Through simulations, engineers gradually explore the design space, aiming to encompass all legal combinations. Coverage tools collect data during simulation runs and subsequently generate coverage reports. These reports enable engineers to identify coverage gaps, prompting modifications to existing tests or the creation of new ones in an iterative fashion until the desired coverage level is achieved [14].
This is the most straightforward way to gauge your verification progress. It analyzes the design code, measuring various aspects of its execution [14]:

- Line Coverage: Tracks how many lines of code have been executed.
- Path Coverage: Counts the different paths through the code and expressions that have been exercised.
- Toggle Coverage: Monitors whether single-bit variables have been assigned both 0 and 1 values.
- Finite State Machine (FSM) Coverage: Assesses which states and transitions in a state machine have been visited.

The beauty of code coverage lies in its automated nature. Most simulators include built-in tools that analyze your design code by adding hidden elements to collect statistics. These tools generate a database after your tests run, which is then converted into user-friendly reports [14]. While code coverage offers valuable insights, it is crucial

to remember its limitations. High code coverage (even 100%) does not guarantee a perfect design. Potential bugs might still exist due to errors in the testing approach or missing functionalities in the implementation [14].
The ultimate goal of verification lies in ensuring the design functions flawlessly in its intended environment, be it a network router or a mobile phone. The design specifications define the expected behavior, while the verification plan outlines how to stimulate, verify, and measure those functionalities [14].
Functional coverage, also known as design coverage, focuses on measuring how well your tests exercise the design's intended functionalities as outlined in the specifications. This includes verifying not only the core functionality but also ancillary features like reset behavior or error handling [14].
An indirect yet insightful measure of verification coverage is the rate at which new bugs are discovered throughout the project life-cycle, as shown in figure 2.26. Tracking the number of bugs identified weekly offers valuable insights into the effectiveness of the verification process and the evolving state of the design [14].
At the project's outset, bug discovery often peaks as engineers accurately craft the testbench and scrutinize the design specification for inconsistencies. Early inspection processes frequently reveal numerous issues, which ideally are rectified before RTL implementation commences. Once the testbench is operational, a surge of bugs is typically uncovered as each module undergoes rigorous verification. Over time, the bug rate ideally diminishes, tapering off as the design approaches tape-out. However, reaching zero bugs does not signify completion; it merely indicates a significant milestone in the verification process [14].
As depicted in figure 2.26, it is not uncommon for bug discovery to persist even after tape-out and the design's deployment to customers. Continuous vigilance and proactive bug hunting remain essential practices to uphold the design's integrity and reliability throughout its life-cycle [14].

2.4.2 Universal Verification Methodology

The Universal Verification Methodology (UVM) stands as a Transaction Level Methodology (TLM) designed to streamline testbench development. It operates as a class library that simplifies the creation of configurable and reusable code, significantly enhancing the efficiency and reliability of hardware verification processes. While a basic understanding of OOP is advantageous, UVM's design philosophy revolves around providing pre-built components that facilitate testbench construction without requiring expertise in OOP intricacies [15].
One of the standout features of UVM lies in its reusability. Once foundational components are established within the UVM framework, they can be seamlessly carried

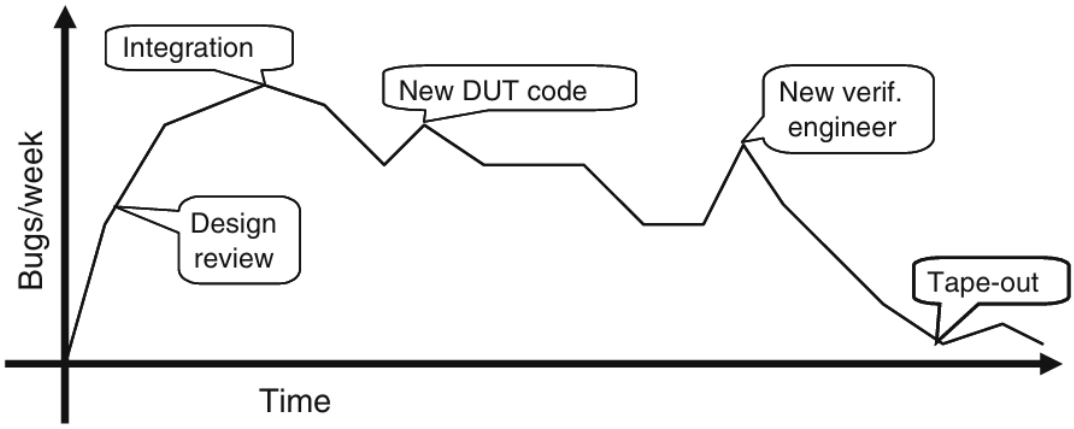


Figure 2.26: Bug Rate Throughout a Typical Project

forward to subsequent projects. Only specific components such as the driver, scoreboard, and transaction sequences need modification, enabling swift adaptation to new project requirements. In contrast, traditional SystemVerilog testbenches lack standardized coding practices, resulting in ad hoc implementations that hinder reusability and maintainability [15].

UVM adopts a class-based architecture where communication between components occurs through transactions. This design choice ensures that communication interfaces among class components remain distinct from implementation details within UVM agent drivers. Transactions facilitate the exchange of data and commands across different components, promoting modularity and flexibility within the testbench environment [15].

The UVM architecture is structured around a layered, object-oriented approach, which fosters the concept of "separation of concerns" among team members involved in testbench development. Each component within a UVM testbench is meticulously designed with a specific purpose and a well-defined interface to interact with other components. This design philosophy aims to boost productivity and promote code reuse across projects [16].

In a UVM testbench, the assembly of these components results in a modular and reusable verification environment. This environment enables test writers to concentrate on the functionality that needs verification, operating at the transaction level. Meanwhile, testbench architects focus on orchestrating how tests interact with the DUT, ensuring seamless integration and comprehensive verification coverage [16].

The UVM Testbench typically instantiates the DUT module and the UVM Test class and configures the connections between them as shown in figure 2.27. If the verification collaterals include module-based components, they are instantiated under the UVM Testbench as well. The UVM Test is dynamically instantiated at run-time, allowing the

UVM Testbench to be compiled once and run with many different tests [17].

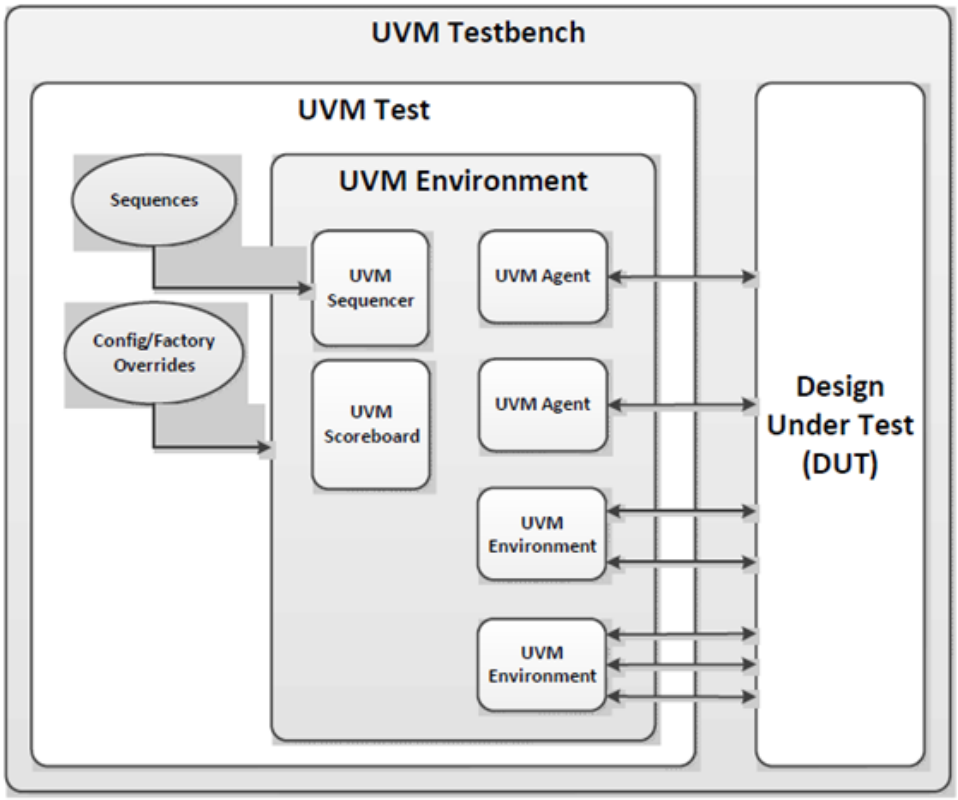


Figure 2.27: Typical UVM Testbench Architecture

The UVM Test is the top-level UVM Component in the UVM Testbench. The UVM Test typically performs three main functions: Instantiates the top-level environment, configures the environment (via factory overrides or the configuration database), and applies stimulus by invoking UVM Sequences through the environment to the DUT. Typically, there is one base UVM Test with the UVM Environment instantiation and other common items. Then, other individual tests will extend this base test and configure the environment differently or select different sequences to run [17].

The UVM Environment is a hierarchical component that groups together other verification components that are interrelated. Typical components that are usually instantiated inside the UVM Environment are UVM Agents, UVM Scoreboards, or even other UVM Environments. The top-level UVM Environment encapsulates all the verification components targeting the DUT [17].

The UVM Scoreboard's main function is to check the behavior of a certain DUT. The UVM Scoreboard usually receives transactions carrying inputs and outputs of the DUT through UVM Agent analysis ports, runs the input transactions through a reference model to produce expected transactions, and then compares the expected output versus the actual output [17].

The UVM Agent, shown in figure 2.28, is a hierarchical component that groups together other verification components that are dealing with a specific DUT interface. A typical UVM Agent includes a UVM Sequencer to manage stimulus flow, a UVM Driver to apply stimulus on the DUT interface, and a UVM Monitor to monitor the DUT interface. UVM Agents might include other components, like coverage collectors, protocol checkers, a TLM model, etc. [17]

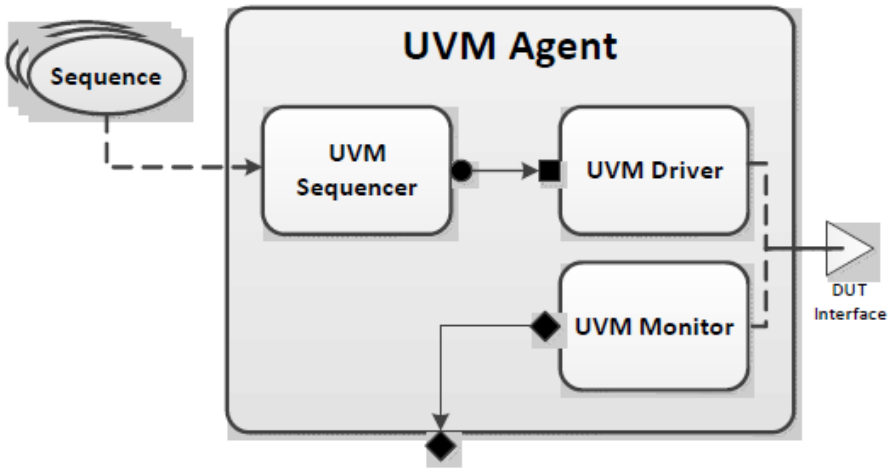


Figure 2.28: UVM Agent

The UVM Sequencer serves as an arbiter for controlling transaction flow from multiple stimulus sequences. More specifically, the UVM Sequencer controls the flow of UVM Sequence Items transactions generated by one or more UVM Sequences [17].
A UVM Sequence is an object that contains a behavior for generating stimulus. UVM Sequences are not part of the component hierarchy. UVM Sequences can be transient or persistent. A UVM Sequence instance can come into existence for a single transaction, it may drive stimulus for the duration of the simulation, or anywhere in-between. UVM Sequences can operate hierarchically with one sequence, called a parent sequence, invoking another sequence, called a child sequence [17].
The UVM Driver receives individual UVM Sequence Item transactions from the UVM Sequencer and applies (drives) it on the DUT Interface. Thus, a UVM Driver spans abstraction levels by converting transaction-level stimulus into pin-level stimulus. It also has a TLM port to receive transactions from the Sequencer and access to the DUT interface to drive the signals [17].
The UVM Monitor samples the DUT interface and captures the information there in transactions that are sent out to the rest of the UVM Testbench for further analysis. Thus, like the UVM Driver, it spans abstraction levels by converting pin-level activity to transactions. To achieve that, the UVM Monitor typically has access to the DUT interface and also has a TLM analysis port to broadcast the created transactions through

[17].

2.5 Literature on PCIe Verification

Verification of any design is one of the most important steps in the process of getting it into the market. This is becoming even more prevalent as designs become more complex and verification takes around 70% of the design cycle time [14]. Therefore, a good survey of previous verification attempts on similar designs is necessary to avoid preventable delays.

In paper [18], the authors designed a physical interface for PCIe express (PIPE) module and developed a SystemVerilog testbench to interface with it. The simulations were performed using Xilinx Integrated Synthesis Environment (ISE) design suite version 14.6 and managed to send the packets at a rate of 62.5 Mega Hertz (MHz) with a data interface width of 32-bits. For future additions, they suggested adding an LTSSM module to the testbench to test the link training process and integrate the process with a SerDes architecture for serial data generation. In paper [19], the authors designed an LTSSM unit and verified its functionality by using a UVM testbench for PCIe Gen3. In paper [20] the authors perform verification of PCIe Gen5 system using UVM and Synopsis Verilog Compiler Simulator (VCS) and Discovery Visual Environment (DVE) tools.

Chapter 3

Project Design and Planning

3.1 Description of The Selected Design

The given design was implemented by a team from Cairo university in 2021 and supervised by Siemens EDA. The RTL was provided by Dr. Hassan Mostafa for us to work on. The project's scope focused on creating a synthesizable RTL implementation for the logical part of the PCIe physical layer. This effort built upon previous work that had implemented the transaction and data link layers using a combination of Verilog and C++. The architecture of the design is shown in figure 3.1 below.

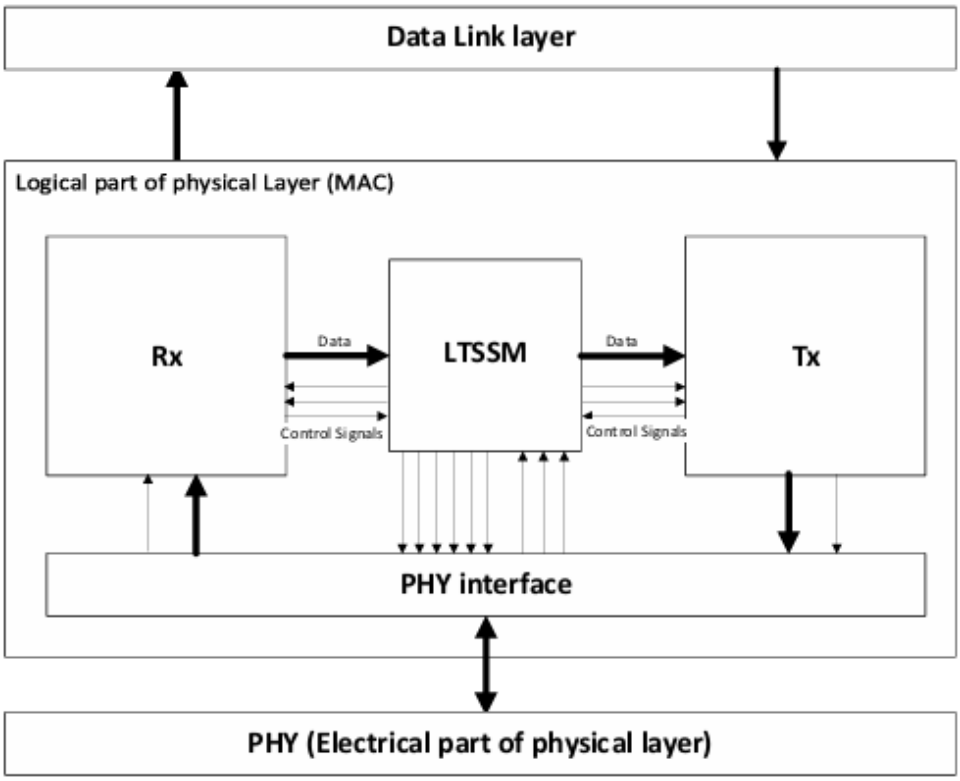


Figure 3.1: Design Architecture

After inspection, the design consists of three main modules:

1. **Link Training and Status State Machine (LTSSM) Module:**

   - Manages the PCIe link state, ensuring proper link training and maintenance.
   - Includes components like the OS Creator for generating ordered sets and a timer module for accurate state transitions.

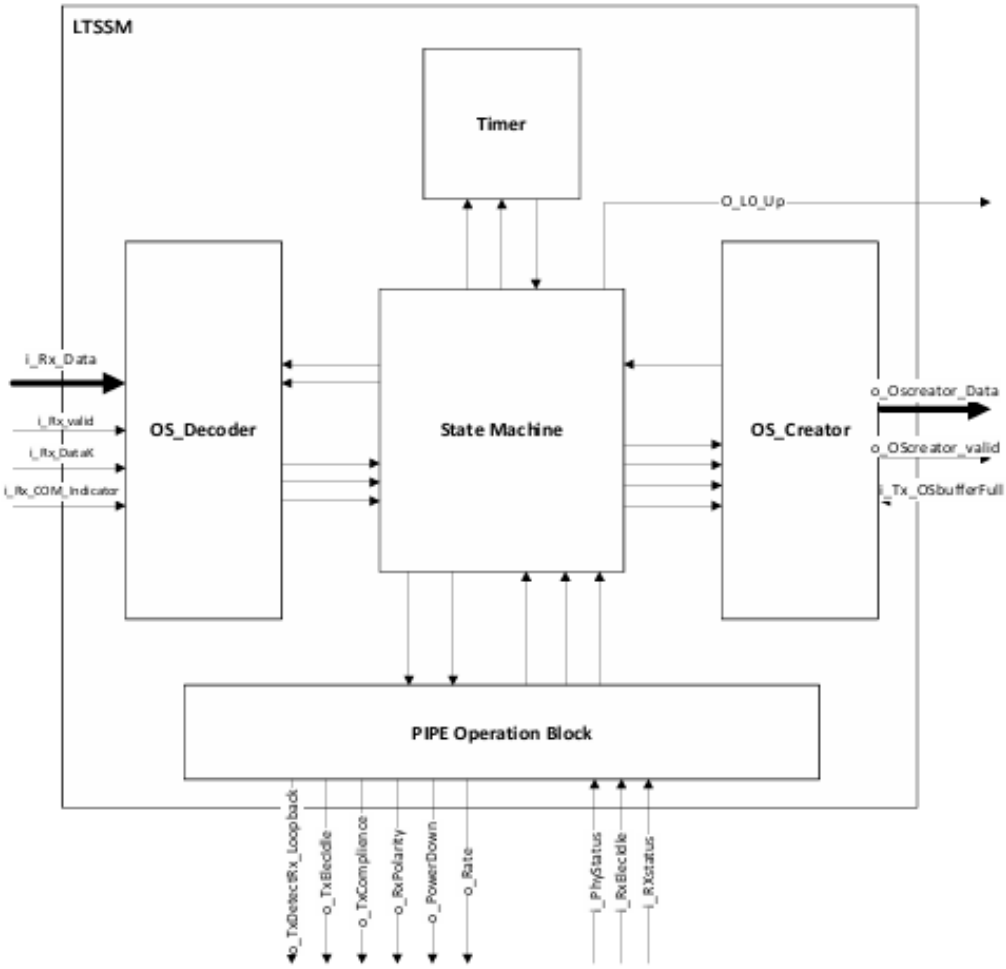


Figure 3.2: LTSSM Architecture

2. **Transmitter (Tx) Module:**

   - Responsible for handling the data transmission from the physical layer to the PCIe link.
   - Manages data transfer, packet segmentation, and control signals to ensure proper data flow and integrity.

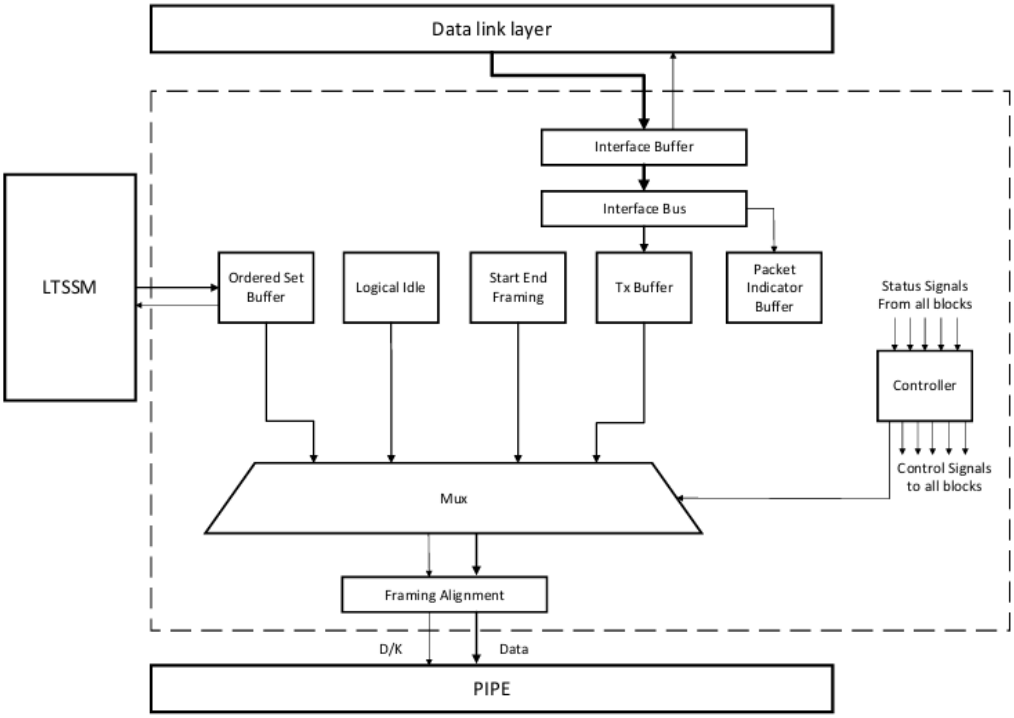


Figure 3.3: Tx Architecture

3. **Receiver (Rx) Module:**

   - Handles incoming data from the PCIe link and processes it.
   - Manages buffer storage, control signals, and ensures correct sequencing and data integrity.

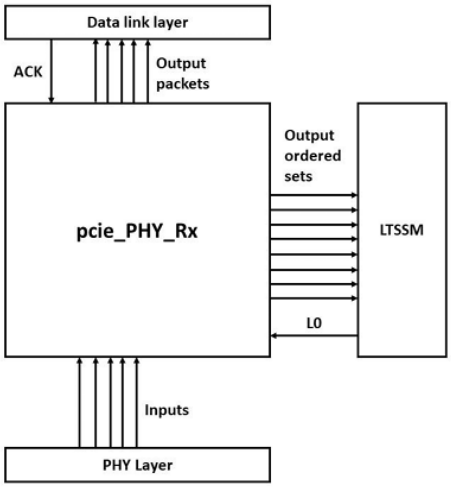


Figure 3.4: Rx Architecture

3.2 Project Purpose and Constraints

3.2.1 Project Purpose

The project's purpose is to verify the digital design of the PHY layer of PCIe Gen1 with verification flow as shown:

1. **Feature Extraction.** This step involves a deep understanding of PCIe standard and makes comparisons between the implemented features in the given RTL code and the standard.

2. **Verification Planning.** For each feature extracted from the designed blocks we must perform the following steps:

   - Block I/O: Understanding the location of the module in the whole design, its I/O signals, and how it communicates with other modules.
   - Verification Architecture: Defining the testbench architecture. This includes components and objects in the environment.
   - Test Items and Sequences: Identify the test items and sequences to be implemeneted.
   - Checking Mechanism: Developing a reference model to compare the implemented features output with it and check the functionality of the implemented design.
   - Coverage Plan: Definition of cover bins and groups that this feature should hit when tested.

3. **Implementing The Verification Environment.** Writing the complete testbench code and UVM environment based on the verification plan.

4. **Evaluating the module.** Reporting coverage, identifying bugs, and assessing the overall state of the block.

3.2.2 Project Constraints

Due to the initial project circumstances and the complexity of the PCIe standard, the most critical constraint is time. The first constraint is to complete the verification by the end of the academic year. The second constraint is to achieve a high code and function coverage, specifically greater than 90 %.

3.3 Project Tasks and Gantt Chart

The project tasks are defined as follows:

1. Learning verification concepts, SystemVerilog, and UVM.
2. Understanding the operation and architecture of PCIe.
3. Examining the design to be verified for completeness.
4. Design and integration of any missing mandatory functionalities or features.
5. Breaking down the design into well-defined modules.
6. Writing detailed specification documents and implementing reference models for each module.
7. Developing verification plans for each module.
8. Conducting block-level verification for all modules.
9. Reporting bugs and coverage results.
10. Fixing bugs in the RTL (whenever possible) and reevaluating the fixed modules for functionality.
11. Performing chip-level verification for the Tx and LTSSM.

The Gantt chart illustrating the project flow is shown in Figure 3.5.

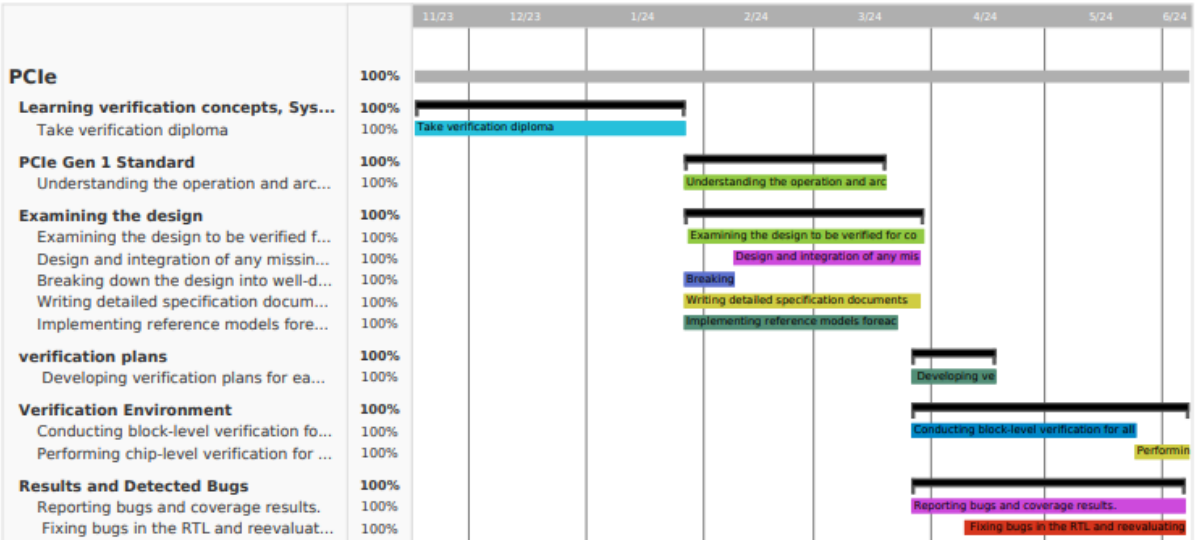


Figure 3.5: Gantt Chart

3.4 Verification Approach

A coverage-driven constrained random functional verification methodology is employed. This approach allows for a more efficient use of resources by automating test generation, systematically exploring the design space, and uncovering unexpected corner cases and subtle bugs. A UVM-based verification environment is used. This offers standardization, modularity, reusability, scalability, and support for advanced verification techniques. Verification will be performed using Mentor QuestaSim as the primary simulator. Other tools such as Synopsys VCS may also be used to ensure compatibility with different simulators. To ensure that the module operation and functionalities are mostly covered, functional coverage of 100% and code coverage of 90% are required as exit criteria.

In verifying both at the block and chip levels, multi-agent UVM environments prove essential due to the complicated communication among modules. Designed with reusability in mind, the agents in block-level environments can be set active or passive depending on the scenario. In block-level scenarios, all agents operate actively, whereas in chip-level verification, some are intentionally set to passive mode.

