POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Master's Degree Thesis

Design and FPGA prototyping of a real-time monitoring unit for the PIPE interface

Supervisors

Prof. Guido MASERA

Dr. Colin GILLY

Candidate

Nicolò RIGOTTI

December 2021

Summary

The purpose of this dissertation is to provide an on-chip solution for monitoring and checking in real-time, exploiting event-based sampling to achieve a larger observation window and reduce the amount of time spent during the hardware debugging.

With the augmentation of design complexity, the contribution of hardware validation has become extremely relevant. The risk of error is indeed raised because of tightening timing budgets inside the SoC and electrical issues outside the SoC (e.g., crosstalk, line attenuation, jitter, etc.).

Unfortunately, the observability of internal signals during the hardware debugging is extremely limited. In the past years, several hardware debugging methodologies have been implemented to address this problem and, nowadays, the leading FPGA vendors (such as Xilinx and Altera) provide integrated solutions to shorten this process.

Embedded Logic Analyzers enable the direct observation of internal signals without using dedicated output pins, because they exploit serial transmission interfaces that are already presented on board. However, they suffer from tight resource limitations that heavily reduce the observation time window and increase the time needed to track the cause of a malfunction.

On the other hand, External Test Equipment bypasses the issues due to resource limitation, by routing interesting signals toward output pins. However, this solution is not suitable for pin-constrained design or wide-bus observation. In addition, output pins usually do not support higher bandwidths.

Since PLDA is particularly interested in monitoring a specific interface (i.e. the PIPE interface), it is possible to adopt an event-based approach. Namely, a set of interesting events is defined and the information is captured and elaborated only when a precise event or error is detected.

This process highly reduces the amount of data to store, since it is able to select which information is worth saving. However, a specific-purpose solution is less flexible and it must be designed to be easily modified to support future releases of the PCI Express protocol.

The conducted work is organized into six chapters. Chapter 1 provides some basic knowledge about the PCI Express. Namely, it illustrates its applications, it explains how the serial transmission operates and it even explores the PIPE interface standard. Chapter 2 looks at the state-of-the-art hardware debugging methodologies currently employed in the ASIC/FPGA development. Hence, it compares the two main approaches and it analyzes some of the solutions that are currently employed. Chapter 3 presents the process that aims to identify the system's constraints and define the key functionalities to implement. In addition, it presents the system architecture and the main reasons to adopt it. Chapter 4 illustrates the development of the hardware. In particular, it discusses the design of each major logic block as well as the verification and validation environments to test the IP core. Chapter 5 describes the software architecture of the application. The final chapter resumes the main outcomes achieved during these five months of internship.

The proposed methodology is conceived during a five-month internship at PLDA, which aims to shorten the SoC/ASIC/FPGA development cycles during third-party PCIe PHY integration and validation.

Company

PLDA is a company with over 20 years of experience in the technology market, with clients in 62 countries and offices in France, Bulgaria, USA and Taiwan. It designs and develops Semiconductor Intellectual Property (SIP) for high-speed interconnection (i.e. PCI Express, CXL and CCIX).

I have personally handled both the system development and the project management. However, my work was reviewed each week by my enterprise tutor, Colin Gilly, and I was supported by a team of four engineers from PLDA, who were always willing to help me in case of technical issues.

Acknowledgements

Firstly, I am very grateful to all the people at PLDA for providing a lovely working environment. In particular, I would like to thank my supervisor Colin Gilly for being the first to believe in me and that gave me the opportunity to work on this project.

Big thanks as well as to Caio Aires, Florentin Bosshard, Thomas De Csiky and Mathieu Rougnon since they have always found some time to answer all my questions and teach me new skills, although they were always overloaded. Special mention to Julien Eydoux, who pushed me during these last months to finalize this dissertation.

Additionally, I would like to extend my gratitude to the Politecnico di Torino and, notably, to my university supervisor Guido Masera for his support.

Then, I would like to express my sincere gratitude to my mother and my father for their love and continuous support. I would not be here without them. I would like to thank my sister who always shows me a model to emulate.

At last, but not the least. I would like to thank Chiara. She has always found a way to cheer me up and has never let me feel alone.

Table of Contents

List of Tables

List of Figures

Acronyms

ACK
ACKnowledged

ADI
Advanced Design Integration

AHB
Advanced High-performance Bus

AMBA
Advanced Microcontroller Bus Architecture

APB
Advanced Peripheral Bus

ASIC
Application Specific Integrated Circuit

AXI
Advanced eXtensible Interface

BRAM
Block Random Access Memory

CPU
Central Processing Unit

CXL
Compute eXpress Link

DLLP

Data Link Layer Packet

DMA

Direct Memory Access

DUT

Device Under Test

EIEOS

Electrical Idle Exit Ordered Set

EIOS

Electrical Idle Ordered Set

EMI

Electro-Magnetic Interference

FF

Flip Flop

FPGA

Field Programmable Gate Array

FSM

Finite State Machine

GPU

Graphics Processing Unit

HDL

Hardware Description Language

ILA

Integrated Logic Analyzer

IO

Input Output

IP

Intellectual Property

LCRC

Link Cyclic Redundancy Code

LTSSM

Link Training and Status State Machine

LUT

Look-Up Table

MVP

Minimum Viable Product

MVVP

Model–View–VieModel

NAK

Not AcKnowledged

NVMe

Non-Volatile Memory Express

PCI

Peripheral Component Interconnect

PCIe

PCI Express

PCI-SIG

Peripheral Component Interconnect Special Interest Group

PCI-X

PCI eXtended

PHY

PHYsical Layer

PIPE

PHY Interface for PCI Express

PLL

Phase-Locked Loop

RAM

Random Access Memory

RTL

Register Transfer Level

SIP

Soft Intellectual Property

SoC

System-on-Chip

SSD

Solid State Disk

SKPOS

SKPip Ordered Set

SWOT

Strength, Weakness, Opportunity, and Threat

TC

Traffic Class

TLP

Transaction Layer Packet

TS

Training Sequence

UART

Universal Asynchronous Receiver-Transmitter

UVM

Universal Verification Methodology

VC

Virtual Channel

WFH

Work From Home

Chapter 1

Notions about PCI Express and PIPE Interface [1]

This chapter offers a brief overview of the Peripheral Component Interconnect Express, commonly known as PCI Express. The first section provides basic information about this protocol. It starts by presenting its possible applications. Then, it describes how the serial transmission operates. Finally, it quickly illustrates the layered structure of PCIe.

Instead, the second section looks into the Physical Layer more deeply. The interface that must be observed and checked (the PIPE interface) is indeed located in this layer.

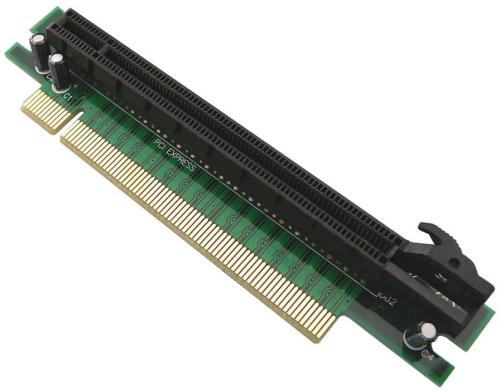


Figure 1.1: PCI Express interface example

[1] **Based on PCI Express Base Specification [1] and Mindshare's guide about PCI Express technology [2]**

1.1 PCI Express Fondaments

PCI Express, formally abbreviated PCIe, is an industry-standard high-performance, general-purpose serial I/O interconnect. Version 1.0 was introduced in 2003 by PCI-SIG to replace the old standards, PCI and PCI-X (PCI eXtended), which had reached a practical ceiling on effective bandwidth mainly due to their parallel architecture.

At present, it is largely employed in consumer, server, or industrial applications. It has quickly become the interconnect standard to link motherboards to external peripherals, such as graphic cards, hard disk drives, Wi-Fi, or Ethernet hardware interfaces.

The PCIe Market Forecast made by Technavio[3] has envisaged a big market's growth in the next four years. This expansion is led by the demand for new data centers, which employ storage technologies with NVMe, a non-volatile memory interface standard that utilizes PCIe interfaces to SSDs. In addition, data centers have also started to employ advanced processing techniques that exploit GPUs and accelerators to support emerging machine learning and artificial intelligence.[4]

1.1.1 PCI Express Link

The physical connection between two PCIe devices is called Link. The communication is differentially driven and dual-simplex since signals can be sent or received simultaneously but they pass through different paths.

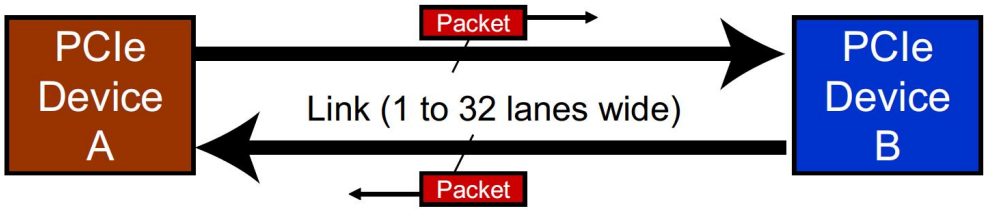


Figure 1.2: PCI Express Link [2]

A differential send and receive signal pair makes a Lane. One lane is sufficient for communication between the devices and no other signals are required. However, a Link can be made up of several lanes.

The number of lanes is called Link width and it is directly proportional to the link bandwidth. More lanes increase the number of bits that can be sent with each clock, but as well as cost, space requirement, and power consumption. It is up to the platform designer to find the correct trade-off.

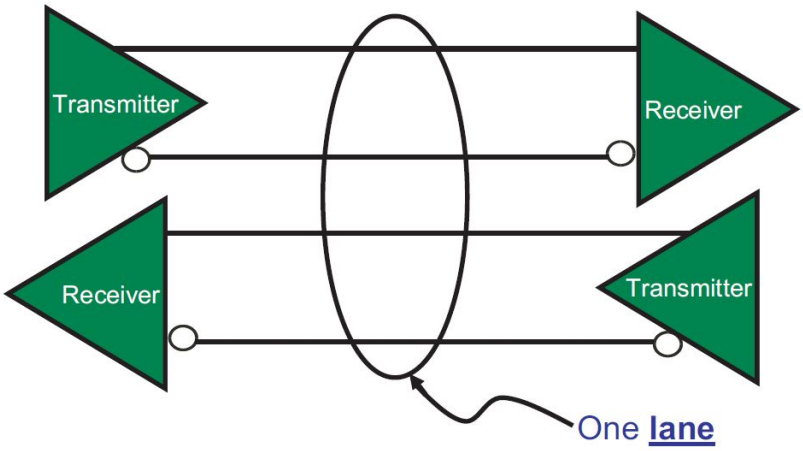


Figure 1.3: PCI Express Lane [2]

1.1.2 PCI Express Versions

At the current time, five versions of PCIe have already been released and the specification of PCI Express 6.0 is on track for being finalized before the end of 2021.[5]

The first generation (Gen1) bit rate is 2.5 GT/s, which means that one lane is 0.25 GB/s. Gen1 indeed employs a 8b/10b encoding, which means it must send ten bits to transmit a byte of data. Hence, the bit rate is divided by ten to achieve the one lane throughput .

The higher versions roughly double the bandwidth of the previous PCIe. However,it is possible to notice from table 1.1 that Gen3 doubles the throughput without doubling the transfer rate. This outcome is achieved because from Gen3 a different type of encoding called 128b/130b is used.

Version	Introd.	Transfer rate	Throughput [GB/s]				
			x1	x2	x4	x8	x16
1.0	2003	2.5 GT/s	0.250	0.500	1.000	2.000	4.000
2.0	2007	5.0 GT/s	0.500	1.000	2.000	4.000	8.000
3.0	2010	8.0 GT/s	0.985	1.969	3.938	7.877	17.754
4.0	2017	16.0 GT/s	1.969	3.938	7.877	17.754	31.508
5.0	2019	32.0 GT/s	3.938	7.877	15.754	31.508	63.015

Table 1.1: Table of PCIe versions and their transfer rate and throughput according to the Link width.

All PCI express versions are compatible with one another. However, the maximum bandwidth of the PCI Express interface is limited by the port with the oldest version. For instance, if a graphic card that supports PCIe Gen5 is connected to a motherboard that only supports PCIe Gen2, the maximum bit rate will be 5.0 GT/s because of the motherboard PCIe port.

In order to assure backward compatibility, the PCIe Link is always initialised in Gen1. Then, if both the devices support higher speeds, a Link Re-Training is performed to change the Link speed.

1.1.3 PCI Express Serial Transport

PCI and PCI-X had two main problems: a high pin count and a speed limit, as consequent to the fact that in a parallel bus model the signal flight time must be smaller than a clock cycle. PCIe overcomes these issues by adopting a serial-based model.

Moving to a serial transport reduces the pin count because sideband signals are not necessary anymore. However, the receiver must know the type of information that has been sent. In PCIe, all transactions are transmitted in defined structures called packets. Knowing the pattern, the receiver can extract what it needs.

In PCIe, the transmitter embeds the clock built into the data stream, so issues related to flight time and clock skew are solved, because the clock and data arrive at the same time. The receiver recovers the clock thanks to a PLL. Since this PLL continuously tunes the recovered clock, changes of temperature or voltage are compensated.

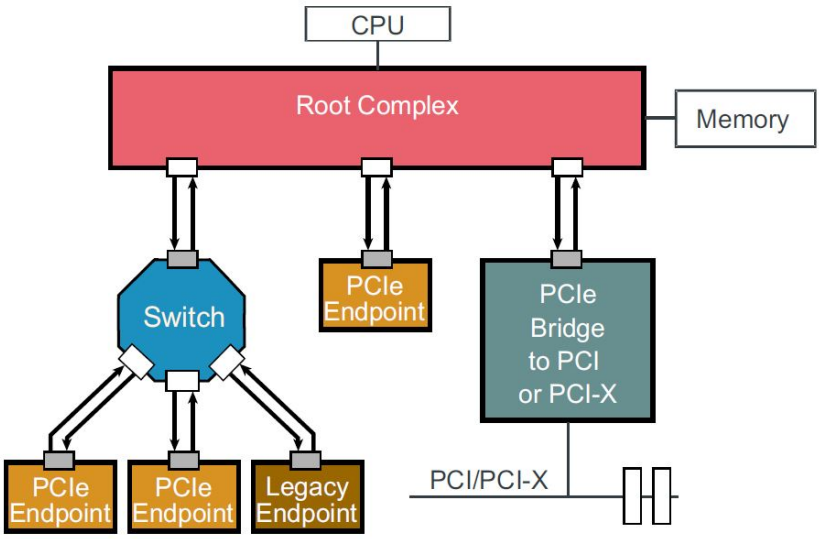


Figure 1.4: PCI Express topology example [2]

1.1.4 PCI Express Topology

Because of the very high speed, the Link must be a point-to-point connection. However, Bridges and Switches are employed to obtain flexible system topologies. The definitions of the system elements are as follow:

- The top of the PCIe hierarchy is the CPU, which is connected to the PCIe buses thanks to the Root Complex.
- Switches provide a fanout or aggregation capability and allow more devices to be attached to a single PCIe port. They act as packet routers and recognize which path a given packet will need to take based on its address.
- Bridges provide an interface to other buses, such as PCI, PCI-X, or even another PCIe bus.
- The endpoint is a device that resides at the bottom of the branches of the tree topology and implements a single Upstream Port toward the Root.

1.1.5 PCI Express Device Layer

PCI Express is characterized as a layered architecture. This division is made to simplify the migration to new versions of the specifications.

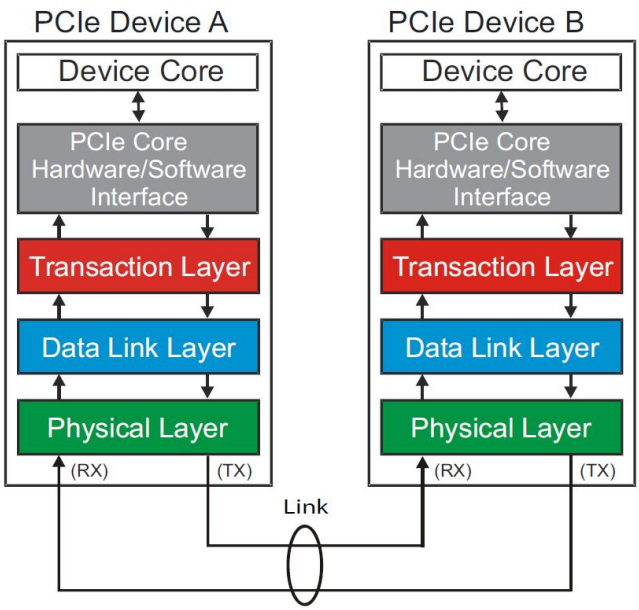


Figure 1.5: PCI Express Device Layers [2]

Four main layers are defined. Each layer can be considered as being logically split into two parts that work independently: a transmit side for outbound traffic and a receive-side for inbound traffic. In general, the transmitter side of one layer communicates with the receiver side of the corresponding layer in the other device.

Packets can go through two or more layers, according to the type. Each layer adds some bits at the beginning and end of the packets. For instance, the Data Link Layer appends a Sequence Number to TLP packets.

Device Core / Software Layer

The core implements the main functionality of the device. It will be either the source or the destination of all Requests. It provides information like transaction type, address, or amount of data to transfer.

Transaction Layer

According to the request generated by the Software Layer, it creates the related outbound packet. It looks at inbound packets and forwards the information contained to the Software Layer.

A Transaction is the combination of a Request packet, that submits a command to the target device, and a Completion packet, sent by the target in return. At the Transaction Layer, transactions can be grouped into four categories: Memory, IO, Configuration, and Messages. Transaction Layer handles them by using TLPs (Transaction Layer Packets).

In addition, this layer provides functionalities concerning:

- The Quality of Service, useful to support time-sensitive transmission. In PCIe, each packet is prioritized by setting a 3-bit field called Traffic Class (TC). Each port has multiple buffers, called Virtual Channer (VC). According to the TC, a packet is placed in the appropriate buffer and a Port Arbitration manages the VC outputs to provide guaranteed service for a given path.

- The Transaction Ordering, i.e. packets with the same Traffic Class are routed through the topology in the correct order.

- The Flow Control, namely that a transmitter sends a packet only if there is enough space in the buffer to receive it. It is completely handled by hardware and transparent for the software.

Data Link Layer

This layer manages the creation and the decoding of Data Link Layer Packets (DLLPs). They are small packets (just 8 bytes) that are responsible for Link maintenance and management. For instance, they are sent to report the buffer space, in the PCIe Flow Control mechanism.

The Data Link Layer is in charge of the error detection and correction in the TLPs. A Link Cyclic Redundancy Code (LCRC) and a Sequence Number are appended by the Data Link Layer to each outgoing TLP. At the receiver, the LCRC is checked and if no error is detected, an ACK DLLP is sent in return. Otherwise, the receiver drops the TLP and it replays a NAK DLLP. The transmitter will be in charge of resending the packet.

The Data Link Layer is involved in power management too. DLLPs are employed to communicate the requests and handshakes related to Link and system power states.

