Functional Verification of PCIe 5.0 MAC Layer

A Graduation Project Thesis

By

Abd-Elrahman Mohammed Ali
Ibrahim Awad
Marwan Mohamed
Mostafa Masoud Ali
Moustafa Mohammed Raafat
Sohaib Alaraby
Yasser Mohamed Shokr
Youssef Ahmed Ali

Under the Supervision of

Dr. Mohamed Elbanna

Faculty of Engineering, Alexanderia University

Alexanderia, Egypt

July 2025

Abstract

PCIe Generation 5 is the latest evolution of the PCI Express standard, doubling the per-lane data rate over Gen4 to 32 GT/s. In this work, we develop a comprehensive verification environment to ensure the design meets timing, protocol, and performance requirements. We employ constrained-random UVM test benches, waveform monitors, and hardware-in-the-loop acceleration to cover both functional and corner cases. Results demonstrate full compliance with the official PCI-SIG specification. Finally, we discuss test coverage achieved and propose enhancements for next-generation PCIe IP cores.

Contents

List of Figures

List of Tables

1 List of Acronyms

- **PCIe**: Peripheral Component Interconnect Express
- **UVM**: Universal Verification Methodology
- **ECRC**: End To End Cyclic Redundancy Check
- **LCRC**: Link Cyclic Redundancy Check
- **EIOS**: Electrical Ideal Orderd Set
- **EIEOS**: Electrical Ideal Exit Orderd Set
- **TLP**: Transaction Layer Packet
- **DLLP**: Data Link Layer Packet
- **IDLA**: Logical Idle
- **EDB**: End Bad
- **STP**: Start TLP
- **SDP**: Start DLLP
- **CDR**: Clock and Data Recovery
- **SOS**: SKP Ordered Set
- **EDS**: Ending of Data Stream
- **FTS**: Fast Training Sequence
- **TS1**: Training Sequence 1
- **TS2**: Training Sequence 2
- **LTSSM**: Link Training and Status State Machine
- **DUT**: Design Under Test
- **QoS**: Quality of Service
- **PIPE**: PHY Interface for PCI Express
- **LPIF**: Link Physical Interface
- **SSD**: Solid State Drives

2 Introduction

2.1 Overview

PCIe (Peripheral Component Interconnect Express) is a high-speed interface standard used to connect devices like GPUs, SSDs, and network cards to the motherboard. It uses scalable lanes—x1, x4, x8, x16—where more lanes mean higher bandwidth.

PCIe is backward compatible and has evolved through several generations, with Gen 5 reaching up to 128 GB/s. As a verification team, our focus is on ensuring the correctness and compliance of the Gen 5 physical layer through functional and protocol verification, covering key aspects like lane negotiation, link training, and error handling.

2.2 PCI and PCI-X

To understand PCI Express (PCIe), it's helpful to first look at PCI and PCI-X. PCI was created in the early 1990s to replace the slower ISA bus, offering better speed and easier device setup. It became a standard thanks to features like configuration space for better software control.

PCI-X came next, improving PCI's speed while staying compatible with older devices. But as a parallel bus, PCI-X had limits—like too many pins and signal issues—that made it hard to scale further.

To fix these problems, PCIe switched to a faster, more efficient serial design. Importantly, PCIe still supports older PCI software, making upgrades easier and cheaper. So, knowing PCI and PCI-X helps in understanding how PCIe works.

2.3 PCI Basics

- **Basics of a PCI-Based System** The older system based on a PCI bus. The system includes a North Bridge, located "north" of the central PCI bus in typical diagrams, which interfaces between the processor and the PCI bus. The North Bridge connects to the processor bus, system memory bus, AGP graphics bus, and the PCI bus. Multiple devices share the PCI bus, either connected directly or via add-in card connectors.

A South Bridge connects the PCI bus to system peripherals such as the ISA bus, which supported legacy devices for several years. The South Bridge typically also serves as the central PCI resource, providing essential system signals such as reset, reference clock, and error reporting.

2.4 Introducing PCI-X

PCI-X is backward compatible with PCI and improves performance via higher clock rates (up to 133 MHz), PLL-based timing, and registered inputs. These enhancements support faster signaling while preserving software compatibility.

2.4.1 PCI-X System Example

Figure 1 shows an Intel 7500-based platform using three PCI-X bridges, each supporting two buses at up to 133 MHz. While bandwidth improved, the need for more buses and components increased system cost and complexity.

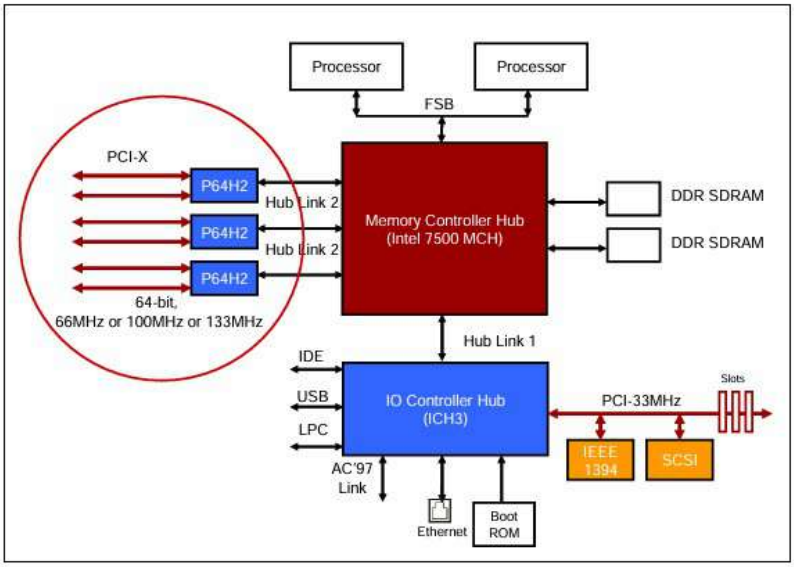


Figure 1: 66 MHz / 133 MHz PCI-X Bus Based Platform

2.4.2 PCI-X Transactions and Features

PCI-X uses burst transfers (typically 128 bytes) with fixed-size attributes, eliminating mid-transfer wait states. It introduces split transactions, where a requester and completer decouple read cycles to improve bus utilization. This protocol raises efficiency to around 85%, compared to 50–60% with standard PCI.

2.5 PCI Evolution: From Parallel to Serial Bus

2.5.1 Limitations of Parallel Bus (PCI, PCI-X)

Traditional parallel buses like PCI and PCI-X face major scalability issues at higher clock speeds due to:

- **Signal skew:** Data bits arrive at slightly different times.
- **Clock skew:** Clock signal reaches devices at different times.
- **Flight time:** Signal propagation delays reduce timing margin.

2.5.2 PCI-X 2.0 and Source-Synchronous Clocking

To overcome limitations and increase bandwidth, PCI-X 2.0 introduced a **source-synchronous model** supporting DDR/QDR modes. The transmitting device provides a *strobe* signal that travels with the data and is used for latching by the receiver.

3 PCI Express (PCIe) Overview

3.1 PCI Express (PCIe)

PCIe replaced the parallel bus with a high-speed serial model, while maintaining **software backward compatibility** with PCI. It uses a **dual-simplex architecture** with independent transmit and receive paths.

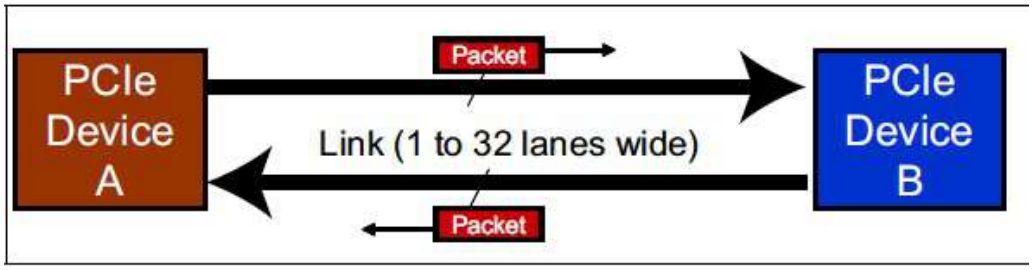


Figure 2: Dual-Simplex Link Architecture

A **Link** consists of one or more **Lanes**, where each Lane is a pair of differential signals for TX and RX. The number of lanes (x1, x2, x4, x8, etc.) defines the **Link Width** and influences bandwidth, cost, and power.

3.2 Serial Model Advantages

PCIe overcomes parallel bus limitations through:

- **Embedded clocking:** Clock is integrated into the data stream (no external clock needed).
- **No clock or signal skew:** Serial transfer eliminates timing uncertainties.
- **Higher signaling rates:** Supports speeds like 2.5, 5.0, and 8.0 GT/s reliably.

3.3 Bandwidth

PCIe bandwidth depends on generation and encoding:

- **Gen1:** 2.5 GT/s with 8b/10b encoding $\Rightarrow$ 0.25 GB/s per Lane (0.5 GB/s bidirectional).
- **Gen2:** 5.0 GT/s $\Rightarrow$ 0.5 GB/s per Lane.
- **Gen3:** 8.0 GT/s using 128b/130b encoding (reduced overhead, doubled throughput).

3.4 Differential Signaling

Each Lane uses **differential pairs** (D+ and D−), improving:

- **Noise immunity**
- **Signal integrity at high frequencies**

3.5 Clock Recovery in PCIe

PCIe embeds the clock in the data stream using 8b/10b encoding, eliminating the need for a shared clock. The receiver recovers the clock with a PLL that synchronizes to the transmitter's timing and compensates for voltage or temperature variations. The recovered clock is then used to latch data into the deserializer.

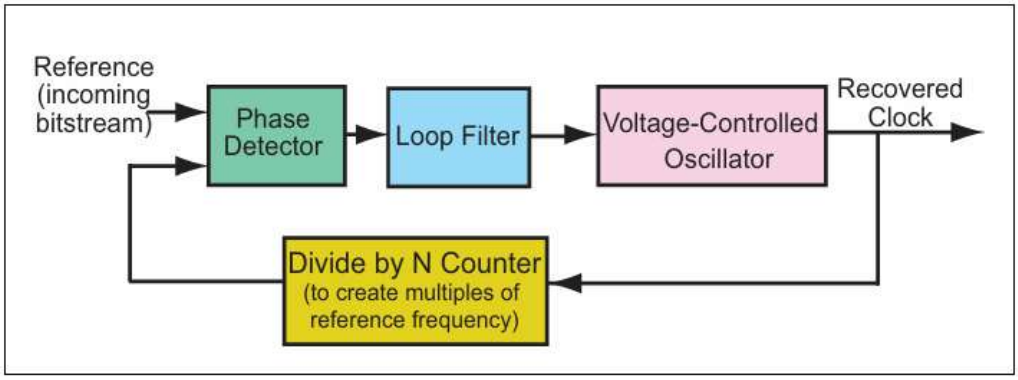


Figure 3: Simple PLL Block Diagram

3.6 Packet-Based Communication

PCIe reduces physical pins by switching from parallel to serial communication and transmitting all data as structured packets. The receiver detects packet boundaries and decodes packet formats to process the data without relying on side-band signals.

3.7 PCIe Topology

PCIe follows a tree topology with the CPU at the root, maintaining compatibility with legacy PCI software. The Root Complex (RC) connects the CPU to the PCIe fabric via Root Ports. Switches route packets between devices, and bridges enable backward compatibility with legacy PCI or PCI-X buses.

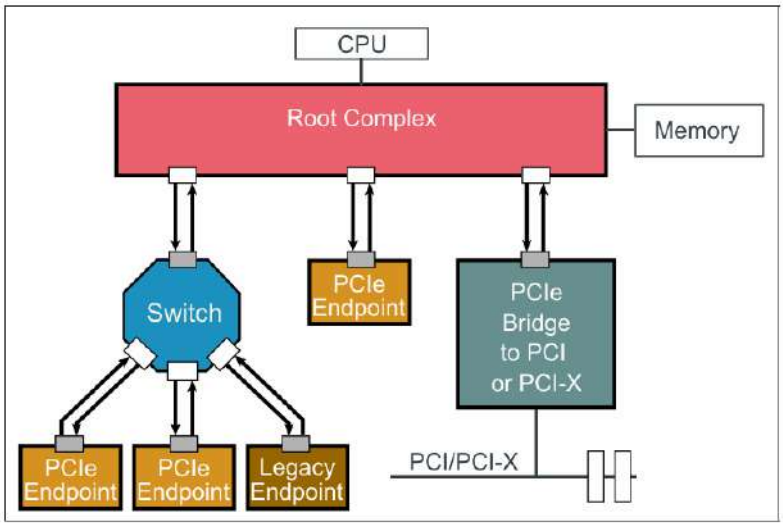


Figure 4: PCIe Topology

- **Endpoints** Endpoints are PCIe devices at the tree's leaves, each with an Upstream Port. They are categorized as:
  - **Legacy PCIe Endpoints:** PCI or PCI-X devices with PCIe interfaces, sometimes using legacy features like IO space.
  - **Native PCIe Endpoints:** Modern PCIe devices designed for PCIe, typically memory-mapped I/O (MMIO).

3.8 Introduction to Device Layers

The PCIe architecture is organized into layers, each with separate transmit (TX) and receive (RX) components for handling outbound and inbound traffic, respectively. This layered structure helps hardware designers upgrade systems more easily by isolating changes to specific layers. Although designs aren't required to strictly follow this partitioning, the layers help define clear interface responsibilities.
The following section outlines the responsibilities assigned to each layer and details the sequence of events involved in accomplishing a data transfer:

- **Device core and interface to Transaction Layer:** deliver the main functionality of the PCIe device. For endpoints, the core can support up to eight functions, each with its own configuration space. In switches, the core contains packet routing logic and an internal bus to forward data. In root complexes, the root core creates a virtual PCI bus 0 that hosts all chipset-integrated endpoints and virtual bridges.
- **Transaction Layer:** is responsible for Transaction Layer Packet (TLP) creation on the transmit side and TLP decoding on the receive side. This layer is also responsible for Quality of Service functionality, Flow Control functionality and Transaction Ordering functionality.
- **Data Link Layer:** is responsible for Data Link Layer Packet (DLLP) creation on the transmit side and decoding on the receive side. Additionally, it manages link error detection and correction through the acknowledgment and negative acknowledgment (Ack/Nak) protocol.
- **Physical Layer:** comprises three primary components of the transmit side, the receive side, and the Link Training and Status State Machine (LTSSM), each responsible for distinct functions essential to reliable link operation.
  - **Transmit Side:** Prepares all packet types (TLPs, DLLPs, Ordered Sets) for transmission through processes like byte striping, scrambling, and encoding (8b/10b for Gen1/2 or 128b/130b for Gen3). It then serializes and sends the data using differential signaling.
  - **Receive Side:** Converts incoming differential signals back into digital form, recovers the clock, deserializes the data stream, and performs decoding, de-scrambling, and byte un-striping to reconstruct the original packets.
  - **LTSSM:** Controls link setup and maintenance by managing initialization, negotiating link width and speed, synchronizing endpoints, and transitioning through various states to maintain a stable connection.

Every PCIe interface, including those in switch ports Figure 6, implements all protocol layers—even though switches mainly forward packets. This is because routing decisions require the Transaction Layer to inspect packet contents.

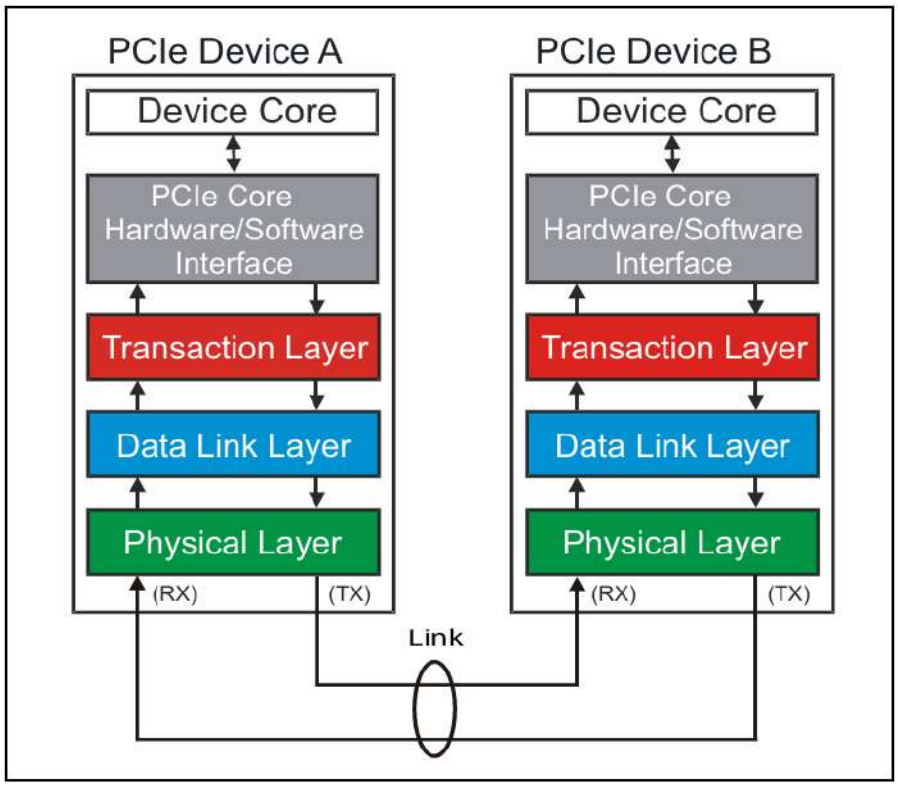


Figure 5: PCIe Device Layers

Each layer on one device communicates with its corresponding layer on the device at the other end of the link. The upper layers (Transaction and Data Link) handle this by formatting data into structured packets that follow known patterns recognizable by the receiving peer layer. These packets pass through intermediate layers during transmission.

The Physical Layer also communicates with its counterpart on the other device, but unlike the upper layers, it does so through direct electrical signaling rather than packet-based communication manner.

3.9 Device Layer Interaction Overview

- **Transaction Layer (Sender):**
  - Constructs Transaction Layer Packet (TLP) using command, address, and metadata.
  - Stores TLP in Virtual Channel buffer.
- **Data Link Layer (Sender):**
  - Appends sequence number and CRC.
  - Retains a copy for retransmission if errors occur.
- **Physical Layer (Sender):**
  - Encodes and transmits TLP over differential signaling lanes.

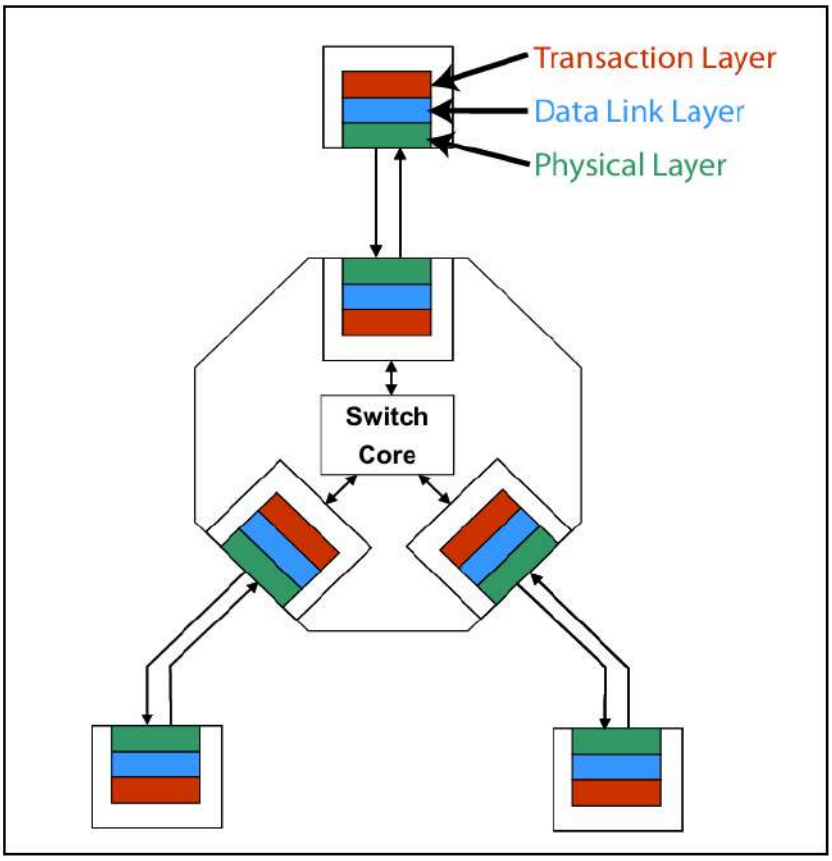


Figure 6: Switch Port Layers

- **Physical Layer (Receiver):**
  - Decodes bitstream and performs initial error checks.
- **Data Link Layer (Receiver):**
  - Validates sequence and CRC.
  - Forwards verified packets to Transaction Layer.
- **Transaction Layer (Receiver):**
  - Buffers, checks, and disassembles TLP.
  - Passes data and commands to the device core.

This structured interaction ensures reliable communication with layered error handling and recovery.

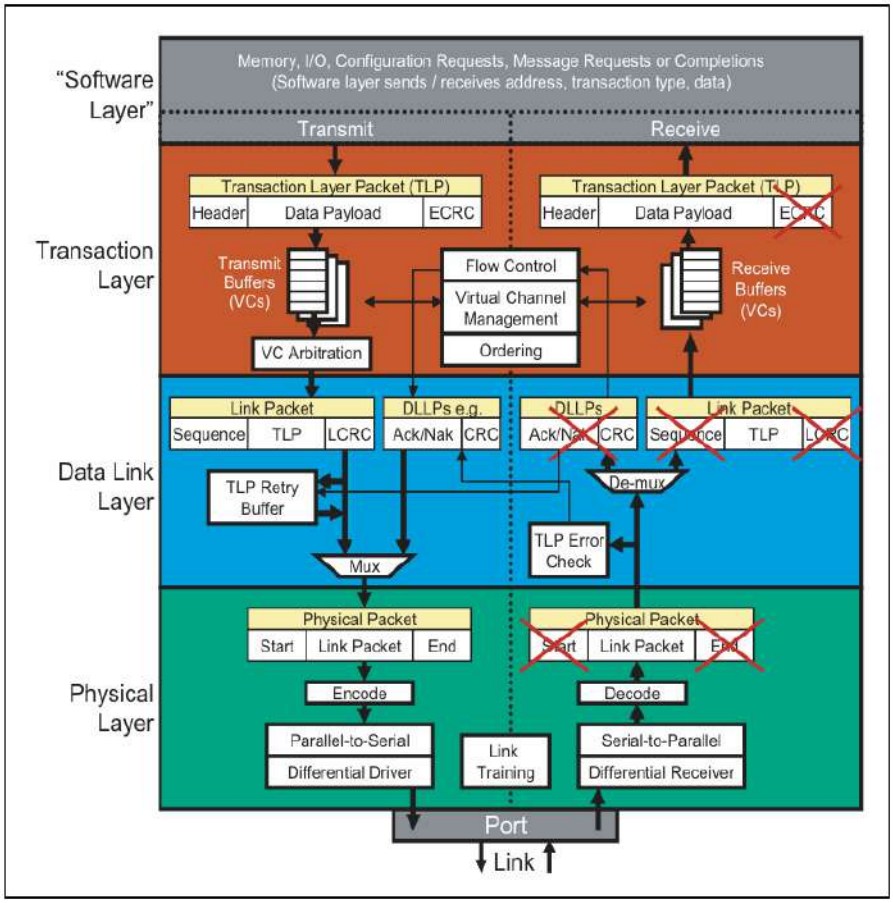


Figure 7: PCI Express Device Layer Stack

3.10 Device Core/Software Layer

The Device Core functions above the Transaction Layer and serves as:

- **Outbound:** Constructs Transaction Layer Packets (TLPs) based on request types like Memory, I/O, Configuration, and Messages.
- **Inbound:** Parses received TLPs and forwards the relevant information to the Software Layer.

It essentially drives the data flow and defines the device's primary behavior in the PCIe communication stack.

3.11 Transaction Layer

The Transaction Layer manages packet creation and interpretation in response to commands from the Software Layer:

- The source of outgoing requests, supplying details like transaction type, target address, and data size.
- The destination for incoming data, which it receives from the Data Link Layer after packet processing.

Key Features:

- Split Transaction Protocol: For non-posted transactions, it matches a Completion packet with its earlier Request.
- **TLP Categories:**
  - Memory, I/O, and Configuration: Inherited from PCI/PCI-X.
  - Messages: Introduced in PCIe for signaling between components.

Transaction Types:

- **Non-Posted (require response):**
  - Memory Reads
  - I/O Writes
  - Configuration Writes
  - Must receive Completion TLPs confirming success.
- **Posted (no response required):**
  - Memory Writes
  - Messages
  - Improve performance by avoiding response delays, though they don't confirm delivery.

Even posted transactions rely on the Data Link Layer's Ack/Nak protocol for reliable delivery at the packet level, despite not needing a Completion response at the Transaction Layer.

3.11.1 TLP Basics

TLPs originate at the Transaction Layer of a transmitter and terminate at the Transaction Layer of the receiver, as shown in Figure 7. The Data Link Layer and Physical Layer add parts to the packet as it moves through the layers of the transmitter, and then verify at the receiver that those parts were transmitted correctly across the Link.

3.11.2 TLP Packet Assembly

- **Transaction Layer:**
  - Device core sends required information to assemble the core of the TLP.
  - Each TLP contains a header (always present), and may include an optional ECRC field.
  - ECRC (End-to-End CRC):
    - 32-bit CRC provides robust error detection, including burst errors.
    - Passed unchanged through service points validated at the destination.
- **Data Link Layer:**
  - Adds a Sequence Number and LCRC field to the packet.
  - LCRC (Link CRC) checks for transmission errors between adjacent nodes.
  - Receiver reports any error detection to the transmitter.
- **Physical Layer:**
  - Adds framing characters to define packet boundaries As shown in Figure 8.
  - **Gen1/Gen2:** Use control characters at start and end.
  - **Gen3/Gen4/Gen5:** Control characters no longer used.

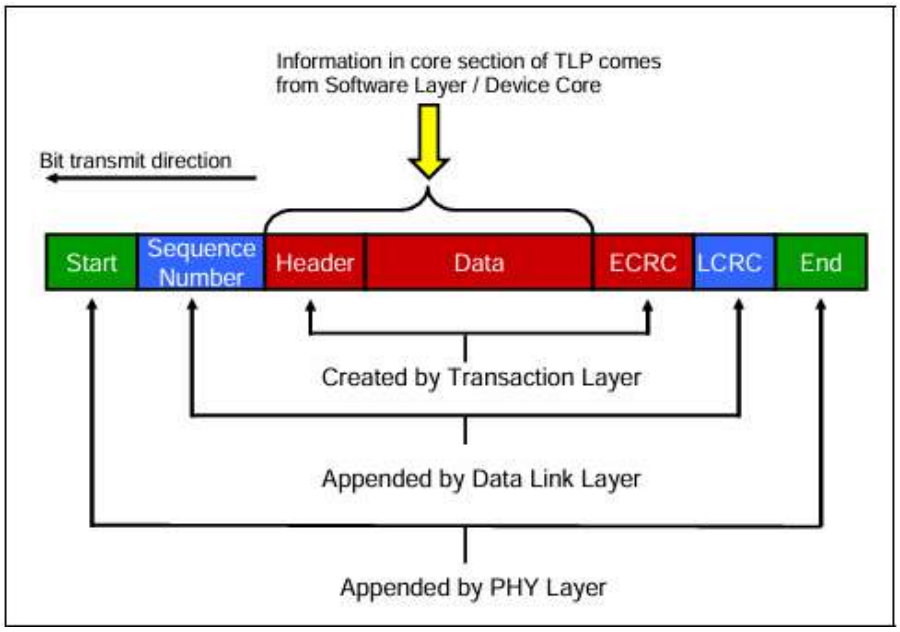


Figure 8: TLP Assembly

3.11.3 TLP Packet Disassembly

Upon receiving a TLP bitstream, the receiver begins disassembly to recover the original data as shown in Figure 9.

- **Physical Layer:**
  - Verifies presence of Start and End characters (applicable to Gen1 and Gen2).
  - Removes framing characters and forwards the packet to the Data Link Layer.
- **Data Link Layer:**
  - Checks for LCRC and sequence number errors.
  - If valid, removes these fields and forwards the packet to the Transaction Layer.
- **Transaction Layer:**
  - **If the receiver is the target device:**
    - Checks the ECRC field for errors.
    - Forwards the validated packet to the core network.
  - **If the receiver is a switch:**
    - Analyzes the TLP header for routing information.
    - Forwards the packet to the appropriate port.
    - May check and report ECRC errors but cannot modify the ECRC field.

3.11.4 Difference Between LCRC and ECRC

In PCIe, LCRC (Link CRC) is added at the Data Link Layer to detect transmission errors between directly connected devices. It is recalculated at each hop. ECRC (End-to-End CRC), added optionally by the Transaction Layer, remains unchanged across the entire path and detects errors that occur beyond individual links, ensuring end-to-end data integrity.

3.11.5 Quality of Service (QoS)

PCIe supports Quality of Service (QoS) to ensure timely data delivery for time-sensitive applications like streaming video or audio. QoS is achieved through several mechanisms:

- **Traffic Class (TC):** Each packet is assigned a 3-bit priority level by software. Higher-numbered TCs generally indicate higher priority.
- **Virtual Channels (VC):** Each port contains multiple hardware buffers (VCs). Packets are placed into VCs based on their TC.
- **VC Arbitration:** When multiple VCs have data ready, hardware logic selects which VC's packet to transmit.
- **Port Arbitration:** Switches resolve competition among input ports for access to output VCs. Arbitration can be hardware-based or software-configured.

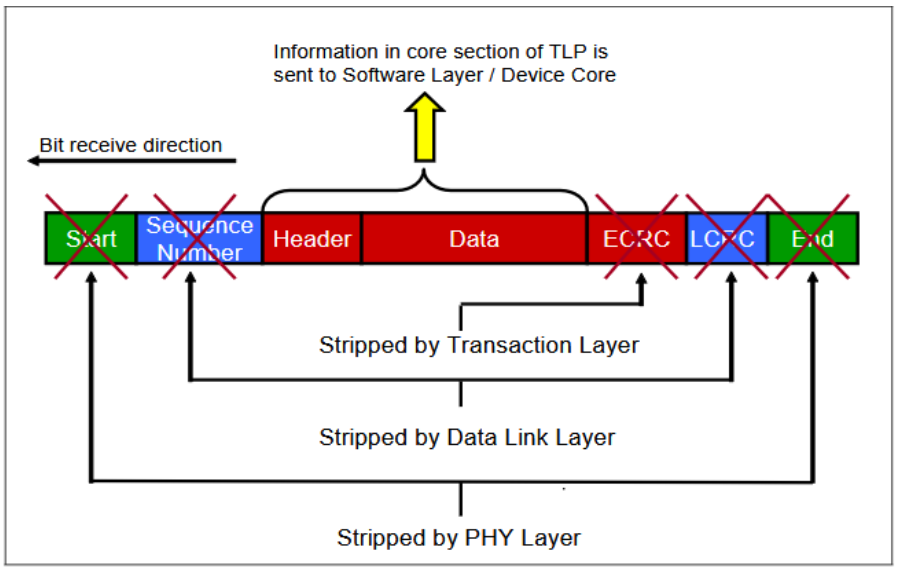


Figure 9: TLP Disassembly

These components work together to ensure high-priority traffic (e.g., video data) gets timely access to system resources, while lower-priority traffic (e.g., SCSI data) can tolerate delays.

3.11.6 Transaction Ordering

PCIe ensures packet ordering within the same Virtual Channel (VC) and Traffic Class (TC) to avoid deadlock or live-lock. The Transaction Layer maintains this ordering, while packets with different TCs have no guaranteed order.

3.11.7 Flow Control

Flow control is a protocol used by serial transports to manage the flow of packets between the sender and receiver. It ensures smooth transport operation and helps avoid potential problems such as receiver overflow and packet loss, which cause performance-wasting events in PCI, like disconnects and retries.

In PCIe flow control, the receiver must advertise the size of its VC buffer (a special buffer where incoming TLPs are stored until they are processed and removed) whenever a TLP is added or removed. The PCIe receiver uses a special type of packet called Flow Control DLLPs to inform the transmitter of the available space in its VC buffer, maintaining proper flow control. Flow control is completely handled by hardware and is transparent to software.

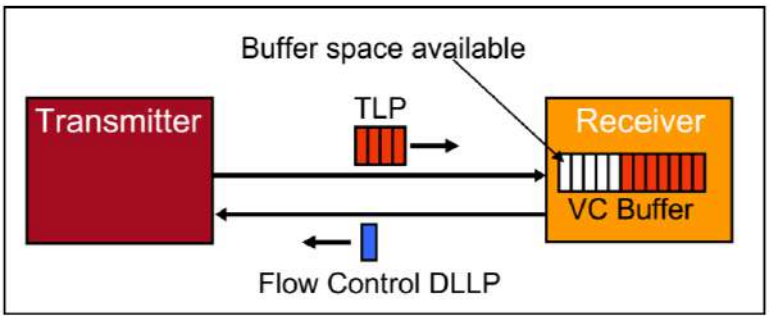


Figure 10: Flow Control

3.12 Data Link Layer

The Data Link Layer is responsible for link management. As part of this, it performs three major functions: TLP error correction, flow control, and link power management. DLLPs are generated in this layer and are used to accomplish link management.

3.12.1 What are DLLPs?

DLLPs is an abbreviation for Data Link Layer Packets which are packets transferred between the Data Link Layers of neighboring devices. DLLPs are not routed beyond these neighboring devices. Their size is fixed at approximately 8 bytes, which is considered small compared to TLPs. This small size is beneficial, as it reduces the overhead of these control packets.

3.12.2 DLLP Assembly and Disassembly

DLLPs are produced by the Data Link Layer of the transmitter. Initially, a DLLP consists of two main parts: the DLLP core and a 16-bit CRC used for error checking. The packet is then forwarded to the Physical Layer of the transmitter, which adds start and end fields to the packet and transmits it differentially across all available lanes. At the receiver side, this process is reversed. The framing characters are removed, and the DLLP is extracted and consumed by the Data Link Layer of the receiver.

3.12.3 Ack/NAK Protocol

The Ack/NAK protocol is essential for error correction and primarily used in a hardware-based automatic retry mechanism. Certain fields are added to the TLP to assist the receiver in performing error checking, such as the LCRC and Sequence Number.

How does the Ack/NAK protocol work?

The transmitter sends a TLP and saves a copy in a special buffer called the Replay Buffer. When the receiver receives the TLP and performs error checking using the LCRC field, it sends an ACK DLLP to the transmitter if no error is found. This ACK DLLP includes the Sequence Number of the last successfully received TLP. Upon receiving the ACK, the transmitter deletes that TLP (and all earlier ones) from its Replay Buffer.

If the receiver detects a TLP error, it drops the TLP and sends a NAK DLLP to the transmitter. This is treated as a replay request. The transmitter then retransmits all unacknowledged TLPs until they are successfully acknowledged.

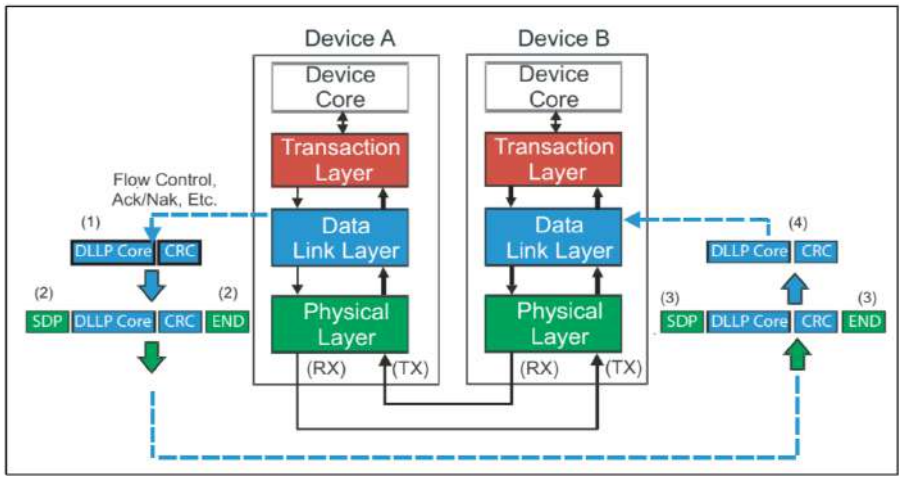


Figure 11: DLLP Origin and Destination

3.13 The Physical Layer

The Physical Layer is the lowest hierarchical layer. Both TLP and DLLP packets are forwarded down to it at the transmitter and up from it at the receiver. This layer is divided into two parts:

- **Logical Part:** which prepares packets for serial transmission and reverses the process for received packets.
- **Electrical Part:** which is the analog interface that connects directly to the Link and contains differential drivers and receivers for each lane.

3.13.1 Physical Layer – Logical Part

The Logical Physical Layer is responsible for preparing TLPs and DLLPs for transmission over the serial PCIe link and recovering them at the receiver. It performs the following steps:

- **Framing:**
  - Adds Start and End characters (framing characters) to both TLPs and DLLPs.
  - Helps the receiver detect where packets begin and end.
  - Data Block Frame Construction:
    - Data Blocks comprise TLPs, DLLP, and Tokens that are used to deliver the information.
    - Five types of Data structures (called Tokens) are also used within a Data Block.
    - Three of the token may be sent at the beginning of a block .These include:
      - Start TLP (STP) followed by a TLP.
      - Start DLLP (SDP) followed by a DLLP.
      - Logical Idle (IDLA) is sent when there is no packet activity.
    - The remaining Tokens are delivered at the end of the Data Block:
      - End of Data Stream (EDS) precedes the transition to Ordered Sets.
      - End Bad (EDB) reports a nullified packet has been detected.
- **Byte Striping:**
  - Distributes each byte of the packet across all available lanes.
  - Each lane acts as an independent serial channel, allowing parallel data transfer.
  - At the receiver, the striped bytes are reassembled back into the original order.
- **Scrambling:**
  - Scrambles each byte before transmission to eliminate long sequences of the same bit pattern.
  - This reduces electromagnetic interference (EMI) and improves signal integrity.

Table 1: Gen3 Scrambler Seed Values

Lane	Seed Value
0	1DBFBCh
1	0607BBh
2	1EC760h

Lane	Seed Value
3	18C0DBh
4	010F12h
5	19CFC9h
6	0277CEh
7	1BB807h

- scrambling replaces the 8b/10b encoding used in earlier generations to maintain DC balance and ensure enough signal transitions for clock recovery. This new scrambling doesn't guarantee perfect balance over short times but works well over longer periods. To reduce the chance of signal interference between Lanes, each Lane uses a different scrambling pattern. There are two ways to do this: the first uses a separate scrambler (LFSR) with a unique starting value (seed) for each Lane, which is faster but uses more hardware and this is supported in the design. The second uses one shared LFSR for all Lanes, creating different outputs by combining bits differently for each Lane using XOR logic. This second option is cheaper but slower due to added complexity.

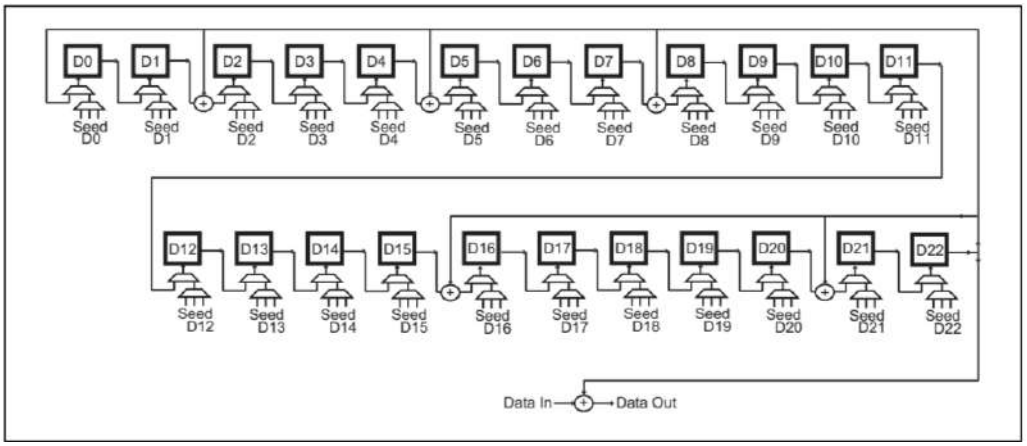


Figure 12: Gen3 Per-Lane LFSR Scrambling Logic

- In PCIe Gen3 - Gen5, scrambling using LFSRs follows specific rules to ensure signal integrity and proper operation. The LFSRs advance only based on the transmitted content and are re-initialized upon detecting EIEOS or FTSOS. Sync Header bits are never scrambled and do not affect the LFSR state. For TS1 and TS2 Ordered Sets, Symbol 0 bypasses scrambling, Symbols 1–13 are scrambled, and Symbols 14–15 may be scrambled depending on DC balance needs. Ordered Sets like FTS, SDS, EIEOS, EIOS, and SOS bypass scrambling entirely, but Transmitters still advance the LFSRs for their Symbols—except in SOS, where neither Transmitters nor Receivers advance them. All Data Block Symbols are scrambled in little-endian order. LFSR seed values are lane-specific and determined during LTSSM's entry into Configuration.Idle, remaining fixed as long as LinkUp is maintained. Unlike 8b/10b, scrambling in 128b/130b cannot be disabled due to its critical role in maintaining signal integrity, and Loopback Slaves must not scramble or de-scramble looped-back bits.

- **Encoding and Transmission**
  - PCIe uses different encoding schemes based on the generation.

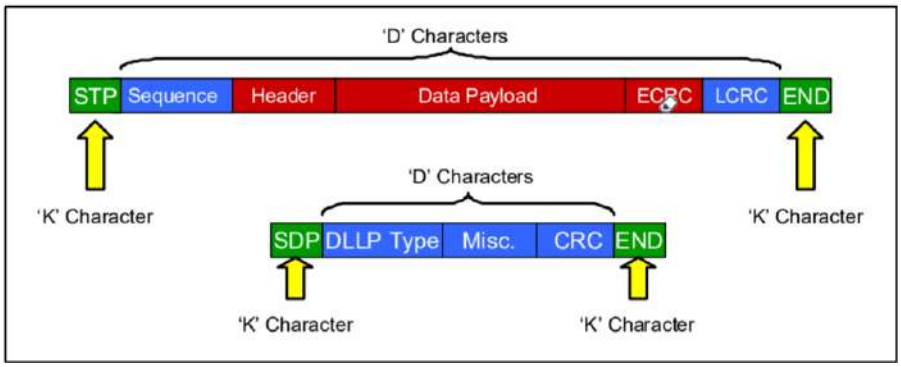


Figure 13: 8b/10b Lane Encoding

– **Gen1 and Gen2:** Use 8b/10b encoding, which converts each 8-bit into a 10-bit symbol. Adds overhead, but provides good signal properties and error detection.

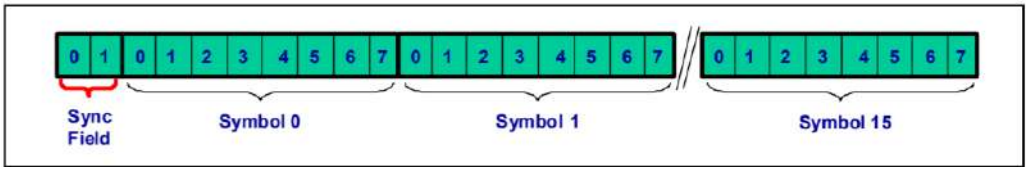


Figure 14: 128b/130b block encoding

– **Gen3 and later:**
  * Use 128b/130b encoding, which is more efficient (less overhead). This scheme skips 8b/10b encoding entirely and relies on scrambling and block lock instead.
  * 128b/130b encoding does not affect bytes being transferred; instead, the characters are grouped into blocks of 16 bytes with a 2-bit Sync field at the beginning of each block. The 2-bit Sync field specifies whether the block includes Data (10b) or Ordered Sets (01b). The ordered sets are similar to the 8b/10b version in that they must be driven on all the lanes simultaneously
– The encoded data is serialized and transmitted differentially over each lane at:
  2.5 GT/s, 5 GT/s, 8 GT/s, 16 GT/s or 32 GT/s

• **Serialization:** In PCIe Gen3 - Gen5, the serializer handles 8-bit data (10-bit in Gen1/Gen2) and shifts it out serially. On the receiver side, Clock and Data Recovery (CDR) extracts timing using a PLL.

Reception Process At the receiving side, the process is reversed:

• **Deserialization:**
  – Bits are converted back into symbols or bytes depending on the encoding scheme.
• **Elastic Buffer:**
  – Used to compensate for small timing differences between the sender and receiver's internal clocks.
  – Prevents data loss due to clock mismatches.
• **Decoding:**
  – Gen1/Gen2: Perform 8b/10b decoding to recover the original bytes.
  – Gen3+: Skip 8b/10b decoding since data was encoded using 128b/130b.
• **De-scrambling and Un-striping:**
  – Data is de-scrambled to recover the original content.
  – Un-striping reassembles bytes from different lanes into a single continuous data stream.
• **Block Alignment:**
  – Gen3 achieves Bit Lock first and then attempts to establish Block Alignment locking. This requires receivers to find the Sync Header that demarcates the Block boundary .
  – Transmitters establish this boundary by sending recognizable EIEOS patterns consisting of alternating bytes of 00h and FFh .
  – The use of EIEOS has expanded from simply exiting Electrical Idle to also serving as the synchronizing mechanism that establishes Block Alignment .

Data Transfere: The Link enters a Data Stream by sending an SDS Ordered Set and transitioning to the L0

Link state. While in a Data Stream multiple Data Blocks are transferred, until the Data Stream ends with an EDS Token (unless an error ends it early).

3.14 Framing Requirements and Error Handling in PCIe

3.14.1 Transmitter Framing Requirements

During a **Data Stream** (from first symbol after SDS to just before any non-SOS Ordered Set or upon Framing Error), the transmitter must follow these rules:

- **TLP Transmission**:
  - An `STP` Token must be immediately followed by the entire TLP.
  - If the TLP is nullified, an `EDB` Token follows it, but is excluded from the TLP length.
  - Only one `STP` may be sent per Symbol Time.
- **DLLP Transmission**:
  - An `SDP` Token must be immediately followed by the complete DLLP.
  - Only one `SDP` may be sent per Symbol Time.
- **Sending SOS (SKP Ordered Set)**:
  - Insert an `EDS` Token in the last dword of the current Data Block.
  - Send the `SOS` as the next Ordered Set Block.
  - Resume the Data Stream with a new Data Block.
  - Multiple `SOS` must be separated by valid Data Blocks.
- **Ending a Data Stream**:
  - Send an `EDS` Token followed by either `EIOS` (for low-power entry) or `EIEOS` (in all other cases).
- **Use of `IDL` Tokens**:
  - Must be transmitted if no framing token (TLP/DLLP) is being sent.
  - In multi-lane links, the first symbol of a new TLP/DLLP must begin in Lane 0 after IDL.
  - `IDL` fills unused lanes within a Symbol Time.

3.14.2 Receiver Framing Requirements

When a receiver detects a Data Stream, it must apply the following rules:

- **TLP Reception (`STP`)**:
  - Check Frame CRC and parity; mismatches are Framing Errors.
  - The symbol after the TLP is the next token; check for `EDB`.
  - Optionally, a zero-length TLP or multiple STPs per Symbol Time are Framing Errors.
- **`EDB` Reception**:
  - Must be received immediately after a TLP.
  - Any incorrect symbols in the EDB sequence indicate a Framing Error.
- **`EDS` Reception**:
  - Terminates the current Data Stream.
  - Only `SKP`, `EIOS`, or `EIEOS` Ordered Sets are valid afterward.
  - Receiving another Data Block immediately is a Framing Error.
- **DLLP Reception (`SDP`)**:

  - Followed directly by DLLP content; next token is after it.
  - Optionally, multiple SDPs in the same Symbol Time are Framing Errors.
- **IDL Reception**:
  - Next token can begin on any DW-aligned Lane (Lane 0 for x4 or narrower).
  - Only IDL or EDS tokens are allowed in the same Symbol Time.
- **Illegal Conditions** (Framing Errors):
  - Ordered Set immediately after SDS.
  - Illegal sync headers (11b or 00b).
  - Ordered Set received without preceding EDS.
  - Data Block immediately follows an EDS.
  - Optionally, non-uniform Ordered Sets across lanes.

3.14.3 Recovery from Framing Errors

If a Framing Error is detected:

- Report a Receiver Error.
- Stop processing the current Data Stream.
- Wait for the next SDS to start a new Data Stream.
- Transition the Link Training and Status State Machine (LTSSM) to Recovery state if in L0.
- Expected recovery time is less than 1 μs.
- Link Layer Ack/Nak mechanism is not necessarily triggered unless packet loss is detected.

3.15 Gen3 Physical Layer Transmit Logic

Figure 15 illustrates a block diagram of the Physical Layer transmit logic that supports Gen3 speeds.

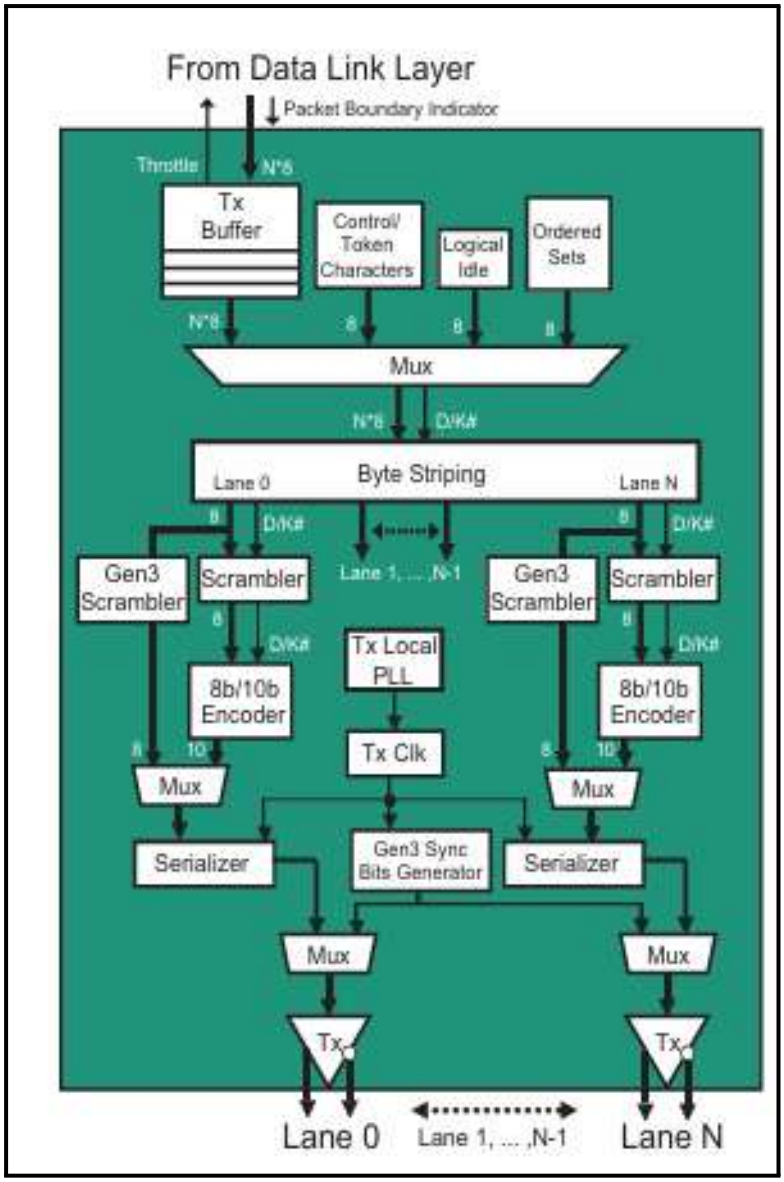


Figure 15: Gen3 Physical Layer Transmitter Details

3.15.1 Multiplexer

The Multiplexer inserts STP/SDP tokens to mark the start of TLPs and DLLPs received from the Data Link Layer. In Gen3, TLP boundaries are determined by the Length field in the STP token, eliminating the need for an explicit end-of-frame character. When a data stream ends or before sending an Ordered Set, an EDS token is inserted. Periodically, SKP Ordered Sets are added based on a timer. Other Ordered Sets (TS1, TS2, FTS, etc.) are inserted based on link requirements and lie outside the Data Stream.

Data is transmitted in blocks, each prefixed with a 2-bit Sync Header, which is duplicated across all lanes by the byte striping logic. When no data is available but the link must stay in L0, IDL (idle) tokens are used as filler and scrambled like regular data.

3.15.2 Byte Striping Logic

This logic spreads the bytes to be delivered across all the available Lanes.

Consider the example shown in Figure 16, where a 4-Lane Link is illustrated.the Sync Header bits appear on all the Lanes at the same time when a new Block begins and define the block type (a Data Block in this example).

Block encoding is handled independently for each Lane, but the bytes (or symbols) are striped across all the Lanes just as they were for the earlier generations of PCIe.

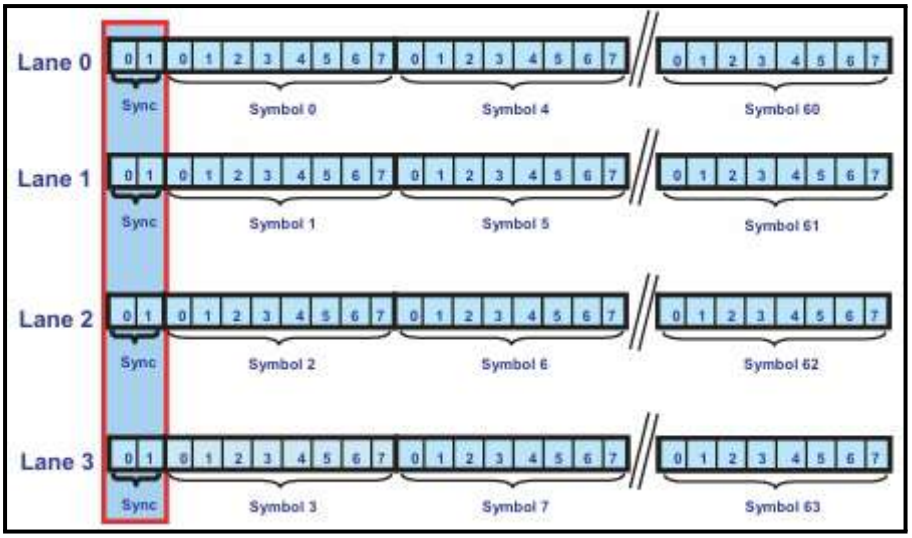


Figure 16: Gen3 Byte Striping x4

3.15.3 Link Training and Initialization

This is an automatic setup process run by the Physical Layer after power-up or reset to prepare the Link for communication. It determines the best link configuration and ensures synchronization.
Some important Steps achieved in this stage:

- **Link Width Negotiation**
  - Determines how many lanes will be used (x1, x4, x8, x16, x32).
- **Link Speed Selection**
  - Chooses the maximum supported data rate (e.g., Gen1, Gen2, Gen3).
- **Lane Reversal Handling**
  - Detects and corrects if lanes are physically connected in reverse order.
- **Polarity Inversion Detection**
  - Handles reversed polarity on differential pairs.
- **Bit Lock Establishment**
  - Recovers the transmitter clock to interpret incoming data bits.
- **Symbol Lock Detection**
  - Identifies the proper starting point in the incoming bit stream.
- **Lane-to-Lane De-skewing**
  - Aligns data from multiple lanes to correct for timing skew between them.

At the end of this stage, the pair of PCIe devices is ready to transfer data to each other reliably.

