Software Development for PCI Express and Aurora Protocol

Major Project Report

Submitted in partial fulfillment of the requirements

for the degree of

Master of Technology

in

Electronics & Communication Engineering

(Embedded Systems)

By

Juhi Wagle

(15MECE08)

Electronics & Communication Engineering Department

Electrical Engineering Department

Institute of Technology-Nirma University

Ahmedabad-382 481

May 2017

Software Development for PCI Express and Aurora Protocol

Major Project Report

Submitted in partial fulfillment of the requirements

for the degree of

Master of Technology

in

Electronics & Communication Engineering

(Embedded Systems)

By

Juhi Wagle

(15MECE08)

Under the guidance of

External Project Guide:

Rasesh Dave

Engineer-SD,

Iter India,IPR

Gandhinagar.

Internal Project Guide:

Prof. Akash Mecwan

Assistant Professor, EC Department,

Institute of Technology,

Nirma University, Ahmedabad.

Electronics & Communication Engineering Department

Electrical Engineering Department

Institute of Technology-Nirma University

Ahmedabad-382 481

May 2017

Declaration

This is to certify that

a. The thesis comprises my original work towards the degree of Master of Technology in Embedded Systems at Nirma University and has not been submitted elsewhere for a degree.

b. Due acknowledgment has been made in the text to all other material used.

- Juhi Wagle

15MECE08

Disclaimer

"The content of this paper does not represent the technology,opinions,beliefs, or positions of Iter India (IPR),its employees,vendors, customers, or associates."

Certificate

This is to certify that the Major Project entitled **"Software Development for PCI Express and Aurora Protocol"** submitted by **Juhi Wagle (15MECE08)**, towards the partial fulfillment of the requirements for the degree of Master of Technology in Embedded Systems, Nirma University, Ahmedabad is the record of work carried out by her under our supervision and guidance. In our opinion, the submitted work has reached a level required for being accepted for examination.The results embodied in this major project, to the best of our knowledge,haven't been submitted to any other university or institution for award of any degree or diploma.

Date: Place: Ahmedabad

Prof. Akash Mecwan

Internal Guide

Program Coordinator

Dr. D.K.Kothari

HOD,EC

Dr. Alka Mahajan

Director

Certificate

This is to certify that the Major Project (Phase- I) entitled **"Software Development for PCI Express and Aurora Protocol"** submitted by **Juhi Wagle(15MECE08)**, towards the partial fulfillment of the requirements for the degree of Master of Technology in Embedded Systems, Nirma University, Ahmedabad is the record of work carried out by her under our supervision and guidance. In our opinion, the submitted work has reached a level required for being accepted for examination.

Date:

Project Guide

Mr. Rasesh Dave

Engineer-SD,

Iter India,(IPR)

Gandhinagar.

Place: Ahmedabad

Project Manager

Mr. N. P. Singh

Deputy Project Manager,

Iter India,(IPR)

Gandhinagar.

Acknowledgements

I would like to express my gratitude and sincere thanks to **Dr. P.N.Tekwani**, Head of Electrical Engineering Department, and **Dr. N.P.Gajjar**, PG Coordinator of M.Tech Embedded Systems program for allowing me to undertake this thesis work and for his guidelines during the review process.

I take this opportunity to express my profound gratitude and deep regards to **Prof. Akash Mecwan**, guide of my major project for his exemplary guidance, monitoring and constant encouragement throughout the course of this thesis. The blessing, help and guidance given by him time to time shall carry me a long way in the journey of life on which I am about to embark.

I would take this opportunity to express a deep sense of gratitude to **Rasesh Dave**,Engineer-SD,Iter India(IPR),Gandhinagar. for his cordial support, constant supervision as well as for providing valuable information regarding the project and guidance, which helped me in completing this task through various stages.

Lastly, I thank almighty, my parents, brother and friends for their constant encouragement without which this assignment would not be possible.

- Juhi Wagle

15MECE08

Abstract

This project involves software development for PCI Express communication and Aurora protocol on Xilinxs Zynq-7045 SoC based ZC 706 Evaluation board with Vivado Design Suite.

Study of PCI Express serial bus and software development is targeted for PCI Express communication with a host system. ZC706 Supports PCI Express Gen 1 and Gen 2 Capability with x4, x2, or x1 Gen 1 and Gen 2 lane width. *Communication of multiple ZC706 boards using PCI Express is also envisaged to increase the channel count.

For communication between multiple Zynq boards, Aurora which is a scalable, lightweight, link-layer protocol that is used to move data across point-to-point serial links are used. Aurora provides a transparent interface to the physical layer, allowing upper layers of proprietary or industry-standard protocols to easily use high-speed transceivers.

*Depends on availability of the additional ZC706 board.

