



UVM-BASED FUNCTIONAL VERIFICATION OF PCIE PROTOCOL USING A REUSABLE VERIFICATION AGENT

Usman Ali Khan Nyazi¹, Dr. M.N. Giri Prasad², Nagari Naveen Kumar³

¹M.tech VLSISD, Department of ECE, JNTUA College of Engineering Anantapur, Andhra Pradesh, India,
usmannnyazi112@gmail.com.

²Adjunct Professor, Department of ECE, JNTUA College of Engineering Anantapur, Andhra Pradesh, India.

³Assistant Professor, Department of ECE, JNTUA College of Engineering Anantapur, Andhra Pradesh, India.

Abstract—Peripheral Component Interconnect express (PCIe) is a layered and serial interconnect protocol for linking processors, memory, and peripheral devices, contains the transaction, data link, and physical layers. As PCIe IP cores advance through higher generations (Gen1 through Gen5) and adopt architectural variants such as non-FLIT mode, ensuring that an endpoint implementation correctly interprets and responds to the full range of Transaction Layer Packets (TLPs) becomes increasingly challenging. Existing verification approaches, often validate only nominal transaction flows, leaving gaps in error scenarios, out-of-bound transactions and support across multiple generations. This work addresses this gap by developing a reusable UVM-based verification environment for systematically verifying PCIe endpoint functionality across Gen1 through Gen5. The environment is built around two UVM agents: an active agent, that generates and drives Memory, I/O, Configuration, and message TLPs to the endpoint over a 256-bit AXI4 interface, and a passive agent, whose monitor observes and captures endpoint responses arriving over the PCIe lanes and the AXI interface. Beyond exercising legal transaction sequences, the environment injects errors in the TLPs to verify that the endpoint behavior conforms to the PCIe specification. Functional coverage was defined across all supported TLP types to quantitatively assess verification completeness. The environment achieved 100% functional coverage across all TLP types, and completion status

received from the endpoint. The verification environment is reusable across PCIe Gen1 through Gen5 endpoint designs, offering a scalable foundation for future verification IP development.

Keywords— Peripheral Component Interconnect Express, Transaction Layer Packet, Functional Coverage, Universal Verification Methodology, Data Link Layer.

I. INTRODUCTION

With the increase in the speed of data transfer among the newer generations of PCIe which is difficult for verifying the successful transaction flow. PCIe uses a layered architecture consisting of transaction layer, data link layer, and physical layer, with each layer performing distinct operations and interacting with others during data transfer.

The Transaction Layer builds the actual packet, called a TLP, which carries memory, I/O, configuration, or message data. The Data Link layer adds sequence numbers and CRC and handles ACK/NAK, so packets don't get lost. The Physical layer then sends this out over the lanes as raw bits. In this work, the Root Complex (RC) is fully modelled inside the UVM testbench, and the Endpoint (EP) is BRAM acts as the design under test.

The flow of the packet moves down through these layers on the RC side, crosses the link, and comes back up through the same layers on the EP side as illustrated in Figure 1.

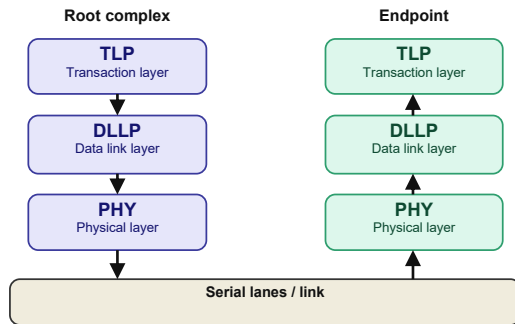


Fig. 1 Layered Protocol interaction between Root Complex and Endpoint.

Researchers have developed Verification environments for PCIe using UVM, targeting different layers, protocol versions, and verification objectives, ranging from Transaction and Data Link Layer co-verification to Physical Layer signalling and SoC-level integration frameworks. While these efforts have advanced PCIe verification considerably, each address only a portion of the requirements relevant to complete, error-aware, coverage-driven endpoint verification environment. A detailed review of this prior work, along with the specific gaps it leaves open, is presented in Section II.

Based on these observations, the present work targets an environment that combined commercial hard IP verification, full Root Complex modelling, a clear active and passive agent separation, systematic error injection across all TLP categories, and complete functional coverage closure which is not fully included in existing literature.

II. RELATED WORK

Several researchers have developed verification environments for PCIe, focusing on specific layers targeting a particular generation. Jagtap et al. [1] developed a UVM based verification IP for the Transaction and Data Link layers, focused on successful transaction flow from the UVM testbench to the EP, but did not cover error scenarios. Manohar and Sarkar

[2] built UVM-based verification for the Transaction layer targeting Gen5 but did not test error cases, out-of-bound transactions, and provided no coverage report. Cao and Li [3] introduce an efficient UVM based verification framework for PCIe at the SoC level using YAML configurations and Python scripts to reduce the automation time at the System-on-Chip level. Kumar and Praveen [4] proposed a verification environment for Gen5 targeting the Data Link layer, validating Link CRC computation and sequence number matching from the root complex to the endpoint, but restricted their scope to the Data link layer of PCIe 5.0. Nema et al. [5] verified Transaction and Data Link layer of PCIe 5.0 against a hypothetical SSD slave device.