Chapter 4

Block-Level Verification

In any verification environment designed to test the functionality of a single block, multiple agents are employed to mimic the communication behavior between the block of interest and other blocks. For example, a global agent is implemented to manage the sequence of global signals, such as clock, reset, and the type of device (upstream or downstream) signals. In environments with numerous agents, the internal components of these agents are typically not detailed for simplicity, as they all share the same structure, as illustrated in the figure 4.1.

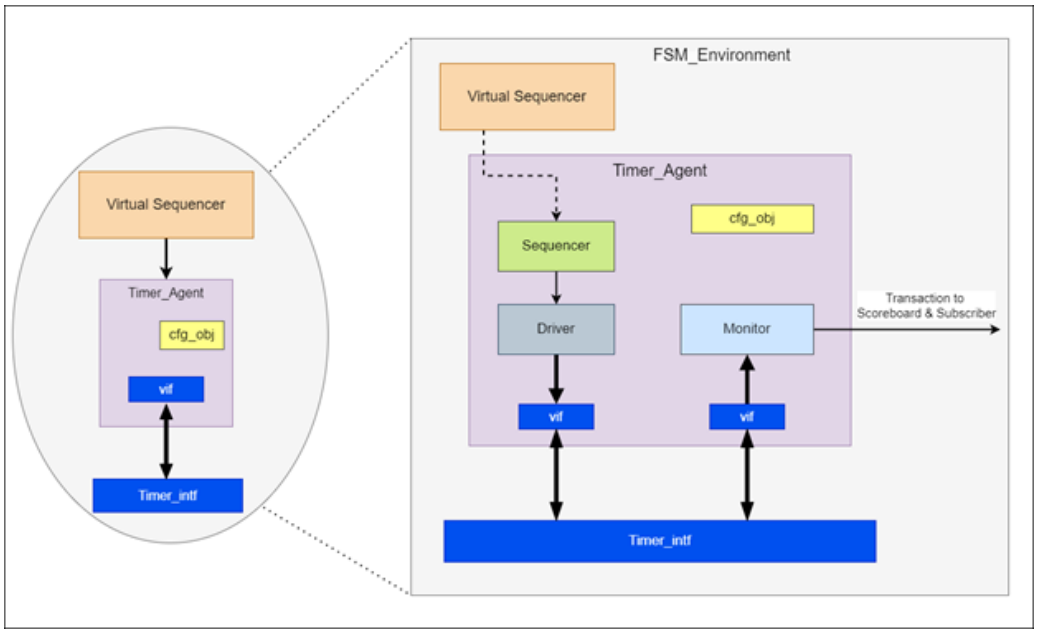


Figure 4.1: Agent internal structure

Due to the presence of multiple agents in a single environment, coordinating stimulus between different drivers is essential. This coordination is achieved through the implementation of virtual sequencers. Virtual sequencers manage the agents' sequencers by executing individual sequence items on a driver via a sequencer port. The overall coordination of the various sequences is handled within a virtual sequence defined

inside the test, as illustrated in figure 4.40 [21].

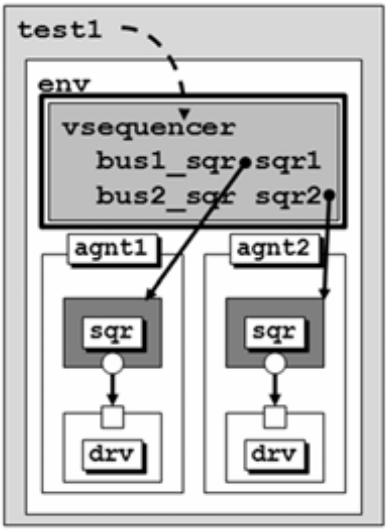


Figure 4.2: Stimulus Coordination using Virtual Sequencer

Note: connections between virtual sequencers and agents, or more specifically sequencers in agents, are not included in UVM hierarchy figures to give more clarity to the implemented hierarchy.
For each agent, two modports are implemented, each containing a clocking block specifying the direction of the included signals. One modport is used as a driving port for the stimulus generation, and the other is a monitoring port for the sampling process.
Regarding the checking mechanism, the design will be compared against a golden reference model implemented in the scoreboard class using SystemVerilog. Transactions from the various agents are combined into a single sequence item that includes all inputs and outputs of the DUT. During the run phase of UVM, the actual and reference outputs will be stored in two separate queues as shown in figure 4.3. In the check phase, these stored outputs will be compared against the reference outputs, with any discrepancies being recorded and reported.

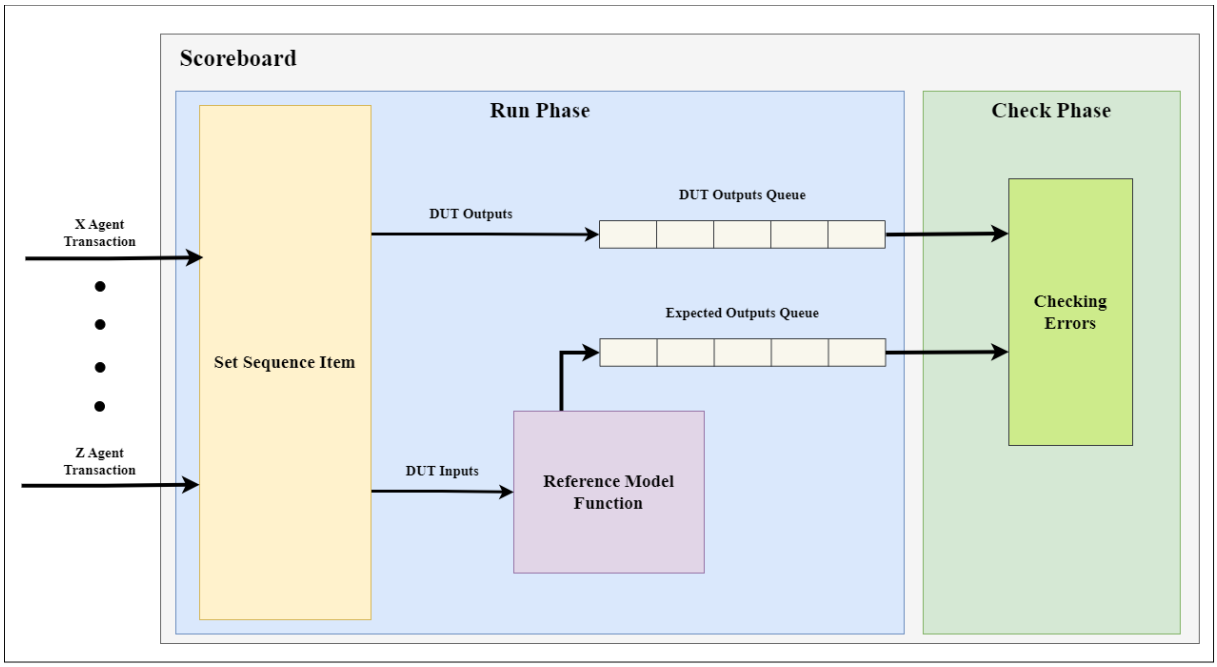


Figure 4.3: Generic Checking Mechanism

4.1 LTSSM Blocks

4.1.1 PIPE Operation Block

Introduction

Design Description:

The PIPE Operation Block module is a submodule under the LTSSM block. It is the connection between the LTSSM and the PHY layer as shown in 4.4. The upper signals are connected to the State Machine while the lower signals are connected to a PHY module given by Mentor Graphics which we don't have access to.

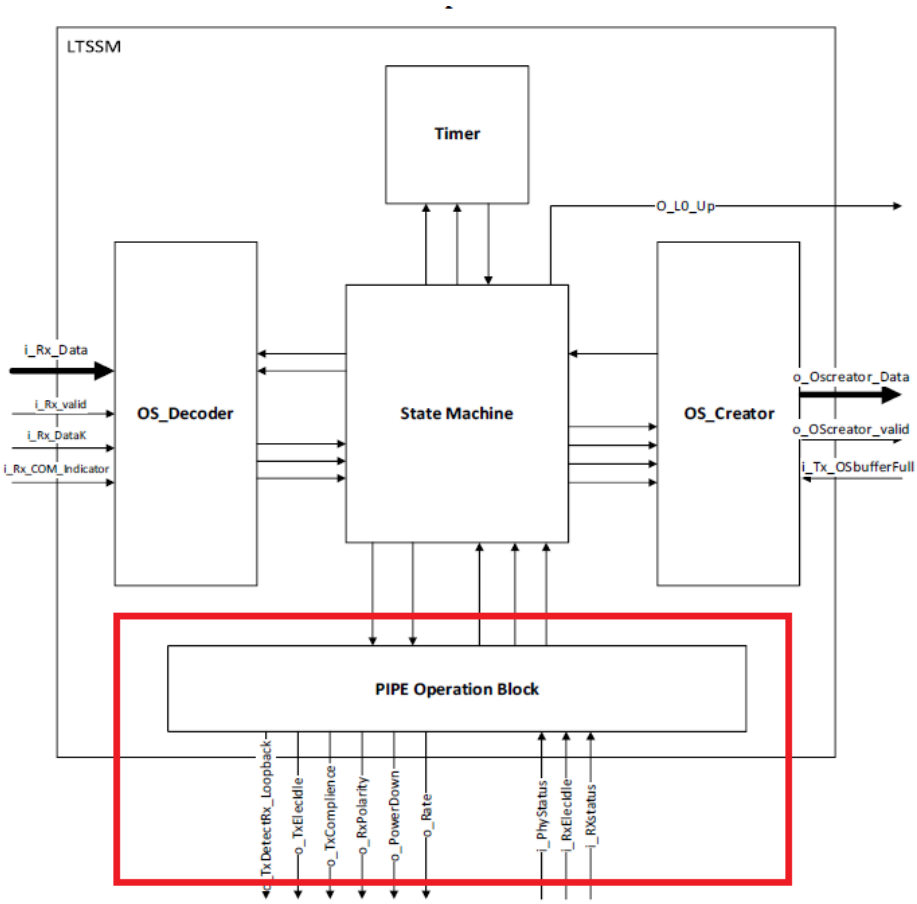


Figure 4.4: LTSSM System - PIPE Block Highlighted

Block I/O Signals:
The I/O signals of the block are shown in figure 4.5 below.
Interaction with Other Modules:

- LTSSM State Machine: PIPE sends information about state of the receiver and whenever another receiver is detected in the physical link. So, it updates the flow during the Detect state.

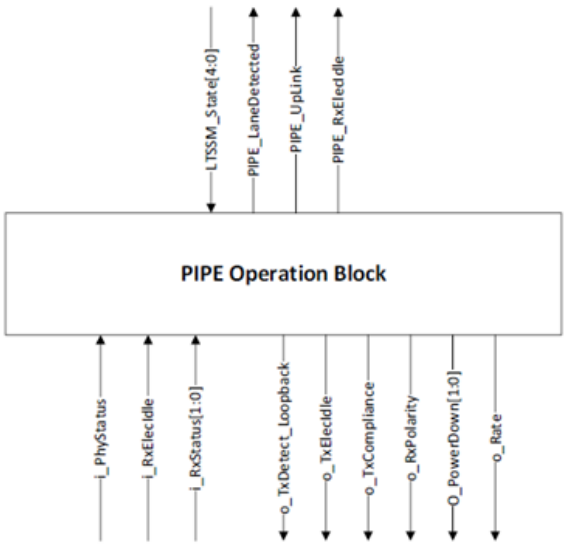


Figure 4.5: PIPE Operation Block Diagram

I/O Signals Description

Note: (*) means that this signal comes from PIPE V3.0 standard.

Name	Direction	Source / destination	Description
i_LTSSM State [4:0]	Input	State Machine	Indicate the current state
i_Phy Status*	Input	PHY Layer	Used to communicate completion of several PHY functions including stable PCLK after Reset_n deassertion, power management state transitions, rate change, and receiver detection.

Continued on next page

(Continued)

i_Rx Status* [2:0]	Input	PHY Layer	Encodes receiver status and error codes for the received data stream when receiving data. Value Description 0 Received data OK 1 1 SKP added 10 A SKP removed 11 Receiver detected 100 Both 8B/10B decode error and receive disparity error 101 Elastic buffer overflow 110 Elastic buffer under flow 111 Receive disparity error
o_Tx DetectRx loopback*	Output	PHY Layer	Used to tell the PHY to begin a receiver detection operation or to begin loopback
o_Tx ElecIdle*	Output	PHY Layer	Tells PHY that the transmitter is electrically idle
o_Tx Compliance*	Output	PHY Layer	Sets the running disparity to negative. Used when transmitting the PCI Express compliance pattern
o_Rx Polarity*	Output	PHY Layer	Tells PHY to do a polarity inversion on the received data.
o_Power Down* [1:0]	Output	PHY Layer	its value is 0 for L0 states, and takes other values for low-power states which are not supported.
o_Rate*	Output	PHY Layer	Control the link signaling rate.
o_LTSSM UpLink	Output	State Machine	Indicates bit and symbol lock, ready to L0 state

Continued on next page

(Continued)

o_LTSSM Rx ElecIdle	Output	State Machine	Mirror the i_RxElecIdle coming from PHY to LTSSM
o_LTSSM Lane Detected	Output	State Machine	Indicate successful detection of a receiver to LTSSM

Detailed Operation

It is used to detect whether there's a receiver connected or not. It takes the state of LTSSM i_LTSSM_State as an input, when it is in the detect active, it checks the i_PhyStatus signal, if it is '1', it then checks the i_RXstatus signal and asserts the o LTSSM LaneDetected signal only if there's a receiver detected. Otherwise, the o LTSSM LaneDetected signal is always '1' unless it is in the detect quite state.

Evaluation

All possible combinations of inputs are injected into the DUT and it showed no bugs.

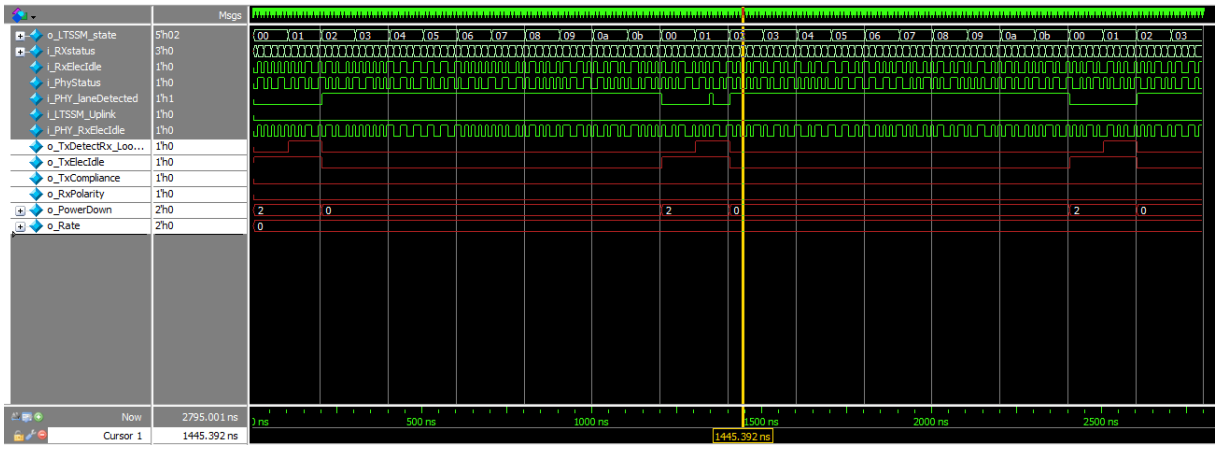


Figure 4.6: PIPE Block Functionality

```
----------------------------------------------------------------------------------------------------------------------------------
UVM_INFO PIPE_env/PIPE_scoreboard.svh(323) @ 2795001: uvm_test_top.env_h.scoreboard_h [PIPE_Scoreboard]
At [2795001 ps]:
Simulation summary: 278 matching, 0 mismatching

----------------------------------------------------------------------------------------------------------------------------------
UVM_INFO PIPE_env/PIPE_subscriber.svh(107) @ 2795001: uvm_test_top.env_h.subscriber_h [PIPE_subscriber]
Final Coverage: Cover Group Coverage = [100 %]
----------------------------------------------------------------------------------------------------------------------------------

--- UVM Report Summary ---

** Report counts by severity
UVM_INFO :  560
UVM_WARNING :    0
UVM_ERROR :    0
UVM_FATAL :    0
```

Figure 4.7: PIPE Block Final Log

Local Instance Coverage Details:

Total Coverage:					100.00%	100.00%
Coverage Type	Bins	Hits	Misses	Weight	% Hit	Coverage
Statements	27	27	0	1	100.00%	100.00%
Branches	8	8	0	1	100.00%	100.00%
FEC Conditions	4	4	0	1	100.00%	100.00%
Toggles	28	28	0	1	100.00%	100.00%

Figure 4.8: PIPE Block Code Coverage Details

Search:

CoverPoints	Total Bins	Hits	Misses	Hit %	Goal %	Coverage %
Phy_status	2	2	0	100.00%	100.00%	100.00%
RxElecIdle	2	2	0	100.00%	100.00%	100.00%
RXstatus	8	8	0	100.00%	100.00%	100.00%
State	12	12	0	100.00%	100.00%	100.00%

Search:

Crosses	Total Bins	Hits	Misses	Hit %	Goal %	Coverage %
State_Phy	24	24	0	100.00%	100.00%	100.00%
State_Rx	96	96	0	100.00%	100.00%	100.00%

Activate Win

Figure 4.9: PIPE Block Functional Coverage

4.1.2 OS Decoder

Introduction

Design Description:

This IP receives data from the PCIe Rx and the state machine. Based on the data's content and the current state, it controls the state transitions during link training and initialization process.

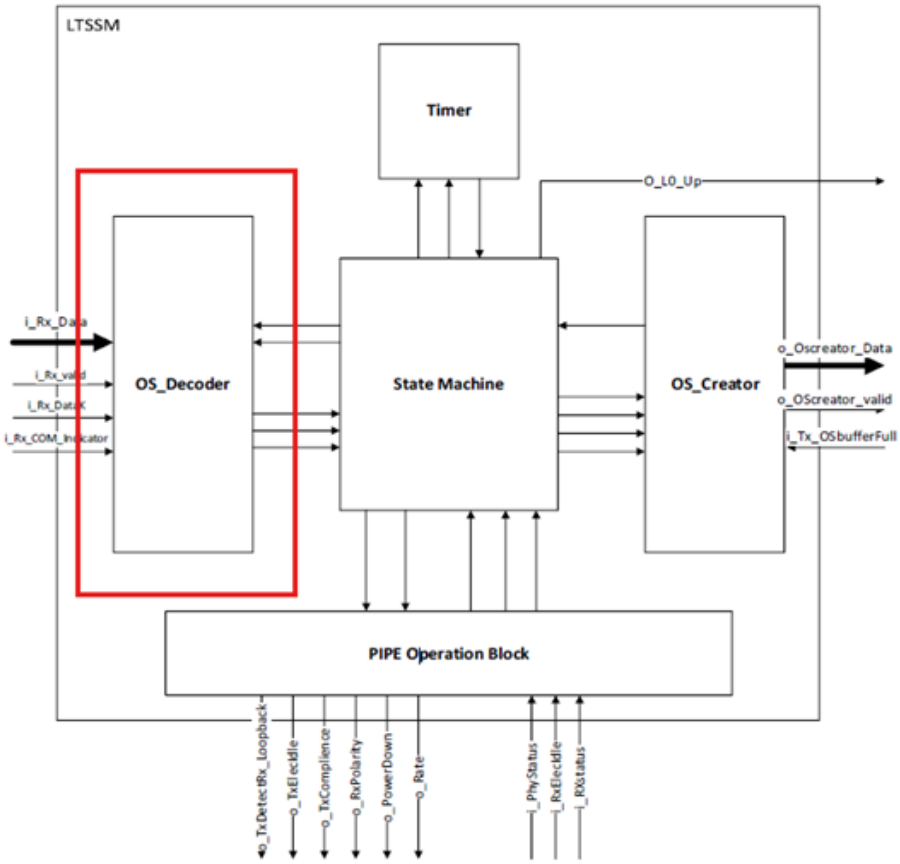


Figure 4.10: LTSSM implementation - Decoder highlighted

Block I/O Signals:

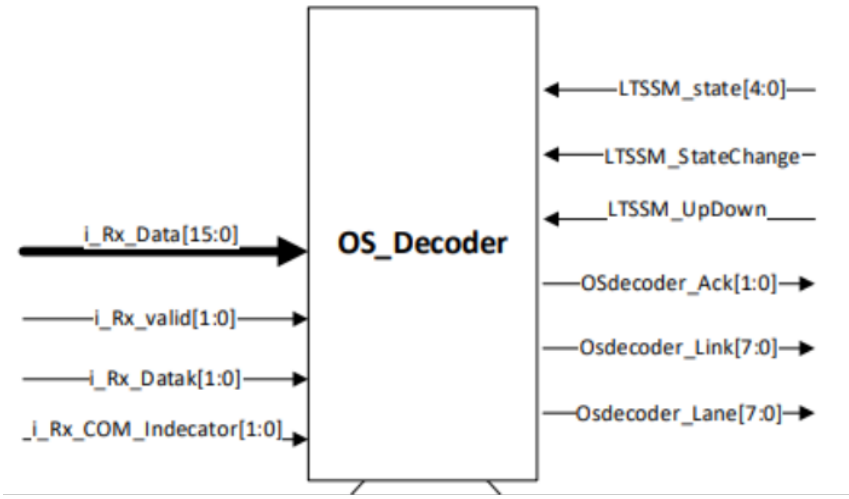


Figure 4.11: OS_Decoder implementation

Interaction with Other Modules:

- PCIe Rx: The decoder gets input data from the receiver. This data can be ordered sets sent from the other PCIe device on the link, and hence they are used to proceed throughout the link training and initialization process.
- LTSSM State Machine: The Decoder gets input signals from the State Machine, which include information about the current state of the link, whenever the state changes, and the type of the device (Upstream or Downstream). Also, the Decoder outputs signals to the State Machine, in which it can acknowledge that a state change should take place and the link and lane numbers of the received data from the receiver (useful in multi-lane communications).

I/O Signals Description

Inputs:

Name	Width	Source	Description
clk	1 bit	Global	Clock signal
resetn	1 bit	Global	Synchronous low-level reset signal
i_Rx_Data	2 Bytes (16 bits)	PCIe Rx	Data from the receiver (sent from the other device on the link).

Continued on next page

(Continued)

i_Rx_DataK	2 bits	PCIe Rx	Each bit corresponds to a byte in the Rx_Data signal. If the bit is 0, it corresponds to a data byte in the Rx_Data, and if it is 1, it corresponds to a control byte in the Rx_Data. Note: control bytes are COM, and bytes of Other OS (FTS, SKP, IDL).
i_Rx_valid	2 bits	PCIe Rx	Each bit indicates the validity of a byte in the Rx_Data signal.
i_Rx_COM Indicator	2 bits	PCIe Rx	Indicates the beginning of a received ordered set in the Rx_Data signal. Note: 11 is an invalid case.
LTSSM_state	5 bits	State Machine	Encoding of the current state of the link.
LTSSM StateChange	1 bit	State Machine	Indicates a change in the state of the link, so that the decoder should reset any counter of the received ordered sets to begin accounting only for those received in the new state.
LTSSM UpDown	1 bit	State Machine	1 for Downstream and 0 for Upstream.

Outputs:

Name	Width	Source	Description
OSdecoder_Ack	2 bits	State Machine	Acknowledgement signal on the number of TS received to control the transition between the states. Note: for states requiring two conditions to occur, this signal is 01 when the first one occurs and 10 when the second one occurs. (e.g. Config_complete requires receiving one then eight TS2/Link/Lane.
OSdecoder_Lane	8 bits	State Machine	The lane number decoded from TS1 and TS2.

Continued on next page

(Continued)

OSdecoder_Lane	8 bits	State Machine	The link number decoded from TS1 and TS2.

Detailed Operation

This IP consists of 4 sub-blocks: filter, TS Detector, Other OS Detector, and the Decoder Main Block. The filter checks if the received data from the PCIe Rx is TS or Other OS, and accordingly it enables TS Detector or Other OS Detector or neither of them (if the data received is not TS nor Other OS). TS Detector stores the content of the TS two byte-by-two byte till the whole 16 bytes are stored, then it outputs a detect signal to the Decoder Main Block. Similarly, Other OS Detector outputs detect and type signals when it receives the 4 bytes of an OS (SKP, FTS, IDL, EIE). Finally, the Decoder Main Block keeps counting whenever a TS is detected only if the TS content is matching with the required in the current state (e.g. TS2/PAD/PAD in Polling.Config state). When the required number of TS is received, it outputs an acknowledgment signal to the state machine to change the state.

Verification approach

In this block-level environment, 3 active agents are used to interact with the Decoder Module through 3 interfaces combining all input and output signals of the Decoder. The agents are:

1. Global
2. FSM
3. Rx

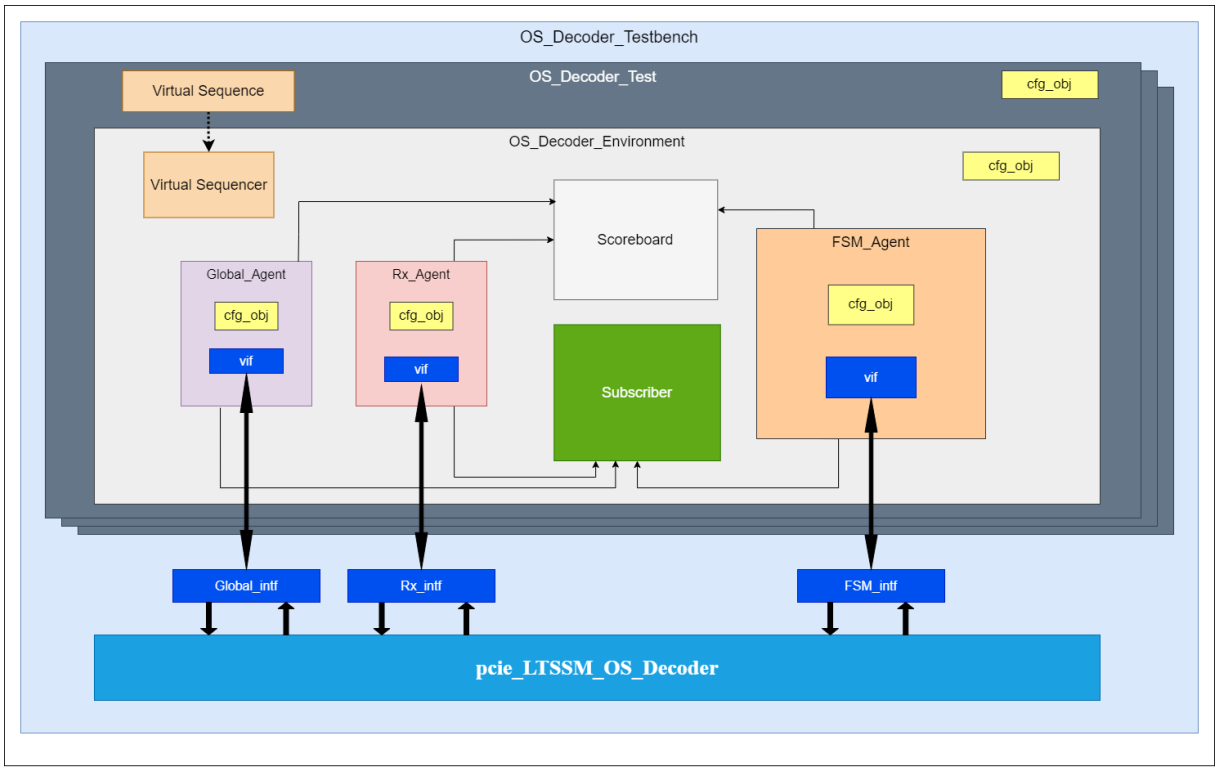


Figure 4.12: Decoder testbench architecture and hierarchy

Test Items and Sequences

Symbols In this subsection, I'll list here all symbols possible and then use these symbols to build test items and sequences as the i_Rx_Data consists of 2 symbols and each symbol is 8 bits.