3.15.4 Physical Layer – Electrical Part

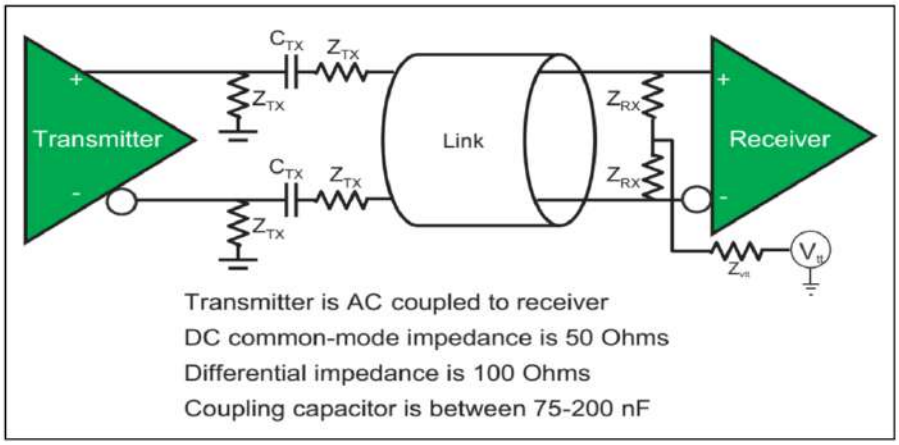


Figure 17: Physical Layer Electrical

The Electrical part of the PCIe Physical Layer is the analog interface responsible for transmitting and receiving serialized data over the PCIe Link. **AC-Coupled Link**

- PCIe Links use AC-coupling, which places capacitors in the signal path between devices.
- These capacitors block low-frequency (DC) signals while allowing high-frequency (AC) components to pass.
- AC-coupling:Allows voltage isolation between transmitter and receiver.It also supports common-mode voltage compatibility and Enhances signal integrity across various device implementations.

3.15.4.1 Transmitter Equalization: 3-Tap FIR Filter

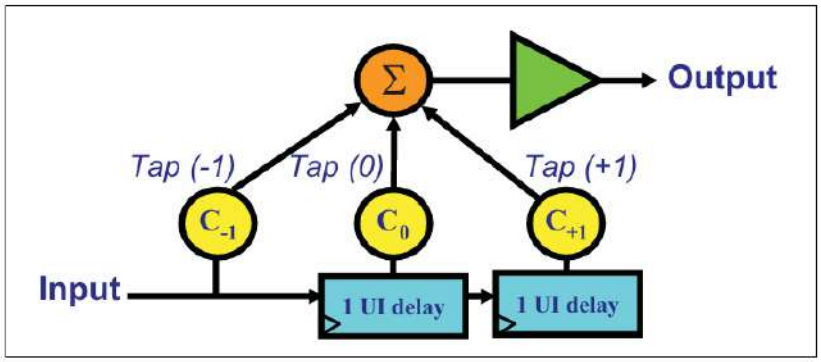


Figure 18: 3-Tap Tx Equalizer

To improve wave shaping at the transmitter, the PCIe specification requires the use of a **3-tap Finite Impulse Response (FIR) filter**. This type of filter is commonly used in high-speed SERDES applications above 6.0 Gb/s and is well-suited for PCIe, as it helps compensate for the channel's tendency to spread the signal over time. The 3-tap FIR filter generates its output based on three bit-time spaced versions of the input:

- **C0 (cursor)** – the main, undelayed input.
- **C−1 (pre-cursor)** – the input delayed by one unit interval (UI).
- **C+1 (post-cursor)** – the input delayed by two unit intervals (UIs).

These components are combined to shape the output waveform. Conceptually, this accounts for the fact that each bit can be influenced by both the previous and following bits. Adjusting the **tap coefficients** (C−1, C0, and C+1) allows the transmitter to optimize the signal shape to reduce inter-symbol interference.

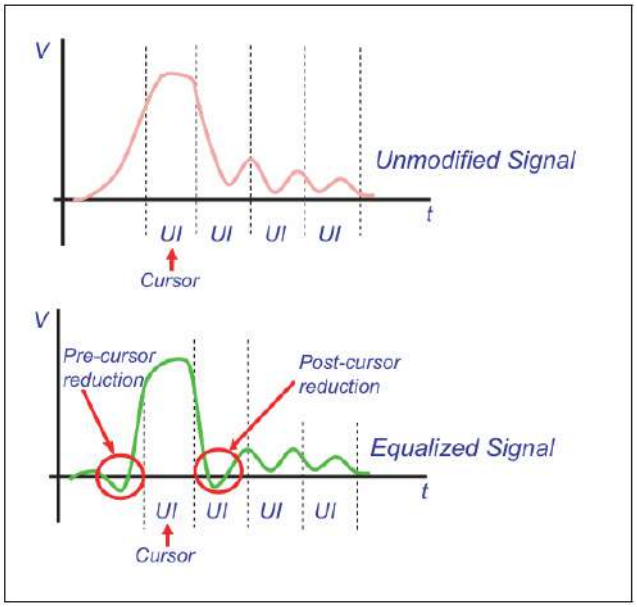


Figure 19: Tx 3-Tap Equalizer Shaping of an Output Pulse

Only the values for C−1 and C+1 are explicitly defined in the PCIe specification. The C0 value is always positive and is calculated such that the sum of the absolute values of all three coefficients equals 1. This normalization ensures a balanced signal output. This equalization technique is essential for maintaining signal integrity at high data rates, allowing the receiver to interpret the transmitted data accurately.

3.15.5 Ordered Sets

Ordered Sets are a form of low-level communication used exclusively within the Physical Layer of PCIe.

Ordered Sets Characteristics:

- Ordered Sets are not standard packets; they do not contain Start and End framing characters.
- They originate from the Transmitter's Physical Layer and terminate at the Receiver's Physical Layer.
- They are not routed across the PCIe fabric and only operate between directly connected devices.

Ordered Sets Structure:

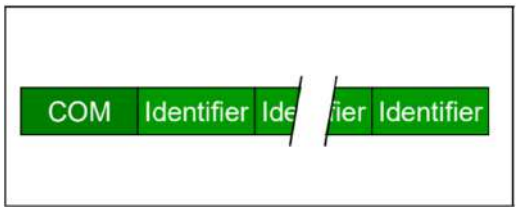


Figure 20: Ordered Set Structure

- Gen1 and Gen2:
  - Begin with a COM character, followed by at least three additional characters.
  - Always a multiple of 4 bytes in size.
  - Used for reliable alignment and recognition at the receiver.

Table 2: PCIe Ordered Sets and Their First Symbols

Ordered Set	First Symbol
EIEOS	00h
EIOS	66h
FTS	55h
SDS	E1

Ordered Set	First Symbol
TSI	1Eh
TS2	2Dh
SKP	AAh

- Gen3 and later:
  - The structure of Ordered Sets is different due to the use of 128b/130b encoding.
  - Format details are covered in the PCIe specification under Link Initialization and Training.

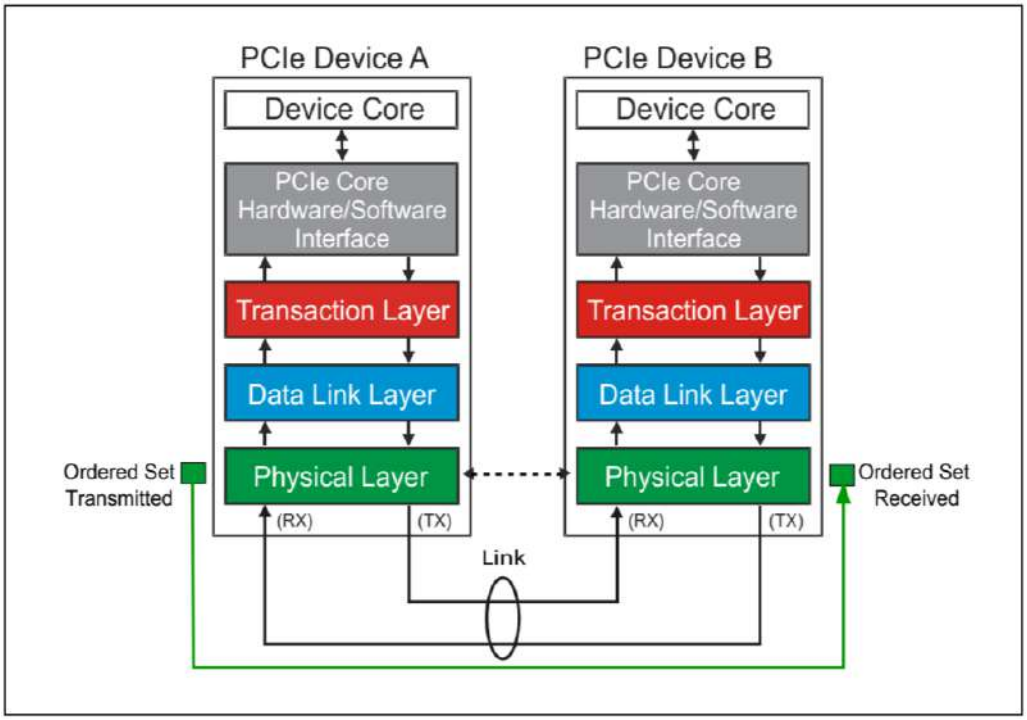


Figure 21: Ordered Sets Origin and Destination

Functions of Ordered Sets:

- Link Training and Initialization used to negotiate:
  - Link width (e.g., x1, x4, x8, x16)
  - Link speed (e.g., Gen1–Gen5)
  - Lane polarity and reversal
  - Bit lock and symbol lock for proper synchronization
- Clock Tolerance Compensation:
  - Used to handle minor differences in clock frequencies between transmitter and receiver.
  - Ensures reliable and continuous data exchange despite clock drift.
- Power Management:
  - Indicate entry into or exit from low-power states (such as L0s or L1).
  - Help coordinate power transitions without data corruption.

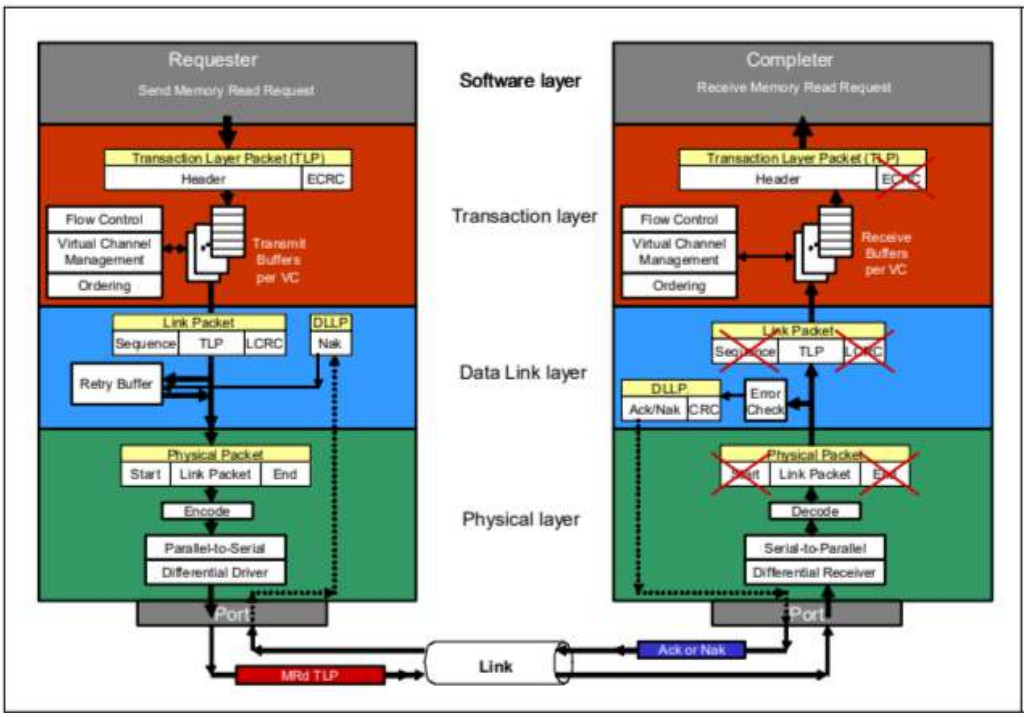


Figure 22: Memory Read Request Phase

Ordered Set Example - SOS: This example explains how a Skip Ordered Set (SOS) is transmitted and recognized in PCIe Gen3. An Ordered Set is marked by a specific Sync Header (01b) and is usually 16 bytes long, except for SOS, which can vary between 8 and 24 bytes to help with clock compensation. A Data Block that ends with an EDS token signals that an Ordered Set, like SOS, is coming next. After SOS, another Data Block must follow. All Lanes must receive the same Ordered Set at the same time. The receiver identifies the SOS using the Sync Header and SKP patterns, and the last part of SOS always includes the current 24-bit scrambler state, which helps maintain the data stream's synchronization.

3.15.6 PCIe Memory Read Request Flow:

- Requester Sends Request:
  - The Requester's Software or Core wants to read memory, so it sends details like the memory address, data size, and Requester ID to the Transaction Layer.
  - The Transaction Layer creates a Memory Read TLP (Transaction Layer Packet), then sends it to a buffer to wait for its turn.
- Flow Control Check:
  - Before sending, PCIe checks if the receiver has enough space (buffer space available). If yes, it sends the TLP to the Data Link Layer.
- Data Link Layer Adds Info:
  - Adds a Sequence Number and LCRC (error-checking code).
  - Stores a copy of the packet in the Replay Buffer in case it needs to be resent.
- Physical Layer Transmission:
  - Adds Start/End markers, splits data across lanes (byte striping), scrambles, and 8b/10b encodes the data.
  - Data is serialized and sent across the PCIe link.
- Completer Receives Data:
  - Reverses the transmission process: de-serializes, decodes, un-stripes, and descrambles the data.
  - Passes the packet to the Data Link Layer.

- Completer Acknowledges:
  - Checks for errors using the LCRC and validates the Sequence Number.
  - If valid, it sends an Ack (DLLP) with a 16-bit CRC back to the Requester.
  - If there's an error, it sends a Nak, and the Requester will resend the packet.
- Requester Handles Ack:
  - When it receives the Ack and checks it's valid, it removes the corresponding packet from the Replay Buffer.
- Completer Prepares to Process:
  - Forwards the TLP to its Transaction Layer.
  - Optional ECRC check is done.
  - If everything is fine, it sends the request details (address, size, etc.) to the Completer's Software to prepare the response.

The Completer sends the requested data back to the Requester in a "Completion with Data" (CplD) packet. This packet goes through all PCIe layers (Transaction, Data Link, Physical) just like the original request. The Requester checks the packet for errors, sends an acknowledgment (Ack) if it's valid, and then passes the data to its software layer. This completes the memory read process.

3.16 Gen3 Physical Layer Receive Logic

The Receiver logic for 8.0 GT/s begins with the Clock and Data Recovery (CDR) circuit, which likely includes a PLL to generate a recovered clock (Rx Clock) by locking onto the Transmitter's clock. This clock latches incoming bits into a deserializer. Once block alignment is achieved during the LTSSM's Recovery state, a divided version of the clock (Rx Clock/8.125) latches 8-bit symbols into the Elastic Buffer.

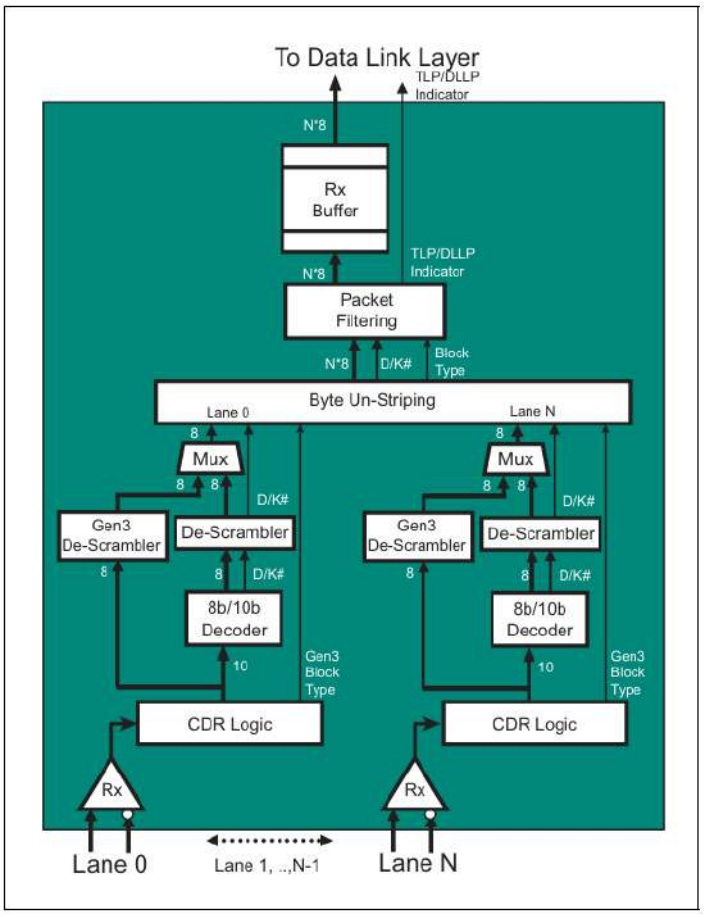


Figure 23: Gen3 Physical Layer Receiver Details

The data is then de-scrambled by Gen3 De-Scrambler and bypassing 8b/10b decoding to the Byte Un-striping logic. Ordered Sets are removed, and the remaining data (TLPs and DLLPs) is sent to the Data Link Layer. The description focuses only on changes for Gen 3 (8.0 GT/s), omitting unchanged components from Gen1/Gen2.

3.16.1 Differential Receiver

The differential receiver logic remains the same as in previous generations. However, there are electrical updates to enhance signal integrity, including improvements in signal compensation and new training procedures for signal equalization.

3.16.2 CDR (Clock and Data Recovery) Logic

- **Rx Clock Recovery**
  - The new scrambling scheme aids clock recovery but doesn't guarantee frequent transitions, so CDR must maintain sync with fewer edges.
  - More robust PLL or DLL circuits may be required for reliable synchronization.
  - The internal clock for the Elastic Buffer is not a simple Rx clock ÷ 8, due to the data format (2-bit Sync Header + 16 bytes).
  - A practical solution involves dividing the clock by 8.125, accounting for the 130-bit block (2 header bits + 128 data bits).
  - Block Type Detection logic can help handle the extra bits, ensuring the Elastic Buffer only receives bytes.
  - For 8.0 GT/s, the internal clock becomes 0.985 GHz (8.0 GHz $\div\, 8.125$), which is approximately 1.5% less than 1.0 GHz—a difference that is generally negligible in practice.
- **Deserializer**
  - Each Lane uses the recovered Rx clock to clock in incoming serial data.
  - The Deserializer converts this data to 8-bit symbols.
  - These symbols are sent to the Elastic Buffer.
  - The buffer uses an Rx clock divided by 8.125 to align with the 128-bit/130-bit structure.
- **Achieving Block Alignment**
  - EIEOS help identify 130-bit block boundaries, appearing as an alternating 00h/FFh byte pattern as shown in Figure 24,Block alignment occurs in three phases:
- Unaligned Phase:
  - Entered after Electrical Idle (e.g., after switching to 8.0 GT/s).
  - The receiver searches for EIEOS to detect the block boundary.
  - It may adjust alignment using SOS until EIEOS is detected.
- Aligned Phase:
  - Receiver has partially found block boundaries.
  - It continues to monitor for EIEOS for fine-tuning.
  - Now also looks for SDS to transition to Locked.
  - If an invalid Sync Header (00b or 11b) is seen, it may revert to Unaligned.
- Locked Phase:
  - Block alignment is fixed—no more adjustments.
  - Receiver expects proper Data Blocks after SDS.
  - If misalignment occurs, some data might be lost.
  - Receiver may return to earlier phases if needed, as long as Data Stream processing is paused.

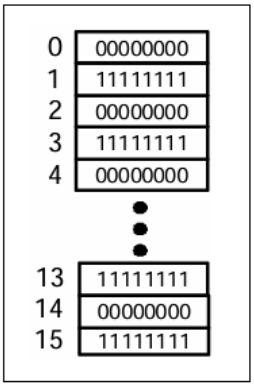


Figure 24: EIEOS Symbol Pattern

- **Block Type Detection**
  - After Block Alignment, the receiver inspects the first 2 bits (Sync Header) of each block to determine its type:
    - Ordered Set Blocks are used only by the Physical Layer and are not forwarded to higher layers.
    - Data Blocks are forwarded to upper layers.
  - The Sync Header helps decide if a block should be passed on or discarded, and aids in converting 130-bit blocks to 128-bit payloads.
  - Since block type info is identical across all lanes, detection logic can be implemented on just one lane (typically Lane 0), However, if Lane Reversal or variable Link widths are supported, the logic may be included in multiple lanes, but only one active lane uses it.

This phased process ensures reliable data alignment and stream integrity.

3.16.3 Receiver Clock Compensation Logic

- At 8.0 GT/s, transmitter and receiver clocks can differ by up to ±300 ppm, potentially causing drift of 1 clock every 1666 cycles.
- The Elastic Buffer in Figure 25 handles this drift by adding or removing SKP symbols, but now does so in groups of 4 instead of one at a time.
- SKP Ordered Sets (SOS) are sent periodically (every 370–375 blocks) to trigger compensation.
- Compensation occurs:
  - Add SKPs if the receiver's local clock is faster (prevent underflow).
  - Remove SKPs if the recovered clock is faster (prevent overflow).
- SOSs must be separated by a Data Block at 8.0 GT/s, unlike previous versions.
- The buffer must be sized to handle worst-case timing differences, especially when large packets like 4KB TLPs are in flight.

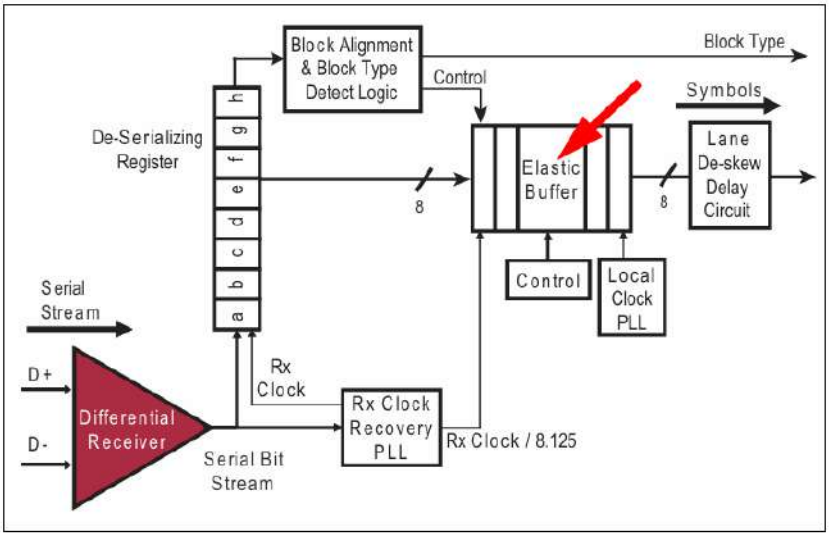


Figure 25: Gen3 Elastic Buffer Logic

Lane-to-Lane Skew

- In multi-lane PCIe links, arrival time differences between lanes (skew) are corrected at the receiver by delaying early lanes so all symbols align.
- This de-skewing is typically done after the Elastic Buffer using digital delays measured in Symbol clock cycles.
- For Gen3, the receiver must handle up to 6ns of skew.
- De-skewing is only performed during certain LTSSM states (e.g., Recovery, Configuration, L0s), and not in L0 during normal data transmission.
- Ordered Sets (like SOS, SDS, EIEOS) are used to align symbols across lanes.
- Tools can use SOS patterns to assist with de-skewing even during active data streams.
- Transmitters must minimize added skew, leaving room for physical variations (like PCB routing differences).
- While 128b/130b encoding lacks 8b/10b's COM symbol, synchronized Ordered Sets that arrive at the same time on all the Lanes still provide reliable de-skew opportunities.

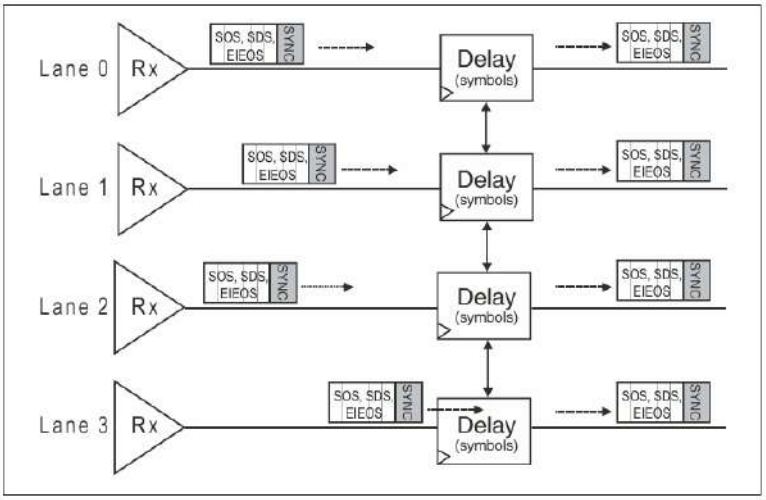


Figure 26: Receiver Link De-Skew Logic

3.16.4 Descrambler

- Receivers recover original data by applying the same scrambling polynomial as the transmitter (XOR operation).
- Implementation can use one or multiple LFSRs, similar to the transmitter.
- At Gen3 (8.0 GT/s) and higher rates, descrambling is always enabled to support clock recovery and signal integrity.
- Unlike Gen1/Gen2, the ”disable scrambling” control bit of TS1s and TS2s is not used at Gen3 speeds and higher.

3.16.5 Byte Un-Striping

The byte un-striping logic remains fundamentally unchanged from its implementation in Gen1 and Gen2. In Gen3, the data output from the Gen3 descrambler is multiplexed with the data from the Gen1/Gen2 descrambler just prior to entering the un-striping stage, as illustrated in Figure 23.

3.16.6 Packet Filtering

Packet filtering removes Logical Idle bytes and Ordered Sets from the serial byte stream, forwarding only TLPs and DLLPs—along with their packet type indicators—to the Data Link layer.

3.16.7 Receive Buffer (Rx Buffer)

The Rx Buffer temporarily stores received TLPs and DLLPs until the Data Link Layer can process them. Designers can define the interface details, such as bus width—wider buses reduce clock frequency but require more signals and logic.

4 Applications

PCIe Gen 5 (5.0) represents a significant leap in bandwidth over Gen 4, doubling the data rate per lane to **32 GT/s** and offering **~4 GB/s per lane (x1) per direction**. This massive bandwidth unlocks applications demanding extreme data throughput, primarily in high-performance computing, data centers, and future consumer/prosumer devices. Here are the key applications:

1. **Ultra-High-Performance Storage (NVMe SSDs):**
    - **Why:** Directly addresses the bottleneck for the fastest NVMe SSDs. Gen 5 SSDs leverage x4 lanes to achieve sequential read/write speeds significantly exceeding 10 GB/s (theoretical max ~16 GB/s for x4), approaching 14 GB/s or higher with compression.
    - **Applications:** Real-time data analytics, massive database workloads, high-resolution video editing/rendering (8K+), scientific simulations, game loading (future titles/engines), enterprise storage arrays, tier-0 caching.

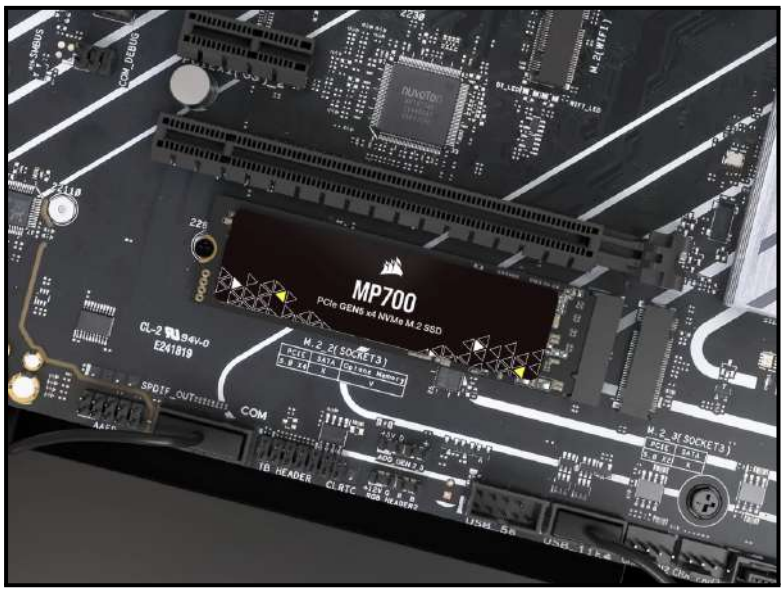


Figure 27: MP700 NVMe™ PCIe® M.2 SSD

    - **Challenge:** Significant heat generation requires advanced cooling solutions (large heatsinks, active fans) on both the SSD and motherboard.
2. **Next-Generation Graphics Processing Units (GPUs):**
    - **Why:** While current high-end GPUs (e.g., RTX 4090) don't fully saturate PCIe Gen 4 x16, future GPUs targeting 8K+ gaming with high refresh rates, advanced ray tracing, complex AI upscaling (DLSS/FSR), and professional visualization/rendering will increasingly demand more bandwidth.
    - **AI/ML:** Training massive AI models benefits immensely from faster data transfer between CPU memory and GPU memory (reducing data loading bottlenecks).

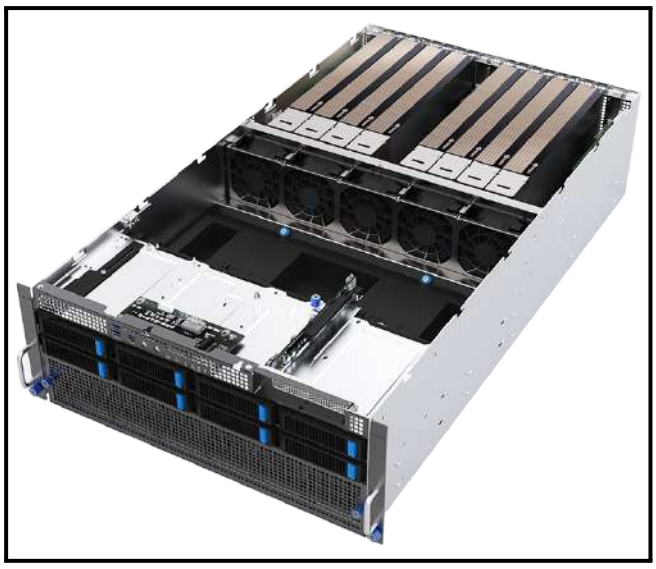


Figure 28: ESC8000-E11P is a dual-socket server powered by 5th Gen Intel® Xeon®

- **Multi-GPU:** High-bandwidth interconnects (like NVLink) may leverage PCIe Gen 5 internally or for CPU connection.

3. **Data Centers & High-Performance Computing (HPC):**
   - **CPU Interconnects:** Faster links between CPUs in multi-socket servers (using PCIe for coherence protocols like UPI/Infinity Fabric).
   - **Accelerators:** Connecting GPUs, FPGAs, ASICs (for AI/ML, cryptography, genomics, finance) with minimal latency and maximum throughput.
   - **Networking:** Enabling next-generation network interface cards (NICs) for 800 Gigabit Ethernet (800GbE) and beyond, requiring the bandwidth of PCIe Gen 5 x16 lanes.
   - **Storage Controllers/Adapters:** Connecting vast arrays of NVMe SSDs (JBOFs - Just a Bunch of Flash) or high-speed network storage (NVMe-oF) to servers.
   - **CXL (Compute Express Link):** CXL 1.1/2.0 builds *on top of* PCIe Gen 5 physical layer. Enables game-changing memory pooling, sharing, and expansion between CPUs, GPUs, accelerators, and smart NICs, drastically improving resource utilization and performance in heterogeneous compute environments.

Figure 29: Composable CXL Memory to meet the highest performance requirements

4. **High-Speed Networking:**
   - **Why:** 400GbE and especially 800GbE network adapters require immense bandwidth to the CPU/chipset.

- **Application:** Hyperscale data centers, cloud providers, high-frequency trading, backbone internet infrastructure. PCIe Gen 5 x16 provides the necessary throughput for these ultra-high-speed NICs without bottlenecking.

5. **Specialized Accelerators:**
   - **Why:** Cards dedicated to specific compute tasks (AI inference, video transcoding, financial modeling, scientific calculations, in-memory databases) often need the fastest possible connection to the CPU and system memory.
   - **Application:** Real-time AI inference at the edge or in data centers, live 4K/8K video production, complex simulations.
6. **High-End Workstations:**
   - **Why:** Combining multiple high-bandwidth devices (multiple Gen 5 SSDs, a high-end GPU, potentially an accelerator card, 100GbE+ networking) requires the aggregate bandwidth headroom provided by PCIe Gen 5 lanes from modern workstation CPUs/chipsets.
   - **Application:** Professional content creation (VFX, animation, CAD/CAM), engineering simulation, scientific research, complex data analysis.

5 Link Initialization and Training

Link Initialization and Training in PCI Express is a hardware-controlled process managed by the Physical Layer through the LTSSM (Link Training and Status State Machine). After a reset, the link is automatically configured for normal operation. Key steps include: **Bit Lock**, where the receiver uses Clock and Data Recovery (CDR) to synchronize with the transmitter's clock; **Symbol Lock**, which uses 8b/10b encoding to determine symbol boundaries via the COM symbol (Gen1/Gen2); and **Block Lock** (Gen3), using EIEOS patterns to find packet boundaries. Other aspects include determining the maximum common **Link Width** and optional **Lane Reversal**, which allows flexible physical routing if supported by one of the devices.

5.1 Ordered Sets in Link Training

During the PCI Express Link Training process, the Physical Layer utilizes special **Ordered Sets**, particularly the **Training Sequences** TS1 and TS2. These sequences are essential for establishing and configuring communication between devices. Their structure varies depending on the link's operating generation (Gen1/Gen2 or Gen3), and each format contains specific fields tailored to the training needs of that generation. The sequences convey important configuration and synchronization information necessary for successful link initialization.

5.2 TS1 and TS2 Ordered Sets

Each Training Sequence (TS1 or TS2) in PCI Express comprises a series of 16 symbols, each conveying specific control and configuration information critical for successful link initialization and training. The initial symbols help establish synchronization (e.g., Symbol Lock and Lane de-skew), while others communicate details such as link number, lane number, number of fast training sequences (N_FTS), supported data rates, and equalization parameters. Additional fields manage link behavior, including bandwidth scaling, loopback, scrambling, and DC balance (especially important in Gen3, where 8b/10b encoding is not used). These sequences adapt in format and function depending on the link speed generation, ensuring compatibility and robust initialization across diverse configurations.

5.2.1 Summary of TS1 Ordered Set Contents

Table 3: TS1 Ordered Set Symbol Descriptions

Symbol	Description
0	Gen1/Gen2: COM (K28.5) Symbol Gen3: 1Eh indicates TS1
1	Link Number: • Non-Gen3 Ports: 0–255, PAD • Downstream Gen3: 0–31, PAD • Upstream Gen3: 0–255, PAD
2	Lane Number: 0–31, PAD
3	N_FTS: Number of FTS Ordered Sets required to exit L0s (0–255)
4	Data Rate Identifier: • Bit 0 — Reserved • Bit 1 — 2.5 GT/s supported (must be 1) • Bit 2 — 5.0 GT/s supported (must be 1 if Bit 3 is set) • Bit 3 — 8.0 GT/s supported • Bits 5:4 — Reserved • Bit 6 — Autonomous Change / Selectable De-emphasis (usage depends on port role and LTSSM state) • Bit 7 — Speed Change (only in Recovery.RcvrLock)
5	Training Control: • Bit 0 — Hot Reset • Bit 1 — Disable Link • Bit 2 — Loopback • Bit 3 — Disable Scrambling (2.5/5.0 GT/s only) • Bit 4 — Compliance Receive

	• Bits 7:5 — Reserved (Set to 0)
6	Gen1/Gen2: TS1 Identifier (D10.2 = 4Ah) or EQ TS1: • Bits 2:0 — Receiver Preset Hint • Bits 6:3 — Transmitter Preset • Bit 7 — Set to 1 Gen3: • Bits 1:0 — Equalization Control • Bit 2 — Reset EIEOS Interval Count • Bits 6:3 — Transmitter Preset • Bit 7 — Use Preset
7	Gen1/Gen2: TS1 Identifier (D10.2 = 4Ah) Gen3: • Bits 5:0 — FS (Full Swing) or Pre-cursor Coefficient • Bits 7:6 — Reserved
8	Gen1/Gen2: TS1 Identifier (D10.2 = 4Ah) Gen3: • Bits 5:0 — LF (Low Frequency) or Cursor Coefficient • Bits 7:6 — Reserved
9	Gen1/Gen2: TS1 Identifier (D10.2 = 4Ah) Gen3: • Bits 5:0 — Post-cursor Coefficient • Bit 6 — Reject Coefficient Values • Bit 7 — Parity (even parity of Symbols 6–8 and bits 6:0 of Symbol 9)
10–13	Gen1/Gen2: TS1 Identifier (D10.2 = 4Ah) Gen3: TS1 Identifier (4Ah)
14–15	Gen1/Gen2: TS1 Identifier (D10.2 = 4Ah) Gen3: TS1 Identifier (4Ah) or DC-Balance Symbol

5.2.2 Summary of TS2 Ordered Set Contents

Table 4: TS2 Ordered Set Symbol Descriptions

Symbol	Description
0	Gen1/Gen2: COM (K28.5) Symbol Gen3: 2Dh indicates TS2
1	Link Number: • Non-Gen3 Ports: 0–255, PAD • Downstream Gen3: 0–31, PAD • Upstream Gen3: 0–255, PAD
2	Lane Number: 0–31, PAD
3	N_FTS: Number of FTS Ordered Sets required to exit L0s (0–255)
4	Data Rate Identifier: • Bit 0 — Reserved • Bit 1 — 2.5 GT/s supported (must be 1) • Bit 2 — 5.0 GT/s supported (must be 1 if Bit 3 is set) • Bit 3 — 8.0 GT/s supported • Bits 5:4 — Reserved • Bit 6 — Autonomous Change / Selectable De-emphasis / Link Upconfigure Capability Used in Polling.Configuration, Configuration.Complete, and Recovery states; reserved otherwise • Bit 7 — Speed Change (only in Recovery.RcvrLock)
5	Training Control: • Bit 0 — Hot Reset • Bit 1 — Disable Link • Bit 2 — Loopback • Bit 3 — Disable Scrambling (2.5/5.0 GT/s only)

	• Bits 7:4 — Reserved (Set to 0)
6	Gen1/Gen2: • TS2 Identifier (45h) encoded as D5.2 or EQ TS2: – Bits 2:0 — Receiver Preset Hint – Bits 6:3 — Transmitter Preset – Bit 7 — Equalization Command Gen3: • Bits 5:0 — Reserved • Bit 6 — Quiesce Guarantee (only in Recovery.RcvrCfg) • Bit 7 — Request Equalization (only in Recovery.RcvrCfg)
7–13	Gen1/Gen2: TS2 Identifier (45h) encoded as D5.2 Gen3: TS2 Identifier (45h)
14–15	Gen1/Gen2: TS2 Identifier (45h) encoded as D5.2 Gen3: TS2 Identifier (45h) or DC-Balance Symbol

5.3 States Overview

Every device must perform initial link training at the base rate of 2.5 GT/s. Figure 30 highlights the states involved in the initial training sequence.

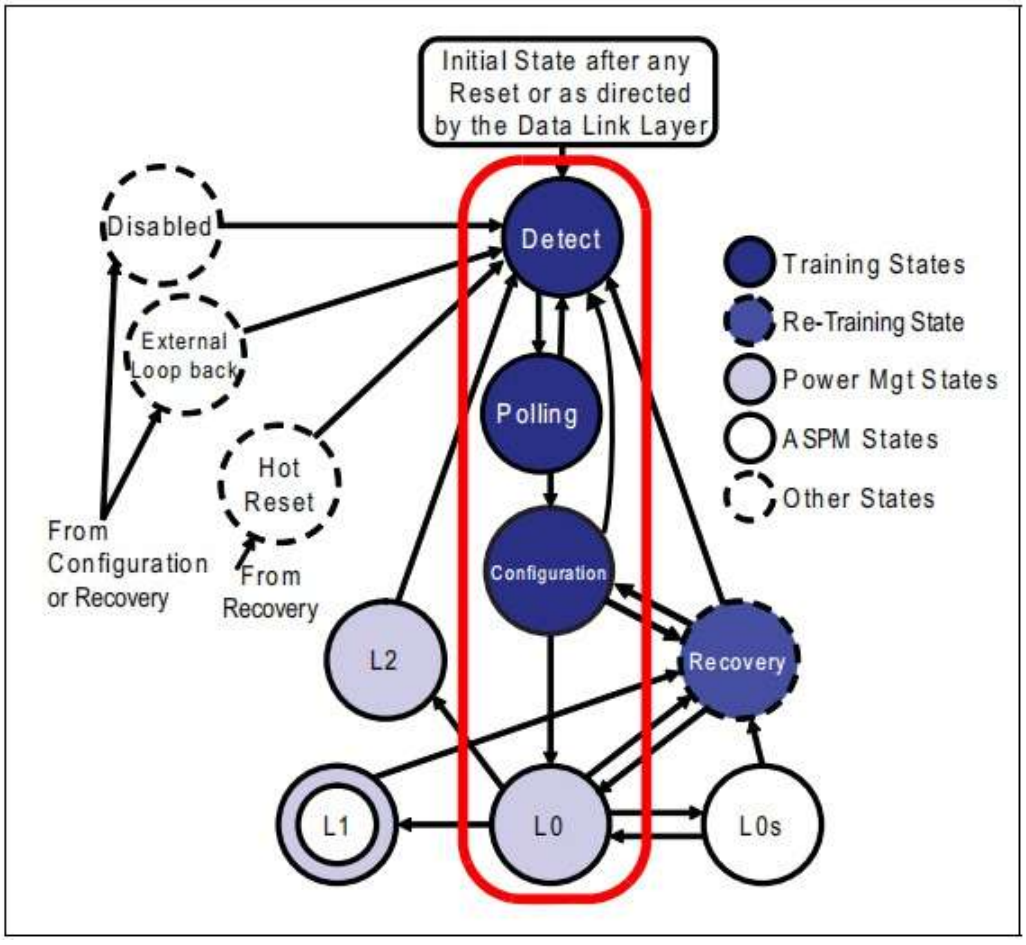


Figure 30: States Involved in Initial Link Training

5.4 Detect States

Each transmitter uses the Detect state to detect a receiver on the opposite link end. Detect state contains two substates: **Detect quiet** and **Detect Active**.

5.4.1 Detect Quiet

This substate is the initial state after reset/power-up (within 20ms) and is also entered upon inability to move forward from other states. It transitions to **Detect.Active** after a 12 ms timeout or when any Lane exits Electrical Idle.

5.4.2 Detect Active

This substate, entered from **Detect.Quiet**, involves the Transmitter testing for a connected Receiver on each Lane. The next substates are:
• **Detect Quiet**: if no Lanes detect a Receiver in 12 ms, go back to Detect.Quiet.
• **Polling State**: If a receiver is detected on all Lanes.

5.5 Polling States

During the Polling state, connected devices transition from electrical idle to exchange TS1s and TS2s to achieve bit and symbol lock, enabling successful reception of each other's ordered-sets.

5.5.1 Polling Active

In **Polling.Active**, transmitters send at least 1024 consecutive TS1s on all detected Lanes. Receivers use the incoming TS1s to acquire **Bit Lock** then either **symbol lock** for Gen1 and Gen2 or **Block lock** for Gen3 and higher. The next substates are:
• **Detect State**: Timeout after 24 ms of no response.
• **Polling Configuration**: if ALL detected Lanes receive 8 consecutive training sequences with Link and Lane set to PAD and one of the following is true:
- TS1s with Link and Lane set to PAD were received with the Compliance Receive (bit 4 of Symbol 5) cleared to 0b.
- TS1s with Link and Lane set to PAD were received with the Loopback (bit 2 of Symbol 5) set to 1b.
- TS2s were received with Link and Lane set to PAD.

5.5.2 Polling Configuration

In this substate, a transmitter switches from sending TS1s to TS2s (with link/lane numbers in PAD) in order to signal readiness to its partner for the next state transition, creating a handshake where both devices must be sending and receiving TS2s before proceeding. The next substates are:
• **Detect State**: Timeout after 48 ms of no response.
• **Configuration State**: After eight consecutive TS2s with Link and Lane set to PAD are received on any detected Lanes, and at least 16 TS2s have been sent since receiving one TS2.

5.6 Configuration State

This state aims to determine Port connectivity and assign Lane/Link numbers (e.g., 8 Lanes with 2 active, or split into two x4 Links). Port roles differ by direction: Downstream Ports act as "leaders" driving Link initialization, while Upstream Ports act as "followers." The Downstream Port assigns Link and Lane numbers through TS1 training sequences, which initially contain PAD placeholders. Upstream Ports echo these values unless conflicts arise. This role-specific interaction governs the remaining initialization process, with number assignments reflected in TS1 fields during configuration.

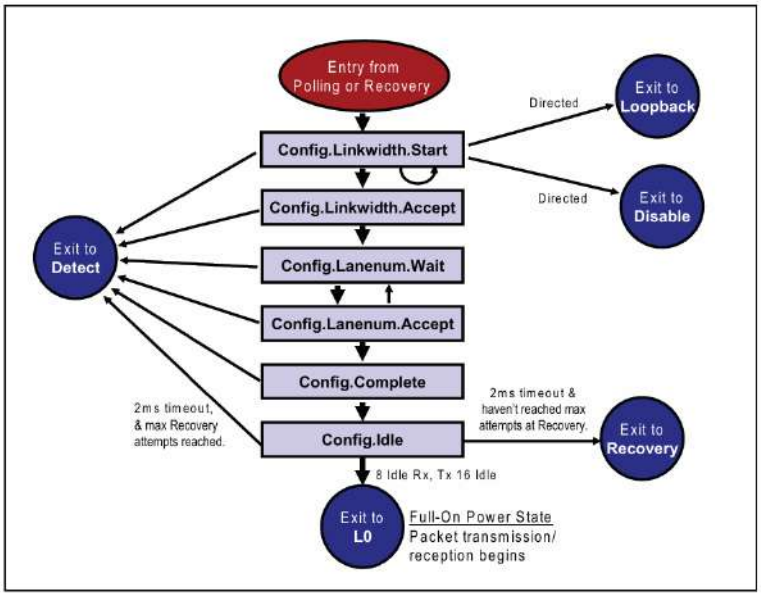


Figure 31: Configuration State Machine

5.6.1 Configuration.Linkwidth.Start

In the **Configuration.Linkwidth.Start** substate, the Downstream Port begins Link width negotiation by sending TS1s with non-PAD Link numbers and PAD Lane numbers, while the Upstream Port sends TS1s with PAD values until it receives valid TS1s. Transitions from this state depend on conditions such as received TS1s, crosslink behavior, timeouts, or higher-layer commands, potentially leading to substates like **Accept**, **Disable**, **Loopback**, or **Detect**.

5.6.2 Configuration.Linkwidth.Accept

In the **Configuration.Linkwidth.Accept** substate, the Upstream Port echoes the received Link number in its TS1s on all active Lanes, while the Downstream Port begins assigning non-PAD Lane numbers using the same Link number. Once the Downstream Port confirms consistent TS1s indicating Link width, it transitions to **Configuration.Lanenum.Wait**. The Upstream Port follows by responding to the proposed Lane numbers, potentially adjusting for configurations like Lane reversal.

5.6.3 Configuration.Lanenum.Wait

In the **Configuration.Lanenum.Wait** substate, both Upstream and Downstream Ports continue sending TS1s with consistent, non-PAD Link and Lane numbers while awaiting confirmation of lane alignment. Lane numbers must be sequential and contiguous starting from 0. If a proper Link can't be established—e.g., due to non-contiguous Lanes or all Lanes receiving PAD values—then the Ports exit to the **Detect** state after a 2ms timeout. Otherwise, if two consecutive TS1s (or TS2s for the Upstream) with matching or updated Lane numbers are received, the Ports transition to **Configuration.Lanenum.Accept**. Multi-Lane Ports are advised to wait before finalizing Link width to tolerate temporary errors or alignment issues.

5.6.4 Configuration.Lanenum.Accept

In the **Configuration.Lanenum.Accept** substate, both the Downstream and Upstream Ports assess whether a valid Link can be established using the received Lane numbers. If two consecutive TS1s (Downstream) or TS2s (Upstream) with matching non-PAD Link and Lane numbers are received and align with what is being transmitted, the Ports transition to **Configuration.Complete**. Lane Reversal is permitted if supported and strictly follows reversed numbering from $n-1$ to 0. If only a subset of Lanes can form a valid Link, the Ports reassign Lane numbers starting from zero and return to **Configuration.Lanenum.Wait**. If no viable Link can be established or all Lanes receive PAD values, they transition to the **Detect** state. Bandwidth management status bits are also updated depending on whether the configuration was initiated for reliability or power optimization.

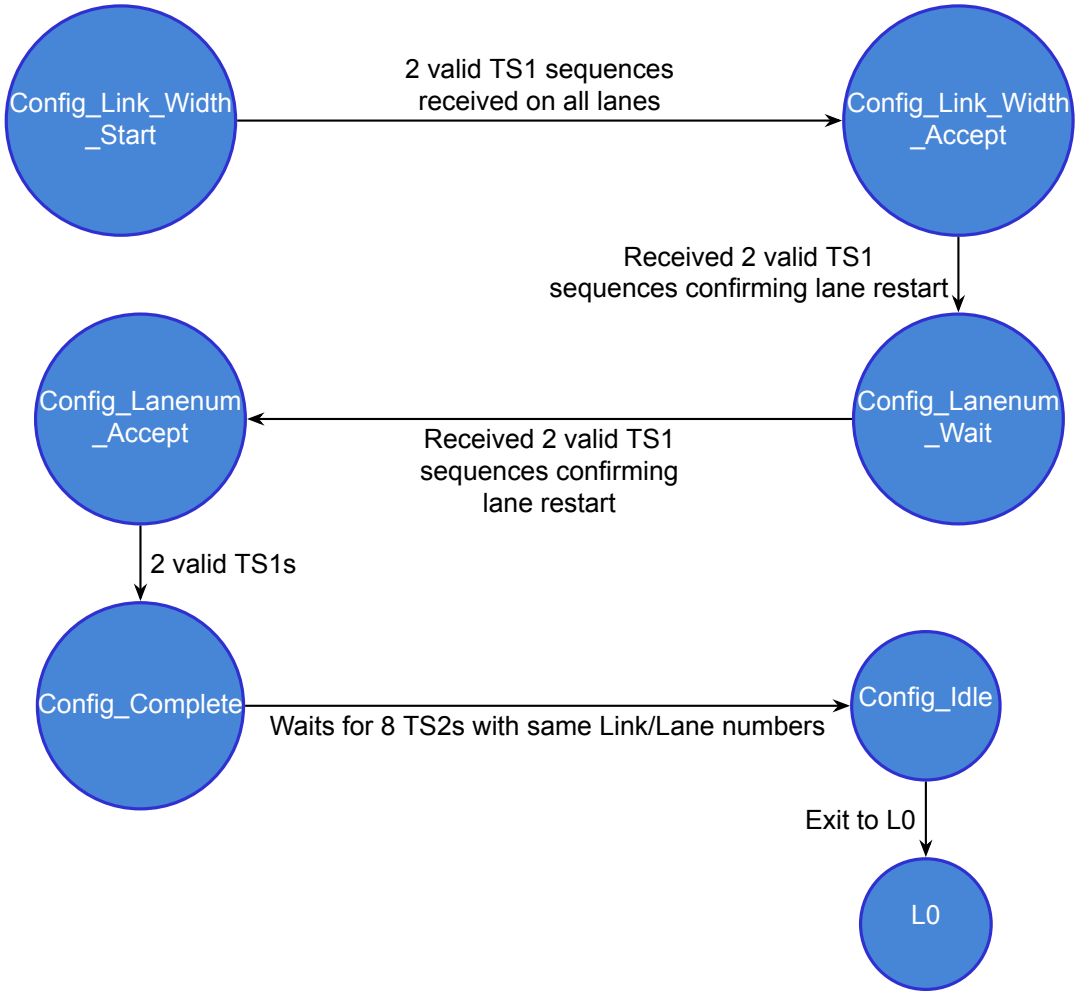
5.6.5 Configuration.Complete

The **Configuration.Complete** substate finalizes link readiness via TS2 Ordered Set exchanges, enabling devices to adjust data rates and upconfigure capabilities *only upon entry* based on TS2-advertised parameters. Downstream/Upstream Lanes synchronize Link/Lane numbers, disable scrambling (in 8b/10b mode if all lanes agree), and complete de-skewing. Transition to **Configuration.Idle** requires receiving 8 aligned TS2s and sending ≥16 TS2s; failures or a 2 ms timeout revert to **Detect**, with non-Link lanes reassigned or idled. Variables like **changed_speed_recovery** reflect negotiated settings, ensuring synchronized configuration.

5.6.6 Configuration.Idle

The **Configuration.Idle** substate transitions the link to operational status (Linkup = 1b) by transmitting Idle data (scrambled zeros for 8b/10b; SDS Ordered Sets followed by Idle Symbols for 128b/130b) while awaiting 8 consecutive Idles on all lanes to enter **L0**. Transition requires sending 16 Idles after initial detection, with 128b/130b additionally requiring no prior timeout from **Configuration.Complete**. Lane de-skew must complete before Data Stream processing. A 2 ms timeout triggers recovery (via **idle_to_rlock_transitioned**, capped at 256 attempts) or reverts to **Detect** if retries exhaust. Software-initiated retraining updates the Link Bandwidth Management bit, and `idle_to_rlock_transitioned` resets to **00h** on **L0** entry.

5.6.7 Downstream TX Configuration State Flow



These states control the downstream transmitter in PCIe PIPE, ensuring valid TS1/TS2 detection, lane alignment, and compliance with ordered set formatting across link training phases.after this the PCIe reach linkup and go to Recovery to change the speed to PCIe gen5.

5.6.8 L0 State

This is the normal, fully-operational Link state, during which Logical Idle, TLPs, and DLLPs are exchanged between Link neighbors. L0 is achieved immediately following the conclusion of the Link Training process. The Physical Layer notifies the upper layers that the Link is ready for operation by setting the `LinkUp` variable.

5.6.9 Two Conditions Causing Automatic Speed Change

- When rates higher than 2.5 GT/s are supported by both partners and the Link is active, a speed change is set in its TS Ordered Sets.
- When both partners support 8.0 GT/s or higher and one wants to perform Tx Equalization.

5.6.10 Link Partner Initiated Transitions

1. If Electrical Idle is detected on all lanes without first receiving an EIOS, the Port may stay in L0 or go to Recovery. If errors occur, it must go to Recovery (e.g., by setting the Retrain Link bit).
2. If TS1/TS2 (or EIEOS for 128b/130b) are received on any configured lanes, it means the link partner has already entered Recovery. The transmitter is allowed to finish sending the current TLP/DLLP, then enter Recovery.
3. If an EIOS is received, and the Receiver doesn't support L0s and hasn't been directed to L1 or L2, then it must go to Recovery.

5.7 Recovery States

5.7.1 Summary: Reasons for Entering Recovery State

A PCIe link may enter the Recovery state due to the following summarized reasons:

- Transitioning from L1 state, as no fast training sequence (FTS) is sent during L1 exit.
- Loss of receiver lock on exiting L0 due to missed FTS timing.
- Availability of a higher data rate after initial training in L0.
- Requested link speed or width change for performance or reliability.
- Software-triggered retraining via the Retrain Link bit.
- Errors like Replay Number Roll-over prompting automatic retraining.
- Detection of TS1 or TS2 ordered sets on active lanes.
- Detection of Electrical Idle without receiving the corresponding Ordered Set.