Nirupama and Narasimha babu [6] proposed a System Verilog and UVM-based testbench for verification of PCIe IP at the subsystem and integration with the SoC level. They reduced the testing time for PCIe by 75% per PCIe IP by using the Register Abstraction Layer-based APIs rather than concentrating on individual layers of the PCIe while ignoring error test scenarios and coverage reports. Santhi and Chellaswamy [7] introduced a hybrid Random Forest reinforcement learning framework for PCIe 6.0 verification, using machine learning to drive stimulus to the PCIe and achieved functional coverage of 95-97%. Rohilla et al. [8] and Vaidya et al. [9] proposed a verification environment for the Physical layer focusing on MAC-PHY signalling with the PIPE interface, and data scrambling, 8b/10b encoding, and LTSSM behaviour, respectively.

Most closely related to the present work, Sree and Subhas [10] developed a UVM sequence library specifically for non-FLIT PCIe Transaction Layer packets, supporting Memory, Configuration, Message, and Completion transaction categories with constrained-random stimulus generation;

however, this work address stimulus generation alone does not report error injection, malformed TLP testing, or functional coverage results. Kuruba and Reddy [11] proposed a modular UVM-based scoreboard architecture for protocol aware verification of non-FLIT PCIe transactions, using a PIPE based root port to endpoint flow with LTSSM synchronized monitoring and reference model-based checking; this work centres on the checking and scoreboard side of verification but similarly does not incorporate systematic error injection or report complete coverage closure. Thomas et al. [12] presented a pre-silicon framework for usage model performance verification of PCIe interfaces in server processor SoCs, shifting bandwidth and latency validation from post-silicon to pre-silicon; unlike the present work, this framework targets performance characterization rather than functional protocol compliance under nominal and fault conditions.

The existing work focuses on developing a verification environment for individual layers of PCIe, targeting specific generations, and using directed stimulus unlike randomization, focused error injection at a single layer [4], and performance verification [12]. However, none of the existing work develops the reusable verification environment that supports Gen1 to Gen5 using reusable UVM agents and functional coverage driven verification and testing all nominal transactions like memory, Configuration, messages TLPs with error injection and out of bound memory address transactions cases, the proposed work addresses these gaps by integrating these capabilities into reusable verification environment.

III. PROPOSED METHOD

This section describes the verification environment developed for this work,

starting with the overall architecture and the path a transaction takes through it, and then going into the individual pieces the active and passive agents, the packet formats they work with, the scoreboard, and the coverage model in enough detail that the environment can be rebuilt from this description.

A. Overview of the Verification environment.

The block diagram of the environment used in this work is shown (Figure 2). There are two halves connected through a PCIe link: the Root Complex side, modeled entirely inside a UVM testbench, and the Endpoint side, which is the AMD UltraScale+ PIO example design acting as the DUT [13].

The connection between the testbench and the DUT is an AXI4-Stream interface, and since the signal names on this interface come up throughout the rest of this section, it is worth explaining what they mean before going further. tdata carries the actual payload being transferred, 256 bits per beat in this design. tvalid is asserted by the sender to indicate that data is present, and tready is asserted by the receiver to indicate it can accept data; a transfer only happens on a clock edge where both signals are high together. tkeep marks which bytes of the current beat are valid, since a beat is not always completely full. tlast marks the final beat of a packet so the receiver knows where it ends. tuser is a sideband bus carrying protocol information that does not fit into the main data path, such as byte-enable flags and sequence numbers. The prefix on each signal name tells you which direction it travels: s_axis_rq_* is the Requester Request path, over which the testbench sends TLPs to the DUT, and m_axis_rc_* is the Requester Completion path, over which the DUT sends completions back. With that established, the flow of transaction through the environment

is tracked block by block. It starts at the test class, which doesn't drive any signals itself- its only job is to pick a sequence and start it.

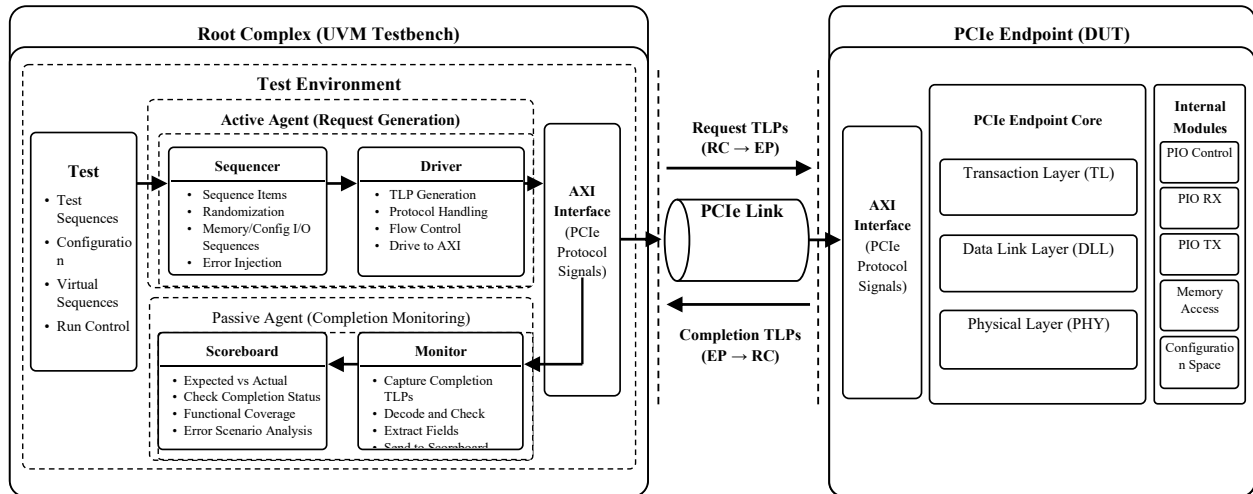


Fig. 2 Block Diagram of the proposed UVM-based verification environment.