Symbol Name	Encoding
PAD	8'F7
TS1	8'h4A
TS2	8'h45
COM	8'hBC
SKP	8'h1C
FTS	8'h3C
IDL	8'h7C
EIE	8'hFC

Test items

The table below shows the test items that will be implemented and used in constructing the test cases. The below will show just an example for the sequences and not all of them as they are too many to be put in the thesis.

Sequence Name	Value	Description
COM_PAD	i_Rx_Data=16'hF7BC (Link#, COM), i_Rx_DataK=2'b11, i_Rx_COM_Indicator=2'b01, i_Rx_valid=2'b11.	This is a combination for the COM symbol before the configuration state so the Link# should be PAD
COM_Link	i_Rx_Data=16'h01BC (Link#, COM), i_Rx_DataK=2'b01, i_Rx_COM_Indicator=2'b01, i_Rx_valid=2'b11.	This is a combination for the COM symbol after the configuration state so the Link# should have a value

Continued on next page

(Continued)

PAD_FTS	i_Rx_Data=16'h3CF7 (N_FTS, lane#), i_Rx_DataK=2'b01, i_Rx_COM_Indicator=2'b00, i_Rx_valid=2'b11.	This combination is for symbols 2 and 3 before the configuration state so the lane# should be PAD
Lane_FTS	i_Rx_Data=16'h3C01 (N_FTS, lane#) i_Rx_DataK=2'b00 i_Rx_COM_Indicator=2'b00 i_Rx_valid=2'b11.	This combination is for symbols 2 and 3 after the configuration state so the lane# should have a value
Rate_TC	i_Rx_Data=16'h0002, i_Rx_DataK=2'b00, i_Rx_COM_Indicator=2'b00, i_Rx_valid=2'b11.	This combination is for symbols 4 and 5 which indicates Training Ctrl, and Rate ID. They have a fixed value which is 16'h0002
TS1	i_Rx_Data=16'h4A4A, i_Rx_DataK=2'b00, i_Rx_COM_Indicator=2'b00, i_Rx_valid=2'b11.	This is the ID of TS1
TS2	i_Rx_Data=16'h4545, i_Rx_DataK=2'b00, i_Rx_COM_Indicator=2'b00, i_Rx_valid=2'b11.	This is the ID of TS1
Not_TS_nor _OS	i_Rx_Data=16'hxxxx i_Rx_DataK=2'bxx i_Rx_COM_Indicator=2'b00 i_Rx_valid=2'b11	This is when the input is not a TS, nor an OS and it is also not logical Idle
Logical_idle	i_Rx_Data=16'h0000, i_Rx_DataK=2'b00, i_Rx_COM_Indicator=2'b00, i_Rx_valid=2'b11.	This is the logical idle sequence

Sequences base

The table below shows the tests that will be applied to the design, the sequences used, and the description of each test. These test cases may contain several sequential inputs one after another.

Test Name	Sequences	Description
TS1/PAD/PAD	COM_PAD - PAD_FTS - RATE_TC - TS1 (x5)	This test case is the training sequence for the TS1 before configuration state
TS2/PAD/PAD	COM_PAD - PAD_FTS - RATE_TC - TS2 (x5)	This test case is the training sequence for the TS2 before configuration state
TS1/PAD/Link#	COM_Link - PAD_FTS - RATE_TC - TS1 (x5)	This test case is the training sequence for the TS1 after configuration state where link# has a value while lane# is PAD
TS2/PAD/Link#	COM_Link - PAD_FTS - RATE_TC - TS2 (x5)	This test case is the training sequence for the TS2 after configuration state where link# has a value while lane# is PAD
TS1/Lane#/Link#	COM_Link - Lane_FTS - RATE_TC - TS1 (x5)	This test case is the training sequence for the TS1 after configuration state where lane# and link# must have values
TS2/Lane#/Link#	COM_Link - Lane_FTS - RATE_TC - TS2 (x5)	This test case is the training sequence for the TS2 after configuration state where lane# and link# must have values

Full Sequences

Complete sequence	Sequences	Description
Polling.Active_to _Polling.Config states	8x TS1/PAD/PAD, LTSSM_state=5'b00010	This sequence is needed for the polling active state to go to the polling config state
Polling.Config_to _Configration states	8x TS2/PAD/PAD, LTSSM_state=5'b00011	This sequence is needed for the polling config state to go to the polling configuration state
Lanenuum.wait_to _Lanenum.Accept States - downstream	2x TS1/Lane#/Link#, LTSSM_state=5'b00110, LTSSM_UpDown=1'b1	This sequence is needed to go from the config.Lanenum.wait state to the config.Lanenum.wait state
Lanenum.Accept_to _Config.Complete States - downstream	2x TS1/Lane#/Link#, LTSSM_state=5'b00111, LTSSM_UpDown=1'b1	This sequence is needed to go from the config.Lanenum.accept state to the config_complete state
Lanenuum.wait_to _Lanenum.Accept States - upstream	2x TS2/Lane#/Link#, LTSSM_state=5'b00110, LTSSM_UpDown=1'b0	This sequence is needed to go from the config.Lanenum.wait state to the config.Lanenum.wait state
Lanenum.Accept_to _Config.Complete States - upstream	2x TS2/Lane#/Link#, LTSSM_state=5'b00111, LTSSM_UpDown=1'b0	This sequence is needed to go from the config.Lanenum.accept state to the config_complete state
Config.complete_to _Config.idle States	1x TS2/Lane#/Link#, then, 8x TS2/Lane#/Link#, LTSSM_state=5'b01000	This sequence is needed to go from the config.complete state to the config idle state

Continued on next page

(Continued)

Config.idle_to_L0	1x IDLE then, 4x IDLE, LTSSM_state=5'b01001	This sequence is needed to go from the config_idle state to the L0 state

Coverage Collection

Functional coverage collection will be performed using the subscriber class. The table below shows the coverage bins needed for coverage collection.

Signal	Coverage Bins Description
reset	0=reset data, 1=active data
Rx_Data	COM_PAD, COM_Link, PAD_FTS, Lane_FTS, DR_LIDLE, TS1_TS1, TS2_TS2, LIDLE_LIDLE
Rx_Valid	11=both bytes are valid, 10 =Valid_Invalid, 01=Inavlid_Valid, 00=Invalid_Invalid
Rx_DataK	00=data_data, 01=data_control, 10=control_data, 11=control_control
Rx_COM_Indicator	10=Com_noCom, 01=noCom_Com, 00=noCom_noCom
LTSSM_State	A bin for each state.
LTSSM_stateChange	1=State change, 0=no state change
LTSSM_UpDown	1=DownStream, 0=UpStream
OS_Decoder_Ack	11=Ack_Ack, 01=noAck_Ack, 00=noAck_noAck
OS_Decoder_Link	01=Link number
OS_Decoder_Lane	01=Lane number

Code coverage will be performed automatically by the tool and the target is to exceed 90 percent in overall code coverage.
There were some exclusions done to the code as there were whole blocks that were dummies such as Other_Os_Detector which was implemented but its functionality was not supported.

Evaluation

Detected Bugs

Lane/Link errors:

- When the decoder is reset, the lane numbers are initiated to 0, however, it should take its default value which is PAD (8h'f7).
- When an 8x sequence is sent and then followed by a 9x sequence, the second sequence is shifted, and the lane, and ack are ruined.
- Lane should be updated after sending a fully processed TS1, however in the RTL, it is updated a cycle late (lanenum wait to lanenum accept transition).
- Link should be updated after sending a fully processed TS1, however in the RTL, it is updated after the second TS1.

Ack errors:

- When a complete sequence is sent, Ack is asserted for only one sequence and then de-asserted which shouldn't happen, ack signal should still be high until state change signal is asserted.
- The transition from config complete to config idle needs 1xTS2/Lane/Link then 8xTS2/Lane/Link, however, the ack is raised after 7xTS2/Lane/Link only.
- In the polling active state, after sending 8xTS1/PAD/PAD the ack signals get asserted but when we start sending 8xTS1/PAD/PAD (to go to the config state) the ack signal is not de-asserted, it is kept at 01 for the whole sequence and then gets de-asserted when next sequence is sent.
- The transition from config idle to L0 states should receive 1xIDLE then 4xIDLE, However the first Ack signal (2'b01) don't appear except after receiving 2xIDLE not 1xIDLE.

Resolving Bugs

All the previously mentioned bugs have been resolved in the RTL, and a comprehensive simulation was conducted using an extensive sequence. The simulation was completed with zero errors, as demonstrated in figure 4.13 below.

```
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO :    2
# UVM_WARNING :    0
# UVM_ERROR :    0
# UVM_FATAL :    0
# ** Report counts by id
# [RNTST]     1
# [TEST_DONE]     1
# ** Note: $finish    : F:/Others/Questa_ins/win64/../verilog_src/uvm-1.1d/src/base/uvm_root.svh(430)
#    Time: 56280 ns  Iteration: 66  Instance: /decoder_tb1
# Break in Task uvm_pkg/uvm_root::run_test at F:/Others/Questa_ins/win64/../verilog_src/uvm-1.1d/src/base/uvm_root.svh line 430
# 0 ps
# 59094 ns
```

Figure 4.13: Decoder Final Log

Code Coverage

Coverage Summary By Instance:

Scope	TOTAL	Statement	Branch	FEC Expression	FEC Condition	Toggle	FSM State	FSM Trans
TOTAL	93.79	89.15	88.81	100.00	100.00	84.82	100.00	100.00
decoder_DUT	80.00	--	--	--	--	80.00	--	--
decoderFilter	93.77	80.00	88.88	100.00	100.00	100.00	--	--
TSdetector	98.92	100.00	100.00	--	100.00	94.64	100.00	100.00
otherOSdetector	95.33	100.00	72.00	100.00	100.00	100.00	100.00	100.00
decoderMain	90.13	82.35	90.21	100.00	100.00	78.12	--	--

Figure 4.14: Decoder code coverage

As illustrated in figure 4.14, the code coverage achieved is 93.8%, which exceeds our target of 90%.

Functional Coverage

We achieved 100% functional coverage as we hit all the cover points that were mentioned above in the coverage section.

Covergroups

Name	Class Type	Coverage	Goal	% of Goal	Status	Included	Merg
/decoder_pkg/decoder_subscriber		100.00%					
TYPE cvr_grp	decoder_subscriber	100.00%	100	100.00...			
CVP cvr_grp::i_resetn_cp	decoder_subscriber	100.00%	100	100.00...			
CVP cvr_grp::i_Rx_Data_cp	decoder_subscriber	100.00%	100	100.00...			
CVP cvr_grp::i_Rx_valid_cp	decoder_subscriber	100.00%	100	100.00...			
CVP cvr_grp::i_Rx_DataK_cp	decoder_subscriber	100.00%	100	100.00...			
CVP cvr_grp::i_Rx_COM_Indicator_cp	decoder_subscriber	100.00%	100	100.00...			
CVP cvr_grp::i_LTSSM_state_cp	decoder_subscriber	100.00%	100	100.00...			
CVP cvr_grp::i_LTSSM_stateChange_cp	decoder_subscriber	100.00%	100	100.00...			
CVP cvr_grp::i_LTSSM_UpDown_cp	decoder_subscriber	100.00%	100	100.00...			
CVP cvr_grp::o_OSdecoder_Ack_cp	decoder_subscriber	100.00%	100	100.00...			
CVP cvr_grp::o_OSdecoder_Link_cp	decoder_subscriber	100.00%	100	100.00...			
CVP cvr_grp::o_OSdecoder_Lane_cp	decoder_subscriber	100.00%	100	100.00...			

Figure 4.15: Decoder Functional Coverage

4.1.3 Timer

Introduction

Design Description:

The timer module is a submodule under the LTSSM block, it is only connected to the state machine as shown in figure 4.16.

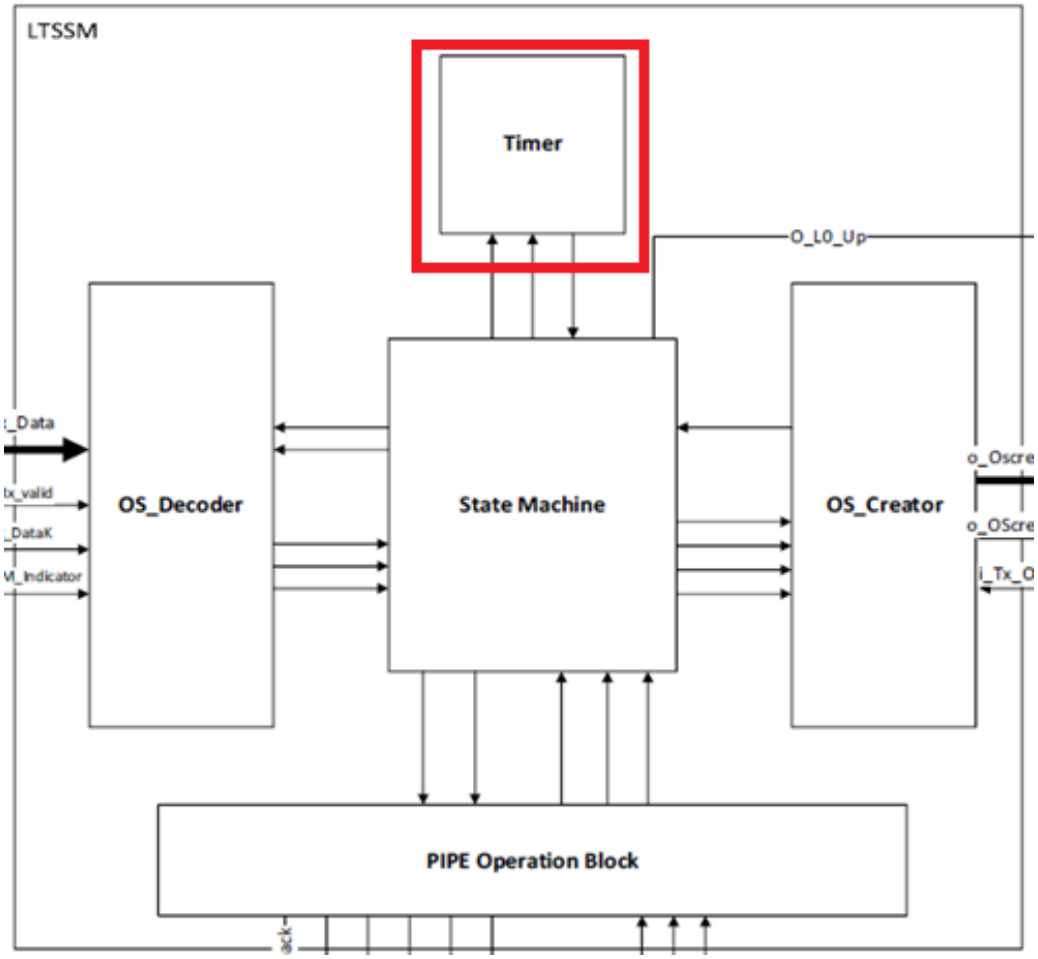


Figure 4.16: LTSSM Implementation – Timer Highlighted

Block I/O Signals:

I/O Signals

Name	Direction	Description
i_time_out value [22:0]	Input	Input from State machine: Time, we need to reach
i_clk	Input	Clock
i_start	Input	Input from State machine. To start count the time - Start = 1, Start count - Start = 0, Stop count and restart all the time data (when we start again, we will start from 0)
i_resetn	Input	Active-low reset

Continued on next page

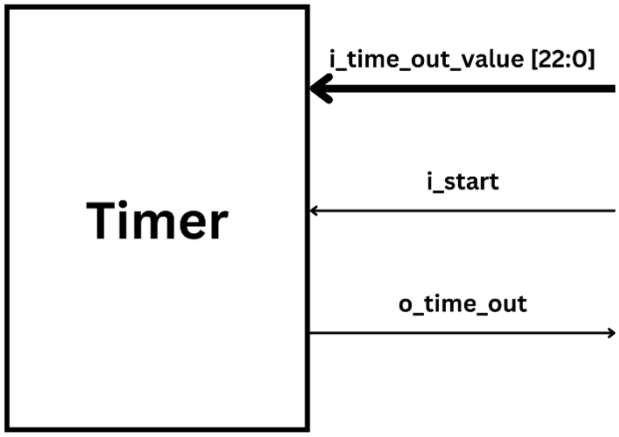


Figure 4.17: Timer Block Diagram

(Continued)

o_time_out	Output	Output to State machine Time Out = 1, when we reach the time needed Time Out = 0, when we still count the time needed

Detailed Operation

The main function of timer is to provide the time out signal to the State Machine according to the given timeout value. So basically, it takes timeout value from the State Machine, and when the start signal is asserted, it starts counting with each clock period until the timeout value is reached then it asserts the timeout signal to the state machine.

Evaluation

Bus function model is implemented for the stimulus generation of the timer module. Different time-out values are injected into the module, and it showed one bug as the default output value was '1'. After modifying the RTL, the bug was solved and all DUT outputs were compatible with the reference outputs.

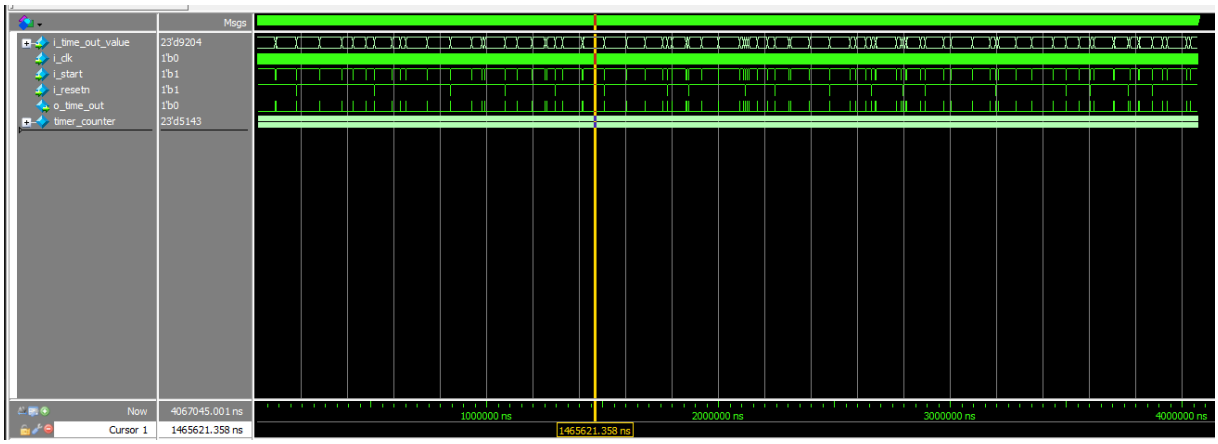


Figure 4.18: Timer Waveforms

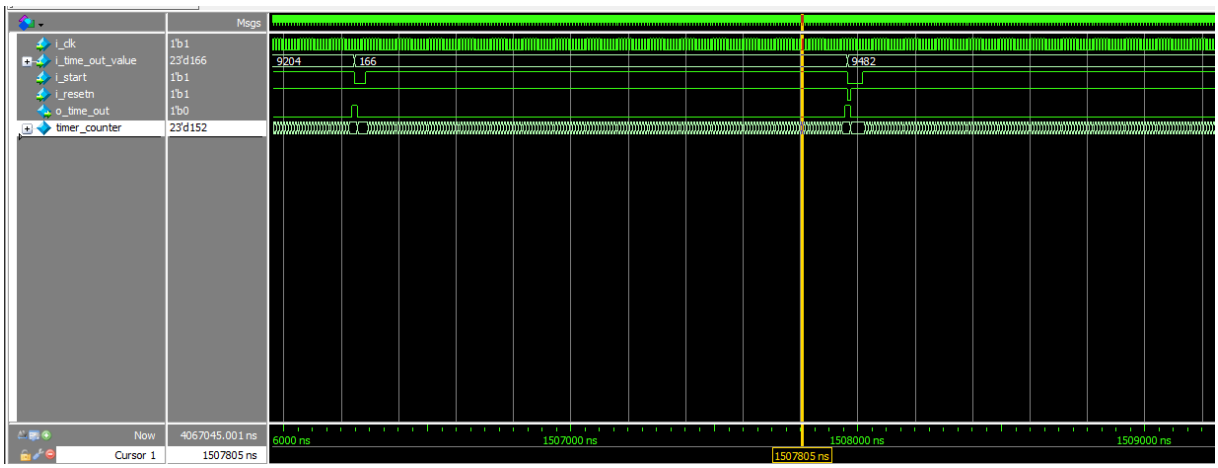


Figure 4.19: Timer Waveforms (zoomed-in)

```
# ----------------------------------------------------------------------------------------------------------------------------------------------------------------
# UVM_INFO Timer_env/Timer_scoreboard.svh(175) @ 4067045001: uvm_test_top.env_h.scoreboard_h [SCOREBOARD]
# At [4067035001 ps]:
# Output data is matching correctly with reference data in Scoreboard
# Reference Counter = 3345
# Output Time Out = 0, Reference Time Out = 0
# Number of Matches = 406703, Number of Mismatches = 0
# ----------------------------------------------------------------------------------------------------------------------------------------------------------------
#
# ------------------------------------------------------------------------------------------------------------------------------------------------------------------
# UVM_INFO Timer_env/Timer_subscriber.svh(97) @ 4067045001: uvm_test_top.env_h.subscriber_h [Timer_subscriber]
# Final Coverage: Cover Group Coverage = [100 %]
# ------------------------------------------------------------------------------------------------------------------------------------------------------------------
#
# --- UVM Report Summary ---
#
```

Figure 4.20: Timer Final Log

Covergroups

Name	Class Type	Coverage	Goal	% of Goal	Status	Included	Merge_instances	Get_inst_coverage
/Timer_pkg/Timer_subscriber		100.00...						
TYPE g1	Timer_subscriber	100.00...	100	100.00...		✓	auto(1)	
CVP g1::Input_Time	Timer_subscriber	100.00...	100	100.00...		✓		
bin Input_Time[0]		641	1	100.00...		✓		
bin Input_Time[1]		1835	1	100.00...		✓		
bin Input_Time[2]		2044	1	100.00...		✓		
bin Input_Time[3]		668	1	100.00...		✓		
bin Input_Time[4]		1791	1	100.00...		✓		
CVP g1::Output_Time	Timer_subscriber	100.00...	100	100.00...		✓		
bin auto[0]		406496.	1	100.00...		✓		
bin auto[1]		207	1	100.00...		✓		
CVP g1::Reset	Timer_subscriber	100.00...	100	100.00...		✓		
bin auto[0]		20	1	100.00...		✓		
bin auto[1]		406684.	1	100.00...		✓		

Figure 4.21: Timer Functional Coverage

4.1.4 OS Creator

Introduction

Design Description:

This module is responsible for creating the required Ordered-Sets (OS) to complete the training process. It serves as an interface between the LTSSM and the Tx. The Creator is composed of 3 sub-modules, one for the creation of Training Sequence (TS) OS, the second one for the creation of other OS, and the third for creating the logical idle symbols.

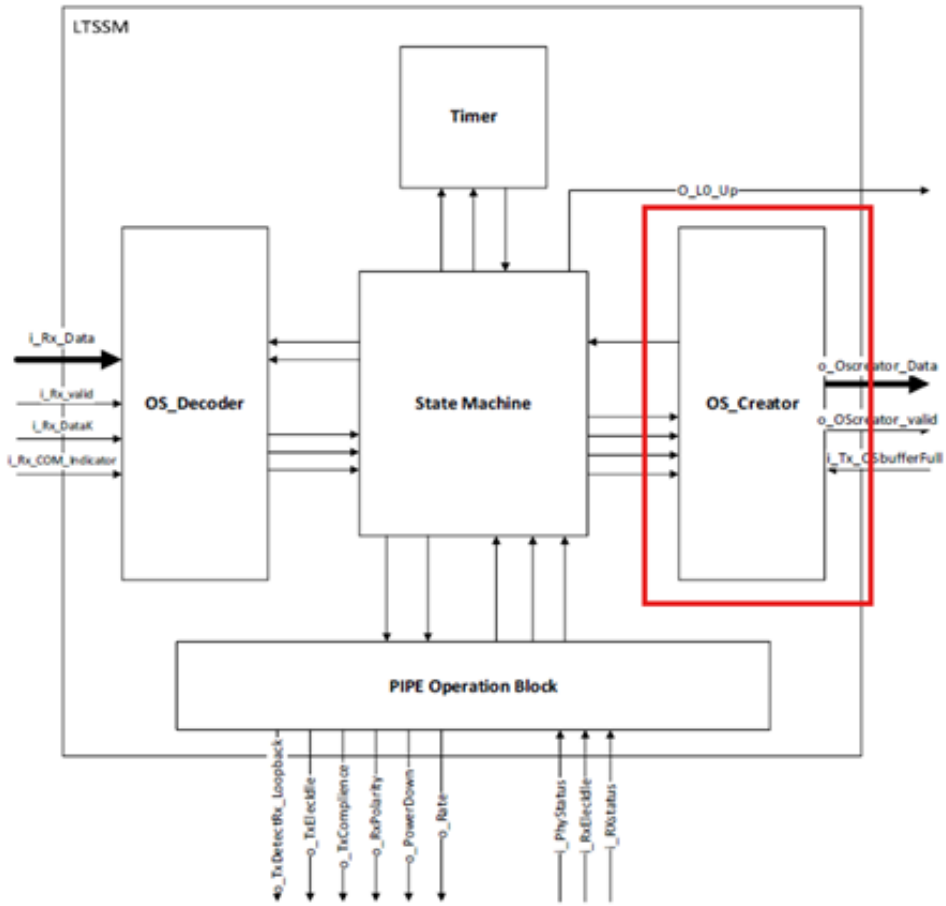


Figure 4.22: LTSSM Implementation – Creator Highlighted

Block I/O Signals:

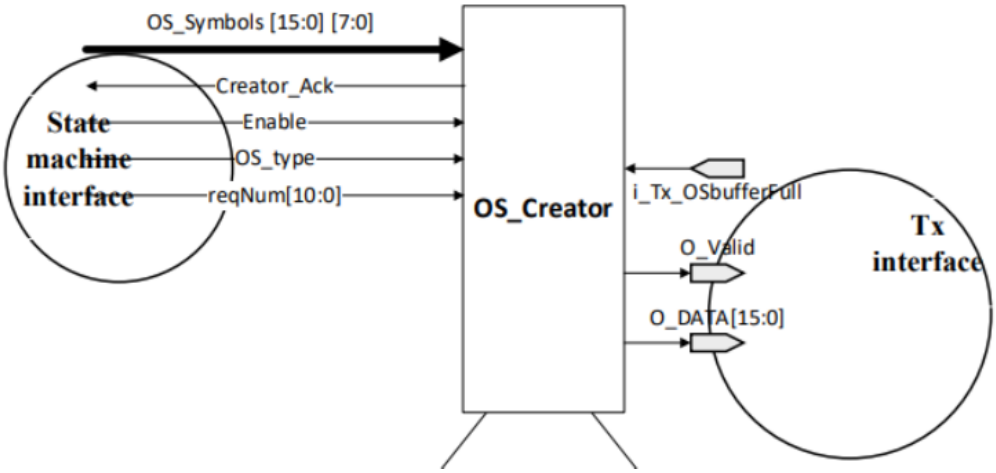


Figure 4.23: OS_Creator implementation

I/O Signals Description

Inputs:

Name	Width	Source	Description
i_clk	1 bit	Global	Clock signal for positive edge registers.
i_resetn	1 bit	Global	Synchronous low-level reset.
i_enable	1 bit	State Machine	Enable signal for the module.
i_Tx_OSbufferFull	1 bit	Tx	Indicates if the transmitter buffer is full or not. If full, the LTSSM needs to stop the transmission of Ordered-Sets data
i_LTSSM_stateChange	1 bit	State Machine	Indicates that a state change has occurred.
i_reset_TS_count	1 bit	State Machine	Resets the value of the TS counter.