5.7.2 Recovery Lock

The target of the Recovery Lock state is to ensure the link has successfully locked onto the training sequences after equalization and speed changes, confirming stable communication parameters before moving forward in the link training process.

Either port can initiate Recovery by sending TS1 to its neighbor. When a port detects incoming TS1s, it knows the other port has entered Recovery, so it also enters Recovery and returns.

A receiver must acquire **Bit Lock** and either **Symbol Lock** (for 8b/10b encoding) or **Block Alignment** (for 128b/130b encoding) to be able to recognize symbols, ordered sets, and packets.

Exit Conditions:

- Exit to **Recovery.RcvrCfg**: The port must receive 8 matching TS1s or TS2s with correct speed and lane info, and EC = 00b at 8.0 GT/s. If the Extended Synch bit is set, at least 1024 TS1s must be sent first. If coming from Recovery.Equalization, the upstream port checks if received coefficients match the final ones.
- At 8.0 GT/s or higher, the link enters Recovery.Equalization only if needed and if `start_equalization_w_preset` is set.

- Otherwise, after a 24 ms timeout:
  - Exit to **Recovery.RcvrCfg** if 8 TS1s/TS2s match sent values and `speed_change` = 1b, and either speed is already > 2.5 GT/s or a higher rate is advertised.
  - Exit to **Recovery.Speed** if the higher speed fails (`changed_speed_recovery` = 0b) or a new speed doesn't work (`changed_speed_recovery` = 1b), fallback to previous or 2.5 GT/s speed.
  - Exit to **Configuration State** if no speed change is needed (`speed_change` = 0b and `directed_speed_change` = 0b), or 2.5 GT/s is the highest supported.
  - Exit to **Detect State** if none of the above conditions are met after the timeout.

5.7.3 Recovery.RcvrCfg

In the Recovery Lock state, bit and symbol lock are confirmed, and the port must determine if there are any other issues to address in the Recovery state.

- If Recovery was entered simply to establish bit and symbol lock after leaving a link power management state, it is likely that TS2s will be exchanged.
- If Recovery was triggered by speed or width changes, this substate decides the appropriate next transition.

During this substate, the transmitter sends TS2s on all configured lanes with the same link and lane numbers configured earlier.

Exit Conditions:

- **Exit to Recovery.Idle:** When eight consecutive TS2s are received with matching link, lane, and rate information, and the `speed_change` bit is 0b or no rate above 2.5 GT/s is supported. Additionally, sixteen TS2s must be sent without interruption by any Electrical Idle Ordered Set (EIEOS).
- **Exit to Recovery.Speed:** If eight consecutive TS2s with the speed change bit set and matching rate info are received. Both link partners must support rates higher than 2.5 GT/s, or the current rate is already above 2.5 GT/s.

  For 8b/10b encoding, at least 32 TS2s with speed change bit set must be sent without interruption after the first such TS2. For 128b/130b encoding, at least 128 TS2s withspeed change bit set are required after the first one.
- **Exit to Configuration:** If 8 consecutive TS1s with mismatched link or lane numbers are received, and the speed change bit is 0 or no rate above 2.5 GT/s is supported.
- **Exit to Detect State:** If no state transition occurs within 48 ms and the data rate is 2.5 or 5.0 GT/s.

5.7.4 Recovery.Equalization

At higher link speeds, signal distortion increases, so PCIe 3.0 introduces Transmitter Equalization to improve signal integrity. Unlike fixed de-emphasis at lower speeds, this method uses a handshake where each receiver analyzes the signal and suggests transmitter settings. This dynamic adjustment helps ensure proper communication without adding hardware complexity.

5.7.4.1 Phase 0

When transitioning to 8.0 GT/s, the Downstream Port (DSP) enters `Recovery.RcvrCfg` and sends Equalization TS2s containing Tx Presets and Rx Hints to the Upstream Port (USP).

- This step is skipped if the link is already at 8.0 GT/s.
- Presets are based on the DSP's Equalization Control register and can vary per lane.
- DSP applies its own values to its transmitter (and optionally receiver) and sends USP values for the upstream side.
- Once the rate changes, DSP enters Phase 1, sending TS1s with EC = 01b and waits for matching TS1s from USP.
- USP starts in Phase 0 by sending TS1s that echo the received presets.

5.7.4.2 Phase 1

In Equalization, the USP receives Equalization TS1s/TS2s, applies supported Tx presets, optionally uses Rx hints, and waits up to 500 ns to assess bit error rate (BER) $\leq 10^{-4}$.

- Upon detecting two valid TS1s, USP sets EC = 01b, enters Phase 1, and hands control to DSP.
- DSP echoes the process, confirming BER $\leq 10^{-4}$, and sends Tx coefficients: FS (Full Swing), LF (Low Frequency), and Post-cursor, which define signal amplitude and shape.
- FS is the maximum voltage level, LF defines the minimum voltage relative to FS, enabling coefficient scaling.
- When DSP receives valid TS1s with EC = 01b, it sets EC = 10b to begin Phase 2 and hands control back to USP.
- If DSP finds signal quality already sufficient, it can skip to EC = 00b to exit Equalization early.

Note: Phase 2 and Phase 3 are not supported in this design.

5.7.5 Recovery.Speed

Upon entering this substate, the transmitter must enter Electrical Idle and wait for the receiver to do the same. The transmitter must remain in Electrical Idle for at least 800 ns if the speed change succeeded, or at least 6 µs if it failed, but no longer than 1 ms.

- An Electrical Idle Ordered Set (EIOS) must be sent before entry if the current rate is 2.5 or 8.0 GT/s, and two EIOSs if the rate is 5.0 GT/s.
- Electrical Idle is recognized either by detecting EIOSs or by other defined detection methods.
- The operating frequency can only change after receiver lanes enter Electrical Idle.
- If the link is already at the highest supported rate, no frequency change occurs even if this substate runs.

Exit conditions:

- **Exit to Recovery.Lock** after timeout if this transition comes from Recovery.RcvrCfg and the speed change was successful.In this case, the data rate on all configured lanes is updated to the highest commonly supported rate.
- If this is the second time entering this substate since Recovery, the data rate reverts to what it was before entering Recovery.
- Otherwise, the data rate falls back to 2.5 GT/s. This fallback accounts for failure cases.
- **Exit to Detect state** if none of the conditions for exiting to Recovery.Lock are met.

5.7.6 Recovery.Idle

Transmitters usually send idle symbols in this substate to prepare for transitioning to the fully operational L0 state.

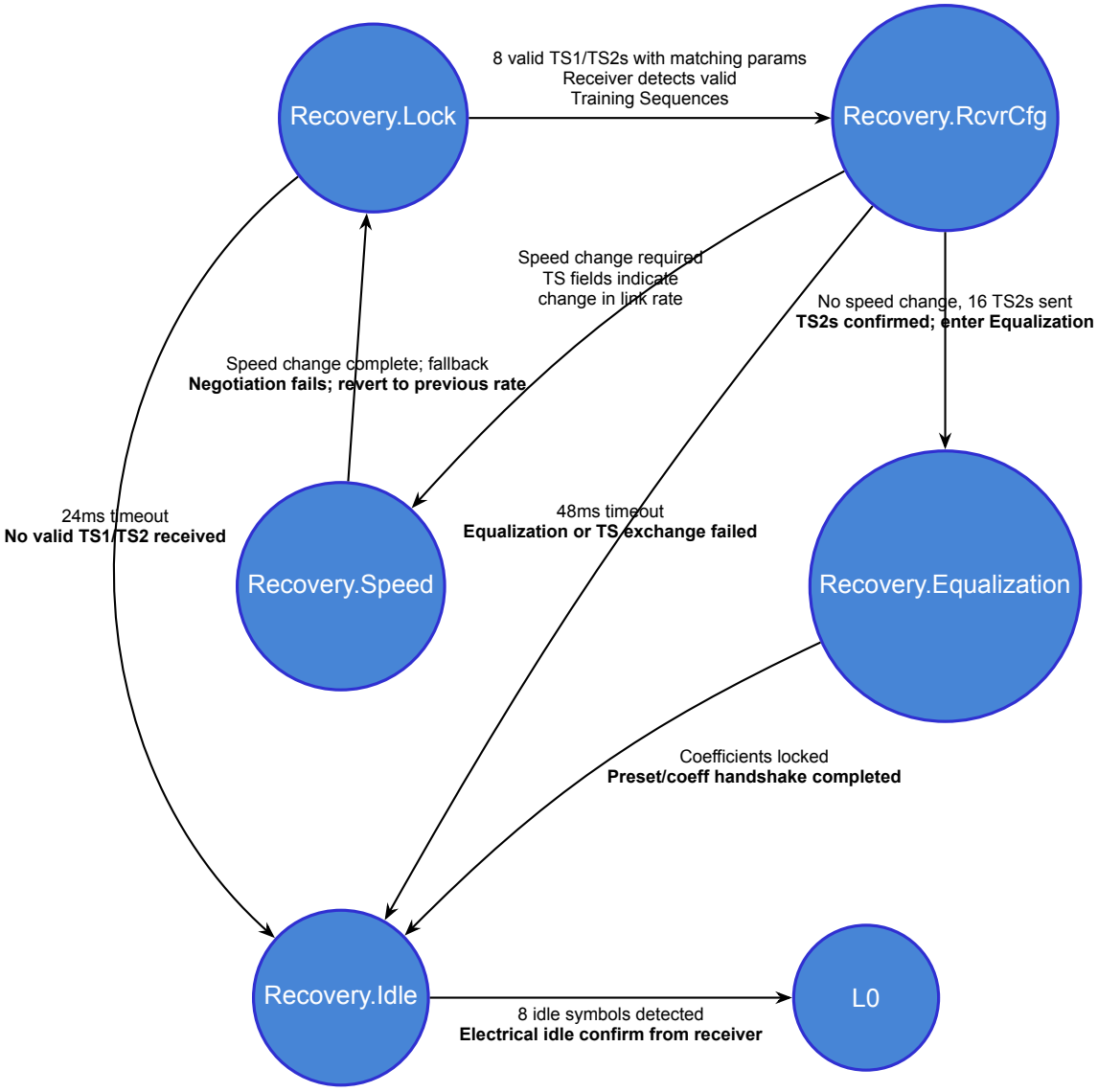
- For 8b/10b encoding, idle data is normally sent on all lanes.
- For 128b/130b encoding, an SDS (Start Data Stream) is sent to start a data stream, followed by idle data symbols on all lanes.

Exit Conditions:

- **Exit to L0:** If either 8b/10b or 128b/130b encoding is used and 8 consecutive idle symbol times are received with 16 idle symbols sent.

  For 128b/130b, the exit must not come from Recovery.RcvrCfg, and deskew must be resolved before data stream processing.
- **Exit to Configuration:** If a higher layer requests reconfiguration (e.g., link width change), or if any lane receives two TS1s with PAD lane numbers signaling a width change.
- **Exits to Detect:** After a 2ms timeout.

5.7.7 Recovery State Flow



6 The Universal Verification Methodology

6.1 What is UVM?

The **Universal Verification Methodology (UVM)** is a standardized, SystemVerilog-based methodology for verifying complex digital designs. It provides a structured and reusable framework for building testbenches in a modular and scalable way. UVM promotes reusable components, constrained-random stimulus, coverage-driven verification, and object-oriented techniques to handle large and complex verification tasks effectively.

6.2 Why UVM?

As digital designs grow in complexity, traditional verification approaches become inefficient and hard to scale. UVM addresses these challenges by:

- Providing a reusable and layered architecture.
- Supporting constrained-random stimulus generation.
- Enabling automation of coverage closure.
- Encouraging code reuse across projects and teams.
- Promoting modularity and abstraction using object-oriented concepts.

6.3 Main UVM Components

A UVM testbench is composed of several key components, each responsible for a specific role in the verification process. Below is a detailed description of the major components:

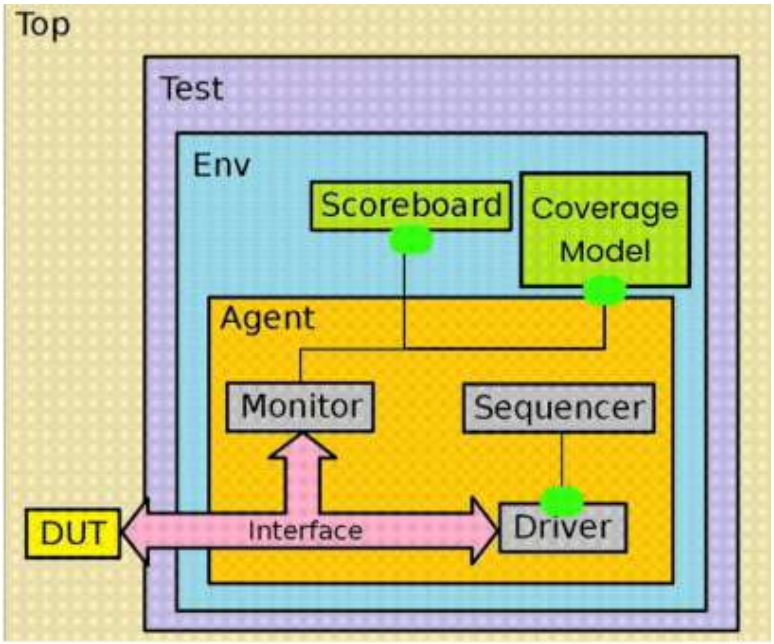


Figure 32: UVM TestBench Architecture

6.3.1 Test

The **test** class is the starting point of the verification. It defines the configuration of the environment and test scenario to run. It instantiates the environment, sets up run-time configuration parameters, and initiates the test sequences. The test class extends from ‘uvm_test‘.

Example: Configuring number of transactions, enabling specific checks, or selecting corner-case tests.

6.3.2 Environment (env)

The **environment** is a container that organizes and holds all the major components of the testbench. It usually includes multiple agents, a scoreboard, and coverage components. The environment defines how these components interact and is typically customized per project.

Example: An environment for a processor may include separate agents for instruction and data buses.

6.3.3 Agent

An **agent** represents a reusable block that contains all components needed to interact with a DUT interface. It typically includes:

- A **driver** to send transactions to the DUT.
- A **monitor** to observe the DUT's output.
- A **sequencer** to control transaction flow.

Agents can be either *active* (drive and monitor) or *passive* (only monitor). This modularity makes it easy to plug multiple agents into one environment.

6.3.4 Driver

The **driver** converts high-level abstract transactions (sequence items) into pin-level signal toggles that interface with the DUT. It takes instructions from the sequencer and drives corresponding signals on the DUT pins.

Example: In a bus protocol, the driver converts a read/write transaction into signal toggling on 'addr', 'data', and 'valid'.

6.3.5 Sequencer

The **sequencer** manages the flow of sequence items (transactions) sent to the driver. It selects, orders, and schedules the transactions to be executed. Sequencers can generate simple or complex patterns, random or directed.

Example: Sending a burst of read operations followed by a write, based on constraints or a defined scenario.

6.3.6 Monitor

The **monitor** observes the DUT interface passively and collects information (e.g., transactions or protocol events). It does not affect simulation but is used for checking functionality and collecting coverage data.

Example: A monitor watching a UART interface decodes serial output into data transactions and sends them to the scoreboard.

6.3.7 Transaction (Sequence Item)

The **transaction** or **sequence item** represents a data-level abstraction of a stimulus or response. It defines fields and constraints and is passed between components (e.g., sequencer and driver).

Example: A transaction for a memory protocol might include fields for address, data, operation type (read/write), and ID.

6.3.8 Scoreboard

The **scoreboard** compares expected outputs (reference model) against actual outputs from the DUT. It implements checkers to validate functional correctness and report mismatches.

Example: Verifying that the data read from memory matches the expected value written earlier.

6.3.9 Testbench Top

The **testbench top module** is the SystemVerilog module that instantiates the DUT and starts the UVM simulation. It connects the DUT and virtual interfaces to the UVM world, allowing drivers and monitors to control or observe real signals.

Example: The top.sv file connects the DUT to the UVM components and calls run_test().

6.4 UVM Phases

UVM defines a set of simulation phases that execute in a specific order to ensure predictable behavior. Key phases include:

- **build_phase**: Components are constructed and configured.
- **connect_phase**: TLM (transaction-level modeling) ports and exports are connected.
- **run_phase**: Actual simulation is performed. Sequences run and DUT responds.
- **check_phase**: Results are checked for errors.
- **report_phase**: Final summary of pass/fail and coverage.

6.5 Functional Coverage

Functional coverage is a user-defined metric that checks whether all required functionalities of the design have been exercised by the testbench. Unlike code coverage, which is tool-generated and structural, functional coverage focuses on what behavior or use-cases are important from a design and verification perspective.

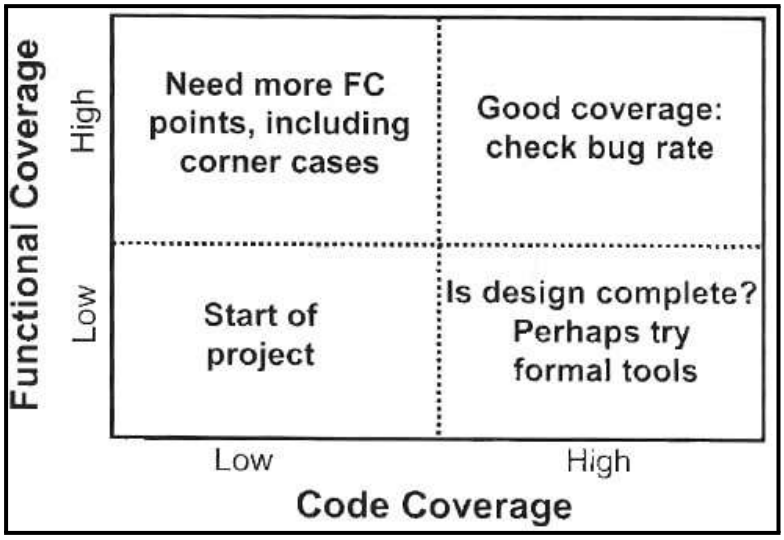


Figure 33: Code Coverage and Functionnal Coverage

Functional coverage helps ensure that:

- All key design features are tested (e.g., different instruction types, address ranges, protocol handshakes).
- Critical corner cases or boundary conditions are exercised.
- Randomized testing is guided and measured meaningfully.

How to write functional coverage: Functional coverage is implemented using `covergroup` constructs in SystemVerilog. Within a covergroup, you define `coverpoints` and `cross` coverage of variables and scenarios of interest.

Example:

```
covergroup mem_cg;
  coverpoint addr { bins low = {[0:15]}; bins high = {[240:255]}; }
  coverpoint op   { bins ops[] = {READ, WRITE}; }
  cross addr, op;
endgroup
```

This example checks whether both read and write operations happen across the low and high memory address regions.

Benefits:

- Helps identify untested functionality.
- Drives test generation (coverage-driven verification).
- Contributes to overall verification completeness.

Coverage closure is the process of writing tests or sequences to hit uncovered bins, ensuring all behaviors are validated.

6.6 Code Coverage

Code coverage is an automatic measure collected by simulators to determine how much of the design code (RTL) was executed during a simulation run. It reflects whether test stimulus reached all parts of the hardware description.

There are several types of code coverage:

- **Line Coverage**: Verifies whether each line of RTL was executed at least once.
- **Toggle Coverage**: Checks whether every signal bit changed from 0 to 1 and 1 to 0 during simulation.
- **Condition and Branch Coverage**: Ensures that all conditions in if/else, case, and ternary expressions were evaluated both true and false.
- **FSM Coverage (Finite State Machine)**: Monitors if all states and transitions were exercised.

Example: If your RTL contains:

```
if (req && !grant) begin
  state <= WAIT;
end
```

Then branch coverage will report incomplete if the case ‘req = 1‘ and ‘grant = 1‘ never occurs.

Why it matters:

- It reveals unused parts of the design.
- It complements functional coverage by ensuring stimulus reached the RTL.
- High code coverage boosts confidence in the quality of verification.

Note: 100% code coverage does not mean a design is bug-free. It just means all code was exercised—not that it behaved correctly.

6.7 Assertions

Assertions are conditions written in SystemVerilog to automatically verify properties of a design during simulation. They act as self-checking mechanisms and help catch bugs close to the source.

Assertions can be:

- **Immediate Assertions**: Checked at the current simulation time. Used for basic conditions like input validity.
- **Concurrent Assertions**: Written using **SystemVerilog Assertions (SVA)** to express temporal relationships over time.

Why assertions are important:

- They localize bugs early by checking protocol or timing violations.
- They reduce the need for manual checkers or scoreboard logic.
- They provide formal properties useful for formal verification tools.

Example of Immediate Assertion:

```
assert (data != 32'hx) else $fatal("Data is undefined!");
```

Example of Concurrent Assertion:

```
property p_grant_after_req;
  @(posedge clk) req |-> ##1 grant;
endproperty
assert property (p_grant_after_req);
```

This states that if ‘req‘ goes high on a clock edge, then ‘grant‘ must be high one clock cycle later.

Best practices:

- Use assertions near the source of signal generation (inside RTL or interfaces).
- Cover key interface-level handshakes, timing relationships, and legal transitions.
- Use assertion messages to make debug easier when failures happen.

6.8 Summary

- **Functional coverage** ensures that the intended design behavior has been thoroughly tested.
- **Code coverage** ensures that all parts of the RTL code are exercised during simulation.
- **Assertions** act as automated runtime checkers that help detect design and protocol errors early.

Combining all three is essential for building a robust and complete verification environment. UVM enables the development of scalable, reusable, and robust testbenches that are suitable for industrial-grade verification. Even for academic projects, applying UVM brings valuable insight into modern hardware verification practices. By understanding the roles of each component—test, environment, agents, driver, sequencer, monitor, scoreboard—you can build flexible and automated testbenches for functional validation of digital designs.

7 Testbench Architecture

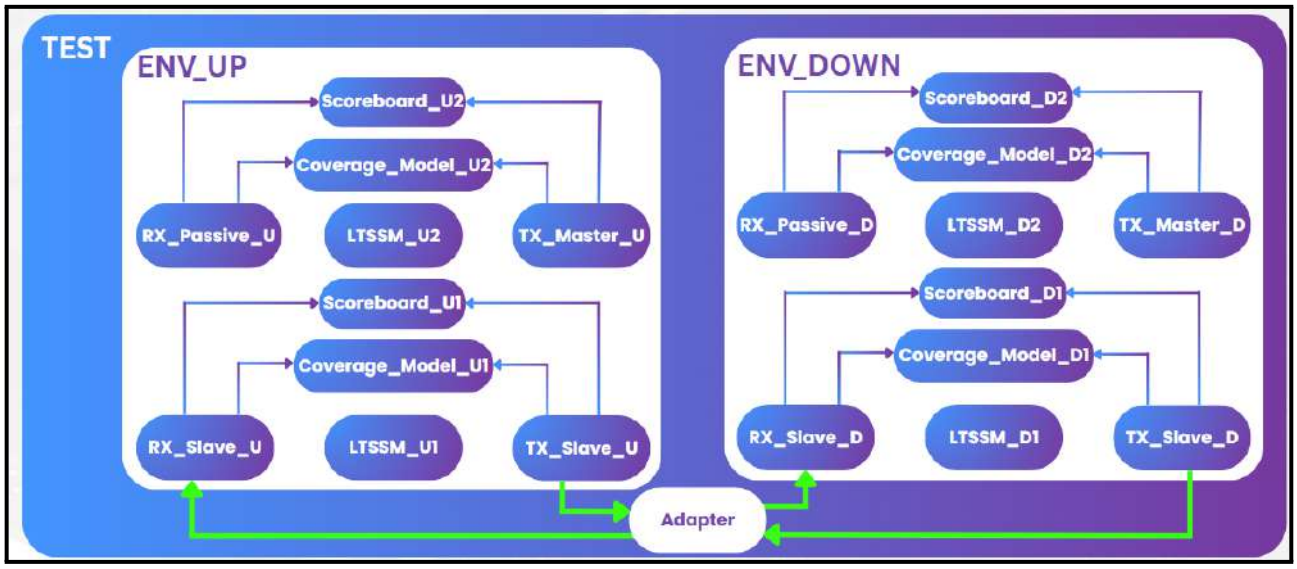


Figure 34: TestBench Architecture

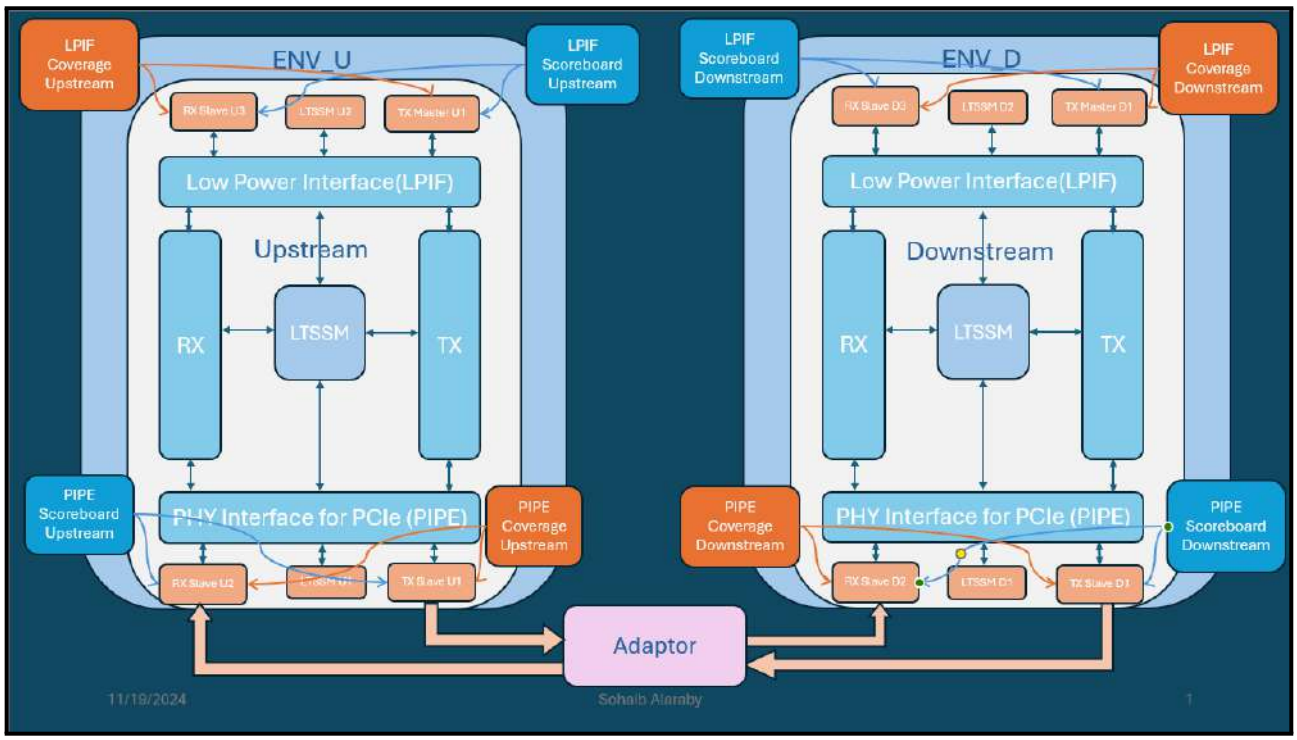


Figure 35: TestBench and Design Architecture

7.1 Introduction:

The DUT consists of two devices: an upstream device and a downstream device, connected together through a PCIe link. The testbench is designed with two separate environments one for the upstream device and one for the downstream device. Since both environments are almost identical in structure and functionality, it is sufficient to explain only one of them in detail.

7.2 Interfaces:

Interfaces are mainly used to simplify DUT and Testbench connections, bundle related signals and improve reusability. There are two interfaces in Testbench: LPIF and PIPE interface.

- **LPIF:**Located between the Physical Layer and the Data Link Layer. 3 agents are using this interface, which is LTSSM2, TX Master, and RX Passive agents.
- **PIPE:** Located between the Logical part and the Electrical part of the Physical Layer. 3 agents are using this interface, which is LTSSM1, TX Slave, and RX Slave agents.

7.3 Sequence items:

There are four types of sequence items used in the testbench: LPIF, PIPE, LTSSM1, and LTSSM2 sequence items. These sequence items are mainly generated by different sequences to send stimulus to the DUT or to navigate through other testbench components for communication and information sharing.

- **LPIF sequence item:** Used in the TX master agent and RX passive agent for sending/receiving stimulus to/from the DUT through LPIF. It is also used for communication between testbench components that work with LPIF, such as the previously mentioned agents, the LPIF scoreboard, and the LPIF coverage model.
- **PIPE sequence item:** Used in the TX slave agent and RX slave agent for sending/receiving stimulus to/from the DUT through PIPE. It is also used for communication between testbench components that work with PIPE, such as the previously mentioned agents, the PIPE scoreboard, and the PIPE coverage model. Electrical part of the Physical Layer. 3 agents are using this interface, which is LTSSM1, TX Slave, and RX Slave agents.
- **LTSSM1 sequence item:** A simplified version of the PIPE sequence item, used in the LTSSM1 agent to drive stimulus to the LTSSM through the PIPE interface.
- **LTSSM2 sequence item:** A simplified version of the LPIF sequence item, used in the LTSSM2 agent to drive stimulus to the LTSSM through the LPIF interface.

7.4 Sequences:

Sequences allow testbench developers to create random, constrained-random, or directed stimulus patterns to verify different DUT scenarios. They are typically executed on a sequencer, which coordinates with the driver to pass sequence items to the DUT.

- **Reset Sequences:** Used to reset the DUT. There are two reset sequences: LTSSM1 Reset Sequence and LTSSM2 Reset Sequence, which are used at the PIPE and LPIF interfaces, respectively.
- **Linkup Sequences:** Used to start the Linkup process. There are two linkup sequences: Linkup1 and Linkup2, which are used at the PIPE and LPIF interfaces, respectively.
- **Force Detect Sequence:** Used to execute the Force Detect feature in the DUT. There is one such sequence: Force Detect LTSSM2 Sequence, which is used at the LPIF interface.
- **Data Transmission Sequences:** Primarily used to generate and transmit different TLPs (Transaction Layer Packets) with varying lengths, along with DLLPs (Data Link Layer Packets), at different data rates and in different orders.
- **Error Injection Sequences:** Used to inject errors into various types of packets exchanged between Downstream and Upstream devices. This is achieved through a non-UVM component called the adapter, which connects the two devices.

Note: Virtual sequences are used to coordinate sequences of the same or different types, ensuring they operate together in harmony.

7.5 Agents:

The main purpose of an agent is to encapsulate all verification components related to driving and monitoring an interface such as the driver, monitor, and sequencer into a single, reusable, and configurable unit. In this testbench, there are six different agents:

- **TX Master Agent**
- **RX Passive Agent**
- **RX Slave Agent**
- **TX Slave Agent**
- **LTSSM1 Agent**
- **LTSSM2 Agent**

7.5.1 The TX Master Agent

Plays a significant role in the data transmission operation since data transmission sequences are started by the test on TX Master sequencer to deliver data stimulus to the TX Master driver to drive it to the TX of the DUT through LPIF. It contains four main components:

- **TX Master Sequencer:** Controls the flow of transactions from the data transmission sequences to the driver.
- **TX Master Driver:** Converts these transactions into pin-level activity, which is applied to the DUT through the LPIF interface.
- **TX Master Monitor:** Extracts the data transmission packets from the LPIF interface (driven by the TX Master Driver) and sends these packets to both the LPIF scoreboard and the coverage model.
- **TX Master Configuration object:** Holds configuration settings (such as interface handles or flags) required by the TX Master Agent and its sub-components.

7.5.2 The RX Passive Agent:

The RX Passive Agent is called passive because it does not contain any active components like a sequencer or driver. Its main role is to extract received TLP and DLLP packets from the LPIF interface and send them to the LPIF scoreboard for checking and Cover model for coverage. It contains two main components:

- **RX Passive Monitor:** Extracts data transmission packets from the LPIF interface (which are sent by the transmitting device) and sends them to the LPIF scoreboard to verify that the received packets match what was transmitted.
- **RX Passive Configuration Object:** Stores configuration settings (such as interface handles or control flags) required by the RX Passive Agent and its sub-components.

7.5.3 The TX Slave Agent:

it monitors all the Packets especially the Ordered sets that are being sent by the TX to the other device through PIPE interface. It contains two main components:

- **TX Slave Monitor:** Used to Predict the TX LTSSM FSM current state and next state by the help of the Ordered sets which are sent through PIPE interface and it takes into account time out and error injection.
- **TX Slave Configuration Object:** Stores configuration settings (such as interface handles or control flags) required by the TX Slave Agent and its sub-components.

7.5.4 The RX Slave Agent:

It monitors all the Packets especially the Ordered sets that are being received by the RX from the other device through PIPE interface. It contains two main components:

- **RX Slave Monitor:** Used to Predict the RX LTSSM FSM current state and next state by the help of the Ordered sets which are received through PIPE interface and it takes into account time out and error injection.
- **RX Slave Configuration Object:** Stores configuration settings (such as interface handles or control flags) required by the RX Slave Agent and its sub-components.

7.5.5 LTSSM1 Agent/LTSSM2 Agent:

Used to initiate LTSSM PIPE interface/LPIF interface signals to prepare It start Link UP. It contains three main components:

- **LTSSM1 Sequencer/LTSSM2 Sequencer:** Controls the flow of transactions from the sequences to the driver.
- **LTSSM1 Driver/ LTSSM2 Driver:** Converts these transactions into pin-level activity, which is applied to the DUT through the PIPE interface / LPIF interface.
- **LTSSM1 Configuration object/ LTSSM2 Configuration object:** Holds configuration settings (such as interface handles or flags) required by the LTSSM1 Agent/LTSSM2 Agent and its sub-components.

7.6 Scoreboards:

A scoreboard is a verification component used to check the correctness of the DUT's behavior by comparing its actual output against the expected output. In this testbench there are two scoreboards:

- **PIPE Scoreboard:** Used to check the correctness of the DUT FSM transitions, number of Ordered sets, the type of Ordered sets and other Important signals.
- **LPIF Scoreboard:** Used to check the correctness of the Data that has been received by RX by comparing it with the data that has been sent by the TX of the other device. This includes comparing both Packets types, both Packets sizes and both Packets actual data.

7.7 Coverage models:

Coverage models are used to measure how much of the design has been tested. Its main goal is to quantify the effectiveness of the stimulus generated by the testbench and to identify untested scenarios or corner cases. In this testbench there are two coverage models:

- **LPIF Coverage model:** Used to cover all the data transmission scenarios for example: sending/receiving different types of packets (TLPs and DLLps), sending/receiving TLPs with different sizes and different data in same and different streams,etc.
- **PIPE Coverage model:** Used to cover all the possible FSM transitions for both TX and RX.

7.8 Adapter:

Adapter is a non-UVM component and located at the midway of the link connection between both Upstream and Downstream devices. Adapter has two main functionality:

- **Bypassing** Data and Control Packets for normal operations of PCIe
- **Error Injection** which is the deliberate insertion of faults, invalid conditions, or protocol violations into the data or control paths of a DUT during simulation to test its robustness, error handling, and recovery mechanisms.

8 Simulation Results

The primary objectives of this thesis are:

8.1 Detect State

Introduction

The Detect state is the first step in PCIe link initialization. It has two parts: **Detect.Quiet** and **Detect.Active**.

- In Detect.Quiet, the transmitter stays in Electrical Idle (not sending anything), and the receiver checks if there's any signal on the line.
- After a short quiet period, the device moves to Detect.Active, where it starts sending beacon signals to see if another device is connected.

If the receiver detects a valid response (proper termination), the device moves to the next state: Polling.

Simulation

The simulation starts with the transmitter in Detect.Quiet, holding Electrical Idle. This lasts for a fixed time, ensuring the receiver sees no unwanted activity. Then, the device enters Detect.Active and sends beacon signals. These signals are shown clearly in the simulation waveform (see Figure 36).

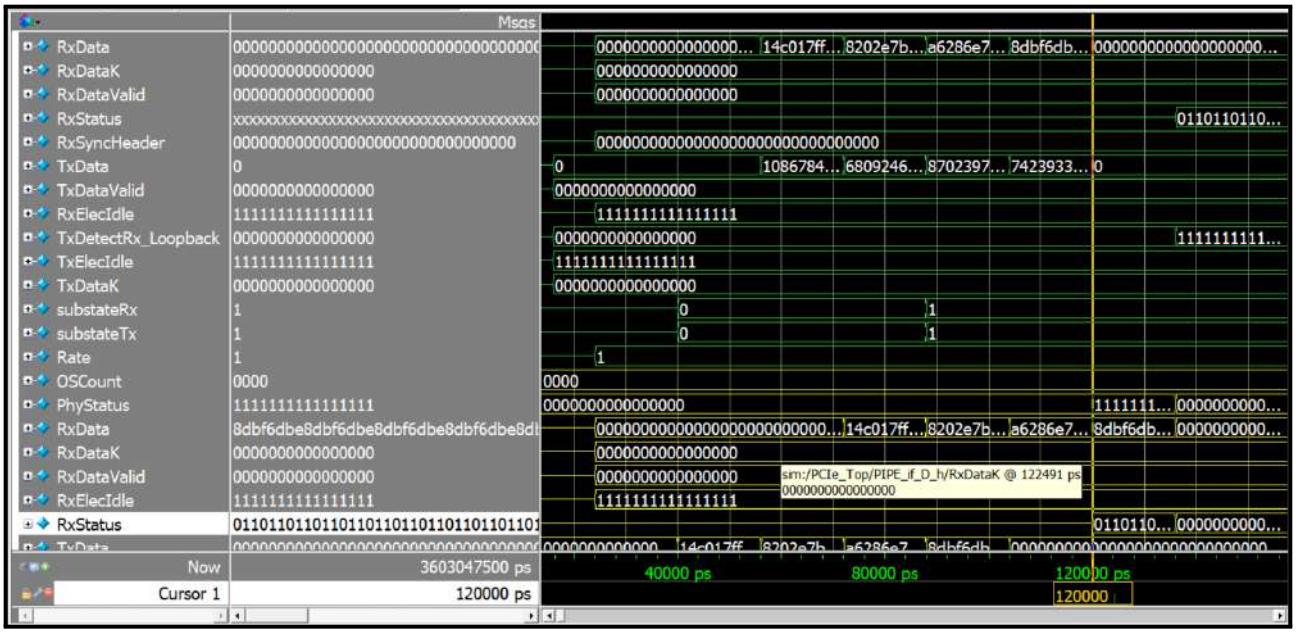


Figure 36: Detect State Simulation Waveform

Result from Transcript

As shown in Figure 37, the simulation log shows a smooth flow:

- Detect.Quiet (no signal sent)
- Then Detect.Active (beacon signals sent)

Scoreboard Check

The scoreboard confirms that:

- The quiet time in Detect.Quiet was correct.
- The number of beacon signals in Detect.Active was as expected.

No errors were found. This proves the Detect state worked properly (Figure 37).

```
# UVM_INFO RX_Slave_D_Monitor.sv(1828) @ 16050: uvm_test_top.PCIe_Env_h.RX_Slave_D_Agent_h.RX_Sla
ve_D_Monitor_h [RX_Slave_D_Monitor] Detect Quiet substate at Downstraem RX side completed success
fully
# UVM_INFO TX_Slave_D_Monitor.sv(451) @ 24050: uvm_test_top.PCIe_Env_h.TX_Slave_D_Agent_h.TX_Slav
e_D_Monitor_h [TX_Slave_D_Monitor] Detect Quiet substate at downstream TX side completed successf
ully
# UVM_INFO PCIe_Scoreboard1_D.sv(700) @ 24050: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard1_D_h [PCIe
_Scoreboard1_D] Downstream Detect_Quiet Approved
# UVM_INFO LTSSM1_U_Driver.sv(92) @ 24050: uvm_test_top.PCIe_Env_h.LTSSM1_U_Agent_h.LTSSM1_U_Driv
er_h [LTSSM1_U_Driver]  before detection
# UVM_INFO RX_Slave_U_Monitor.sv(1925) @ 32050: uvm_test_top.PCIe_Env_h.RX_Slave_U_Agent_h.RX_Sla
ve_U_Monitor_h [RX_Slave_U_Monitor] Detect Quiet substate at Upstream RX side completed successfu
lly
# UVM_INFO TX_Slave_U_Monitor.sv(464) @ 32050: uvm_test_top.PCIe_Env_h.TX_Slave_U_Agent_h.TX_Slav
e_U_Monitor_h [TX_Slave_U_Monitor] Detect Quiet substate at Upstream TX side completed successful
ly
# UVM_INFO PCIe_Scoreboard1_U.sv(762) @ 32050: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard1_U_h [PCIe
 Scoreboard1_U] Upstream Detect Quiet Approved
```

Figure 37: Detect State Scoreboard and Transcript Output

8.2 Polling.Active

Introduction

After the Detect state is completed, the link moves to the Polling state. The first substate is **Polling.Active**.

- In Polling.Active, the transmitter starts sending TS1 Ordered Sets repeatedly.
- These TS1s are used to announce the transmitter's presence and share link information.

The goal is to ensure both sides of the link are awake and ready to train.

Simulation

The simulation shows that once the receiver confirms termination, the transmitter enters Polling.Active and sends TS1 Ordered Sets continuously. These appear as repeated data patterns in the waveform (Figure 38).

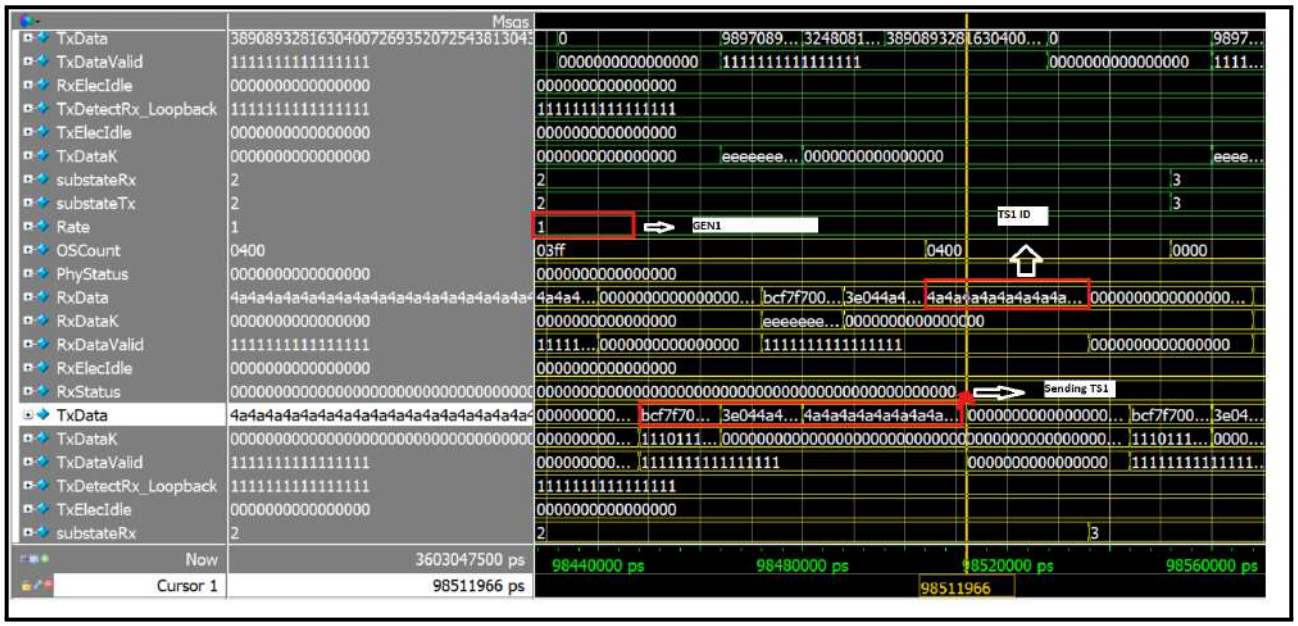


Figure 38: Polling.Active Simulation Waveform

Result from Transcript

As shown in Figure 39, the TX begins sending TS1s at approximately 46,000 ns, and the RX detects them shortly after.

Scoreboard Check

The scoreboard confirms:

- The number of TS1 Ordered Sets sent is correct.
- The timing between transitions is within expected limits.

No protocol errors were found (Figure 39).

```
# UVM_INFO TX_Slave_D_Monitor.sv(939) @ 98504000: uvm_test_top.PCIe_Env_h.TX_Slave_D_Agent_h.TX_S
lave_D_Monitor_h [TX_Slave_D_Monitor] Polling_Active substate at Downstream TX side completed suc
cessfully
# UVM_INFO RX_Slave_D_Monitor.sv(1884) @ 98504000: uvm_test_top.PCIe_Env_h.RX_Slave_D_Agent_h.RX_
Slave_D_Monitor_h [RX_Slave_D_Monitor] Polling_Active substate at Downstraem RX side completed su
ccessfully
# UVM_INFO PCIe_Scoreboard1_D.sv(707) @ 98504000: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard1_D_h [P
CIe_Scoreboard1_D] Downstream Polling_Active Approved
# UVM_INFO TX_Slave_U_Monitor.sv(1034) @ 98520000: uvm_test_top.PCIe_Env_h.TX_Slave_U_Agent_h.TX_
Slave_U_Monitor_h [TX_Slave_U_Monitor] Polling_Active substate at Upstream TX side completed succ
essfully
# UVM_INFO RX_Slave_U_Monitor.sv(1978) @ 98520000: uvm_test_top.PCIe_Env_h.RX_Slave_U_Agent_h.RX_
Slave_U_Monitor_h [RX_Slave_U_Monitor] Polling_Active substate at Upstream RX side completed succ
essfully
# UVM_INFO PCIe_Scoreboard1_U.sv(769) @ 98520000: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard1_U_h [P
CIe_Scoreboard1_U] Upstream Polling_Active Approved
```

Figure 39: Polling.Active Scoreboard and Transcript Output

8.3 Polling.Configuration

Introduction

In this substate:

- Devices continue sending TS1s while monitoring incoming ones.
- After validation, they switch to TS2s to indicate readiness.

Simulation

After TS1s are exchanged, TS2s begin appearing (Figure 40).

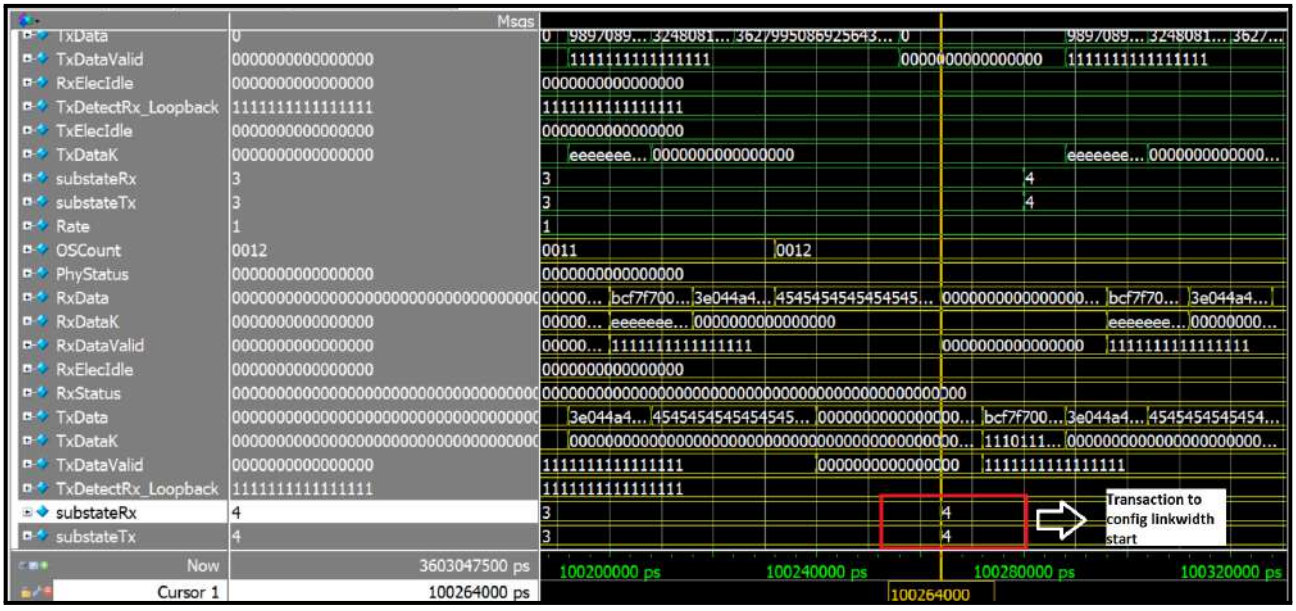


Figure 40: Polling.Configuration Simulation Waveform

Scoreboard Check from Transcript

- TS2 sequences correct
- Link settings matched

(Figure 41)

```
# UVM_INFO RX_Slave_U_Monitor.sv(2020) @ 99368001: uvm_test_top.PCIe_Env_h.RX_Slave_U_Agent_h.RX_
Slave_U_Monitor_h [RX_Slave_U_Monitor] Polling Configuration substate at Upstream RX side complet
ed successfully
# UVM_INFO RX_Slave_D_Monitor.sv(1926) @ 99384001: uvm_test_top.PCIe_Env_h.RX_Slave_D_Agent_h.RX_
Slave_D_Monitor_h [RX_Slave_D_Monitor] Polling Configuration substate at Downstraem RX side compl
eted successfully
# UVM_INFO TX_Slave_D_Monitor.sv(1057) @ 100232000: uvm_test_top.PCIe_Env_h.TX_Slave_D_Agent_h.TX
_Slave_D_Monitor_h [TX_Slave_D_Monitor] Polling Configuration substate at Downstream TX side comp
leted successfully
# UVM_INFO PCIe_Scoreboard1_D.sv(710) @ 100232000: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard1_D_h [
PCIe_Scoreboard1_D] Downstream Polling_Configuration Approved
# UVM_INFO TX_Slave_U_Monitor.sv(1140) @ 100248000: uvm_test_top.PCIe_Env_h.TX_Slave_U_Agent_h.TX
_Slave_U_Monitor_h [TX_Slave_U_Monitor] Polling Configuration substate at Upstream TX side comple
ted successfully
# UVM_INFO PCIe_Scoreboard1_U.sv(772) @ 100248000: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard1_U_h [
PCIe_Scoreboard1_U] Upstream Polling_Configuration Approved
```

Figure 41: Polling.Configuration Scoreboard and Transcript Output

8.4 Config.Link_Width.Start

Introduction
This is the first substate of Configuration.

- The transmitter sends TS1s with link width and lane info.
- The receiver verifies the data.

Simulation
TS1s are visible with correct info (Figure 42).

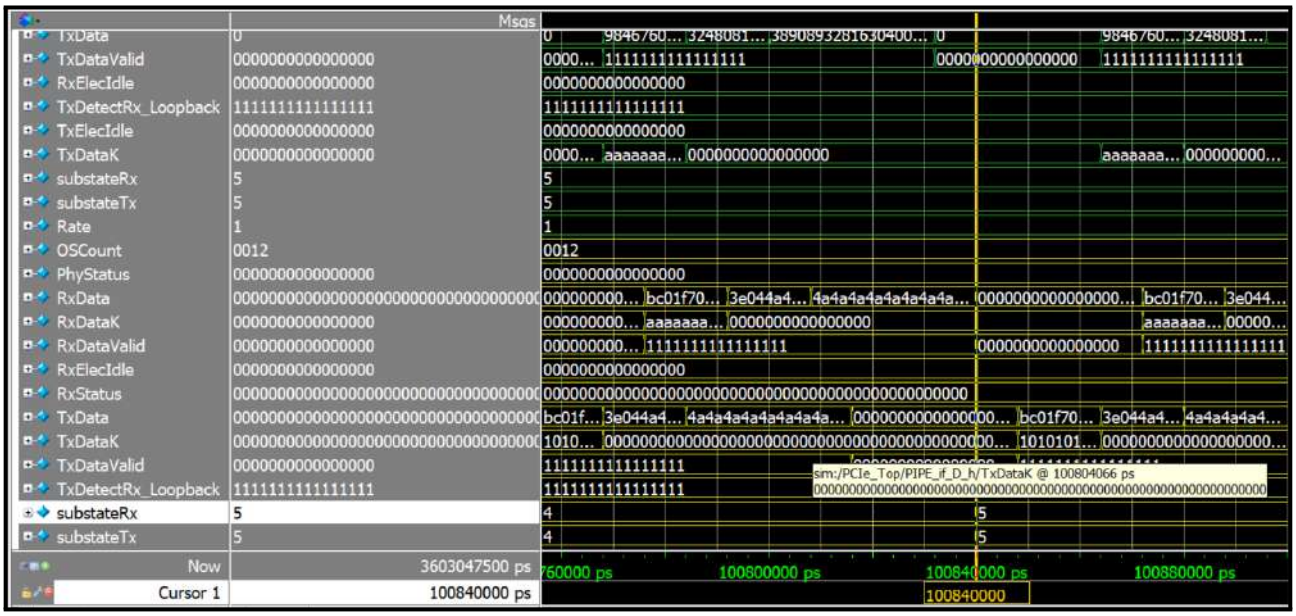


Figure 42: Config.Link_Width.Start Simulation Waveform

Result from Transcript

Starts around 100848,000 ns, TS1s are validated (Figure 43).

Scoreboard Check

- TS1s and link width verified across lanes

(Figure 43)

```
# UVM_INFO RX_Slave_U_Monitor.sv(2060) @ 100520001: uvm_test_top.PCIe_Env_h.RX_Slave_U_Agent_h.RX
_Slave_U_Monitor_h [RX_Slave_U_Monitor] Config Link Width Start substate at Upstream RX side comp
leted successfully
# UVM_INFO TX_Slave_U_Monitor.sv(1257) @ 100576000: uvm_test_top.PCIe_Env_h.TX_Slave_U_Agent_h.TX
_Slave_U_Monitor_h [TX_Slave_U_Monitor] Config_Link_Width_Start substate at Upstream TX side comp
leted successfully
# UVM_INFO PCIe_Scoreboard1_U.sv(775) @ 100576000: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard1_U_h [
PCIe_Scoreboard1_U] Upstream Config_Link_Width_Start Approved
# UVM_INFO RX_Slave_D_Monitor.sv(1968) @ 100824001: uvm_test_top.PCIe_Env_h.RX_Slave_D_Agent_h.RX
_Slave_D_Monitor_h [RX_Slave_D_Monitor] Configuration Linkwidth Start substate at Downstraem RX s
ide completed successfully
# UVM_INFO TX_Slave_D_Monitor.sv(1179) @ 100848000: uvm_test_top.PCIe_Env_h.TX_Slave_D_Agent_h.TX
_Slave_D_Monitor_h [TX_Slave_D_Monitor] Config_Link_Width_Start substate at Downstream TX side co
mpleted successfully
# UVM_INFO PCIe_Scoreboard1_D.sv(713) @ 100848000: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard1_D_h [
PCIe_Scoreboard1_D] Downstream Config_Link_Width_Start Approved
```

Figure 43: Config.Link_Width.Start Scoreboard and Transcript Output

8.5 Config.Link_Width.Accept

Introduction

This substate confirms acceptance of proposed link width.

- Receiver validates TS1s and sends confirmation TS1s.

Simulation

Receiver begins echoing TS1s (Figure 44).