The sequence builds a `pcie_tlp_item` describing the transaction (memory read or write, configuration read or write) and sends it to the sequencer which forwards it to the driver through the usual `get_next_item()` / `item_done()` handshake. The driver is where the request actually takes shape: depending on the request type carried by the item. It assembles the matching TLP header, places it on the `s_axis_rq_tdata` and waits for `s_axis_rq_tready` before the beat is the considered sent. This crosses into the DUT through `pcie_if` virtual interface, which is simply the binding between the testbench's signals and the AXI4-Stream ports on the hard IP core. From there the transaction moves through the PIPE-level link modeled inside board, where the Root Port and Endpoint models are connected directly through Xilinx pipe signals rather than an actual serial link, since the whole thing runs in simulation.

On the Endpoint side, what happens next depends on the request type. Configuration reads and writes never reach the PIO application at all — the hard IP core

answers `CfgRd0/1` and `CfgWr0/1` out of its own internal configuration-space logic, and the `cfg_mgmt_*` ports exposed to the PIO

application are simply tied off. Memory requests take a different route: they are handed to the PIO RX engine inside `pio_ep`, which pulls the address and byte enables out of the header and either writes to the on-chip memory or starts a read. That memory is four Block RAM banks, giving 8KB total, addressed relative to whichever BAR the request falls into. Once the RX engine has resolved the request, the PIO TX engine takes over and builds the response a Completion, a Completion with Data, or, when the address does not decode to anything programmed, an Unsupported Request completion.

The completion leaves the DUT on `m_axis_rc_tdata`, crosses back over the same simulated link, and arrives on the RC side at the monitor. The monitor never drives anything on the interface. `m_axis_rc_tready` is held high for the entire simulation so the DUT is never stalled waiting on the testbench it only watches tuser for the start-of-packet indication, extracts the completion fields as the beats arrive, and once `tlast` appears, packages the result into an item and forwards it through

its analysis port. That item reaches the scoreboard, which is where the pass or fail decision is actually made, by comparing the completion data against whatever was written earlier for that same address. Put together, a single write-then-read cycle moves through the environment as: sequence, driver, AXI4-Stream request path, hard IP or PIO application, on-chip memory, PIO TX engine, AXI4-Stream completion path, monitor, scoreboard. Everything past the driver is either fixed DUT behavior or passive observation, which is what keeps the active and passive sides of the environment cleanly separated stimulus only ever originates in one place.

B. Transaction Layer Packet Frame

Every transaction in this environment is expressed as a descriptor placed on the 256-bit AXI4-Stream data bus, accompanied by sideband information on tuser. Two descriptor formats are involved: the request descriptor constructed by the driver and transmitted on s_axis_rq_tdata, and the completion descriptor received on m_axis_rc_tdata and decoded by the monitor. Because these layouts determine how both agents are implemented, they are defined here before the agents themselves are described.

C. Request Transaction Layer Packet

All request types share a common descriptor structure occupying the lower 128 bits of the transmitted beat, with the upper 128 bits used for payload in the case of write transactions as shown in figure 3.

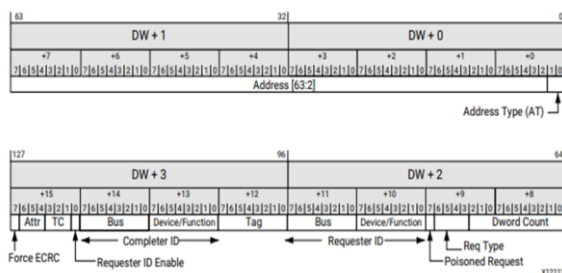


Fig. 3 Request TLP frame.

The request type field determines how the remainder of the descriptor is interpreted. For memory transactions, the full 64-bit address is significant, and the driver selects between 32-bit and 64-bit addressing.

with the upper address word driven to zero when the condition does not hold. For configuration transactions, only the register offset is meaningful, and it is placed in bits [11:2] of the address field, with the payload length fixed at one Double Word regardless of the value carried by the sequence item.

The number of Double Words carried by a transaction follows directly from the payload size.

For write transactions exceeding four Double Words, the payload cannot be accommodated in a single beat, since the first beat is partly occupied by the descriptor.

D. Completion Transaction Layer Packet

Completions returned by the Endpoint occupy a three Double Word descriptor (96 bits), followed by payload data where applicable. Completion packet structure is shown in figure 3.

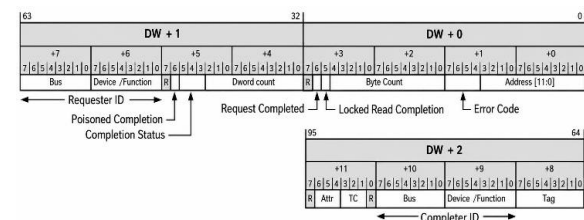


Fig. 4 Completion TLP.

The completion status field carries the outcome of the transaction and takes one of four values: 000 for Successful Completion, indicating the request was serviced normally; 001 for Unsupported Request, returned when the target address does not decode to any enabled region; 010 for Configuration Request Retry Status, returned when the Endpoint is not yet ready to service

configuration accesses; and 100 for Completer Abort. All four are decoded and counted separately by the monitor, since the distinction between them is what allows a negative-path result to be recognized as correct behavior rather than a failure.