Continued on next page

(Continued)

i_OSreqNum	11 bits	State Machine	Number of repetitions of the OS.
i_OScreatorTypes	2 bits	State Machine	Choose from the different creator OS types. 00:Other_OS, 01:TS OS, 10:IDLE OS, else:no output.
i_OScreatorSymbol0	8 bits	State Machine	Value of byte 0 of the OS. Examples: PAD, COM, FTS, SKP, etc.
i_OScreatorSymbol1	8 bits	State Machine	Value of byte 0 of the OS. Examples: PAD, COM, FTS, SKP, etc.
i_OScreatorSymbol2	8 bits	State Machine	Value of byte 0 of the OS. Examples: PAD, COM, FTS, SKP, etc.
i_OScreatorSymbol3	8 bits	State Machine	Value of byte 0 of the OS. Examples: PAD, COM, FTS, SKP, etc.
i_OScreatorSymbol4	8 bits	State Machine	Value of byte 0 of the OS. Examples: PAD, COM, FTS, SKP, etc.
i_OScreatorSymbol5	8 bits	State Machine	Value of byte 0 of the OS. Examples: PAD, COM, FTS, SKP, etc.
i_OScreatorSymbol6	8 bits	State Machine	Value of byte 0 of the OS. Examples: PAD, COM, FTS, SKP, etc.
i_OScreatorSymbol7	8 bits	State Machine	Value of byte 0 of the OS. Examples: PAD, COM, FTS, SKP, etc.
i_OScreatorSymbol8	8 bits	State Machine	Value of byte 0 of the OS. Examples: PAD, COM, FTS, SKP, etc.

Continued on next page

(Continued)

i_OScreatorSymbol9	8 bits	State Machine	Value of byte 0 of the OS. Examples: PAD, COM, FTS, SKP, etc.
i_OScreatorSymbol10	8 bits	State Machine	Value of byte 0 of the OS. Examples: PAD, COM, FTS, SKP, etc.
i_OScreatorSymbol11	8 bits	State Machine	Value of byte 0 of the OS. Examples: PAD, COM, FTS, SKP, etc.
i_OScreatorSymbol12	8 bits	State Machine	Value of byte 0 of the OS. Examples: PAD, COM, FTS, SKP, etc.
i_OScreatorSymbol13	8 bits	State Machine	Value of byte 0 of the OS. Examples: PAD, COM, FTS, SKP, etc.
i_OScreatorSymbol14	8 bits	State Machine	Value of byte 0 of the OS. Examples: PAD, COM, FTS, SKP, etc.
i_OScreatorSymbol15	8 bits	State Machine	Value of byte 0 of the OS. Examples: PAD, COM, FTS, SKP, etc.

Outputs:

Name	Width	Destination	Description
o_OScreator_Data	16 bits	Tx	Value of the OS data.
o_OScreator_valid	1 bit	Tx	Indicates the validity of the OS data.
o_OScreator_Ack	1 bit	State Machine	Indicates that the number of transmitted OS is equal to or more than the required number.

Detailed Operation

As shown in the figure below, the Creator consists of four modules. Based on the values of i_enable and i_OScreatorTypes, the demultiplexer enables only one of the other three modules at an instance. The TS Creator generates TS1 and TS2 OS required for link training and initialization. The Other OS Creator generates other OS such as SKP, FTS, and IDLE. The IDLE Creator generates the logical idle (8'h00) symbols. Each of these three modules generates a number of sequences or symbols according to the value of the i_OSreqNum signal. 16 i_OScreatorSymbol input signals, each with a byte width, are used to generate each OS from symbols defined in the State Machine. All 16 symbols are inputs to the TS Creator while only the first 4 symbols are inputs to the Other OS Creator. The IDLE Creator does not need any of these symbols.

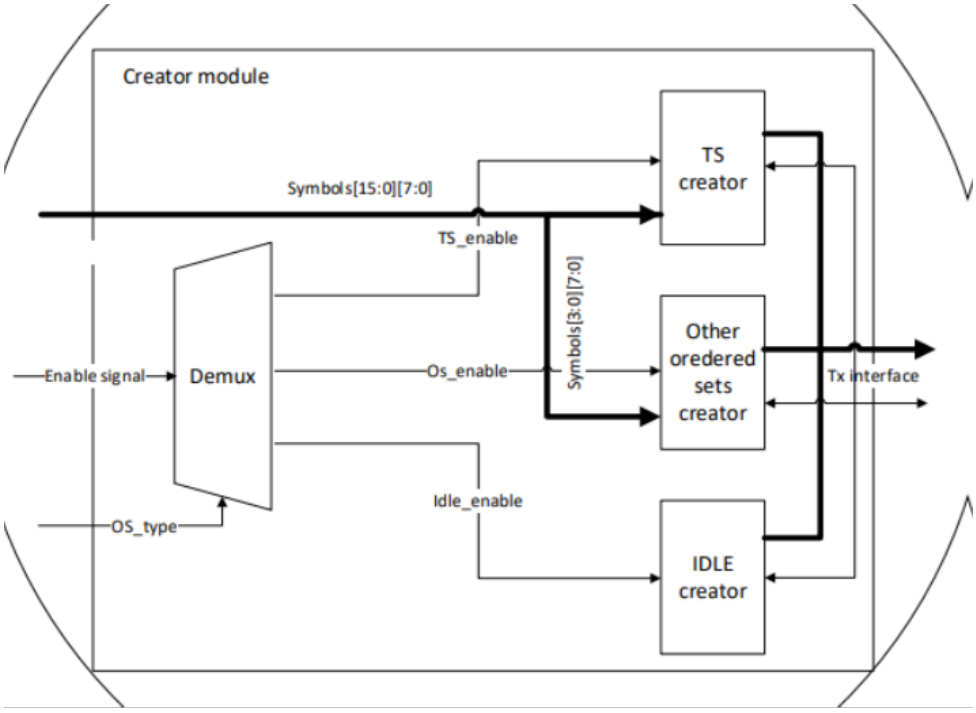


Figure 4.24: OS_Creator internal modules

Verification approach

In this block-level environment, 3 active agents are used to interact with the OS Creator Module through 3 interfaces combining all input and output signals of the Creator. The agents are:

1. Global
2. Transmitter (Tx)
3. FSM

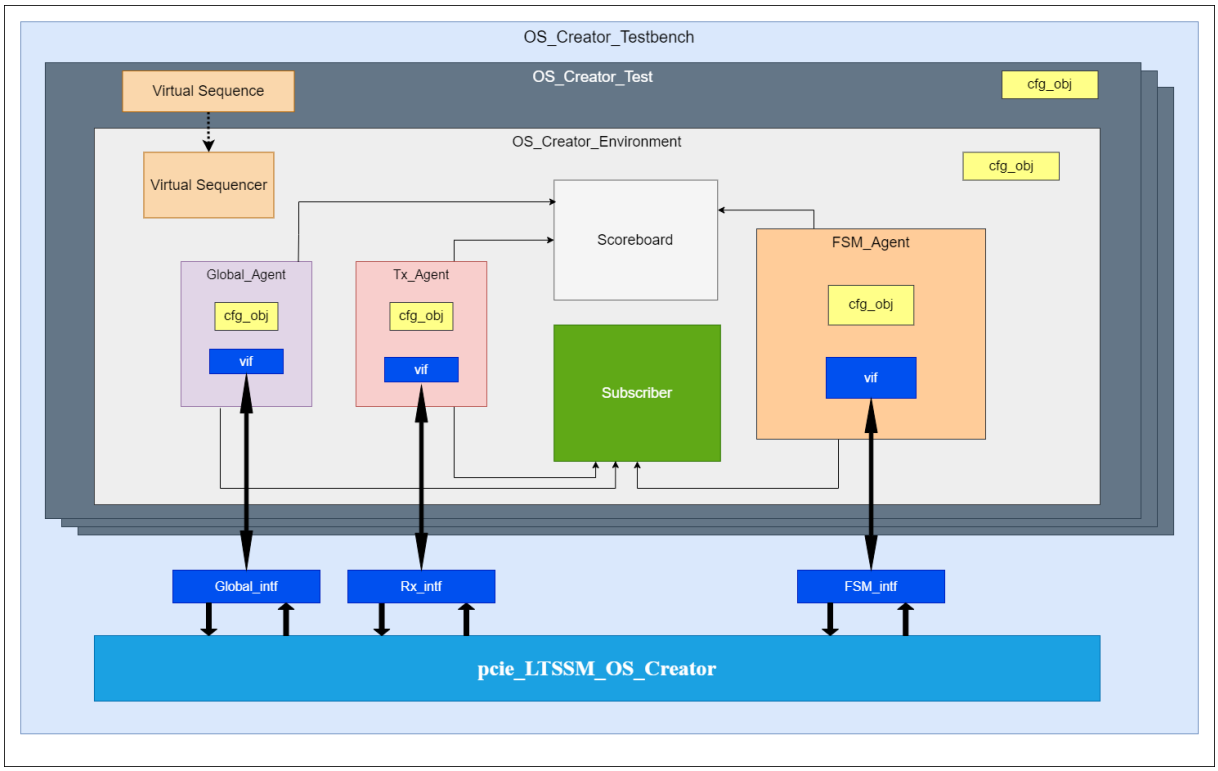


Figure 4.25: OS_Creator

Test Items and Sequences

Test items:

Test item	Description

Continued on next page

(Continued)

i_enable	0: disable the module from capturing the input OS symbols. 1: enable the module to capture the input OS symbols.
i_Tx_OSbufferFull	0: when Tx buffer is not full. 1: when Tx buffer is full.
i_LTSSM_stateChange	0: when no state change has occurred. 1: when a state change has occurred.
i_reset_TS_count	0: do nothing. 1: reset the counter.
i_OScreatorTypes	00: Other OS. 01: TS OS. 10: IDLE OS. else: no output.

Sequence Layer:
The following section will list some of the sequences that are used to make that creator do its operation.
N.B. These sequences might contain multiple clock cycles which will be denoted by xk where k is the number of cycles.

Sequence Name	Sequence	Description
Reset_seq	i_resetn = 1'b0	This sequence represents the reset procedure for the module.

Continued on next page

(Continued)

STS1	i_enable = 1′b01 i_Tx_OSbufferFull = 1′b0 i_LTSSM_stateChange = 1′b0 i_reset_TS_count = 1′b0 i_OSreqNum = 11′d1 i_OScreatorTypes = 2′b01 i_OScreatorSymbol0 = 8′hBC (COM) i_OScreatorSymbol1 = PAD || Link# i_OScreatorSymbol2 = PAD || Lane# i_OScreatorSymbol3 = N_FTS i_OScreatorSymbol4 = DR_ID i_OScreatorSymbol5 = LIDLE i_OScreatorSymbol6...15 = TS1_ID	This sequence represents sending a single TS1 OS
MTS1	Same as STS1 but i_OSreqNum = 11′dk where k is the number of repetitions	This sequence is an example of multiple repetitions of TS1 OS.
STS2	Same as STS1 but i_OScreatorSymbol6...15 = TS2_ID	This sequence represents sending a single TS2 OS
MTS2	Same as STS but i_OSreqNum = 11′dk where k is the number of repetitions	This sequence is an example of multiple repetitions of TS2 OS.
SSKP	i_enable = 1′b01 i_Tx_OSbufferFull = 1′b0 i_LTSSM_stateChange = 1′b0 i_reset_TS_count = 1′b0 i_OSreqNum = 11′d1 i_OScreatorTypes = 2′b00 i_OScreatorSymbol0 = 8′hBC (COM) i_OScreatorSymbol1 = SKP i_OScreatorSymbol2 = SKP i_OScreatorSymbol3 = SKP	This sequence represents sending a single SKP OS
MSKP	Same as SSKP but i_OSreqNum = 11′dk where k is the number of repetitions	This sequence is an example of multiple repetitions of SKP OS
SFTS	Same as SKP but i_OScreatorSymbol1...3 = FTS	This sequence represents sending a single FTS OS

Continued on next page

(Continued)

MFTS	Same as FTS but i_OSreqNum = 11'dk where k is the number of repetitions	This sequence is an example of multiple repetitions of FTS OS
SLIDLE	i_enable = 1'b01 i_Tx_OSbufferFull = 1'b0 i_LTSSM_stateChange = 1'b0 i_reset_TS_count = 1'b0 i_OSreqNum = 11'd1 i_OScreatorTypes = 2'b10	This sequence represents sending a single LIDLE OS
MLIDLE	Same as LIDLE but i_OSreqNum = 11'dk where k is the number of repetitions	This sequence is an example of multiple repetitions of LIDLE OS

Many combinations are done using the above sequences in verifying the OS Creator.

Coverage Collection

Functional coverage collection will be performed using the subscriber class. The table below shows some examples of the coverage bins that are implemented in the code.

Signal	Coverage Bins Description
i_enable	0: no enable, 1: enable
LTSSM_stateChange	1: State change, 0: no state change
i_reset_TS_count	1: reset TS counter, 0: no reset TS Counter
i_OScreatorTypes	00: other TS, 01: TS_OS, 10: IDLE_OS
i_OScreatorSymbol1	Separate bins for: PAD, Lane, SKP, FTS
i_OScreatorSymbol11	Separate bins for: TS1_ID, TS2_ID

Evaluation

Detected Bugs

Data errors:

- Sending some OS, resetting, and then sending another OS leads to an incorrect first OS (all zeros) even with disabling the module, resetting TS count, and asserting LTSSM state change.

- If buffer full is indicated during sending multiple OS, the data out is not zeros as it should be
- Sending multiple OS, the data out is a repeated OS even after it is finished and the acknowledge is asserted. The output should be all zeros instead.

Ack errors:

- For sending LIDLE, having i_OSreqNum = 1 is the same as i_OSreqNum = 2 as we already send 2 LIDLE symbols in a single transaction, same goes for any 2 consecutive numbers (3 and 4, 5 and 6, ...).

Valid errors:

- The valid signal remains asserted even after the OS is finished. It should be de-asserted directly after the OS is finished.

Resolving Bugs

Most of the previously mentioned bugs have been resolved in the RTL, except for a few bugs which required a big change to the whole design so we couldn't solve them. A comprehensive simulation was conducted using an extensive sequence after solving most of the bugs and as shown in figure 4.26 the errors were reduced from 1240 errors to 89 errors.

```
#
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO : 6248
# UVM_WARNING :    0
# UVM_ERROR :   89
# UVM_FATAL :    0
# ** Report counts by id
# [RNTST]     1
# [TEST_DONE]     1
# [creator_scoreboard]    90
# [creator_scoreboard - DUT]  1561
# [creator_scoreboard - REF]  1561
# [creator_subscriber]  3123
# ** Note: $finish    : C:/questasim64_10.7c/win64/../verilog_src/uvm-1.1d/src/base/uvm_root.svh(430)
#    Time: 15615001 ps  Iteration: 65  Instance: /creator_tb
# Break in Task uvm_pkg/uvm_root::run_test at C:/questasim64_10.7c/win64/../verilog_src/uvm-1.1d/src/base/uvm_root.svh line 430
```

Figure 4.26: Creator Final Log

Code Coverage

As illustrated in figure 4.27, the code coverage achieved is 92.2%, which exceeds our target of 90%.

Coverage Summary By Instance:

Scope	TOTAL	Statement	Branch	FEC Condition	Toggle
TOTAL	92.19	94.02	87.75	92.30	94.69
creator_DUT	81.09	83.87	62.50	--	96.91
TScreator	96.27	96.49	90.47	100.00	98.11
otherOScreator	95.22	100.00	100.00	83.33	97.56
LIdleCreator	80.83	93.75	80.00	--	68.75

Figure 4.27: Creator Code Coverage

Functional Coverage

We achieved 100% functional coverage as we hit all the cover points that were mentioned above in the coverage section.

Covergroups

Name	Class Type	Coverage	Goal	% of Goal	Status	Included	Merge_ins
/creator_pkg/creator_subscriber		100.00%					
TYPE cvr_grp	creator_subscriber	100.00%	100	100.00...		✓	
CVP cvr_grp::i_enable_cp	creator_subscriber	100.00%	100	100.00...		✓	
CVP cvr_grp::i_LTSSM_stateChange_cp	creator_subscriber	100.00%	100	100.00...		✓	
CVP cvr_grp::i_reset_TS_count_cp	creator_subscriber	100.00%	100	100.00...		✓	
CVP cvr_grp::i_OSreqNum_cp	creator_subscriber	100.00%	100	100.00...		✓	
CVP cvr_grp::i_OScreatorTypes_cp	creator_subscriber	100.00%	100	100.00...		✓	
CVP cvr_grp::i_OScreatorSymbol0_cp	creator_subscriber	100.00%	100	100.00...		✓	
CVP cvr_grp::i_OScreatorSymbol1_cp	creator_subscriber	100.00%	100	100.00...		✓	
CVP cvr_grp::i_OScreatorSymbol2_cp	creator_subscriber	100.00%	100	100.00...		✓	
CVP cvr_grp::i_OScreatorSymbol3_cp	creator_subscriber	100.00%	100	100.00...		✓	
CVP cvr_grp::i_OScreatorSymbol4_cp	creator_subscriber	100.00%	100	100.00...		✓	
CVP cvr_grp::i_OScreatorSymbol5_cp	creator_subscriber	100.00%	100	100.00...		✓	
CVP cvr_grp::i_OScreatorSymbol6_cp	creator_subscriber	100.00%	100	100.00...		✓	
CVP cvr_grp::i_OScreatorSymbol7_cp	creator_subscriber	100.00%	100	100.00...		✓	
CVP cvr_grp::i_OScreatorSymbol8_cp	creator_subscriber	100.00%	100	100.00...		✓	
CVP cvr_grp::i_OScreatorSymbol9_cp	creator_subscriber	100.00%	100	100.00...		✓	
CVP cvr_grp::i_OScreatorSymbol10_cp	creator_subscriber	100.00%	100	100.00...		✓	
CVP cvr_grp::i_OScreatorSymbol11_cp	creator_subscriber	100.00%	100	100.00...		✓	
CVP cvr_grp::i_OScreatorSymbol12_cp	creator_subscriber	100.00%	100	100.00...		✓	
CVP cvr_grp::i_OScreatorSymbol13_cp	creator_subscriber	100.00%	100	100.00...		✓	
CVP cvr_grp::i_OScreatorSymbol14_cp	creator_subscriber	100.00%	100	100.00...		✓	
CVP cvr_grp::i_OScreatorSymbol15_cp	creator_subscriber	100.00%	100	100.00...		✓	
CVP cvr_grp::i_Tx_OSbufferFull_cp	creator_subscriber	100.00%	100	100.00...		✓	
CVP cvr_grp::o_OScreator_Ack_cp	creator_subscriber	100.00%	100	100.00...		✓	
CVP cvr_grp::o_OScreator_Data_cp	creator_subscriber	100.00%	100	100.00...		✓	
CVP cvr_grp::o_OScreator_valid_cp	creator_subscriber	100.00%	100	100.00...		✓	

Wave | Analysis | Covergroups | uvm_root.svh

Figure 4.28: Creator functional coverage

4.1.5 State Machine

Introduction

Design Description:

This IP receives data from the PCIe Rx and the state machine. Based on the data's content and the current state, it controls the state transitions during link training and initialization process.

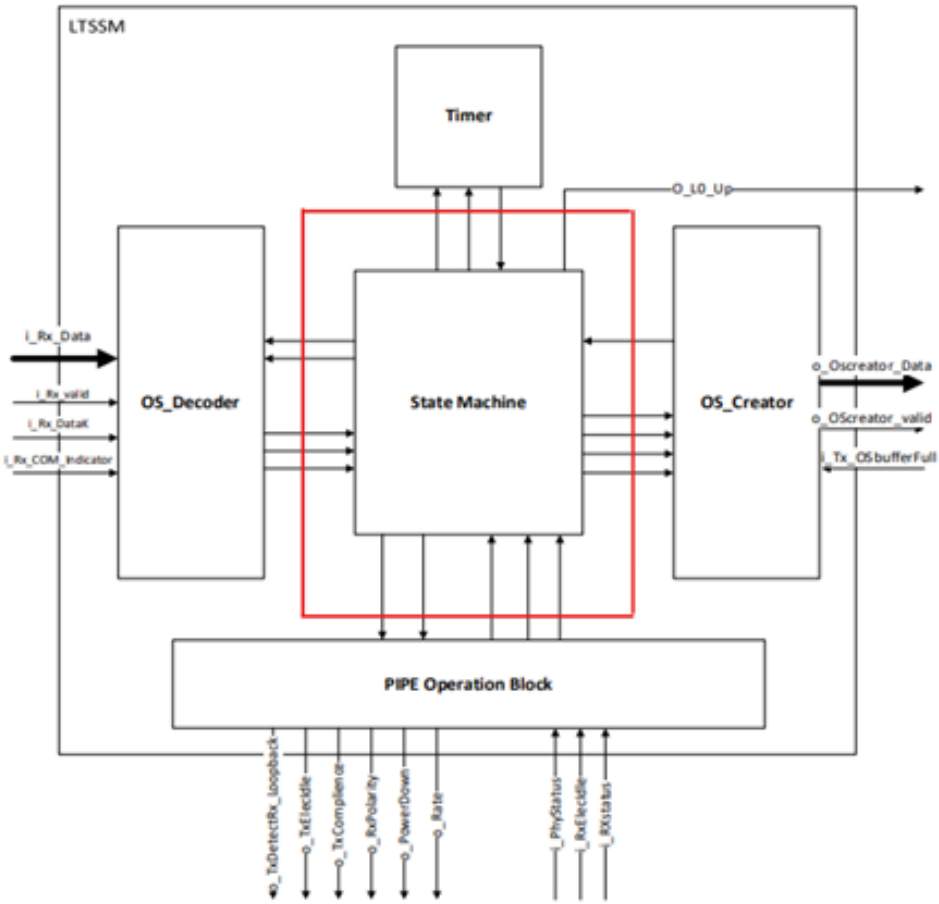


Figure 4.29: LTSSM System - FSM Highlighted

Block I/O Signals:

s **Interaction with Other Modules:**

The State machine functions as follows:

- Take the required signals from PIPE and Decoder for controlling the transitions between the states in the State machine module.
- Out the number of TS need to be sent to Order set creator block.
- Out the LTSSM states to PIPE interface, and get back the signals required for controlling the transitions between the states in the LTSSM block.
- After finishing the LTSSM training, L0_up signal indicates the normal operation.

I/O Signals Description

Inputs:

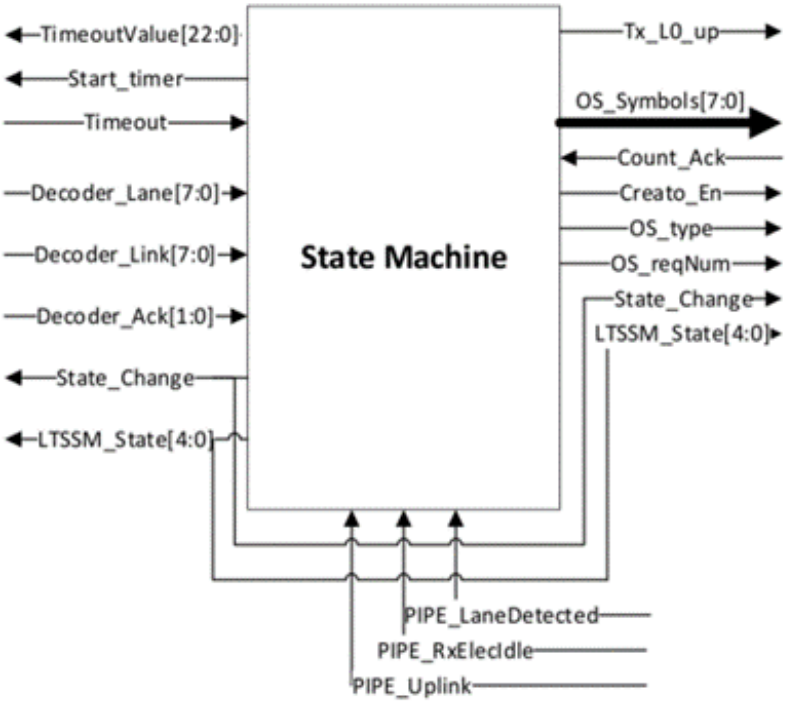


Figure 4.30: FSM Block Diagram

Name	Width	Source	Description
i_PCLK	1 bit	Global	Global clock
i_resetn	1 bit	Global	Global reset
i_timer time-out	1 bit	Timer	Input from timer. Timeout =1, time Timeout value Timeout=0, otherwise.
i_LTSSM Up-Down	1 bit	Module parameter	Parameter to indicate if the PCIe module is downstream or upstream 1 = Downstream 0 = Upstream
i_LTSSM Up-Link	1 bit	PIPE	Indicates bit and symbol lock, ready to L0 state
i_OScreator Ack	1 bit	Order set creator	To tell the State Machine the required number of ordered sets had been transmitted OScreator_Ack = 1, Transmitted OS OS_reqNum OScreator_Ack = 0, otherwise.

Continued on next page

(Continued)

i_PHY laneDetected	1 bit	PIPE	indicate successful detection of a receiver to LTSSM
i_PHY_Rx ElecIdle	1 bit	PIPE	Mirror the i_PHY_RxElecIdle coming from PHY to LTSSM
i_OSdecoder Ack	2 bits	decoder	Acknowledgement signal on the number of TS order sets received to Control the transition between states.
i_OSdecoder Link	8 bits	decoder	The link number decoded from TS1 and TS2
i_OSdecoder Lane	8 bits	decoder	The lane number decoded from TS1 and TS2

Outputs:

Name	Width	Destination	Description
o_timer timeoutValue	23 bits	Timer	Time needed by timer to timeout
o_timer_start	1 bit	Timer	start count the time: timer_start=1, Start count timer_start =0, Stop count and restart all the time data (when we start again, we will start from 0)
o_OScreator Symbol0	8 bits	Order set creator	The order set symbol 0 data
o_OScreator Symbol1	8 bits	Order set creator	The order set symbol 1 data
o_OScreator Symbol2	8 bits	Order set creator	The order set symbol 2 data
o_OScreator Symbol3	8 bits	Order set creator	The order set symbol 3 data
o_OScreator Symbol4	8 bits	Order set creator	The order set symbol 4 data
o_OScreator Symbol5	8 bits	Order set creator	The order set symbol 5 data
o_OScreator Symbol6	8 bits	Order set creator	The order set symbol 6 data
o_OScreator Symbol7	8 bits	Order set creator	The order set symbol 7 data
o_OScreator Symbol8	8 bits	Order set creator	The order set symbol 8 data
o_OScreator Symbol9	8 bits	Order set creator	The order set symbol 9 data
o_OScreator Symbol10	8 bits	Order set creator	The order set symbol 10 data
o_OScreator Symbol11	8 bits	Order set creator	The order set symbol 11 data

Continued on next page

(Continued)

o_OScreator Symbol12	8 bits	Order set creator	The order set symbol 12 data
o_OScreator Symbol3	8 bits	Order set creator	The order set symbol 13 data
o_OScreator Symbol14	8 bits	Order set creator	The order set symbol 14 data
o_OScreator Symbol15	8 bits	Order set creator	The order set symbol 15 data
o_OScreator enable	1 bit	Order set creator	To enable the creator to send the order set
o_OScreator types	2 bits	Order set creator	Choose between TS creator or other order set creator.
o_OScreator resetTScounter	1 bit	Order set creator	Restart the counting of the transmission of the order set
o_LTSSM_state	5 bits	PIPE decoder	To inform the decoder of the current state
o_LTSSM state_Change	1 bit	decoder	to inform the decoder of the change in the state
o_LTSSM_L0 up	1 bit	Transmitter	To give the transmitter block permission to work in normal operation. o_LTSSM_L0_UP=1, indicate the L0 state.
o_OScreator reqNum	11 bits	Order set creator	Number of repetitions of the order set.

Detailed Operation

The State machine module is responsible for determining the content of the training sets sent to the transmitter. The structure of the training sets is shown below. There are two flavors of training sets, TS1 and TS2, the two flavors are used as a handshaking method to synchronize between the downstream and upstream ports of the link. Each training set consists of 16 8-bit symbols sent in series.
The LTSSM consists mainly of 11 top-level states: Detect, Polling, Configuration, Recovery, L0, L0s, L1, L2, hot reset, Loopback, and Disable. but in our design, we decide to implement only 5 states: Detect, Polling, Configuration, Recovery, and L0. The normal flow of link training is Detect → Polling → Configuration → L0. The FSM module

controls the transitions between the states based on time-out conditions, acknowledgment of the number of transmitted ordered sets from the OS Creator module, and acknowledgment of the number of received ordered sets from the OS Decoder module.