Physical Layer

The Physical Layer is the lowest hierarchical layer. All types of packets are processed in this layer. It is responsible for several procedures, i.e. byte striping and scrambling. More detailed information is reported in the next Section

1.2 Physical Layer Overview

The Physical Layer is split into two sections: the logical part and the electrical part. The Logical Physical Layer is responsible for preparing the outbound packets to be transmitted and for reversing this process for the inbound packets. The Electrical Physical Layer is an analog interface, made up of differential drivers and receivers for each lane.

In the Logical Physical Layer, Start and End characters are added to TLPs and DLLPs to allow the receiver to detect packet boundaries. Then, each packet is split

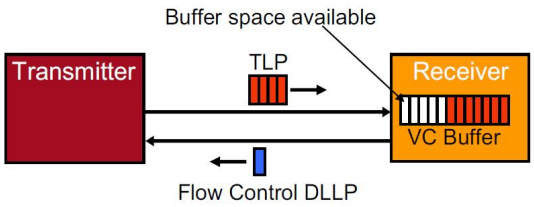


Figure 1.6: Flow Control basics [2]

into bytes that are transmitted by all the lanes independently (a process called byte striping). The packet is put back together at the receiver.

The Physical Layer is responsible for scrambling each byte before the transmission. Scrambled bytes reduce repetitive patterns on the transmission line, to decrease cross-talks among lanes or other EMI (Electro-magnetic interference).

1.2.1 Encoding

Before being transmitted, characters are encoded in the Logic Physical Layer. This encoding is used to achieve different goals.

Firstly, encoding ensures that the data stream has enough edges so that the receiver can recover the embedded clock. Then, it reduces the drifting voltage that occurs when the signal average is predominantly one level or the other, so the line charges up. The encoder compensates for this inequality by adding a disparity bit. Finally, encoding also facilitates the detection of transmission errors.

For PCIe v1.0 and v2.0, bytes are encoded using 8b/10b, a coding scheme patented by IBM in 1984 that translates the 8-bit characters into 10-bit symbols. From PCIe v3.0, 128b/130b encoding is employed.

1.2.2 Ordered Sets

The Logical Physical Layer is responsible for generating and decoding Ordered Sets. Technically, they are not packets because they do not have any Start or End characters. However, they have some precise patterns that can be easily detected by the receiver. Ordered Sets are replicated on all Lanes at the same time because each Lane is technically independent.

Specification [1] reports different types of Ordered Set. Each one has a different goal. For instance, TS1 and TS2 Ordered Sets are used for Link initialization and training. Electrical Idle Ordered Sets are sent to inform the receiver that the Link passes to a lower-power state. SKP Ordered Sets are used for Clock Tolerance Compensation.

1.2.3 Link Training and Initialization

The Physical Layer is even responsible for the initialization and training process of the Link. The Link Training and Status State Machine (LTSSM) automatically handles this sequence. Namely, to configure and initialize the Link, it must perform the following tasks:

Bit Lock. In the beginning, the Receiver's clock is not yet synchronized with one of the incoming signals. Hence, the receiver must recover the Transmitter's clock.

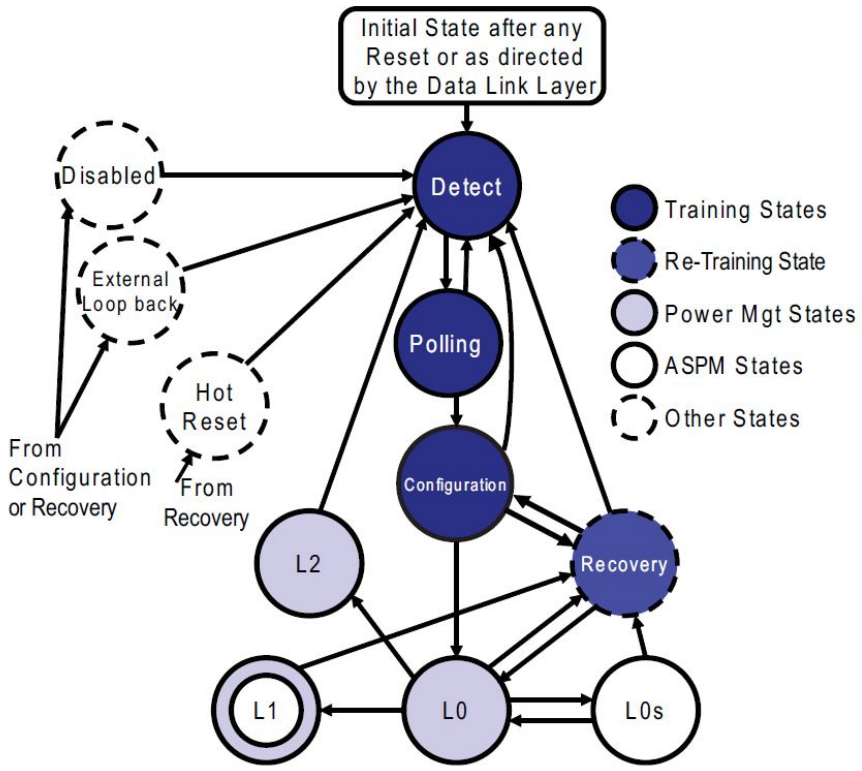


Figure 1.7: Link Training and Status State Machine (LTSSM) [2]

Symbol/Block Lock. Even if, after the Bit Lock, the receiver can detect the single bits, it doesn't know the boundaries of the 10-bit Symbols (PCIe v1.0 or v2.0) or the 130-bit Block (PCIe v3.0 or further). Hence, the transmitter sends TS Ordered Sets that are easily recognizable, so the receiver can detect their boundaries.

Link/Lane Number Negotiation. The Downstream Port (the "leader") assigns the Link number and Lane number.

Lane Reversal. Sometimes it is easier to inverse the Lane numbers to simplify the routing and allow the Lanes to be wired directly without crisscrossing.

Polarity Inversion. Even the D+ and D- differential pairs might be inverted. Each Lane can automatically correct it during the training. Link Data Rate. To assure backward compatibility, after a reset the Link initialization will always use the default 2.5 Gbit/s data rate. However, if a higher data rate is available, it is possible to change the data rate after the initial training.

Equalization. When the data rate is equal to or greater than 8.0 GT/s, an Equalization procedure is demanded to meet the signal quality requirements.

Detect	The initial state after reset. In this state, a device detects whether a receiver is present on the other side of the Link.
Polling	In this state, Transmitters begin to send TS1 and TS2 Ordered Sets so that Receivers can use them to: • Achieve Bit Lock • Acquire Symbol Lock or Block Lock • Correct Lane polarity inversion • Lane available and Lane data rates
Configuration	Upstream and Downstream components now play specific roles as they continue to exchange TS1s and TS2s to: • Determine Link width • Assign Lane numbers • Check for Lane reversal and correct it • De-skew Lane-to-Lane timing differences
L0	This is the normal, fully active state of a Link when packets and ordered sets can be exchanged.
Recovery	This state is entered when the Link needs re-training, for instance when errors occur in L0. Bit Lock and Symbol/Block lock are re-established as in the Polling state, but by taking much less time.
L0s	This state is designed to provide some power savings while affording a quick recovery time back to L0.
L1	This state provides greater power savings by trading off a longer recovery time than L0s does.
L2	In this state, the main power is turned off to achieve a greater power savings.
Loopback	This state is used for testing.
Disable	This state allows a configured Link to be disabled. In this state, the Transmitter is in the Electrical Idle state while the Receiver is in the low impedance state.

Hot Reset	Software can reset a Link by setting the Secondary Bus Reset bit in the Bridge Control register.

Table 1.2: LTSSM main states

1.2.4 PHY Interface for PCI Express (PIPE)

In the Physical Layer, there is the PHY Interface for PCI Express (PIPE), a standard defined by Intel to help integration. It indeed enables the development of functionality equivalent PCI Express, SATA, USB and DisplayPort. [6]

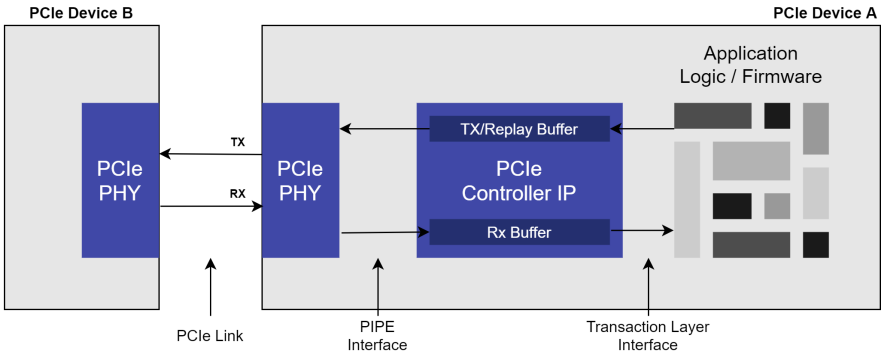


Figure 1.8: PHY and PCIe Controller

Referring specifically to PCI Express, this interface connects the MAC layer (that is located inside of the PCIe Controller) to the PCIe PHY. At this level, data are not serialized yet and they are sent (and received) through the TxData (and RxData). According to the data rate and the PCIe generation, this bus can have 8, 16, 32, or 64 wires.

Signals of the PIPE interface are synchronized with PCLK. This clock is provided by a PLL that uses the reference clock (CLK) as the source.[6] PCLK frequency might vary over time, for instance when the data rate is increased during the Link Re-Training. However, it is possible to find out the PLCK rate by looking at the signal PLCK_Rate [4:0]. The PCIe Controller also provides the current LTSSM state and the Equalization phase.

The specification of PCIe PIPE 4.4.1 introduced the Message Bus interface. This interface provides a way to initiate and participate in non-latency-sensitive PIPE operations using a small number of wires.[6]

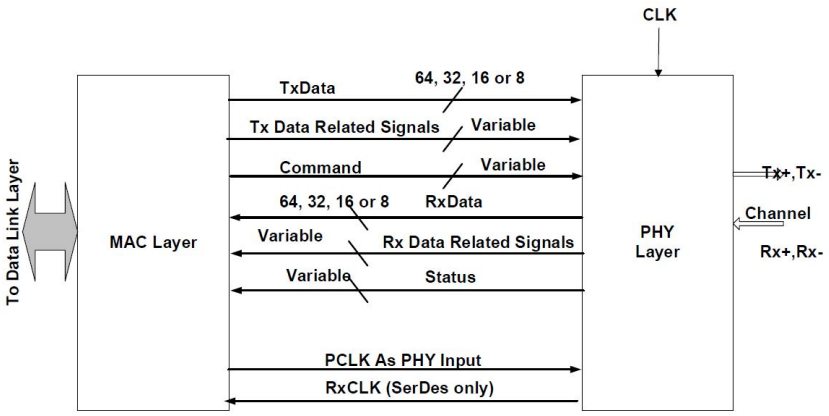


Figure 1.9: PIPE Interface [6]

Chapter 2

Analysis of Hardware Debugging Methodologies

This chapter wants to present the result of the study on hardware debugging methodologies. In the first place, the design process is described to quickly provide some context around validation and hardware debugging. Then, the two main categories of hardware debugging tools are compared to highlight their advantages and disadvantages. Finally, some specific tools are analyzed to show if they are a possible solution to monitor and check the PIPE interface.

This chapter notably illustrates the design process of Soft IP on FPGA for two reasons. Firstly, because the solution to monitor and check the PIPE interface will be only prototyped on an FPGA, since the internship periodo is too limited. Secondly, even in ASIC production, the design is often prototyped on FPGA, because it is more flexible than ASIC, so the time of design, validation, and testing is strongly reduced. [7] However, the ASIC design process includes some additional phases that are not described in this thesis.

2.1 Design Process Overview for synthesis on FPGA

The design flow of a Soft Intellectual Property (SIP) counts four main steps: specification, RTL design, synthesis, and implementation. However, the augmentation of design complexity has increased the risk of errors so much that verification and validation have become extremely relevant and they can take fifty percent of the design cycle.[7]

The specification is the first step of the process. The engineer must identify the requirements and the functionalities that the SIP must have. These functionalities

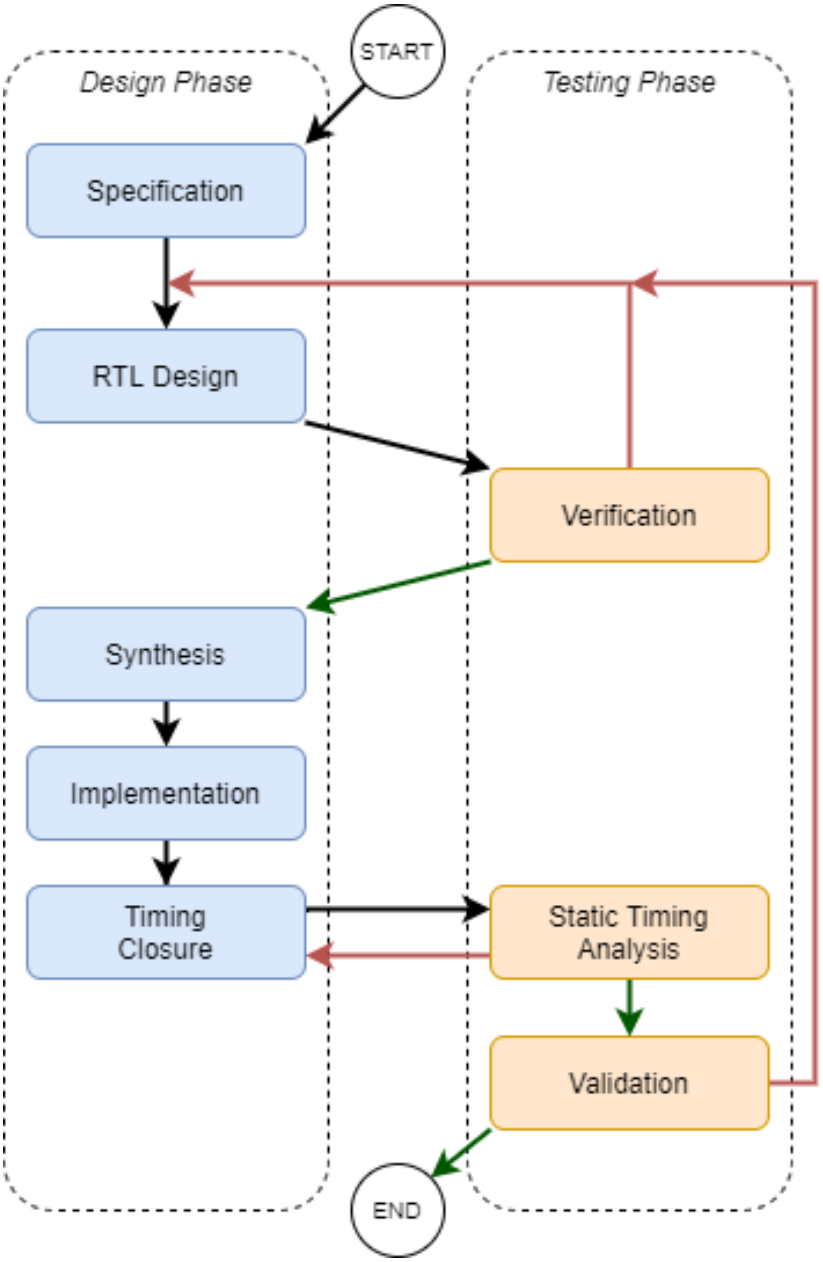


Figure 2.1: FPGA Design Process

are then described in hardware description languages (HDLs) such as VHDL or Verilog. HDLs use the register transfer level (RTL) abstraction, which represents the flow of signals between registers and the logical operation performed by these signals.[8]

The next step is verification. The goal of the verification technique is to check

if the implemented design meets specifications before the design is manufactured. Simulation is used to verify the design behavior. A wide range of stimuli are applied to the design and the simulated design outputs are compared with the expected ones. It is an easy way to catch most of the logical errors, especially if the resulting outputs are well-known. However, an exhaustive check for complex design is practically unfeasible.[7]

Techniques, such as formal verification, have been conceived to mathematically prove or disprove the correctness of the design according to a certain formal specification. There are two main approaches to formal verification. The first one is equivalence checking and it consists in comparing a reference model with the targeted design. The other approach is called property (or model) checking. In this case, it checks the satisfaction of design properties that are formed in a dedicated verification language.[8]

After the verification, the design description is translated from an HDL description to a gate-level representation or a logic component. This description, however, is not designed for a specific FPGA. The next phase, called implementation, takes the netlist and it does the optimization, the placement, and the routing. This process generates a bitstream file that can be uploaded into the FPGA.Along with the bitstream file, the synthesis tool provides a report that includes the result of the Static Timing Analysis.

The Static Timing Analysis computes the critical path. Namely, it calculates all the delay due by combinational logic between an input and a register, two registers, or a register and an output. The critical path is the highest delay among those. If the critical path does not meet the timing constraint, then the design must be modified to reduce the critical path.

The design is now implemented physically, however, a final step must be performed. Verification cannot, indeed, guarantee that the design is error-free. Simulation (so verification too) is based on models that cannot emulate all the physical characteristics of the design. Hence, it cannot catch several design problems, such as timing violation, EMI, or signal integrity issues.[7]

The goal of validation is to identify design bugs that are present in the physical circuit. Unfortunately, the observability of the internal signals is very limited, and tracing back the error to the source can be frustrating. Several hardware debugging methodologies have been conceived to address this problem. In broad terms, they can be divided into two categories: external test equipment and embedded logic analyzer.[7]

2.2 Embedded Logic Analyzers

Because capturing data in consecutive clock cycles is essential for identifying the timing-related bugs, some techniques have been developed to provide real-time observability of a limited set of internal signals.[8]

Embedded Logic Analyzers rely on on-chip buffers that sample a set of internal signals. Captured data is then transmitted to the external debug software via a low bandwidth interface, (e.g. JTAG or UART).

This solution is widely employed because it is usually relatively inexpensive. Major FPGA vendors, indeed, offer embedded logic analyzer cores free of charge (e.g. SignalTap from Altera and LogiCORE ILA from Xilinx).[9]

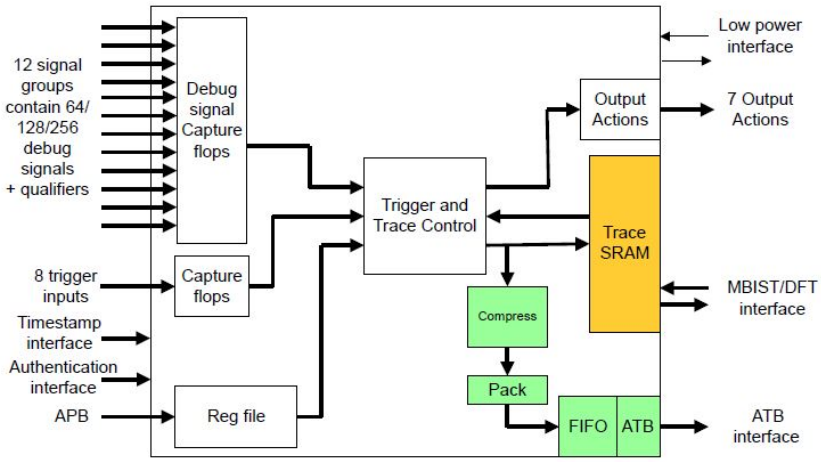


Figure 2.2: Arm ELA600 block diagram [10]

In addition, embedded logic analyzers do not use additional pins, since they usually exploit serial transmission interfaces that are already present on board. Conversely, the embedded logic analyzer core must employ part of the internal FPGA logic resources and memory blocks that could be used to implement the design. This matter limits the buffer size, so, as a consequence, the observation window too.[9] In addition, the IO ports usually suffer from bandwidth limitation.