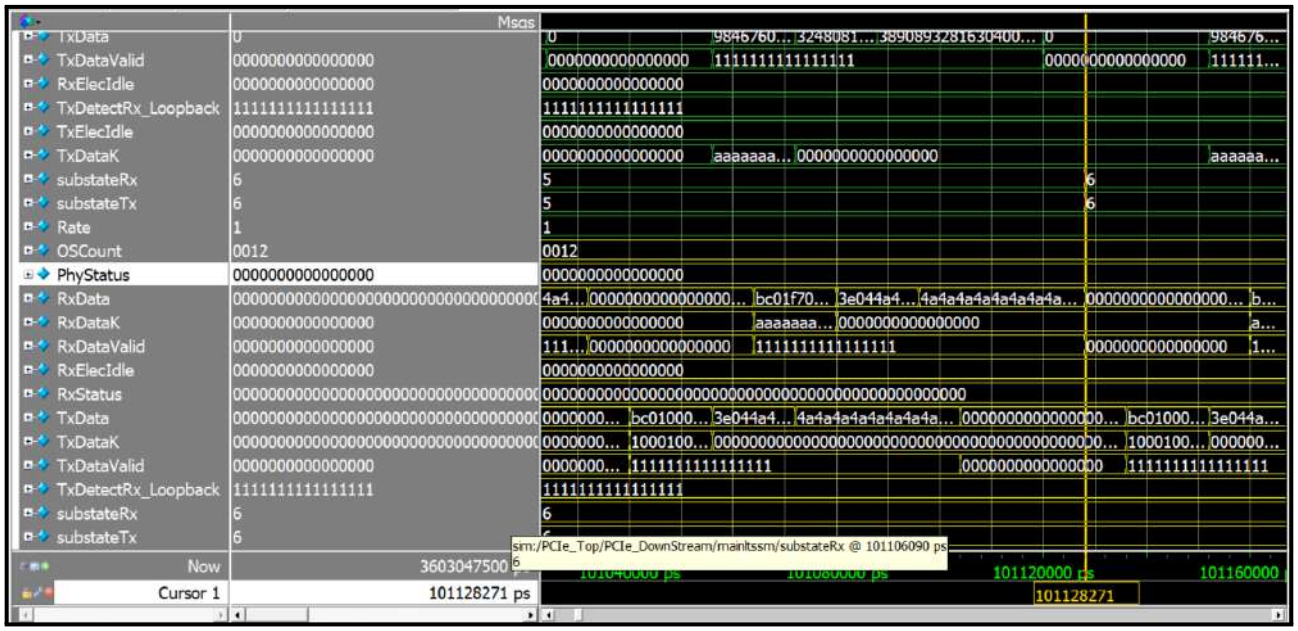


Figure 44: Config.Link_Width.Accept Simulation Waveform

Result from Transcript

Seen around 101152,000 ns (Figure 45).

Scoreboard Check

- TS1s exchanged correctly
- Width settings matched

(Figure 45)

```
# UVM_INFO RX_Slave_D_Monitor.sv(2017) @ 101016001: uvm_test_top.PCIe_Env_h.RX_Slave_D_Agent_h.RX
_Slave_D_Monitor_h [RX_Slave_D_Monitor] Configuration Linkwidth Accept substate at Downstraem RX
side completed successfully
# UVM_INFO TX_Slave_D_Monitor.sv(1289) @ 101096000: uvm_test_top.PCIe_Env_h.TX_Slave_D_Agent_h.TX
_Slave_D_Monitor_h [TX_Slave_D_Monitor] Config Link Width Accept substate at Downstream TX side c
ompleted successfully
# UVM_INFO PCIe_Scoreboard1_D.sv(716) @ 101096000: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard1_D_h [
PCIe_Scoreboard1_D] Downstream Config_Link_Width_Accept Approved
# UVM_INFO RX_Slave_U_Monitor.sv(2105) @ 101096001: uvm_test_top.PCIe_Env_h.RX_Slave_U_Agent_h.RX
_Slave_U_Monitor_h [RX_Slave_U_Monitor] Config Link Width Accept substate at Upstream RX side com
pleted successfully
# UVM_INFO TX_Slave_U_Monitor.sv(1365) @ 101152000: uvm_test_top.PCIe_Env_h.TX_Slave_U_Agent_h.TX
_Slave_U_Monitor_h [TX_Slave_U_Monitor] Config Link Width Accept substate at Upstream TX side com
pleted successfully
# UVM_INFO PCIe_Scoreboard1_U.sv(778) @ 101152000: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard1_U_h [
PCIe_Scoreboard1_U] Upstream Config_Link_Width_Accept Approved
```

Figure 45: Config.Link_Width.Accept Scoreboard and Transcript Output

8.6 Config.Lanenum.Wait

Introduction

This substate ensures all lanes are aligned before config completion.

- Devices wait for valid TS1s with correct lane numbers from all lanes.

Simulation

Lane-wise TS2 verification is performed (Figure 46).

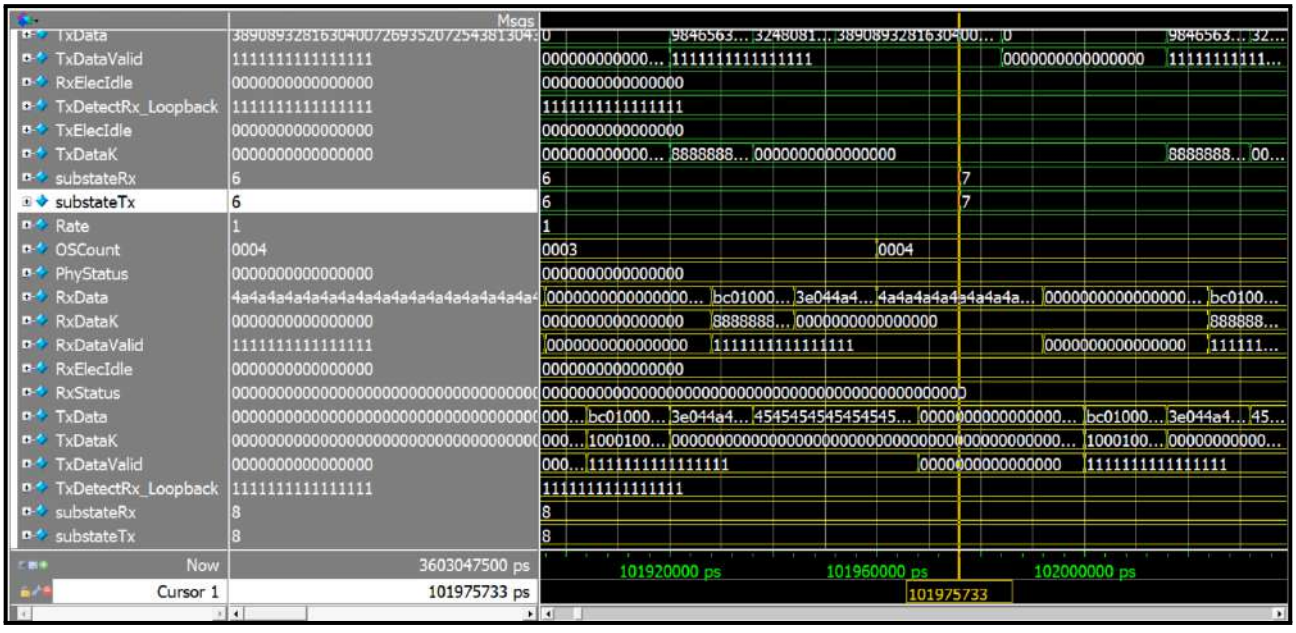


Figure 46: Config.Lanenum.Wait Simulation Waveform

Result from Transcript

Confirmed by 101424,000 ns (Figure 47).

Scoreboard Check

- All lanes active and ordered
- No missing or invalid TS1

(Figure 47)

```
# UVM_INFO RX_Slave_D_Monitor.sv(2075) @ 101400001: uvm_test_top.PCIe_Env_h.RX_Slave_D_Agent_h.RX
_Slave_D_Monitor_h [RX_Slave_D_Monitor]   Configuration Lanenum Wait substate at Downstraem RX si
de completed successfully
# UVM_INFO TX_Slave_D_Monitor.sv(1398) @ 101424000: uvm_test_top.PCIe_Env_h.TX_Slave_D_Agent_h.TX
_Slave_D_Monitor_h [TX_Slave_D_Monitor] Config Lanenum Wait substate at Downstream TX side comple
ted successfully
# UVM_INFO PCIe_Scoreboard1_D.sv(719) @ 101424000: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard1_D_h [
PCIe_Scoreboard1_D] Downstream Config_Lanenum_Wait Approved
```

Figure 47: Config.Lanenum.Wait Scoreboard and Transcript Output

8.7 Configuration.Lanenum.Accept

Introduction

During this substate the Downstream Port must decide if a Link can be established with the Lane numbers returned by the Upstream Port for the Downstream Port If two consecutive TS1s are received with the same non-PAD Link and Lane numbers, and they match the Link and Lane numbers being transmitted in the TS1s for all the Lanes the next substate will be Configuration.Complete for the Upstream If two consecutive TS2s are received with the same non-PAD Link and Lane numbers, and they match the Link and Lane numbers being transmitted in the TS1s for those Lanes, all is well and the next substate will be Configuration.Complete.

Simulation

The simulation confirms the correct sequence As shown in figure 48, Downstream is Receiving TS1

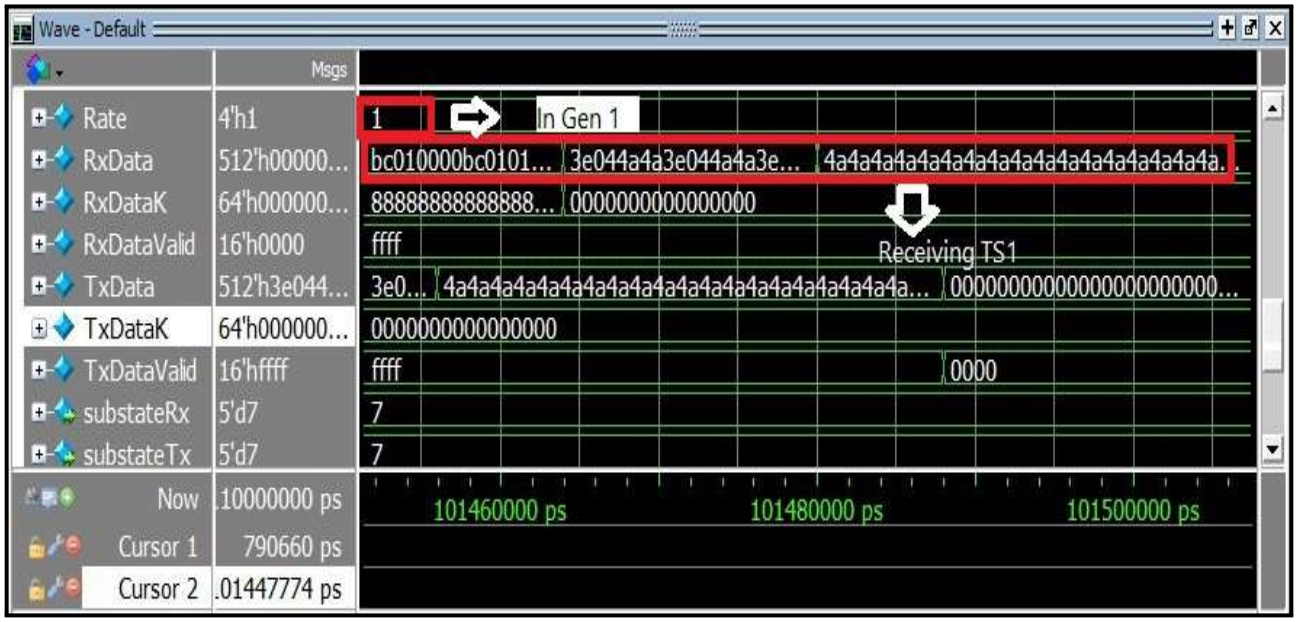


Figure 48: Downstream Receiving TS1

After receiving 2 consecutive TS1 Downstream proceeds to Configuration.Complete Substate as shown in figure 49,

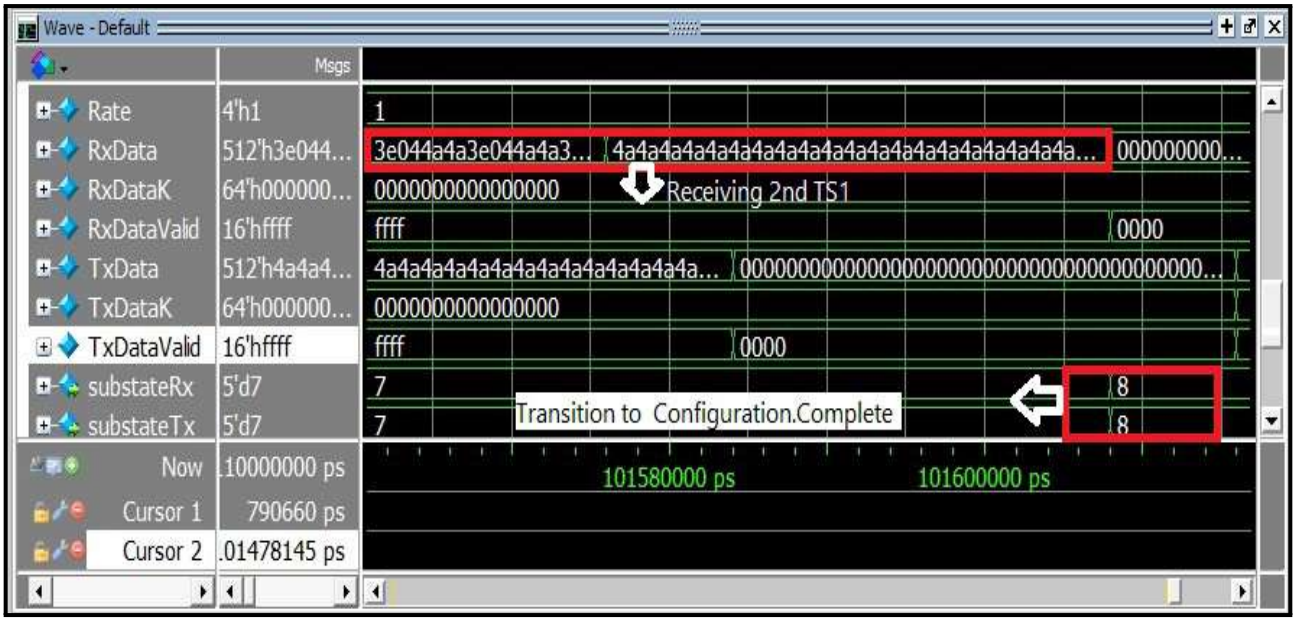


Figure 49: Downstream Transition to Configuration.Complete after receiving 2 TS1

As shown in the figure 50, Upstream is Receiving TS2

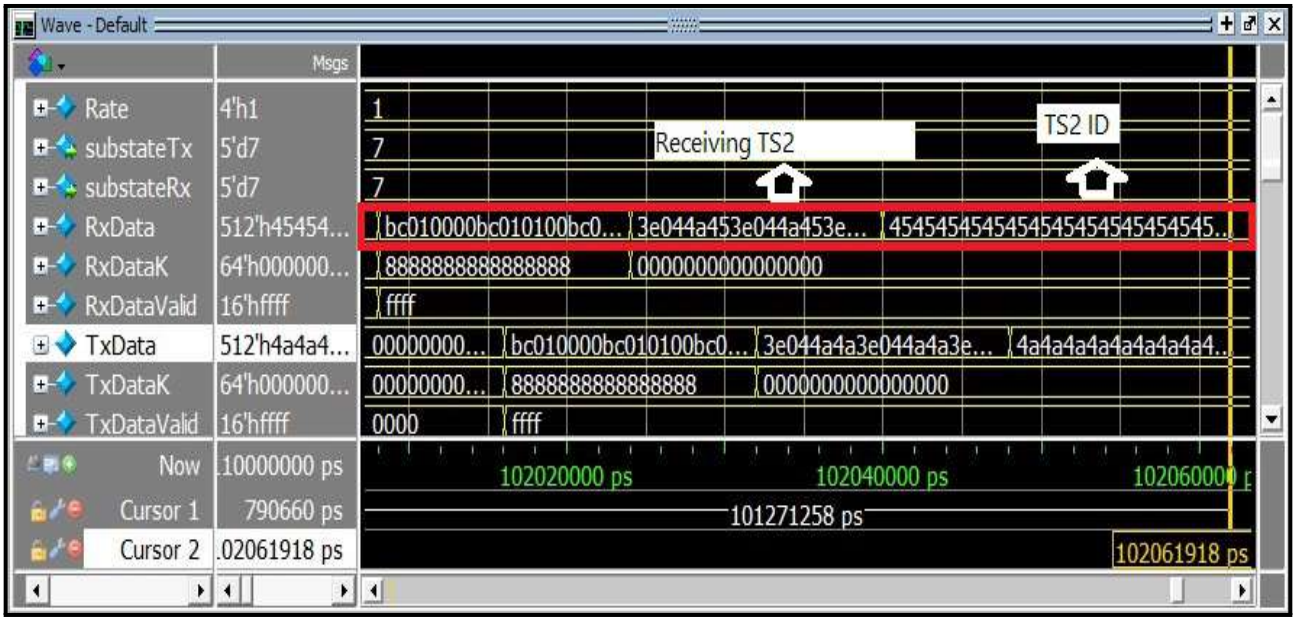


Figure 50: Upstream Receiving TS2

After receiving 2 consecutive TS2 it proceeds to Configuration.Complete Substate as shown in figure 51,

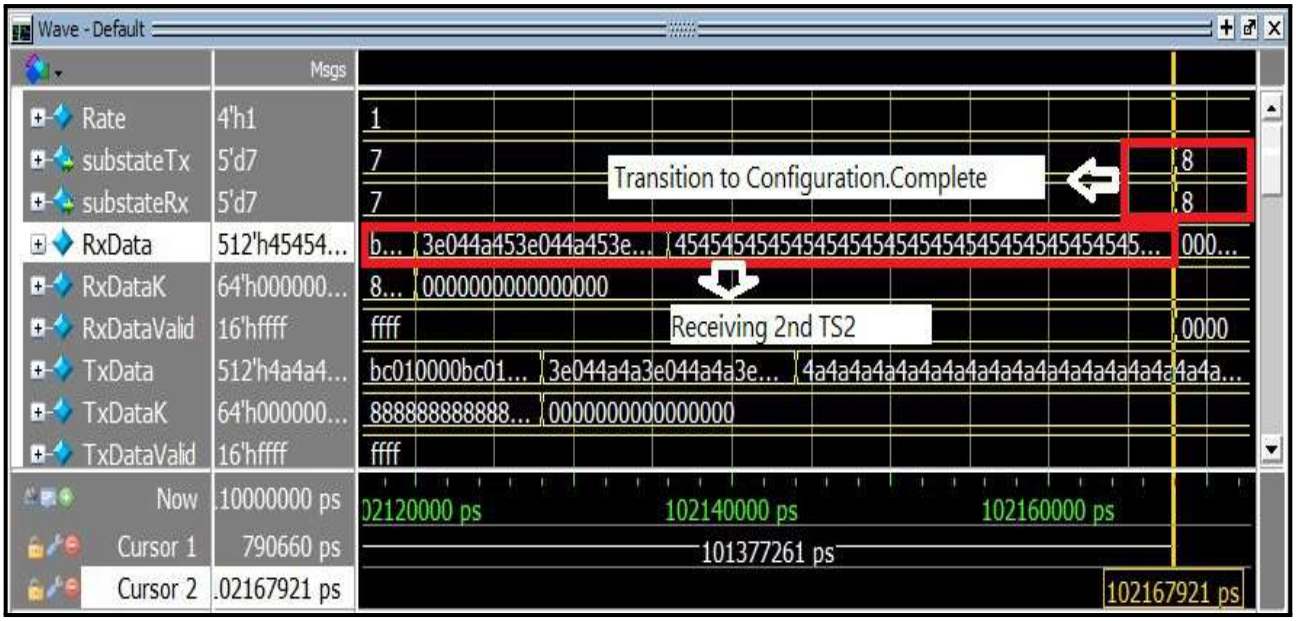


Figure 51: Upstream Transition to Configuration.Complete after receiving 2 TS2

Results from Transcript for TX and RX in Two Devices

Transcript Results for Downstream and Scoreboard Checker for Downstream shown in figure 52

```
# UVM_INFO E:/(Faculty of Engineering, Alexandria University)/Level 4/Graduation Project/Term 2/Last Env/Grad_BY_Youssef/
TB/RX_Slave_D_Monitor.sv(2144) @ 101592001: uvm_test_top.PCIe_Env_h.RX_Slave_D_Agent_h.RX_Slave_D_Monitor_h [RX_Slave_D_M
onitor]  Configuration Lanenum Accept substate at Downstraem RX side completed successfully
# UVM_INFO E:/(Faculty of Engineering, Alexandria University)/Level 4/Graduation Project/Term 2/Last Env/Grad_BY_Youssef/
TB/TX_Slave_D_Monitor.sv(1505) @ 101616000: uvm_test_top.PCIe_Env_h.TX_Slave_D_Agent_h.TX_Slave_D_Monitor_h [TX_Slave_D_M
onitor] Config Lanenum Accept substate at Downstream TX side completed successfully
# UVM_INFO E:/(Faculty of Engineering, Alexandria University)/Level 4/Graduation Project/Term 2/Last Env/Grad_BY_Youssef/
TB/PCIe_Scoreboard1_D.sv(722) @ 101616000: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard1_D_h [PCIe_Scoreboard1_D] Downstream C
onfig_Lanenum_Accept Approved
```

Figure 52: Configuration.Lanenum.Accept in Downstream Completed successfully

Transcript Results for Upstream and Scoreboard Checker for Upstream shown in figure 53

```
# UVM_INFO E:/(Faculty of Engineering, Alexandria University)/Level 4/Graduation Project/Term 2/Last Env/Grad_BY_Youssef/
TB/RX_Slave_U_Monitor.sv(2221) @ 102056001: uvm_test_top.PCIe_Env_h.RX_Slave_U_Agent_h.RX_Slave_U_Monitor_h [RX_Slave_U_M
onitor] Config Lanenum Accept substate at Upstream RX side completed successfully
# UVM_INFO E:/(Faculty of Engineering, Alexandria University)/Level 4/Graduation Project/Term 2/Last Env/Grad_BY_Youssef/
TB/TX_Slave_U_Monitor.sv(1573) @ 102112000: uvm_test_top.PCIe_Env_h.TX_Slave_U_Agent_h.TX_Slave_U_Monitor_h [TX_Slave_U_M
onitor] Config Lanenum Accept substate at Upstream TX side completed successfully
# UVM_INFO E:/(Faculty of Engineering, Alexandria University)/Level 4/Graduation Project/Term 2/Last Env/Grad_BY_Youssef/
TB/PCIe_Scoreboard1_U.sv(784) @ 102112000: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard1_U_h [PCIe_Scoreboard1_U] Upstream Con
fig_Lanenum_Accept Approved
```

Figure 53: Configuration.Lanenum.Accept in Upstream Completed successfully

8.8 Configuration.Complete

Introduction

for both Upstream and Downstream devices the next state will be Configuration.Idle when all Lanes sending TS2s receive 8 TS2s with matching Link and Lane numbers (non-PAD), matching rate identifiers, and matching Link Upconfigure Capability bit in all of them. At least 16 TS2s must also be sent after receiving one TS2.

Simulation

As shown in Figure 54 Downstream is receiving the first TS2

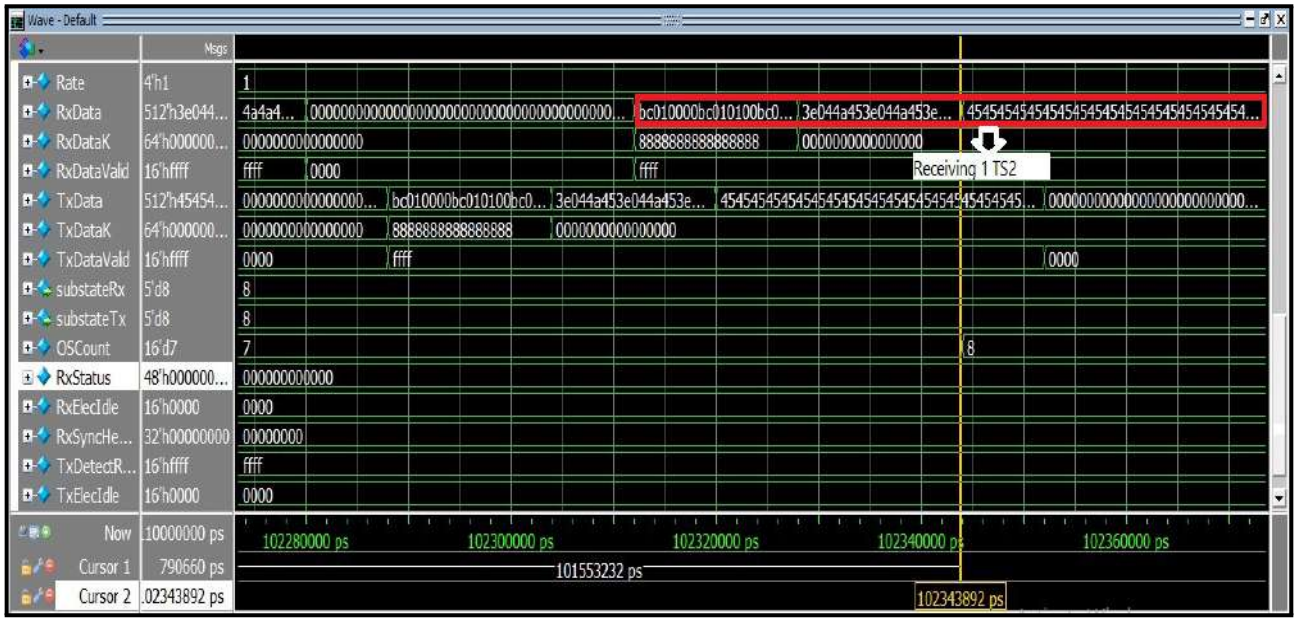


Figure 54: Downstream Receiving first TS2

After receiving 1 TS2, it must send at least 16 TS2 to proceed to Configuration.Idle Substate as shown in Figure 55

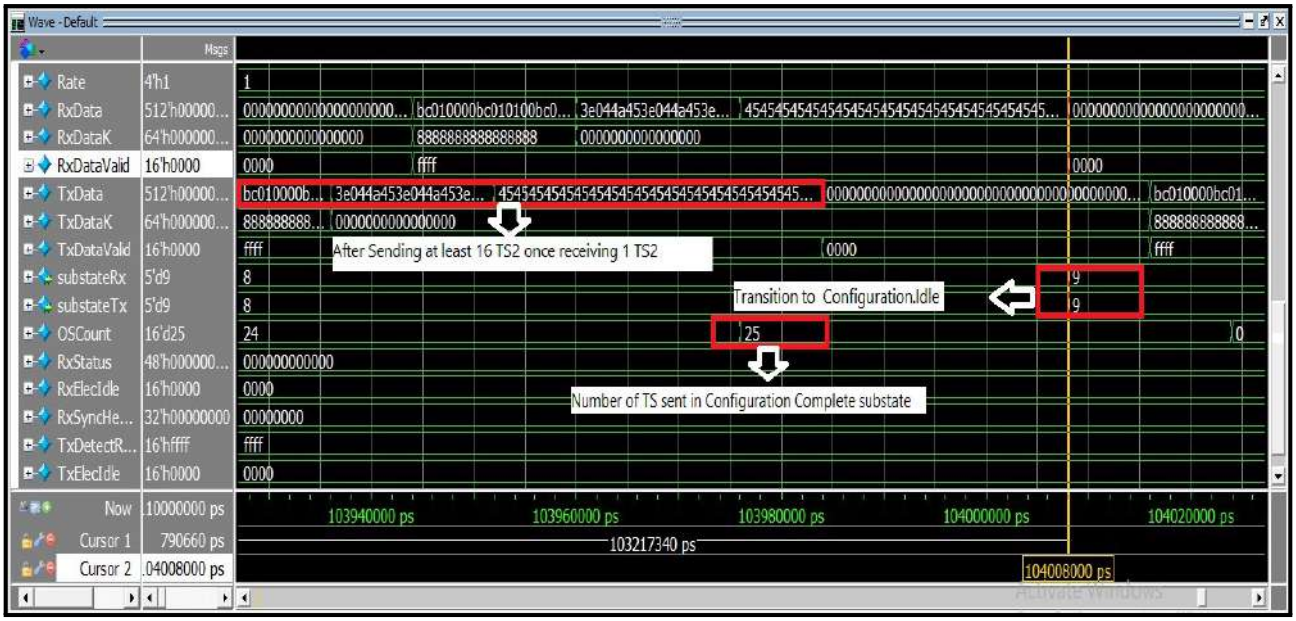


Figure 55: Downstream Sending 16 TS2 to transition to Configuration.Idle

As shown in Figure 56 Upstream is receiving the first TS2

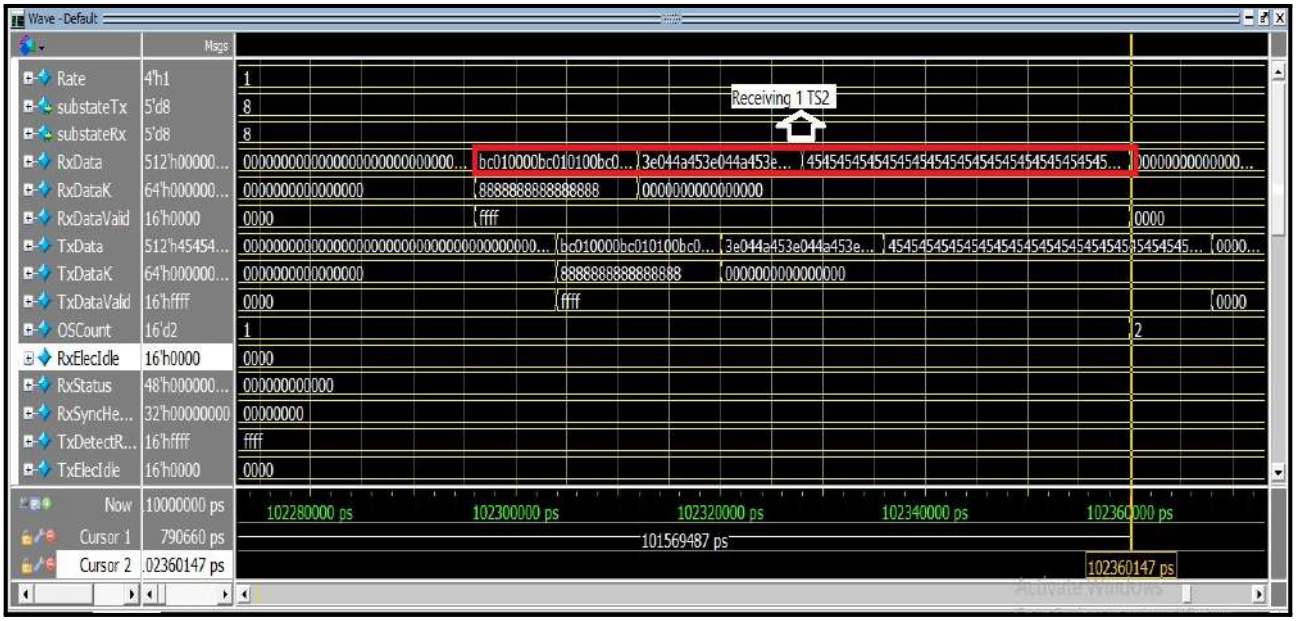


Figure 56: Upstream Receiving first TS2

After receiving 1 TS2, it must send at least 16 TS2 to proceed to Configuration.Idle Substate as shown in Figure 57

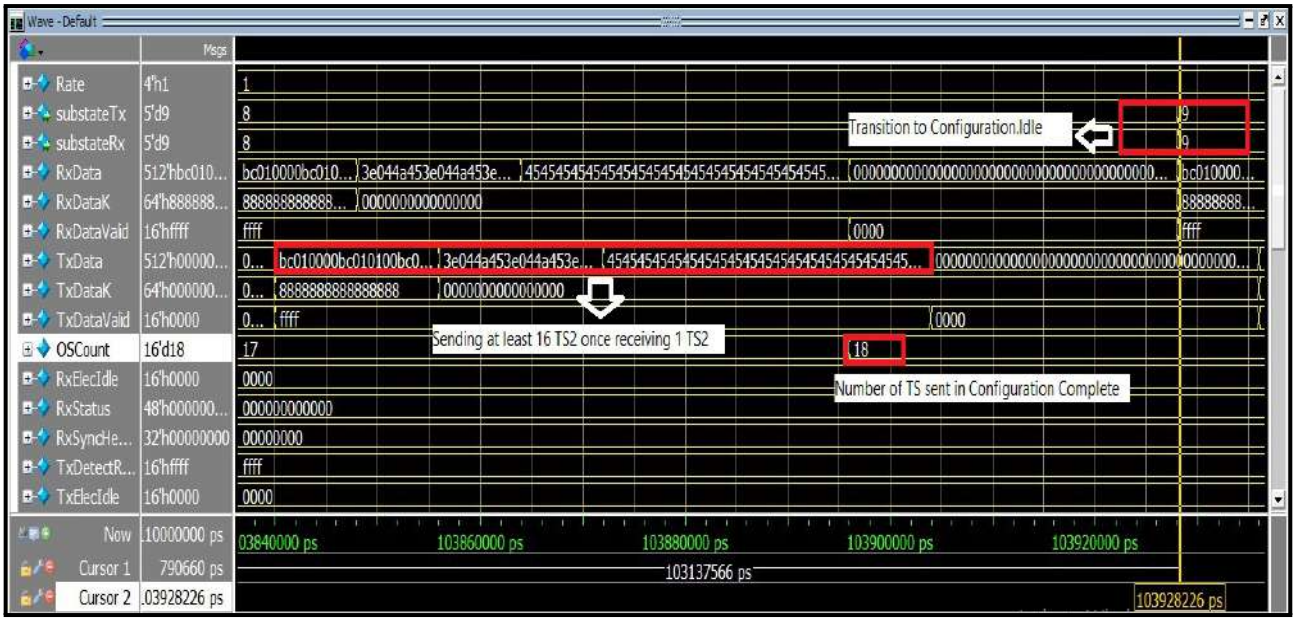


Figure 57: Upstream Sending 16 TS2 to transition to Configuration.Idle

Results from Transcript for TX and RX in Two Devices

Transcript Results for Downstream and Scoreboard Checker for Downstream shown in figure 58

```
# UVM_INFO E:/(Faculty of Engineering, Alexandria University)/Level 4/Graduation Project/Term 2/Last Env/Grad_BY_Youssef/
TB/TX_Slave_D_Monitor.sv(1616) @ 104072000: uvm_test_top.PCIe_Env_h.TX_Slave_D_Agent_h.TX_Slave_D_Monitor_h [TX_Slave_D_M
onitor] Config Complete substate at Downstream TX side completed successfully
# UVM_INFO E:/(Faculty of Engineering, Alexandria University)/Level 4/Graduation Project/Term 2/Last Env/Grad_BY_Youssef/
TB/RX_Slave_D_Monitor.sv(2193) @ 104072000: uvm_test_top.PCIe_Env_h.RX_Slave_D_Agent_h.RX_Slave_D_Monitor_h [RX_Slave_D_M
onitor]  Configuration Complete substate at Downstraem RX side completed successfully
# UVM_INFO E:/(Faculty of Engineering, Alexandria University)/Level 4/Graduation Project/Term 2/Last Env/Grad_BY_Youssef/
TB/PCIe_Scoreboard1_D.sv(725) @ 104072000: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard1_D_h [PCIe_Scoreboard1_D] Downstream C
onfig_Complete Approved
```

Figure 58: Configuration.Complete in Downstream Completed successfully

Transcript Results for Upstream and Scoreboard Checker for Upstream shown in figure 59

```
# UVM_INFO E:/(Faculty of Engineering, Alexandria University)/Level 4/Graduation Project/Term 2/Last Env/Grad_BY_Youssef/
TB/TX_Slave_U_Monitor.sv(1686) @ 103992000: uvm_test_top.PCIe_Env_h.TX_Slave_U_Agent_h.TX_Slave_U_Monitor_h [TX_Slave_U_M
onitor] Config Complete substate at Upstream TX side completed successfully
# UVM_INFO E:/(Faculty of Engineering, Alexandria University)/Level 4/Graduation Project/Term 2/Last Env/Grad_BY_Youssef/
TB/RX_Slave_U_Monitor.sv(2262) @ 103992000: uvm_test_top.PCIe_Env_h.RX_Slave_U_Agent_h.RX_Slave_U_Monitor_h [RX_Slave_U_M
onitor] Config Complete substate at Upstream RX side completed successfully
# UVM_INFO E:/(Faculty of Engineering, Alexandria University)/Level 4/Graduation Project/Term 2/Last Env/Grad_BY_Youssef/
TB/PCIe_Scoreboard1_U.sv(787) @ 103992000: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard1_U_h [PCIe_Scoreboard1_U] Upstream Con
fig Complete Approved
```

Figure 59: Configuration.Complete in Upstream Completed successfully

8.9 Configuration.Idle

Introduction

In this substate, the transmitter is sending Idle data and waiting for the minimum number of received Idle data so this Link can transition to L0 the next state is L0 if 8 consecutive Idle data symbol times are received on all configured Lanes, and 16 symbol times of idle data were sent after receiving one Idle Symbol.

Simulation

The simulation confirms the correct sequence As shown in figure 60,For the Downstream sending and receiving idle symbols

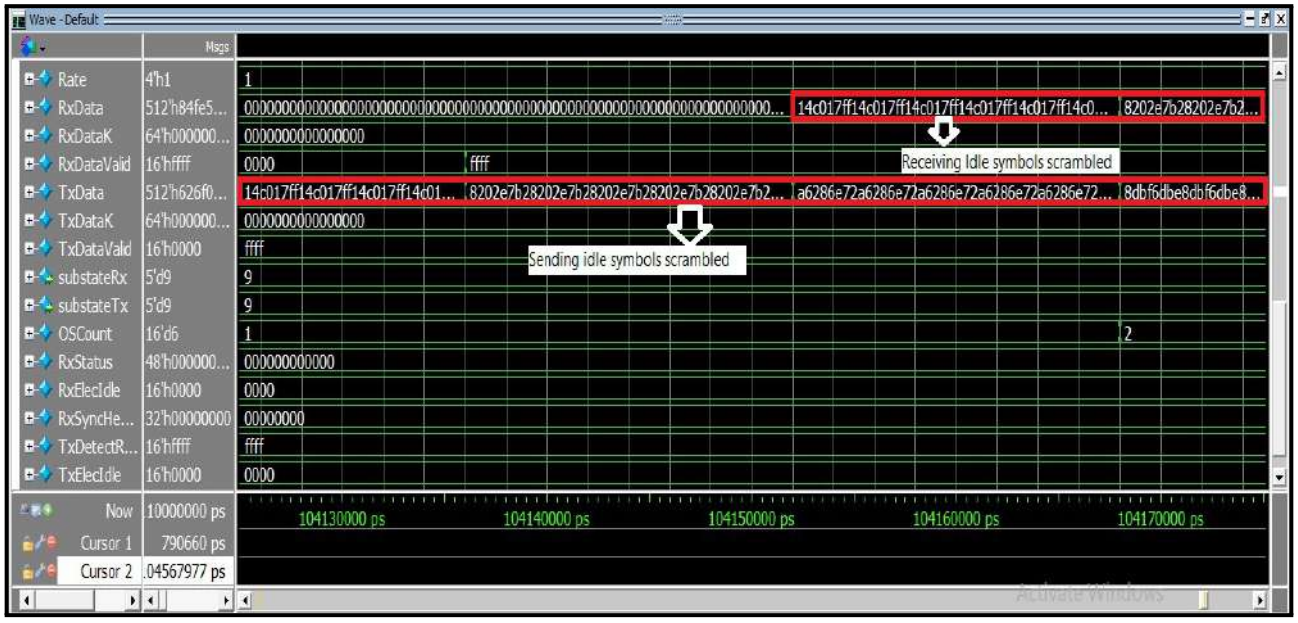


Figure 60: Downstream sending and receiving scrambled idle symbols

Downstream transitions to L0 completing the the linkup in Gen 1 after receiving 8 idle symbols as shown in figure 61,

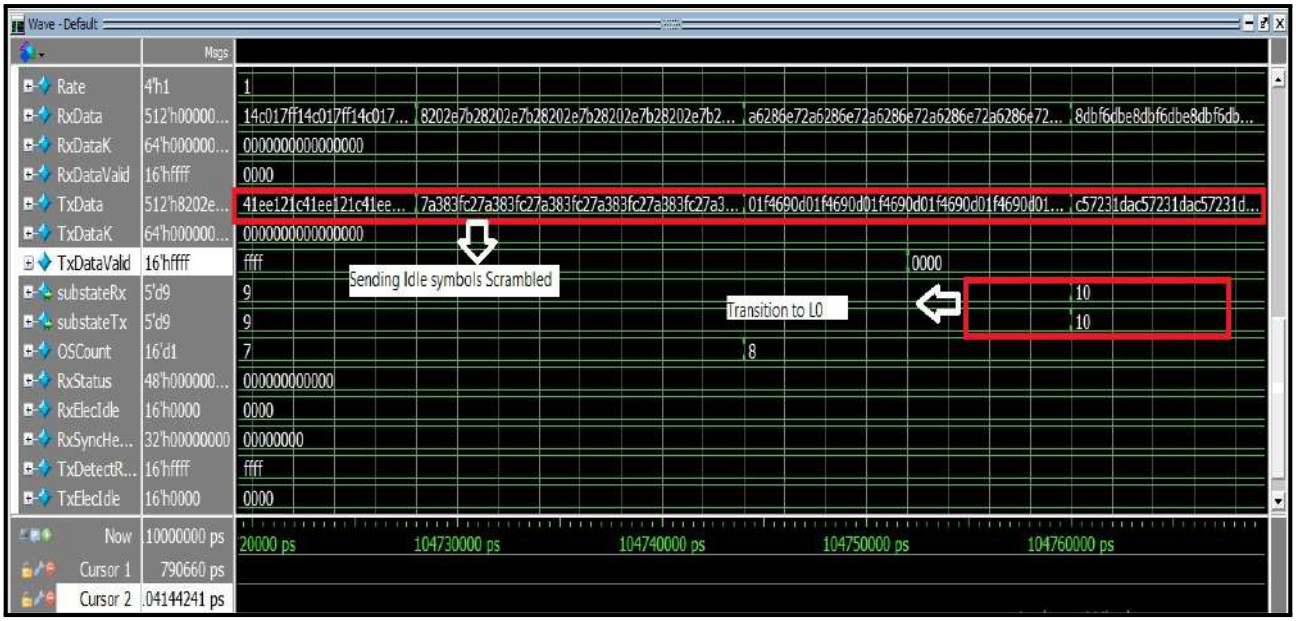


Figure 61: Downstream Transitions to L0 in Gen1

As shown in figure 62,For the Upstream sending and receiving idle symbols

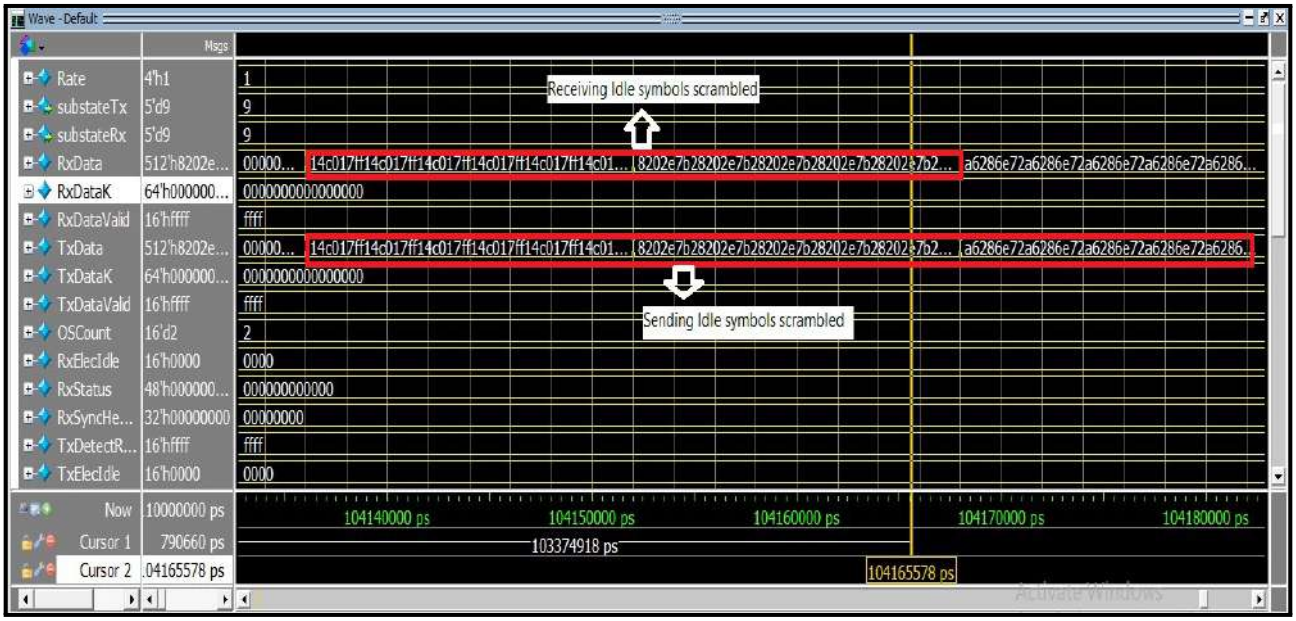


Figure 62: Upstream sending and receiving scrambled idle symbols

Upstream transitions to L0 completing the the linkup in Gen 1 after receiving 8 idle symbols as shown in figure 63,

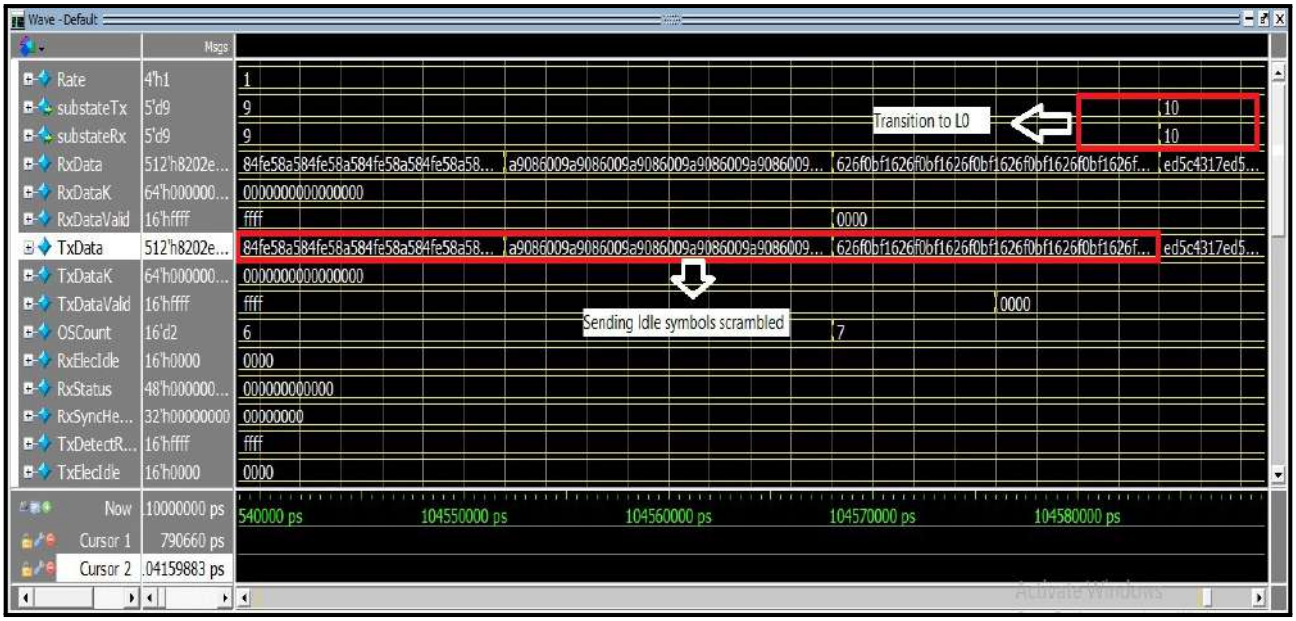


Figure 63: Upstream Transitions to L0 in Gen1

Results from Transcript for TX and RX in Two Devices

Transcript Results for Downstream shown in figures 64 and 65

```
# UVM_INFO E:/(Faculty of Engineering, Alexandria University)/Level 4/Graduation Project/Term 2/Last Env/Grad_BY_Youssef/
TB/TX_Slave_D_Monitor.sv(1656) @ 104184000: uvm_test_top.PCIe_Env_h.TX_Slave_D_Agent_h.TX_Slave_D_Monitor_h [TX_Slave_D_M
onitor] Config Idle substate at Downstream TX side completed successfully
# UVM_INFO E:/(Faculty of Engineering, Alexandria University)/Level 4/Graduation Project/Term 2/Last Env/Grad_BY_Youssef/
TB/RX_Slave_D_Monitor.sv(2239) @ 104184000: uvm_test_top.PCIe_Env_h.RX_Slave_D_Agent_h.RX_Slave_D_Monitor_h [RX_Slave_D_M
onitor]  Configuration Idle substate at Downstraem RX side completed successfully
```

Figure 64: Configuration.Idle in Downstream Completed successfully

```
# UVM_INFO E:/(Faculty of Engineering, Alexandria University)/Level 4/Graduation Project/Term 2/Last Env/Grad_BY_Youssef/
TB/PCIe_Scoreboard1_D.sv(728) @ 104184000: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard1_D_h [PCIe_Scoreboard1_D] Downstream C
onfig_Idle Approved
```

Figure 65: Downstream Scoreboard checker in Configuration.Idle

Transcript Results for Upstream shown in figures 66 and 67

```
# UVM_INFO E:/(Faculty of Engineering, Alexandria University)/Level 4/Graduation Project/Term 2/Last Env/Grad_BY_Youssef/
TB/TX_Slave_U_Monitor.sv(1730) @ 104200000: uvm_test_top.PCIe_Env_h.TX_Slave_U_Agent_h.TX_Slave_U_Monitor_h [TX_Slave_U_M
onitor] Config Idle substate at Upstream TX side completed successfully
# UVM_INFO E:/(Faculty of Engineering, Alexandria University)/Level 4/Graduation Project/Term 2/Last Env/Grad_BY_Youssef/
TB/RX_Slave_U_Monitor.sv(2303) @ 104168000: uvm_test_top.PCIe_Env_h.RX_Slave_U_Agent_h.RX_Slave_U_Monitor_h [RX_Slave_U_M
onitor] Config Idle substate at Upstream RX side completed successfully
```

Figure 66: Configuration.Idle in Upstream Completed successfully

```
# UVM_INFO E:/(Faculty of Engineering, Alexandria University)/Level 4/Graduation Project/Term 2/Last Env/Grad_BY_Youssef/
TB/PCIe_Scoreboard1_U.sv(790) @ 104200000: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard1_U_h [PCIe_Scoreboard1_U] Upstream Con
fig_Idle Approved
```

Figure 67: Upstream Scoreboard checker in Configuration.Idle

8.10 L0

Introduction

This is the fully-operational Link state, during which Logical Idle, TLPs and DLLPs are exchanged between Link neighbors. The next state will be Recovery if a change in the Link speed or Link width is indicated, or if the Link partner initiates this by going to Recovery or Electrical Idle Once receiving TS1 that on all configured Lanes using the same Link and Lane numbers that were set in the Configuration state , the link proceeds to Recovery states to change the speed if that was the purpose of entering the recovery states by setting the speedchange bit in TS1 to 1'b1.