E. Active Agent

The active agent is responsible for generating the transactions request to the AXI interface which will be sent to the endpoint over the link. It contains sequencer, and driver. The flow of transactions from the sequencer to the driver and driver to the

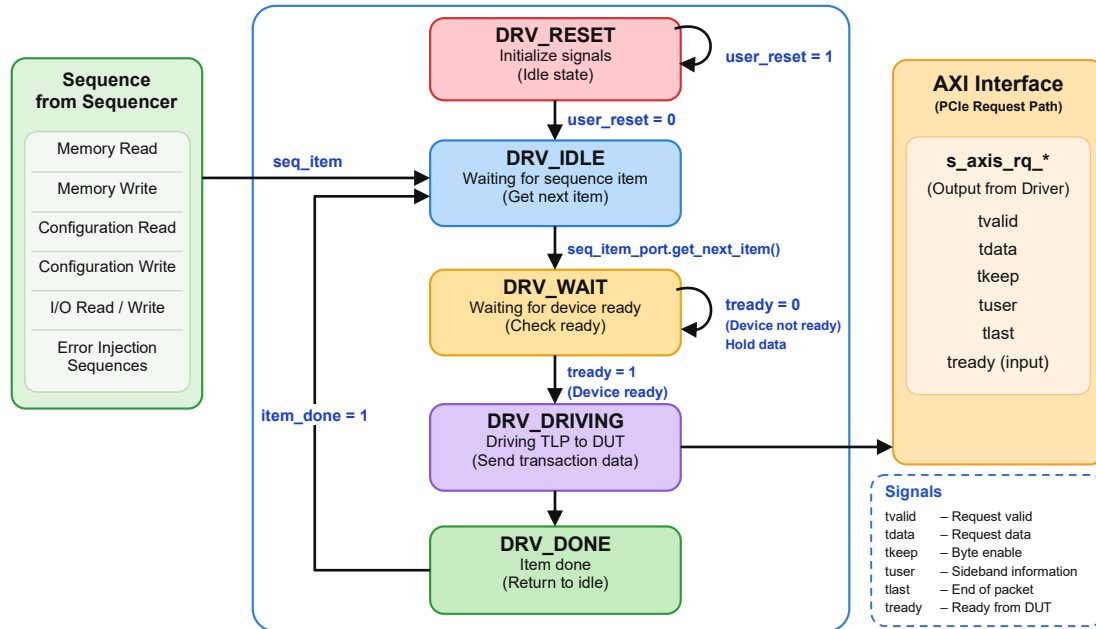


Fig. 5 Block diagram of the active agent and driver state machine.

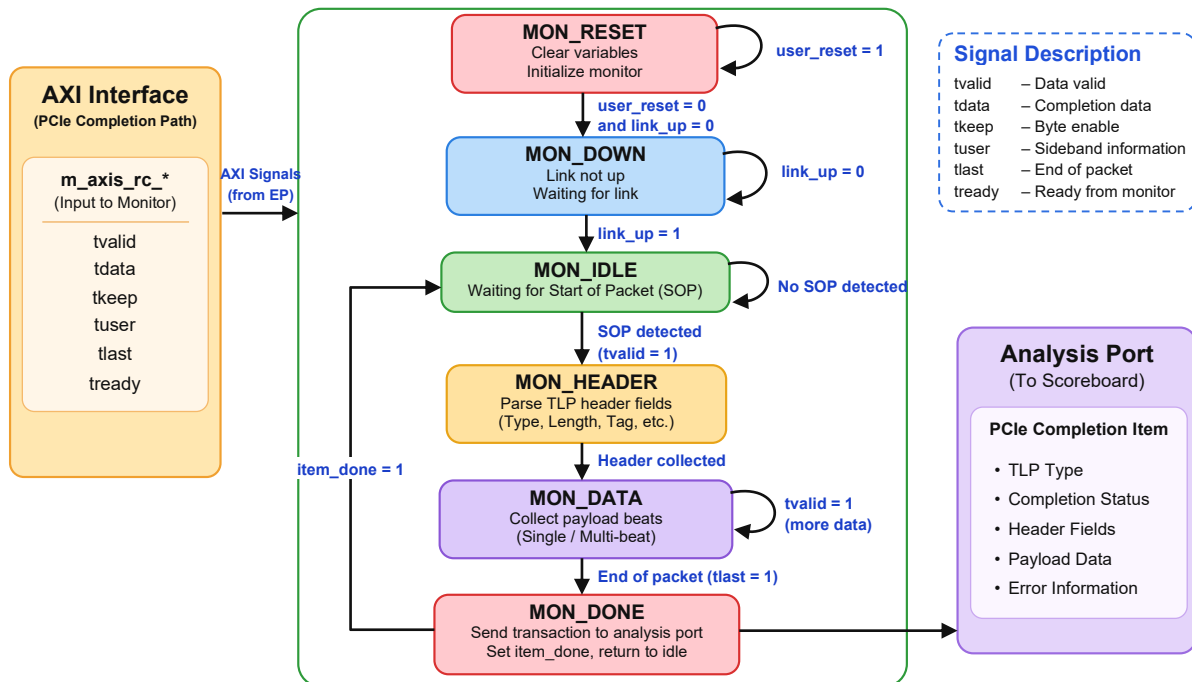


Fig. 6 Block diagram of the passive agent and monitor state machine.

interface is illustrated in Figure 5. The figure 5. Also include State machine of the driver as shown there are five states of the driver.