Verification approach

In this block-level environment, six active agents are used to interact with the State Machine Module through six interfaces combining all input and output signals of the State Machine. The agents are:

1. Global
2. Transmitter (Tx)
3. Timer
4. OS Decoder
5. OS Creator
6. PIPE Operation Block

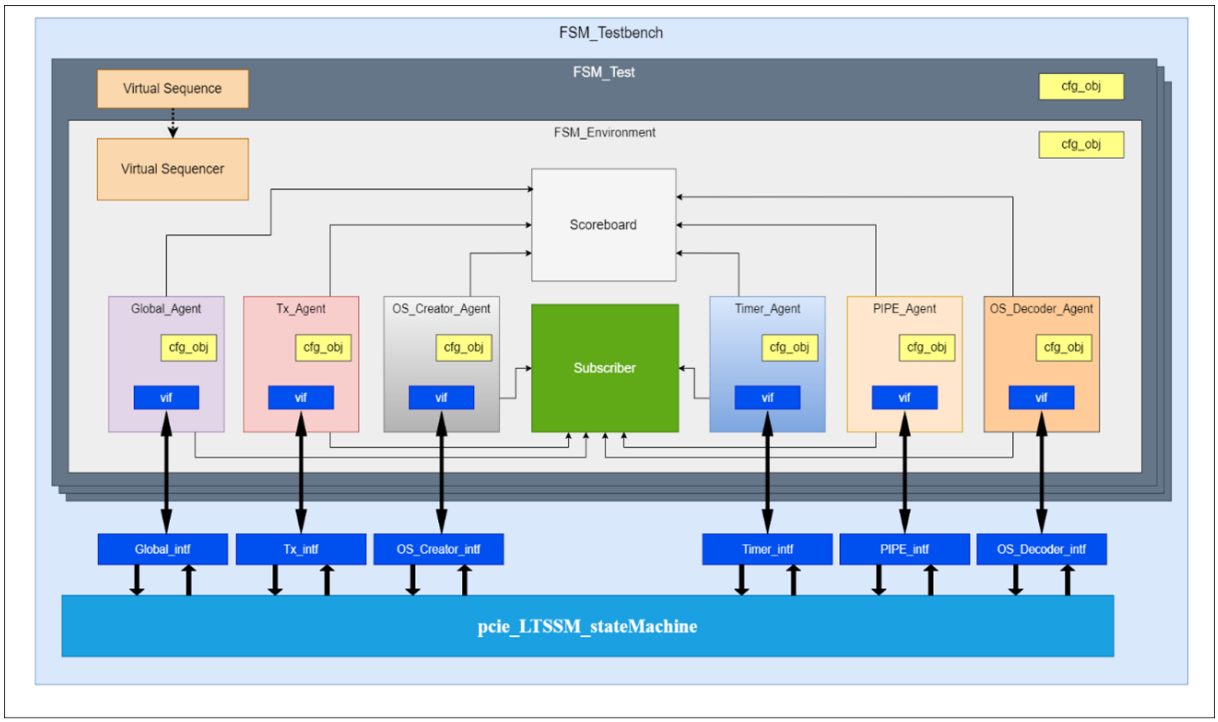


Figure 4.31: FSM UVM Hierarchy

Test Items and Sequences

Test Items:

- Timeout (in any state):
  1. Time is greater than or equals Timeout value, i_timer_timeout = 1.
  2. Time is less than Timeout value, i_timer_timeout = 0.
- Type of the device:
  1. Upstream, i_LTSSM_UpDown = 0.
  2. Downstream, i_LTSSM_UpDown = 1.
- Receiver detection:
  1. Succeeded, i_PHY_laneDetected = 1.
  2. Failed, i_PHY_laneDetected = 0.
- Receiver in electrical Idle:
  1. Yes, i_PHY_RxElecIdle = 1.
  2. No, i_PHY_RxElecIdle = 0.
- Acknowledgement signals (i_OSdecoder_Ack & i_OScreator_Ack):
  1. Both asserted.
  2. Only Decoder asserted.
  3. Only Creator asserted.
- State of operation:
  1. Detect_Quiet.
  2. Detect_Active.
  3. Polling_Active.
  4. Polling_configuration.
  5. Config_Linkwidth_start.
  6. Config_Linkwidth_Accept.
  7. Config_Lanenum_wait.
  8. Config_Lanenum_Accept.
  9. Config_complete.
  10. Config_idle.
  11. L0.

Test Cases:

Test Name	Sequences	Description
Test 01	Reset, 1 clk cycle, Receiver out of electrical Idle, 1 clk cycle, Detected receiver, 1 clk cycle, (Both Ack signals asserted i_LTSSM_UpDown = 0, 1 clk cycle) repeated	Normal flow of link training of an upstream device (Detect, Polling, Configuration, L0).
Test 02	Reset, 1 clk cycle, Receiver out of electrical Idle, 1 clk cycle, Detected receiver, 1 clk cycle, (Both Ack signals asserted i_LTSSM_UpDown = 1, 1 clk cycle) repeated	Normal flow of link training of a downstream device.
Test 03	Reset, 1 clk cycle, Receiver in electrical Idle (wait till timeout), 1 clk cycle, No detected receiver.	Failed detection of a receiver on the other side of the link.
Test 04	Reset, 1 clk cycle, Receiver out of electrical Idle, 1 clk cycle, Detected receiver, 1 clk cycle, Only Decoder asserted Ack (till timeout).	States looping between Detect and Polling states due to not receiving acknowledgement from the Creator in Polling_Active state.
Test 05	Reset, 1 clk cycle, Receiver out of electrical Idle, 1 clk cycle, Detected receiver, 1 clk cycle, Both Ack signals asserted, 1 clk cycle, Only Decoder asserted Ack (till timeout).	States looping between Detect and Polling states due to not receiving acknowledgement from the Creator in Polling_Config state.
Test 06	Reset, 1 clk cycle, Receiver in electrical Idle (wait till timeout), 1 clk cycle, Detected receiver, 1 clk cycle, Both Ack signals asserted, 1 clk cycle, Only Creator asserted Ack (till timeout).	Returning from Config_Linkwidth_start to Detect due to not receiving acknowledgement from the Decoder.

Continued on next page

(Continued)

Test 07	Reset, 1 clk cycle, Receiver out of electrical Idle, 1 clk cycle, Detected receiver, 1 clk cycle, (Both Ack signals asserted, 1 clk cycle) repeated, Only Creator asserted Ack (till timeout).	Returning from Config_Lanenum_Accept to Detect due to not receiving acknowledgement from the Decoder.
Test 08	Reset, 1 clk cycle, Receiver out of electrical Idle, 1 clk cycle, Detected receiver, 1 clk cycle, (Both Ack signals asserted, 1 clk cycle) repeated, Only Creator asserted Ack (till timeout).	Returning from Config_idle to Detect due to not receiving second acknowledgement from the Decoder.
Test 09	Sequences of Test 01, (wait till timeout) repeated.	Normal flow, then periodic transmission of SKP OS.

Coverage Collection

Functional coverage collection will be performed using the subscriber class. The table below shows the coverage bins needed for coverage collection.

Signal	Coverage Bins Description
i_timer_timeout	Transition from 0 to 1 to 0.
i_LTSSM_UpDown	Have the values 0 and 1.
i_PHY_laneDetected	Have the values 0 and 1.
i_OScreator_Ack i_OSdecoder_Ack	Cross bins covering all possible combinations when i_OSdecoder_Ack = 2'b01 or 2'b10.
i_timer_timeout i_PHY_RxElecIdle	Cross bin covering the case when i_timer_timeout = 1 i_PHY_RxElecIdle = 0.
i_OSdecoder_Link i_OSdecoder_Lane	Cross bins covering the cases: [(PAD, PAD) - (Non-PAD, PAD) - (Non-PAD, Non-PAD)].
o_OScreator_enable o_OScreator_types	Cross bins covering all possible combinations when o_OScreator_enable = 1.
o_OScreator_resetTScounter o_LTSSM_state	Cross bins covering all possible combinations when the state is Config_complete or Config_idle.
o_LTSSM_L0_up	Has the values 0 and 1.
o_OScreator_Symbol2	Have the values: PAD - Non-PAD - SKP - 0.

Continued on next page

(Continued)

o_OScreator_Symbol15	Have the values: TS1 - TS2 - 0.
o_LTSSM_state	Have all possible values. Transitions through normal flow. Transitions due to timeout.

Code coverage will be performed automatically by the tool. Some exclusions are added to exclude the toggling behaviors of some signals or some bits in signals from the code coverage computations. For example, toggle coverage of the fifth bit of any signal describing a state is excluded, as only 11 states are supported, they can be represented by only four bits and the last bit will always be zero.

Evaluation

The module is found to be compatible with the reference model throughout the different test scenarios (downstream device normal states flow – upstream device normal states flow – time-out conditions in each state – random combinations of time-out and acknowledgement signals).

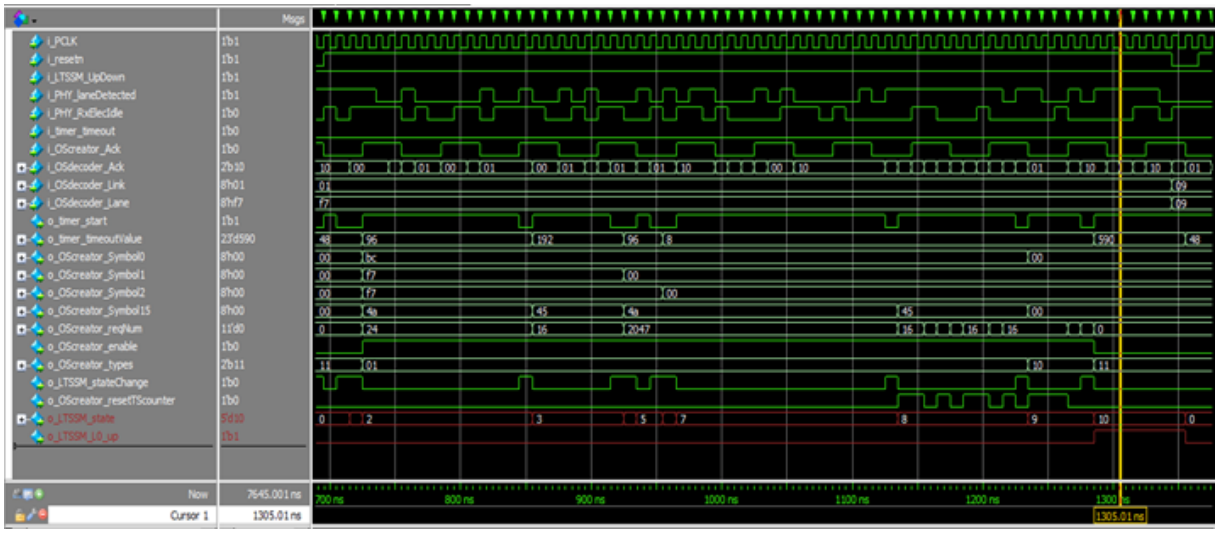


Figure 4.32: Downstream Device States Flow from Detect Quiet to L0

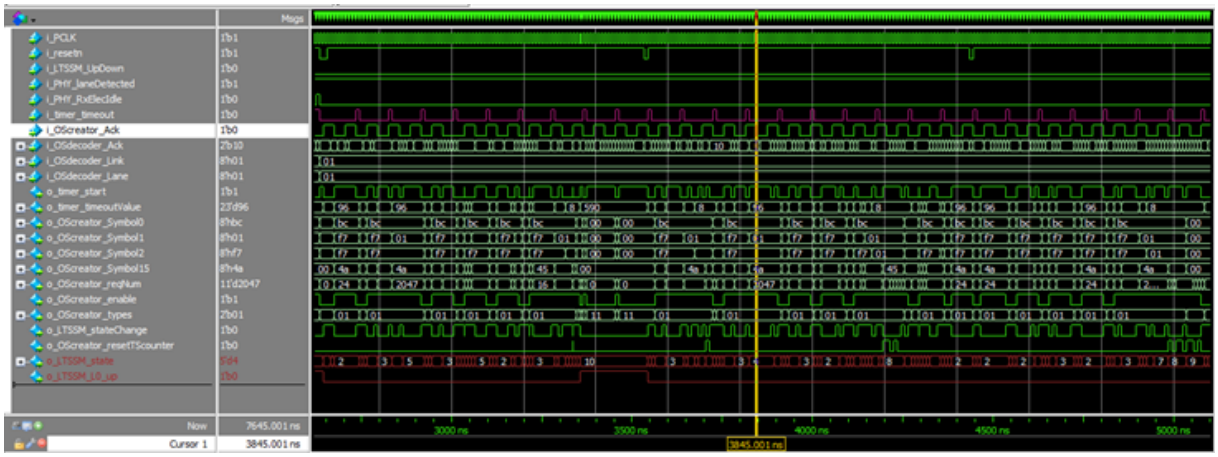


Figure 4.33: Time-out condition in each state

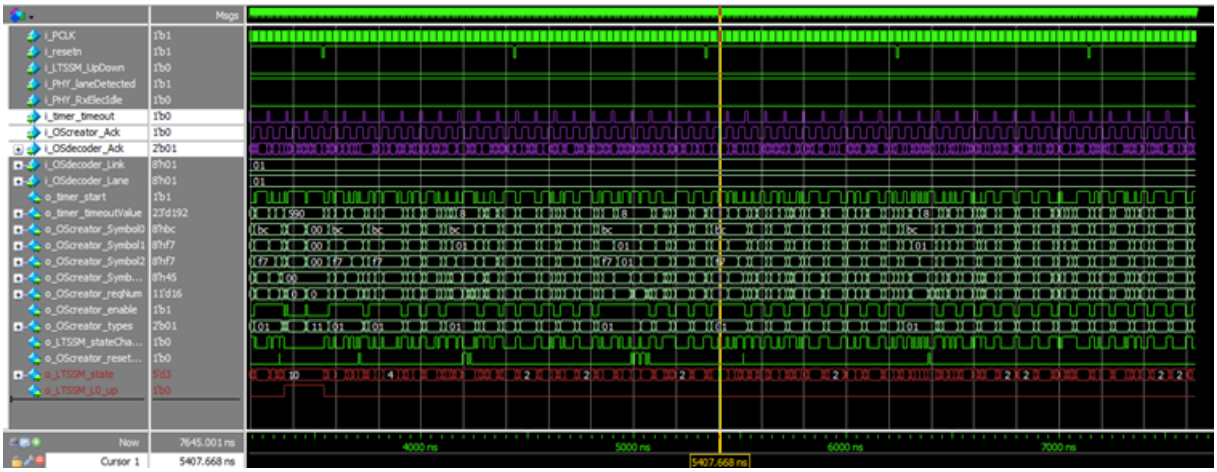


Figure 4.34: Random combinations of Ack and time-out signals

```
# ----------------------------------------------------------------------------------------------------------------------------------
# UVM_INFO FSM_env/FSM_scoreboard.svh(636) @ 7645001: uvm_test_top.env_h.scoreboard_h [FSM Scoreboard]
# At [7645001 ps]Simulation summary: 763 matching, 0 mismatching
#
# ----------------------------------------------------------------------------------------------------------------------------------
# UVM_INFO FSM_env/FSM_subscriber.svh(247) @ 7645001: uvm_test_top.env_h.subscriber_h [FSM_subscriber]
# Final Coverage: Cover Group Coverage = [100 %]
#
#
# ----------------------------------------------------------------------------------------------------------------------------------
#
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO : 1530
# UVM_WARNING :    0
# UVM_ERROR :    0
# UVM_FATAL :    0
```

Figure 4.35: FSM Final Log

Coverage Results:

Running the different test cases, 100% functional coverage and 99.3% code coverage are achieved.

The figures below show the code and functional coverage.

Total Coverage:					99.36%	99.33%
Coverage Type	Bins	Hits	Misses	Weight	% Hit	Coverage
Statements	392	392	0	1	100.00%	100.00%
Branches	57	57	0	1	100.00%	100.00%
FEC Conditions	34	34	0	1	100.00%	100.00%
Toggles	120	116	4	1	96.66%	96.66%
FSMs	31	31	0	1	100.00%	100.00%
States	11	11	0	1	100.00%	100.00%
Transitions	20	20	0	1	100.00%	100.00%

Figure 4.36: FSM Code Coverage Details

Search:

Crosses	Total Bins	Hits	Misses	Hit %	Goal %	Coverage %
Both_Ack	2	2	0	100.00%	100.00%	100.00%
enable_types	3	3	0	100.00%	100.00%	100.00%
Link_Lane	3	3	0	100.00%	100.00%	100.00%
Reset_state	4	4	0	100.00%	100.00%	100.00%
State_ReqNum	8	8	0	100.00%	100.00%	100.00%
Time_DetectQuiet	1	1	0	100.00%	100.00%	100.00%

Figure 4.37: FSM Cross Bins Coverage

Search:

CoverPoints	Total Bins	Hits	Misses	Hit %	Goal %	Coverage %
Creator_Ack	1	1	0	100.00%	100.00%	100.00%
Creator_enable	2	2	0	100.00%	100.00%	100.00%
Creator_ReqNum	6	6	0	100.00%	100.00%	100.00%
Creator_types	4	4	0	100.00%	100.00%	100.00%
Decoder_Ack	2	2	0	100.00%	100.00%	100.00%
L0_up	2	2	0	100.00%	100.00%	100.00%
Lane_num	2	2	0	100.00%	100.00%	100.00%
Link_num	2	2	0	100.00%	100.00%	100.00%
LTSSM_state	21	21	0	100.00%	100.00%	100.00%
Out_Timeout	5	5	0	100.00%	100.00%	100.00%
Phy_RxIdle	2	2	0	100.00%	100.00%	100.00%
Reset	2	2	0	100.00%	100.00%	100.00%
Reset_TSCounter	2	2	0	100.00%	100.00%	100.00%
Symbol15	3	3	0	100.00%	100.00%	100.00%
Symbol_2	4	4	0	100.00%	100.00%	100.00%
Timer_Timeout_Transition	1	1	0	100.00%	100.00%	100.00%
Timer_Timeout_Value	2	2	0	100.00%	100.00%	100.00%
Up_Down	2	2	0	100.00%	100.00%	100.00%

Figure 4.38: FSM Coverpoints

4.2 Transmitter Blocks

4.2.1 Tx Buffers Collection

Introduction:

The buffers module collection is a group of four modules in the PCIe transmitter IP that have been grouped together for verification purposes. The position of the modules in the transmitter architecture is shown below in figure 4.40. They are responsible for receiving and splitting the incoming packet from the DataLink layer.

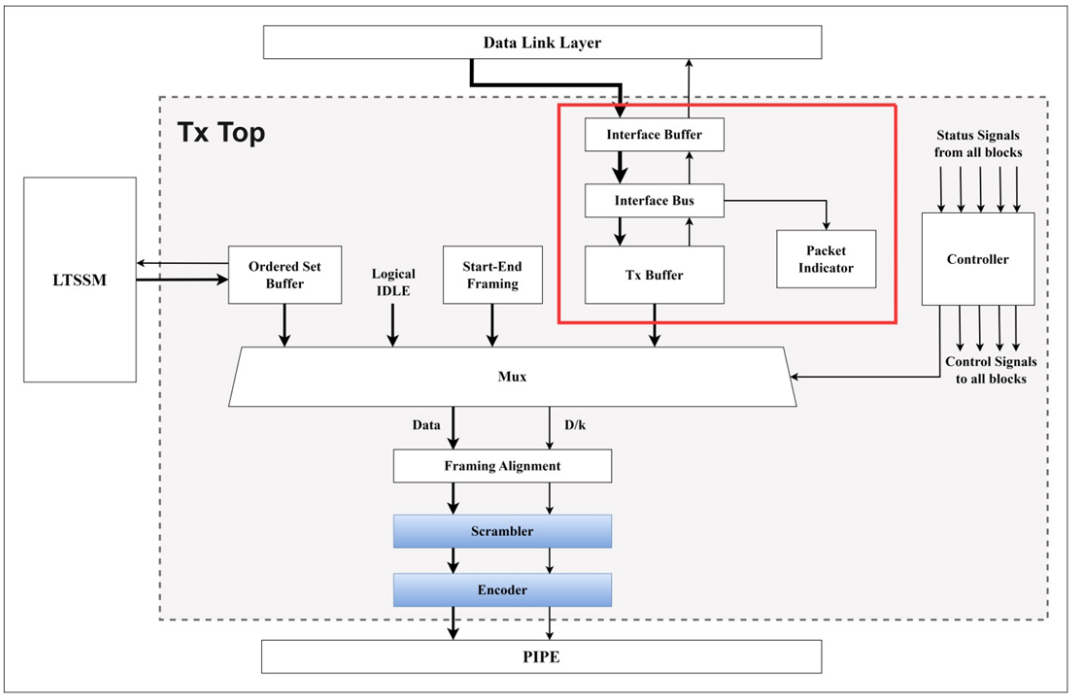


Figure 4.39: Position of the buffers collection modules

I/O Signals Description:

The figure below shows the signal block diagram of the buffers collection:
The table below shows a description of the inputs and output signals of the buffers collection.

Name	Width	Source	Description
i_clk	1 bit	Global	Clock signal for positive edge registers.
i_reset_n	1 bit	Global	Synchronous low-level reset.

Continued on next page

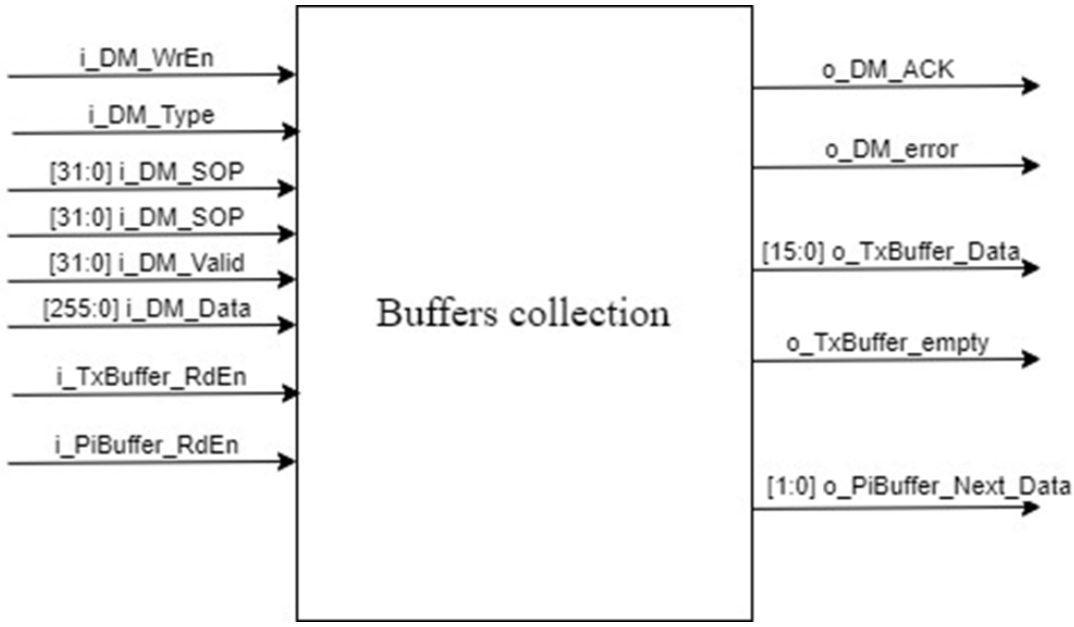


Figure 4.40: Buffers Collection Block Diagram

(Continued)

i_DM_WrEn	1 bit	Data Link Layer	Write enable, the data link layer asserts it to insert a packet if no other packet is already stored (checked by o_DM_ACK = 0).
i_DM_Type	1 bit	Data Link Layer	Type of packet: 0 for TLP and 1 for DLLP.
i_DM_SOP	32 bits	Data Link Layer	The start of packet indicator. Each bit refers to the corresponding byte in i_DM_Data. The bit which equals 1 indicates to the start byte of the packet.
i_DM_EOP	32 bits	Data Link Layer	The end of packet indicator. Each bit refers to the corresponding byte in i_DM_Data. The bit which equals 1 indicates to the end byte of the packet.

Continued on next page

(Continued)

i_DM_Valid	32 bits	Data Link Layer	The valid indicator. Each bit refers to the corresponding byte in i_DM_Data.
i_DM_Data	32 bytes (256 bits)	Data Link Layer	The data which comes from the data link layer. If the packet is more than 32 Bytes, each 32 bytes of it comes in a single clock cycle.
i_TxBuffer_RdEn	1 bit	Controller	Read enable of the Tx buffer. When it is 1, the buffer outs two bytes to the MUX.
i_PiBuffer_RdEn	1 bit	Controller	Read enable of the Packet Indicator buffer. When it is 1, the controller reads the data in the buffer.
o_DM_ACK	1 bit	Data Link Layer	When end of a packet is received, this flag is asserted to prevent the data link layer from sending any other packet till the stored packet is completely transmitted. On the other hand, when this signal is low, the data link layer can send packets to the buffer and the interface bus cant read the data from the buffer.
o_DM_error	1 bit	Data Link Layer	Signal used for debugging
o_TxBuffer_Data	2 bytes (16 bits)	Multiplexer	Two bytes of the transmitted packet.
o_TxBuffer_empty	1 bit	Controller	Empty flag to indicate that the buffer is empty.
o_PiBuffer_Next_Data	2 bits	Controller	Two bits to indicate the start of packet and its type.

Detailed operation:

The buffers collection consists of the following modules

- **Interface buffer**: This module communicates with the Data Link layer to receive and store the incoming packet. It consists of an 17-row FIFO of 256-bits to store the incoming data packet. It also stores the type of the incoming packet (TLP of DLLP), and the positions of the start and the end of the packet.
- **Interface bus**: This module splits the incoming packet into 2-byte chunks over many clock cycles to be used by the other modules.
- **Tx Buffer**: This buffer stores the 2-byte chunks of data until they are requested by the controller. It consists of a 2048-row FIFO of 16-bits.
- **PI Buffer**: This buffer stores the 2-bit packet indicator that specifies the type of the packet and gives them to the controller. There is a 1-item shift between the Tx buffer and the PI buffer such that the controller can see the packet indicator of the next Tx packet.

Verification approach:

Separate agents will be used for every independent module that the DUT interacts with (The DataLink layer, the controller, and the muxes collection). Additionally, a global agent was implemented to manage the clock and reset signals. So the buffers environment will have 4 agents in total All these agents are active; however, the controller and muxes collection agents will be deactivated during the transmitter chip verification phase. This is because these modules reside within the transmitter and, at the Chip Verification state, it will not be driving signals but solely monitoring them.

Test items and sequences:

To create sequences that can be synchronized with all the agents, a basic, customizable, single-cycle sequence was created for each agents. The basic sequences were then combined together in virtual sequences that have access to all the sequencers in the testbench.

For the buffers collection test, these basic sequences were used to create the following test items:

- **Reset sequence**: resets all modules of the RTL for 5 cycles.
- **Short packet write**: Writes a packet that is only 1 Row long into the interface buffer.