Embedded logic analyzers can only operate in state mode: the core samples data synchronous to a specified clock that is present in the FPGA design, hence it can not provide accurate signal timing relationships. And besides, they do not have a way of correlating the captured information to board-level or system-level information.[9]

Finally, if it is necessary to change the set of probed signals, the whole design must be resynthesized. For complex SIP, this process could take from several hours to even an entire day.

Tool Name	Company
Integrated Logic Analyzer (ILA)	Xilinx
SignalTap	Intel
XpressAGENT Logic Analyzer	PLDA
AXI BFM Monitor Checker	PLDA
Arm CoreSight ELA-600	Arm

Table 2.1: Embedded Logic Analyzer examples

2.3 External Test Equipment

Because of the limitations embedded logic analyzers have, sometimes it is preferable to use some external test equipment (it can be an external logic analyzer, a protocol analyzer or even an oscilloscope).[9] In this case, interesting signals are routed to FPGA pins, so that they can be captured by the debug tool.

Figure 2.3: Summit T3-16 PCI Express Protocol Analyzer by LeCroy [11]

This solution does not use FPGA resources, hence there are no strong constraints on the observation window. However, it requires dedicated output pins. It is an issue if the design is pin-constrained or whether it is necessary to observe wide buses.[9]

External test equipment is usually more expensive than an embedded logic analyzer, nonetheless, it provides more acquisition modes and trigger capabilities. It implements different trigger state modes and it can capture very long capture in timing mode with very high resolution. External test equipment also enables correlating the captured data with other system information.[9]

Tool Name	Company
Summit T3-16 PCI Express Protocol Analyzer	LeCroy
Inspector v2.0	PLDA

Table 2.2: External Test Equipment examples

2.4 Integrated Logic Analyzer (ILA) by Xilinx

Xilinx provides the Integrated Logic Analyzer (ILA) in its Vivado Design Suite as a debug tool for their FPGAs. ILA is made up of an IP core that captures interesting signals after the detection of a trigger event. These data are consequently transmitted via JTAG to the computer and it is possible to visualize them as waveforms in the Vivado Lab interface.[12]

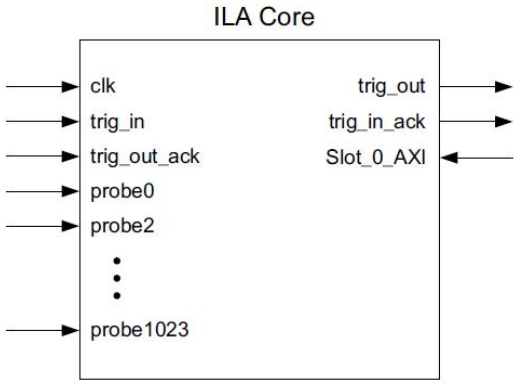


Figure 2.4: ILA Core Symbol [12]

The Vivado GUI allows the user to set the trigger events. The trigger comparator can detect a precise pattern or an edge transition. Multiple trigger conditions can be combined using "AND" and "OR" boolean operators.[9]

This tool is normally used at PLDA because it allows the observation of any FPGA internal signal in real-time. However, the debug process using ILA takes a considerable amount of time. The constraint on buffer size forces users to select a limited number of probed signals. This condition turns the debug process into a cyclic procedure where the engineer selects a few signals, synthesizes the design, runs the test, and manually analyzes the waveforms. If these signals are not the cause of the bug, he must select other signals, resynthesize, and so on.

If the problem source is at low level, the debug process could even demand ten or more cycles. Since only the ILA initial setting and synthesis might take a full day, tracking back the error source could require 3/4 weeks or even more.

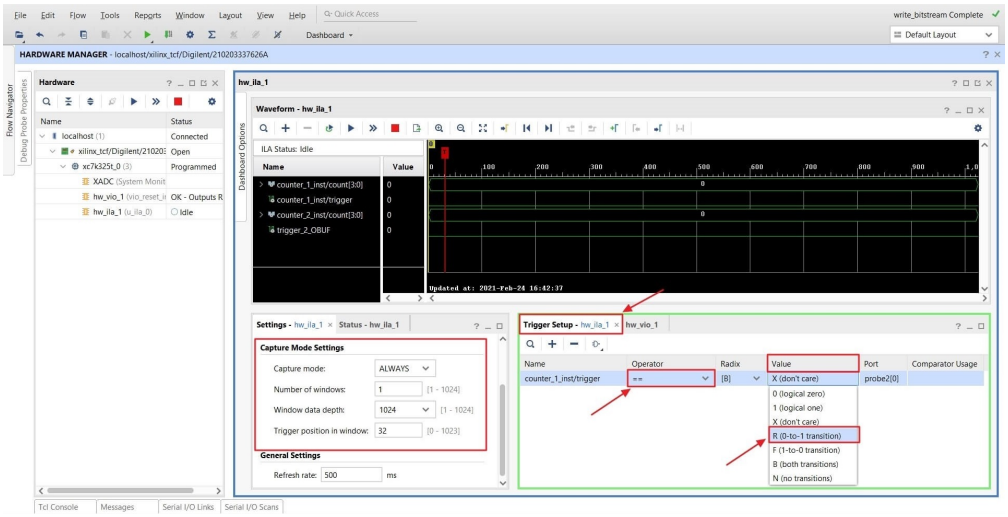


Figure 2.5: ILA GUI in Vivado Suite

2.5 Inspector for PCI Express by PLDA

The Inspector for PCI Express is a solution conceived by PLDA to observe in real-time a PCIe link. It is composed of a board that must be plugged between the Upstream port and the Downstream port. The interposer card analyzes the Link traffic and sends data to the computer via UART. This information is consequently

Figure 2.6: Inspector for PCIe plugging [13]

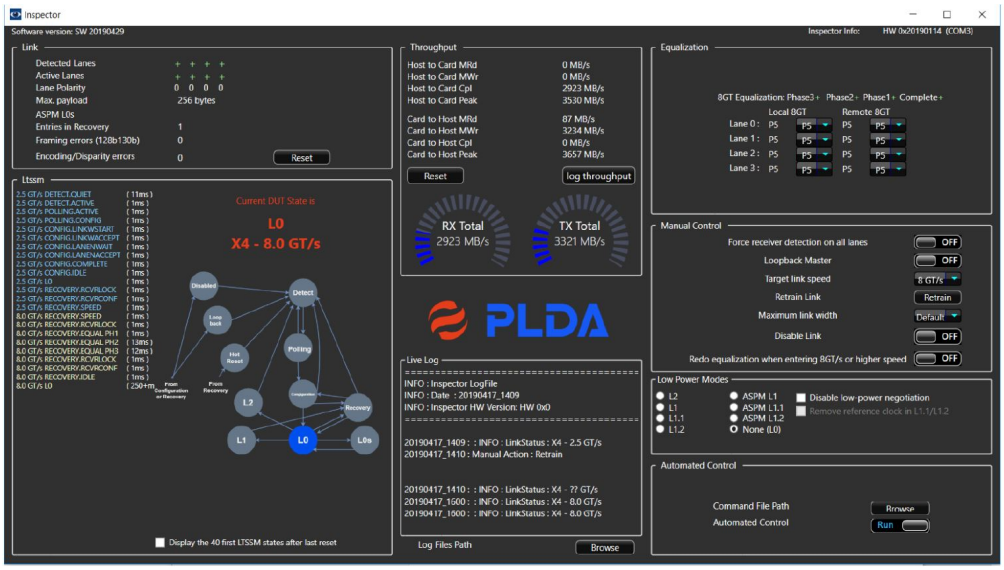


Figure 2.7: Inspector for PCIe GUI [13]

shown by the Inspector software GUI.[13]

In particular, it enables design, test, and validation engineers to:

- Monitor PCIe interface power-on process [13]
- Diagnose PCIe interface PHY and link issues [13]
- Analyze throughput performance in real-time, in the production environment [13]
- Check PCIe interface reliability [13]
- Perform PCIe interface stress tests and random tests via scripting (exerciser functionality) [13]

The Inspector is useful to detect link errors, however, it is not able to track back the cause, because it is not integrated into the same chip of the device under test (DUT), so it cannot directly probe signals of the PIPE interface.

The Inspector shows the LTSSM state of its Downstream port, but it cannot ensure consistency with the Upstream port of the DUT. In case of system failure, the user cannot know when the DUT gets stuck, especially if the Upstream port employs a PHY different from the one used in the Inspector.

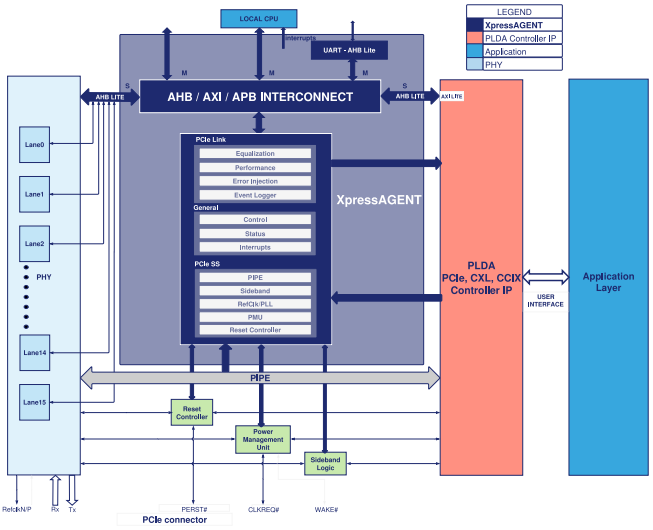


Figure 2.8: XpressAGENT integration block diagram [**PLDA**]

2.6 XpressAGENT by PLDA

XpressAGENT is a tool that PLDA is developing to simplify the control management of PCIe/CXL PHY based subsystems that employ a PLDA Controller. Its main goal is to save time during the validation by providing relevant functionalities that should be implemented otherwise by PLDA's clients.[14]

XpressAGENT can programmatically configure via AMBA interface the PLDA IP and the PHY for link testing or reliability testing. It can even monitor the PIPE interface in real-time by logging performance, link states, error events, etc. Acquired data can be transmitted via UART and visualized on the XpressAGENT User Interface.[14]

A logic analyzer is implemented in the XpressAGENT IP, as well. It implements the same functionalities of an ILA, but it can be synthesized even for ASIC applications.[14] It inherits all the disadvantages of ILA, for instance, the long debugging time. There is no checking logic implemented in the hardware.

2.7 AXI BFM Monitor by PLDA

In its AXI Bus Functional Model (BFM), PLDA has integrated an AXI Monitor that ensures the correct behavior of the AXI interface. This submodule samples the AXI signals, elaborates them in real-time to compute the AXI performance and to detect any event or errors, then saves these data in memory. Through a backdoor, the acquired data are transmitted to the computer, where they are translated into report files, readable by human users.

Even though this solution employs part of the FPGA resources, it still provides a wider observation window, because it logs only when an event or an error occurs. This elaboration optimizes the saved data and reduces the necessary buffer space. However, this tool is specific for monitoring an AXI interface and it cannot be used for other purposes.

2.8 Dissertation Motivation

The study of hardware debugging methodologies that are currently used during the validation has shown that a solution fitted to monitor and check the PIPE interface does not exist.

The general-purpose tools are too inefficient and they require a debugging time that is too important. The more specific tools either are not suited for the PIPE Interface, or they do not provide enough information to track back the cause of malfunctioning.

Hence, the next chapters will present the process to design a solution that monitors and checks the behavior of the PIPE interface with the objective to significantly reduce the hardware debugging time.

Chapter 3

Definition of the Solution

The research presented in the previous chapter has pointed out the absence of an efficient solution to monitor and check the PIPE interface, so as to reduce the time for PCIe PHY integration and validation.

This chapter will, thereby, expose the result of the study aimed at defining a new product. It will define the product's objectives, identify its constraints, and describe the process to select the Minimum Viable Product. Finally, Finally, it introduces the designed system architecture and it explains the decisions taken.

3.1 Objectives

As analyzed in chapter 2.5, the Inspector for PCIe is a useful tool to detect if the Link does not work properly. However, it provides high-level information, so a deeper investigation must be performed to find out the cause of the failure.

Instead, Xilinx ILA or similar Logic Analyzer provide a lot of low-level information. Hence, they are indispensable to identify the exact signal that affects the Link operation. Unfortunately, a large number of signals are involved in the Link operation and checking them all would require an important amount of time.

Engineer's experience helps reduce the debugging time, since he is able to select a smaller group of signals and he utilizes an effective top-down approach. However, the debugging phase could still take several weeks or even months of investigation.

This investigation time could be significantly reduced by implementing a tool that monitors and checks the PIPE interface, so as to provide some key information that allows him to detect the signals to probe in one or two attempts.

Hence, the main goal of this project is to design a new solution to monitor and check the behavior of the PIPE interface at-speed. This can be more specifically expressed through four objectives:

PIPE Interface Monitoring: The solution must provide all the information

about the state of the PIPE Interface and the events that occur that occur during Link Training and Re-Training.

PIPE Interface Checking: The solution must check protocol requirements and report whether there are any errors or lockups during Link Training and Re-Training.

Usability: Users must be able to visualize the test outcomes and interact with the system through a user interface.

Temporal Relation: The module must provide a time reference to establish a temporal relation among detected events and errors, so that the user can look at what happened when the failure occurred.

3.2 Constraints

The product's constraints are split into two categories: the ones that must be considered during all the development steps and the ones that must be fulfilled by the final product.

3.2.1 Prototype Constraints

Synthesizable This tool is meant for hardware debugging, so it must be synthesizable and implementable on an FPGA.

Independent The tool must not be affected by the behavior of the DUT. Hence, it must:

- Provide an External access
- Employ an unrelated Reset

Transparent The tool must not affect the transmission in any way.

High-Speed Sampling The module must be able to capture all the data and control signals of the PIPE interface.

Range Sampling Rate from 62.5 MHz to 1 GHz

3.2.2 Final Constraints

Resources Limitation The module must not produce any design constraint for the DUT, so it can only exploit the spare resources.

Max Used LUTs (DUT / module)	10%
Max Used BRAMS (DUT / module)	15%
Max Gate Count (DUT / module)	10%

Configurable The design must support all the PIPE configurations of the different PLDA PCIe Controllers. Notably:

Lane Configuration	x1, x2, x4, x8, x16
PIPE Version	v5.2
Signaling Rate [GT/s]	2.5, 5.0, 8.0, 16.0, 32.0
Data Path Width	8, 16, 32, 64
Architecture	Original PIPE
Elastic Buffer Op Model	Nominal Hall Full buffer

Observation Window The module must provide information about an interval wide enough to observe the link training and retraining.

3.3 Identification of the Minimum Viable Product (MVP)

Even if the project objectives are defined, they are not exhaustive enough to describe the specific functionalities of the system. However, PCI Express is a complex protocol that counts a lot of rules, but the time to deliver the prototype is quite short[1]. To be as effective as possible, the functionalities have been defined using the Minimum Viable Product (MVP) technique.

3.3.1 Minimum Viable Product Definition

The Minimum Viable Product is a concept introduced and popularized by Eric Ries in his book *The Lean Startup*[15].

> "The minimum viable product is that version of a new product which allows a team to collect the maximum amount of validated learning about customers with the least effort."
> *Eric Ries* [15]

[1]The duration of the interniship is five months. Two of those have been spent to learn the PCIe protocol, familiarize with the PLDA environment and study the hardware debugging methodologies.

In other words, it is the initial version of a product that implements enough features to be used by consumers and no more. Future product releases will be implemented according to the user feedback.[16]

MVP must have three key characteristics:

> "It has enough value that people are willing to use it or buy it initially.
> It demonstrates enough future benefit to retain early adopters.
> It provides a feedback loop to guide future development."[16]

This method is useful to validate the product idea before making a large investment. It avoids developing features that consumers do not want, so as to save and optimize both time and resources.[16]

3.3.2 The selection of functionalities

Firstly, a list of possible functionalities is drawn up by looking at the PCIe specification[1], the PIPE specification[6], and Mindshare's guide about PCIe[2]. Then, a hierarchical tree diagram is produced by analyzing the dependencies among these functionalities (for instance, the module must be able to detect the different ordered sets to identify most of Link Training events).

This initial draft is presented to different engineers and managers at PLDA to be reviewed. To avoid a polarized perspective, this small survey targets people that works in different team (i.e. Design team, Verification team, Validation team and Advanced Design & Integration (ADI) team). The hierarchical tree diagram is modified thanks to their feedback. Priorities have been assigned to each feature in order of complexity. The complexity of both implementation and specifications have been considered, as too complex specifications could lead to bugs in the implementation.

Figure 3.1 represents the final version of the hierarchical tree diagram. The arrows represent the dependencies between the functionalities. The priority increases from the top to the bottom. The highest functionalities are the core features that must be necessarily implemented in the first release.

Thanks to this diagram, the first two versions of the prototype are set out. The prototype v1.0 implements the basic features that allows the user to monitor the behaviour of the PIPE interface and to understand how this tool works. The version 2.0 adds some key checking functionalities that would significantly help the validation engineer.

Unfortunately, the prototype v2.0 will most likely not be complete before the end of the internship, because of the short deadline. However, it could be a future product development.

3.4 Requirements

A requirement is a:

> "Statement that identifies a product or process operational, functional, or design characteristic or constraint, which is unambiguous, testable or measurable, and necessary for product or process acceptability."[17]

By keeping this definition in mind, the selected functionalities are translated in an exhaustive list of requirements. Each requirement is enumerated to facilitate the testing phase.

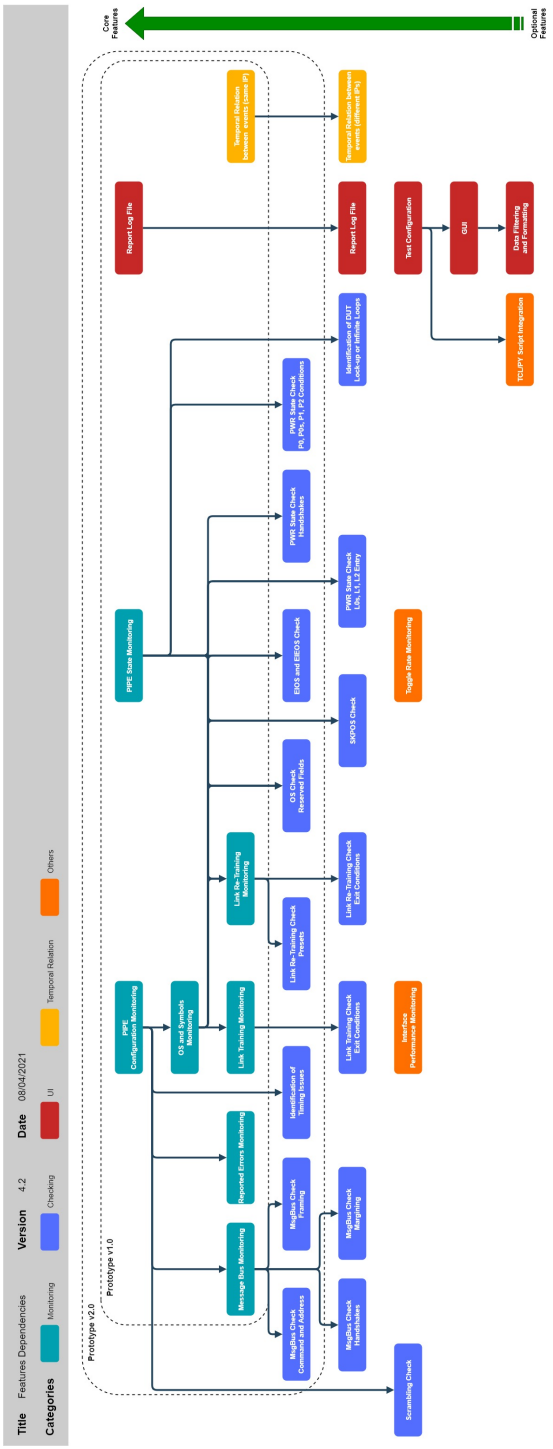


Figure 3.1: Functionalities hierarchical tree diagram

3.5 System Architecture of PIPE Monitor Checker

The technical specification of the product is delineated, so the system architecture can be designed by following those guidelines.