When the device reset is high it doesn't send any transactions to the interface. When reset is low the driver is in idle state waiting for the request TLPs from the sequencer. When the sequence is ready to send from the sequencer to the driver it captures the sequences and changes to driving state, before the driver send the request TLPs to the Endpoint over the link it checks whether the endpoint is ready to accept the incoming requests from the driver over the link. Then driver drives all the TLPs and shifted to done state. As the cycle repeats for all the incoming TLPs.

The request interface contains the AXI signals such as tdata which hold the request TLP fields in the 256-bit width, tvalid remains high when transmitting the sequences and tready is signal coming from the endpoint states that the device is ready or not, tuser carries the side band information of the sequence and tkeep is high when transmitting the sequences.

Sequencer acts as queue, stores all the incoming sequences coming from the sequence and stores it and send one by one as first come and first serve basis.

F. *Passive Agent*

The passive agent is responsible for collecting the response from the endpoint over the link. It contains only monitor. It doesn't drive any information to the device it only collects the data from the endpoint. The state diagram of the monitor and the flow of the transaction from the interface to monitor is shown in figure 6. As seen in figure 6, there are 6 states for the monitor. The response are collected from the m_axis_rc_tdata interface of the AXI it includes the side band information signals as shown in figure 6.

During the reset high the monitor was in reset state. When reset released it is waiting for the link establishment between the endpoint and UVM testbench which acts as a root complex. After link up the monitor is waiting for the response from the endpoint for the request sent from the drive. When the start of packet (SoP) is detected, the monitor starts collecting the data

from the AXI interface. The collected response from the endpoint is sent to the scoreboard using the UVM analysis port.

Scoreboard consists of reference model which stores the address and tag and requester ID from the transmitting TLP which are driven from the driver and collects the responses from the monitor and compares both requester ID and completer ID whether same or not, based on the tag field of the both the frame and reports whether it received data as expected and status the completion whether successful completion received with data or completion abort or unsupported request.

G. *Functional Coverage*

Functional coverage was implemented to measure the completeness of the verification environment with respect to the PCIe Transaction Layer Packet (TLP) scenarios exercised during simulation. The coverage model was developed using System Verilog cover groups and cover points within the UVM verification environment.

IV. RESULTS AND DISCUSSION

This section presents the results of the proposed work. Verification environment was build using System Verilog programming language using Universal Verification Methodology in the Xilinx Vivado 2025.2 version tool. In this work, the UVM performs as root complex behavior responsible for generating TLPs through the driver.

We first targeted PCIe of generation 5 later we also tested the same verification environment for previous generations. The results demonstrated the successful verified the verification environment for PCIe Gen1 through Gen5. Figure 7 shows the simulation waveform for the driving TLPs through the driver on the AXI4 requester interface. As shown in figure 7 the s_axis_rq_tdata is 256-bit width where lower 128-bits occupied by header TLP as discussed in section 3.2. Configuration Type0 with address 0xd0 is sent to know the device ID and vendor ID of the endpoint. The completion for the request driven is received on the completion interface of the AXI interface as shown in figure 8. It received completion with

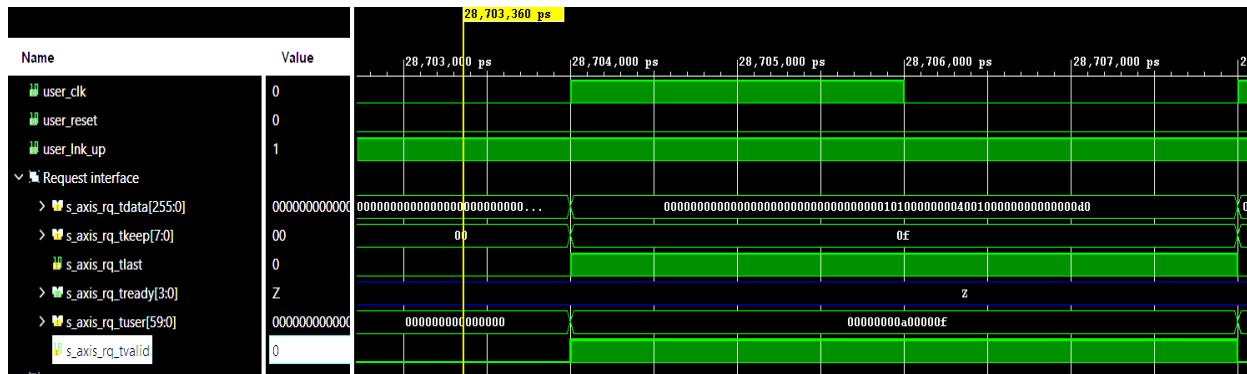


Fig. 7 Simulation waveform of the request interface driving configuration read request