Simulation

The simulations show that Upstream is receiving TS1 in L0 as shown in figure 68

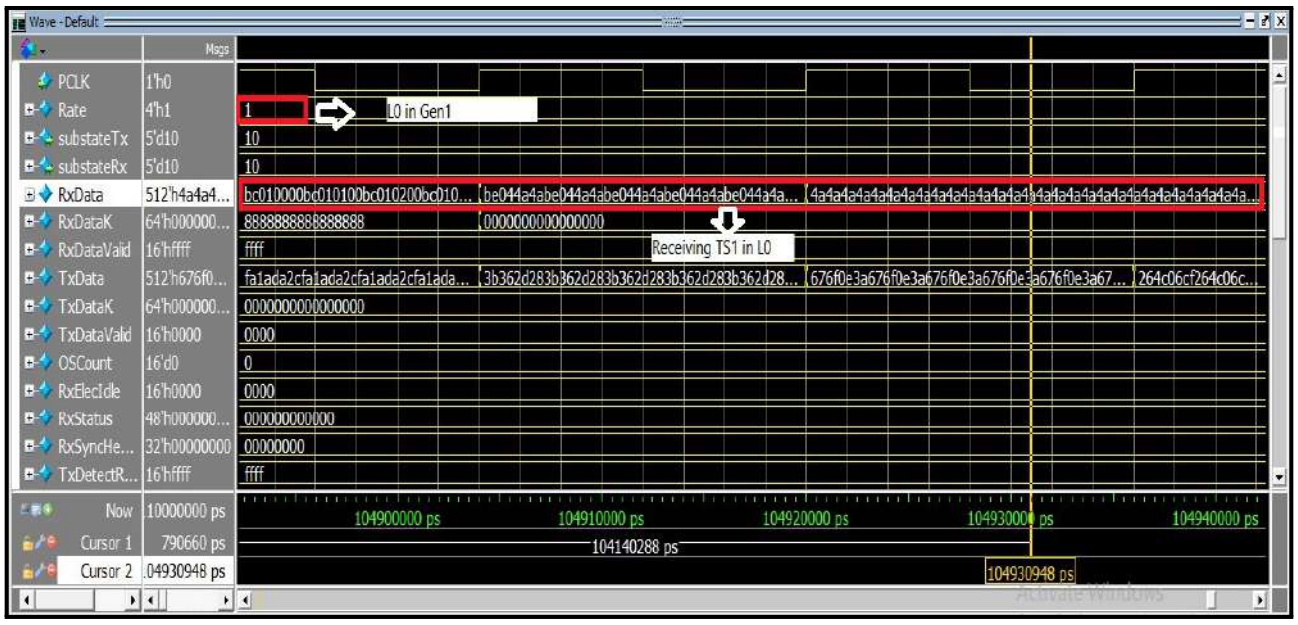


Figure 68: Upstream receiving TS1 in L0

Upstream Receiving TS1 in L0 and transitioning to Recovery substates due to speed change as shown in figure 69

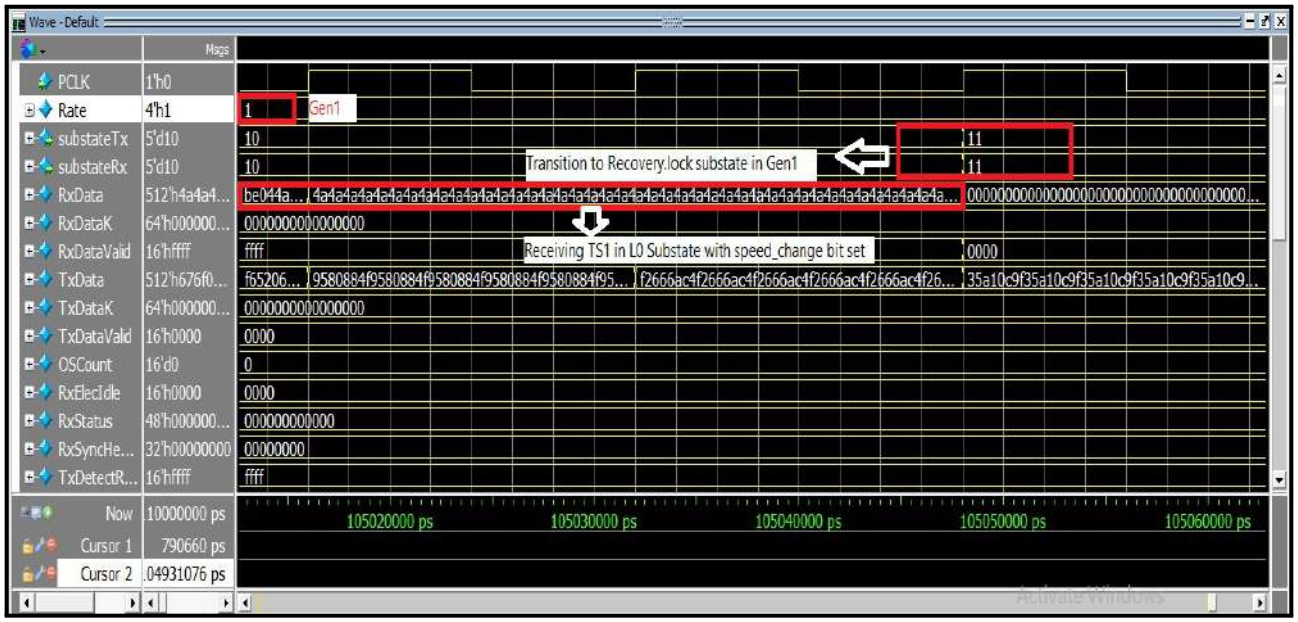


Figure 69: Upstream transitions to Recovery.RcvrLock substate

Results from Transcript for TX and RX in Two Devices

Transcript Results for Downstream shown in figures 70, 71

```
# UVM_INFO E:/(Faculty of Engineering, Alexandria University)/Level 4/Graduation Project/Term 2/Last Env/Grad_BY_Youssef/
TB/TX_Slave_D_Monitor.sv(1712) @ 104216000: uvm_test_top.PCIe_Env_h.TX_Slave_D_Agent_h.TX_Slave_D_Monitor_h [TX_Slave_D_M
onitor] Reached L0 state at Downstream TX side successfully
# UVM_INFO E:/(Faculty of Engineering, Alexandria University)/Level 4/Graduation Project/Term 2/Last Env/Grad_BY_Youssef/
TB/RX_Slave_D_Monitor.sv(2292) @ 104184000: uvm_test_top.PCIe_Env_h.RX_Slave_D_Agent_h.RX_Slave_D_Monitor_h [RX_Slave_D_M
onitor] Reached L0 state at Downstream RX side successfully
```

Figure 70: L0 Reached in Downstream

```
# UVM_INFO E:/(Faculty of Engineering, Alexandria University)/Level 4/Graduation Project/Term 2/Last Env/Grad_BY_Youssef/
TB/PCIe_Scoreboard1_D.sv(731) @ 105176001: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard1_D_h [PCIe_Scoreboard1_D] Downstream L
0 Approved
```

Figure 71: Downstream Scoreboard checker in L0

Transcript Results for Upstream shown in figures 72, 73

```
# UVM_INFO E:/(Faculty of Engineering, Alexandria University)/Level 4/Graduation Project/Term 2/Last Env/Grad_BY_Youssef/
TB/TX_Slave_U_Monitor.sv(1783) @ 104936001: uvm_test_top.PCIe_Env_h.TX_Slave_U_Agent_h.TX_Slave_U_Monitor_h [TX_Slave_U_M
onitor] Reached L0 state at Upstream TX side successfully
# UVM_INFO E:/(Faculty of Engineering, Alexandria University)/Level 4/Graduation Project/Term 2/Last Env/Grad_BY_Youssef/
TB/RX_Slave_U_Monitor.sv(2345) @ 104168000: uvm_test_top.PCIe_Env_h.RX_Slave_U_Agent_h.RX_Slave_U_Monitor_h [RX_Slave_U_M
onitor] Reached L0 state at Upstream RX side successfully
```

Figure 72: L0 Reached in Upstream

```
# UVM_INFO E:/(Faculty of Engineering, Alexandria University)/Level 4/Graduation Project/Term 2/Last Env/Grad_BY_Youssef/
TB/PCIe_Scoreboard1_U.sv(793) @ 104936001: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard1_U_h [PCIe_Scoreboard1_U] Upstream L0
Approved
```

Figure 73: Upstream Scoreboard checker in L0

8.11 Recovery.RcvrLock in Gen1

Introduction

In this state the link achieves the bit lock and symbol/block lock. The next state will be Recovery.RcvrCfg if 8 consecutive TS1s or TS2s are received whose Link and Lane numbers match what is being sent and their speed change bit is equal to the directed speed change variable.

Simulation

The Downstream is receiving TS1 as shown in Figure 74

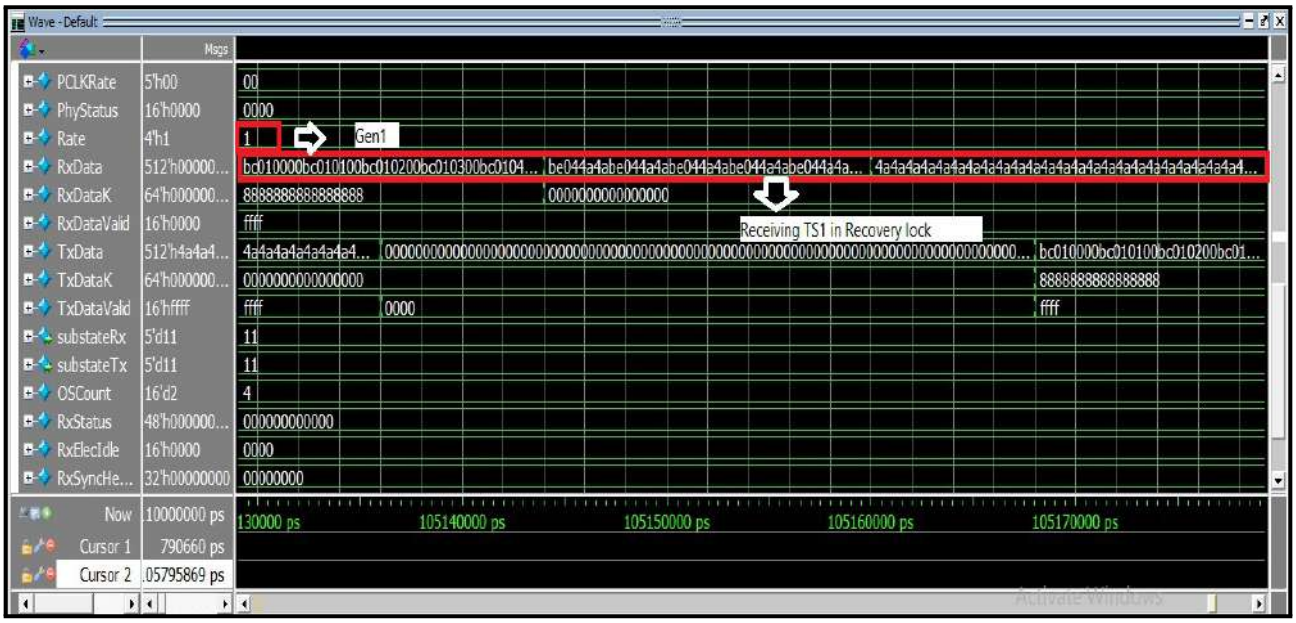


Figure 74: Downstream Receiving TS1 in Recovery.RcvrLock

Downstream transitions to Recovery.RcvrCfg substate as shown in Figure 75

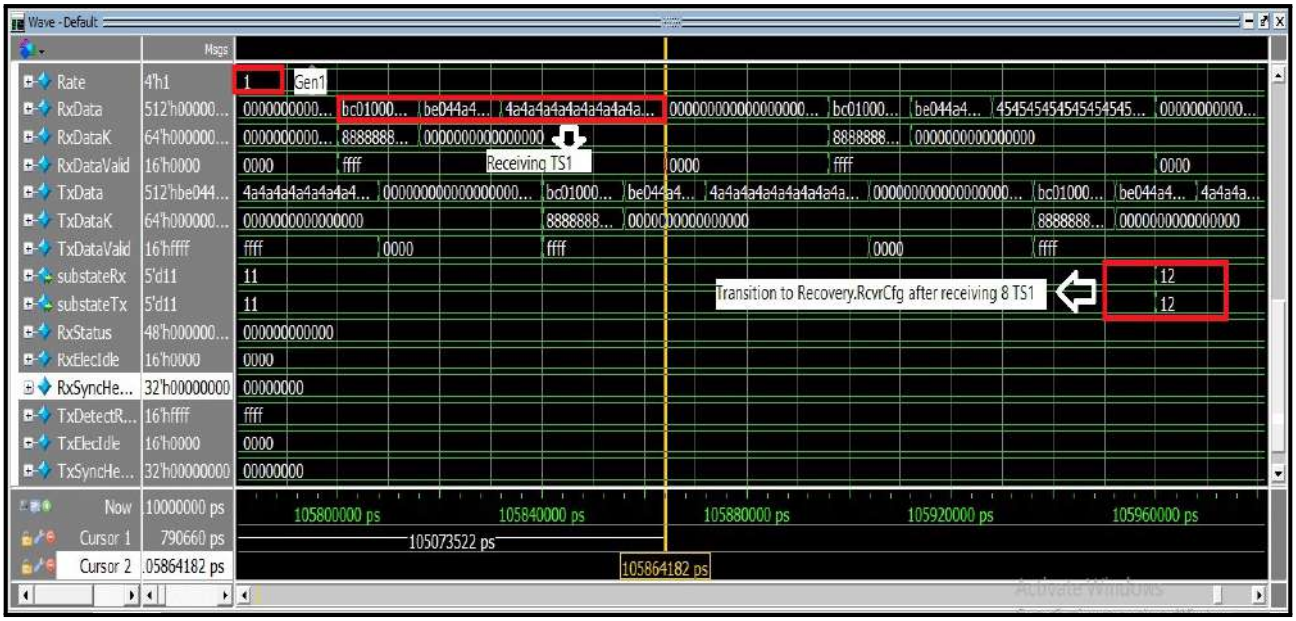


Figure 75: Downstream Receiving TS1 and transitions to Recovery.RcvrCfg

The Upstream is receiving TS1 as shown in Figure 76

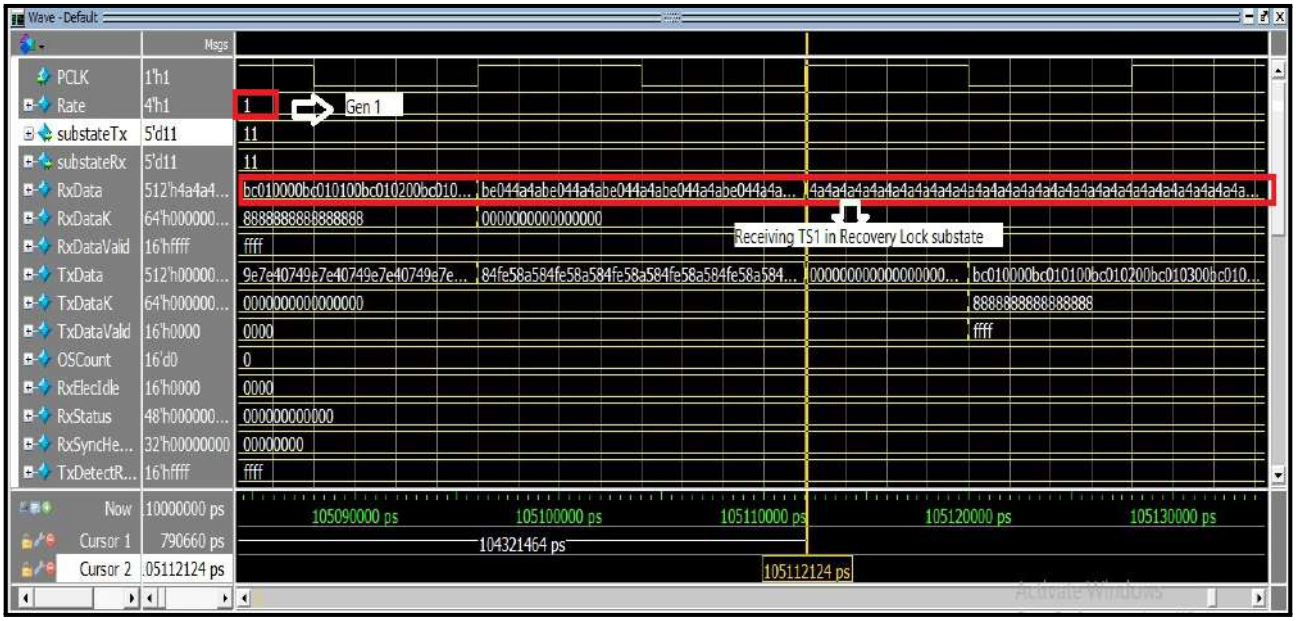


Figure 76: Upstream Receiving TS1 in Recovery.RcvrLock

Upstream transitions to Recovery.RcvrCfg substate as shown in Figure 77

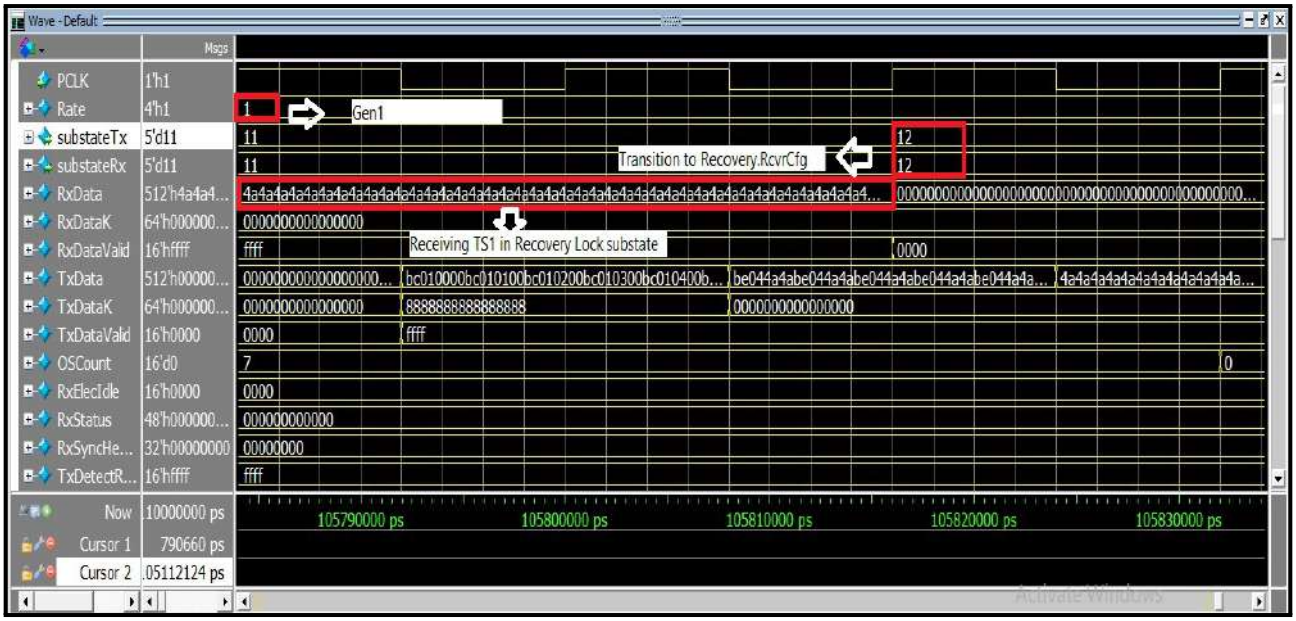


Figure 77: Upstream Receiving TS1 and transitions to Recovery.RcvrCfg

Results from Transcript for TX and RX in Two Devices

Transcript Results for Downstream and Scoreboard Checker for Downstream shown in Figure 78

```
# UVM_INFO E:/(Faculty of Engineering, Alexandria University)/Level 4/Graduation Project/Term 2/Last Env/Grad_BY_Youssef/
TB/TX_Slave_D_Monitor.sv(2009) @ 105992001: uvm_test_top.PCIe_Env_h.TX_Slave_D_Agent_h.TX_Slave_D_Monitor_h [TX_Slave_D_M
onitor] Recovery.RcvrLock substate at Downstream TX side completed successfully
# UVM_INFO E:/(Faculty of Engineering, Alexandria University)/Level 4/Graduation Project/Term 2/Last Env/Grad_BY_Youssef/
TB/RX_Slave_D_Monitor.sv(2348) @ 105992001: uvm_test_top.PCIe_Env_h.RX_Slave_D_Agent_h.RX_Slave_D_Monitor_h [RX_Slave_D_M
onitor]  Recovery RcvrLock substate at Downstraem RX side completed successfully
# UVM_INFO E:/(Faculty of Engineering, Alexandria University)/Level 4/Graduation Project/Term 2/Last Env/Grad_BY_Youssef/
TB/PCIe_Scoreboard1_D.sv(737) @ 105992001: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard1_D_h [PCIe_Scoreboard1_D] Downstream R
ecovery RcvrLock Approved
```

Figure 78: Recovery.RcvrLock in Downstream Completed successfully

Transcript Results for Upstream and Scoreboard Checker for Upstream shown in figure 79

```
# UVM_INFO E:/(Faculty of Engineering, Alexandria University)/Level 4/Graduation Project/Term 2/Last Env/Grad_BY_Youssef/
TB/TX_Slave_U_Monitor.sv(2049) @ 105848001: uvm_test_top.PCIe_Env_h.TX_Slave_U_Agent_h.TX_Slave_U_Monitor_h [TX_Slave_U_M
onitor] Recovery.RcvrLock substate at Upstream TX side completed successfully
# UVM_INFO E:/(Faculty of Engineering, Alexandria University)/Level 4/Graduation Project/Term 2/Last Env/Grad_BY_Youssef/
TB/RX_Slave_U_Monitor.sv(2397) @ 105848001: uvm_test_top.PCIe_Env_h.RX_Slave_U_Agent_h.RX_Slave_U_Monitor_h [RX_Slave_U_M
onitor] Recovery.RcvrLock substate at Upstream RX side completed successfully
# UVM_INFO E:/(Faculty of Engineering, Alexandria University)/Level 4/Graduation Project/Term 2/Last Env/Grad_BY_Youssef/
TB/PCIe_Scoreboard1_U.sv(796) @ 105848001: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard1_U_h [PCIe_Scoreboard1_U] Upstream Rec
overy RcvrLock Approved
```

Figure 79: Recovery.RcvrLock in Upstream Completed successfully

8.12 Recovery.RcvrCfg in Gen1

Introduction

To exit to Recovery.Speed Eight consecutive TS2s are received on any configured Lane with the speed change bit set, identical rate identifiers, identical values in Symbol 6.

Simulation

The Downstream is receiving TS2 as shown in Figure 80

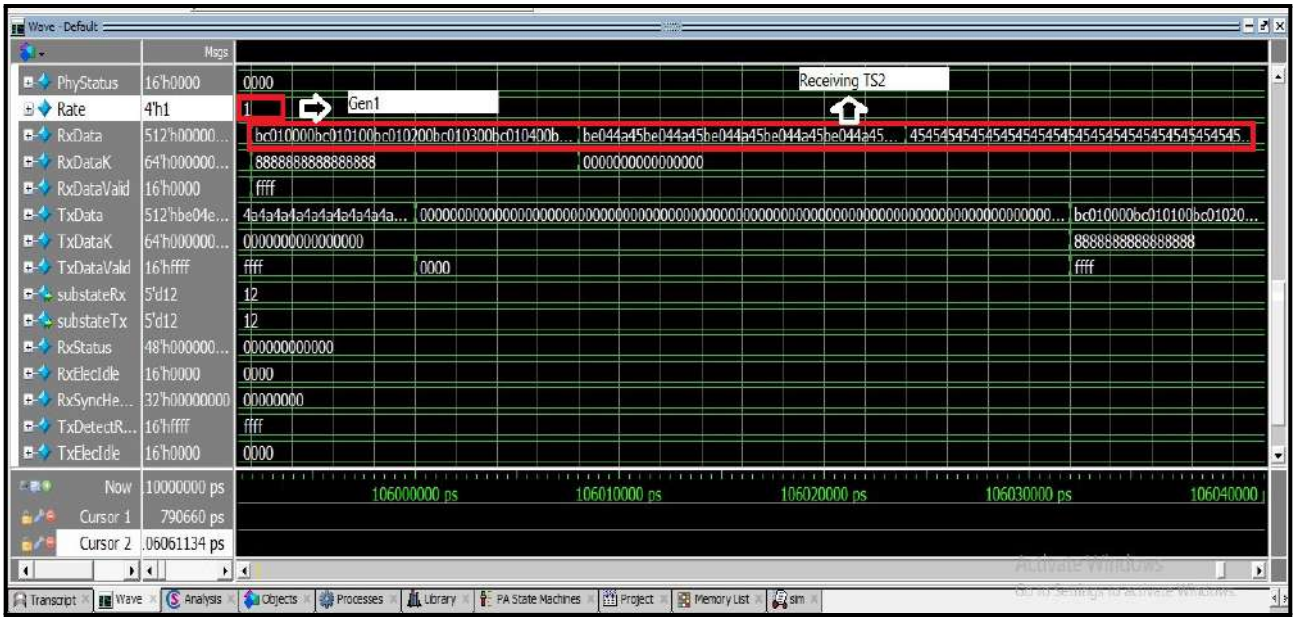


Figure 80: Downstream Receiving TS2 in Recovery.RcvrCfg

After receiving 8 consecutive TS2s and 32 consecutive TS2s Sent on any configured lane Downstream Transitions to Recovery.Speed as shown in Figure 81

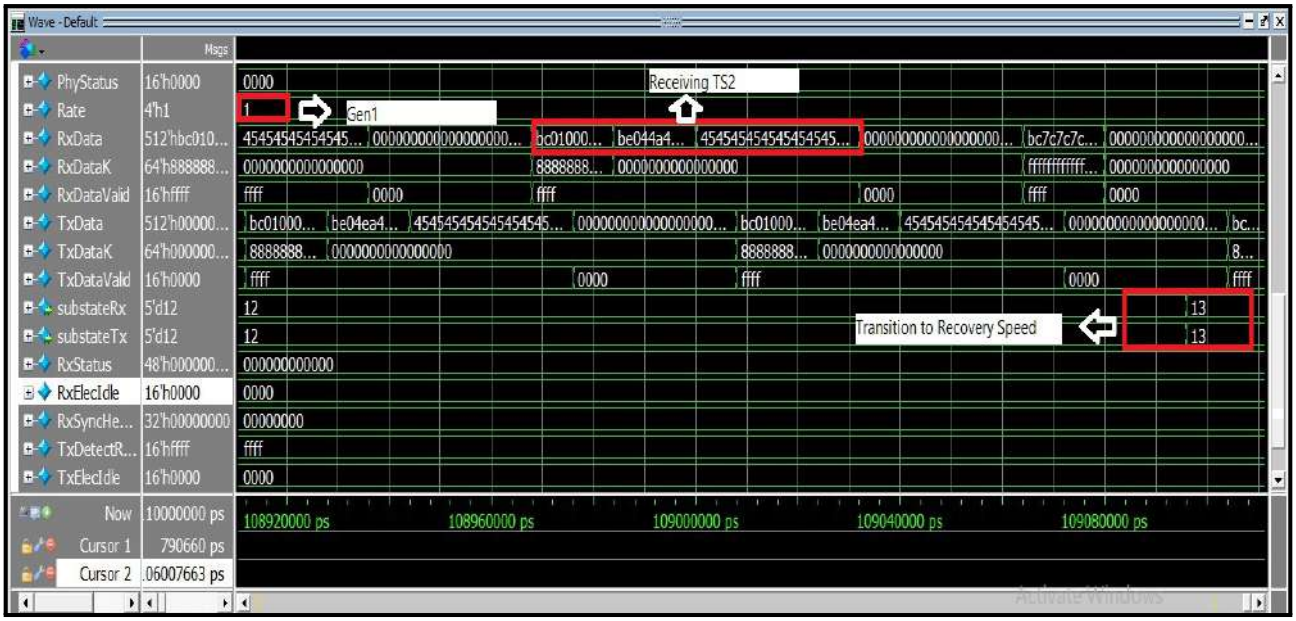


Figure 81: Downstream Receiving 8 TS2s and transitions to Recovery.Speed

The Upstream is receiving TS2 as shown in Figure 82

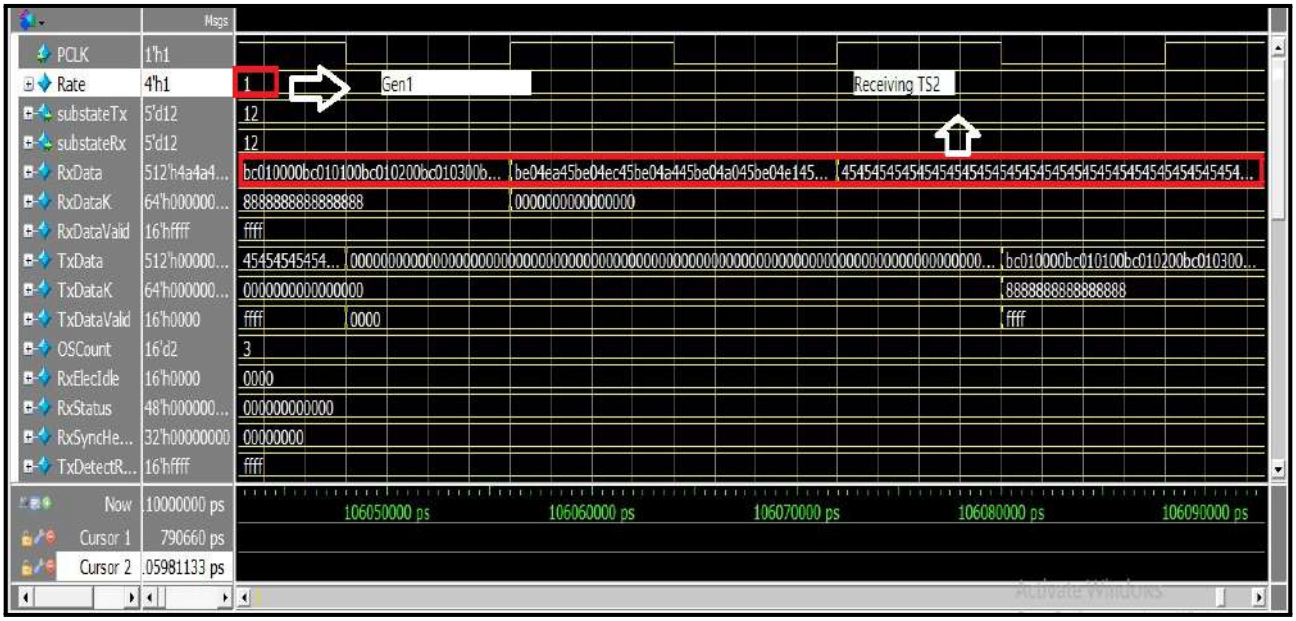


Figure 82: Upstream Receiving TS2 in Recovery.RcvrCfg

After receiving 8 consecutive TS2s and 32 consecutive TS2s Sent on any configured lane Upstream Transitions to Recovery.Speed as shown in Figure 83

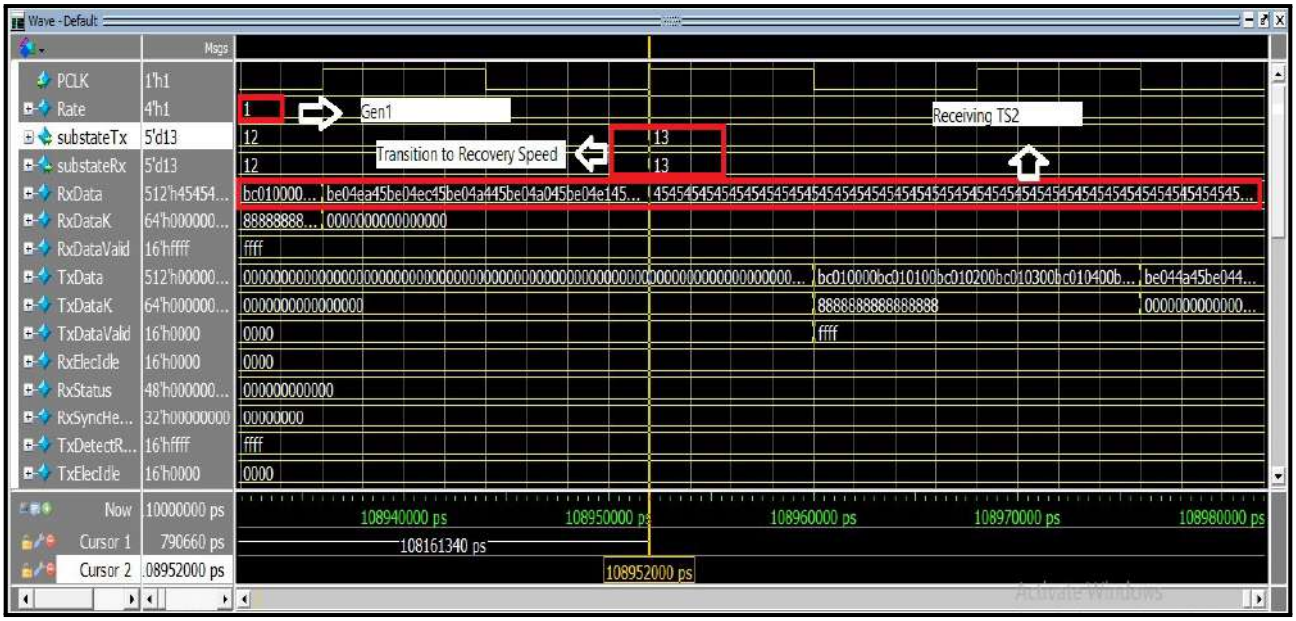


Figure 83: Upstream Receiving 8 TS2s and transitions to Recovery.Speed

Results from Transcript for TX and RX in Two Devices

Transcript Results for Downstream shown in figures 84, 85

```
# UVM_INFO E:/(Faculty of Engineering, Alexandria University)/Level 4/Graduation Project/Term 2/Last Env/Grad_BY_Youssef/
TB/TX_Slave_D_Monitor.sv(2373) @ 109064001: uvm_test_top.PCIe_Env_h.TX_Slave_D_Agent_h.TX_Slave_D_Monitor_h [TX_Slave_D_M
onitor] Recovery.RcvrCfg substate at Downstream TX side completed successfully
# UVM_INFO E:/(Faculty of Engineering, Alexandria University)/Level 4/Graduation Project/Term 2/Last Env/Grad_BY_Youssef/
TB/RX_Slave_D_Monitor.sv(2416) @ 106616001: uvm_test_top.PCIe_Env_h.RX_Slave_D_Agent_h.RX_Slave_D_Monitor_h [RX_Slave_D_M
onitor]  Recovery RcvrCfg  substate at Downstraem RX side completed successfully
```

Figure 84: Recovery.RcvrCfg in Downstream Completed successfully

```
# UVM_INFO E:/(Faculty of Engineering, Alexandria University)/Level 4/Graduation Project/Term 2/Last Env/Grad_BY_Youssef/
TB/PCIe_Scoreboard1_D.sv(740) @ 109064001: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard1_D_h [PCIe_Scoreboard1_D] Downstream R
ecovery RcvrCfg Approved
```

Figure 85: Downstream Scoreboard checker in Recovery.RcvrCfg

Transcript Results for Upstream shown in figures 86, 87

```
# UVM_INFO E:/(Faculty of Engineering, Alexandria University)/Level 4/Graduation Project/Term 2/Last Env/Grad_BY_Youssef/
TB/TX_Slave_U_Monitor.sv(2377) @ 108920001: uvm_test_top.PCIe_Env_h.TX_Slave_U_Agent_h.TX_Slave_U_Monitor_h [TX_Slave_U_M
onitor] Recovery.RcvrCfg substate at Upstream TX side completed successfully
# UVM_INFO E:/(Faculty of Engineering, Alexandria University)/Level 4/Graduation Project/Term 2/Last Env/Grad_BY_Youssef/
TB/RX_Slave_U_Monitor.sv(2475) @ 106760001: uvm_test_top.PCIe_Env_h.RX_Slave_U_Agent_h.RX_Slave_U_Monitor_h [RX_Slave_U_M
onitor] Recovery.RcvrCfg substate at Upstream RX side completed successfully
```

Figure 86: Recovery.RcvrCfg in Upstream Completed successfully

```
# UVM_INFO E:/(Faculty of Engineering, Alexandria University)/Level 4/Graduation Project/Term 2/Last Env/Grad_BY_Youssef/
TB/PCIe_Scoreboard1_U.sv(799) @ 108920001: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard1_U_h [PCIe_Scoreboard1_U] Upstream Rec
overy RcvrCfg Approved
```

Figure 87: Upstream Scoreboard checker in Recovery.RcvrCfg

8.13 Recovery.Speed

Introduction

When entering the **Recovery.Speed** substate, the device must place its Transmitter into *Electrical Idle* and wait for its Receiver to detect Electrical Idle as well. Before entering this substate, an *Electrical Idle Ordered Set (EIOS)* must be transmitted. Once both ends confirm Electrical Idle, the device updates the link speed to the highest commonly supported value.

To exit the **Recovery.Speed** substate, the device starts transmitting *Electrical Idle Exit Ordered Sets (EIEOS)*. As shown in the figure, the device begins by sending *EIOS* at Gen1 speed to enter the Electrical Idle state.

Simulation

The simulation confirms the correct sequence, showing *EIOS* transmission at Gen1, followed by entry into Electrical Idle and successful speed negotiation. As shown in Figure 88, the device begins sending *EIOS* at Gen1 to enter the Electrical Idle state.

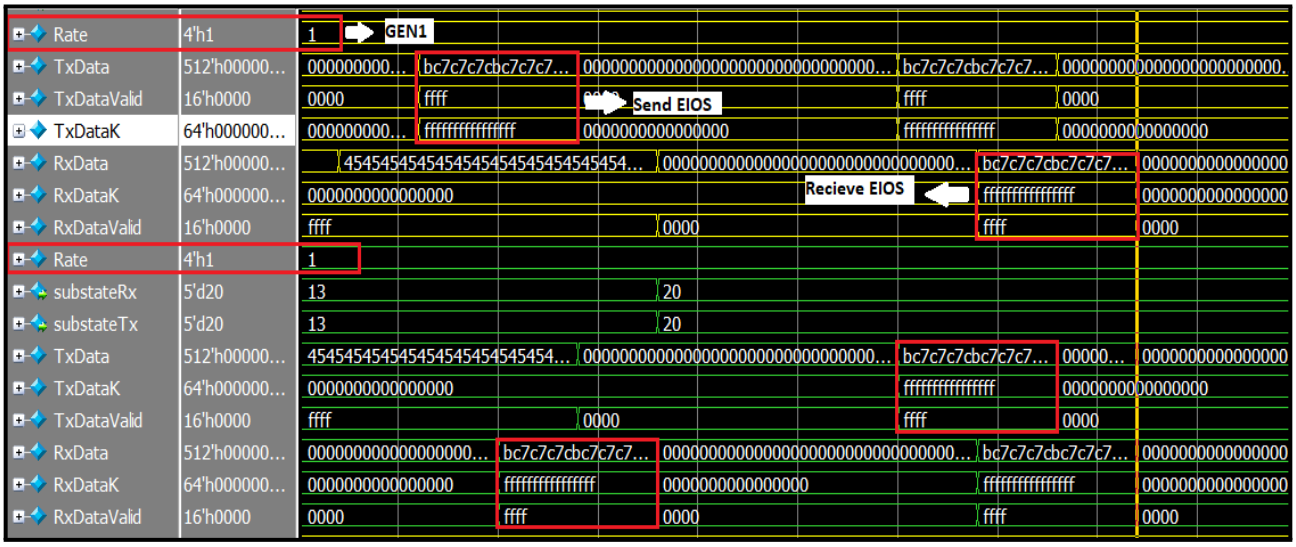


Figure 88: Exchange of EIOS between Devices

Now we wait to see if speed change successfully as shown in Figure 89

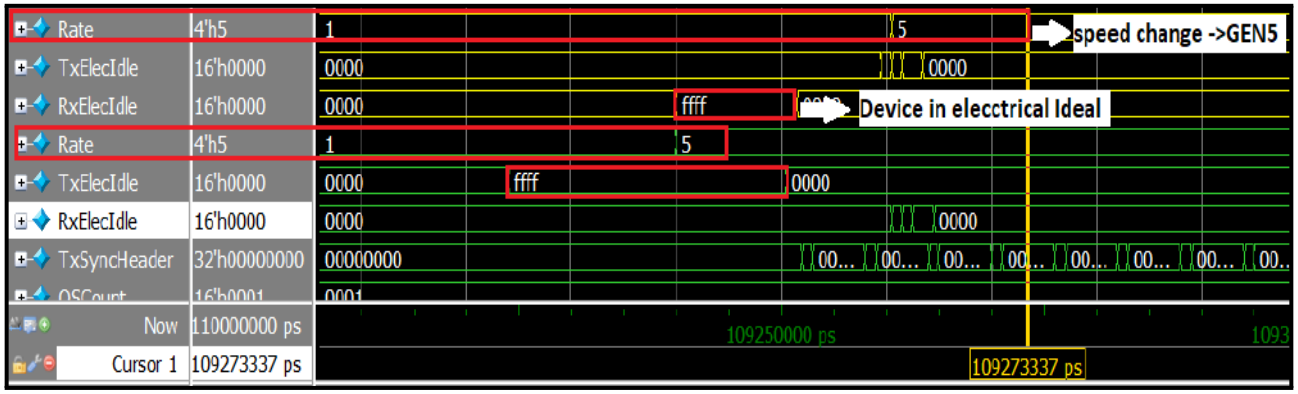


Figure 89: speed change successfully

As shown in the Figure 90, we begin sending EIEOS (Electrical Idle Exit Ordered Set) to exit the Electrical Idle state, using the Sync Header to indicate whether the transmitted byte is data or control.

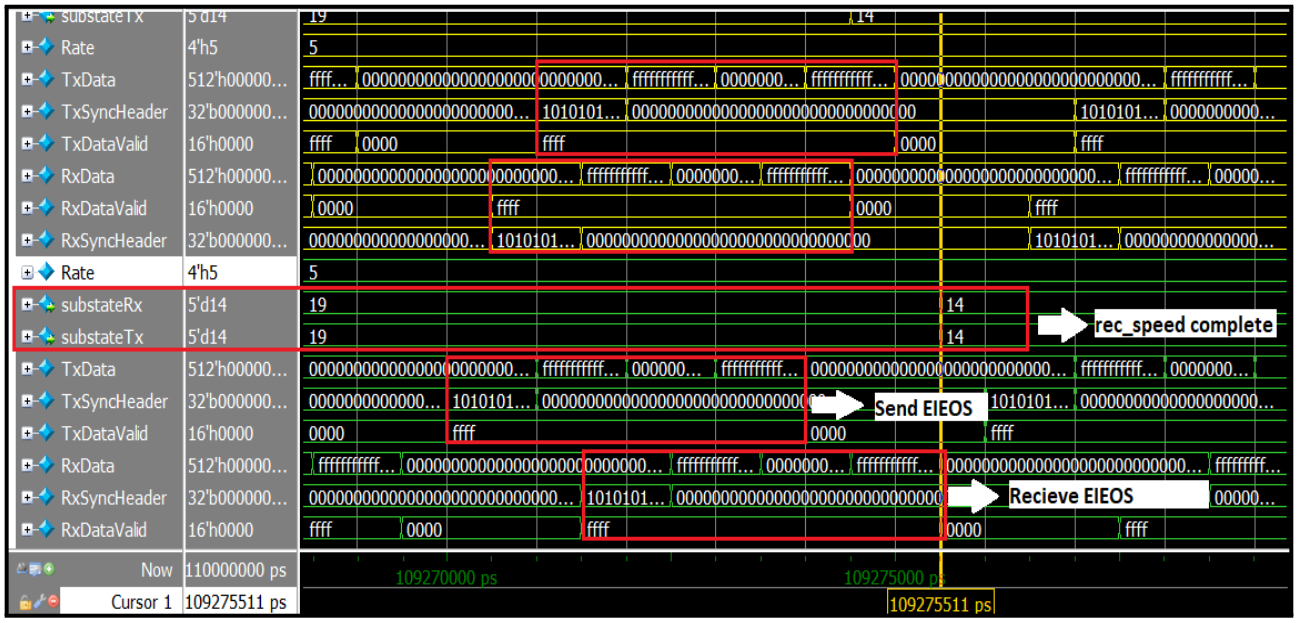


Figure 90: Exchange of EIEOS between Devices

Result from Transcript for TX and RX in Two Devices

As shown in the Figure 91, the **Recovery.Speed** substate completes at approximately 109,275 ns. This closely matches the transcript timing, indicating that the Ordered Set (OS) was successfully transmitted.

Result from Scoreboard

The scoreboard approval As shown in Figure 92 confirms that the number of transmitted Ordered Sets (OS) is correct.

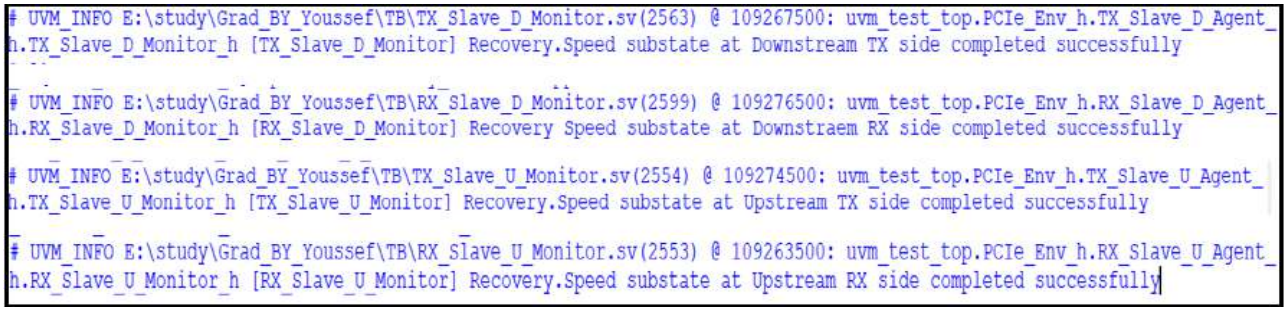


Figure 91: TX and RX Checker for Recovery state

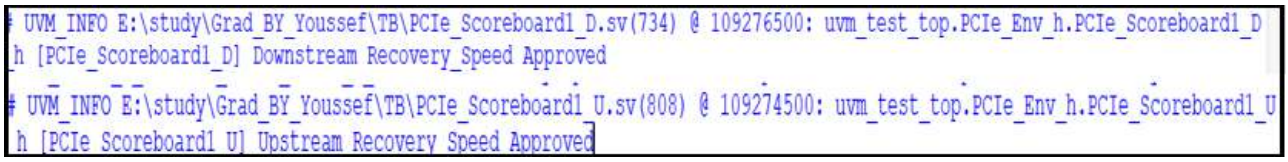


Figure 92: Scoreboard checker for Recovery state

8.14 Recovery.Lock

Introduction

We are now operating in **Gen5** and must establish proper equalization parameters to ensure good signal integrity. To achieve this, the system checks whether the start equalization w preset variable is set to `1'b1`. If this condition is met, the link proceeds to the Equalization states.

Result from Transcript for TX and RX in Two Devices

As shown in the Figure 93, the Recovery.Lock substate completes at approximately 109,277 ns, confirming that the preset variable was correctly set and the transition to the Equalization state occurred as expected.

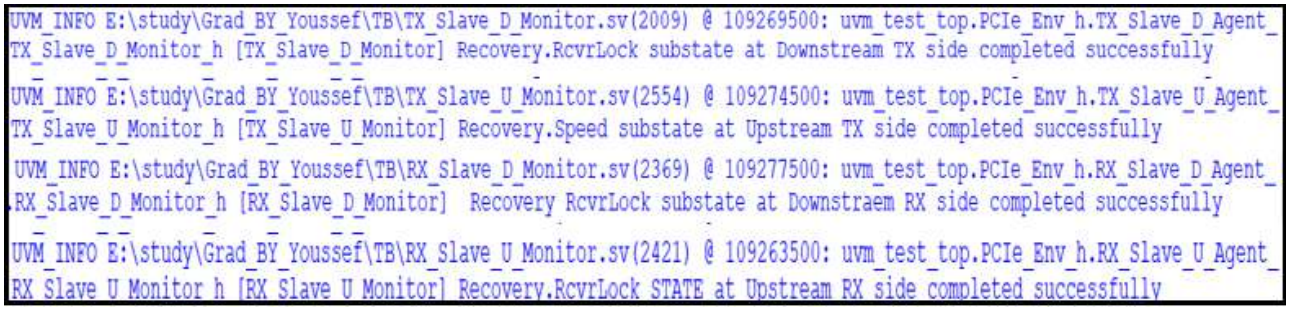


Figure 93: TX and RX Checker for Recovery Lock

8.15 Phase 0

Introduction

When transitioning to 8.0 GT/s, the Downstream Port enters the **Recovery.RcvrCfg** substate and transmits scrambled EQ TS2s containing Tx Presets and Rx Hints to the Upstream Port. These values are derived from the Equalization Control register and may differ for each lane. The Downstream Port uses the DSP values for its own transmitter (and optionally receiver) and sends the USP values to the Upstream Port.

After the link rate is successfully updated, the Downstream Port transitions to Phase 1, where it begins sending scrambled TS1s with EC = 01b. It then waits for the Upstream Port to respond with matching TS1s that also carry EC = 01b.

Simulation

As shown in the Figure 94, the Upstream Port starts transmitting scrambled TS1s.

In our environment, a descrambler is used to capture data from the TxData port, perform descrambling, and verify the EC bits, as illustrated in the Figure 95.

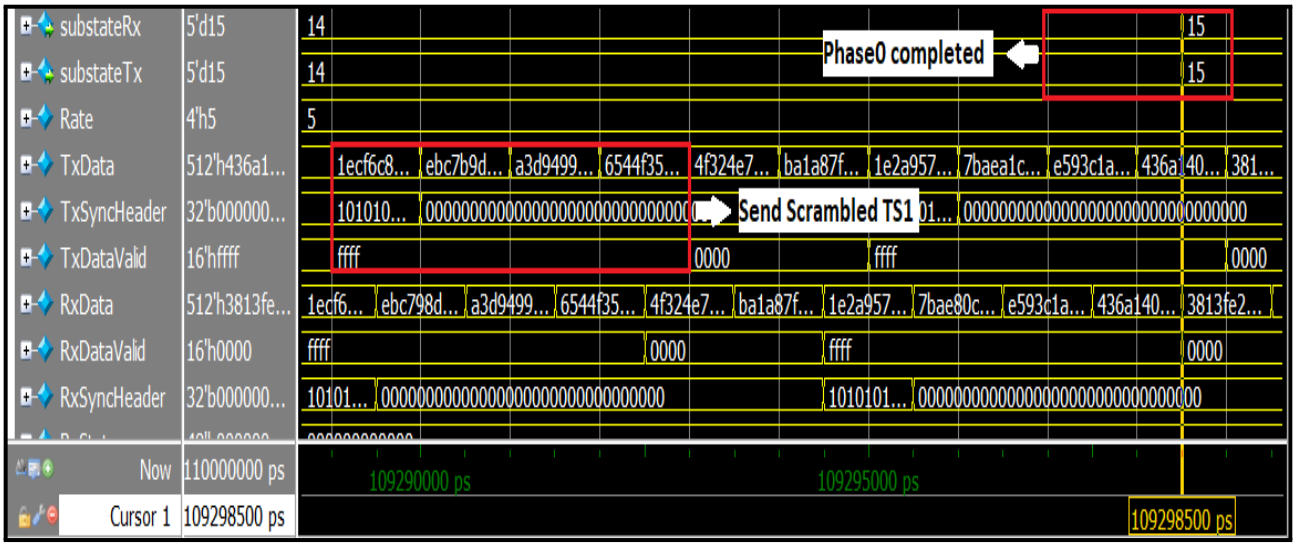


Figure 94: Exchange TS1 between devices in Phase0

```
Lane_Datades = PIPE_vif_h.TxData[i*`MAXPIPEWIDTH +: `MAXPIPEWIDTH];

Descrambled_Data= Descrambled_Data << `MAXPIPEWIDTH ;

Descrambled_Data = apply_descramble(de_scrambler,Lane_Datades,i,`GEN1);
```

Figure 95: Descrambler function

Result from Transcript for TX and RX in Two Devices

As shown in the transcript Figure 96, Phase 0 completed successfully at the Upstream side around 109,295 ns, matching the simulation timing.

```
UVM_INFO E:\study\Grad_BY_Youssef\TB\TX_Slave_U_Monitor.sv(2669) @ 109295000: uvm_test_top.PCIe_Env_h.TX_Slave_U_Agent_
TX_Slave_U_Monitor_h [TX_Slave_U_Monitor] Phase0 substate at Upstream TX side completed successfully
UVM_INFO E:\study\Grad_BY_Youssef\TB\RX_Slave_U_Monitor.sv(2646) @ 109291501: uvm_test_top.PCIe_Env_h.RX_Slave_U_Agent_
RX_Slave_U_Monitor_h [RX_Slave_U_Monitor] Phase0 substate at Upstream RX side completed successfully
```

Figure 96: TX and RX Checker in Phase0

Result from Scoreboard

The scoreboard approval As shown in Figure 97 confirms that the number of scrambled TS1 is correct.

```
UVM_INFO E:\study\Grad_BY_Youssef\TB\PCIe_Scoreboard1_U.sv(802) @ 109295000: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard1_U
 [PCIe_Scoreboard1_U] Upstream Phase0 Approved
```

Figure 97: Scoreboard checker in phase0

8.16 Phase 1

Introduction

The Downstream Port (DSP) achieves a Bit Error Rate (BER) of 10^{-4} by detecting consecutive TS1s. During this phase, it transmits its Tx presets along with Full Swing (FS), Low Frequency (LF), and Post-cursor coefficients. If the DSP receives TS1s with EC = 01b and determines that the link quality is sufficient, it continues with the equalization process. However, if the signal integrity is already deemed adequate, the DSP sets EC = 00b to exit equalization early.

Simulation

The Downstream device begins sending TS1s with EC = 2'b00 As shown in Figure 98 and Upstream Device replay with TS1 likewise As shown in Figure 99 , indicating that signal integrity is sufficient. Consequently, it transitions directly to the

Recovery.Lock state, bypassing Phase 2. The Upstream device, upon receiving TS1s with EC = 2'b00, acknowledges that the Downstream device has determined the link is stable and responds by sending TS1s with the same EC value.

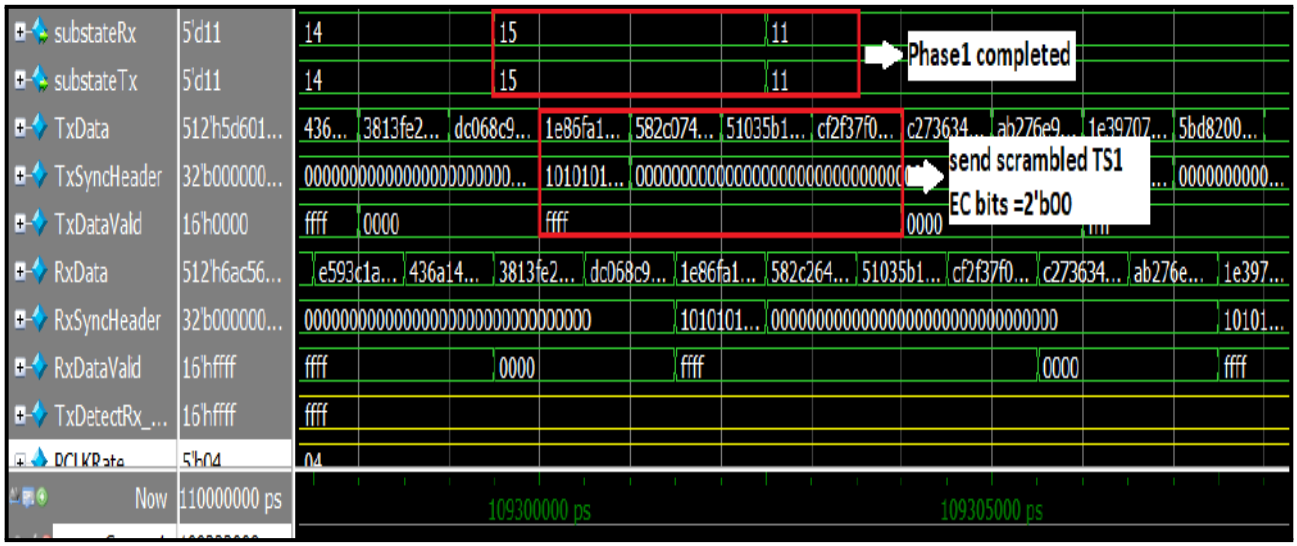


Figure 98: send Scrambled TS1 in Downstream Device

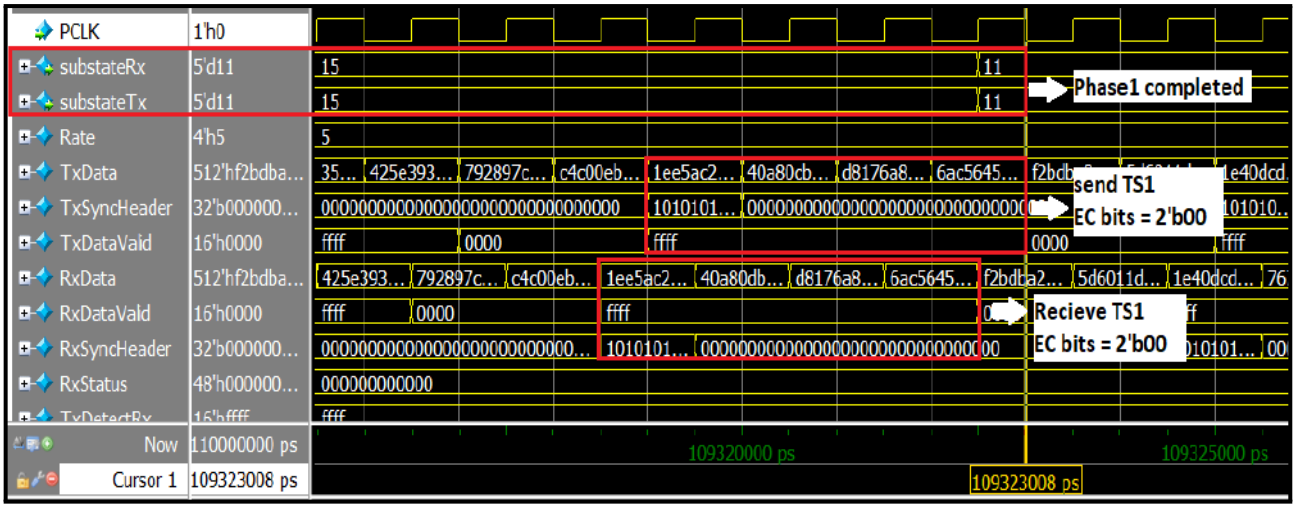


Figure 99: send Scrambled TS1 in Upstream Device

Result from Transcript for TX and RX in Two Devices

As shown in the transcript Figure 100, Phase 1 completed successfully on the Downstream side at approximately 109,294 ns, while the Upstream side completed slightly later at around 109,315 ns. This timing difference confirms proper sequencing and synchronization between both devices.