The device must be able to monitor and check the internal signals of the PIPE interface. Since it is not possible to route all of them to dedicated output pins to observe them externally, then the solution will be on-chip.

Since this tool must be able to monitor and check a wide time interval and during this period it is not possible a cause of resource limitation to sample and save all the signals into memory, then signals shall be elaborated and only the events and errors shall be logged.

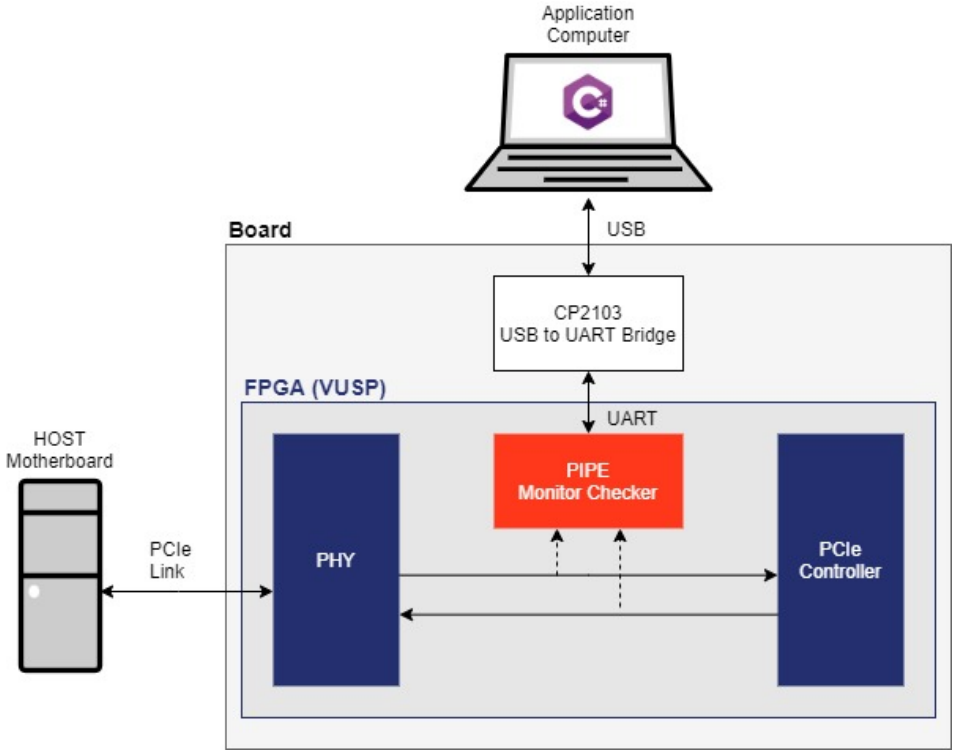


Figure 3.2: PIPE Monitor Checker System Architecture

Hence, the product (i.e. the PIPE Monitor Checker) shall consist of:

- a soft IP, directly integrated into the same chip of the DUT, that samples the PIPE Interface signals and elaborates them to detect errors and events. Elaborated data is saved in the on-chip RAMs.

- a software application that reads data from the IP's memory via an external peripheral. The read information is provided to the user thanks to a user interface.

The prototype is implemented on a Xilinx Virtex UltraScale+ FPGA provided by PLDA.

3.5.1 Analysis on signal commutation in the PIPE interface

Using the verification environment of the PLDA XpressRICH-AXI[2] Controller IP, it has been possible to simulate the PIPE interface with different configurations (i.e changing PCIe generation, Link number and PIPE version) and export the outcomes in vcd format with Questa Advanced Simulator.

A python script has been implemented to take one or more vcd files as input, count how many times a signal commutes and produce a text file as output. By giving as input the simulations about PIPE interface, this script shows that most of PIPE signals commute with a frequency significantly slower than pl_pclk (figure 3.3).

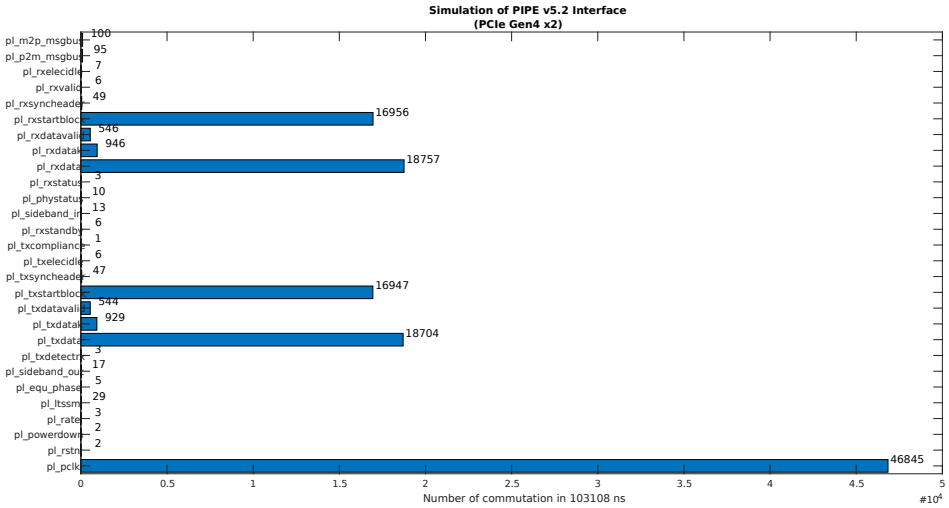


Figure 3.3: Functionalities hierarchical tree diagram

[2]**XpressRICH-AXI™** is a configurable and scalable PCIe controller Soft IP that provides a configurable, flexible AMBA AXI interconnect interface to the user. [18]

The average commutation frequency is computed as:

$$\overline{f^s_{com}}(T) = \frac{1}{T}\sum_{t_k=0}^{T} h^s_{com}(t_k)$$
$$where \quad h^s_{com}(t_k) = \begin{cases} 1 & \text{if signal s commutes at the instant } t_k \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

Figure 3.4 shows that, beside the parallel data signals (pl_txdata and pl_rxdata), the other signals have an average commutation frequency with the order of magnitude of MHz or even slower.

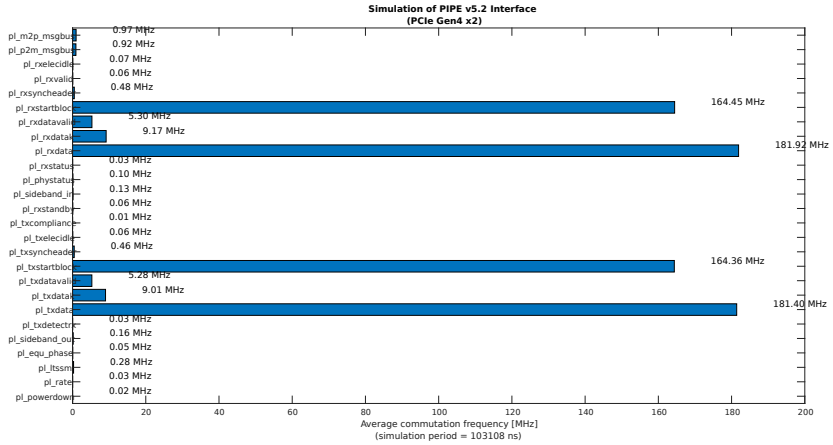


Figure 3.4: Functionalities hierarchical tree diagram

In addition, some of these signals (i.e. pl_ltssm) are widely employed during the Link Training or in the Recovery state[3], but, after these first microseconds, they remain almost constant (figure 3.5). Hence, if buffers were correctly sized to store the captured data during the Link Training, even a low-speed interface could manage the data streaming.

3.5.2 External Interface

The analysis presented in section 3.5.1 has evidenced that a low-speed interface could manage the transmission if the signal sampling is event-triggered and not time-triggered.

[3]In the average commutation frequency graph of figure 3.5, the first peak coincides with the Link Training. Instead the other two occur when the LTSSM enters in the Recovery state to increase the Link speed.

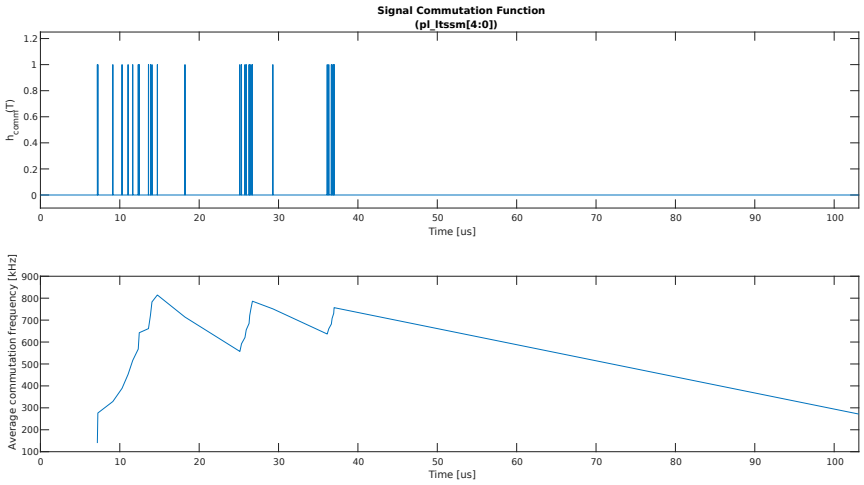


Figure 3.5: Commutation of pl_ltssm (PIPE v5.2 Gen4 x2)

UART standard is chosen for two reasons:

- The board provided by PLDA has already integrated the CP2103 USB to UART Bridge that allows to communicate the computer via USB.
- Inspector for PCIe from PLDA employs UART too. Hence, it is possible to adapt its UART transceiver to save time. In addition, PLDA has for the Inspector a protocol to read or write memory cells via UART.

UART	
Baud rate	128 000 bps
Data bits	8 bits
Parity	odd
Stop bit	1 bit
No flow control	

Table 3.1: Uart Serial Link Configuration

Chapter 4

Organization of the product development

This brief chapter summaries the efforts spent in the project management. Notably, it presents the outcome of the risk analysis and shows the project scheduling.

4.1 Risk Analysis and Risk Management

Risk management is a key procedure of project management. Its goal is to anticipate a possible negative event by setting up a preventive action to reduce its impact. So, even if it is impossible to avoid a threat, risk management improves the chance to complete a project successfully by reducing its consequences. [19]

Risk management process can be split into four steps that must be performed iteratively during each phase of the project: [19]

Risk Identification Detect and list the sources of risks. This process must be a key topic in the regular project status and reporting meetings.

Risk Evaluation Estimate the probability that a threat occurs and its possible impact on the project.

Risk Handling Plan measure and mechanism to reduce the negative consequence of a possible risk.

Risk Controlling Record and monitor risks and handling mechanisms.

Several models have been developed to analyse and evaluate risks. Some of them are quantitative and assign numerical values to risk. Others are qualitative

and employ tools, such as SWOT Analysis[1], cause and effect diagrams, or decision matrix. [20]

For this project, risk analysis is performed by using a quantitative methodology presented by Alain Humbert during his course of project management at Grenoble INP ESISAR.

Table 4.1 presents the list of risks identified during this project. Instead, table 4.2 provides the description of the numerical values employed in the risk evaluation.

#	DESCRIPTION	CATEGORY	IMPACT	IMPACT LEVEL	PROBABILITY	CRITICITY	CONTROL	RISK EXPOSURE	PREVENT. PARADE
1	Lack of experience and knowledge.	Technical	Get stuck for a technical problem.	3	4	7	2	14	Plan ahead the list of tasks and define the related tech skills. For each skill, identify useful resources or key people. Peer reviews to prevent possible issues.
2	Time for doing a task incorrectly estimate.	Organizational	Do not have enough time to actually achieve all the planned goals.	2	4	6	2	12	Trying to always consider the worst-case scenario to estimate the time. Have the project schedule reviewed by people with more experience. Periodically update the project schedule.
3	Inability to work for several days (i.e serious illness).	Human	Forced to stop working for several days.	4	1	5	2	10	Always document the work done, so that another person is able to complete it.
4	Force to WFH[2].	External	Unable to go to the laboratory.	1	5	6	1	6	Assure a remote connection.
5	Loss of motivation.	Human	Loss of efficiency.	2	3	5	2	10	Keep in touch with the team members. Organize task that can be done in parallel and among which you can switch to avoid monotony.

[1] *SWOT Analysis* Strength, Weakness, Opportunity, and Threat Analysis

[2] *WFH* Work From Home

6	Get lost.	Organizational	Loss of efficiency.	2	4	6	1	6	Plan elementary subtasks in the short period and track the completed ones. Periodically update the project schedule to always have the overall view.

Table 4.1: List of risks

Impact Level	Represent the level of impact on the project	1: very low 2: low 3: medium	4: high 5: very high
Probability	Represent the occurrence level of the risk	1: very low 2: low 3: medium	4: high 5: very high
Criticity	= Impact Level + Probability	(between 2 and 10)	
Control	Represent the control level of the risk	1: heavily controlled 2: moderately controlled 3: sparsely or not controller	
Risk Exposure	= Control * Criticity	(between 2 and 30)	

Table 4.2: Description of the numerical values employed in risk evaluation

4.2 Project Development Scheduling

The risk management in section 4.1 has pointed out the necessity of a project scheduling. It indeed makes the development more smooth, since it delineates strategy and a logic workflow. Since the beginning of the project, project scheduling can provide a clear picture of the requirements, by estimating the effort, the resources and the knowledge necessary to complete a task. [21]

The approach to product development is based on Agile Methodology. The iterative increments (or Sprints) are identified according to the functionalities selected in section 3.3. A specified period of time (time box) is assigned to each increment. During this time box, the feature is designed, implemented and tested.

A detailed scheduling is represented with a GANTT chart, because it allows following the workflow, as well. Figure 1 synthesizes the project scheduling defined

at the beginning of the development.

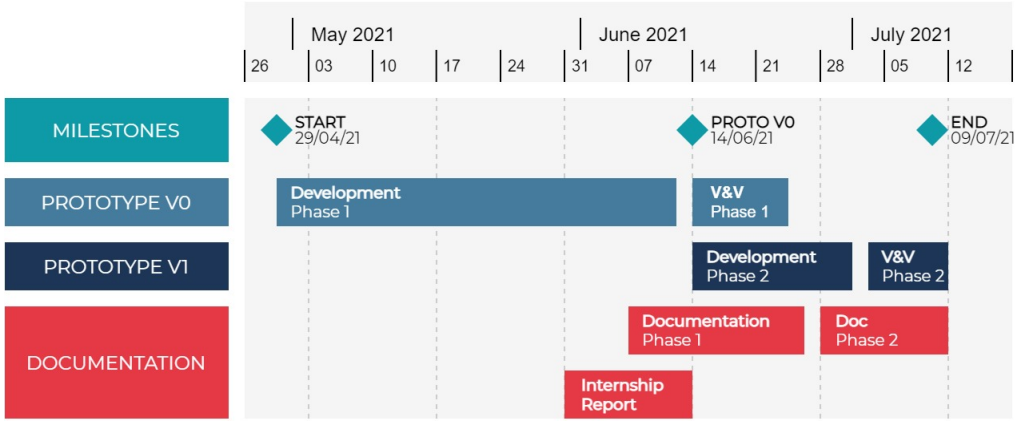


Figure 4.1: Project schedule dated 23/04/2021

Chapter 5

Development of the on-chip IP Core

This chapter describes how the IP core works internally and what problems lead to some specific solution.

The approach to illustrate is top-down and it reflects the real process that has been followed during the development. Hence, it firstly presents the top architecture. Then, it shows the logic blocks in charge of monitoring and checking the PIPE interface. Finally, it explains how the computer can access the RAMs via UART.

The last two sections of this chapter briefly mention how the IP Core has been tested. Namely, it has been initially verified using simulation, then it has been prototyped and tested on an FPGA.

5.1 IP Top Architecture

The IP core architecture recalls the one of a conventional logic analyzer. Some submodules are in charge of sampling the PIPE signals and loading data in BRAM, whereas the rest of the module handles the stream of data to the host computer via UART.

Since the goal is to observe the PIPE interface enough to check several use cases, signals are not directly saved in memory, as the ILA. Instead, data are elaborated, and they are logged only when an error or a relevant event is detected.

5.1.1 Clock Domain

The selection of the clock domain has been a critical problem. Since PIPE signals are synchronized with PCLK, using an independent clock would have required several synchronizers. So, this solution would have taken a lot of resources and

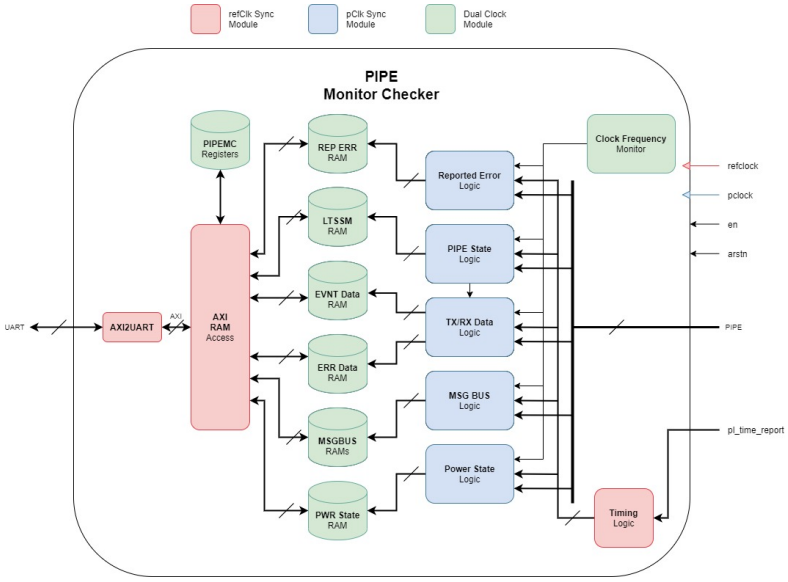


Figure 5.1: Top Architecture of the PIPE Monitor Checker IP

it would have produced other issues. Specifically, PCLK frequency can vary even when the device is running, so the module must detect and handle these changes of frequency to avoid oversampling or missing data.