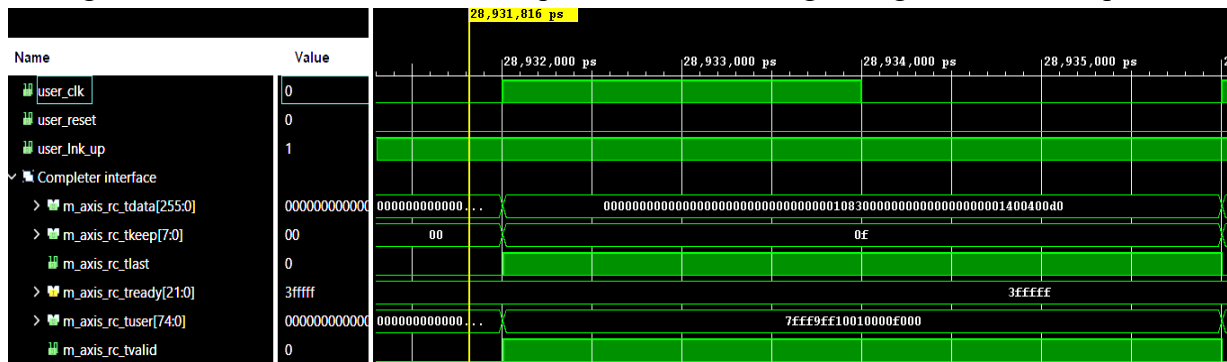


Fig. 8 Simulation waveform of the completion interface received completion with data.

```

=====
PCIe PIO Write-Readback Test START
=====
agent.drv [RQ_DRV] [DRV-FSM]   DRV_RESET | TLP packets_sent=0 | link_up=x | reset=x
agent.mon [RC_MON] [MON-FSM]   MON_RESET | Completions Received =0 | link_up=x | tvalid=x | SOP bit =x
agent.mon [RC_MON] RC Monitor started - tready held HIGH permanently
rc_agent.mon [RC_MON] [MON-FSM] MON_IDLE | Completions Received =0 | link_up=x | tvalid=0 | SOP bit =0
rq_agent.drv [LTSSM] [LTSSM] 0x00 : Detect.Active | lnk_up=x | reset=x | drv_fsm= DRV_RESET
rq_agent.drv [RQ_DRV] [DRV-FSM] DRV_IDLE | TLP packets_sent=0 | link_up=x | reset=x
v.rc_agent.mon [RC_MON] [MON-FSM] MON_RESET | Completions Received =0 | link_up=x | tvalid=0 | SOP bit =0

env.rq_agent.drv [LTSSM] [LTSSM] 0x01 : Detect.Active | lnk_up=0 | reset=1 | drv_fsm= DRV_IDLE
env.rq_agent.drv [LTSSM] [LTSSM] 0x00 : Detect.Active | lnk_up=0 | reset=1 | drv_fsm= DRV_IDLE
env.rq_agent.drv [LTSSM] [LTSSM] 0x01 : Detect.Active | lnk_up=0 | reset=1 | drv_fsm= DRV_IDLE
env.rq_agent.drv [LTSSM] [LTSSM] 0x02 : Polling.Active | lnk_up=0 | reset=1 | drv_fsm= DRV_IDLE
env.rq_agent.drv [LTSSM] [LTSSM] 0x04 : Polling.Configuration | lnk_up=0 | reset=1 | drv_fsm= DRV_IDLE
env.rq_agent.drv [LTSSM] [LTSSM] 0x05 : Configuration.Linkwidth.Start | lnk_up=0 | reset=1 | drv_fsm= DRV_IDLE
env.rq_agent.drv [LTSSM] [LTSSM] 0x06 : Configuration.Linkwidth.Accept | lnk_up=0 | reset=1 | drv_fsm= DRV_IDLE
env.rq_agent.drv [LTSSM] [LTSSM] 0x08 : Configuration.Lanenum.Wait | lnk_up=0 | reset=1 | drv_fsm= DRV_IDLE
env.rq_agent.drv [LTSSM] [LTSSM] 0x07 : Configuration.Lanenum.Accept | lnk_up=0 | reset=1 | drv_fsm= DRV_IDLE
env.rq_agent.drv [LTSSM] [LTSSM] 0x09 : Configuration.Complete | lnk_up=0 | reset=1 | drv_fsm= DRV_IDLE
env.rq_agent.drv [LTSSM] [LTSSM] 0x0a : Configuration.Idle | lnk_up=0 | reset=1 | drv_fsm= DRV_IDLE
env.rq_agent.drv [LTSSM] [LTSSM] 0x10 : L0 *** LINK ACTIVE *** | lnk_up=0 | reset=1 | drv_fsm= DRV_IDLE
env.rq_agent.drv [LTSSM] [LTSSM] 0x0b : Recovery.RcvrLock | lnk_up=0 | reset=1 | drv_fsm= DRV_IDLE
env.rc_agent.mon [RC_MON] [MON-FSM] MON_DOWN | Completions Received =0 | link_up=0 | tvalid=0 | SOP bit =0
env.rq_agent.drv [LTSSM] [LTSSM] 0x0d : Recovery.RcvrCfg | lnk_up=0 | reset=0 | drv_fsm= DRV_IDLE
env.rq_agent.drv [LTSSM] [LTSSM] 0x0c : Recovery.Speed | lnk_up=0 | reset=0 | drv_fsm= DRV_IDLE
env.rq_agent.drv [LTSSM] [LTSSM] 0x0b : Recovery.RcvrLock | lnk_up=0 | reset=0 | drv_fsm= DRV_IDLE
env.rq_agent.drv [LTSSM] [LTSSM] 0x28 : Recovery.Equalization.Phase0 | lnk_up=0 | reset=0 | drv_fsm= DRV_IDLE
env.rq_agent.drv [LTSSM] [LTSSM] 0x29 : Recovery.Equalization.Phase1 | lnk_up=0 | reset=0 | drv_fsm= DRV_IDLE
env.rq_agent.drv [LTSSM] [LTSSM] 0x0b : Recovery.RcvrLock | lnk_up=0 | reset=0 | drv_fsm= DRV_IDLE
env.rq_agent.drv [LTSSM] [LTSSM] 0x0d : Recovery.RcvrCfg | lnk_up=0 | reset=0 | drv_fsm= DRV_IDLE
env.rq_agent.drv [LTSSM] [LTSSM] 0x0e : Recovery.Idle | lnk_up=0 | reset=0 | drv_fsm= DRV_IDLE
env.rq_agent.drv [LTSSM] [LTSSM] 0x10 : L0 *** LINK ACTIVE *** | lnk_up=0 | reset=0 | drv_fsm= DRV_IDLE
env.rc_agent.mon [RC_MON] [MON-FSM] MON_IDLE | Completions Received =0 | link_up=1 | tvalid=0 | SOP bit =0

```

Fig. 9 Simulation log showing LTSSM states.

data for configuration Type0 read request.