Result from Scoreboard

As shown in Figure 101 the scoreboard output, the number of scrambled TS1 Ordered Sets transmitted is confirmed to be correct.

```
UVM_INFO E:\study\Grad_BY_Youssef\TB\TX_Slave_D_Monitor.sv(2721) @ 109294000: uvm_test_top.PCIe_Env_h.TX_Slave_D_Agent_
TX_Slave_D_Monitor_h [TX_Slave_D_Monitor] Phase1 substate at Downstream TX side completed successfully
UVM_INFO E:\study\Grad_BY_Youssef\TB\RX_Slave_U_Monitor.sv(2694) @ 109315501: uvm_test_top.PCIe_Env_h.RX_Slave_U_Agent_
RX_Slave_U_Monitor_h [RX_Slave_U_Monitor] Phase1 substate at Upstream RX side completed successfully
UVM_INFO E:\study\Grad_BY_Youssef\TB\TX_Slave_U_Monitor.sv(2805) @ 109325000: uvm_test_top.PCIe_Env_h.TX_Slave_U_Agent_
TX_Slave_U_Monitor_h [TX_Slave_U_Monitor] Phase1 substate at Upstream TX side completed successfully
RX_Slave_D_Monitor_h [RX_Slave_D_Monitor] Phase1 substate at Downstraem RX side completed successfully
UVM_INFO E:\study\Grad_BY_Youssef\TB\PCIe_Scoreboard1_D.sv(743) @ 109298501: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard1_D
```

Figure 100: TX and RX Checker in Phase1

```
UVM_INFO E:\study\Grad_BY_Youssef\TB\PCIe_Scoreboard1_D.sv(743) @ 109298501: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard1_D
 [PCIe_Scoreboard1_D] Downstream Phase1 Approved
UVM_INFO E:\study\Grad_BY_Youssef\TB\PCIe_Scoreboard1_U.sv(805) @ 109325000: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard1_U
 [PCIe_Scoreboard1_U] Upstream Phase1 Approved
```

Figure 101: Scoreboard checker In Phase 1

8.17 Recovery.Lock in Gen5

Introduction

Here, we return to the **Recovery.Lock** state for the third time, now operating at **Gen5** speed. In this phase, both devices exchange scrambled TS1s, transmitting approximately 8 TS1s to transition into the **Recovery.Cfg** state and continue progressing toward the **L0** state.

Simulation

As shown in Figure 102, both devices begin exchanging scrambled TS1s containing the previously configured link and lane numbers. The simulation shows the Ordered Set (OS) count reaching 8, confirming that the required number of TS1s has been successfully exchanged to proceed to the **Recovery.Cfg** state.

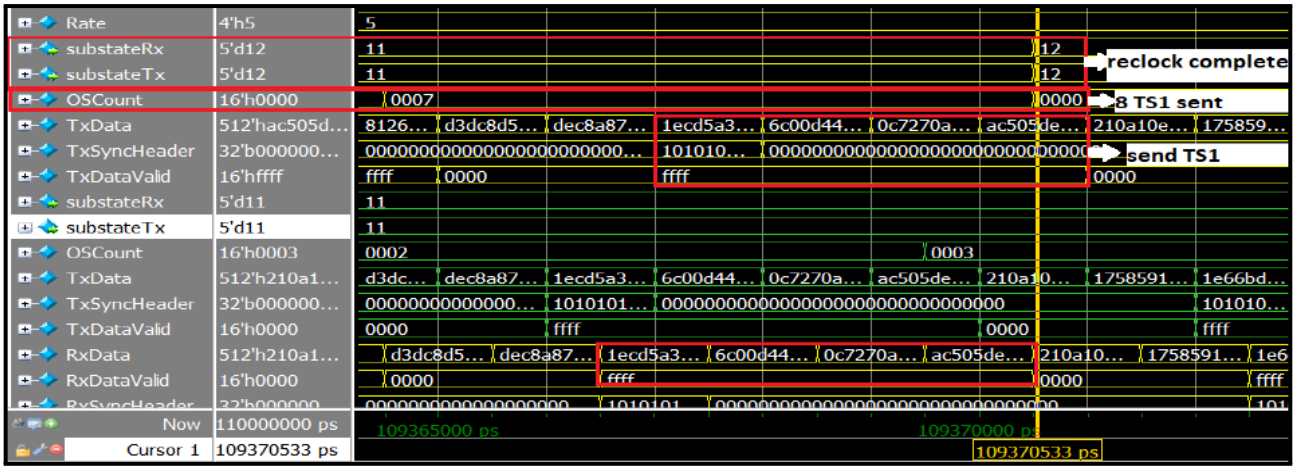


Figure 102: Exchange scrambled TS1 between devices in rec Lock

Result from Transcript for TX and RX in Two Devices

As shown in the transcript Figure 103, **Recovery.Lock** completed successfully at approximately 109,371 ns, aligning with the simulation timing. This confirms a correct transition, with TS1s properly exchanged and containing the same link and lane numbers.

Result from Scoreboard

As shown in Figure 104 the scoreboard output, the successful exchange of 8 scrambled TS1 Ordered Sets between the two devices is confirmed.

```
UVM_INFO E:\study\Grad_BY_Youssef\TB\TX_Slave_U_Monitor.sv(2049) @ 109376501: uvm_test_top.PCIe_Env_h.TX_Slave_U_Agent_
TX_Slave_U_Monitor_h [TX_Slave_U_Monitor] Recovery.RcvrLock substate at Upstream TX side completed successfully
UVM_INFO E:\study\Grad_BY_Youssef\TB\TX_Slave_D_Monitor.sv(2009) @ 109357501: uvm_test_top.PCIe_Env_h.TX_Slave_D_Agent_
TX_Slave_D_Monitor_h [TX_Slave_D_Monitor] Recovery.RcvrLock substate at Downstream TX side completed successfully
UVM_INFO E:\study\Grad_BY_Youssef\TB\RX_Slave_U_Monitor.sv(2397) @ 109363501: uvm_test_top.PCIe_Env_h.RX_Slave_U_Agent_
RX_Slave_U_Monitor_h [RX_Slave_U_Monitor] Recovery.RcvrLock substate at Upstream RX side completed successfully
UVM_INFO E:\study\Grad_BY_Youssef\TB\RX_Slave_D_Monitor.sv(2348) @ 109371500: uvm_test_top.PCIe_Env_h.RX_Slave_D_Agent_
RX_Slave_D_Monitor_h [RX_Slave_D_Monitor]  Recovery RcvrLock substate at Downstraem RX side completed successfully
```

Figure 103: TX and RX Checker in Reclock GEN5

```
UVM_INFO E:\study\Grad_BY_Youssef\TB\PCIe_Scoreboard1_D.sv(737) @ 109371500: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard1_D
 [PCIe_Scoreboard1_D] Downstream Recovery_RcvrLock Approved
UVM_INFO E:\study\Grad_BY_Youssef\TB\PCIe_Scoreboard1_U.sv(796) @ 109376501: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard1_U
 [PCIe_Scoreboard1_U] Upstream Recovery_RcvrLock Approved
```

Figure 104: Scoreboard checker in Reclock GEN5

8.18 Recovery.Cfg in Gen5

Introduction

At this stage, bit, symbol, or block lock has been successfully established. Since Recovery was entered solely to re-establish lock after exiting a power management state, the ports now exchange 16 TS2 Ordered Sets and then transition to the **Recovery.Idle** state.

Simulation

As shown in Figure 105, both devices begin exchanging TS2s with matching link and lane numbers. The speed

change bit is set to 0, indicating that no further speed transition is required. After at least 16 TS2s are sent and received, the link proceeds to the **Recovery.Idle** state.

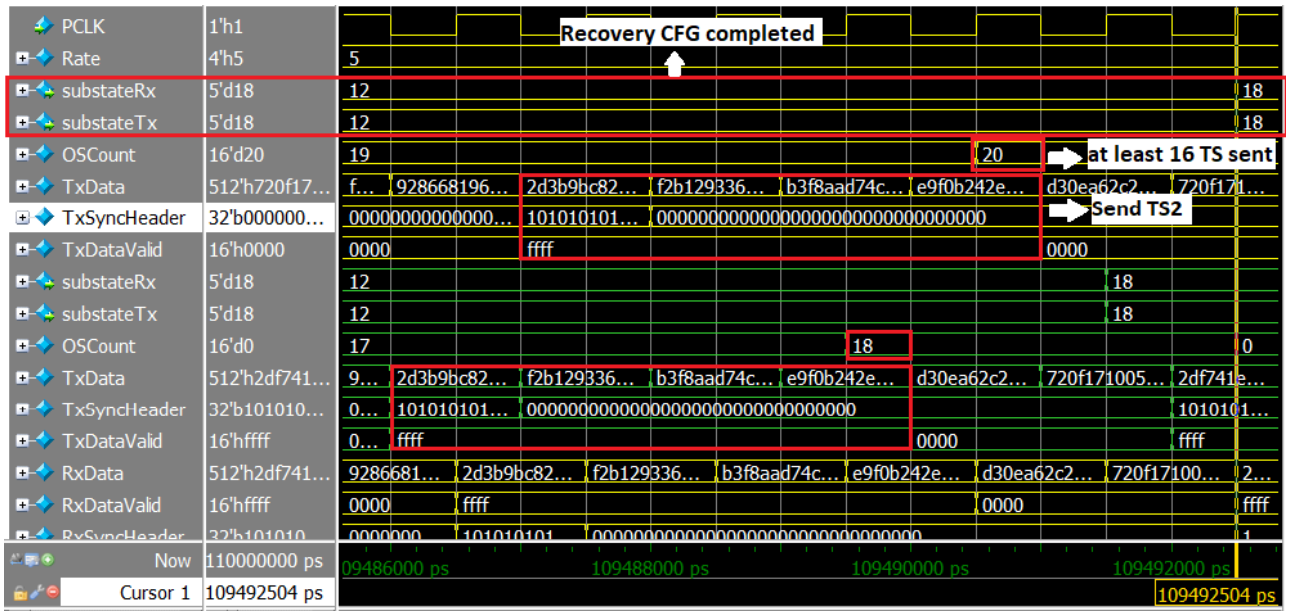


Figure 105: send scrambled TS2 in Rec CFG

Result from Transcript for TX and RX in Two Devices

As shown in the transcript Figure 106, **Recovery.Cfg** completed successfully at approximately 109,424 ns, closely matching the simulation timing. This confirms a correct transition with properly exchanged TS2s containing consistent link and lane numbers and a deasserted speed change bit.

```
UVM_INFO E:\study\Grad_BY_Youssef\TB\TX_Slave_U_Monitor.sv(2377) @ 109496501: uvm_test_top.PCIe_Env_h.TX_Slave_U_Agent_
TX_Slave_U_Monitor_h [TX_Slave_U_Monitor] Recovery.RcvrCfg substate at Upstream TX side completed successfully
UVM_INFO E:\study\Grad_BY_Youssef\TB\RX_Slave_D_Monitor.sv(2441) @ 109424501: uvm_test_top.PCIe_Env_h.RX_Slave_D_Agent_
.RX_Slave_D_Monitor_h [RX_Slave_D_Monitor]  Recovery RcvrCfg  substate at Downstraem RX side completed successfully
UVM_INFO E:\study\Grad_BY_Youssef\TB\RX_Slave_U_Monitor.sv(2500) @ 109441501: uvm_test_top.PCIe_Env_h.RX_Slave_U_Agent_
.RX_Slave_U_Monitor_h [RX_Slave_U_Monitor] Recovery.RcvrCfg substate at Upstream RX side completed successfully
UVM_INFO E:\study\Grad_BY_Youssef\TB\TX_Slave_D_Monitor.sv(2373) @ 109489501: uvm_test_top.PCIe_Env_h.TX_Slave_D_Agent_
.TX_Slave_D_Monitor_h [TX_Slave_D_Monitor] Recovery.RcvrCfg substate at Downstream TX side completed successfully
```

Figure 106: TX and RX Checker in Rec CFG GEN5

Result from Scoreboard

As shown in Figure 107 the scoreboard output and, the successful exchange of at least 16 TS2 Ordered Sets between the two devices is confirmed.

```
UVM_INFO E:\study\Grad_BY_Youssef\TB\PCIe_Scoreboard1_D.sv(740) @ 109489501: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard1_D
 [PCIe_Scoreboard1_D] Downstream Recovery RcvrCfg Approved
UVM_INFO E:\study\Grad_BY_Youssef\TB\PCIe_Scoreboard1_U.sv(799) @ 109496501: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard1_U
 [PCIe_Scoreboard1_U] Upstream Recovery RcvrCfg Approved
```

Figure 107: Scoreboard checker in Rec CFG GEN5

8.19 Recovery.Idle

Introduction

In this substate, transmitters typically send *Idle* symbols to prepare for transitioning to the fully operational **L0** state. For **128b/130b** encoding, the sequence begins with the transmission of a *SKP Ordered Set* to initiate the data stream, followed by an *SDS Ordered Set*, and then approximately 8 *Idle* data symbols on all lanes.

Simulation

As shown in Figure 108, the transition begins with the transmission of the *SKP Ordered Set*, followed by the *SDS* symbol as shown in Figure 109. Afterward, the *idle* symbols are sent to complete the transition to the **L0** state As shown in Figure 110. At this point, the Physical Layer to Link Layer interface enters the **Active** state.

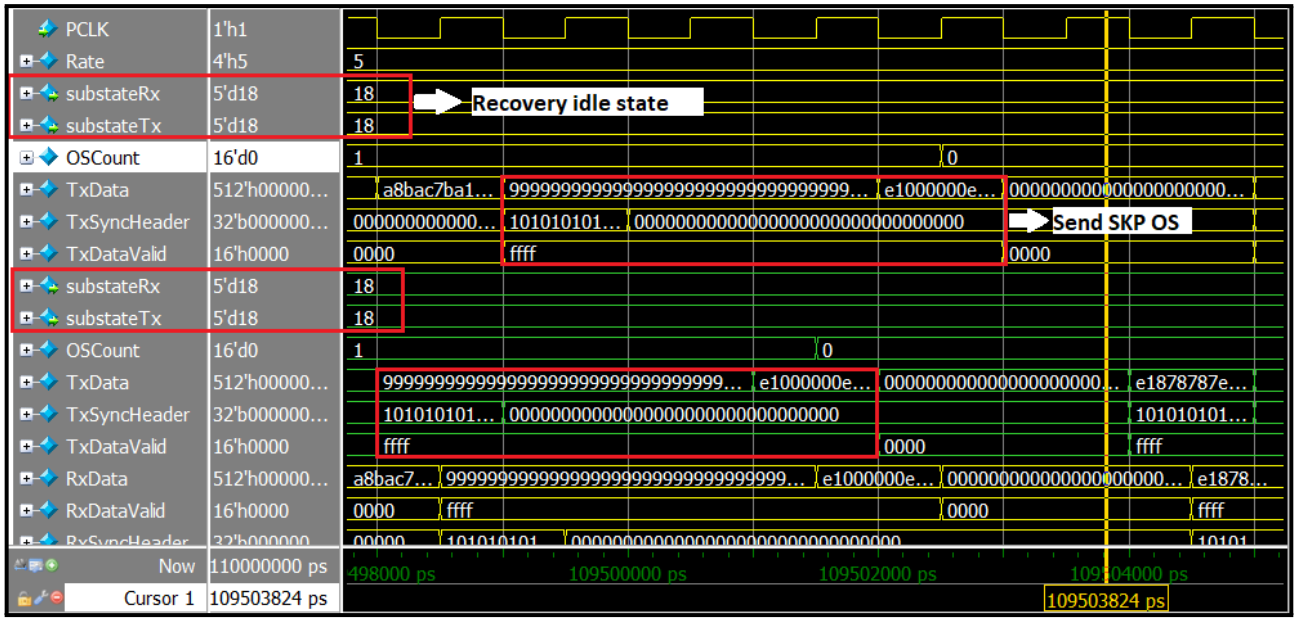


Figure 108: Send SKP OS

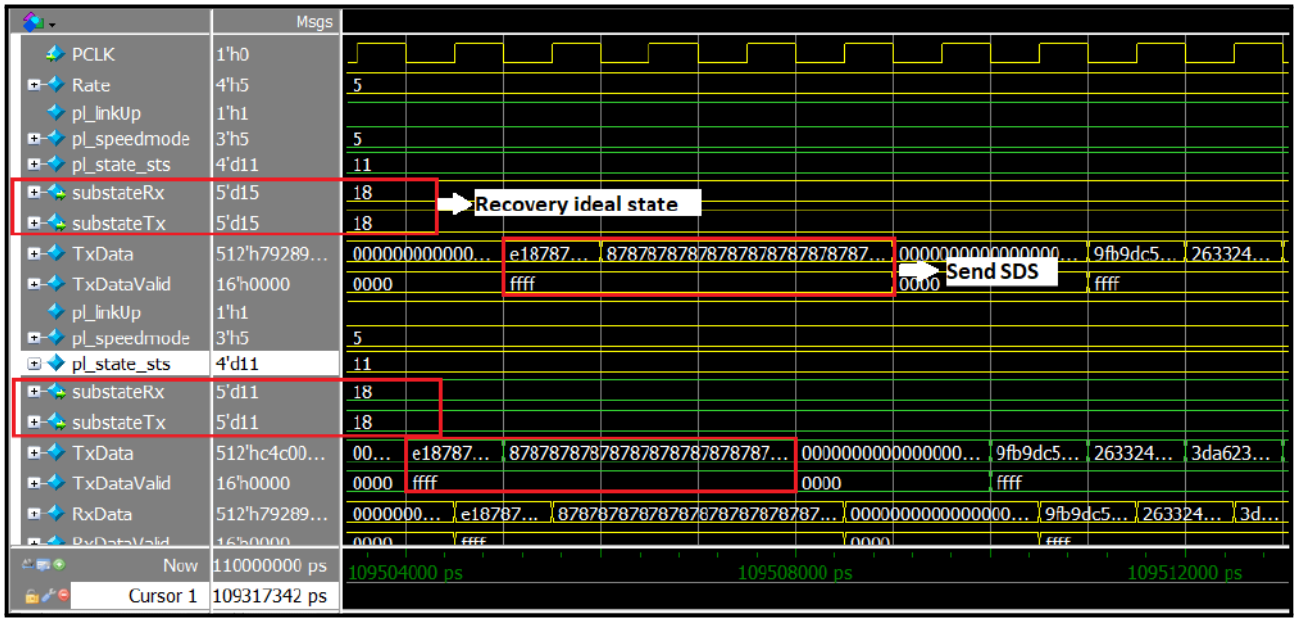


Figure 109: Send SDS OS

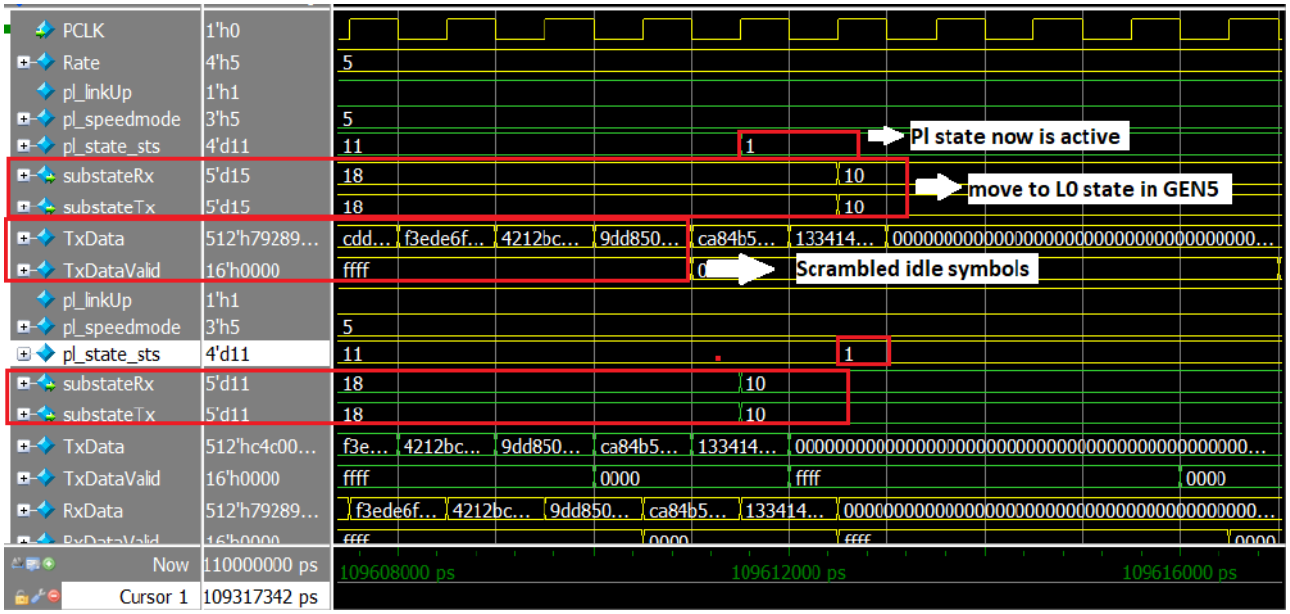


Figure 110: Send Idle symbols

Result from Transcript for TX and RX in Two Devices

As shown in the transcript Figure 111, **Recovery.Idle** completed successfully at approximately 109,603 ns, closely matching the simulation timing. This confirms a correct transition with the proper exchange of *SKP*, *SDS*, and *Idle* symbols.

```
UVM_INFO E:\study\Grad_BY_Youssef\TB\TX_Slave_D_Monitor.sv(2975) @ 109603500: uvm_test_top.PCIe_Env_h.TX_Slave_D_Agent_
TX_Slave_D_Monitor_h [TX_Slave_D_Monitor] Recovery.Idle substate at Downstream TX side completed successfully
UVM_INFO E:\study\Grad_BY_Youssef\TB\RX_Slave_D_Monitor.sv(2639) @ 109603500: uvm_test_top.PCIe_Env_h.RX_Slave_D_Agent_
RX_Slave_D_Monitor_h [RX_Slave_D_Monitor] Recovery Idle substate at Downstraem RX side completed successfully
UVM_INFO E:\study\Grad_BY_Youssef\TB\TX_Slave_U_Monitor.sv(3051) @ 109604500: uvm_test_top.PCIe_Env_h.TX_Slave_U_Agent_
TX_Slave_U_Monitor_h [TX_Slave_U_Monitor] Recovery.Idle substate at Upstream TX side completed successfully
UVM_INFO E:\study\Grad_BY_Youssef\TB\RX_Slave_U_Monitor.sv(2594) @ 109510500: uvm_test_top.PCIe_Env_h.RX_Slave_U_Agent_
RX_Slave_U_Monitor_h [RX_Slave_U_Monitor] Recovery.Idle substate at Upstream RX side completed successfully
```

Figure 111: TX and RX Checker in rec idle

Result from Scoreboard

The scoreboard approval shown in Figure 112,confirms that at least 8 *Idle* symbols were exchanged between the two devices.

```
UVM_INFO E:\study\Grad_BY_Youssef\TB\PCIe_Scoreboard1_U.sv(811) @ 109510500: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard1_U
 [PCIe_Scoreboard1_U] Upstream Recovery_Idle Approved
UVM_INFO E:\study\Grad_BY_Youssef\TB\PCIe_Scoreboard1_D.sv(746) @ 109510500: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard1_D
 [PCIe_Scoreboard1_D] Downstream Recovery_Idle Approved
```

Figure 112: Scoreboard checker in rec idle

9 Verification of Packets Transmission and Reception

At this section, we verify the transmission and reception of TLPs and DLLPs between downstream and upstream devices, after both have entered the L0 full operational state, across the following scenarios:

- Transmission and reception of TLPs.
- Transmission and reception of DLLPs.
- Transmission and reception of both TLPs and DLLPs.

Note: The link to the physical interface of the downstream device is colored green, while the link to the physical interface of the upstream device is colored yellow.

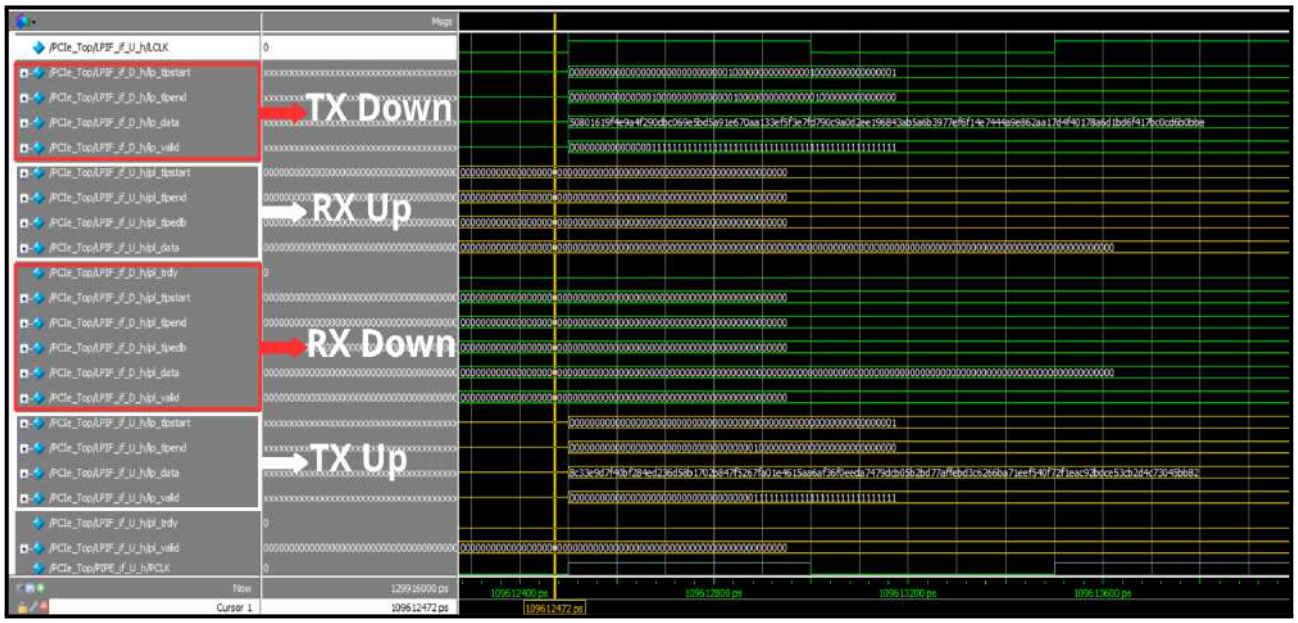


Figure 113: Downstream and upstream decives interfaces

9.1 Transmission and Reception of TLPs in one transfer

In this part, we transmit and receive TLPs from downstream to upstream and from upstream to downstream.

Figure 114: TLPs Sequences start running

9.1.1 Transmission of TLPs in one transfer from downstream device to upstream device

In this example shown in the simulation results includes the transmission of three back-to-back TLPs, as indicated by the `lp_tlpstart` and `lp_tlpend` signals. The data on `lp_data` is validated by the `lp_valid` signal.

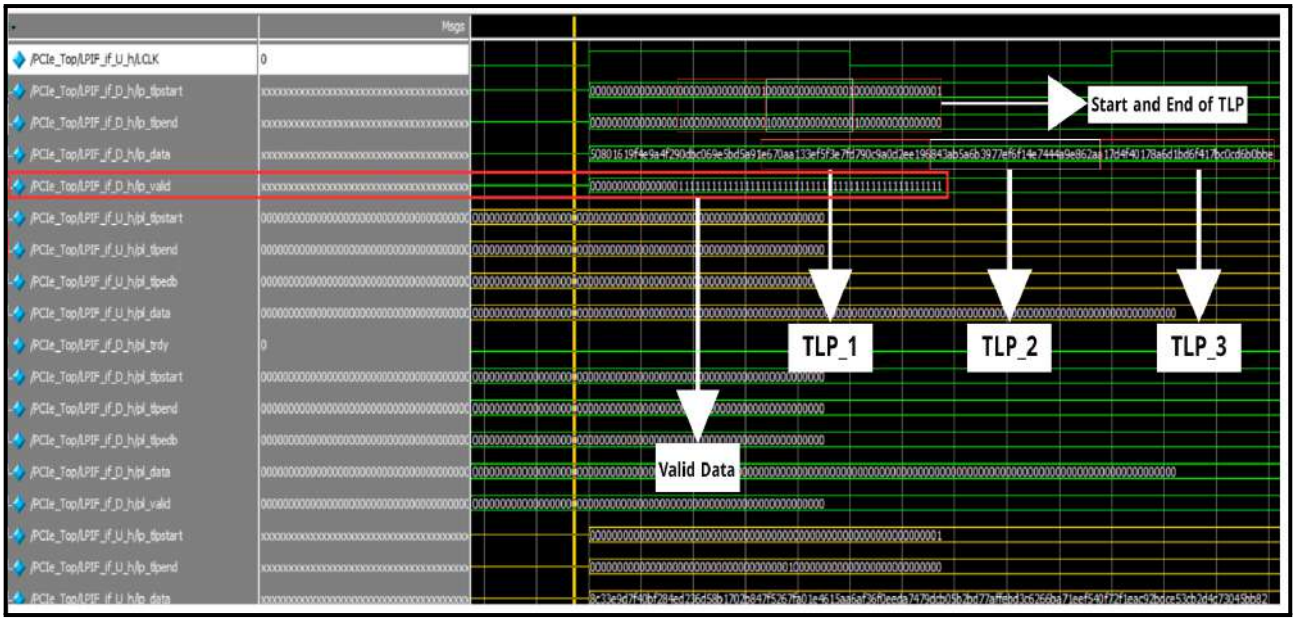


Figure 115: Transmission of three TLPs

9.1.2 Reception of TLPs in one transfer from downstream device at upstream device

This simulation results show the reception of three back-to-back TLPs, where the start and end of each TLP are marked by the `pl_tlpstart` and `pl_tlpend` signals. The `pl_valid` signal validates the data on the `pl_data` signal.

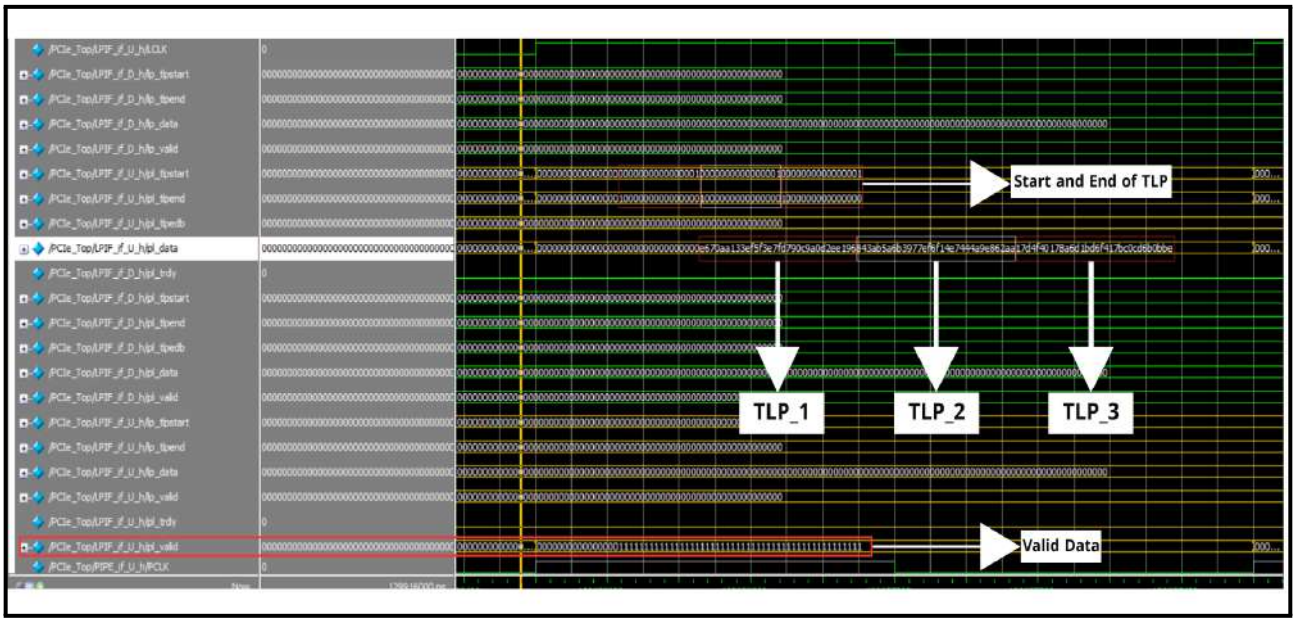


Figure 116: Reception of three TLPs

9.1.3 Transmission of TLPs from upstream device to downstream device

the example shown in the simulation results includes the transmission of one TLP of maximum size, as indicated by the `lp_tlpstart` and `lp_tlpend` signals. The data on `lp_data` is validated by the `lp_valid` signal.

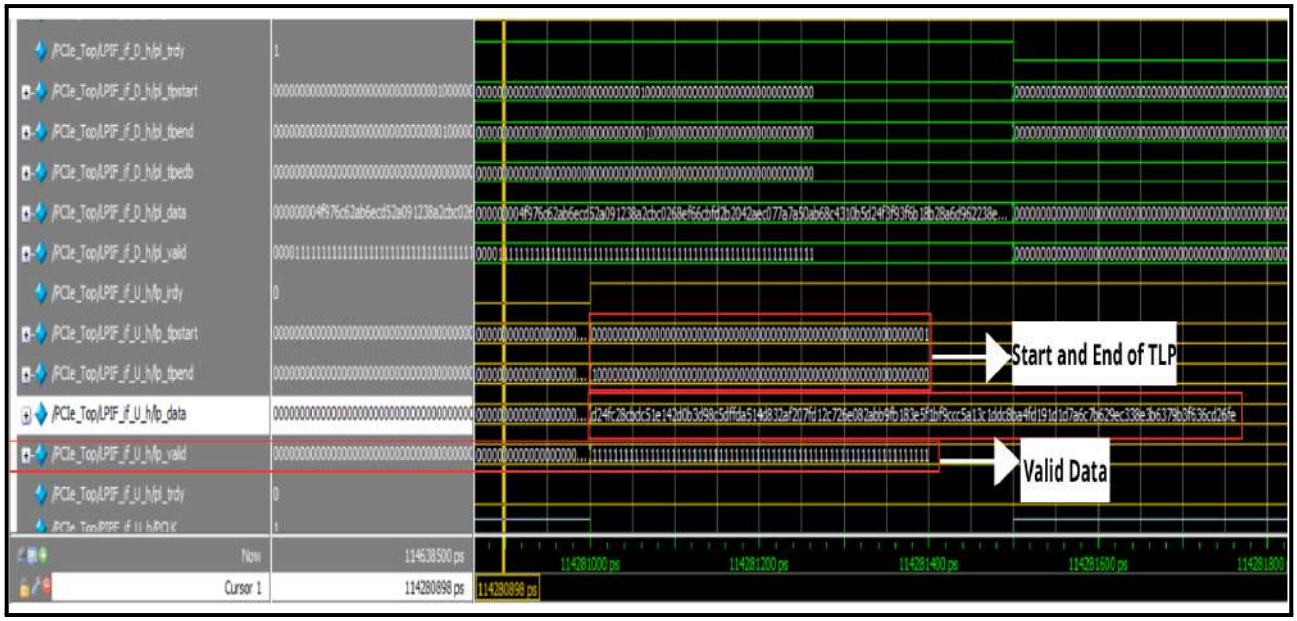


Figure 117: Transmission of three TLPs

9.1.4 Reception of TLPs from upstream device to downstream device

The output results show the reception of one TLP of maximum size, where the start and end of the TLP are marked by the `pl_tlpstart` and `pl_tlpend` signals. The `pl_valid` signal validates the data on the `pl_data` signal.

When the TX transmits a TLP packet, it appends an STP token of 1 DW to the 15 DWs of data, since the width of `lp_data` is 16 DWs (512 bits). The entire packet cannot be transmitted at once, so the first 15 DWs are sent in the initial transfer, and the remaining 1 DW is transmitted in the next transfer.

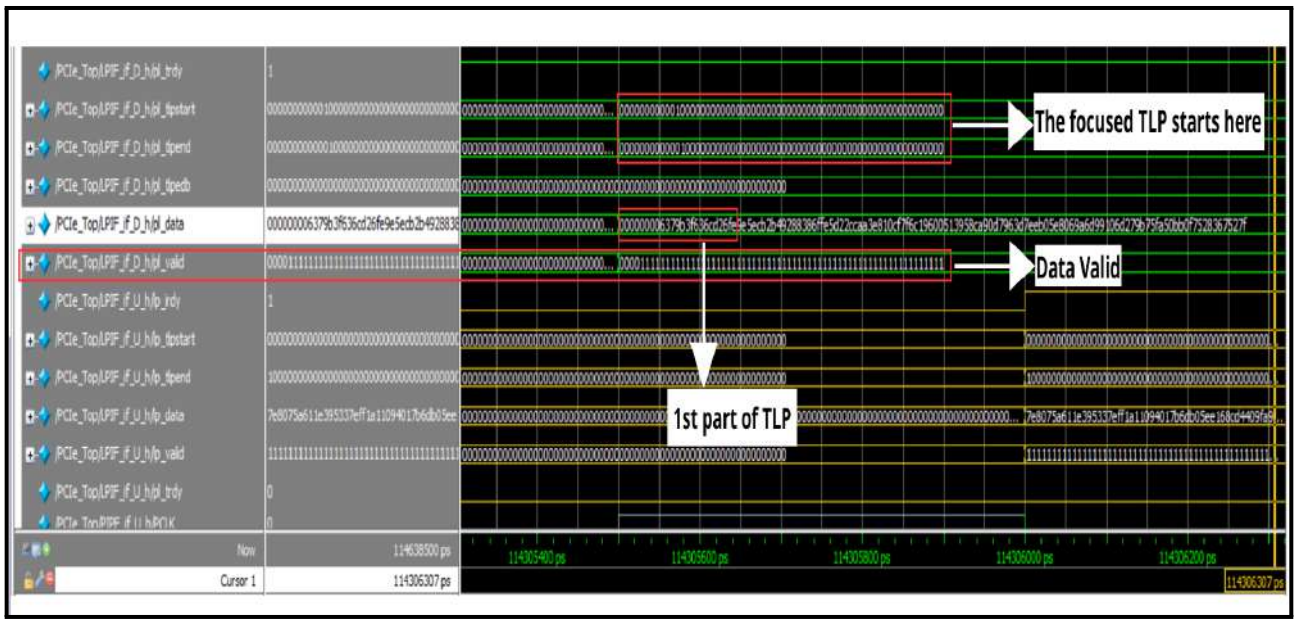


Figure 118: Reception the 1st part of TLP

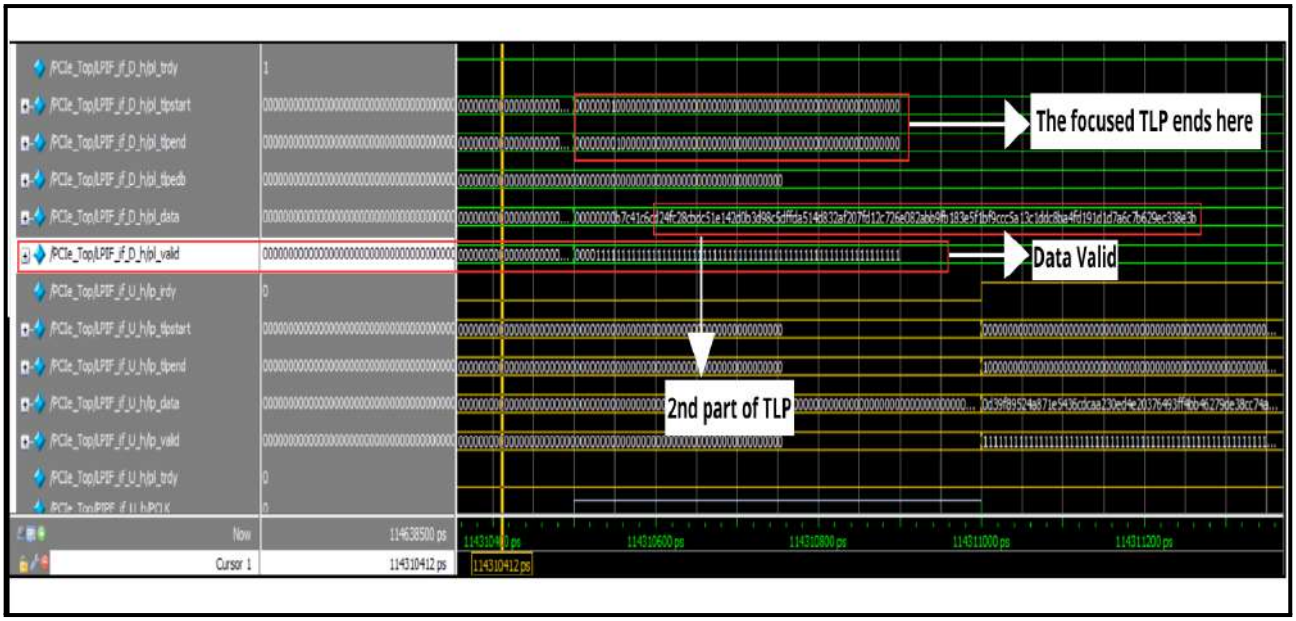


Figure 119: Reception the 2nd part of TLP

9.2 Transmission and Reception of DLLPs

In this part, we transmit and receive DLLPs from downstream to upstream and from upstream to downstream.

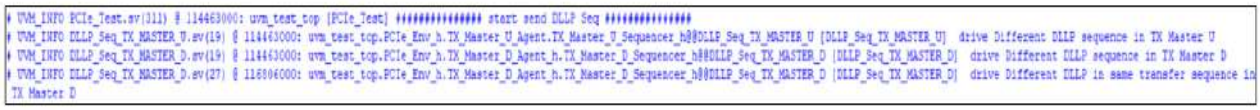


Figure 120: DLLPs Sequences start running

9.2.1 Transmission of DLLPs from downstream device to upstream device

In this example shown in the simulation results includes the transmission of three back-to-back DLLPs, as indicated by the `lp_dllpstart` and `lp_dllpend` signals. The data on `lp_data` is validated by the `lp_valid` signal.

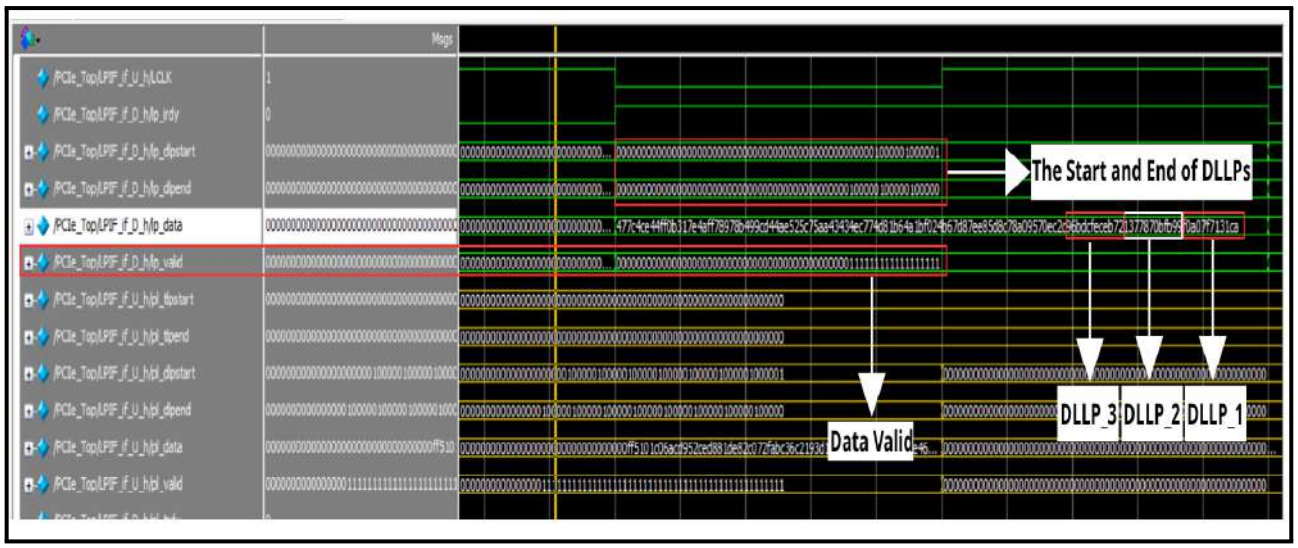


Figure 121: Transmission of three DLLPs

9.2.2 Reception of DLLPs from downstream device at upstream device

This simulation results show the reception of three back-to-back DLLPs, where the start and end of each DLLP are marked by the `pl_dllpstart` and `pl_dllpend` signals. The `pl_valid` signal validates the data on the `pl_data` signal.

The TX buffers the accumulated data and sends it when the buffer is full — in other words, when 16 DWs (512 bits) are filled with data.

In this example, 6 DLLPs from the previous transfer were not transmitted, so they are sent along with the 3 DLLPs of the current transfer. However, each DLLP requires a 1 W SDP token at its start. Since each DLLP occupies 2 DWs (1 W for the SDP and 3 W for the payload), only 8 DLLPs can be transmitted in a single transfer.

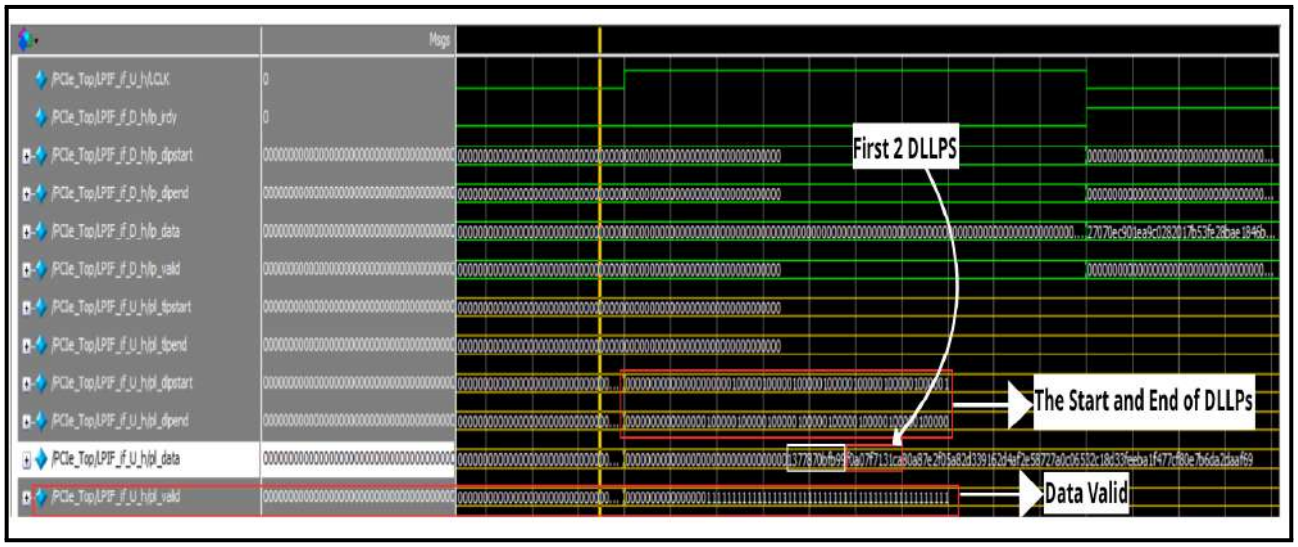


Figure 122: Reception the 1st part of three DLLPs

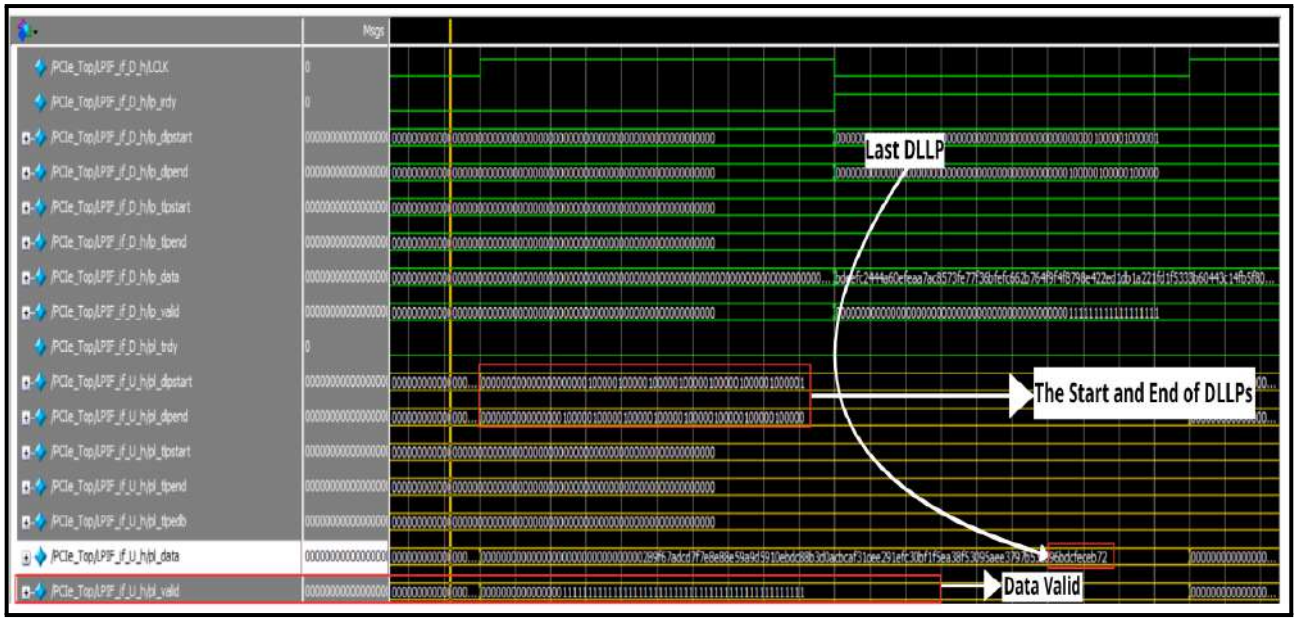


Figure 123: Reception the 2nd part of three DLLPs

9.2.3 Transmission of DLLPs from upstream device to downstream device

the example shown in the simulation results includes the transmission of two DLLPs, as indicated by the `lp_dllpstart` and `lp_dllpend` signals. The data on `lp_data` is validated by the `lp_valid` signal.

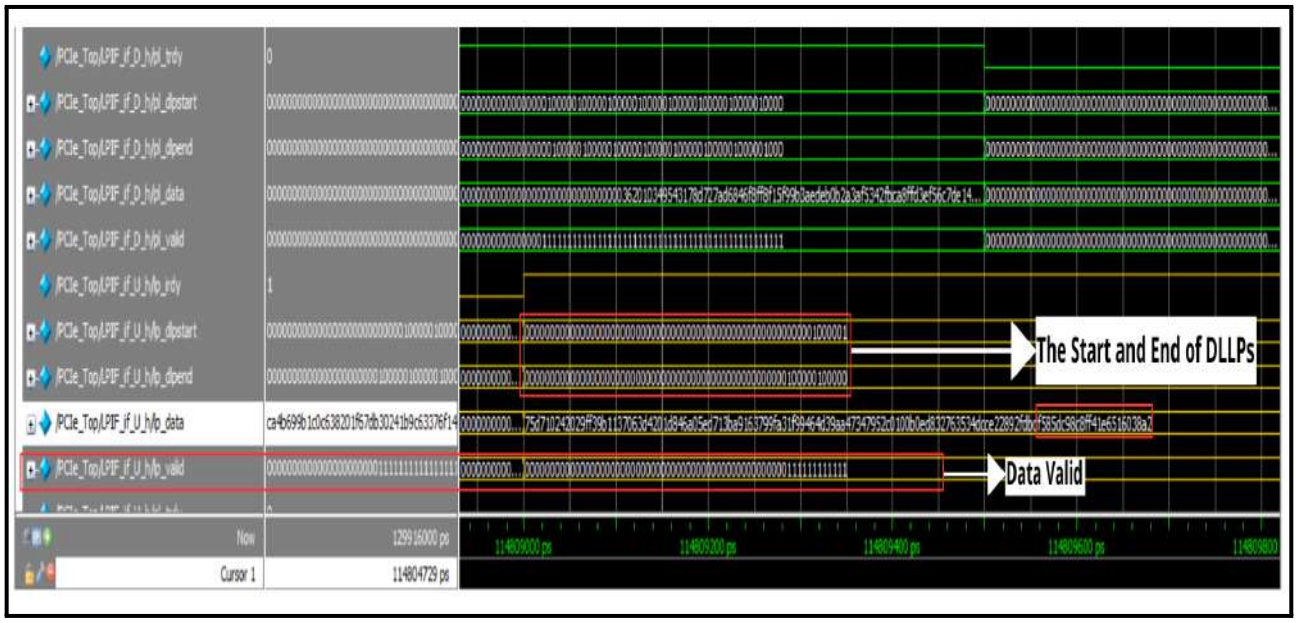


Figure 124: Transmission of two DLLPs

9.2.4 Reception of DLLPs from upstream device to downstream device

The output results shows the reception of two DLLPs, where the start and end of each DLLP are marked by the `pl_dllpstart` and `pl_dllpend` signals. The `pl_valid` signal validates the data on the `pl_data` signal.

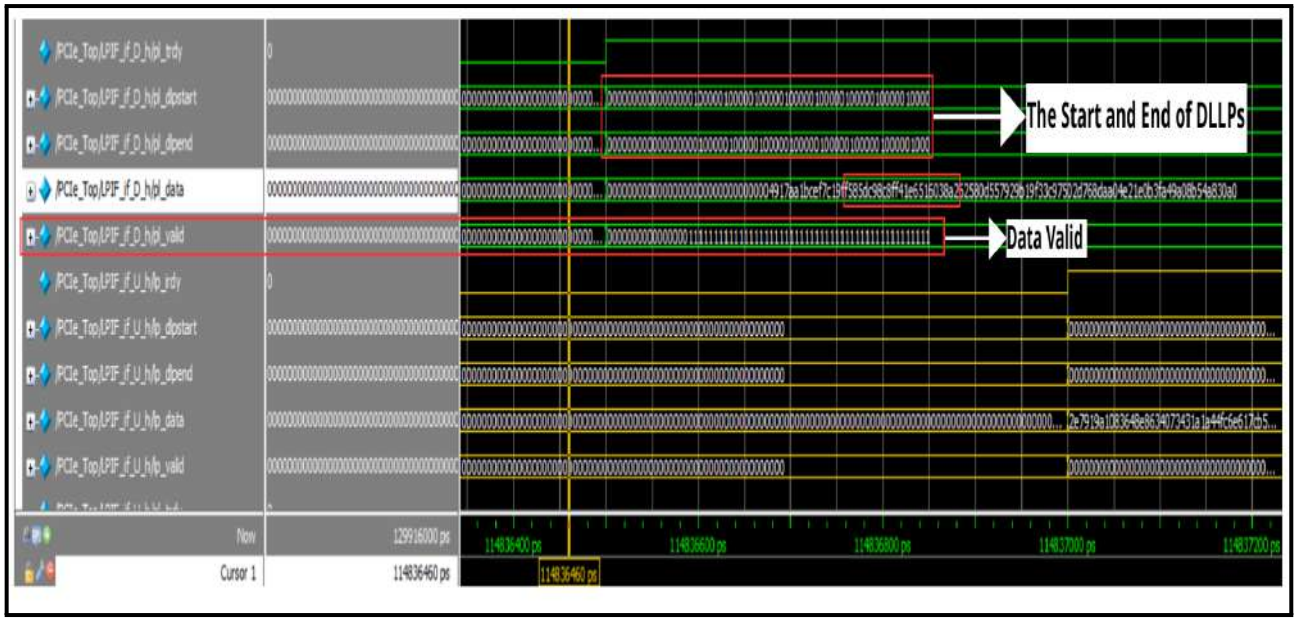


Figure 125: Reception of two DLLPs

9.3 Transmission and Reception of TLPs and DLLPs in one transfer

In this part, we transmit and receive mix of TLPs and DLLPs from downstream to upstream and from upstream to downstream.

```
# UVM_INFO PCIe_Test.sv(315) @ 116806000: uvm_test_top [PCIe_Test] ################ start send TLP and DLLP Seq ################
# UVM_INFO DLLP_Seq_TX_MASTER_U.sv(26) @ 116826000: uvm_test_top.PCIe_Env_h.TX_Master_U_Agent.TX_Master_U_Sequencer_h@DLLP_Seq_TX_MASTER_U [DLLP_Seq_TX_MASTER_U]  drive Different DLLP in same
transfer sequence in TX Master U
```

Figure 126: TLPs and DLLPs Sequences start running

9.3.1 Transmission of TLPs and DLLPs in one transfer from downstream device to upstream device

In this example shown in the simulation results includes the transmission of one TLP and two DLLPs back-to-back, as marked by the `lp_tlpstart`, `lp_tlpend`, `lp_dllpstart` and `lp_dllpend` signals. The data on `lp_data` is validated by the `lp_valid` signal.