However, the UART module cannot employ PCLK. Its baud rate is indeed derived from the frequency of the received clock and the PCLK frequency is too high compared to the maximum UART baud rate. In addition, PCLK is not constant and it can be even disabled during specific low-power states. So, if its frequency changed, the UART transceiver would not be able to communicate with its pair anymore. Without mentioning that, while PCLK is disabled during low-power states, the entire tool would not be accessible.

For these reasons, the core is split into two different clock domains: the logic blocks, that sample and elaborate the PIPE signals, are synchronized with PCLK, whereas the UART backdoor submodules refer to the constant REFCLK.

This solution is possible, because the submodules synchronized with PCLK only need to write into memory, instead the UART is only used to read. Hence, the two clock domains are connected by dual-clock RAMs that allows reading and writing with different frequencies.

Clock Frequency Monitor

A module, called Clock Frequency Monitor, is added to disable the Logic submodules when it detects that the PCLK is turned off. In this way, it assures that logics do not detect any false error or event.

This module estimates the frequency of PCLK using a counter. A reference clock with a constant and well-known frequency (REFCLK) is applied as its enable, whereas PCLK as its clock. Hence, the counter computes the number of PCLK cycles in a positive semi period of REFCLK. Using this value, it is possible to derive the PCLK frequency.

$$C_{OUT} = \frac{T_{REF}}{2} \cdot \frac{1}{T_{PCLK}} \quad \Rightarrow \quad f_{PCLK} = \frac{C_{OUT}}{2} f_{REF} \tag{5.1}$$

However, if the frequency of PCLK is not a multiplier of the REFCLK frequency, an error is produced on the estimated frequency.

$$f_{PCLK} + \Delta f_{PCLK} = \frac{C_{OUT} + \Delta C_{OUT}}{2} f_{REF} \quad \Rightarrow \quad \Delta f_{PCLK} = \frac{\Delta C_{OUT}}{2} f_{REF} \tag{5.2}$$

This error is directly proportional to the frequency of REFCLK. So, a frequency divider is applied to REFCLK to reduce it. In the Clock Frequency Monitor, this frequency divider is implemented with a simple counter.

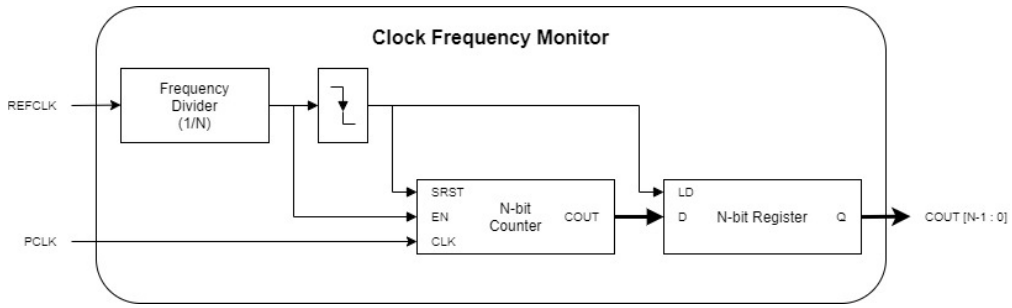


Figure 5.2: Clock Frequency Monitor Architecture

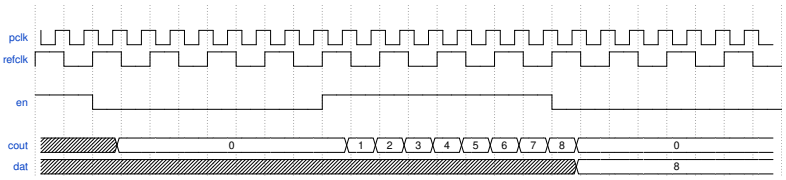


Figure 5.3: Timing Diagram of the Clock Frequency Monitor signals

5.1.2 Modularity

PCI Express is a protocol that evolves constantly. As written in section 1.1.2, almost six generations have been released in only eighteen years. This product

must be able to keep up with changes. Different mechanisms are applied to make the design as modular as possible and to simplify the integration of new features or new generations.

First of all, modules are described in Verilog using parameters to be easily reused with different or new specifications. The global parameters are listed in the *pipemc_config_h.v* file, where it is possible to change, for instance, the width and depth of memories.

Elaborated data is saved into memory as packets, whereas events and errors are encoded using numerical identifiers. Hence, new events can be easily assigned to unused code. If all the available numbers are already allocated, it is possible to increase the size of a specific packet in the *pipemc_config_h.v* file.

The unit in charge of the signal elaboration is split into different logic blocks according to the signal that they must monitor. Therefore, a module tracks the commands sended via Message Bus. Another component decodes the traffic that flows through *TxData* and *RxData*. This division allows modifications of a single block without affecting the rest of the design.

Lastly, submodules are used to describe tasks that are executed by different blocks (i.e. memory writing). So, the function must be written, simulated and verified only once, thus saving time.

5.2 PIPE State Logic

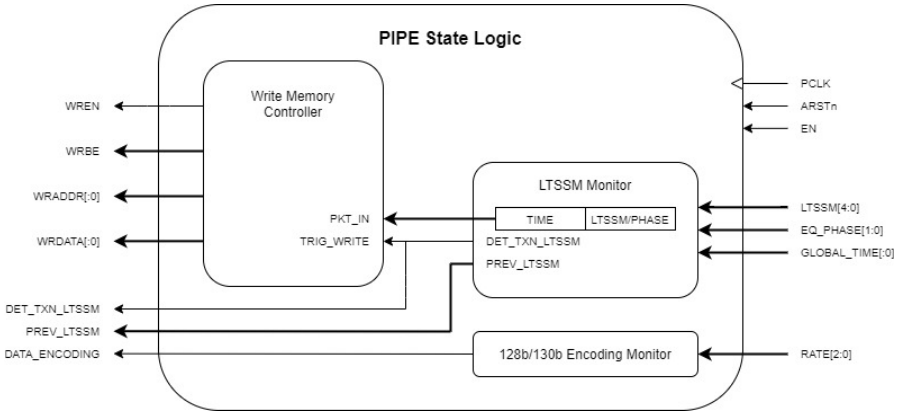


Figure 5.4: PIPE State Logic block

This module oversees the state of the PIPE interface. It tracks when there is a transition of the LTSSM, or when the equalization phase changes. In these cases, it generates a packet where it is reported the time when the transaction occurs and the LTSSM state code, or the equalization phase code (figure 5.5). Then, it sends

this packet to the Write Memory Controller that saves it into memory.

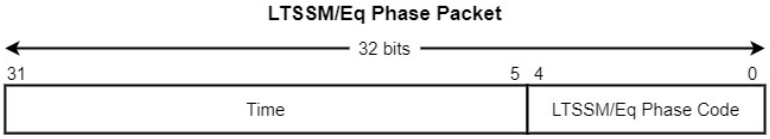


Figure 5.5: LTSSM/Equalization Phase packet

The PLDA XpressRICH controller encodes the 25 LTSSM substates using 5 bits. Hence, it is possible to employ the unused identifiers to include the four equalization phases into the 5-bits encoding (table 5.1), especially since LTSSM state does not change when the equalization phase does.

LTSSM substate	**Equ. Phase**	**ltssm code**	**equ_phase code**	**LTSSM\Eq. Phase code**
Detect.Quiet		hx00		hx00
Detect.Active		hx01		hx01
Polling.Active		hx02		hx02
Polling.Compliance		hx03		hx03
Polling.Configuration		hx04		hx04
Config.Linkwidthstart		hx05		hx05
Config.Linkwidthaccept		hx06		hx06
Config.Lanenumwait		hx07		hx07
Config.Lanenumaccept		hx08		hx08
Config.Complete		hx09		hx09
Config.Idle		hx0A		hx0A
Recovery.Receiverlock		hx0B		hx0B
Recovery.Equalization	Phase 0	hx0C	00b	hx0C
Recovery.Equalization	Phase 1	hx0C	01b	hx1D
Recovery.Equalization	Phase 2	hx0C	10b	hx1E
Recovery.Equalization	Phase 3	hx0C	11b	hx1F
Recovery.Speed		hx0D		hx0D
Recovery.ReceiverConfig		hx0E		hx0E
Recovery.Idle		hx0F		hx0F
L0		hx10		hx10
L0s		hx11		hx11
L1.Entry		hx12		hx12
L1.Idle		hx13		hx13
L2.Idle/L2.Rransmitwake		hx14		hx14
Reserved		hx15		hx15

Disable		hx16		hx16
Loopback.Entry		hx17		hx17
Loopback.Active		hx18		hx18
Loopback.Exit		hx19		hx19
Hotreset		hx1A		hx1A

Table 5.1: LTSSM substate and Equalization phases encoding

The PIPE State Logic employs a pipelined architecture that allows it to track transactions even when they occur each clock cycle (figure 5.6). The latency introduced by pipelining is constant and well-known, so it can be easily compensated by the software subsequently .

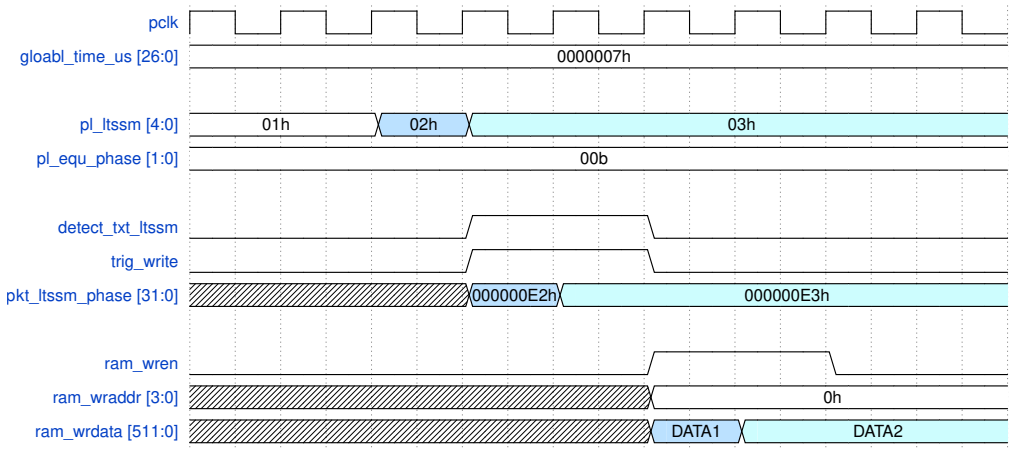


Figure 5.6: Timediagram of LTSSM packet generation and logging.

5.2.1 Write Memory Controller

Write Memory Controller is a submodule employed by several logic blocks of PIPE Monitor Checker. It is in charge of writing packets into memory. In particular, it generates the correct memory address. If the packet length is different from the RAM datapath width, it figures which bytes of the line have to be overwritten. Lastly, it asserts the write enable.

At the moment, the address generation is incremental. Namely, when a line is complete, the address is increased by one. When the last address is reached, the memory is considered full and no other packets are saved.

A circular buffer might be implemented in the future to expand the observation window. In that case, the Write Memory Controller would be responsible for

watching the Last Read and Last Write memory pointers to generate the correct address.

By changing the module parameters, it is possible to adapt the packet length and datapath width. However, the datapath width must be a multiple of the packet length, because there is no logic implemented to handle the writing of a packet over two different rows.

5.3 Reported Error Logic

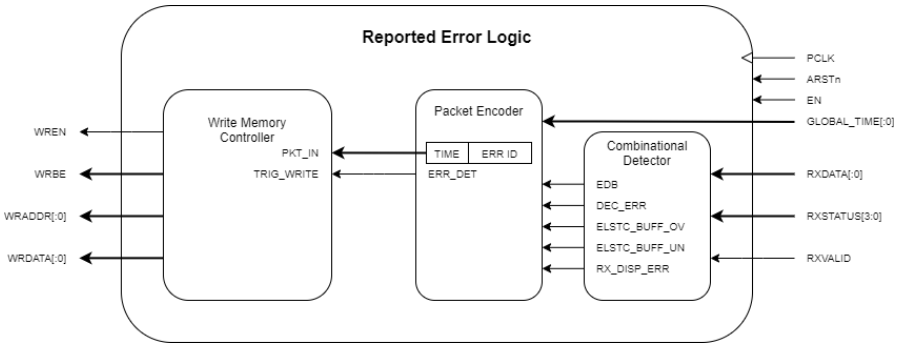


Figure 5.7: Reported Error Logic module

Specification[6] assigns to the PHY the responsibility for detecting different types of errors on the receiver side (i.e. decode error, elastic buffer overflow, etc.). When it spots one, it communicates it to the PCIe Controller by using RxStatus and, in some cases, by placing an EDB symbol in the data stream, instead of the bad byte. [6]

RxStatus			**Description**
2	**1**	**0**	
0	0	0	Received data OK
0	0	1	SKPOS added
0	1	0	SKPOS removed
0	1	1	Receiver detected
1	0	0	Both 8/10B (128/130B) decode error and (optionally) Receive Disparity error
1	0	1	Elastic Buffer overflow
1	1	0	Elastic Buffer underflow
1	1	1	Receive disparity error

Table 5.2: RxStatus[2:0] encoding [6]

The Report Error Logic is in charge of identifying when PHY reports an error through RxStatus. A simple combinational logic checks RxData and RxStatus and when it detects an interesting transmission, it asserts a flag. The packet encoder catches the flag, so it creates a new packet that is therefore saved into memory by the Writing Memory Controller.

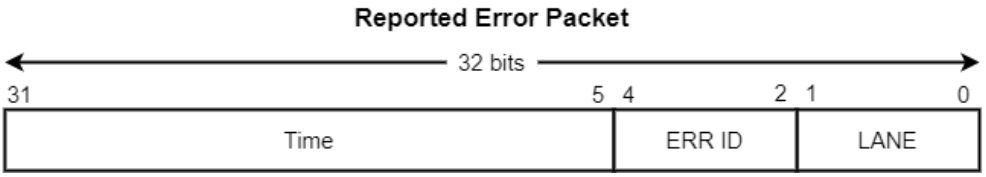


Figure 5.8: Reported Error Packet

5.4 Message Bus Logic

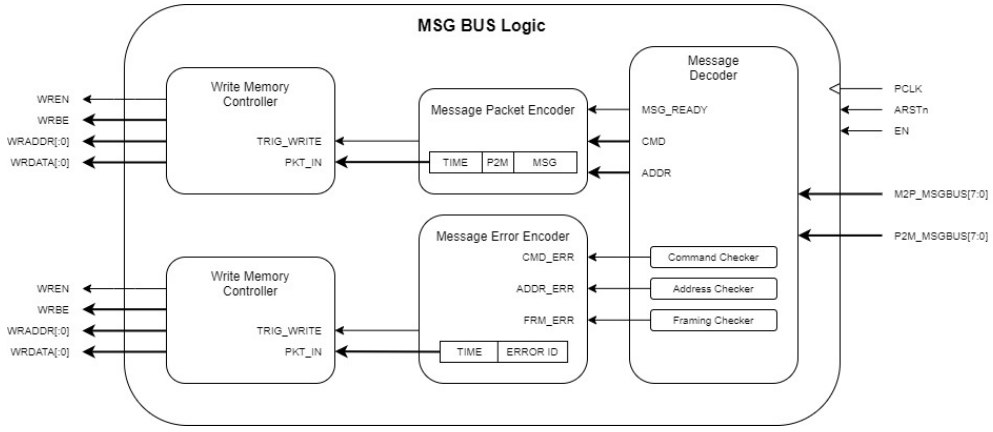


Figure 5.9: Message Bus Logic module

As mentioned in section 1.2.4, the specification of PIPE 4.4.1 introduced an interface called Message Bus that provides a way to initiate PIPE operations using a small number of wires. Namely, this interface can be used to set receiver margin, select transmitter de-emphasis, change elastic buffer depth, or control other important receiver/transmitter functions.

All controls and status bits related to these PIPE operations are mapped into 8-bit registers that are hosted in 12-bit address space in the PHY and the MAC. These registers are accessible via read and write commands that are transmitted by using m2p_msgbus[7:0] and p2m_msgbus[7:0] .

Encoding	Command	Cycles to Transmit
4'b0000	NOP	1
4'b0001	write_uncommitted	3
4'b0010	write_committed	3
4'b0011	read	2
4'b0100	read_completion	2
4'b0101	write_ack	1
Others	reserved	N/A

Table 5.3: Message Bus commands [6]

One of the responsibilities for the Message Bus Logic is to detect when a command is sent and log this transmission into memory, so as the validation engineer can check if the configuration via message bug is performed correctly.

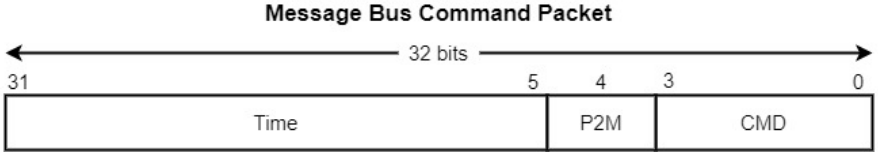


Figure 5.10: Message Bus Packet

Since each lane has its own message bus interface, this module is cloned in Verilog several times using a generate loop construct. Hence, each 16-bit interface is monitored independently to avoid transmission loss. However, this solution employs a lot of resources, so a more optimized design should be implemented in the future version.

Unfortunately, verification and validation tests frequently detect bugs related to Message Bus. It is indeed a new interface and the new releases of the PIPE specification added several new functionalities to it. Hence the PLDA design engineers must often modify the logic in charge of Message Bus and this operation obviously increases the probability of generating an error.

It is therefore essential to provide an automatic check of the Message Bus interface. Namely, the Message Bus Logic verifies :

- If the transmitted command exists;
- If the transmitted address points to a register that is not reserved;
- If the Message Bus framing respects the rules defined by the specification[6]:

1. All zeroes must be driven on the message bus when idle. [6]

2. All idle to non-idle transition indicates the start of a transaction; a new transaction can start immediately the cycle after the end of the previous transaction without an intervening idle. [6]
3. The number of cycles to transmit a transaction depends on the command. [6]
4. The cycles associated with one transaction must be transferred in contiguous cycles. [6]

Once an error is detected, a packet is generated and it is logged into memory. Since a command message packet and an error packet are likely created concurrently, they are saved into two different RAMs.