All the request TLP and completion TLPs are synchronized with clock of frequency 100MHz. Sequences are driven after reset is released and link is established between the

LTSSM and sequences are driven and received after link up.

The simulation logs the configuration read request TLP of the figure 9 is shown in

```
MON_IDLE | Completions Received =0 | link_up=1 | tvalid=0 | SOP bit =0
DRV_WAIT | TLP packets_sent=0 | link_up=1 | reset=0
DRV_WAIT | TLP packets_sent=0 | link_up=1 | reset=0
DRV_WAIT | CfgRd0 preparing | Address=0x034 | Waiting for ready...
DRV_DRIVING | TLP packets_sent=0 | link_up=1 | reset=0
DRV_DRIVING | Driving CfgRd0 packet | Address=0x0d0 | Length=1DW | TLP_header[127:0]=0x01010000000040010000000000000d0, SEQUENCE_number=0xa | Parity=0x0 tuser=0xa00000f
DRV_DONE | TLP packets_sent=0 | link_up=1 | reset=0
DRV_IDLE | TLP packets_sent=1 | link_up=1 | reset=0
MON_HEADER | Completions Received =0 | link_up=1 | tvalid=1 | SOP bit =1
MON_HEADER | Completion Addr=0x50 | Status=000 ( SC ) | ReqID=0x0000 | CplrID=0x0000
MON_DONE | Completions Received =0 | link_up=1 | tvalid=1 | SOP bit =1
MON_DONE | TLP RX #1 | Status=SC (Successful completion) | ReqID=0x0000 | CplrID=0x0000 | ByteCnt=4 | Len=1DW | LowAddr=0x50 | Data[0]=0x10830000
MON_IDLE | Completions Received =1 | link_up=1 | tvalid=1 | SOP bit =1
```

Fig. 10 Simulation log showing successful completion for the configuration read request.

```
[DRV-FSM] DRV_WAIT | MWr64 preparing | Address=0x400000010 | Length=1DW | Data=0x14330201 | Waiting for tready...
[DRV-FSM] DRV_DRIVING | TLP packets_sent=25 | link_up=1 | reset=0
[DRV-FSM] DRV_DRIVING | Driving MWr64 Packet | Address=0x400000010 | Length=1DW | TLP_header=0x000000000000000000000001433020101010008000008010000000400000010
[DRV-FSM] DRV_DRIVING | MWr64 complete | Tag=0x08 | 1 beat(s) | Data[0]=0x14330201
[DRV-FSM] DRV_DONE | TLP packets_sent=25 | link_up=1 | reset=0
[DRV-FSM] DRV_IDLE | TLP packets_sent=26 | link_up=1 | reset=0
[DRV-FSM] DRV_WAIT | TLP packets_sent=26 | link_up=1 | reset=0
[DRV-FSM] DRV_WAIT | TLP packets_sent=26 | link_up=1 | reset=0
[DRV-FSM] DRV_WAIT | MRd64 preparing | Address=0x400000010 | Length=1DW | Waiting for ready...
[DRV-FSM] DRV_DRIVING | TLP packets_sent=26 | link_up=1 | reset=0
[DRV-FSM] DRV_DRIVING | Driving MRd64 packet | Address=0x400000010 | Length=1DW | TLP_header[127:0]=0x01010009000000010000000400000010
[DRV-FSM] DRV_DONE | TLP packets_sent=26 | link_up=1 | reset=0
[DRV-FSM] DRV_IDLE | TLP packets_sent=27 | link_up=1 | reset=0
[MON-FSM] MON_HEADER | Completions Received =23 | link_up=1 | tvalid=1 | SOP bit =1
[MON-FSM] MON_HEADER | Completion Addr=0x10 | Status=001 ( UR ) | ReqID=0x0000 | CplrID=0x0000
[MON-FSM] MON_DONE | Completions Received =23 | link_up=1 | tvalid=1 | SOP bit =1
[MON-FSM] MON_DONE | TLP RX #24 | Status= UR (Unsupported Req) | ReqID=0x0000 | CplrID=0x0000 | ByteCnt=4 | Len=0DW | LowAddr=0x10 | NoData
N] UR tag=0x02 - EP rejected TLP (BAR not programmed or mem space disabled)
```

Fig. 11 Error scenario test for memory read request without configuration of base address register.

root complex and endpoint. Link Training and Status State Machine (LTSSM) is a Finite State Machine occurring at physical layer of the PCIe it determines the status of the link as shown in figure 9. LTSSM has detect, polling, configuration, recovery states. Link initially at detection states after the device is detected it starts polling by sending parameter negotiation of the receiver and device configuration like link speed, link width after negotiation the link is established. Figure 9 shows the state of

figure 11 received successful completion status with data. We also tested the endpoint device memory of 64-bit address and 32-bit address by initiating a write request and followed by 64-bit and 32-bit read request, by configuring all 6 base address registers from BAR0 to BAR5 and received a successful completion status with data for all the base address register.