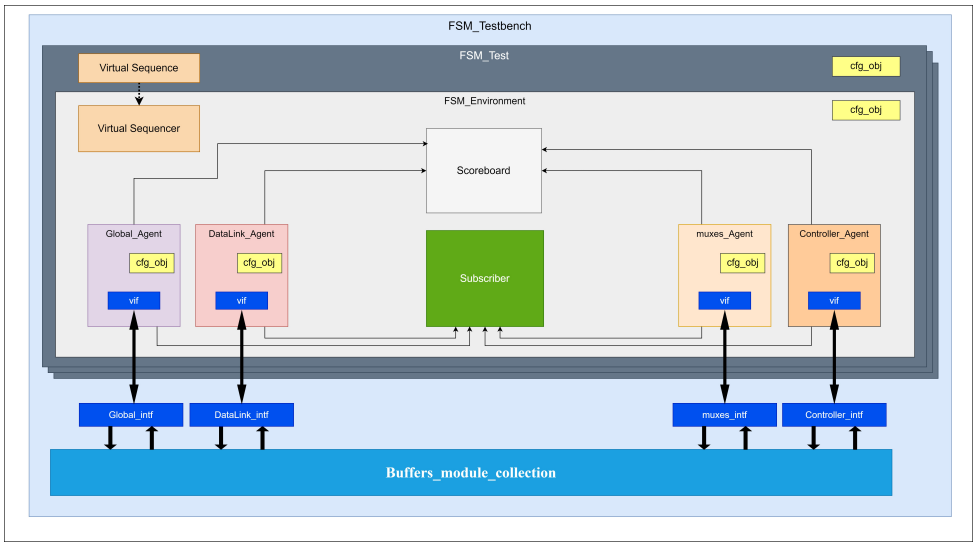


Figure 4.41: Buffers Collection testbench architecture

- **Long packet write**: Writing a packet with a random number of rows (as many as 17 Rows) into the interface buffer.
- **Read or no read**: Read/don't read data from Tx buffer and PI buffer.

Two virtual sequences were created for the buffers collection and they are shown in the table below.

Sequence name:	Test item series	Description
Test operation sequence	· Reset sequence · 50 short packet writes · 10 long packet writes **Parallel with:** ·Read sequence until the end of the other branch (with the read-enable randomized)	This sequence tests the general operation of the RTL using many packets of different lengths and randomized read-enable

Continued on next page

(Continued)

Test Tx-buffer full	·Reset sequence ·14 long packet write (with the row size set to the maximum number 17) **Parallel with:** · 2800 Read sequences with read-enable disabled · Read sequences until the end of the other branch with read-enable enabled	The sequence is desgined the behavior of the design when the Tx-buffer becomes full (size=2048). This is done by continously writing into the Tx-buffer without reading from it until it gets full. Then, wait some time and start reading the data until it gets empty again.

The figure below shows the waveforms generated after successfully performing the two sequences.

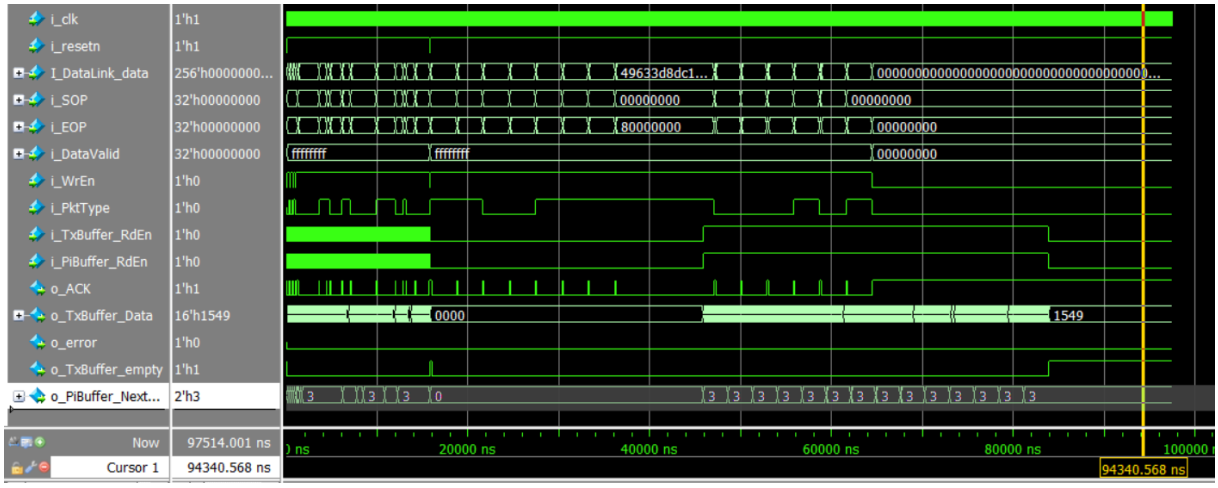


Figure 4.42: Buffers test waveforms

We can see how the read enable behaves randomly in for the first virtual sequence, Then it switches to zero once the second virtual sequence starts at around 16000ns. The Tx buffer becomes full at around 40000ns and becomes unable to accept more packets from the DataLink layer. The read enable turns on again at around 45000ns and the DataLink packets start flowing again until they run out. At around 84000ns, the Tx buffer becomes depleted and the read enable is automatically forced low by the controller.

Coverage Collection:

Functional coverage collection will be performed using the subscriber class. The table below shows some examples for the coverage bins that are implemented in the code.

coverpoint	Coverage Bins Description
i_reset_n_cp	reset_data = {0} , active_data = {1}
i_WrEn_cp	reset_input_case = {0}; insert_a_packet = {1}; start_paket_seq = (0 = 1 = 1 = 1);
i_DataLink_cp	null_data_in = {256'h_0};
i_SOP_cp	new_paket = {32'h_0000_0001};
i_DataValid_cp	invalid_data = {32'h_0000_0000}; valid_data = {32'h_FFFF_FFFF};
i_PktType_cp	TLP_data = {0}; DLLP_data = {1};
i_TxBuffer_RdEn_cp	TxBuffer_hold = {0};
i_PiBuffer_RdEn_cp	TxBuffer_out_data = {1}; bins PiBuffer_hold = {0}; bins PiBuffer_out_data = {1};
o_ACK_cp	prevent_sending_data = {1};
o_TxBuffer_Data_cp	send_packets = {0}; null_data_out = {16'h_0000}; ones_data_out = {16'h_FFFF};
o_PiBuffer_Next_Data_cp	start_new_paket = (3= 1 = 3); invalid_data_out = {2'b00}; new_paket_start_1 = {2'b10}; new_paket_start_2 = {2'b01};
o_TxBuffer_empty_cp	valid_data_out = {2'b11}; tx_buffer_hold_data = {0}; empty_tx_buffer = {1};

The code coverage will be performed automatically by QuestaSim, We aim to achieve 90% code coverage.

Evaluation:

We detected the following bugs in the code:

- **Start/End indicators**: It was found that the interface bus switches the position of the START and END symbols during transmission.
- **RTL drives XXXXX**: It was found that the PI-buffer sends XXXXX values in the output when it has not been filled yet

Both of these errors have been fixed and a new version of the RTL has been created.

Coverage:

The functional coverage report for our created cover groups is shown below.

CoverPoints	Total Bins	Hits	Misses	Hit %	Goal %	Coverage %
i_DataLink_cp	1	1	0	100.00%	100.00%	100.00%
i_PiBuffer_RdEn_cp	2	2	0	100.00%	100.00%	100.00%
i_PktType_cp	2	2	0	100.00%	100.00%	100.00%
i_reset_n_cp	2	2	0	100.00%	100.00%	100.00%
i_SOP_cp	3	3	0	100.00%	100.00%	100.00%
i_TxBuffer_RdEn_cp	2	2	0	100.00%	100.00%	100.00%
i_WrEn_cp	3	3	0	100.00%	100.00%	100.00%
o_ACK_cp	2	2	0	100.00%	100.00%	100.00%
o_PiBuffer_Next_Data_cp	5	5	0	100.00%	100.00%	100.00%
o_TxBuffer_Data_cp	2	2	0	100.00%	100.00%	100.00%
o_TxBuffer_empty_cp	2	2	0	100.00%	100.00%	100.00%

Figure 4.43: buffers functional coverage

We can see that we have achieved 100% functional coverage.
As for the code coverage, we first had to add some exclusions to parts of the code that are impossible to cover. One example of this is in the interface bus, where it appears that there were plans to allow the start of the packet to be at places other than the least-significant byte, but this functionality was then dropped as it is not present in any of the other modules.
After making these necessary exclusions, the code coverage report came out as follows.

Scope	TOTAL	Statement	Branch	FEC Expression	FEC Condition	Toggle
TOTAL	90.00	100.00	95.89	75.00	100.00	79.15
buffers_DUT	83.70	100.00	--	100.00	--	51.10
InterfaceBuffer	92.92	100.00	95.00	100.00	100.00	69.60
InterfaceBus	96.55	100.00	100.00	--	100.00	86.20
TxBuff	84.58	100.00	92.30	50.00	--	96.03
PiBuff	84.32	100.00	92.30	50.00	--	95.00

Figure 4.44: buffers code coverage

We have successfully achieved 90% code coverage.

4.2.2 Tx Muxes

Introduction

The group of modules (Start-End Framing, PCIe Mux, and Framing Alignment) will be treated as a bundled module as shown in figure 4.45. Start-End Framing is a simple Mux that is used to add 1 byte of characters to indicate the start of a TLP (STP), the start of DLLP (SDP), the end of a correct data (END), or the end of an incorrect data (EDB). PCIe Mux is also a simple mux that is used to choose between two bytes data from four different modules. These modules are the Tx buffer, the Start–End Framing, the logical idle, and the ordered set buffer. Framing Alignment is used only when the Data Mux chooses the Start-End Framing, and its main job is to remove the 2 bytes added from the Start-End Framing.

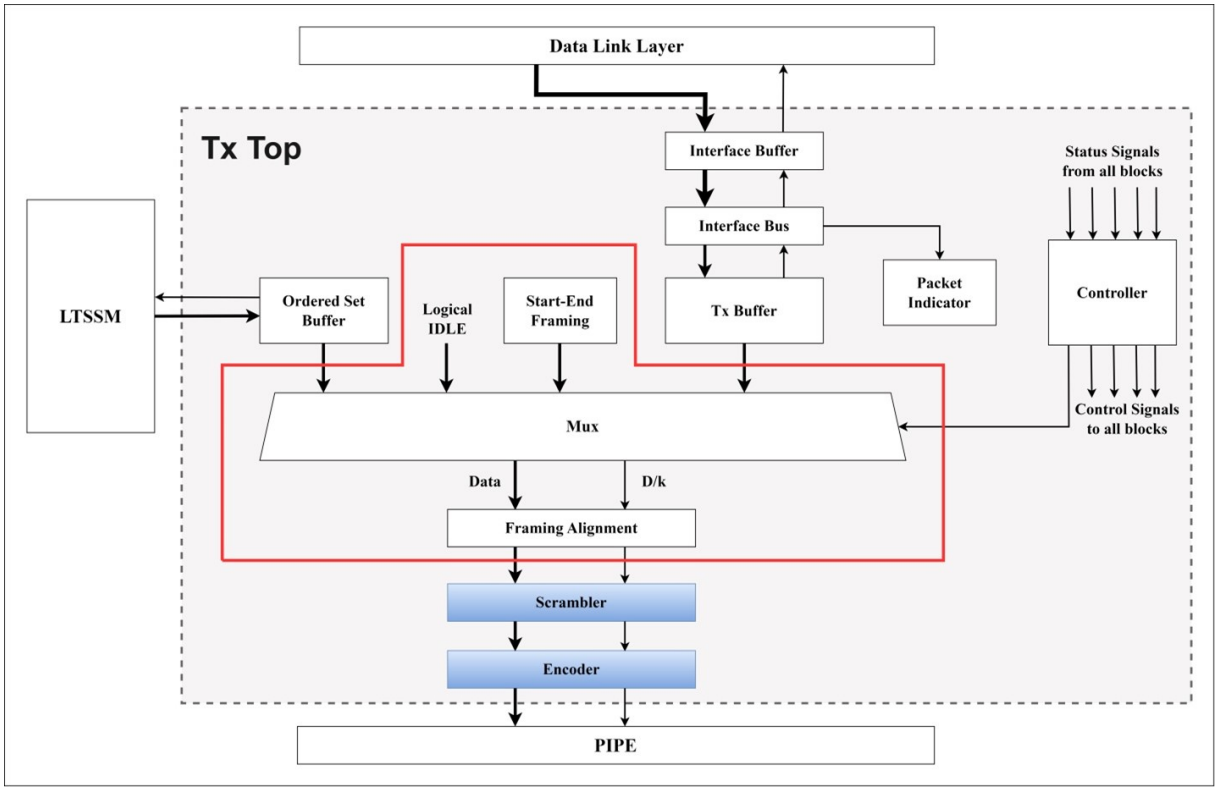


Figure 4.45: Tx_Muxes interaction with other blocks.

I/O Signals Description

Block Diagram

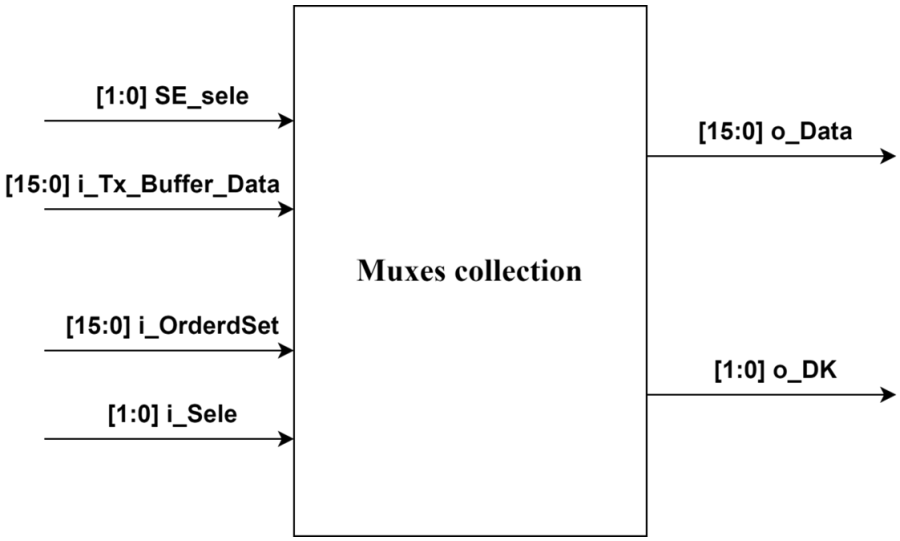


Figure 4.46: Muxes Block Diagram

Inputs

Name	Width	Source	Description
i_clk	1 bit	Global	Clock signal at positive edge registers
STP, 8'h00	16 bits	16'hFB00	Start of TLP
SDP, 8'h00	16 bits	16'h5C00	Start of DLLP
8'h00, END	16 bits	16'h00FD	End of the correct TLP or DLLP
8'h00, EDB	16 bits	16'h00FE	End of the TLP or DLLP with an error
SE_sele	2 bits	Controller	The selection line of the SEMux. 00 → STP, 01 → SDP, 10 → END, 11 → EDB
STP, 8'h00	16 bits	16'hFB00	Start of TLP
i_Tx_Buffer_Data	16 bits	TX Buffer	The two bytes data of the packet.
i_SE_Data	16 bits	Start-End Framing	The two bytes of the Start – End characters
i_Logical_Idle	16 bits	Logicle Idle	The two bytes of the logical Idle characters.
i_OrderdSet	16 bits	OS Buffer	The two bytes of the ordered set packets
i_Sele	2 bits	Controller	The selection line of PCIe Mux and Framing Alignment. 00 → Tx Buffer Data, 01 → Start-End Framing Data, 10 → Logical Idle, 11 → OS Buffer Data.

Table 4.2: Tx_Muxes Block inputs

Outputs

Name	Width	Source	Description
o_Data	16 bits	PIPE	Two Bytes data after Framing Alignment
o_DK	2 bits	PIPE	One bit for each byte. If the bit equals 0, it means the corresponding byte is data. If the bit equals 1, it means the corresponding byte is control.

Table 4.3: Tx_Muxes Block outputs

Muxes operation

Start-End Framing:
For the Start-End Framing, it is just a simple mux that takes its select signal from the controller and chooses which DataFrame to use as input to the PCIe Mux. The selection line of the Start-End Framing Mux is as follows.

- 00 → STP (Start of TLP)
- 01 → SDP (Start of DLLP)
- 10 → END (End of the correct TLP or DLLP)
- 11 → EDB (End of the TLP or DLLP with an error) (not deployed)

PCIe Mux:
For the PCIe Mux, it chooses which data to pass based on the Select signal coming from the controller. The selection line of the PCIe Mux is as follows:

- 00 → Tx Buffer Data
- 01 → Start-End Framing Data
- 10 → Logical Idle
- 11 → OS Buffer Data

Framing Alignment:
Framing Alignment block is used to remove the overhead bytes that are added in Start–End framing module only if the PCIe Mux chose the SE framing, otherwise, it just passes the data and control signals exactly as it got them from the PCIe Mux.

Verification Approach

Testbench Architecture and Hierarchy

The testbench will follow standard UVM architecture. It will contain two agents, one for the interface between the buffers and the datalink layer and one for the interface between the buffers and the other modules of the Transmitter. Each agent will contain a Sequencer for managing the incoming sequence, a Driver for driving the interface, and monitor to sample the interface. Both monitors send their data to the scoreboard for error checking and to the subscriber for coverage collection. The schematic for the UVM environment is shown below.

Test Items and Cases

The following are the designed test items:

- Reset sequence.
- Tx packet sequence.
- Idle date sequence.
- OS training set.

Since this is a very simple collection of modules, the UVM test will simply consists of sending a random series of these test items to test the behavior of the DUT.
A Tx packet should start by sending an STP or SDP symbol from the SE mux, followed by the Tx data, then end with an END symbol from the SE mux.

Results and Coverage

Functional coverage

Functional coverage collection will be performed using the subscriber class. The table below shows the coverage bins needed for coverage collection. Figure[h] We can see

Signal	Coverage Bins Description
i_reset_n_cp	1 bin for reset = 0, and 1 bin for reset = 1
i_SE_sele_cp	3 bins for SE_sele = 0,1, and 2 respectively
i_Mux_sel_cp	4 bins for each of the possible values of Mux_sel
Tx_Buffer_Data_cp	(if reset = 1, Mux_sel = 0) 1 bin for Tx_data = 0, and 1 bin for Tx data = FFF
i_OrderdSet_cp	(if reset = 1, Mux_sel = 3) 1 bin for OS_data = 0, and 1 bin for OS_data = FFF

Table 4.4: Covergroup Tx_Muxes

that our test has achieved 100% coverage over the created cover groups.

cvr_grp

Summary	Total Bins	Hits	Hit %
Coverpoints	15	15	100.00%
Crosses	0	0	0.00%

Search:

CoverPoints	Total Bins	Hits	Misses	Hit %	Goal %	Coverage %
i_Mux_sel_cp	4	4	0	100.00%	100.00%	100.00%
i_OrderdSet_cp	3	3	0	100.00%	100.00%	100.00%
i_reset_n_cp	2	2	0	100.00%	100.00%	100.00%
i_SE_sele_cp	3	3	0	100.00%	100.00%	100.00%
Tx_Buffer_Data_cp	3	3	0	100.00%	100.00%	100.00%

Figure 4.47: Cover group Tx_Muxes.

Code coverage

Some exclusions had to be added before starting code coverage. an example of these exclusions is some of the if-statements in the Data-Mux module which contains copy-pasted conditions in multiple sections that contain uneeded conditions. Code coverage will be performed automatically.

Evaluation and Detected Bugs

- The data mux was found to incorrectly invert the DK signals of the outgoing data.
- The framing alignment module mixed between synchronous and asynchronous signals which can introduce many timing violations. A new version of the RTL was made were both of these issues have been fixed.

4.2.3 OS Buffer

Introduction

Design Description

The OS Buffer, shown in figure 4.48, is the interface between LTSSM and the Tx block. The LTSSM inserts the ordered set packets like TS1, TS2 and SKP in this buffer whenever it wants. And the TX controller decides when to transmit the data from the OS Buffer across the PIPE.

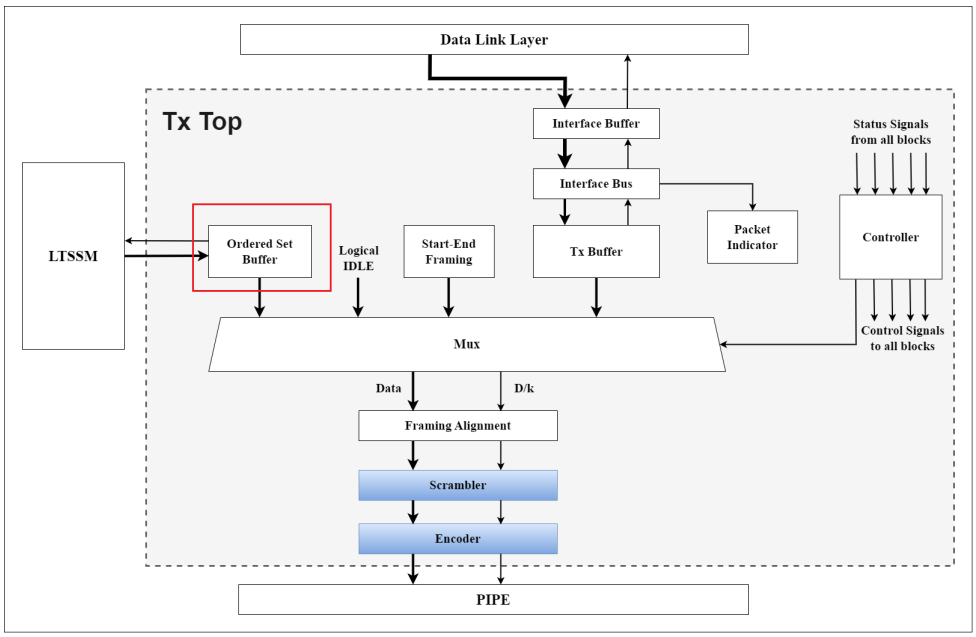


Figure 4.48: Tx System - OS Buffer Highlighted

Block I/O Signals:
The block diagram of the OS Buffer is shown below in figure 4.49.

Interaction with Other Modules:

The OS Buffer receives training set symbols from the LTSSM interface (if o_OS_buffer_full is low) and stores it in an internal FIFO queue (depth = 2048, each is 2 bytes). If the OS_buffer contains some data, it will lower the o_OS_buffer_empty flag to tell the controller that there are pending training sets. The controller will then decide when to transmit this data through the Data_mux to the PIPE

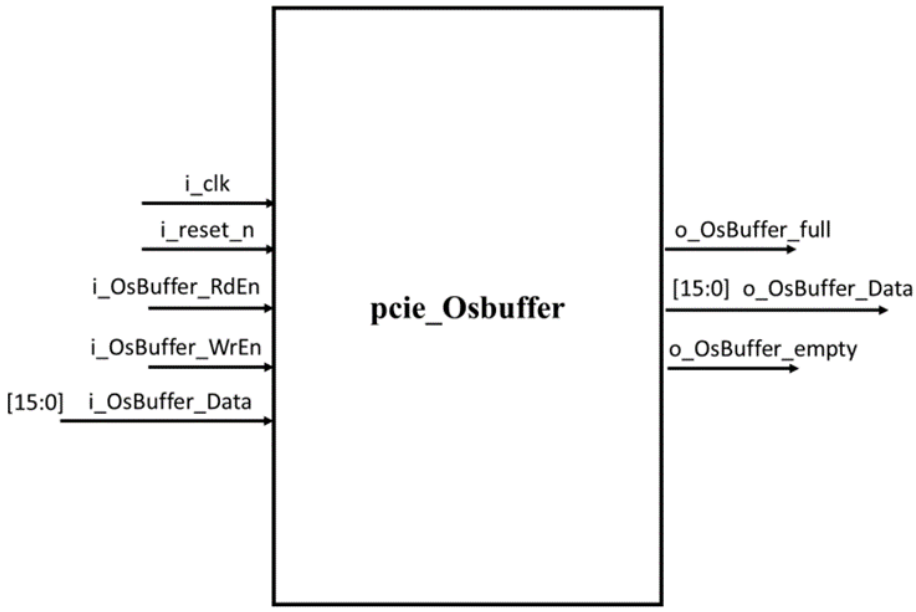


Figure 4.49: OS Buffer Block Diagram

I/O Signals Description

Inputs:

Name	Width	Source	Description
i_clk	1 bit	Global	Clock signal at positive edge registers
i_reset_n	1 bit	Global	Synchronous negative-edge reset
i_OsBuffer_RdEn	1 bit	Input from the controller	Input from the controller. Read enable of the buffer. When it is 1, the buffer outs two bytes to the MUX.
i_OsBuffer_WrEn	1 bit	Input from the LTSSM	Write enable of the buffer. When it is 1, the LTSSM inserts two bytes in the buffer.
i_OsBuffer_Data	16 bits	Input from the LTSSM	Two bytes of the ordered set packet.

Outputs:

Name	Width	Destination	Description
o_OsBuffer_Data	16 bits	Output to the MUX	Two bytes of the ordered set packet.

Continued on next page

(Continued)

o_OsBuffer _full	1 bit	Output to the LTSSM	Full flag to indicate that the buffer is full.
o_OsBuffer _empty	1 bit	Output to the controller	Empty flag to indicate that the buffer is empty.

Detailed Operation

The function of the OS_buffer can be described in the following points:

- If the buffer is not full (depth = 2048, each is 2 bytes), it stores "i_OsBuffer_Data" in the internal FIFO queue as long as "i_OsBuffer_WrEn" is asserted.
- In case "i_OsBuffer_RdEn" is driven high while the buffer is not empty, 2 bytes will be removed from the FIFO and be transmitted to the Data_mux through "o_OsBuffer_Data"

Verification Approach

The OS Buffer environment consists of four agents as shown in figure 4.50 below.

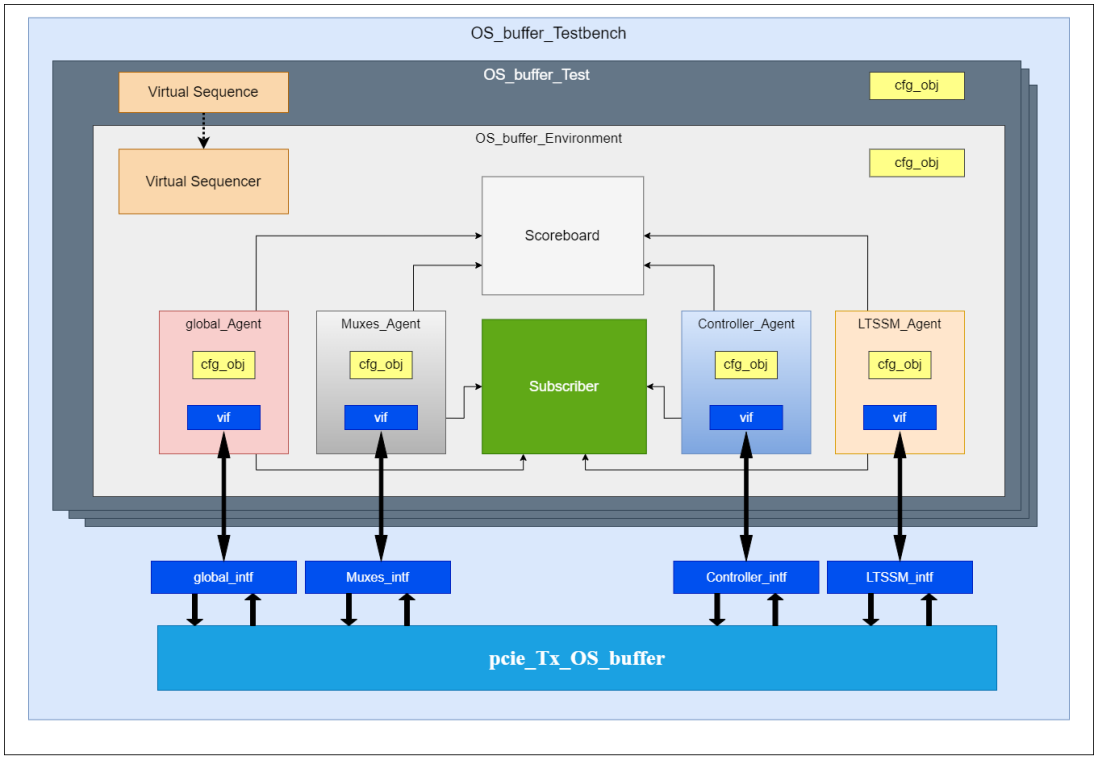


Figure 4.50: OS Buffer Testbench Architecture

Test Items and Sequences

Test Items:

1. i_OsBuffer_RdEn:
   - 0: The buffer read operation is disabled.
   - 1: The buffer read operation is enabled.
2. i_OsBuffer_WrEn:
   - 0: The buffer write operation is disabled.
   - 1: The buffer write operation is enabled.
3. i_OsBuffer_Data:
   - Invalid value
   - Valid value

Test Sequences:

Sequence Name	Description
reset_sequence	This sequence resets the buffer then de-asserts the reset signal. It is applied to the global sequencer.
write_LTSSM _sequence	This sequence writes data into the buffer. It is applied to the LTSSM sequencer.
no_write_LTSSM _sequence	This sequence de-asserts the write signal. It is applied to the LTSSM sequencer.
read_controller _sequence	This sequence reads data from the buffer. It is applied to the controller sequencer.
no_read_controller _sequence	This sequence de-asserts the read signal. It is applied to the controller sequencer.