5.5 Data Logic

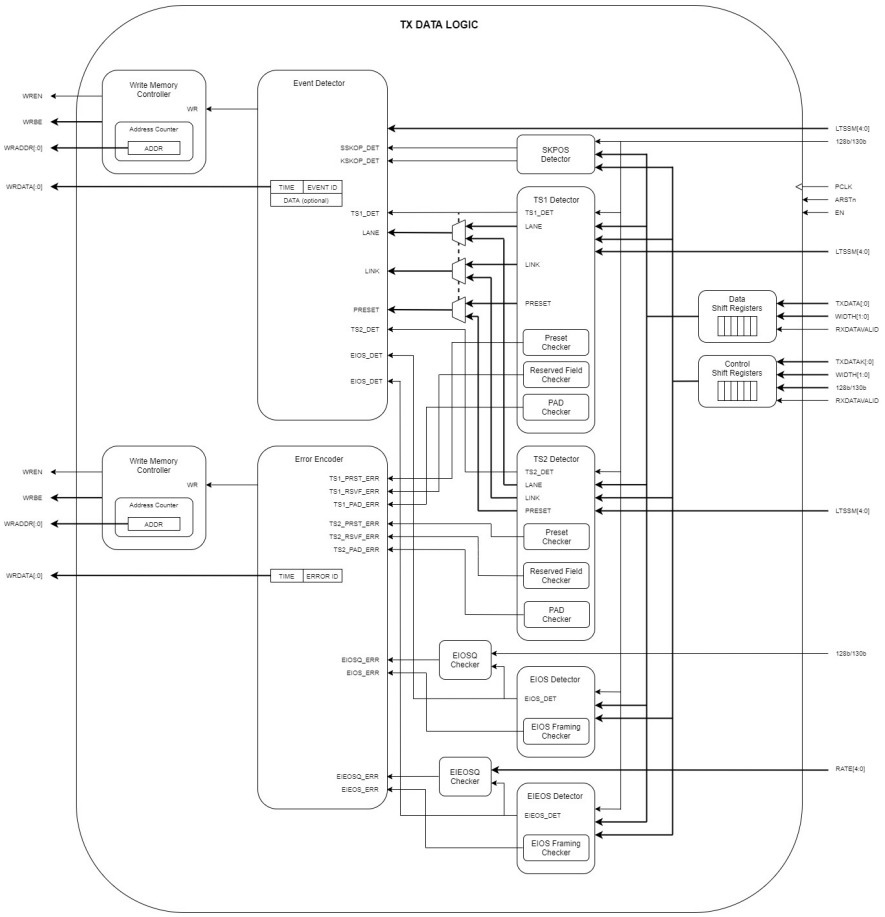


Figure 5.11: Data Logic module

Data Logic is the most complex logic block since it is in charge of monitoring and checking the traffic through RxData and TxData. It must indeed detect some key events, such as Symbol Lock and Speed Change.

Data is elaborated several times before being logged. Firstly, signals from TxData and RxData are sampled and processed by different decoders that detect any ordered set. In case of detection, a pulse is sent to the Event Detector.

The event detector is in charge of putting together all the information to identify an event. It looks at the data transmitted with the ordered set and it counts their number of occurrences, as well. This information is then encoded in a packet and saved into memory.

At the same time, the Ordered Set encoders check the sequence of symbols and if they find an error, they send a pulse to the Error Detector. This module is in charge of creating an error packet when a pulse is received by decoders. It must also check several rules, such as Electrical Idle Entry Rules, Data Rate Change and Determination Rules or SKPOS Transmission Rules.

As for the Message Bus Logic, the monitoring and the checking blocks are separated and they point to two different RAMs, so that an error is correctly logged even if an event occurs at the same moment. And vice versa.

5.5.1 Shift Registers

Most of the ordered sets are not identifiable by looking at the first symbol. Hence,a simple combinational block is not enough to detect them.

At first, a finite state machine could seem as a good solution, because they can easily detect a specific pattern without logging all the symbol sequence. Unfortunately, as mentioned before, the Link speed usually changes during the initial Link Training and Re-Training. The Link speed, for its part, affects the PIPE data width, because, when the speed is lower, the PHY and the MAC do not use all the wires of TxData and RxData.

Looking at the PIPE data width signal (pl_width), the finite state machine is able to select which wire it must take into account. However, it would need a lot of states to handle all the possible configurations.

On the other hand, a shift register can easily handle all the possible widths, by only switching the number of bits to shift. Once all the needed symbols are stored, the detector can easily check for an ordered set using simple combinational logic.

5.5.2 Data Packet

In some cases, it is necessary to have some relevant data as payload of the packet. This need forces to have packets of different sizes. Unfortunately, the implemented Write Memory Controller can handle only packets with the same size.

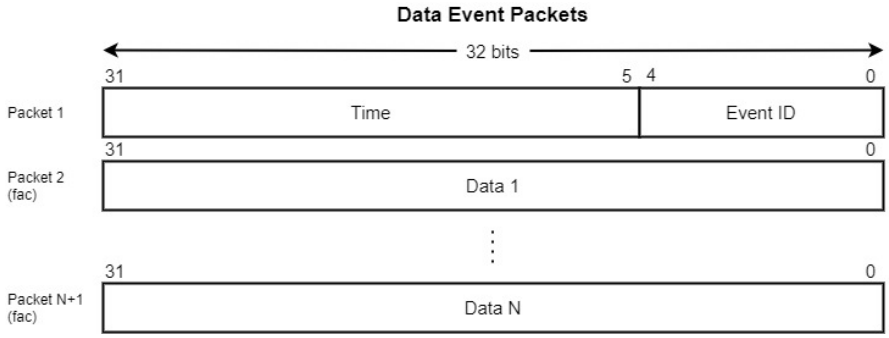


Figure 5.12: Data Event Packets

In order to handle bigger data, the information is split into several 32-bit packets and a buffer is added before the Write Memory Controller to handle the bigger streaming demand. On the software side, when a packet includes a particular ID event, the application knows that the payload is contained in the packets of the next addresses.

Encoding	**Event**
5'h00	Receiver Detection
5'h01	Start Bit/Symbol Lock Sequence
5'h02	Bit/Symbol Lock Complete Successfully
5'h03	Bit/Symbol Lock Failed
5'h04	Link Number assignation
5'h05	Lane Number Assignation
5'h06	Detect a Possible Speed Change
5'h07	Speed Change Request
5'h08	Unexpected change of Link number
5'h09	Unexpected change of Lane number
5'h0A	EIEOS Detected
Others	Reserved

Table 5.4: Data Event encoding

5.6 Power State Logic

PIPE specification[6] includes even a section related to Power Management. Namely, MAC can direct PHY to a low power state, so as to perform some power saving measure.

Specification[6] defines four PHY standard power states:

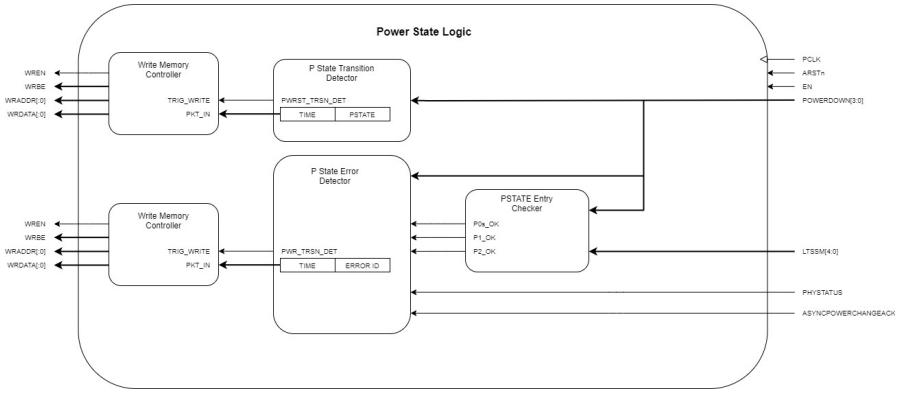


Figure 5.13: Power State Logic Module

P0 state It is the PHY power management state for the Link Training and the normal Link operation.

P0s state PCLK is operational, but the transmit channel is idle.

P1 state Selected internal clocks in the PHY can be turned off, but the PCLK must remain operational. Both receive and transmit channels are idle.

P2 state Selected internal clocks in the PHY and PCLK can be turned off.

PHY can implement specific PHY power management states [6], if they meet all the constraints provided in the PCIe Specification[1].

MAC employs PowerDown and PhyStatus signals to direct the PHY in a new power management state. One of the tasks of Power State Logic is to monitor these signals to detect when a transaction occurs, so as to save it into memory.

In addition, specification provides different rules that PHY must meet to be directed in a different power management state. For instance, it must be in a specific LTSSM state to transit to a low power state. Or PhyStatus must be asserted or deasserted after a particular event.

Hence, Power State Logic must check all these rules and log if an error occurs.

5.7 AXI to UART

As mentioned in section 3.5.2, the data saved into RAMs can be accessed via UART.

Inspector for PCIe by PLDA employs UART too, so it is possible to adapt its UART module for the PIPE Monitor Checker. Namely, the UART Transmitter and Receiver can be used unmodified, since their main task is to serialize or deserialize

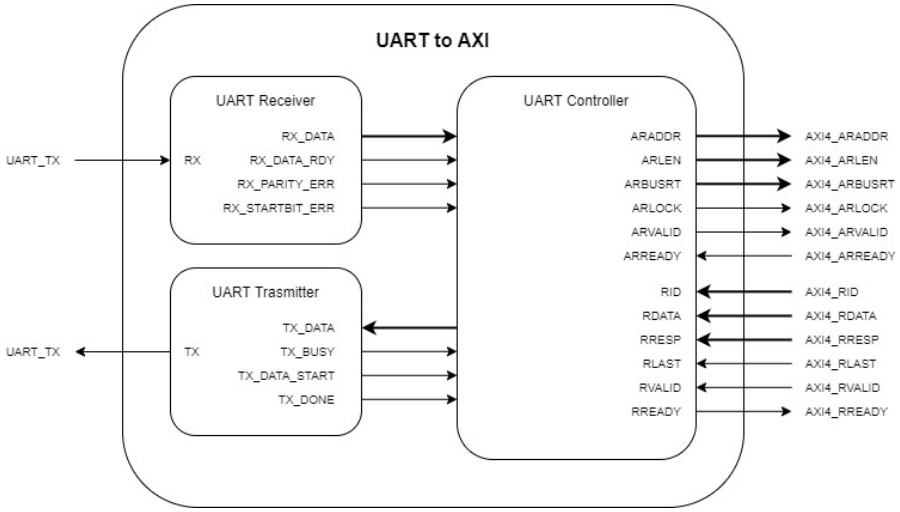


Figure 5.14: AXI to UART module

the streamed data. Instead, the UART Controller is modified to provide an AXI4 interface.

Three main reasons let to employ an AXI interface: The interface from the UART controller of Inspector was not standard. An UART module, that provides a standard interface as AXI, could be more easily reused by PLDA in other IPs. AXI4 Protocol implements Read and Write Bursts, which would improve system reading and writing performance significantly. The AXI RAM Access module derives from a module that has already been implemented for the PLDA AXI BFM and that employs a AXI interface.

As for the Inspector for PCIe, the UART Controller is implemented with a finite state machine. The task of the UART controller is to handle the application protocol to access the memory via UART.

The Memory Access protocol is the same which is employed in the Inspector and that requires to send two packets for each Write/Read access. Namely, the first packet includes the Write or Read command. The second one provides the address. In response, the device answers by transmitting the data. Since the RAM memory width is 512 bit (same as AXI data width), whereas the UART packet is 8-bit, 64 packets are sended in return.

5.8 AXI RAM Access

The goal of AXI RAM Access is to address the Read access to the correct RAM. Which means that it must decode the global memory address received by the UART to AXI module in order to assert the proper Read Enable. At the same time, it

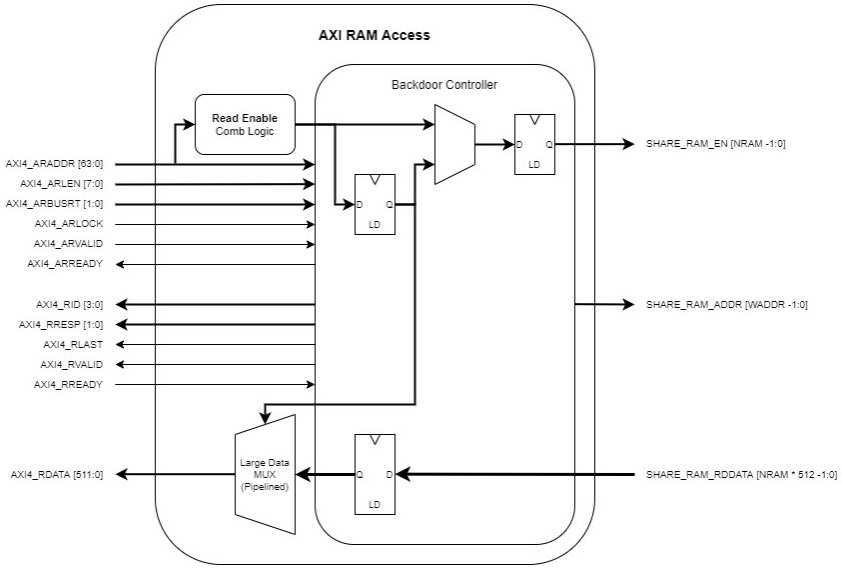


Figure 5.15: AXI RAM Access module

must derive the local address to send with the Read Enable. After that, it must acquire the proper Read Data and send it to the AXI interface.

Ideally, this task could be performed by using some simple combinational blocks, such as multiplexers and decoders. Unfortunately, two main issues require a more complex design:

1. The design timing constraints are quite strict, hence it is necessary to assure that the critical path of the AXI RAM Access does not affect the maximum clock frequency at all.

2. The number of RAMs or their sizes will likely change in the future, because of optimizations or new feature implementations. If the RAM size changes, then the decoder must be modified, as well. Whereas if it is the number of RAMs that changes, then multiplexers and decoders must be adapted, hence almost all the work must be redone.

At PLDA, they usually use Verilog templates to solve issues related to designs that are affected by different parameters. In short, they write a first draft of the module, which is then elaborated automatically by a script. According to some particular parameters, the script adds or modifies some lines and creates the final Verilog file.

Unfortunately, the time to develop the prototype is not enough to apply this method, because it required a lot of time to program the script and in particular

to test it. Hence, as for other modules, parameters and generate blocks have been exploited to have a customizable module.

Five parameters are at the base the customization:

MEM_ADDR_WIDTH_MAX It is the depth of the biggest RAM. Hence, it is as well the address width of the biggest RAM.

RAM_1_N It is the number of RAMs that has a depth equal to $MEM\_ADDR\_WIDTH\_MAX$.

RAM_2_N It is the pairs of RAMs that have a depth equal to $MEM\_ADDR\_WIDTH\_MAX - 1$.

RAM_4_N It is the quartets of RAMs that have a depth equal to $MEM\_ADDR\_WIDTH\_MAX - 2$.

RAM_8_N It is the quartets of RAMs that have a depth equal to $MEM\_ADDR\_WIDTH\_MAX - 3$.

This escamotage allows to easily derive the width of the global address and it significantly simplifies the decoding of the global address.

Let's take an example. If the design needs two 6-bit RAMs, two 5-bit RAMs and four 4-bit RAMs. Then, the parameters will be:

$$\begin{cases} MEM\_ADDR\_WIDTH\_MAX = 6 \\ RAM\_1\_N = 2 \\ RAM\_2\_N = 1 \\ RAM\_4\_N = 1 \\ RAM\_8\_N = 0 \end{cases} \tag{5.3}$$

From this data, it is easy to compute the width of the global address (called for this example gl_addr)

$$\begin{aligned} GL\_ADDR\_WIDTH &= MEM\_ADDR\_WIDTH\_MAX + \\ &+ log_2\left(RAM\_1\_N + RAM\_2\_N + RAM\_4\_N\right) = \\ &= 6 + log_2(2 + 1 + 1) = 8 \end{aligned} \tag{5.4}$$

In addition, an easy relation can be determined between the global address and the RAMs:

- If $gl\_addr[7:6] = 00_2$, it points to the first 6-bit RAM.
- If $gl\_addr[7:6] = 01_2$, it points to the second 6-bit RAM.
- If $gl\_addr[7:6] = 10_2$, it points to the pair of 5-bit RAMs. Then, the decoder checks gl_addr[5] to find out which one is involved.

- If $gl\_addr[7:6] = 11_2$, it points to the quartet of 4-bit RAMs. Then, the decoder checks gl_addr[5:4].

The local address can be derived from the global address by considering only the least significant bits.

This decoding algorithm is generalized and written in Verilog.

Listing 5.1: Decoder to generate Read Enable

```
1  //----------------------------------------------------------------
2  // Combinational Logic to Compute RAMs' Read Enable Signals
3  //----------------------------------------------------------------
4  genvar i, j, k, l;
5
6  generate
7    // Enable for with ADDR_WIDTH = MEM_ADDR_WIDTH_MAX
8    for(i=0; i < RAM_1_N; i=i+1) begin
9      assign shared_ram_rden_c [i] = (araddr[MEM_ADDR_WIDTH_MAX +:
       ADDR_EXTRA_WIDTH] == i);
10   end
11
12   // Enable for with ADDR_WIDTH = MEM_ADDR_WIDTH_MAX - 1
13   for(j=0; j < RAM_2_N; j=j+1) begin
14     assign shared_ram_rden_c [j + RAM_1_N     ] = araddr[
       MEM_ADDR_WIDTH_MAX +: ADDR_EXTRA_WIDTH] == (RAM_1_N + j) && araddr
       [MEM_ADDR_WIDTH_MAX - 1 +: 1] == 1'b0;
15     assign shared_ram_rden_c [j + RAM_1_N + 1] = araddr[
       MEM_ADDR_WIDTH_MAX +: ADDR_EXTRA_WIDTH] == (RAM_1_N + j) && araddr
       [MEM_ADDR_WIDTH_MAX - 1 +: 1] == 1'b1;
16   end
17
18   // Enable for with ADDR_WIDTH = MEM_ADDR_WIDTH_MAX - 2
19   for(k=0; k < RAM_4_N; k=k+1) begin
20     assign shared_ram_rden_c [k + RAM_1_N + RAM_2_N     ] = araddr[
       MEM_ADDR_WIDTH_MAX +: ADDR_EXTRA_WIDTH] == (RAM_1_N + RAM_2_N + k)
       && araddr[MEM_ADDR_WIDTH_MAX - 2 +: 2] == 2'b00;
21     assign shared_ram_rden_c [k + RAM_1_N + RAM_2_N + 1] = araddr[
       MEM_ADDR_WIDTH_MAX +: ADDR_EXTRA_WIDTH] == (RAM_1_N + RAM_2_N + k)
       && araddr[MEM_ADDR_WIDTH_MAX - 2 +: 2] == 2'b01;
22     assign shared_ram_rden_c [k + RAM_1_N + RAM_2_N + 2] = araddr[
       MEM_ADDR_WIDTH_MAX +: ADDR_EXTRA_WIDTH] == (RAM_1_N + RAM_2_N + k)
       && araddr[MEM_ADDR_WIDTH_MAX - 2 +: 2] == 2'b10;
23     assign shared_ram_rden_c [k + RAM_1_N + RAM_2_N + 3] = araddr[
       MEM_ADDR_WIDTH_MAX +: ADDR_EXTRA_WIDTH] == (RAM_1_N + RAM_2_N + k)
       && araddr[MEM_ADDR_WIDTH_MAX - 2 +: 2] == 2'b11;
24   end
25
26   // Enable for with ADDR_WIDTH = MEM_ADDR_WIDTH_MAX - 3
27   for(l=0; l < RAM_8_N; l=l+1) begin
```

```
28      assign shared_ram_rden_c [1 + RAM_1_N + RAM_2_N + RAM_4_N     ] =
        araddr[MEM_ADDR_WIDTH_MAX +: ADDR_EXTRA_WIDTH] == (RAM_1_N +
        RAM_2_N + RAM_4_N + 1) && araddr[MEM_ADDR_WIDTH_MAX - 3 +: 3] ==
        3'b000;
29      assign shared_ram_rden_c [1 + RAM_1_N + RAM_2_N + RAM_4_N + 1] =
        araddr[MEM_ADDR_WIDTH_MAX +: ADDR_EXTRA_WIDTH] == (RAM_1_N +
        RAM_2_N + RAM_4_N + 1) && araddr[MEM_ADDR_WIDTH_MAX - 3 +: 3] ==
        3'b001;
30      assign shared_ram_rden_c [1 + RAM_1_N + RAM_2_N + RAM_4_N + 2] =
        araddr[MEM_ADDR_WIDTH_MAX +: ADDR_EXTRA_WIDTH] == (RAM_1_N +
        RAM_2_N + RAM_4_N + 1) && araddr[MEM_ADDR_WIDTH_MAX - 3 +: 3] ==
        3'b010;
31      assign shared_ram_rden_c [1 + RAM_1_N + RAM_2_N + RAM_4_N + 3] =
        araddr[MEM_ADDR_WIDTH_MAX +: ADDR_EXTRA_WIDTH] == (RAM_1_N +
        RAM_2_N + RAM_4_N + 1) && araddr[MEM_ADDR_WIDTH_MAX - 3 +: 3] ==
        3'b011;
32      assign shared_ram_rden_c [1 + RAM_1_N + RAM_2_N + RAM_4_N + 4] =
        araddr[MEM_ADDR_WIDTH_MAX +: ADDR_EXTRA_WIDTH] == (RAM_1_N +
        RAM_2_N + RAM_4_N + 1) && araddr[MEM_ADDR_WIDTH_MAX - 3 +: 3] ==
        3'b100;
33      assign shared_ram_rden_c [1 + RAM_1_N + RAM_2_N + RAM_4_N + 5] =
        araddr[MEM_ADDR_WIDTH_MAX +: ADDR_EXTRA_WIDTH] == (RAM_1_N +
        RAM_2_N + RAM_4_N + 1) && araddr[MEM_ADDR_WIDTH_MAX - 3 +: 3] ==
        3'b101;
34      assign shared_ram_rden_c [1 + RAM_1_N + RAM_2_N + RAM_4_N + 6] =
        araddr[MEM_ADDR_WIDTH_MAX +: ADDR_EXTRA_WIDTH] == (RAM_1_N +
        RAM_2_N + RAM_4_N + 1) && araddr[MEM_ADDR_WIDTH_MAX - 3 +: 3] ==
        3'b110;
35      assign shared_ram_rden_c [1 + RAM_1_N + RAM_2_N + RAM_4_N + 7] =
        araddr[MEM_ADDR_WIDTH_MAX +: ADDR_EXTRA_WIDTH] == (RAM_1_N +
        RAM_2_N + RAM_4_N + 1) && araddr[MEM_ADDR_WIDTH_MAX - 3 +: 3] ==
        3'b111;
36    end
37  endgenerate
```