The error scenarios are also tested by without configuring the base address register and by directly sending the memory

request TLP, received completion with unsupported request status as illustrated in figure 11.

The completion abort and configuration request retry status are also obtained as the per the PCIe specification. We disabled the configuration space and driving the configuration request through the driver received the configuration request retry completion status. Full range of 4KB address range memory address are tested by generated all address memory TLP through the constrained randomization sequence and achieved a successful completion with data. Out-of-bound address which exceeding the 4KB address space where also verified.

Memory write with data and poisoned bit high is driven with address and followed by memory read sequence with same address is driven to test the data is dropped or not, received completion without data.

We created a 33 testcases covering all TLP types memory write, read, configuration type0 and type1 write and read, I/O write and read for legacy PCIe devices and messages from the endpoint to the root complex. Sequences are generated through the constrained randomization without manually sending sequence one by one which reduces the verification time. Table I shows code coverage summary.

TABLE I
CODE COVERAGE SUMMARY

Total files	Total modules	Statement coverage score	Branch coverage score
54	82	81.3451	76.8875

V. CONCLUSION

This work proposes UVM-based functional verification of a non-FLIT-based PCIe endpoint, with the UVM testbench

performing root complex behavior for verifying the endpoint functionality. The verification environment contains two agents: an active agent responsible for driving stimulus and a passive agent that collects responses from the endpoint. The results demonstrate a reusable verification environment that supports Gen1 to Gen5 and validates TLP types for memory, I/O, configuration, and message. Functional coverage across the TLP types achieved 100%, and overall code coverage was 81.3451%. In the future, this verification environment will extend to support FLIT-based PCIe beyond Gen5 and will verify die-to-die transactions by placing the verification environment on one die and the endpoint on the other to confirm successful communication between dies via Universal Chiplet Interconnect.

REFERENCES

- [1] S. P. Jagtap, V. Ingale and A. Gokhale, "PCIe Transaction and Data link Layers Verification IP Development using UVM," 2022 IEEE 3rd Global Conference for Advancement in Technology (GCAT), Bangalore, India, 2022, pp. 1-4.
- [2] M. B. V. S. Manohar and S. Sarkar, "UVM-Based Verification of the Transaction Layer of PCIe Gen5," 2026 IEEE Global Symposium on Emerging and Communication Technologies (GSEACT), Hyderabad, India, 2026, pp. 1-5.
- [3] S. Cao and J. Li, "Design and Implementation of an Efficient UVM-Based Verification Framework for a PCIe Endpoint Module," 2026 5th International Conference on Electronics, Integrated Circuits and Communication Technology (EICCT), Chengdu, China, 2026, pp. 812-817.
- [4] R. J. R. Kumar and P. A, "A UVM-Based Verification Framework for PCIe Gen5 Data Link Layer Reliability," 2026 IEEE Global Symposium on Emerging



and Communication Technologies (GSEACT), Hyderabad, India, 2026, pp. 1-5.

[5] U. Nema, V. Agarwal, V. Ingale and A. Gokhale, "Development of Verification IP for PCI Express 5.0 and its Integration with a Hypothetical Slave Device," 2024 15th International Conference on Computing Communication and Networking Technologies (ICCCNT), Kamand, India, 2024, pp. 1-6.

[6] N. K. G and N. B. G V L, "Interoperable SoC Framework for PCIe Design Verification Challenges," 2024 IEEE 5th Women in Technology Conference (WINTeCHCON), Bengaluru, India, 2024, pp. 1-6.

[7] S. K and C. C, "Hybrid Random Forest-Reinforcement Learning Framework for Efficient Verification of PCIe 6.0 SoC Designs," 2026 International Conference on Smart Electronic Devices and Intelligent Systems (ICSEDIS), Chiramanangad, India, 2026, pp. 281-289.

[8] G. Rohilla, D. Mathur and U. Ghanekar, "Functional Verification of MAC-PHY Layer of PCI Express Gen5.0 with PIPE Interface using UVM," 2020 International Conference for Emerging Technology (INCET), Belgaum, India, 2020, pp. 1-5.

[9] V. N. Vaidya, V. Ingale and A. Gokhale, "Development of Verification IP of Physical Layer of PCIe," 2022 IEEE 3rd Global Conference for Advancement in Technology (GCAT), Bangalore, India, 2022, pp. 1-5.

[10] M. B. Sree and C. Subhas, "UVM Sequence Library Development for NonFLIT PCIe Transaction Layer Packets," 2026 1st International Conference on VLSI Circuits and Systems (ICVCS), Kolkata, India, 2026, pp. 1-6.

[11] V. Kuruba and P. R. Reddy, "A Modular UVM-Based Scoreboard Architecture for Protocol-Aware Verification of Non-FLIT PCIe Transactions," 2026 1st International Conference on VLSI Circuits and Systems (ICVCS), Kolkata, India, 2026, pp. 1-6.

[12] T. Thomas et al., "Left-shifter: A pre-silicon framework for usage model based performance verification of the PCIe interface in server processor system on chips," 2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Singapore, Singapore, 2022, pp. 90-98.

[13] AMD, UltraScale+ Devices Integrated Block for PCI Express v1.3 Product Guide (PG213), v1.3, Jul. 31, 2026.

[Online] <https://docs.amd.com/r/en-US/pg213-pcie4-ultrascale-plus/Introduction>.