Coverage Collection

The implemented cover groups are shown in the table below.

Signal	Coverage Bins Description

Continued on next page

(Continued)

i_OsBuffer _RdEn	single read - multiple reads
i_OsBuffer _WrEn	single write - multiple writes
i_OsBuffer _Data	The signal takes one of the following values at least once: {COM, PAD}, {COM, Link}, {PAD, NFTS}, {Lane, NFTS}, {DR_ID, LIDLE}, {TS1_ID, TS1_ID}, {TS2_ID, TS2_ID}, {COM, SKP}, {SKP, SKP}, {COM, FTS}, {FTS, FTS}, {LIDLE, LIDLE}
o_Os Buffer_Data	The signal takes one of the following values at least once: {COM, PAD}, {COM, Link}, {PAD, NFTS}, {Lane, NFTS}, {DR_ID, LIDLE}, {TS1_ID, TS1_ID}, {TS2_ID, TS2_ID}, {COM, SKP}, {SKP, SKP}, {COM, FTS}, {FTS, FTS}, {LIDLE, LIDLE}
o_Os Buffer _full	full
o_Os Buffer _empty	empty

Figures 4.51 and y show the code and functional coverage of the OS Buffer module, respectively.

Total Coverage:					99.67%	99.92%
Coverage Type	Bins	Hits	Misses	Weight	% Hit	Coverage
Statements	34	34	0	1	100.00%	100.00%
Branches	16	16	0	1	100.00%	100.00%
FEC Expressions	2	2	0	1	100.00%	100.00%
FEC Conditions	4	4	0	1	100.00%	100.00%
Toggles	252	251	1	1	99.60%	99.60%

Figure 4.51: OS Buffer Code Coverage

Evaluation and Detected Bugs

Figure 4.53 below shows a repeated bug in the OS Buffer module. When reading from the OS Buffer when it is empty, the output is not zeros.

CoverPoints	Total Bins	Hits	Misses	Hit %	Goal %	Coverage %
i_OsBuffer_Data_cp	12	12	0	100.00%	100.00%	100.00%
i_OsBuffer_RdEn_cp	1	1	0	100.00%	100.00%	100.00%
i_OsBuffer_WrEn_cp	1	1	0	100.00%	100.00%	100.00%
o_OsBuffer_Data_cp	12	12	0	100.00%	100.00%	100.00%
o_OsBuffer_empty_cp	1	1	0	100.00%	100.00%	100.00%
o_OsBuffer_full_cp	1	1	0	100.00%	100.00%	100.00%

Figure 4.52: OS Buffer Functional Coverage

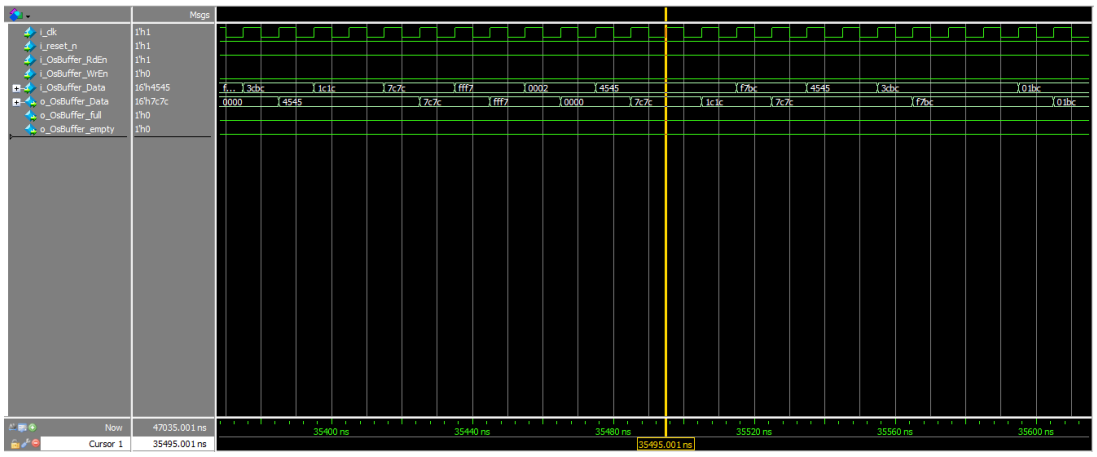


Figure 4.53: OS Buffer Read Empty Bug

4.2.4 Encoder and Scrambler

Introduction

PCIe gen 1 uses scrambling and 8b10b encoding of outgoing data packets to improve the reliability of the electrical communication channel. The receiver reverses this encoding procedure to recover the original data.

Interaction with Tx Blocks

The encoding module receives the final outgoing packets from the original PCIe IP and performs scrambling and 8b10b encoding on them. It then sends the resulting data to the PIPE interface instead of the original data. The PIPE interface has to be modified to account for the larger size of the encoded data (each 8 bits are converted into 10 bits). 2 copies of the encoder module are implemented in the Top Transmitter module, 1 for each outgoing byte packet.

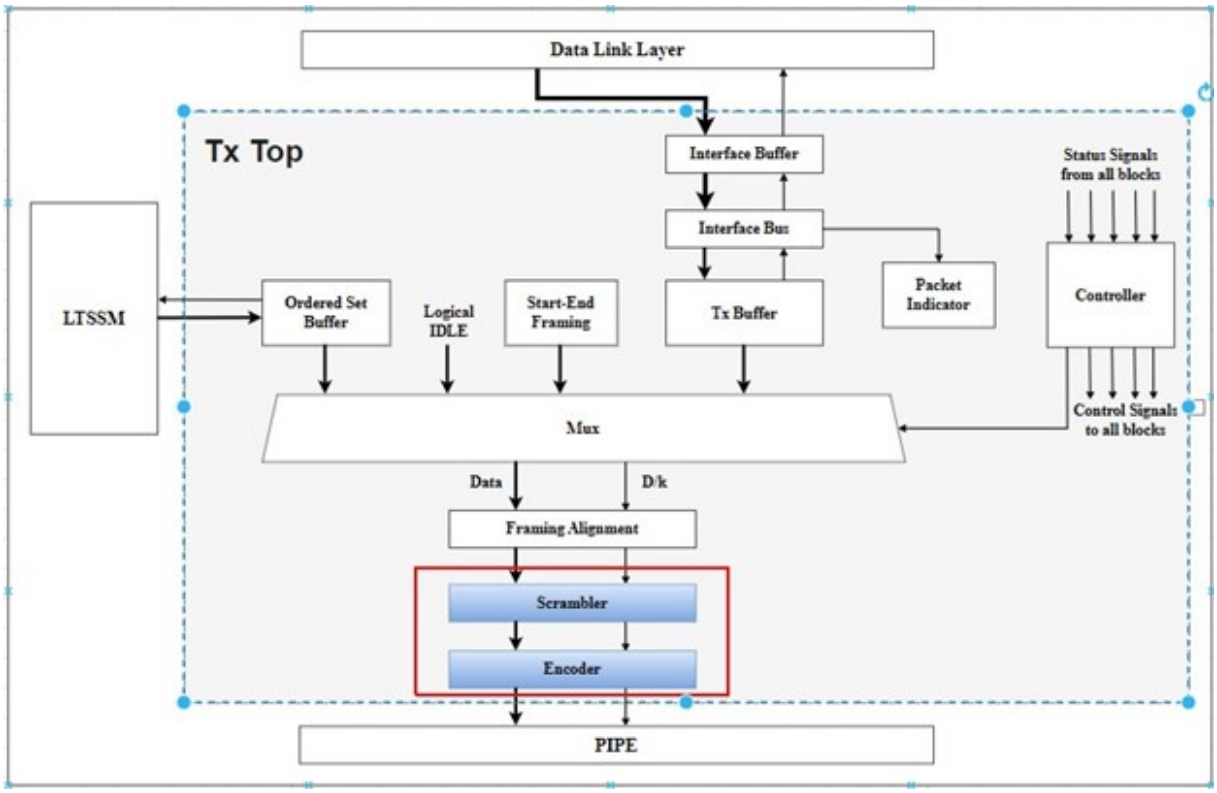


Figure 4.54: The position of the modules in the transmitter architecture.

I/O Signals Description

Inputs

Outputs

Detailed Operation

Block Diagram

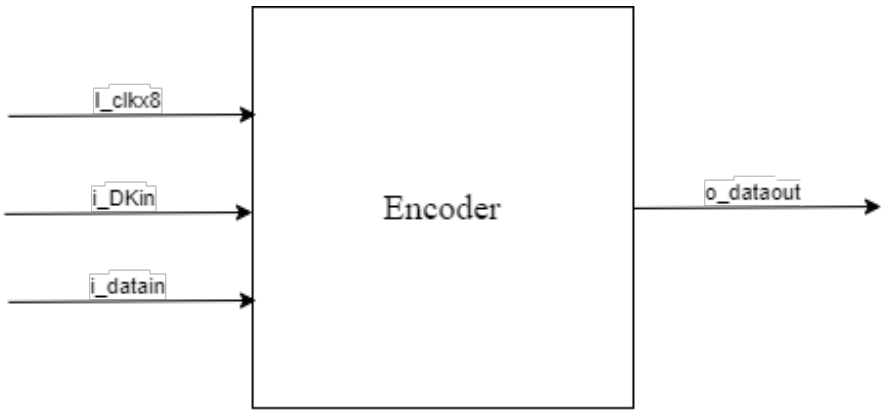


Figure 4.55: Block Diagram encoding

Name	Width	Source	Description
i_clk	1 bit	Global	Clock signal at positive edge registers
i_clkx8	1 bit	Global	Another clock signal with 8 times the rate of the main clock.
i_reset_n	1 bit	Global	Synchronous negative-edge reset
i_DKin	2 bits	Input from the Framing Allignment module	If it equals 1, the input is a control signal. If it equals 0, the input is a data signal.
i_datain	16 bits	Input from the Framing Allignment module	Incoming data bits

Table 4.5: Encoding Block inputs

Name	Width	Source	Description
o_dataout	20 bits	Output to the modified PIPE interface	Outgoing encoded data.

Table 4.6: Encoding Block outputs

Encoding operation

The function of the encoder can be described in the following points: The encoder consists of 2 modules, a scrambler and an 8b10b encoding module.

- The scrambler feeds its output to the 8b10b encoding module.
- The scrambler consists of an LFSR that generates a random sequence.
- The random sequence is XOR'd with the input data to generate the scrambled output.
- Control signals are not scrambled and remain the same, only data signals are scrambled.
- If a COM signal is received, the random sequence of the LFSR is reset (this is used to synchronize the scramblers in the transmitter and the receiver)
- The 8b10b encoding module takes the scrambled data and generates the final output.
- The 8b10b encoding module functions as a lookup table that takes each input into its corresponding output.

- Each possible input has only ONE or TWO possible outputs, based on the value of the current running disparity (The difference between the number of outgoing 1 and 0 bits)
- o The 8b10b encoder balances the disparity to maintain DC balance of the electrical network.
- The D/K signal is also used in determining the output (control and data signals with the same i_datain content will still have different output).
- To reverse the encoding process, the encoded data will have to pass through an 8b10b decoder module by using a reverse lookup table to retrieve the original scrambled data. The scrambled data is then passed through another scrambler that is synchronized with the one in the transmitter (using the COM signal reset) and the output will be the original data.

Configuration

To initialize this module:

1. A reset signal will first have to be sent to initialize the entire PCIe device.
2. A COM signal will have to be transmitted from the LTSSM to synchronize the scramblers in the transmitter and the reciever.

Verification Approach

Testbench Architecture and Hierarchy

The testbench will follow standard UVM architecture. It will contain two agents, one for the interface between the Muxes IP interface and one for the interface between the PIPE and the Encoding modules of the Transmitter. Each agent will contain a Sequencer for managing the incoming sequence, a Driver for driving the interface, and monitor to sample the interface. Both monitors send their data to the scoreboard for error checking and to the subscriber for coverage collection. The schematic for the UVM environment is shown below.

Test Items and Cases

The following are the designed test items:

- Reset sequence.
- Control data sequence.

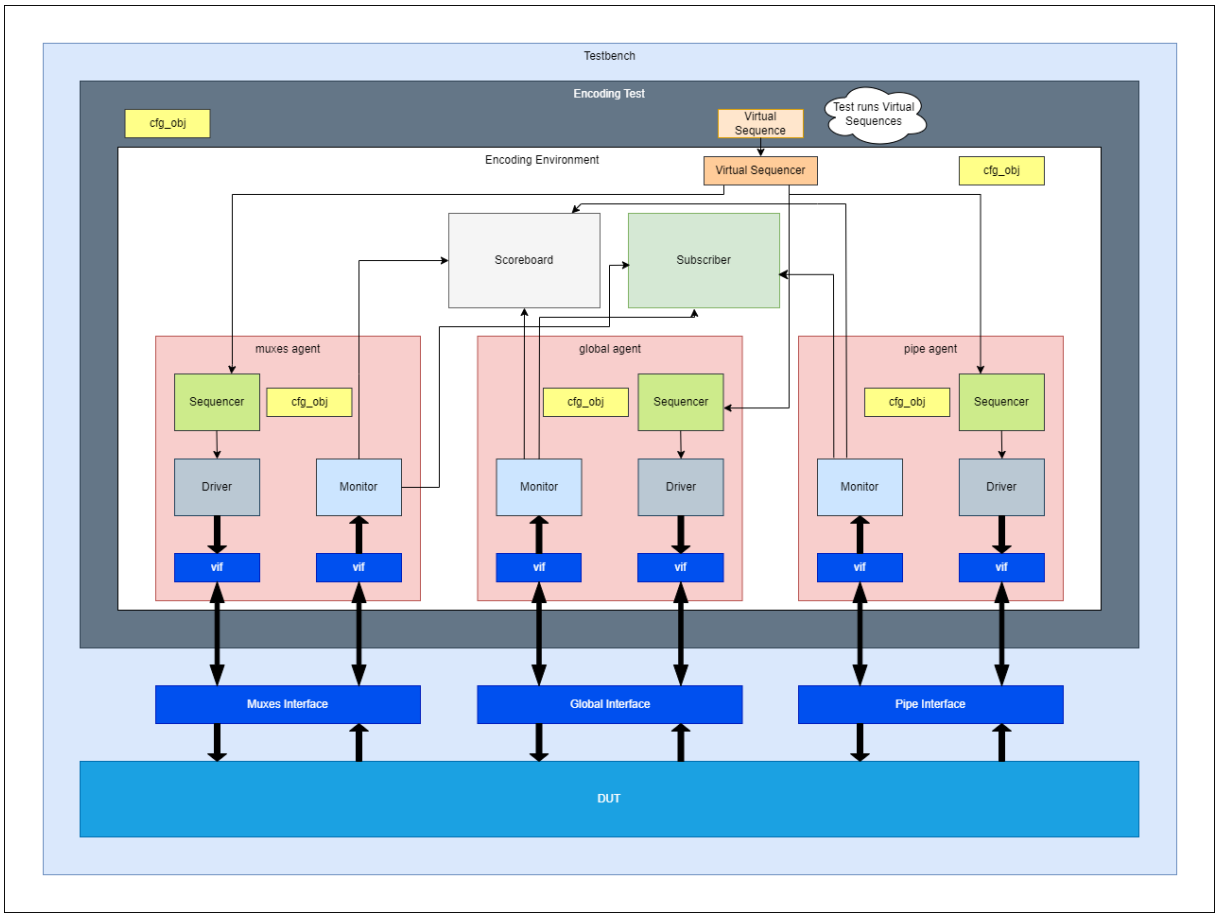


Figure 4.56: Architecture and Hierarchy Encoding Environment

- Normal date sequence.

Since this is a simple collection of modules, the UVM test will simply consist of sending a random series of these test items to test the behavior of the DUT.

Checking Mechanism

The design will be compared against a golden reference model implemented in the scoreboard class using SystemVerilog. Any discrepancies will be recorded and reported.

Results

Functional coverage

Functional coverage collection will be performed using the subscriber class. The table below shows the coverage bins needed for coverage collection.

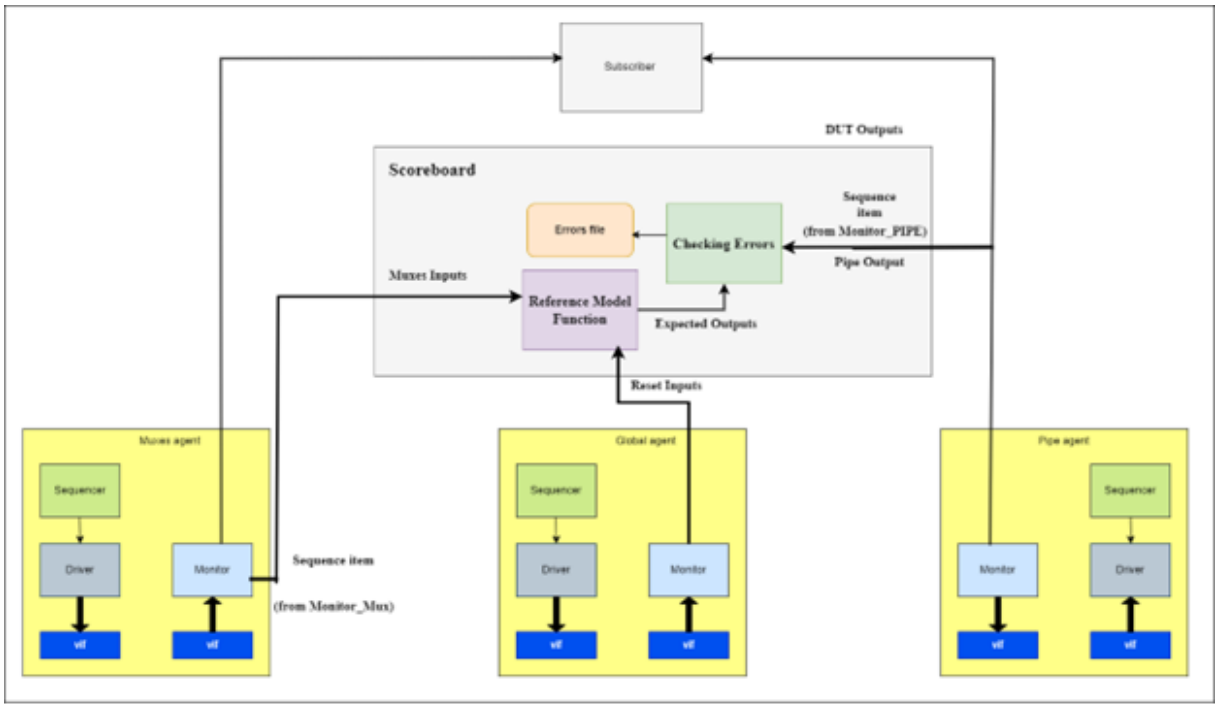


Figure 4.57: Checking mechanism encoding.

Signal	Coverage Bins Description
i_reset_n_cp	1 bin for reset = 0, and 1 bin for reset = 1
i_Dkin_cp	1 bin for data = 10, and 1 bin for control = 01
i_datain	Auto bins for each of the possible values of i_datain
o_dataout_cp	Auto bins for ranges values of o_dataout

Table 4.7: Covergroup encoder

We can see that our test has achieved 100% coverage over the created cover groups.

Code coverage

Code coverage was be performed automatically. The results are shown below.

Covergroups type /encoder_pkg/encoder_subscriber/cvr_grp

Summary	Bins	Hits
Coverpoints	118	118

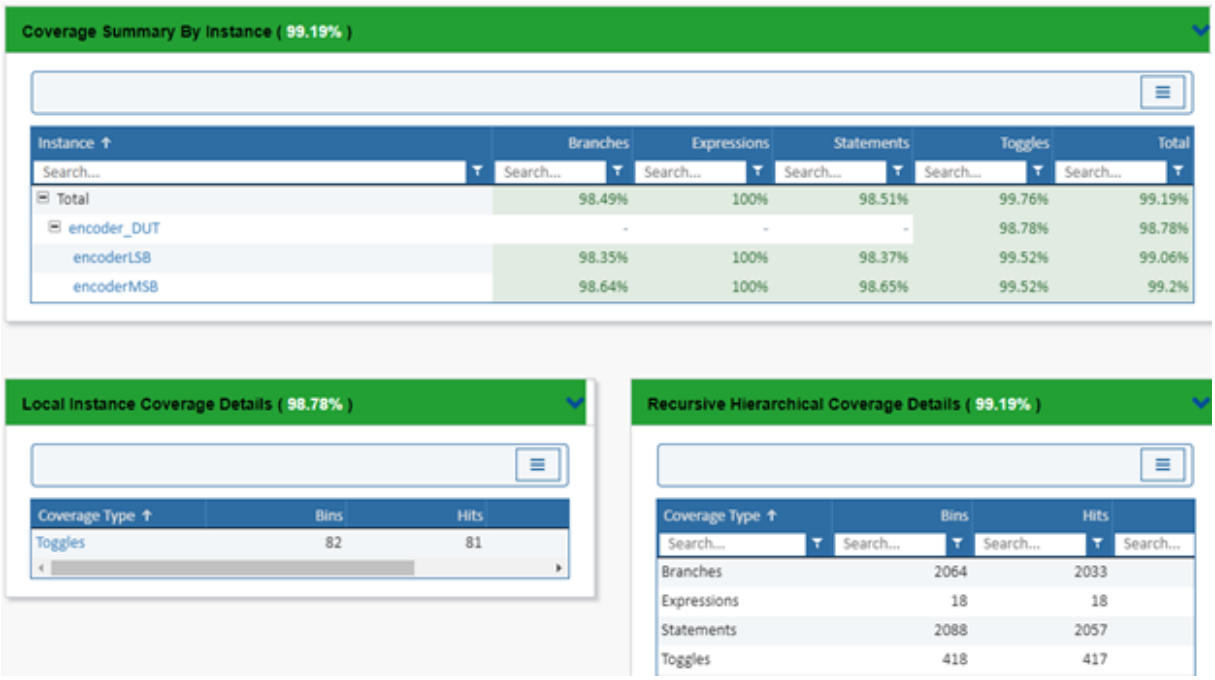
coverpoints

Coverpoints	Bins	Hits	Misses	Goal	Coverage
i_DKin_cp	4	4	0	100	100%
i_datain_cp	64	64	0	100	100%
o_dataout_1_cp	25	25	0	100	100%
o_dataout_2_cp	25	25	0	100	100%

Figure 4.58: Cover group encoding.

Code Coverage is 99.19%, which is acceptable.

Evaluation and Detected Bugs



Coverage Summary By Instance (99.19%)

Instance	Branches	Expressions	Statements	Toggles	Total
Total	98.49%	100%	98.51%	99.76%	99.19%
encoder_DUT	-	-	-	98.78%	98.78%
encoderLSB	98.35%	100%	98.37%	99.52%	99.06%
encoderMSB	98.64%	100%	98.65%	99.52%	99.2%

Local Instance Coverage Details (98.78%)

Coverage Type	Bins	Hits
Toggles	82	81

Recursive Hierarchical Coverage Details (99.19%)

Coverage Type	Bins	Hits
Branches	2064	2033
Expressions	18	18
Statements	2088	2057
Toggles	418	417

Figure 4.59: Code Coverage encoding.

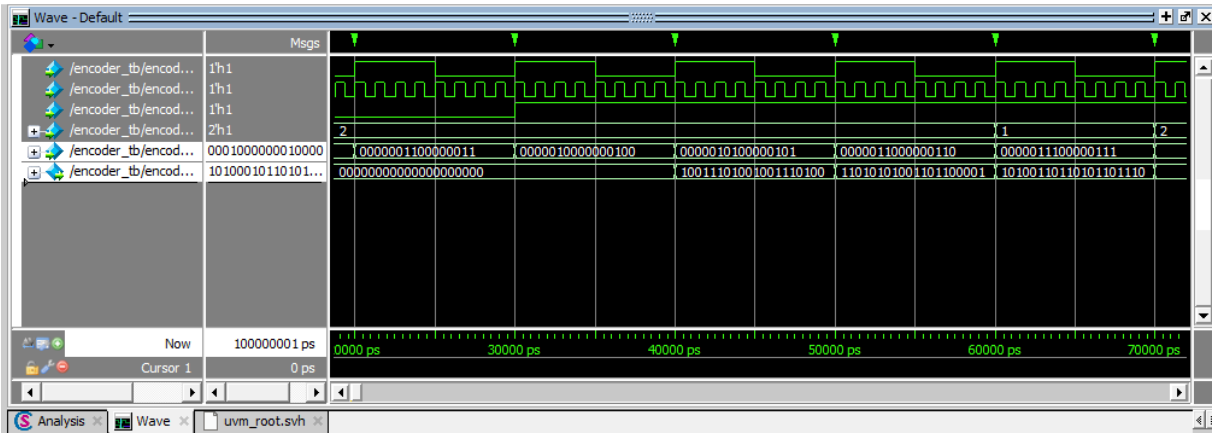


Figure 4.60: Encoder Waveform Results

This unit is: GOOD TO GO

4.2.5 Controller

Introduction

The Tx Controller is the brain of the Tx block as it controls all operations of the Tx block during link training and normal data transmission. Druring Link training, the Transmitter will be receiving data from the LTSSM and transmitting it to the PIPE interface, while during normal operation, the data will be coming from the higher

Data-Link layer.

I/O Signals Description

Inputs

Name	Width	Source	Description
i_clk	1 bit	Global	Clock signal at positive edge registers
i_reset_n	1 bit	Global	Synchronous negative-edge reset
L0	1 bit	Input from the LTSSM	If it equals 1, the normal operation mode is activated. If it equals 0, LTSSM is still in link training or initialization mode.
OS_Empty	1 bit	Input from the Tx buffer	If it equals 1, the Tx buffer is empty. If it equals 0, the Tx buffer is not empty.
Pi_Buffer	2 bits	Input from the packet indicator buffer	The two bits data of the packet indicator: • 00 = invalid data • 01 = current byte is the start of a transaction packet • 10 = current byte is the start of a data link packet • 11 = Valid data,but not the start of a packet

Table 4.8: Controller Block inputs

Outputs

Detailed Operation

The function of the controller can be described in the following points:

- If L0 equals zero, the MUX can't take data from Tx buffer.
- If there is data on the ordered set buffer, the MUX should take this data until the ordered set buffer gets empty.
- If the ordered set buffer is empty and the Tx buffer is empty, the MUX should select the logical idle.
- If the ordered set buffer is empty and the Tx buffer is not empty, the MUX should select the Start – End framing with the correct choice then take the data from the Tx_buffer.

Name	Width	Source	Description
Mux_Sel	2 bits	Output to the Multiplexer	The selection line of the multiplexer. • 00 = select Tx_buffer data. • 01 = select SE_buffer data. • 10 = select logical idle signal • 11 = select OS_buffer data
SE_Sel	2 bits	Output to the Start – End Framing	The selection line of the multiplexer of the start – End Framing. • 00 = select STP signal (start of transaction packet). This is also the default value when not in use. • 01 = select SDP signal (start of data link packet). • 10 = select END signal. • 11 = select EDB signal (not used in this implementation).
OS_rd_en	1 bit	Output to the Ordered set buffer	If it equals 1, the ordered set buffer outs two bytes of its data.
TxBuffer_rd_en	1 bit	Output to the Tx_buffer	If it equals 1, the Tx buffer outs two bytes of its data.
BiBuffer_rd_en	1 bit	Output to the Packet indicator buffer	If it equals 1, the Packet indicator buffer outs two bytes of its data.