On the other hand, in order to solve the timing constraint issues, pipelines have been introduced both in the decoder and in the multiplexer.

5.9 Internal Registers

Some internal registers have been added to provide some status information about the PIPE Monitor Checking. In the future releases, these registers might include even some control bits to configure the device via software.

As RAMs, these registers are accessible via UART. The internal registers list is shown in table 5.5.

LAST_WRITE register includes all the last addresses that have been written

Row Name	Address	Size	Description
PIPE_INFO	0x00	512	**Parameters of the PIPE interface:** 511:29 reserved 28 G_PIPE_INTF 20:27 G_PCIE_NUM_LANES 19:0 G_PIPE_WIDTH
LAST_WRITE	0x01	512	**Last Written Address:** 511:16 reserved 15:12 Message Bus 11:8 Data Events 7:4 Reported Errors 3:0 LTSSM
TEST_01	0x02	512	**Debugging Test Pattern 1**
TEST_02	0x03	512	**Debugging Test Pattern 2**

Table 5.5: Internal Registers

by the monitoring and checking logic blocks. The software application checks this register to determine which RAM address it must read.

TEST_01 and TEST_02 are registers that contain well-known patterns. There are useless during the normal PIPE Monitor operation, but they are employed during the hardware test to check that the UART works properly and it is possible to correctly read data from memory.

5.10 IP Verification

Some unit tests on the main components are performed to assure that they can handle some specific use cases. However, it is not enough to assure the correct operation of the PIPE Monitor Checker.

More comprehensive tests must be performed, hence it has been integrated in the reference design of the XpressRICH-AXI[1] from PLDA. Verification team has indeed already implemented a verification environment that can be adapted for the PIPE Monitor Checker.

The verification environment of XpressRICH-AXI has an architecture that reflects the main rules of the Universal Verification Methodology.

[1]**XpressRICH-AXI™** is a configurable and scalable PCIe controller Soft IP that provides a configurable, flexible AMBA AXI interconnect interface to the user. [18]

5.10.1 Universal Verification Methodology (UVM)

Universal Verification Methodology is a standard that has been defined to reduce the development time of an IP, since it enables designing a modular, scalar and reusable testbench. [22]

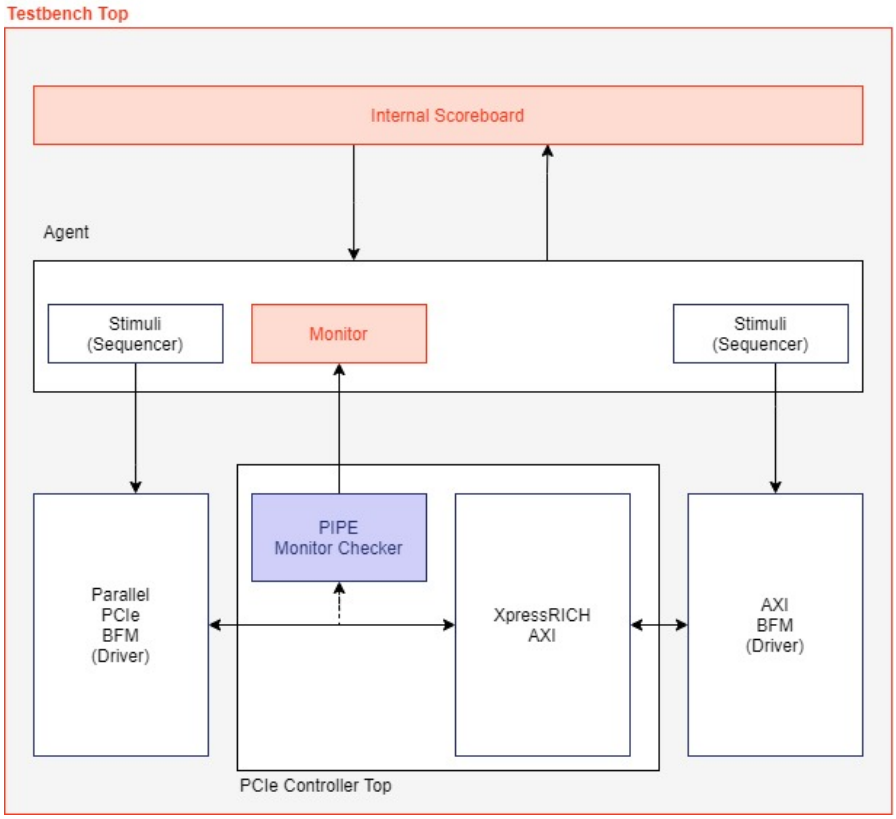


Figure 5.16: Verification Environment

The main UVM components are described hereafter.

Driver is the component that handles a particular interface of the design. It knows the target protocol and it emulates the device that should communicate with the DUT. [23]

Sequencer is in charge of generating the requests and responses that the driver must send to the DUT. [24]

Monitor is the passive component that captures data from the DUT to check its operation. [25]

Scoreboard is the verification component that checks the operation of DUT. It

takes the data from the monitor and it compares them with a reference model. Eventually, it informs if the test is passed or failed. [26]

5.10.2 Verification Environment

The verification environment is presented in figure 5.16.

Drivers and sequencers are the same from XpressRICH-AXI verification environment. Instead, the monitor and the scoreboard are designed ad hoc to verify the functioning of PIPE Monitor Checker.

5.11 Hardware Validation

During the hardware validation, as for the verification phase, the PIPE Monitor Checker observes the traffic generated by the XpressRICH-AXI, even.

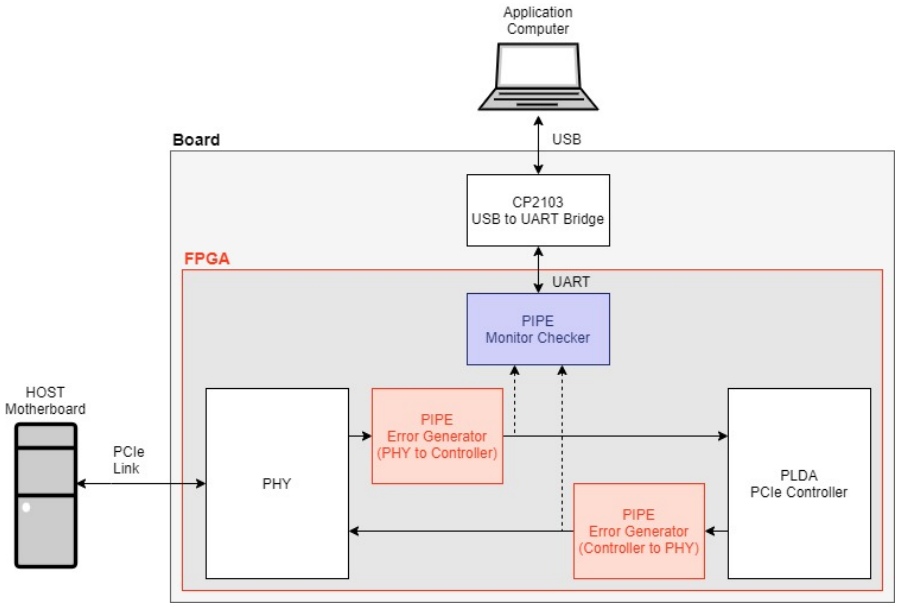


Figure 5.17: Hardware Validation Configuration

The first test is performed to check the correct operation of the UART. Hence, the PIPE Monitor Checker application is not used and the packets are sent using a simple serial terminal, called RealTerm.

At first, a read request packet is sent. An ACK packet in return confirms that the IP Core and the serial terminal share the same UART configuration. Consequently, an address packet that points to TEST_01 and TEST_02 registers is transmitted. If the IP Core responds by sending the correct pattern, then it is

possible to assume that the memory reading via UART works properly and the software application can be used to test the other functionalities.

The validation environment includes a module between PHY and MAC, called PIPE Error Generator. Its task is to force signals and emulate possible PIPE interface errors. However, since the first prototype of PIPE Monitor Checker implements mostly monitoring functionalities. The Error Generator development is contemplated for future releases.

5.12 Conclusion

The basic functionalities of the IP core have been developed and tested in simulation using QuestaSim. It supports all the different lane configurations, PCIe versions and PIPE widths that have been defined in section 3.2.2. However, the verification environment is not complete yet. In particular, the automatic checking performed by the scoreboard is still to be implemented.

The reference design of the XpressRICH-AXI with a PIPE Monitor Checker integrated in its top module has been successfully synthesized for a single lane Gen4 configuration.

The resources exploits by the PIPE Monitor Checker for this configuration are:

- 438 LUTs[2] out of the 124465 employed by the entire reference design;
- 126 FF[3] out of the 126826 employed by the entire reference design;
- 10 kB RAM;

Hence, this tool does not affect the resources exploited by the entire reference design significantly. The constraint defined in section 3.2.2 is met.

The synthesized design has been eventually programmed in the Xilinx FPGA. The memory access has been tested successfully using RealTerm. Unfortunately, due to the end of the internship, the hardware validation is still incomplete.

[2]**LUT** Look-Up Table

[3]**FF** Flip Flop

Chapter 6

Development of the Software Application

This chapter describes the software application of the PIPE Monitor Check. In particular, it firstly explains more deeply how the protocol application to access the IP core memory works. Then, it illustrates the software architecture which is based on Model-ModelView-View (MVVM) design pattern.

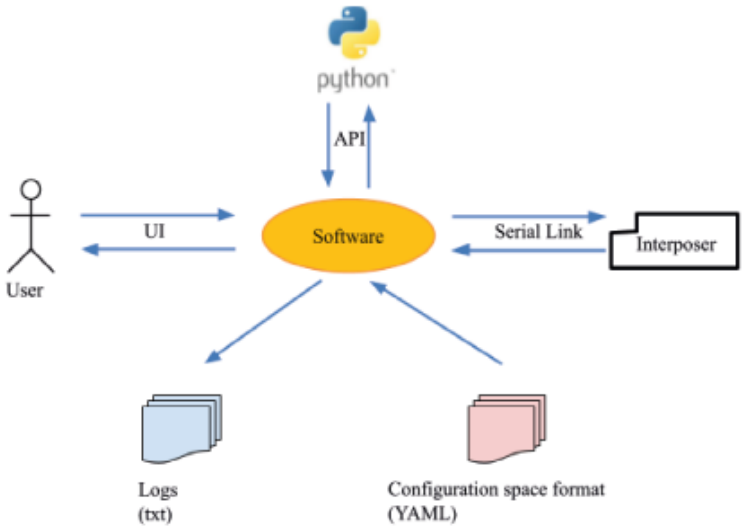


Figure 6.1: Interface of Inspector Software [27]

The goals of the PIPE Monitor Checker application are accessing the data inside of the IP core and presenting this information to the user. Since an application that provides these same functionalities has been already implemented for the Inspector for PCIe, it has been decided to adapt the Inspector software to save time.

The software is written in C#, based on the .NET framework v4.5 and Windows Presentation Foundation (WPF) User Interface (UI) framework. *(Inspector Software Specification [27])*

6.1 Memory Access Protocol

As mentioned in section 5.7, PLDA has implemented an applicative protocol to access the internal registers of the Inspector for PCIe via UART. PIPE Monitor Checker employs the same exact protocol to access the RAMs of the IP core.

In this protocol, the software application acts as master and it sends Write/Read requests to the IP core. Each request is made up of different 8-bit packets that are sent one by one. IP core checks parity and sends a completion packet for each received byte to inform whether the transmission has succeeded. In case of failure, the access is cancelled and the application must repeat the request. [27]

In general, execution flow follows three phases. Firstly, the communication is idle and the software waits until the serial port is connected. Then, a synchronization phase is executed to reset the UART controller of the IP core. Eventually, the software can send the write and read requests. [27]

Name	**Length**	**Description**
SYNC_1	1 byte	First message sent to initiate synchronization between the software and the Interposer. **Content: 0xFF**
SYNC_2	1 byte	Second message sent to initiate synchronization between the software and the Interposer. **Content: 0xFE**
SYNC_3	1 byte	Third message sent to initiate synchronization between the software and the Interposer. **Content: 0xFD**
SYNC_4	1 byte	Fourth message sent to initiate synchronization between the software and the Interposer. **Content: 0xFC**
WRITE	1 byte	Message sent to initiate the write request between the software and the Interposer. **Content: 0x00**
READ	1 byte	Message sent to initiate the read request between the software and the Interposer. **Content: 0x01**
ACK	1 byte	Message sent to acknowledge requests from software Bit 0: set to 1 if a parity error is detected by HW in the software request. Bit 1: set to 1 if a start bit error is detected by HW in the software request. Bit 2..7: reserved.

Table 6.1: Protocol Words [27]

6.1.1 Synchronization phase

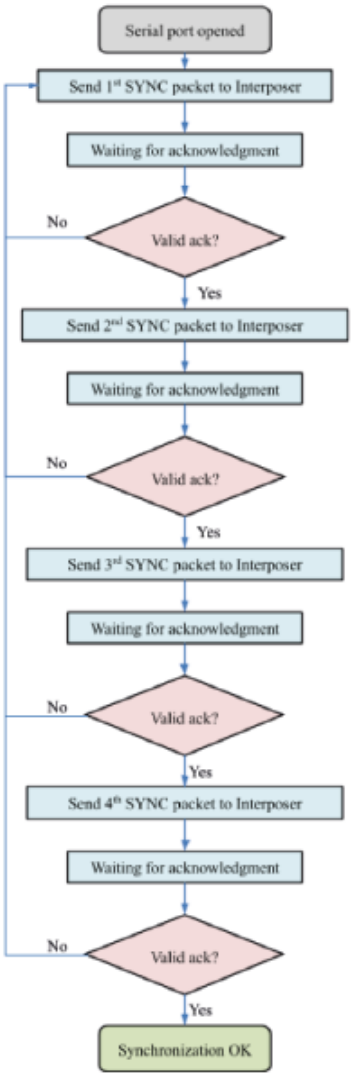


Figure 6.2: Synchronization process [27]

During this phase, the software sends a sequence of four SYNC packets. If all the four transmissions succeed and no error is detected, then the synchronisation is achieved and the UART controller of the IP core resets. [27]

The software can start a synchronization at any time to reset the UART controller, even in the middle of a read access. [27]

6.1.2 Read Access

The software must send a Read command to start a read access. Then, it transmits the address and it waits to receive the packets with the requested data. As mentioned before, read and address packets are checked and a completion is transmitted in return. If an error occurs, the access must be performed a second time. [27]

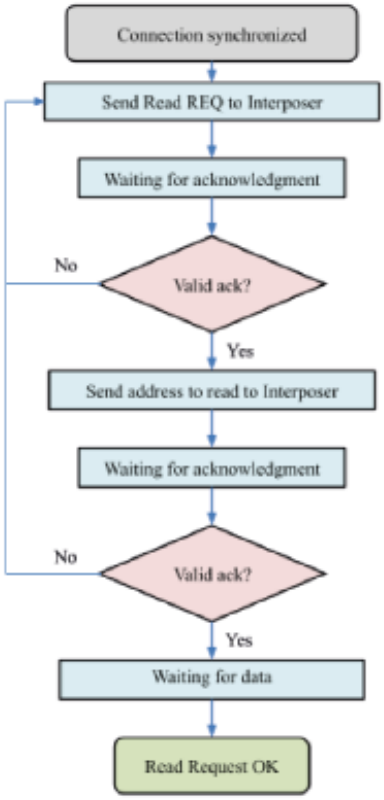


Figure 6.3: Read access [27]

6.2 Software Architecture

The software architecture is based on Model-ModelView-View (MVVM), an architectural pattern that is employed in an application to separate the development of user-interfaces (UI) from that of the business logic and behaviour. [28]

Three components are at the core of MVVM.

Model is the object that contains the domain-specific data or information, but they do not handle any formatting of data. [28]

View is the components which users actually interact with. It is in charge of formatting and presenting data to users. [28]

ModelView is the link between the model and the view. It is a specialized controller that converts the raw data of the Model into a View information and helps to maintain the Model updated. [28]

The Model component is written in C# and it is made up of ".cs" files. The ModelView is composed of ".xalm.cs" files. The View is outlined using XAML, since it allows to describe:

- The windows layout
- The graphical elements (i.e, Text, Buttons)
- The graphic style
- The data binding among the graphical elements

6.2.1 View Description

The view is described by different XAML files. Each file delines a box of the grind in the graphic interface.