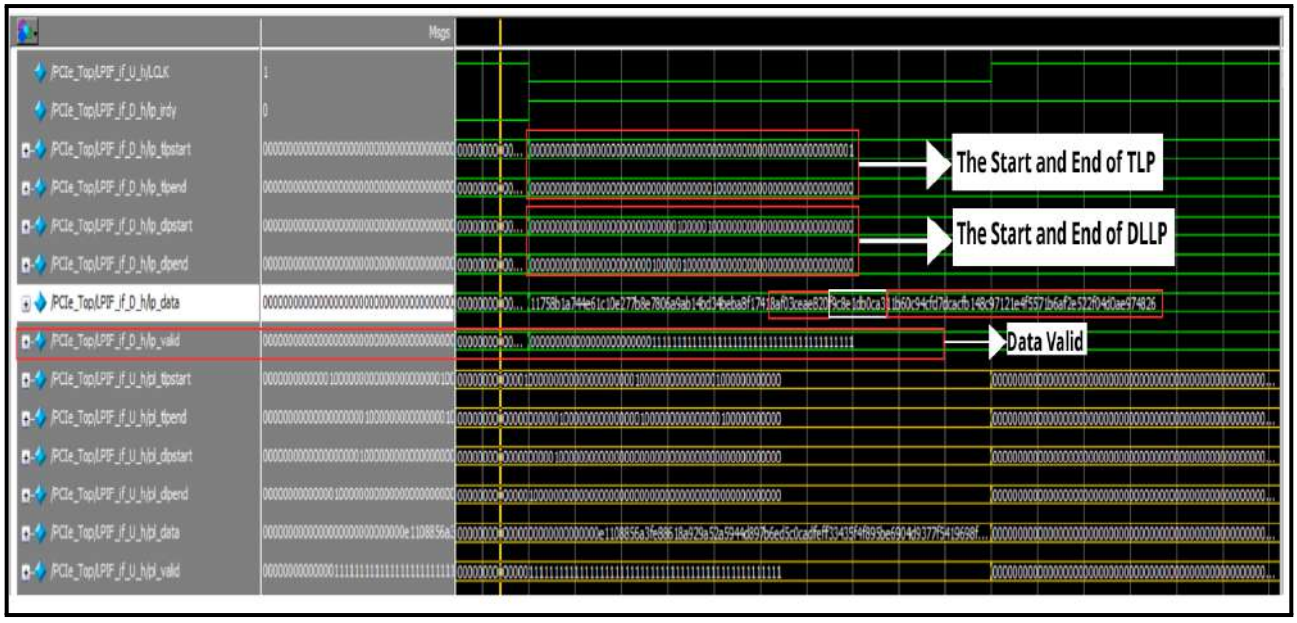


Figure 127: Transmission of one TLP and two DLLPs

9.3.2 Reception of TLPs and DLLPs from downstream device at upstream device

This simulation results show the reception of one TLP and two DLLPs back-to-back, where the start and end of each TLP and DLLP are marked by the `lp_tlpstart`, `lp_tlpend`, `lp_dllpstart` and `lp_dllpend` signals. The data on `lp_data` is validated by the `lp_valid` signal.

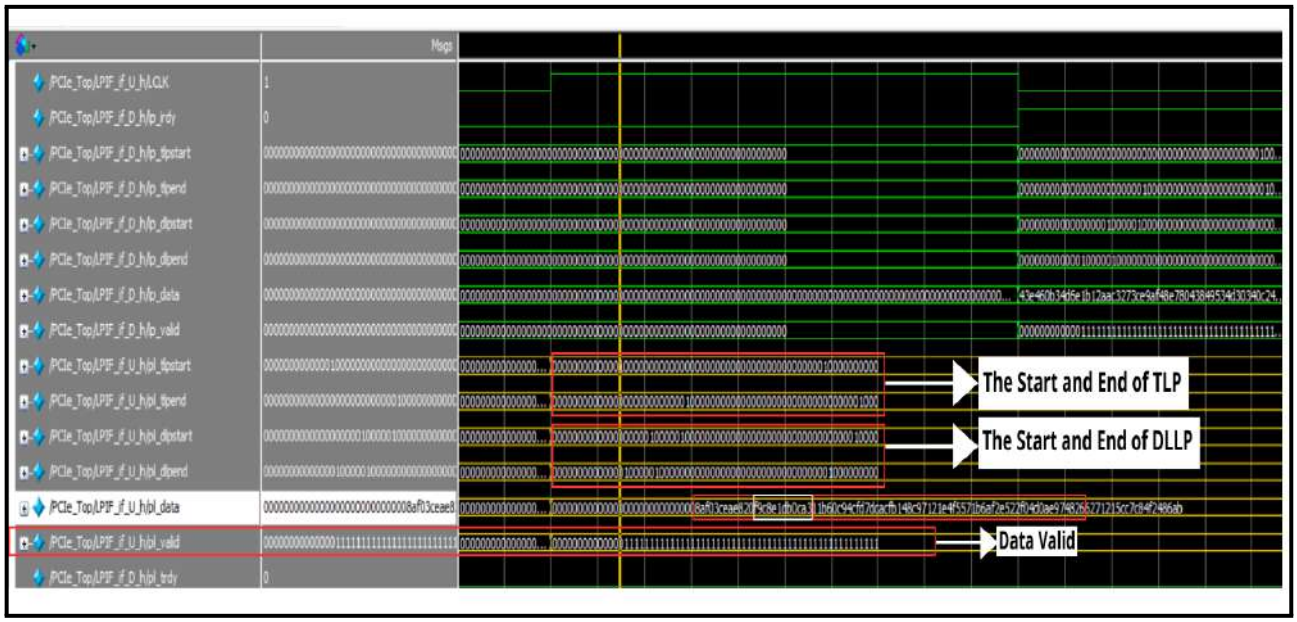


Figure 128: Reception of one TLP and two DLLPs

9.3.3 Transmission of TLPs and DLLPs from upstream device to downstream device

In this example shown in the simulation results includes the transmission of one TLP and two DLLP back-to-back, as marked by the `lp_tlpstart`, `lp_tlpend`, `lp_dllpstart` and `lp_dllpend` signals. The data on `lp_data` is validated by the `lp_valid` signal.

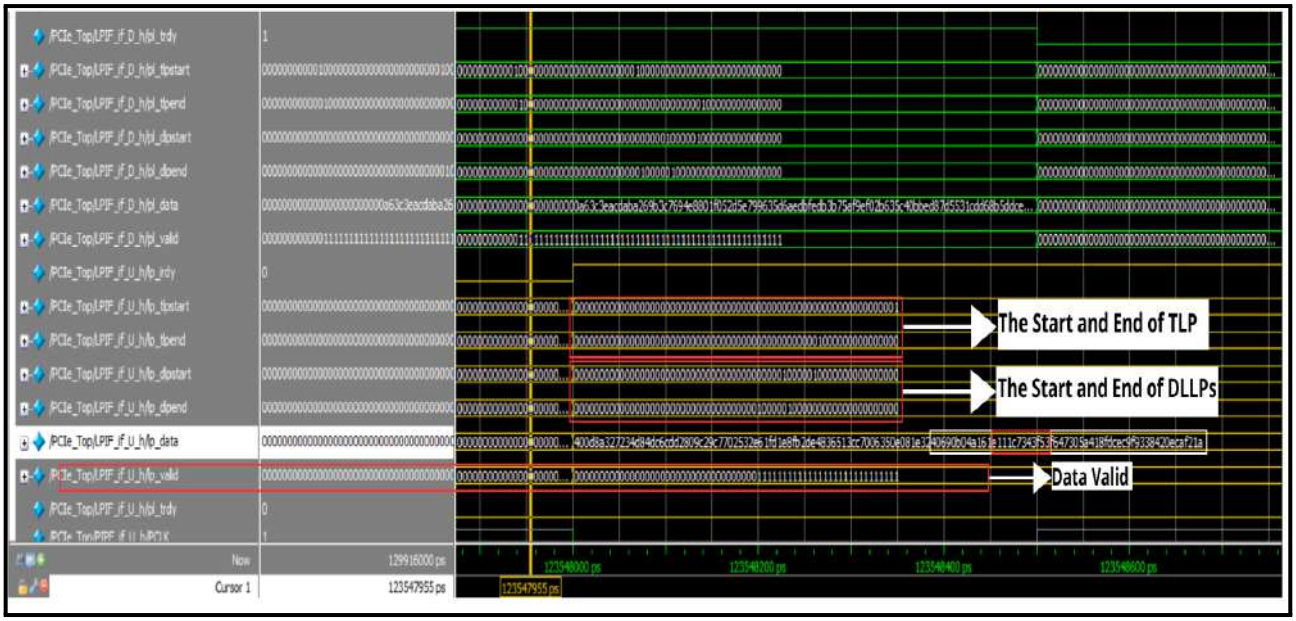


Figure 129: Transmission of one TLP and two DLLPs

9.3.4 Reception of TLPs and DLLPs in one transfer from upstream device to downstream device

This simulation results show the reception of of one TLP and two DLLPs back-to-back, where the start and end of each TLP and DLLP are marked by the `lp_tlpstart`, `lp_tlpend`, `lp_dllpstart` and `lp_dllpend` signals. The data on `lp_data` is validated by the `lp_valid` signal.

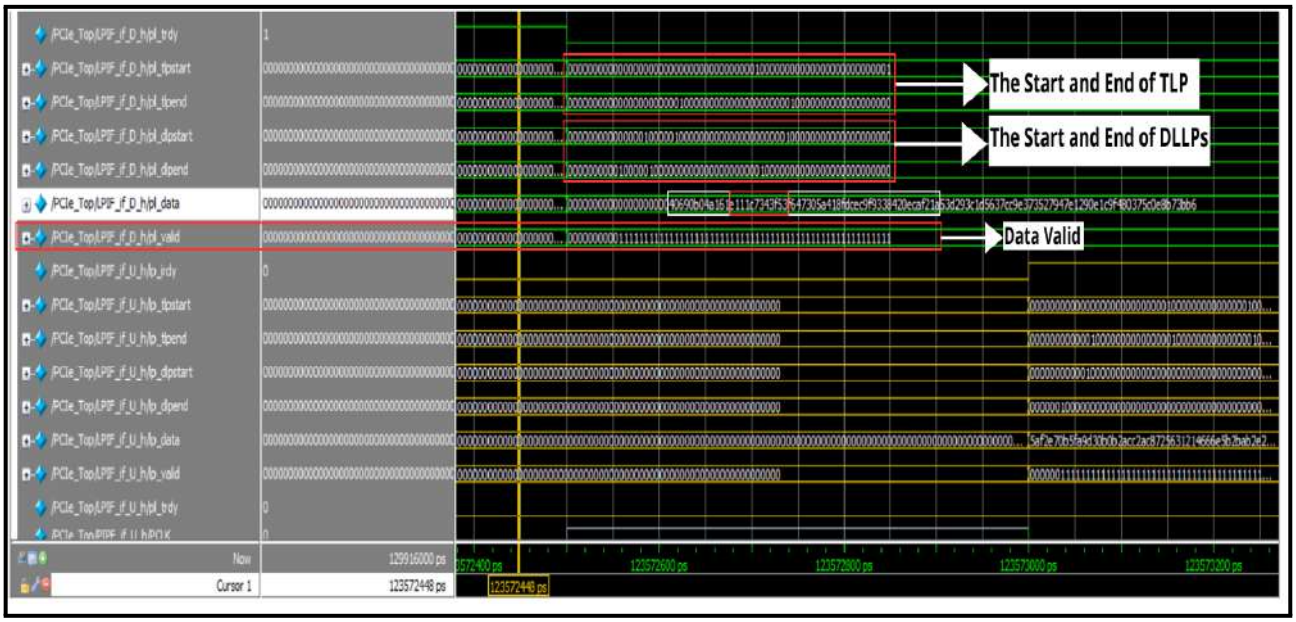


Figure 130: Reception of one TLP and two DLLPs

10 Error Injection

10.1 Introduction to error injection in digital verification

Error injection is a critical technique in the digital verification process, used to validate the robustness and fault-handling capabilities of digital systems.By intentionally introducing errors into data paths, control signals, or protocol transactions, verification engineers can observe how a device under test (DUT) behaves under faulty or unexpected conditions, ensuring reliability and compliance with specifications.

In our setup, we implement error injection between two PCIe (Peripheral Component Interconnect Express) devices.This is achieved through a custom-designed adapter module placed in the data path between the downstream and upstream devices.The adapter monitors the PCIe traffic and selectively injects errors such as corrupting data symbols or modifying protocol fields based on predefined conditions or randomized criteria. This allows us to evaluate how the receiving device handles malformed packets, invalid states, or protocol violations during actual communication, thereby strengthening the verification coverage and exposing corner-case failures.

This methodology not only enhances the quality of verification but also simulates real-world scenarios, such as transmission faults, which might affect data integrity in deployed systems.

10.2 Error Injection Sequence

we need a sequence to run , this sequence must do the normal operation for linkup or speed recovery until it reaches the state we choose to inject error in .

There is a sequence for each substate in linkup or recovery that only differs in the substate_err field that represents the substate we want to test the device's behavior in it if data is corrupted .

We run these sequences sequentially .

```
class polling_active_error_seq extends uvm_sequence #(PIPE_seq_item);
 `uvm_object_utils(polling_active_error_seq)

    PIPE_seq_item seq_item;

    function new (string name = "polling_active_error_seq");

    super.new(name);

    endfunction

    task body ();

    seq_item = PIPE_seq_item::type_id::create("seq_item");

    start_item(seq_item);

        seq_item.operation = 2'b01;
        seq_item.substate_err = `Polling_Active;

    finish_item(seq_item);

    endtask

endclass
```

Figure 131: Example of Error Injection Sequence

Each sequence for a substate enables a task for this substate that injects the error with different means like corrupting COM characters , mixing TS1 AND TS2 identifiers , changing EC bits , changing supported speed field , changing LF value and FS value .

10.3 Polling state

10.3.1 Polling Active:

During the Polling.Active substate, both the upstream and downstream DUTs exchange TS1 OS. Each receiver (Rx) inspects the incoming TS1s to ensure the correctness of their fields; for example, the COM field should equal 8'hBC. Any TS1 with an invalid field is discarded by the Rx. Concurrently, the transmitter (Tx) of both DUTs continuously sends TS1s up to 1024 TS1s during this phase, If the receiver detects and accepts at least eight valid TS1s, the LTSSM of the DUT transitions to the next substate, Polling.Configuration. If fewer than eight valid TS1s are received, the transmitter continues sending TS1s until either:

- Eight valid TS1s are received, or
- A timeout of 24 ms occurs.

If the timeout expires without meeting the criteria, both DUTs revert back to the Detect.Quiet state.

```
case(err_index)

    3'b000:begin //com injection

        sym = 1;
        get_cycle = 0;

        if(TS_num_to_up == get_cycle)begin //first part of TS
            for (int j =1 ;j<=`LANESNUMBER ; j=j+1 ) begin

                PIPE_seq_item_From_Down.TxData[(j*`MAXPIPEWIDTH)-((sym-1)*8)-1 -: 8] = 8'h00; // block the com

            end
        end
    end
```

Figure 132: Example of COM Field Error Injection in TS1 OS

To evaluate the robustness of the link initialization process, TS1 error injection is performed by the Adapter. For each TS1 transmitted from the downstream Tx to the upstream Rx (whenever `PIPE_seq_item_f.substate_err == `Polling_Active`), the Adapter introduces an error in one of the following TS1 fields (COM field, Link number, Lane number, Rate bits or TS1 ID), A similar injection is applied in the opposite direction—from the upstream Tx to the downstream Rx.

```
case(err_index)

    3'b000:begin //com injection ···
    end
    3'b001:begin //link num injection ···
    end
    3'b010:begin //lane num injection ···
    end
    3'b011:begin //rate bits injection ···
    end

    3'b100:begin //TS1_id injection ···
    end

endcase
```

Figure 133: Different Types of TS1 Error Injection

Note: Since the PIPE interface is 32 bits wide per lane, transmitting a complete TS1 (128 bits) at Gen 1 requires four clock cycles.

Simulation Results:

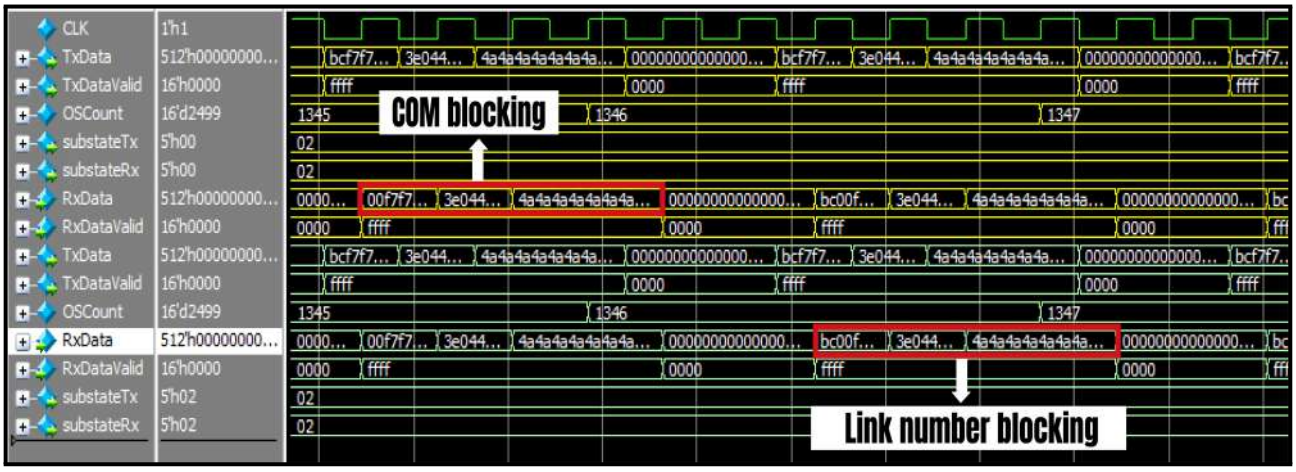


Figure 134: Error Injection Simulation for Polling.Active

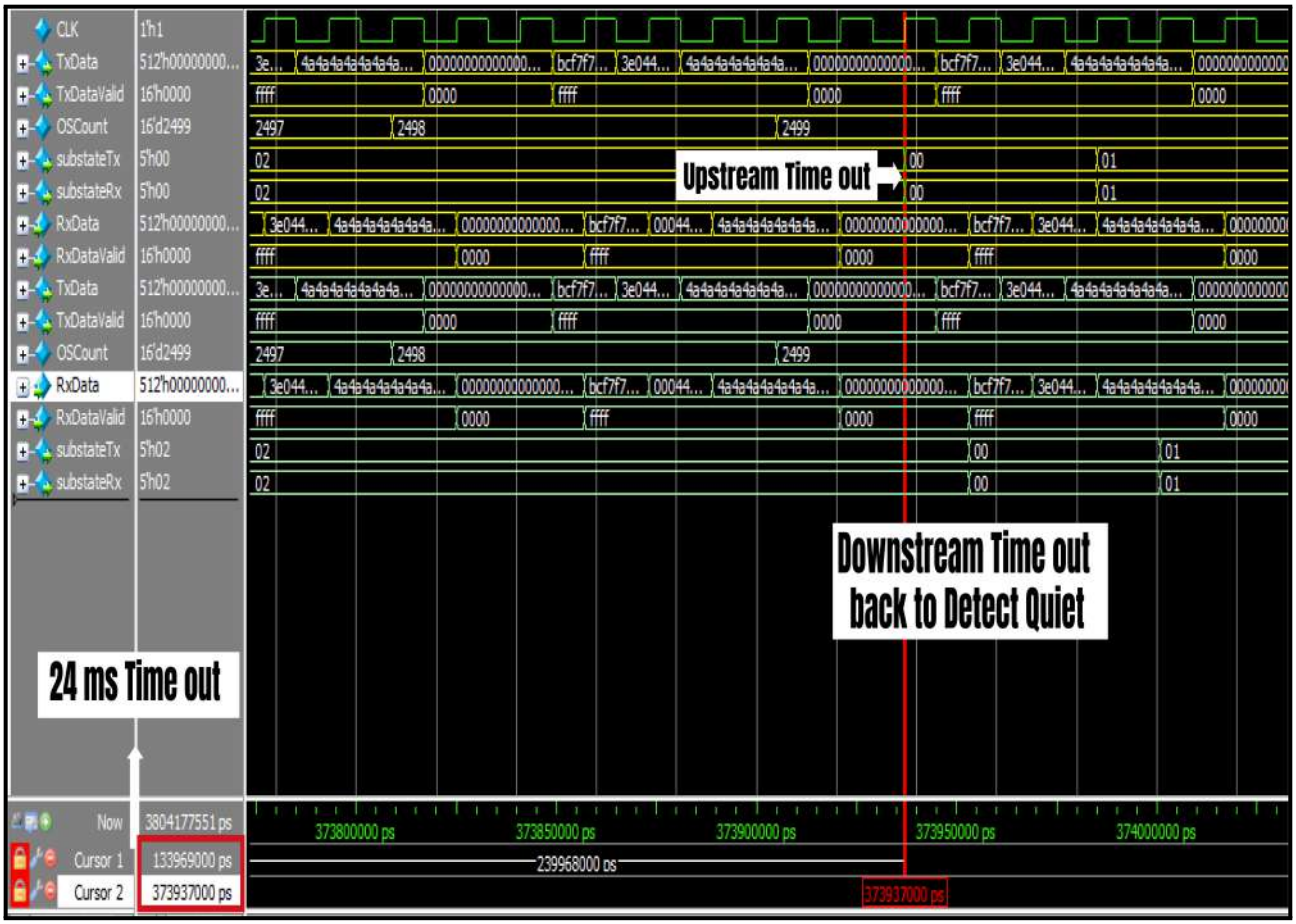


Figure 135: Polling.Active Timeout (24ms timeout)

10.3.2 Polling Configuration:

In the Polling.Configuration substate, both DUTs exchange TS2 ordered sets. similar to the behavior in Polling.Active, The receiver (Rx) examines each incoming TS2 to verify the correctness of its fields; for ex-

ample, the Link Number and Lane Number fields must be set to PAD (8'hF7).
A transition to the Configuration state occurs when:

- The receiver successfully detects eight consecutive valid TS2s, and
- The transmitter (Tx) has sent at least sixteen TS2s after receiving one valid TS2.

If this condition is not met within the 48 ms timeout period, the LTSSM transitions back to the Detect.Quiet state.
Similar to Polling.Active, error injection mechanisms are applied to validate robustness. These include corrupting the following TS2 fields (COM field, Link Number, Lane Number, Rate bits, TS2 ID).

```
case(err_index2)

    3'b000:begin //com injection···
    end
    3'b001:begin //link num injection···
    end
    3'b010:begin //lane num injection···
    end
    3'b011:begin //rate bits injection···
    end

    3'b100:begin //TS2_id injection


        if(PIPE_seq_item_From_Up.TxData == {64{8'h45}})begin
            for (int j =1 ;j<=`LANESNUMBER ; j=j+1 ) begin

                PIPE_seq_item_From_Up.TxData[(j*`MAXPIPEWIDTH)-1 -: 32] = 32'h00000000; // block TS2 id

            end
        end

    end

endcase
```

Figure 136: Different Types of Error Injection in Polling.Configuration

Such error injections test the DUTs' ability to discard invalid TS2s and enforce correct state transitions only when valid link negotiation occurs.

Figure 137: Error Injection Simulation for Polling.Configuration

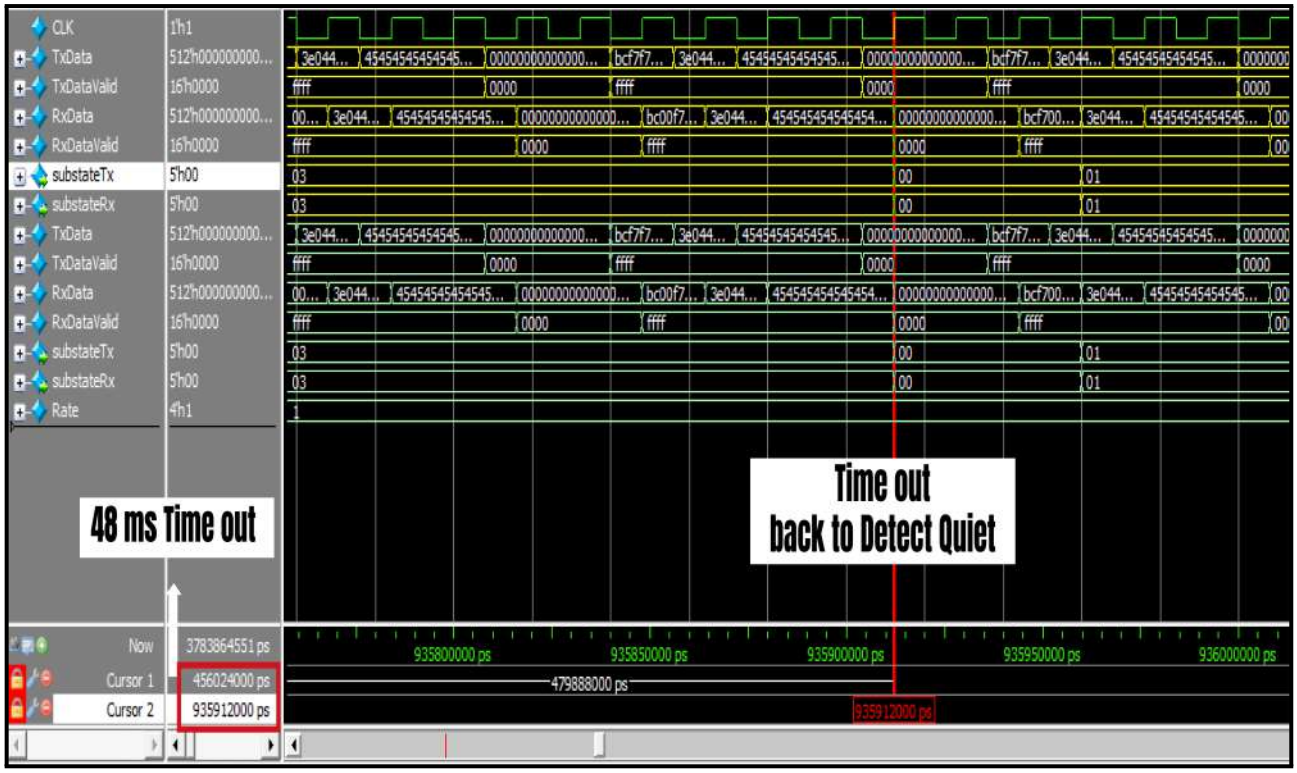


Figure 138: Polling.Configuration Timeout (48ms Timeout)

10.4 Configuration state

10.4.1 Configuration link width start

- In this LTSSM substate, the DUTs exchange TS1 ordered sets containing the negotiated link number. For the LTSSM to transition to the subsequent substate, Configuration.Linkwidth.Accept, the RX must receive two consecutive valid TS1s in which the link number is present and not marked as PAD.
- If the RX fails to receive the expected TS1s within a 24 ms timeout window, a timeout condition is triggered and the LTSSM transitions back to the Detect state.
- To verify the robustness and compliance of the RTL design under faulty link conditions, various error scenarios are deliberately introduced into the TS1s. These scenarios are intended to evaluate how the state machine handles corrupted or malformed TS1 sequences. The goal is to ensure that the RTL adheres strictly to the PCIe specification under each failure condition and that it transitions appropriately in the presence of invalid or incomplete training sequences.

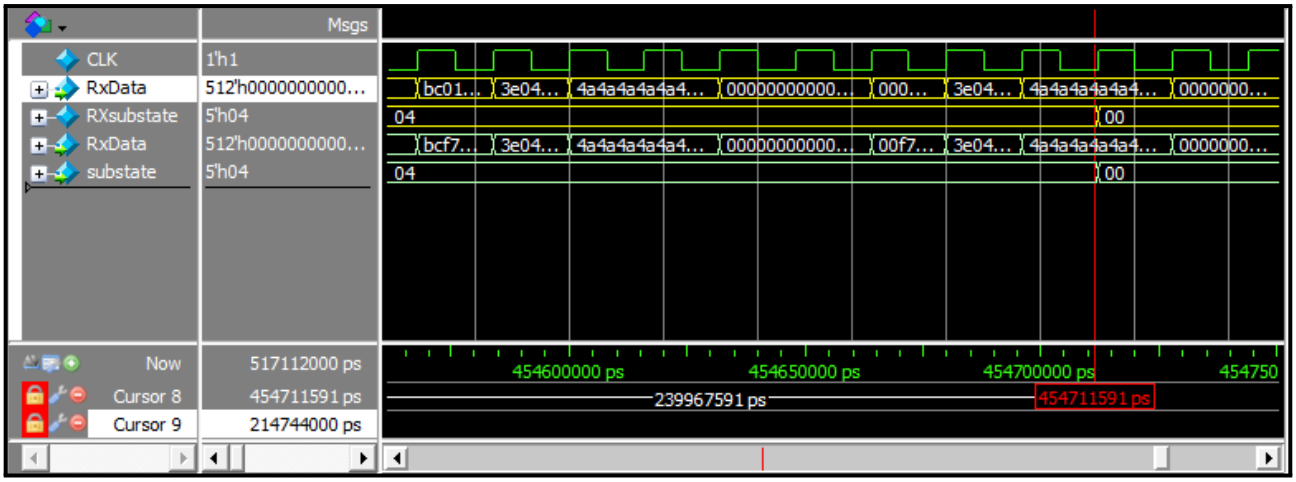


Figure 139: Configuration.Linkwidth.start simulation (24ms Timeout)

10.4.2 Configuration lane number wait

- In this LTSSM substate, the TX of the downstream DUT transmits TS2 ordered sets that include the negotiated lane number. In response, the TX of the upstream DUT sends TS1 ordered sets containing the same lane number.
- If the DUT successfully receives two consecutive correct TS1/TS2 ordered sets with matching lane numbers, the LTSSM transitions to the next substate.
- However, if the expected ordered sets are not received within a 2 ms timeout window, the LTSSM deems the training unsuccessful and reverts to the Detect state.

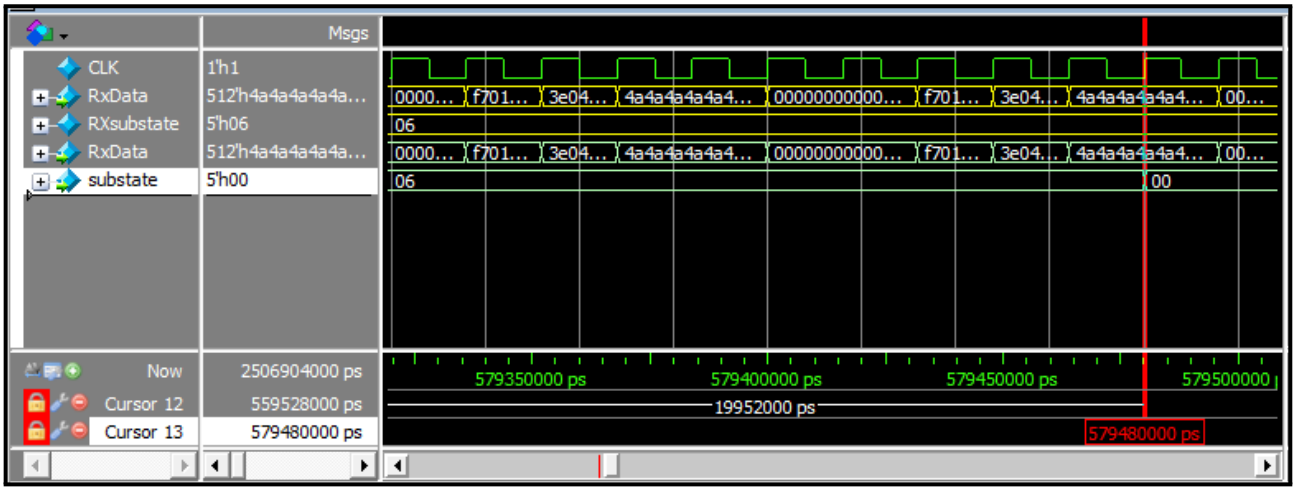


Figure 140: Configuration.lanenum.wait simulation (2ms Timeout)

10.4.3 Configuration lane number accept

- If two consecutive TS1s (Downstream) or TS2s (Upstream) with matching non-PAD Link and Lane numbers are received and align with what is being transmitted, the Ports transition to **Configuration.Complete**.
- If no viable Link can be established or all Lanes receive PAD values, they transition to the **Detect** state.

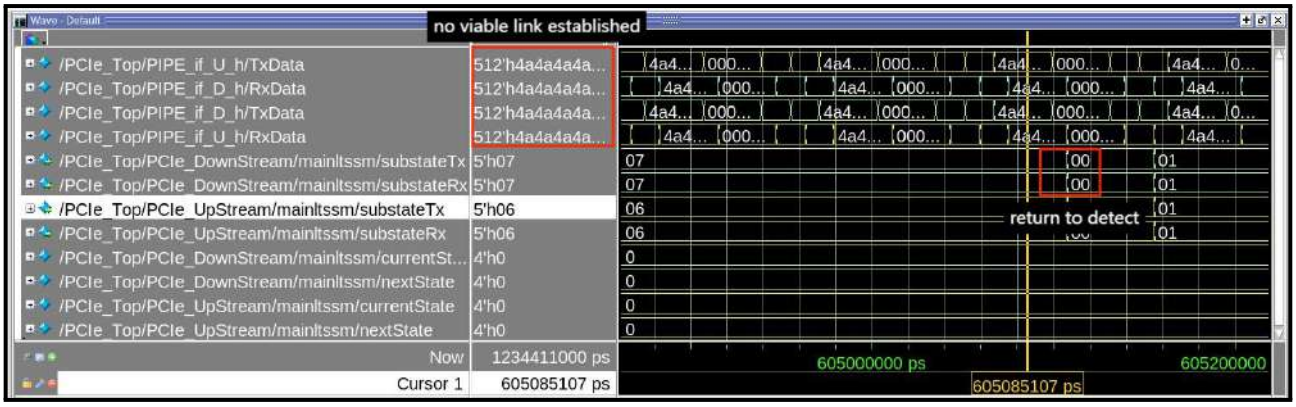


Figure 141: Example of error injection in Configuration lane number accept in GEN1

10.4.4 Configuration complete

- Transition to **Configuration.Idle** requires receiving 8 aligned TS2s and sending ≥16 TS2s.
- failures or a 2 ms timeout revert to **Detect**.

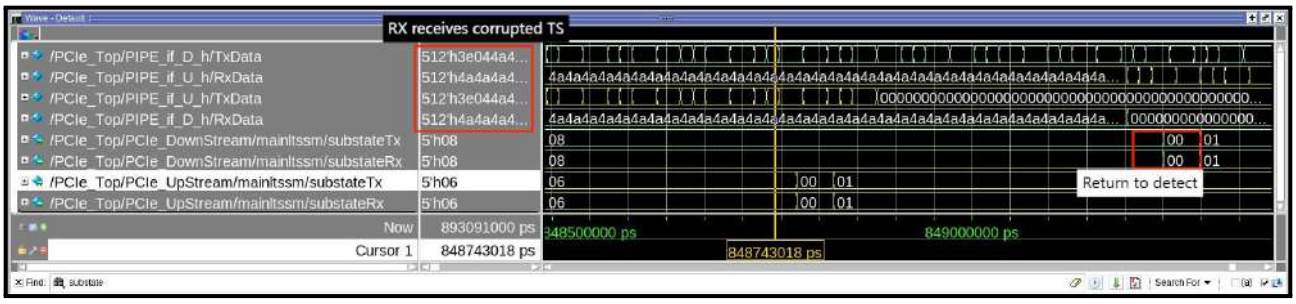


Figure 142: Example of error injection in Configuration complete in GEN1

10.4.5 Configuration Idle

In this state we have 3 exit cases:

- Transition requires sending 16 Idles after initial detection.
- The Physical Layer reports to the upper layers that the link is operational (Linkup = 1b).
- A 2 ms timeout triggers recovery (via **idle_to_rlock_transitioned**, limited to 256 attempts) or reverts to **Detect** if retries reach the limit.

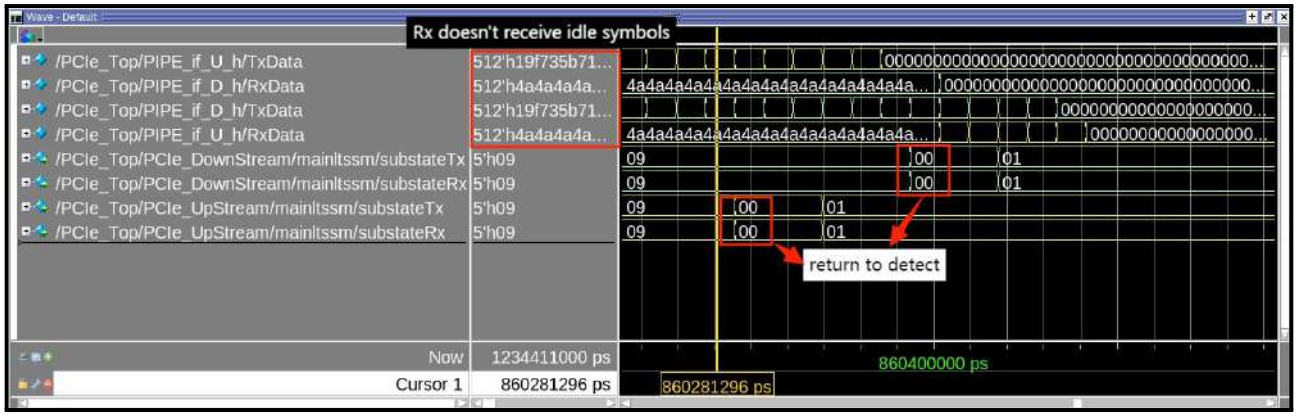


Figure 143: Example of error injection in Configuration idle in GEN1

10.5 Recovery state

To remember why we enter the recovery states from the normal operational L0 state refer to 5.7.1

10.5.1 Recovery Receiver Lock

After entering the recovery lock substate, there are 6 exit cases. We are going to cover the cases where an error occurs. (You can refer to 5.7.2 for the full details). After 24ms timeout:

- Exit to **Recovery.Speed** if the higher speed fails (`changed_speed_recovery` = 0b) or a new speed doesn't work (`changed_speed_recovery` = 1b), fallback to previous or 2.5 GT/s speed.
- Exit to **Detect State** if none of the above conditions are met after the timeout.

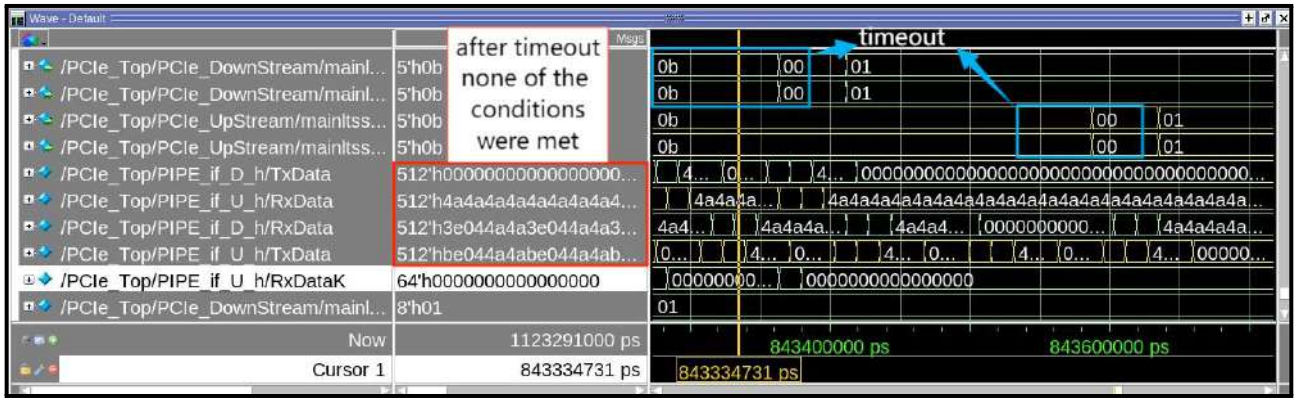


Figure 144: Example of error injection in Recovery lock in GEN1

10.5.2 Recovery Rcvr.CFG

There are two cases we need to check :

- If RX receives 8 consecutive TS1 it should go back to Configuration Linkwidth Start substate .
- If we reach 48ms and RX hasn't received 8 consecutive TS2 with speed change bit set to one or 8 consecutive scrambled TS2 with speed change bit set to zero , it times out to Detect Quiet substate .

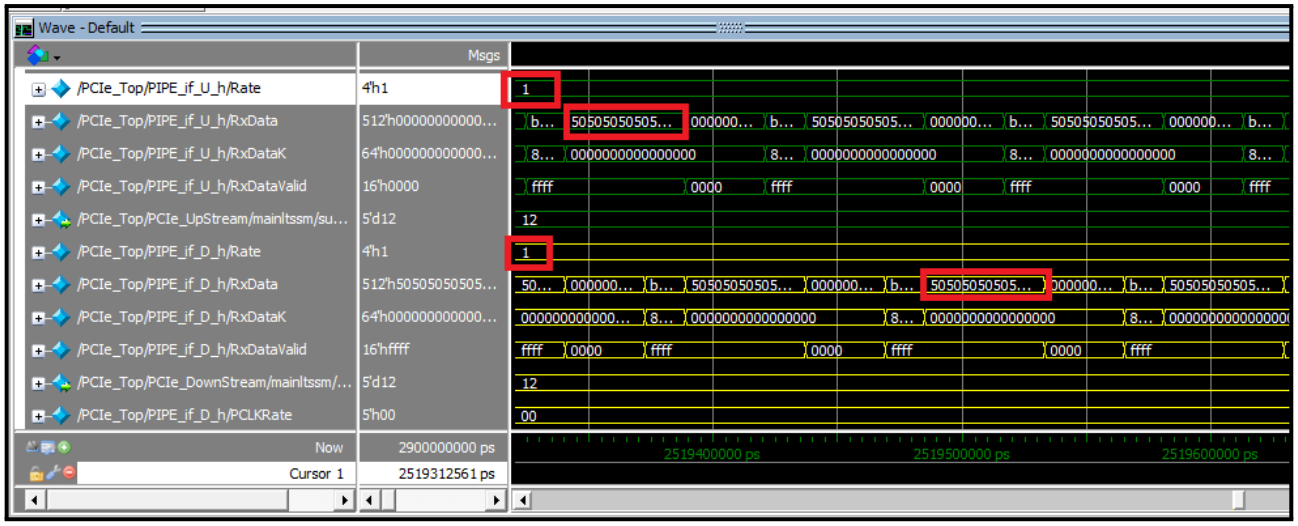


Figure 145: Example of error injection in Recovery Rcvr.CFG in GEN1

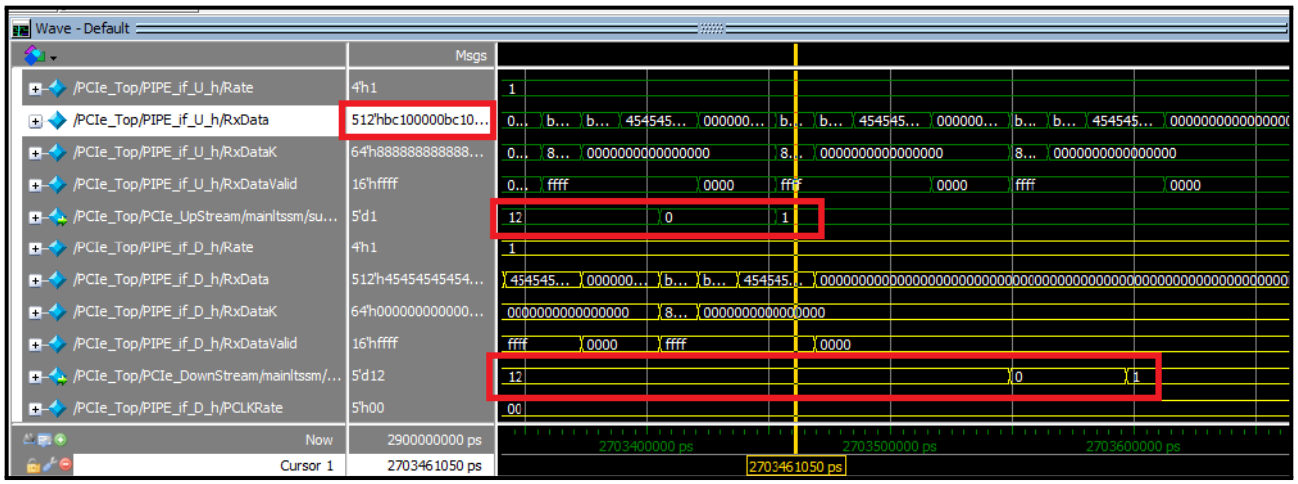


Figure 146: Example of error injection in Recovery Rcvr.CFG in GEN1

These simulation results show the errors injected to the data received in link numbers and TS identifiers that led to time out and going back to Detect Quiet (GEN1) .

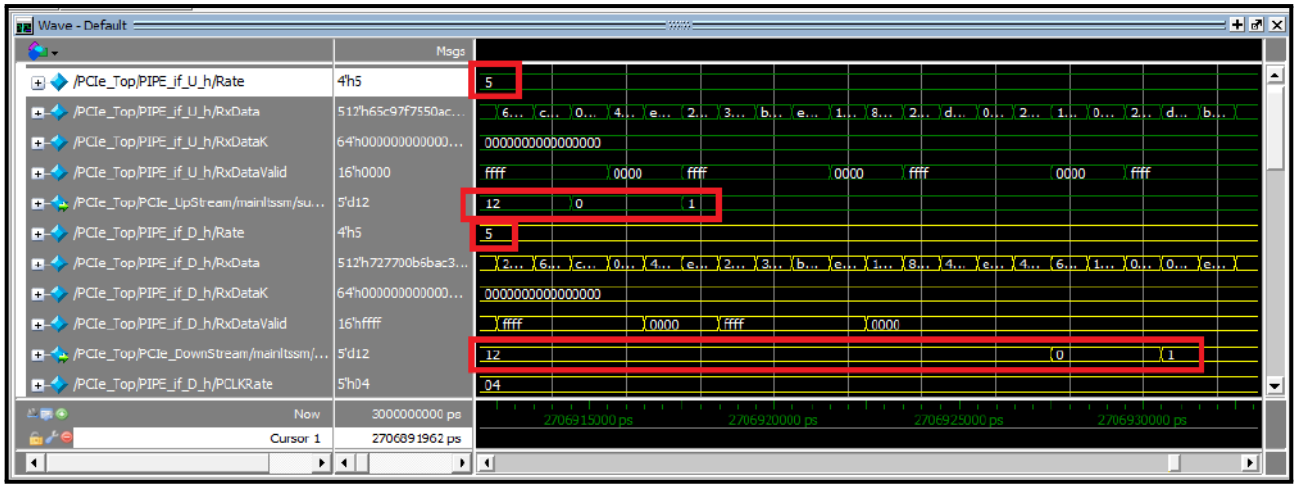


Figure 147: Example of error injection in Recovery Rcvr.CFG in GEN5

This snippet shows the same but for scrambled TS in GEN5

```
# UVM_INFO C:\Users\DELL\Desktop\Graduation_Project\PCIe_Env_Template\Grad_BY_Youssef\TB\PCIe_Scoreboard1_D.sv(324) @ 2703565000:
d1_D] Downstream RX [Recovery_RcvrCfg] Time Out To Detect_Quiet
```

Figure 148: Downstream RX times out after 48ms

```
# UVM_INFO C:\Users\DELL\Desktop\Graduation_Project\PCIe_Env_Template\Grad_BY_Youssef\TB\PCIe_Scoreboard1_U.sv(369) @ 2703405001:
d1_U] Upstream RX [Recovery_RcvrCfg] Time Out To Detect_Quiet
```

Figure 149: UPstream RX times out after 48ms

10.5.3 Recovery Speed

This substate times out to Detect Quiet substate after 48ms if RX hasn't received more than one EIEOS .

The error is injected here by corrupting these EIEOS coming to RX or the EIOS (bc7c7c7c).

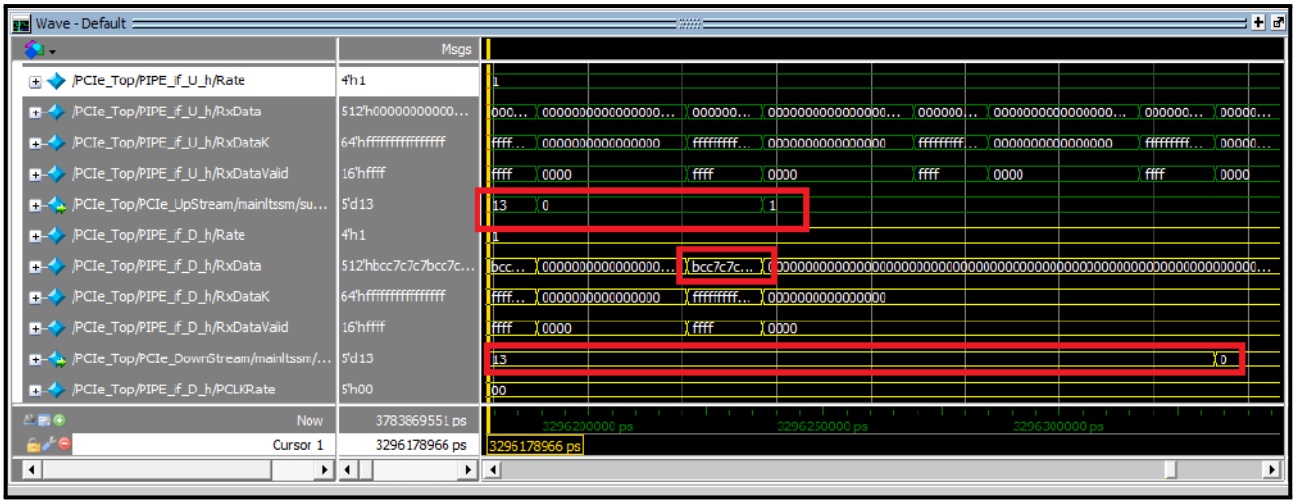


Figure 150: Error Injection in Recovery Speed

This snippet shows the error occurred in EIEOS (equal to zeros) and EIOS (bcc7c7c7) that led to time out

```
# UVM_INFO C:\Users\DELL\Desktop\Graduation_Project\PCIe_Env_Template\Grad_BY_Youssef\TB\PCIe_Scoreboardl_U.sv(297) @ 3294109001:
l_U] Upstream RX [Recovery_Speed] Time Out To Detect_Quiet
```

Figure 151: Upstream RX times out after 48ms

```
# UVM_INFO C:\Users\DELL\Desktop\Graduation_Project\PCIe_Env_Template\Grad_BY_Youssef\TB\PCIe_Scoreboardl_D.sv(279) @ 3296509001:
l_D] Downstream RX [Recovery_Speed] Time Out To Detect_Quiet
```

Figure 152: Downstream RX times out after 48ms

10.5.4 Phase 1

This substate times out to Recovery Speed substate after 12ms if RX hasn't received two consecutive scrambled TS1 with EC bits set to 2'b00 and the presets are set according to specs.

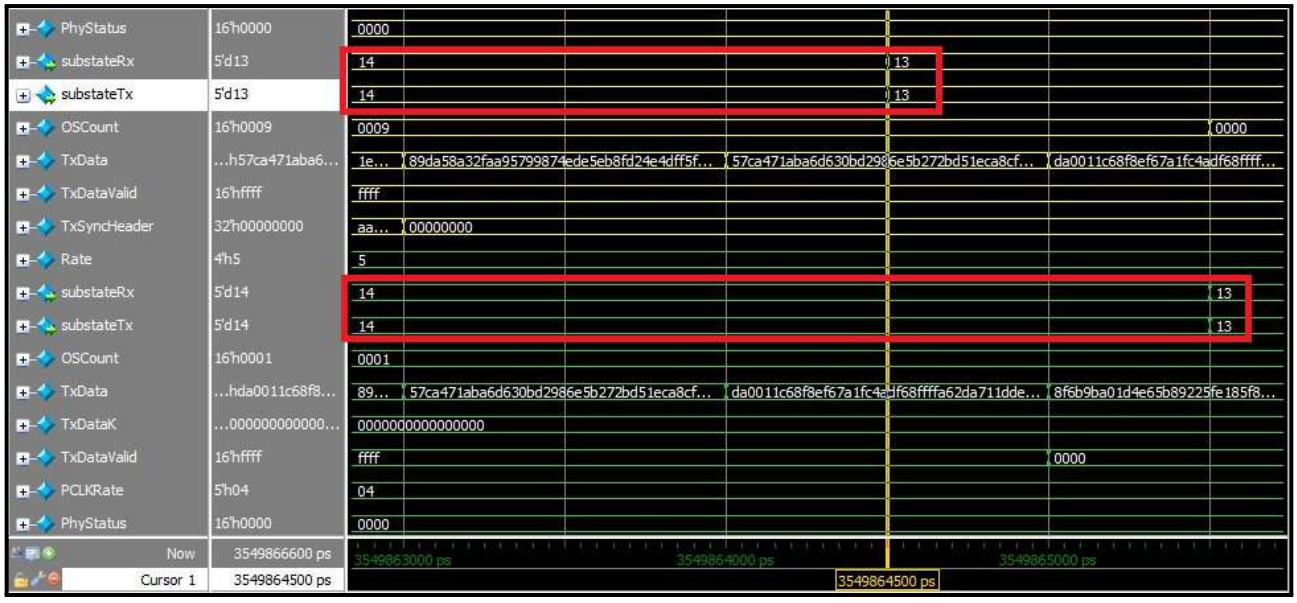


Figure 153: Error Injection in Phase 1

This snippet shows the error introduced in the link number field and EC bits field that led to time out

```
# UVM_INFO C:\Users\DELL\Desktop\Graduation_Project\PCIe_Env_Template\Grad_BY_Youssef\TB\PCIe_Scoreboard1_U.sv(702) @ 3529556500:
d1_U] Upstream TX [Phase1] Time Out To Recovery_Speed
```

Figure 154: Upstream device times out after 12ms

```
# UVM_INFO C:\Users\DELL\Desktop\Graduation_Project\PCIe_Env_Template\Grad_BY_Youssef\TB\RX_Slave_D_Monitor.sv(2573) @ 3649592500:
or_h [RX_Slave_D_Monitor] Time out occur in Phase1 , we get back to Recovery speed state
```

Figure 155: Downstream device times out after 12ms

10.5.5 Recovery Idle

This substate contains three different data blocks coming to the RX which are SKP ordered set , SDS and IDLE symbols .

To make the transition to L0 for full operation in GEN5 RX should receive one SKP followed by one SDS followed by 8 consecutive scrambled IDLE symbols .