Table 4.9: Controller Block outputs

- If a new data comes to the ordered set buffer while the MUX selects the Tx buffer, the MUX should still select the Tx buffer till the current packet is finished with END then select the ordered set buffer.

Verification approach:

Separate agents will be used for every independent module that the DUT interacts with (The OS-buffer, the buffers collection, the LTSSM, and the muxes collection). Additionally, a global agent was implemented to manage the clock and reset signals. So the buffers environment will have 5 agents in total All these agents are active; however, the OS-buffer, Tx-buffers and muxes collection agents will be deactivated during the transmitter chip verification phase. This is because these modules reside within the transmitter and, at the Chip Verification state, they will not be driving signals but solely monitoring them.

Test items and sequences:

To create sequences that can be synchronized with all the agents, a basic, customizable, single-cycle sequence was created for each agents. The basic sequences were then combined together in virtual sequences that have access to all the sequencers in the testbench.

For the controller test, these basic sequences were used to create the following test items:

- **Reset sequence**: resets all modules of the RTL for 5 cycles.
- **LTSSM sequence**: Sets L0 as 0 or 1 randomly
- **OS buffer sequence**: Sets OS-buffer-empty as 0 or 1 randomely.
- **Tx-buffers sequence**: Sets Tx-buffer-empty and PI-buffer-out to random values

The full test consisted of random selections of these test items.

Coverage Collection:

Functional coverage collection will be performed using the subscriber class. The table below shows some examples for the coverage bins that are implemented in the code.

Signal	Coverage Bins Description
i_Mux_Sel_cp:	1 bin for each of the four values of Mux_Sel
i_OS_rd_en_cp:	1 bin for OS_rd_en = 0, and 1 bin for OS_rd_en = 1
i_TxBuffer_rd_en_cp:	1 bin for TxBuffer_rd = 0, and 1 bin for TxBuffer_rd = 1
i_BiBuffer_rd_en_cp:	1 bin for BiBuffer_rd_en = 0, and 1 bin for BiBuffer_rd_en = 1
i_SE_Sel_cp:	1 bin for each of the values of i_SE_Sel, except for the EDB value

The code coverage will be performed automatically by QuestaSim, We aim to achieve 90% code coverage.

Evaluation:

We did not detect any bugs in the controller RTL.

Coverage:

The functional coverage report for our created cover groups is shown below.

Coverpoints	Bins	Hits	Misses	Goal	Coverage
i_Mux_Sel_cp	4	4	0	100	100%
i_SE_Sel_cp	3	3	0	100	100%
i_OS_rd_en_cp	2	2	0	100	100%
i_TxBuffer_rd_en_cp	2	2	0	100	100%
i_BiBuffer_rd_en_cp	2	2	0	100	100%

Figure 4.61: Controller test functional coverage

We have successfully achieved 100% functional coverage.

Before checking the code coverage, we first had to add some exclusions. This is because controller RTl contains many if-statements that test for impossible input values. After adding these exclusions, the resulting code coverage report is shown below.

Local Instance Coverage Details (90.26%)

Coverage Type	Bins	Hits	Misses	Coverage
Branches	33	32	1	96.96%
Conditions	12	11	1	91.66%
FSM States	4	4	0	100%
FSM Transitions	10	6	4	60%
Statements	161	153	8	95.03%
Toggles	48	47	1	97.91%

Figure 4.62: Controller Code Coverage

We have successfully achieved 90% code coverage for the controller.

Chapter 5

Chip-Level Verification

5.1 LTSSM System

5.1.1 Overview

After verifying each block individually, the next step is to perform a chip-level verification of the top LTSSM RTL. To achieve this, we utilized the UVM feature that allows agents to be deactivated and made passive. This deactivation disables the drivers and sequencers while allowing the monitors to sample internal signals for checking and coverage collection.

5.1.2 Top Environment

We constructed a top-level environment where all the smaller environments (PIPE, Decoder, Creator, Timer, FSM) were instantiated. These are the same environments used previously for individual block verification. This top environment is instantiated within a top test, where we deactivated all the internal agents such as the decoder, creator, timer, and PIPE agents in the FSM environment. The external agents, including Rx, Tx, PHY, and global, remained active as they drove the entire LTSSM chip from the outside. The sequences used in block-level verification were reused and extended in this chip-level verification with modifications to include only the sequences of active agents.

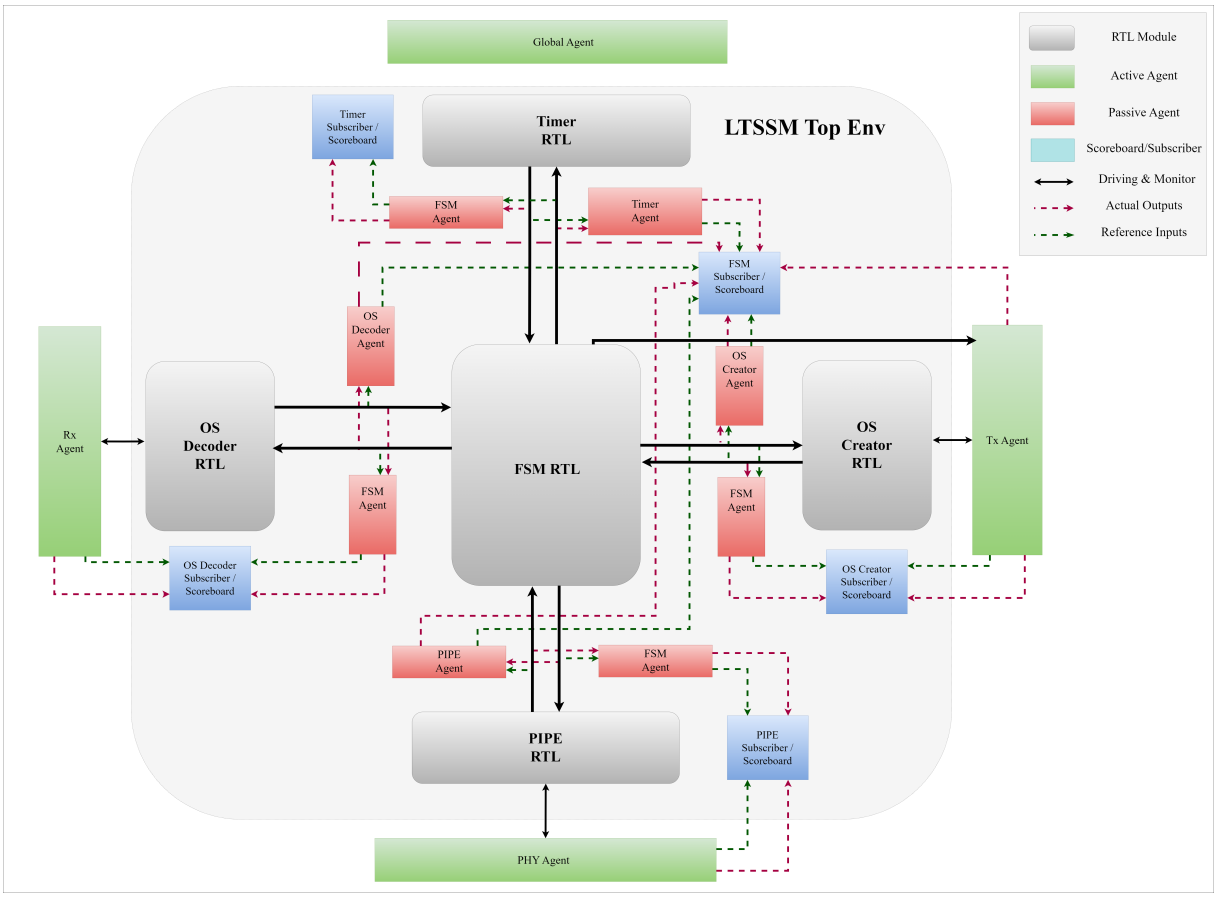


Figure 5.1: Top LTSSM Environment

5.1.3 Binding Interfaces with Modules

To monitor the internal signals of each passive agent, we needed to bind their signals with their interfaces. This required creating an intermediate interface that instantiates the main interface and connects its signals with the module signals. For example, a *top_timer_intf* was created with the same port list as the timer module. This *top_timer_intf* is bound to the timer module in the testbench as shown in the following line of code: *"bind LTSSM_DUT.timer Top_Timer_if intf_timer_top(.*);"* Inside this top interface, the *timer_intf* is instantiated and in an *always_comb* process, the timer interface signals are driven by the port list of the top timer interface, which is already bound to the timer module. This setup ensured that the timer interface samples the same internal signals present in the timer. The same approach was applied to all internal modules to sample all internal data for coverage collection and checking mechanisms.

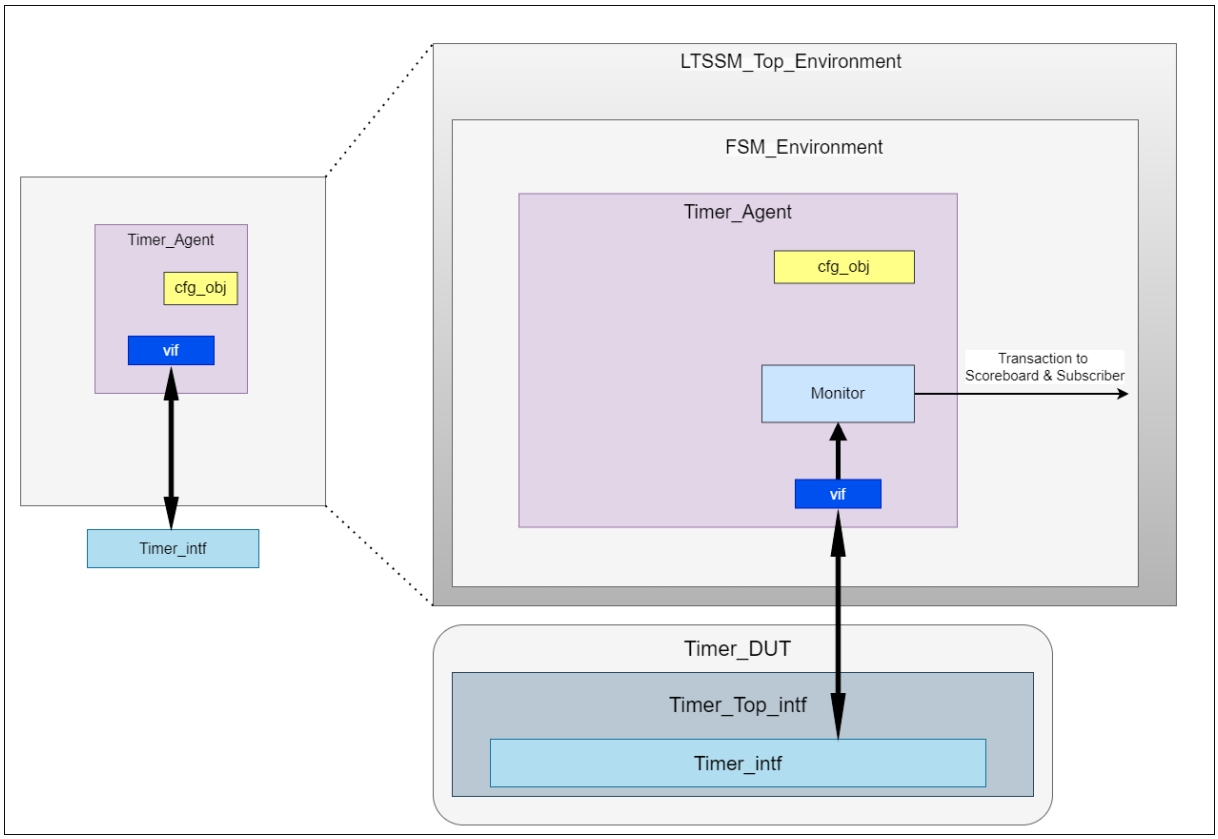


Figure 5.2: Binding in Passive Agent

5.1.4 Evaluation

Code Coverage Results

The table below shows the coverage for each block inside the chip with an overall of 95% code coverage which exceeds our target of 90%.

Module name	**Coverage percentege**
OS_Decoder	91%
OS_Creator	95%
Timer	96%
PIPE	98%
FSM	97%
Overall	**95%**

Functional Coverage Results

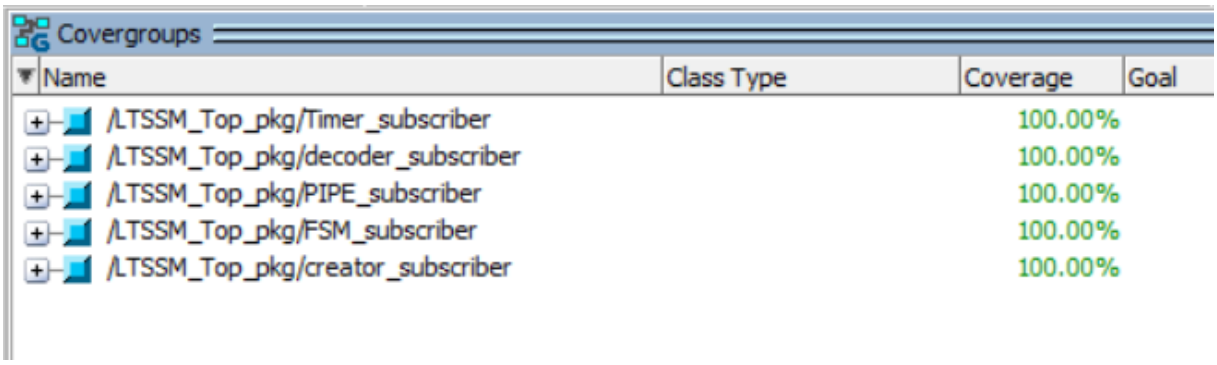


Figure 5.3: LTSSM Top Functional Coverage

As shown in the functional coverage figure, 100% coverage is achieved.

LTSSM Errors

As shown in the figure below, there are many errors but all of these errors are in the creator module, all the other modules had no errors in them. As we said in a previous section, the creator was beyond.

```
#
# UVM_INFO FSM_env/FSM_scoreboard.svh(1811) @ 70095000: uvm_test_top.env_h.FSM_env_h.scoreboard_h [FSM_scoreboard]
# Comparison Passed - No Errors Found
#
# ----------------------------------------------------------------------------------------------------------------------------------
# ----------------------------------------------------------------------------------------------------------------------------------
#
# UVM_INFO PIPE_env/PIPE_scoreboard.svh(402) @ 70095000: uvm_test_top.env_h.PIPE_env_h.scoreboard_h [PIPE_scoreboard]
# Comparison Passed - No Errors Found
#
# ----------------------------------------------------------------------------------------------------------------------------------
# ----------------------------------------------------------------------------------------------------------------------------------
#
#
# ----------------------------------------------------------------------------------------------------------------------------------
# UVM_INFO creator_env/creator_scoreboard.svh(203) @ 70095000: uvm_test_top.env_h.creator_env_h.scoreboard_creator_h [creator_scoreboard]
# Comparison Failed - Error Count Summary:
#  Errors in [7008] Transactions: Data Errors = [4605], Valid Errors = [813], Ack Errors = [668]
#
# ----------------------------------------------------------------------------------------------------------------------------------
# ----------------------------------------------------------------------------------------------------------------------------------
#
#
# ----------------------------------------------------------------------------------------------------------------------------------
# UVM_INFO Decoder_env/decoder_scoreboard.svh(218) @ 70095000: uvm_test_top.env_h.decoder_env_h.scoreboard_h [decoder_scoreboard]
# Comparison Passed - No Errors Found
#
# ----------------------------------------------------------------------------------------------------------------------------------
# ----------------------------------------------------------------------------------------------------------------------------------
#
# UVM_INFO Timer_env/Timer_scoreboard.svh(225) @ 70095000: uvm_test_top.env_h.timer_env_h.scoreboard_h [Timer_scoreboard]
# Comparison Passed - No Errors Found
#
# ----------------------------------------------------------------------------------------------------------------------------------
# ----------------------------------------------------------------------------------------------------------------------------------
#
```

Figure 5.4: LTSSM Top Errors Report

5.2 Tx System

5.2.1 Top Environment

Similarly, after verifying each block inside the Tx system, the next step is to perform a chip-level verification of the top Tx RTL. a top-level environment was created in which all the block-level environments (Tx Buffers, Tx Muxes, OS Buffer Mux, Controller, Encoder - Scrambler) were instantiated. This top environment is instantiated within a top test, where we deactivated all the internal agents such as the Tx Buffers, Muxes, and OS Buffer agents in the Controller environment. The external agents, including LTSSM, DLL, PIPE, and global, remained active as they drove the entire Tx chip from the outside. The sequences used in block-level verification were reused and extended in this chip-level verification with modifications to include only the sequences of active agents. The binding mechanism was implemented to connect internal signals to passive agent, as illustrated in the LTSSM top environment.

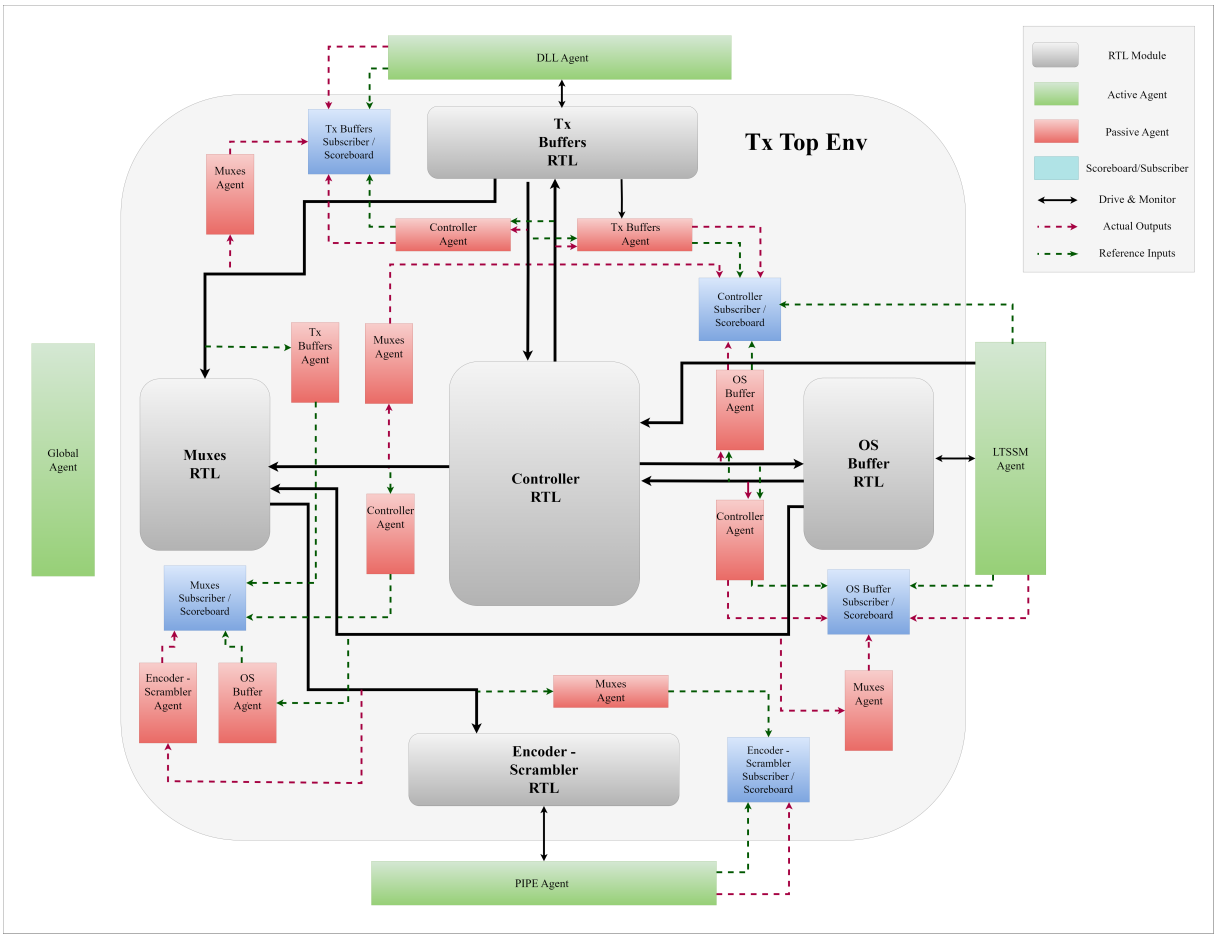


Figure 5.5: Top Tx Environment

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this thesis, we successfully verified the Logical Sub-Block of the PCIe Gen1 PHY Layer using a UVM-based approach. The primary objectives included identifying the mandatory functionalities from the specifications, understanding the implemented design, developing a comprehensive verification plan, and achieving high code and functional coverage. Through rigorous block-level and chip-level verification processes, we were able to meet and exceed our coverage targets, demonstrating the robustness and reliability of our design.

Our verification process involved constructing detailed environments for individual blocks (such as PIPE, Decoder, Creator, Timer, and FSM) and integrating these into a top-level environment. This integrated environment facilitated comprehensive testing and validation of the entire LTSSM system, including the Rx and Tx components. The functional and code coverage results were impressive, with an overall code coverage of 90% and 100% functional coverage, indicating that our verification strategy was thorough and effective.

Moreover, our work highlighted some errors, particularly within the Creator, Decoder, FSM, and Tx Buffers modules, most of which were addressed and resolved to ensure the overall functionality of the system. The successful completion of this project demonstrates the effectiveness of using UVM for functional verification in complex VLSI designs and sets a strong foundation for further advancements.

6.2 Future Work

While this thesis has accomplished significant milestones in verifying the PCIe PHY Layer, there are several areas for future work that can build upon our findings:

1. Receiver Block Enhancement: Further work is needed regarding the receiver (Rx)

block. Enhancements in this area could include performing block- and chip-level verification.

2. Top-Level Environment Implementation: Implementing a comprehensive top-level environment that includes the LTSSM, Tx, and Rx blocks combined. This would facilitate more rigorous testing and verification, ensuring that all components interact correctly and efficiently within the system.

3. Two-Device System: Implementing a more scalable UVM-environment that includes two PCIe devices that communicate with each other.

4. Scalability Testing: Conduct extensive scalability testing to ensure that the PHY layer design can handle increasing data loads and higher bandwidth requirements as demanded by evolving PCIe standards.

By addressing these areas, future research can further enhance the robustness, performance, and security of PCIe PHY Layer designs, ensuring they meet the growing demands of modern computing systems.

References

[1] S. N. Nag, "Technical analysis of pcie to pcie 6: A next-generation interface evolution," *World Journal of Engineering and Technology*, vol. 11, pp. 504–525, 2023.

[2] Statista, "Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2020, with forecasts from 2021 to 2025," 2021.

[3] K. Lender and C. Morrison, "Retimers to the rescue: Pci express® specifications reach their full potential," in *PCI-SIG® Educational Webinar Series 2019*, 2019.

[4] Synopsys, *PCI Express Complete Solution*. 2021.

[5] M. Jackson and R. Budruk, *PCI Express Technology*. MindShare, Inc., 2012.

[6] PCI-SIG, *PCI Express® Base Specification Revision 6.0*. 2021.

[7] PCI-SIG, *PCI Express® Base Specification Revision 1.0*. 2002.

[8] Intel, *PHY Interface for The PCI Express 3.0 Architecture Revision .9*. 2010.

[9] *IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language*. IEEE Std 1800-2017, 2018.

[10] H. Foster, *The 2022 Wilson Research Group Functional Verification Study*. Siemens EDA, 2022.

[11] *IEEE Standard for Universal Verification Methodology Language Reference Manual*. IEEE Std 1800.2-2020, 2020.

[12] J. G. B. Wile and W. Roesner, *Comprehensive Functional Verification*. Elsevier, 2005.

[13] J. Bergeron, *Writing Testbenches using SystemVerilog*. Springer, 2006.

[14] C. Spear and G. Tumbush, *SystemVerilog for Verification*. Springer, 2012.

[15] A. B. Mehta, *ASIC/SoC Functional Design Verification*. Springer, 2018.

[16] S. EDA, *UVM Cookbook*. 2018.

[17] Accellera, *Universal Verification Methodology (UVM) 1.2 User's Guide*. 2015.

[18] R. Javali and J. T. Colaco, "Efficient implementation of high speed pci express mac transmitter with pipe interface," in *IEEE Sponsored 3rd International Conference on Electronics and Communication Systems*, 2016.

[19] K. N. Chandana and R. K. Karunavathi, "Link initialization and training in mac layer of pcie 3.0," *International Journal of Computer Science and Information Technologies*, vol. 6, no. 3, pp. 2717–2719, 2015.

[20] M. D. R. Geetanjali and G. Umesh, "Functional verification of mac-phy layer of pci express gen5.0 with pipe interface using uvm," in *2020 International Conference for Emerging Technology*, 2020.

[21] C. E. Cummings and J. Bergeron, "Using uvm virtual sequencers virtual sequences," *Sunburst Design*, 2016.

Appendices

Appendix I: Control Characters

These are special characters sent on the link for management purposes, each has its own 10-bit symbol, description, functionality, and properties as follows [5]:

Character Name	Symbol	Description	Functionality
COM	K28.5	First character of any ordered set	Used by Rx to achieve symbol and bit lock during link training. Have easily recognizable encodings: they start with two bits of one polarity followed by five bits of the opposite polarity (0011111010 or 1100000101)
PAD	K23.7	Packet filler	Used to fill extra lanes if a packet to be sent doesn't cover all the available lanes in a multi-lane link
SKP	K28.0	Part of SKIP ordered sets	Used in clock tolerance compensation
STP	K27.7	Start of a TLP	Inserted to identify the start of a TLP
SDP	K28.2	Start of a DLLP	Inserted to identify the start of a DLLP
END	K29.7	End of Good Packet (TLP or DLLP)	Appended to identify the end of an error- free TLP or DLLP
EDB	K30.7	End of a bad or 'nullified' TLP	Inserted to identify the end of a TLP that a forwarding device (such as a switch) wishes to 'nullify'. When a receiver sees a nullified packet, it discards the packet and does not return an ACK or NAK
FTS	K28.1	Part of the FTS Ordered Set	Sent by a device to recover a link from the L0s standby state back to the full-on L0 state
IDL	K28.3	Part of the EIOS	Sent to inform the receiver that the Link is transitioning into a low power state
EIE	K28.7	Part of the EIEOS	Added in the PCIe 2.0 spec and used to help an electrically idle link to begin the wake-up process.

Table 6.1: Control Characters

Appendix II: Ordered Sets

Series of characters that are neither TLPs nor DLLPs, but they are exchanged between link partners in certain times and amounts. They always start with a COM character, and are used for things like Link training, clock tolerance compensation, and changing Link power states as follows [5]:

Ordered Set Name	Width	Transfer Conditions	Functionality
Training Sequences (TS1 & TS2)	16 Symbols	Exchanged during link training and initialization process	Allow the Link partners to achieve bit lock and Symbol lock, negotiate the link speed, and report other variables associated with Link operation
Electrical Idle	4 Symbols	Sent prior to entering low-power state	Receivers detect this Ordered Set and prepare for the Link to go to into Electrical Idle by ignoring input errors until exiting from Electrical Idle. Shortly after sending EIOS, the Transmitter reduces its differential voltage to less than 20mV peak
Fast Training Sequencer (FTS)	4 Symbols	Sent to take a Link from the low-power L0s state back to the fully operational L0 state	The receiver detects the FTSs, recognizes that the Link is exiting from Electrical Idle, and uses them to recover Bit and Symbol Lock
SKP	4 Symbols	Transmitted at regular intervals for Clock Tolerance Compensation	the Receiver evaluates the SOS and internally adds or removes SKP Symbols as needed to prevent its elastic buffer from under-flowing or over-flowing
Electrical Idle Exit	4 Symbols	Sent to exit the electrical idle Link state in PCIe 2.0	This low-frequency pattern guarantees that the Receiver will properly detect an exit from Electrical Idle

Table 6.2: Ordered Sets