For the first release of PIPE Monitor Checker, the graphic interface is as simple as possible. It is split into six boxes and each box shows data from a specific memory space.

6.2.2 Module Description

Main Module

The purpose of the main module and the interaction among their classes have not been changed from the Inspector Software.

This module defines the main view file, aggregating all the others, and the main model. The purpose of the main model is to:

- manage the synchronization process between the Interposer and the software,
- manage the main cycle, send commands to the Interposer and read its register values.

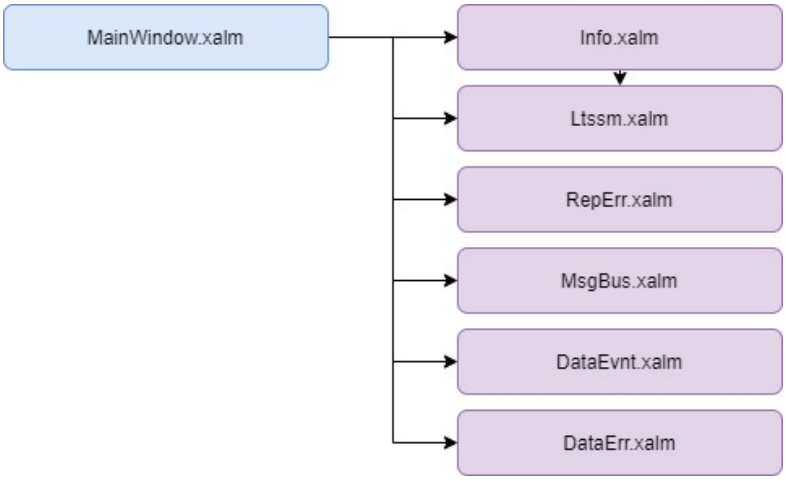


Figure 6.4: XALM files tree

The module contains the following files:

App.xaml: application UI entry point; defines MainWindow as the UI that is automatically shown when an application starts.

App.xaml.cs: application entry point.

BaseUI.xaml: base for all View descriptions.

BaseUI.xaml.cs: defines common properties for all ViewModel classes.

InspectorModel.cs: implements common properties and services for all Model classes.

InspectorProperty.cs: extends INotifyPropertyChanged to implement property updates.

InspectorRegister.cs: defines how to read and write data from/to Interposer.

InspectorRegisterDecoder.cs: abstract class, defines how to read and decode data from Interposer

(Inspector Software Specification [27])

Specific Modules

Instead, some new specific modules are implemented to decode and format the data captured by the IP core.

Let's take as an example the LTSSM module which is in charge of decoding the raw data extracted from the LTSSM RAM and presenting the list of detected transactions in the LTSSM.

Ltssm.xaml: description of the UI (View part).

Ltssm.xaml.cs: interface between the View and the Model.

LtssmStatesDecoder.cs: instantiates LtssmStatesDecoder to decode data into LTSSM memory.

LtssmHelpers.cs: defines all LTSSM states.

LtssmModel.cs: model, instantiates the LTSSM decoder.

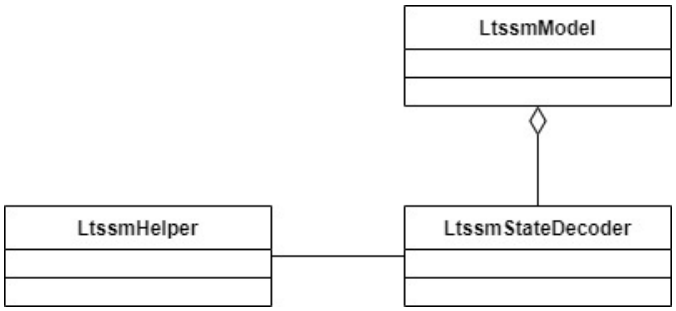


Figure 6.5: LTSSM module class diagram

Serial Interface

This module handles the communication with the IP core via UART. Since the PIPE Monitor Checker employs the applicative protocol of the Inspector for PCIe, this module is almost unchanged.

The modified function is the read access. Since the data width is 512 bit and the UART can transmit only a byte at a time, the application must wait for 64 packets.

Serial Interface contains the following files:

InspectorInterface.cs: Implement UART applicative protocol.

SerialInterface.cs: drives the UART interface.

SerialPortCfg.cs: defines the parameters of the UART configuration.

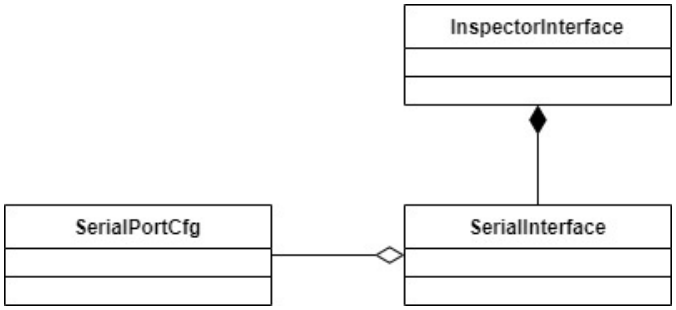


Figure 6.6: Serial Interface module class diagram [27]

6.3 Conclusion

Due to lack of time, the development of the user interface has not been finalised yet. Namely, most of the decoders have been already implemented, but all the view files must be entirely written. In addition, no tests were carried out yet.

However, since the software architecture is already designed and the view files from the Inspector Software Application can be taken as an example to write the new ones, its finalisation would require approximately one month.

Chapter 7

Conclusion

The state of the art about hardware debugging has shown how several methodologies have been developed to simplify this process. However, when a more complex design is involved, tracking back the cause of an error can take several weeks, because the amount of signals related to a single function can be huge.

To solve this problem, the dissertation has presented a solution to monitor and check a specific interface. By knowing the behaviour of this interface, the module is able to detect a set of events, or to check some possible errors that might occur. The memory space is thereby optimized, since the module saves only interesting data.

In addition, even when the provided information is not enough to establish the exact signal that causes the malfunctioning, the hardware debugging iterations made with an Embedded Logic Analyzer (such as ILA or SignalSnap) are significantly reduced, since the group of signals that are likely the cause of the error is highly bounded.

7.1 Future Improvements

The absence of a user interface significantly jeopardises the usability of the tool, because the user is forced to manually analyze and decode the saved data. Hence, the completion of the software application would be certainly the first task to fulfil, since it would finally provide a first usable prototype.

It is also essential to check the functioning of the tool. Thus, the verification environment should be complete and some new use cases should be implemented. Furthermore, once the user interface is available, a comprehensive hardware validation should be performed.

Some modules of the IP modules could be even optimized to reduce the employed resources, for instance the Message Bus Logic could become quite consistent, since

the current solution duplicates the module foreach lane.

Even the AXI RAM Access could be improved, namely a new algorithm could be designed to have more flexibility in the choice of connected RAMs. Speaking of memory,a more detailed study should be conducted to define the proper size of RAMs to allocate. At the moment, a default value is assigned to all the RAMs' depths and no test has been conducted to check if the memory is enough to monitor the period of Link Training and Re-Training.

The study about functionalities described in section 3.3 has presented several new possible features that could be likely developed in the future. Thanks to the modularity of the architecture, their implementation should not require much time. The core structure is already working, hence only the detectors must be improved.

At the end of the project, the tool was presented to the engineers of PLDA and it aroused interest from engineers of the Advanced Design & Integration team, who proposed to integrate it into the XpressAGENT. Hence, in the future, the PIPE Monitor Checker could be sold with the XpressAGENT to PLDA's clients to simplify the testing of ASIC solutions.

Appendix A

AMBA AXI protocol

The Advanced extensible Interface (AXI) protocol is part of the ARM Advanced Microcontroller Bus Architecture (AMBA) specifications, which is an on-chip interconnect standard introduced by ARM.

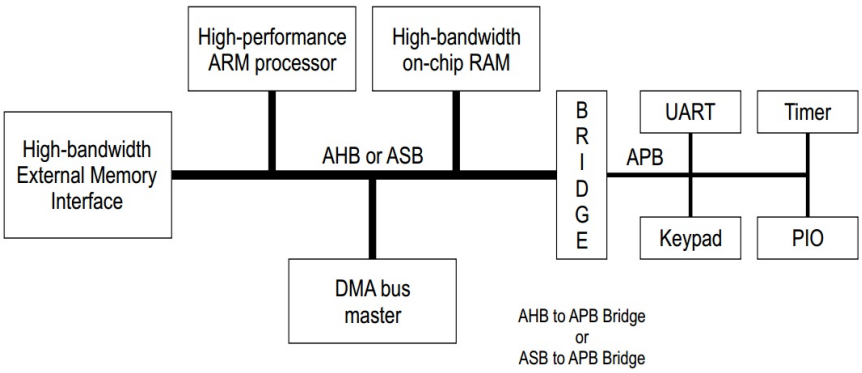


Figure A.1: AMBA-based SoC architecture [29]

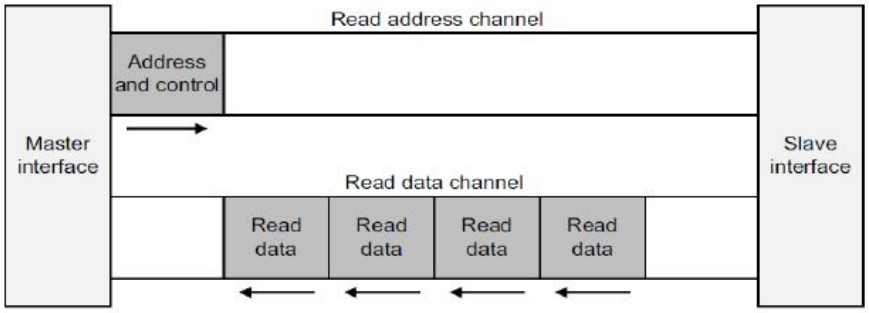
The objective of AMBA specification is to make an IP technology independent, so as to be reused across multiple designs. In addition, it offers a large flexibility to work with different SoCs, since it supports different power, performance and area requirements. [29]

Besides the AXI protocol, AMBA family includes the Advanced Peripheral Bus (APB) protocol and the Advanced High-performance Bus (AHB) protocol. APB is the simpler one. It is designed for low bandwi dth control accesses (such as register interfaces) and it has low-power consumption. Instead, AHB is pipelined and it is used for memory access, so it supports wide data bus configuration. [29]

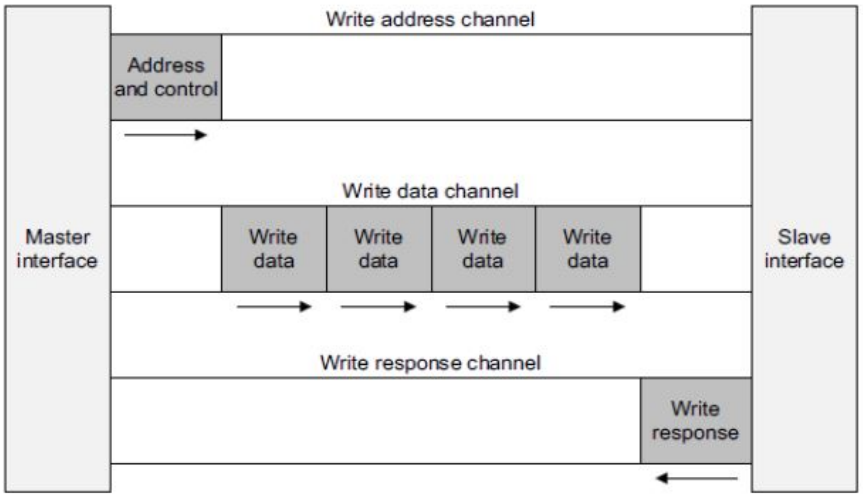
The AXI protocol has been invented to meet the demand for more bandwidth. Whereas AHB is a single channel bus, AXI is a multi-channel bus and it supports

full-duplex mode. The communication is indeed handles using five independent channels:

- Read address channel
- Read data channel
- Write data channel
- Write address channel
- Write response channel



(a)



(b)

Figure A.2: (a)AXI Read transaction.[30] (b) AXI Write transaction.[30]

The AXI protocol is strongly transaction-oriented and its topology is based on master and slave devices. The master is the component that initiates the transaction, whereas the slave is the components that receives transactions and responds to them, for instance a microcontroller can be a master component and it is connected to a memory which is the slave.

> "It is possible that a single component can act as both a slave component and as a master component. For example, a Direct Memory Access (DMA) component can be a slave component when it is being programmed and a master component when it is initiating transactions to move data." *(AMBA AXI and ACE Protocol Specification. ARM. [30])*

Each AXI transaction complies with a two-way handshakes mechanism which is based on two signals: VALID and READ. the master asserts the VALID signals to inform the slave that new address/data are available.[30] The slave asserts the READY signal to indicate when it is in condition to accept a new incoming information. The transaction takes place when both VALID and READY signals are asserted. [30]

AXI protocol specification [30] defines some simple rules:

- A source can assert VALID before READY is asserted.
- Once VALID is asserted, it must remain asserted until the handshake occurs and the source must keep its information stable.
- A destination can wait for VALID to be asserted before asserting the corresponding READY.
- If READY is asserted, it is possible to deassert READY before VALID is asserted.

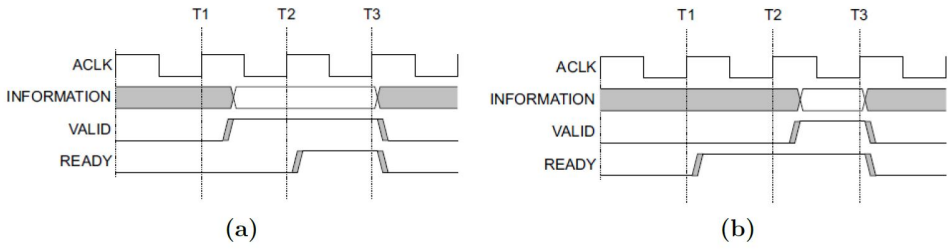


Figure A.3: AXI handshake examples.[30]

Bibliography

[1] *PCI Express Base Specification Revision 5.0*. PCI-SIG. May 2019 (cit. on pp. 1, 8, 26, 49).

[2] Mike Jackson and Ravi Budruk. *PCI Express Technology. Comprehensive Guide to Generations 1.x, 2.x and 3.0*. MindShare, Inc., Sept. 2012 (cit. on pp. 1–5, 7, 9, 26).

[3] *PCI Express Market by Application and Geography. Forecast and Analysis 2020-2024*. Tech. rep. Technavio, Apr. 2020 (cit. on p. 2).

[6] *PHY Interface for the PCI Express, SATA, USB 3.1, DisplayPort and USB Architectures. Version 5.2*. Intel Corporation. Aug. 2019 (cit. on pp. 11, 12, 26, 43, 45, 46, 48, 49).

[7] *In-Circuit FPGA Debug. Challenges and Solutions*. Tech. rep. Microsemi, Mar. 2014 (cit. on pp. 13, 15).

[8] Ehab Alfons Anis Daoud. «On-Chip Debug Architectures for Improving Observability during Post-Silicon Validation». PhD thesis. Hamilton, Ontario, Canada: McMaster University, Sept. 2008 (cit. on pp. 14–16).

[9] *Simplifying Xilinx and Altera FPGA Debug. Application Note*. Tech. rep. Techtronix (cit. on pp. 16–18).

[10] *Arm® CoreSight™ ELA-600 Embedded Logic Analyzer. Technical Reference Manual*. Arm. Aug. 2020 (cit. on p. 16).

[11] *Summit T3-16 PCI Express Protocol Analyzer. Quick Start Guide*. LeCroy. 2012 (cit. on p. 17).

[12] *Integrated Logic Analyzer v6.2. LogiCORE IP Product Guide*. Xilinx. Oct. 2016 (cit. on p. 18).

[13] *Inspector for PCIe - 3434. Getting Started Guide*. PLDA. Apr. 2019 (cit. on pp. 19, 20).

[14] *XpressAGENT. Solution Overview*. PLDA. Mar. 2021 (cit. on p. 21).

[15] Eric Ries. *The Lean Startup : How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses*. Currency, Oct. 2011 (cit. on p. 25).

[17] *Systems and Software Engineering. Recommended practice for architectural description of software-intensive systems*. Geneva, Switzerland: ISO/IEC 42010:2007, 2007 (cit. on p. 27).

[19] T. Lavanya N. & Malarvizhi. «Risk analysis and management. A vital key to effective project management». In: *PMI® Global Congress 2008*. Asia Pacific, Sydney, New South Wales, Australia. Newtown Square, PA, Mar. 2008 (cit. on p. 33).

[21] William A. Moylan. «Planning and scheduling. The yin and yang of managing a project». In: *Project Management Institute Annual Seminars & Symposium*. San Antonio, TX. Newtown Square, PA, Oct. 2002 (cit. on p. 35).

[27] *Inspector for PCIe. Software Specification*. PLDA. Apr. 2021 (cit. on pp. 59–62, 64, 66).

[29] *Architetture di bus per Architetture di bus per System-On- Chip*. University of Bologna - Corso di Architettura dei Sistemi Integrati, 2002-2003 (cit. on p. 69).

[30] *AMBA AXI and ACE Protocol Specification. AXI3, AXI4, and AXI4-Lite - ACE and ACE-Lite*. ARM. 2011 (cit. on pp. 70, 71).

Sitography

[4] Niraj Mathur. *Optimizing The Data Center With PCI Express 4.0. While PCIe 4.0 took a long time to get here, there are big benefits ahead*. Dec. 14, 2017. URL: `https://semiengineering.com/optimizing-the-data-center-with-pci-express-4-0/` (cit. on p. 2).

[5] Anton Shilov. *PCIe 6.0 Specification Hits Milestone: Complete Draft Is Ready*. Nov. 4, 2020. URL: `https://www.tomshardware.com/news/pcie-6-specification-hits-milestone-complete-draft-is-ready` (cit. on p. 3).

[16] Richard Becker. *Minimum Viable Product (MVP)*. Aug. 14, 2020. URL: `https://www.techopedia.com/definition/27809/minimum-viable-product-mvp` (cit. on p. 26).

[18] *XpressRICH-AXI Controller IP for PCIe 5.0*. URL: `https://www.plda.com/products/xpressrich5-axi` (cit. on pp. 30, 55).

[20] ADAM HAYES. *Risk Analysis*. Feb. 19, 2021. URL: `https://www.investopedia.com/terms/r/risk-analysis.asp` (cit. on p. 34).

[22] *UVM Tutorial*. URL: `https://www.chipverify.com/uvm/uvm-tutorial` (cit. on p. 56).

[23] *UVM Driver*. URL: `https://www.chipverify.com/uvm/uvm-driver` (cit. on p. 56).

[24] *UVM Sequencer*. URL: https://www.chipverify.com/uvm/uvm-sequencer (cit. on p. 56).

[25] *UVM Sequencer*. URL: https://www.chipverify.com/uvm/uvm-monitor (cit. on p. 56).

[26] *UVM Scoreboard*. URL: https://www.chipverify.com/uvm/uvm-scoreboard (cit. on p. 57).

[28] Addy Osmani. *Understanding MVVM. A Guide For JavaScript Developers*. Apr. 10, 2012. URL: https://addyosmani.com/blog/understanding-mvvm-a-guide-for-javascript-developers/ (cit. on pp. 62, 63).