The error can be injected to corrupt any of these data blocks and after 2ms it times out to Detect Quiet substate .

There is a case if RX receives two scrambled TS1 with PAD link numbers and lane numbers then the next state is Configuration Linkwidth Start substate .

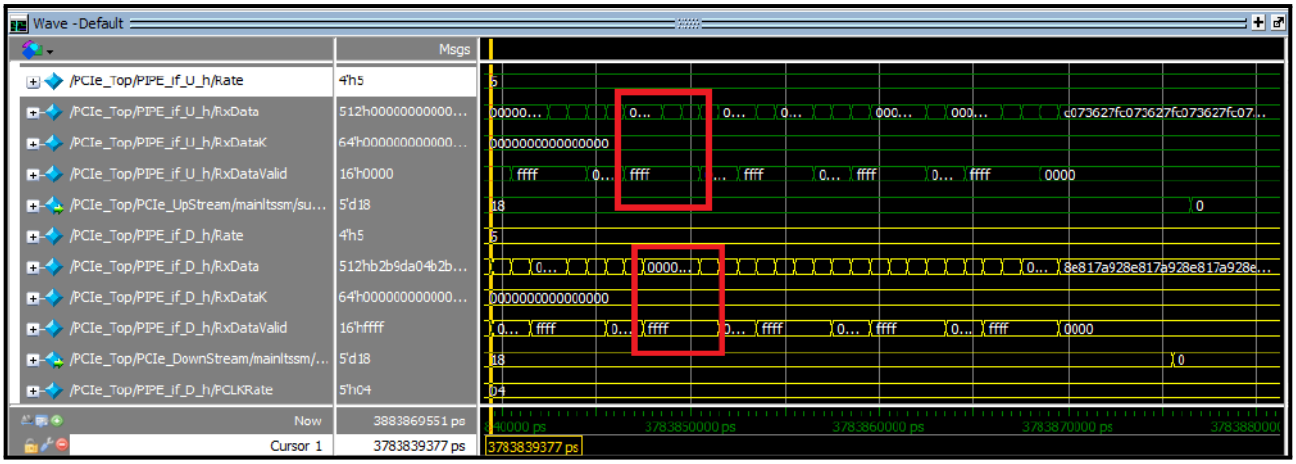


Figure 156: Error Injection in Recovery Idle

This snippet shows the inserted zeros in the scrambled idle symbols that led to time out

```
# UVM_INFO C:\Users\DELL\Desktop\Graduation_Project\PCIe_Env_Template\Grad_BY_Youssef\TB\PCIe_Scoreboardl_U.sv(390) @ 3783821501:
dl_U] Upstream RX [Recovery_Idle] Time Out To Detect_Quiet
```

Figure 157: Upstream device times out after 2ms

```
# UVM_INFO C:\Users\DELL\Desktop\Graduation_Project\PCIe_Env_Template\Grad_BY_Youssef\TB\PCIe_Scoreboardl_D.sv(345) @ 3783869501:
dl_D] Downstream RX [Recovery_Idle] Time Out To Detect_Quiet
```

Figure 158: Downstream device times out after 2ms

11 Verification Results and Reported Bugs

At this section, we present our verification results, including outputs from the scoreboard and code and functional coverage reports, and a summary of the bugs found, reported, and resolved during the simulation process.

11.1 Functional and Code Coverage Reports

In this part, we show the code and functional coverage achieved during verification.

11.1.1 Functional Coverage

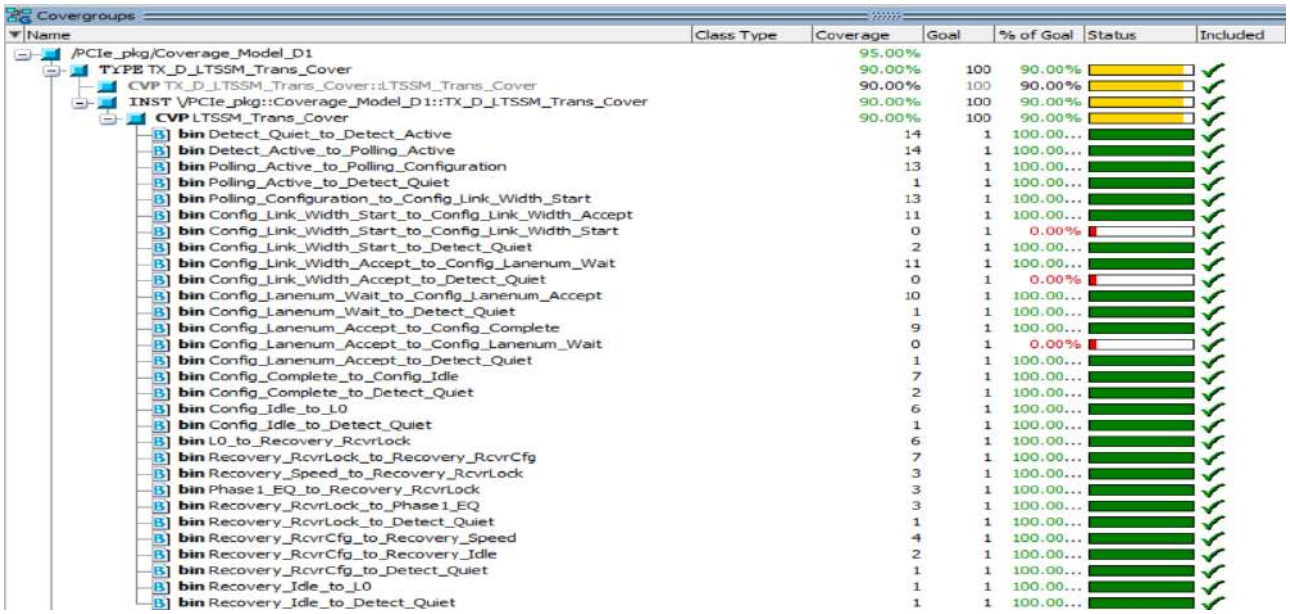


Figure 159: LTSSM Transitions at TX Downstream Side

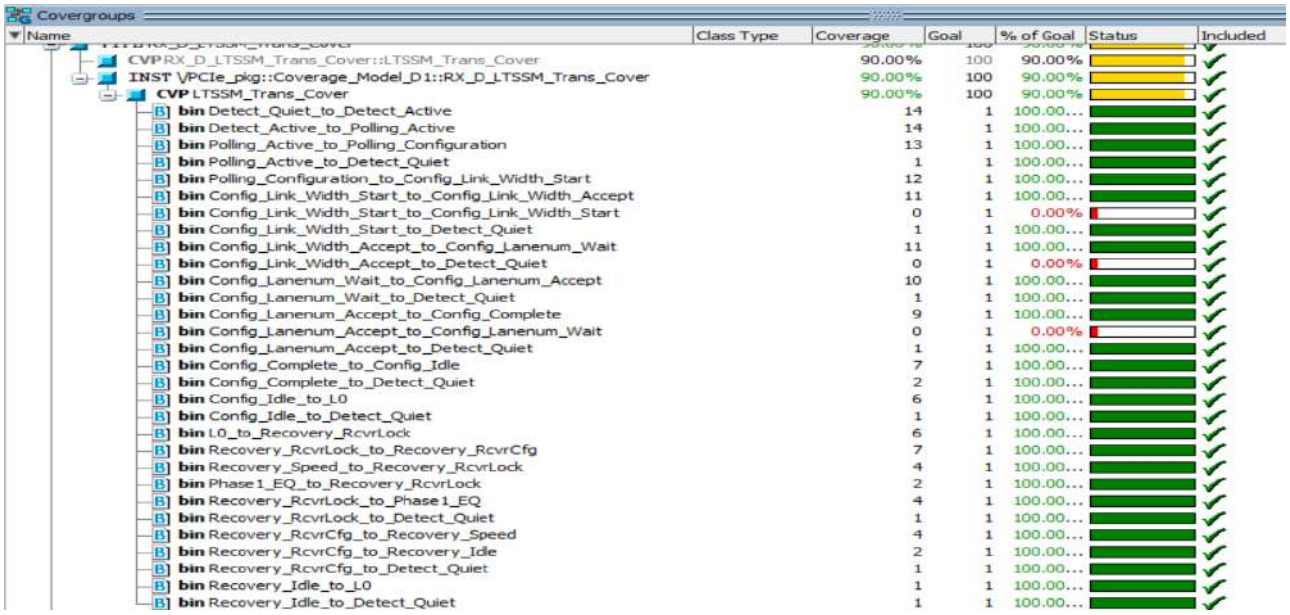


Figure 160: LTSSM Transitions at RX Downstream Side

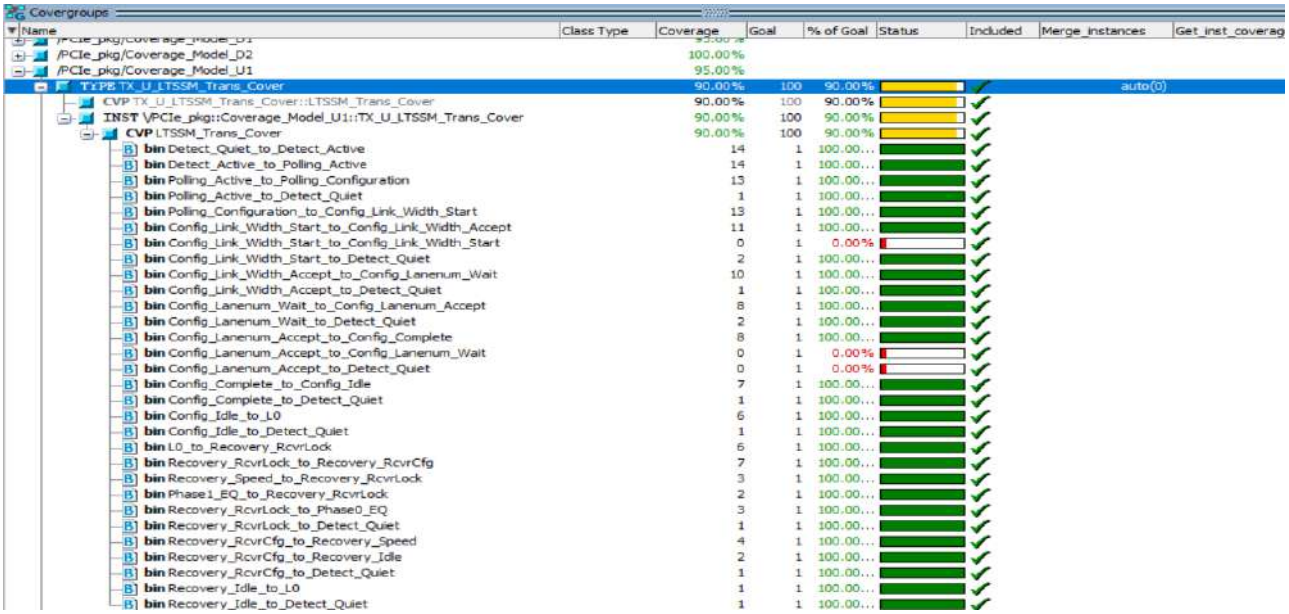


Figure 161: LTSSM Transitions at TX Upstream Side

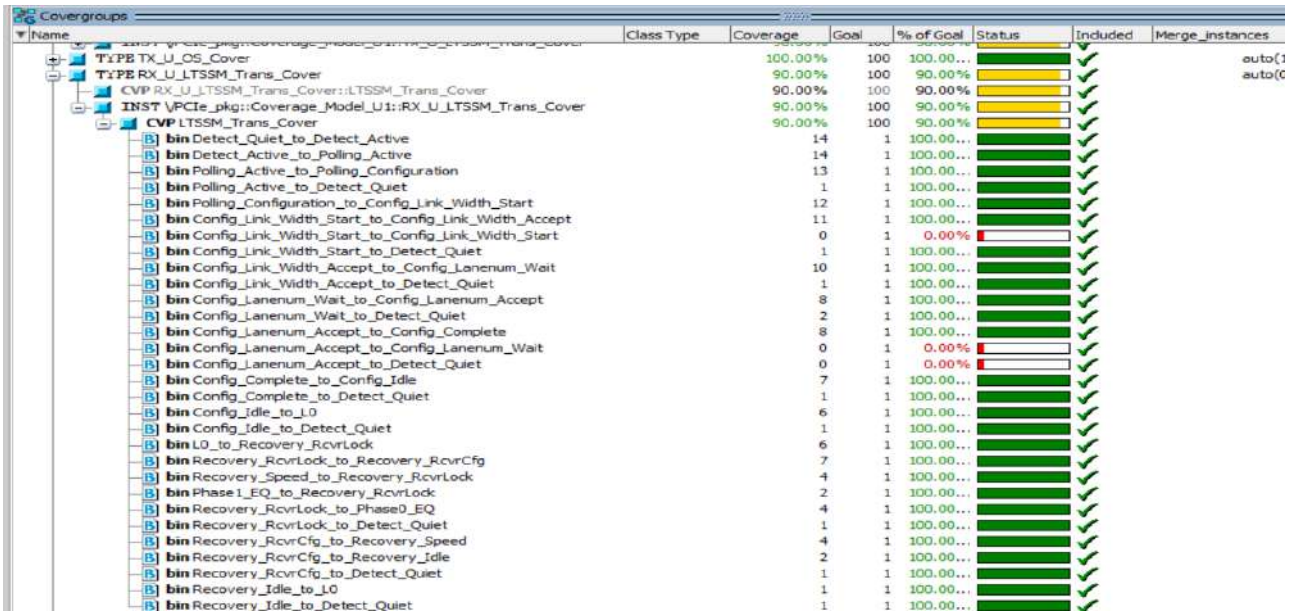


Figure 162: LTSSM Transitions at RX Upstream Side

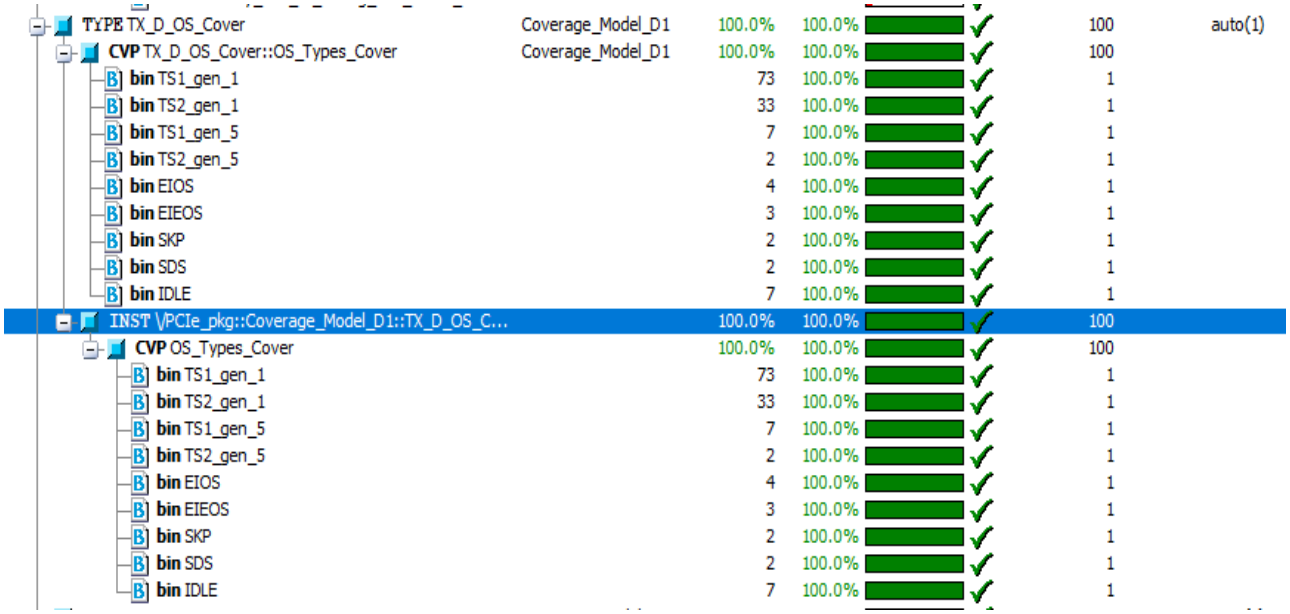


Figure 163: OS Types Covered at TX Downstream Side

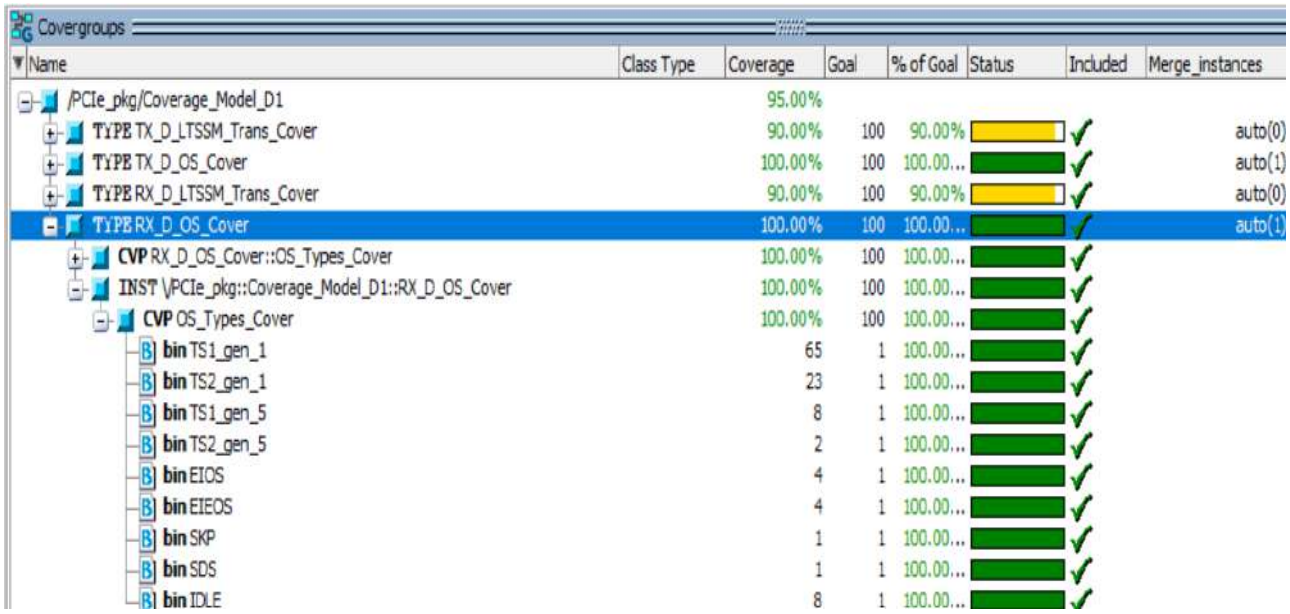


Figure 164: OS Types Covered at RX Downstream Side

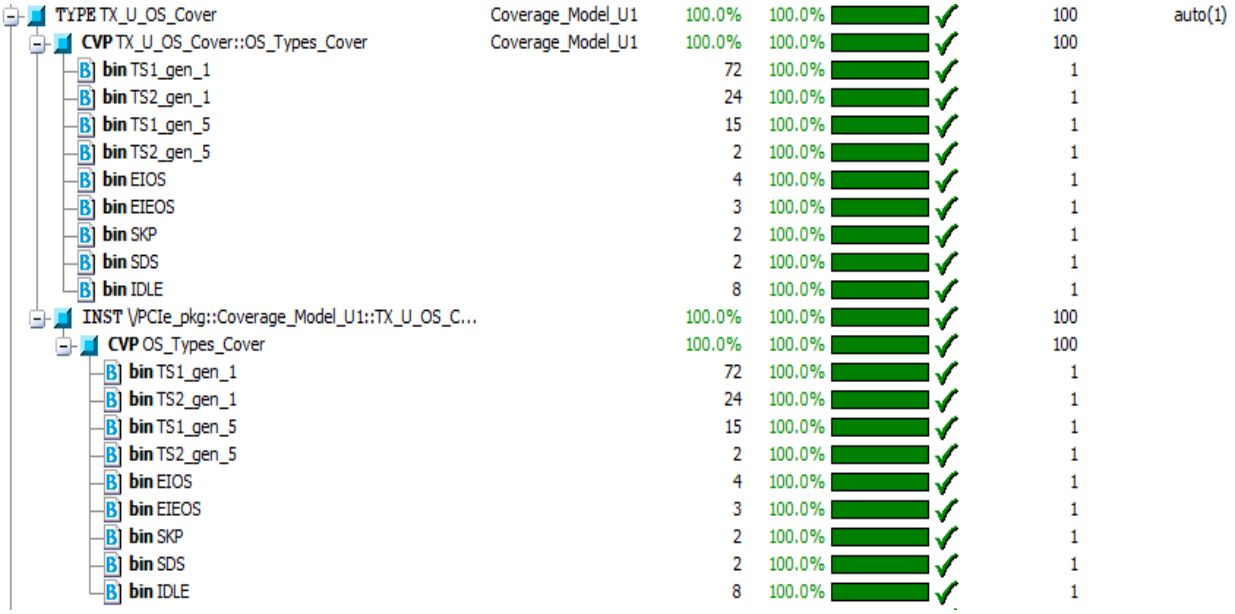


Figure 165: OS Types Covered at TX Upstream Side

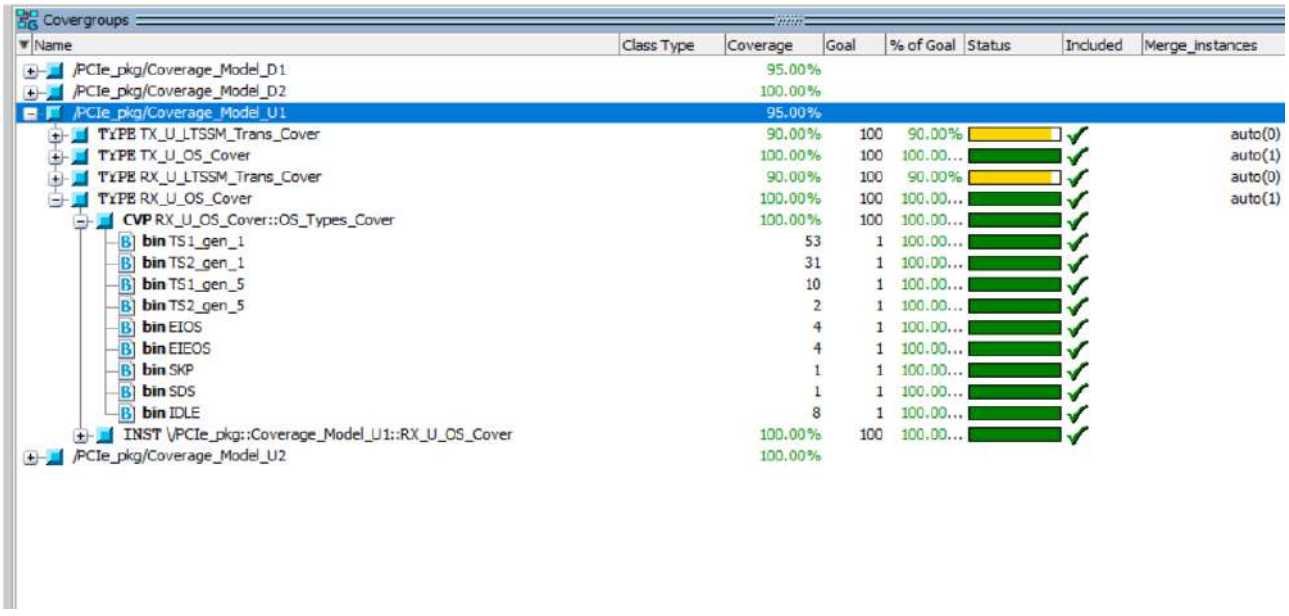
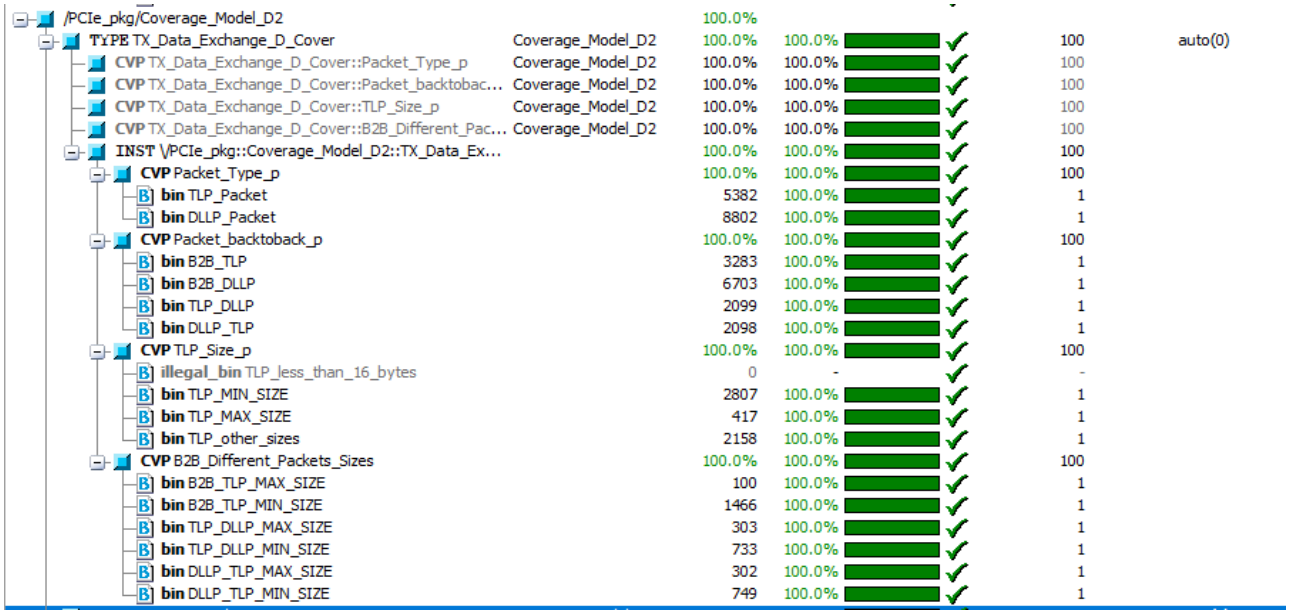
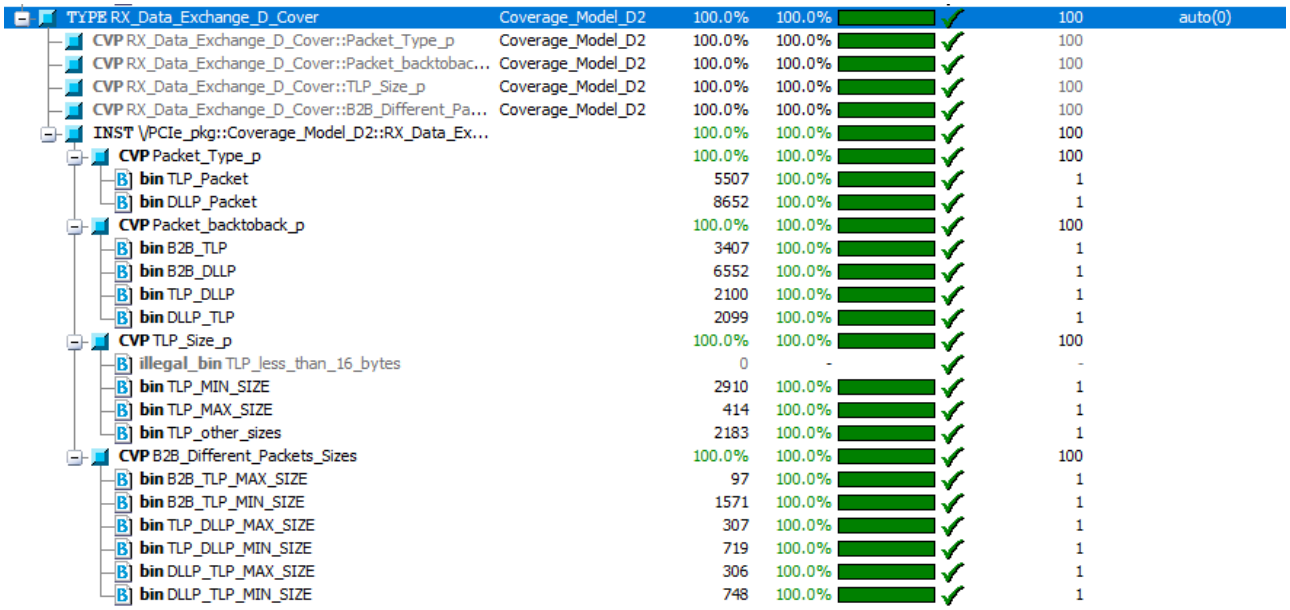


Figure 166: OS Types Covered at RX Upstream Side



/PCIe_pkg/Coverage_Model_D2		100.0%			
TYPE TX_Data_Exchange_D_Cover	Coverage_Model_D2	100.0%	100.0%	100	auto(0)
CVP TX_Data_Exchange_D_Cover::Packet_Type_p	Coverage_Model_D2	100.0%	100.0%	100	
CVP TX_Data_Exchange_D_Cover::Packet_backtobac...	Coverage_Model_D2	100.0%	100.0%	100	
CVP TX_Data_Exchange_D_Cover::TLP_Size_p	Coverage_Model_D2	100.0%	100.0%	100	
CVP TX_Data_Exchange_D_Cover::B2B_Different_Pac...	Coverage_Model_D2	100.0%	100.0%	100	
INST \/PCIe_pkg::Coverage_Model_D2::TX_Data_Ex...		100.0%	100.0%	100	
CVP Packet_Type_p		100.0%	100.0%	100	
bin TLP_Packet		5382	100.0%	1	
bin DLLP_Packet		8802	100.0%	1	
CVP Packet_backtoback_p		100.0%	100.0%	100	
bin B2B_TLP		3283	100.0%	1	
bin B2B_DLLP		6703	100.0%	1	
bin TLP_DLLP		2099	100.0%	1	
bin DLLP_TLP		2098	100.0%	1	
CVP TLP_Size_p		100.0%	100.0%	100	
illegal_bin TLP_less_than_16_bytes		0	-	-	
bin TLP_MIN_SIZE		2807	100.0%	1	
bin TLP_MAX_SIZE		417	100.0%	1	
bin TLP_other_sizes		2158	100.0%	1	
CVP B2B_Different_Packets_Sizes		100.0%	100.0%	100	
bin B2B_TLP_MAX_SIZE		100	100.0%	1	
bin B2B_TLP_MIN_SIZE		1466	100.0%	1	
bin TLP_DLLP_MAX_SIZE		303	100.0%	1	
bin TLP_DLLP_MIN_SIZE		733	100.0%	1	
bin DLLP_TLP_MAX_SIZE		302	100.0%	1	
bin DLLP_TLP_MIN_SIZE		749	100.0%	1	

Figure 167: Packets Types Covered at TX Downstream Side



TYPE RX_Data_Exchange_D_Cover	Coverage_Model_D2	100.0%	100.0%	100	auto(0)
CVP RX_Data_Exchange_D_Cover::Packet_Type_p	Coverage_Model_D2	100.0%	100.0%	100	
CVP RX_Data_Exchange_D_Cover::Packet_backtobac...	Coverage_Model_D2	100.0%	100.0%	100	
CVP RX_Data_Exchange_D_Cover::TLP_Size_p	Coverage_Model_D2	100.0%	100.0%	100	
CVP RX_Data_Exchange_D_Cover::B2B_Different_Pa...	Coverage_Model_D2	100.0%	100.0%	100	
INST \/PCIe_pkg::Coverage_Model_D2::RX_Data_Ex...		100.0%	100.0%	100	
CVP Packet_Type_p		100.0%	100.0%	100	
bin TLP_Packet		5507	100.0%	1	
bin DLLP_Packet		8652	100.0%	1	
CVP Packet_backtoback_p		100.0%	100.0%	100	
bin B2B_TLP		3407	100.0%	1	
bin B2B_DLLP		6552	100.0%	1	
bin TLP_DLLP		2100	100.0%	1	
bin DLLP_TLP		2099	100.0%	1	
CVP TLP_Size_p		100.0%	100.0%	100	
illegal_bin TLP_less_than_16_bytes		0	-	-	
bin TLP_MIN_SIZE		2910	100.0%	1	
bin TLP_MAX_SIZE		414	100.0%	1	
bin TLP_other_sizes		2183	100.0%	1	
CVP B2B_Different_Packets_Sizes		100.0%	100.0%	100	
bin B2B_TLP_MAX_SIZE		97	100.0%	1	
bin B2B_TLP_MIN_SIZE		1571	100.0%	1	
bin TLP_DLLP_MAX_SIZE		307	100.0%	1	
bin TLP_DLLP_MIN_SIZE		719	100.0%	1	
bin DLLP_TLP_MAX_SIZE		306	100.0%	1	
bin DLLP_TLP_MIN_SIZE		748	100.0%	1	

Figure 168: Packets Types Covered at RX Downstream Side

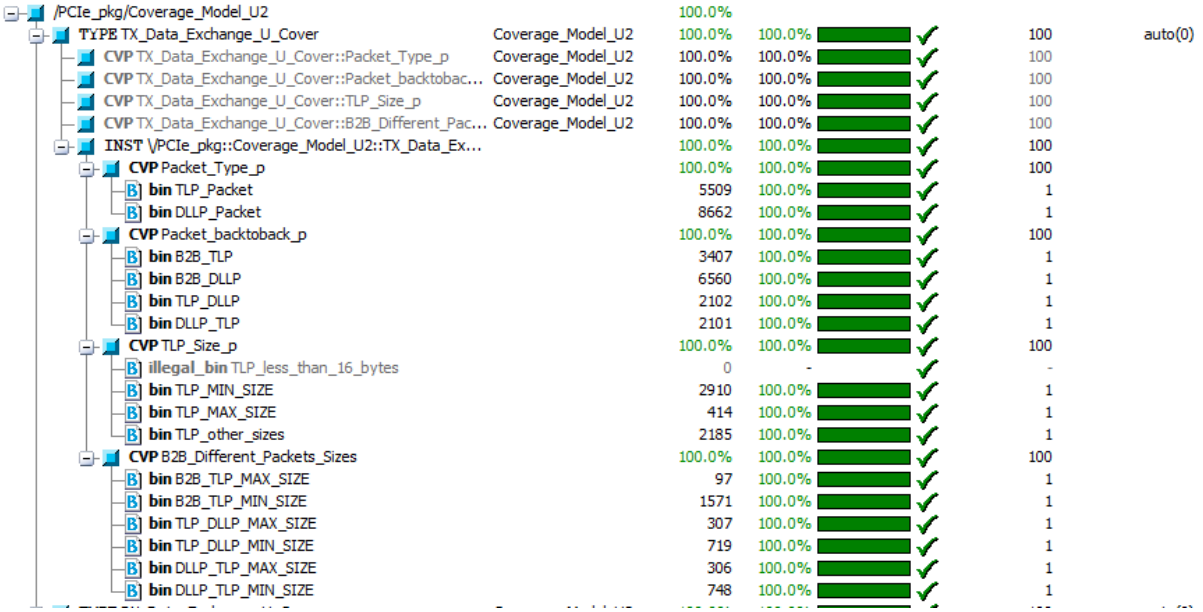


Figure 169: Packets Types Covered at TX Upstream Side

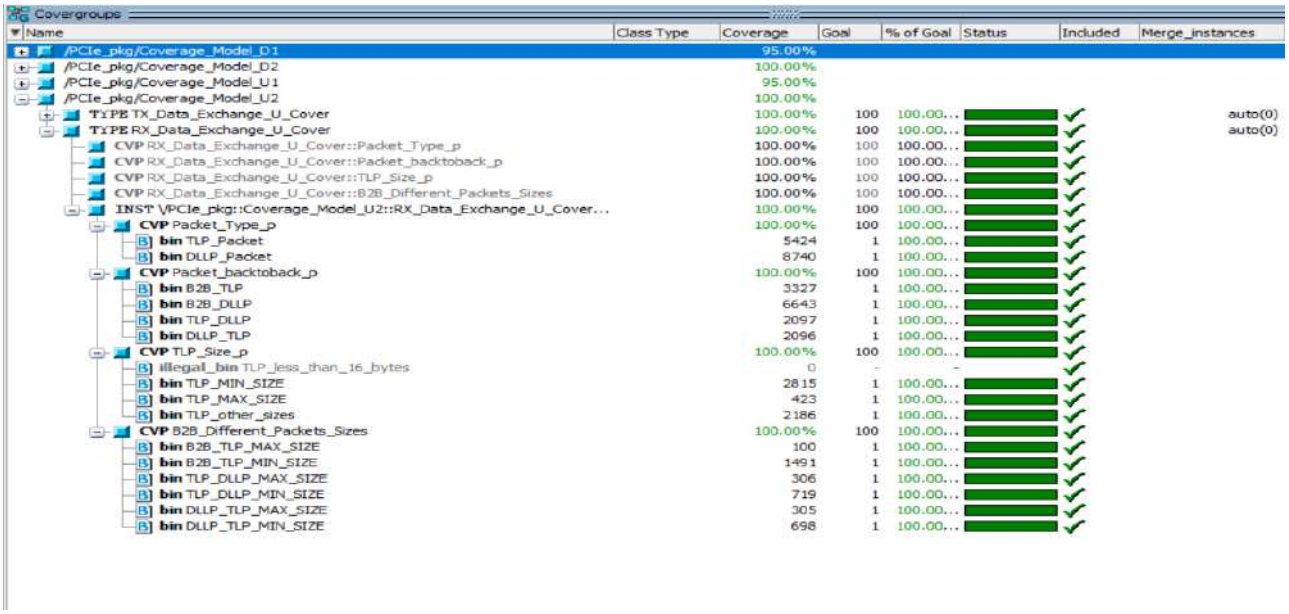


Figure 170: Packets Types Covered at RX Upstream Side

11.1.2 Code Coverage

```
Total Coverage By File (code coverage only, filtered view): 94.2%
```

Figure 171: Total Code Coverage

11.2 LPIF Scoreboard Results

In this part, we present the scoreboard results after transmitting and receiving various TLPs and DLLPs.

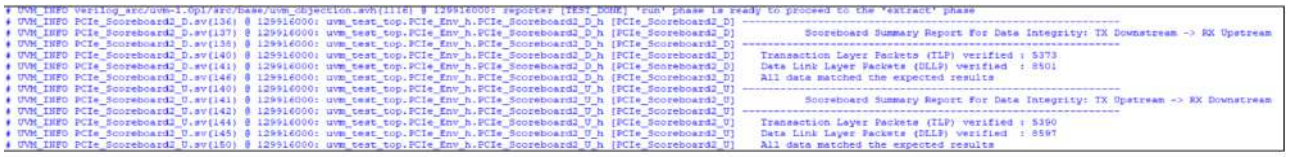


Figure 172: Summary report of LPIF scoreboards in both devices

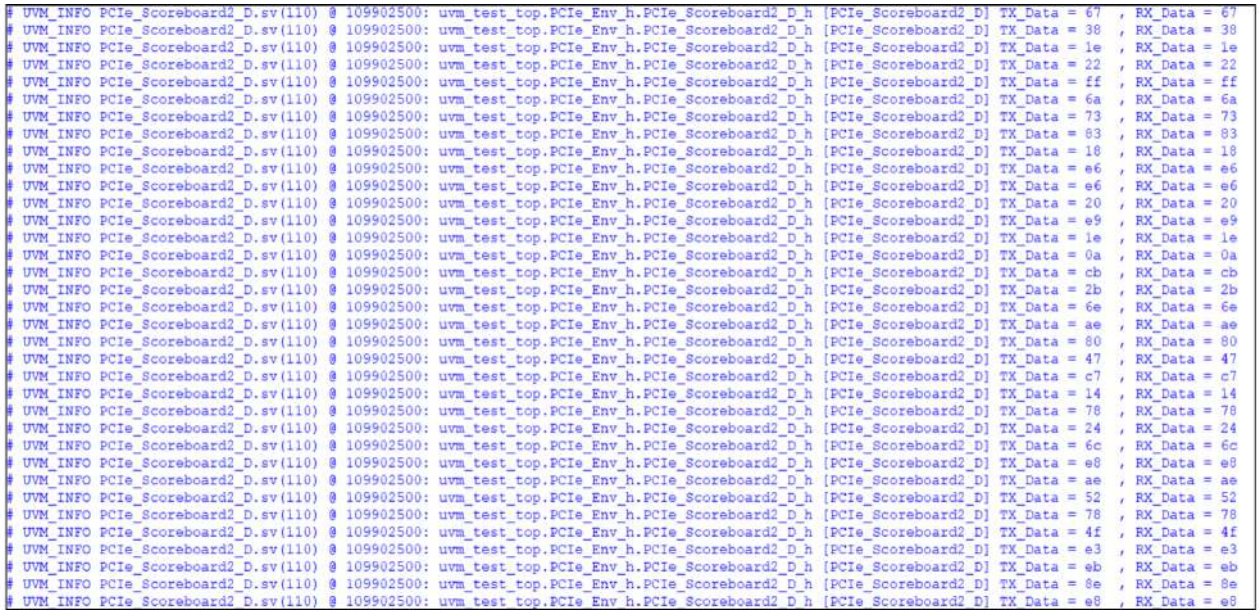


Figure 173: The scoreboard checks the data on the downstream device

```
# UVM_INFO PCIe_Scoreboard2_U.sv(114) @ 110077500: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard2_U_h [PCIe_Scoreboard2_U] TX_Data = ca  , RX_Data = ca
# UVM_INFO PCIe_Scoreboard2_U.sv(114) @ 110077500: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard2_U_h [PCIe_Scoreboard2_U] TX_Data = 80  , RX_Data = 80
# UVM_INFO PCIe_Scoreboard2_U.sv(114) @ 110077500: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard2_U_h [PCIe_Scoreboard2_U] TX_Data = e7  , RX_Data = e7
# UVM_INFO PCIe_Scoreboard2_U.sv(114) @ 110077500: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard2_U_h [PCIe_Scoreboard2_U] TX_Data = 47  , RX_Data = 47
# UVM_INFO PCIe_Scoreboard2_U.sv(114) @ 110077500: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard2_U_h [PCIe_Scoreboard2_U] TX_Data = 2d  , RX_Data = 2d
# UVM_INFO PCIe_Scoreboard2_U.sv(114) @ 110077500: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard2_U_h [PCIe_Scoreboard2_U] TX_Data = c1  , RX_Data = c1
# UVM_INFO PCIe_Scoreboard2_U.sv(114) @ 110077500: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard2_U_h [PCIe_Scoreboard2_U] TX_Data = b4  , RX_Data = b4
# UVM_INFO PCIe_Scoreboard2_U.sv(114) @ 110077500: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard2_U_h [PCIe_Scoreboard2_U] TX_Data = 47  , RX_Data = 47
# UVM_INFO PCIe_Scoreboard2_U.sv(114) @ 110077500: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard2_U_h [PCIe_Scoreboard2_U] TX_Data = 26  , RX_Data = 26
# UVM_INFO PCIe_Scoreboard2_U.sv(114) @ 110077500: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard2_U_h [PCIe_Scoreboard2_U] TX_Data = fa  , RX_Data = fa
# UVM_INFO PCIe_Scoreboard2_U.sv(114) @ 110077500: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard2_U_h [PCIe_Scoreboard2_U] TX_Data = 2d  , RX_Data = 2d
# UVM_INFO PCIe_Scoreboard2_U.sv(114) @ 110077500: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard2_U_h [PCIe_Scoreboard2_U] TX_Data = e8  , RX_Data = e8
# UVM_INFO PCIe_Scoreboard2_U.sv(114) @ 110077500: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard2_U_h [PCIe_Scoreboard2_U] TX_Data = c5  , RX_Data = c5
# UVM_INFO PCIe_Scoreboard2_U.sv(114) @ 110077500: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard2_U_h [PCIe_Scoreboard2_U] TX_Data = 5a  , RX_Data = 5a
# UVM_INFO PCIe_Scoreboard2_U.sv(114) @ 110077500: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard2_U_h [PCIe_Scoreboard2_U] TX_Data = 87  , RX_Data = 87
# UVM_INFO PCIe_Scoreboard2_U.sv(114) @ 110077500: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard2_U_h [PCIe_Scoreboard2_U] TX_Data = 06  , RX_Data = 06
# UVM_INFO PCIe_Scoreboard2_U.sv(114) @ 110077500: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard2_U_h [PCIe_Scoreboard2_U] TX_Data = 6e  , RX_Data = 6e
# UVM_INFO PCIe_Scoreboard2_U.sv(114) @ 110077500: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard2_U_h [PCIe_Scoreboard2_U] TX_Data = a8  , RX_Data = a8
# UVM_INFO PCIe_Scoreboard2_U.sv(114) @ 110077500: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard2_U_h [PCIe_Scoreboard2_U] TX_Data = 98  , RX_Data = 98
# UVM_INFO PCIe_Scoreboard2_U.sv(114) @ 110077500: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard2_U_h [PCIe_Scoreboard2_U] TX_Data = 21  , RX_Data = 21
# UVM_INFO PCIe_Scoreboard2_U.sv(114) @ 110077500: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard2_U_h [PCIe_Scoreboard2_U] TX_Data = 4c  , RX_Data = 4c
# UVM_INFO PCIe_Scoreboard2_U.sv(114) @ 110077500: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard2_U_h [PCIe_Scoreboard2_U] TX_Data = 77  , RX_Data = 77
# UVM_INFO PCIe_Scoreboard2_U.sv(114) @ 110077500: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard2_U_h [PCIe_Scoreboard2_U] TX_Data = b9  , RX_Data = b9
# UVM_INFO PCIe_Scoreboard2_U.sv(114) @ 110077500: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard2_U_h [PCIe_Scoreboard2_U] TX_Data = 1b  , RX_Data = 1b
# UVM_INFO PCIe_Scoreboard2_U.sv(114) @ 110077500: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard2_U_h [PCIe_Scoreboard2_U] TX_Data = f0  , RX_Data = f0
# UVM_INFO PCIe_Scoreboard2_U.sv(114) @ 110077500: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard2_U_h [PCIe_Scoreboard2_U] TX_Data = 54  , RX_Data = 54
# UVM_INFO PCIe_Scoreboard2_U.sv(114) @ 110077500: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard2_U_h [PCIe_Scoreboard2_U] TX_Data = 74  , RX_Data = 74
# UVM_INFO PCIe_Scoreboard2_U.sv(114) @ 110077500: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard2_U_h [PCIe_Scoreboard2_U] TX_Data = f5  , RX_Data = f5
# UVM_INFO PCIe_Scoreboard2_U.sv(114) @ 110077500: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard2_U_h [PCIe_Scoreboard2_U] TX_Data = c1  , RX_Data = c1
# UVM_INFO PCIe_Scoreboard2_U.sv(114) @ 110077500: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard2_U_h [PCIe_Scoreboard2_U] TX_Data = 42  , RX_Data = 42
# UVM_INFO PCIe_Scoreboard2_U.sv(114) @ 110077500: uvm_test_top.PCIe_Env_h.PCIe_Scoreboard2_U_h [PCIe_Scoreboard2_U] TX_Data = f8  , RX_Data = f8
```

Figure 174: The scoreboard checks the data on the upstream device

11.3 UVM Summary Report

```
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO : 1068
# UVM_WARNING :    0
# UVM_ERROR :    0
# UVM_FATAL :    0
# ** Report counts by id
# [Adapter]        3
# [Coverage_Model_D1]        3
# [Coverage_Model_D2]        3
# [Coverage_Model_U1]        3
# [Coverage_Model_U2]        3
# [DLLP_Seq_TX_MASTER_D]        2
# [DLLP_Seq_TX_MASTER_U]        2
# [Inject_Error_vseq]       15
# [LTSSM1_U_Driver]       28
# [LTSSM2_D_Driver]        1
# [LTSSM2_U_Driver]        1
# [PCIe_Scoreboard1_D]      169
# [PCIe_Scoreboard1_U]      169
# [PCIe_Scoreboard2_D]        6
# [PCIe_Scoreboard2_U]        6
# [PCIe_Test]       10
# [Questa UVM]        2
# [RNTST]        1
# [RX_Slave_D_Driver]        3
# [RX_Slave_D_Monitor]      155
# [RX_Slave_U_Driver]        3
# [RX_Slave_U_Monitor]      153
# [TEST_DONE]        1
# [TLP_Seq_TX_MASTER_D]        1
# [TLP_Seq_TX_MASTER_U]        1
# [TX_Master_D_Monitor]        3
# [TX_Master_U_Agent]        2
# [TX_Master_U_Monitor]        3
# [TX_Slave_D_Monitor]      159
# [TX_Slave_U_Monitor]      157
```

Figure 175: UVM Summary Report

11.4 Bugs Summary

Bug ID	Description
Bug_1	Mismatch between Scrambler and Descrambler on the upstream and downstream devices, leading to misalignment.
Bug_2	COM character should always bypass scrambling and descrambling, regardless of whether it's a K-character or D-character.
Bug_3	In Gen1 or Gen2, when attempting to transmit TS1 or TS2 during the L0 state, the Ordered Sets (OS) are being descrambled, which is incorrect behavior.
Bug_4	Incorrect state transition from Phase 1 to Rx_Recovery.RcvrLock after equalization in the Main LTSSM.
Bug_5	Tx LTSSM issue: OS generator sends TS1 with EC=00 when the connected device is downstream, which is only valid for upstream direction.
Bug_6	At the start of Equalization, the upstream device must initiate by sending TS1 with EC=00.
Bug_7	In Phase 1, after the downstream device sends TS1 with EC=00, it should immediately exit the equalization process and notify upper layers of completion. However, the current design waits for TS1 with EC=01 from upstream, which is incorrect.
Bug_8	When in Phase 1, if the condition for transitioning to Recovery.RcvrLock is met, the design should go to that state. Instead, it transitions to an incorrect state in the Main LTSSM.
Bug_9	Missing linkup signal in the RX block: Certain RX functionalities depend on this signal, but it's not present in the current design ports.
Bug_10	In Gen3/4/5 data transfer, the asynchronous header should reference only the first symbol of the data block, but the RTL treats the en ↓ lock as part of the header.

Figure 176: Bugs Summary Part 1

Bug_11	In Gen3/4/5, the length field in the first block of the packet is not correctly encoded, which affects proper data interpretation.
Bug_12	Speed mismatch between Device1 and Device2: The link fails to retrain to the highest common speed, as expected per PCIe spec.
Bug_13	The LTSSM made transition to the config state although polling config condition is not met.
Bug_14	The LTSSM made transition to the Configuration.Idle state although Configuration.Complete condition is not met.
Bug_15	FS and LF are not assigned to output port in downstream TX_LTSSM, causing TX data to become xxxx.
Bug_16	The 32-bit LFSR for Gen3_4_5 updates on posedge, while LMC data updates on negedge, leading to outputting TX data twice per cycle.
Bug_17	Data in Recovery.Idle are not scrambled in both RX and TX.
Bug_18	TXs send incorrect SDS OS (Gen3 SDS instead of Gen5).
Bug_19	TXs send incorrect SKP OS (Gen3 SKP instead of Gen5).
Bug_20	TXs send SDS at L0.
Bug_21	The scrambler isn't synchronized with the descrambler in L0 due to an incorrect LFSR reset, impacting sequence generation.
Bug_22	Remaining data of TLP is not received; only the part containing STP is received.
Bug_23	When sending back-to-back max-sized TLPs, the remaining data of the first TLP is incorrect due to constant high valid signal affecting the length counter.

Figure 177: Bugs Summary Part 2

Bug_24	When sending four minimum-sized TLPs in one transfer, `tlp_start` incorrectly marks the start of the fourth TLP.
Bug_25	Transition from Detect.Quiet to Detect.Active occurs immediately due to Rx Electrical Idle never deasserting and timeout incorrectly set to 0 ms.
Bug_26	RX flags TLP as bad if last byte matches 0xC0, instead of requiring full 4-byte EDB pattern.
Bug_27	RX only processes the first DLLP if multiple are sent back-to-back, as detection fails after the first.
Bug_28	TX wrongly counts DLLP as a TLP when both are transmitted in one transfer, inserting unintended STP.
Bug_29	At Recovery.CFG, after timeout, the environment returns to Detect but RTL does not.
Bug_30	Timeout isn't asserted after 48ms exactly.
Bug_31	At Recovery.Speed, after timeout, the environment returns to Detect but RTL does not.
Bug_32	At Phase 1, after timeout, the environment enters Recovery.Speed but RTL does not.
Bug_33	RTL doesn't go to Detect after 12ms as required.
Bug_34	In Detect state, TxDetectRx remains high instead of deasserting.
Bug_35	After returning to Detect from L0 and detecting RX TS, next state's TS has incorrect ID in symbol 6.
Bug_36	After returning from states beyond L0 to Detect, the system gets stuck in Detect.
Bug_37	After timeout and returning to Detect to retry link-up, the upstream device fails to progress past L0.
Bug_38	In Polling.Active, DUT transitions forward on timeout without receiving enough TS. Fixed in RTL.
Bug_39	In Polling.Config, DUT transitions forward on timeout without receiving enough TS. Fixed in RTL.
Bug_40	In Polling.Active, DUT accepts corrupted TS1s due to insufficient validation. Fixed in RTL.
Bug_41	TS1 validation was only checked on Lane 0; spec requires valid TS1s on all lanes.

Figure 178: Bugs Summary Part 3

12 References

References

[1] Accellera Systems Initiative, *UVM 1.2 User Guide*. Available: https://accellera.org/downloads/standards/uvm

[2] Bergeron, J. et al., *Writing Testbenches Using SystemVerilog*, 2nd ed., Springer, 2006.

[3] Spear, C., and Tumbush, G., *SystemVerilog for Verification: A Guide to Learning the Testbench Language Features*, 3rd ed., Springer, 2012. Available: https://link.springer.com/book/10.1007/978-1-4614-0715-7

[4] Accellera Systems Initiative, *Universal Verification Methodology (UVM) Overview*. Available: https://accellera.org/activities/vip/uvm

[5] IEEE Standards Association, *IEEE 1800.2-2020 - IEEE Standard for UVM*. Available: https://standards.ieee.org/ieee/1800.2/

[6] Doulos, *Introduction to UVM*. Available: https://www.doulos.com/knowhow/systemverilog/uvm-intro/

[7] Mentor Graphics, *UVM Cookbook - Universal Verification Methodology*, Verification Academy. Available: https://verificationacademy.com/cookbook/uvm-universal-verification-methodology/

[8] A. G. Warade, N. N. Jambhulkar, and P. V. Chavan, "Design and Implementation of PCI Express Protocol using VHDL," *International Research Journal of Engineering and Technology (IRJET)*, vol. 4, no. 10, Oct. 2017. Available: https://www.irjet.net/archives/V4/i10/IRJET-V4I1008.pdf

[9] Intel Corporation, "PHY Interface for the PCI Express* and SATA Architectures with PIPE (PHY Interface for the PCI Express Architecture)," Revision 0.9, Sept. 2002. Available: https://www.intel.in/content/dam/doc/white-paper/phy-interface-pci-express-sata3-specification-v09.pdf

[10] R. Guo, L. Zhang, Y. Wang, and W. Hu, "Design and Implementation of PIPE Interface for PCI Express PHY," *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2020. DOI: 10.1109/ISCAS45731.2020.9180912. Available: https://ieeexplore.ieee.org/document/9154176

[11] PCI-SIG, *PCI Express Base Specification Revision 5.0, Version 1.0*, May 2019. Available: https://picture.iczhiku.com/resource/eetop/SYkDTqhOLhpUTnMx.pdf

[12] Wikipedia contributors, "PCI Express," *Wikipedia, The Free Encyclopedia*. Available: https://en.wikipedia.org/wiki/PCI_Express