Contents

Declaration **iii**

Disclaimer **iv**

Certificate **v**

Certificate **vi**

Acknowledgements **vii**

Abstract **viii**

List of Figures **xiv**

1 Introduction **xv**

1.1 Motivation xv

1.2 Problem Statement xv

1.3 Overview xvi

1.4 Thesis Organization xvi

2 Introduction to Zynq Architecture **1**

2.1 Introduction 1

2.2 Embedded Design Flow using The Zynq-7000 All Programmable SoC 3

2.3 Detail Study of Processing System 4

2.4 Detail Study of Programmable Logic 9

2.5 The AXI Standard 13
2.6 Zynq-7000 and ARM Trust Zone Technology 15

3 PCI Express System Architecture 17
3.1 Introduction 17
3.2 PCIE Terminologies 18
3.3 Introduction to PCI Express Transactions 20
3.4 PCI Express Device Layers 21
3.5 Function of Each PCI Express Device Layer 21
3.6 Transaction Layer Packet Routing Basics 27
3.7 Plug-And-Play Configuration of Routing Options 30
3.8 Introduction to the Packet-Based Protocol 30
3.9 PCI Express Configuration 34

4 Introduction to PCI Tree 40
4.1 Introduction 40
4.2 Explanation of various fields in the window 44

5 7 Series FPGAs Integrated Block for PCI Express 48
5.1 Introduction 48
5.2 Core Interfaces 49
5.3 Detailed Example Design 51

6 ZC706 PCIe Design Creation 61
6.1 ZC706 PCIe Design Creation 61

7 ZC706 PCIe Targeted Reference Design 67
7.1 Data Flow 68
7.2 Hardware Test Setup 69

8 Introduction to Aurora Protocol **79**

8.1 Introduction 79

8.2 8B/10B Data Transmission and Reception 79

8.3 The physical coding sublayer (PCS) 86

8.4 The physical Medium Attachment (PMA) 87

9 Conclusion **88**

10 Future Scope **89**

Bibliography **90**

List of Figures

2.1 Zynq-7000 All Programmable SoC Block Diagram [1] 3

2.2 Embedded Design Flow using The Zynq-7000 All Programmable SoC[1] 3

2.3 I/O Peripheral Interfaces[1] 7

2.4 Programmable Logic in Zynq[1] 9

3.1 PCIE Topology[4] 20

3.2 Block Diagram for PCI Express Device Layer [4] 22

3.3 Structure of TLP at the Transaction Layer [4] 23

3.4 Flow Control Process [4] 25

3.5 TLP and DLLP structure at Data Link Layer[4] 26

3.6 TLP and DLLP structure at Physical Layer[4] 26

3.7 Generic System Memory And IO Address Maps[4] 29

3.8 Example of Memory Write Request TLP[4] 31

3.9 Example of Memory Read Request TLP[4] 32

3.10 Example of Completion TLP[4] 33

3.11 Topology View at Startup[4] 34

3.12 Type0 Configuration Read Request Packet Header[4] 36

3.13 Type0 Configuration Write Request Packet Header[4] 36

3.14 Type 0 PCI Configuration Space Header[4] 36

3.15 Command Register[4] 38

4.1 Main window of PCItree[4] 41
4.2 Memory window of PCItree[4] 42
4.3 Memory map view of PCItree[4] 43
4.4 PCI INT routing of the PCIbus[4] 44
4.5 Tree list[4] 45
4.6 Config Space Dump[4] 46

5.1 PIO Design Component[4] 53
5.2 PIO 64-bit Application[4] 54
5.3 Root Port Model and Top-Level Endpoint[4] 55
5.4 Root Port Model and Top-Level Endpoint[4] 56

6.1 Compiled Example Design [6] 62
6.2 PCI Tree [6] 65

7.1 Zynq-7000 PCIe Targeted Reference Design Block Diagram [7] 69
7.2 ZC706 Board Programming Setup [8] 70
7.3 ZC706 Board TRD Setup in Host Computer [8] 71
7.4 LEDS Showing PCIe Status [8] 71
7.5 Directory Structure of the ZC706 PCIe TRD [8] 72
7.6 Performance Mode GUI [8] 73
7.7 HDMI Display for Color Bar Display [8] 74
7.8 HDMI Display for Sobel Output Display [8] 74
7.9 Performance Mode Plots [8] 76
7.10 Video running on Host [8] 76
7.11 Video running on HDMI Monitor with SOBEL set to None [8] 77
7.12 Video running on HDMI Monitor with HW/SW SOBEL Filter [8] . . 77
7.13 Exiting the Qt GUI [8] 78

8.1 Aurora 8B/10B Channel Overview 80
8.2 Initialization Overview 81

8.3 Channel Bonding Procedure 82
8.4 Channel Verification Procedure 82
8.5 Native Flow Control Command Octet Format 83
8.6 User Flow Control PDU Format 83
8.7 Transmission Procedures 85
8.8 Receive Procedures 86

Chapter 1

Introduction

1.1 Motivation

Multiple chopper configuration is used for high voltage power supplies. In the control mechanism for high voltage power supplies, communication and precise control signals for individual chopper module is required. This leads to control system with the combination of processor and FPGA. Zynq is a new generation of all Programmable System on Chip. It integrates a dual core ARM cortex A9 processor with traditional FPGA logic. PCI Express (Peripheral Component Interconnect Express), officially abbreviated as PCIE, is a high-speed serial computer expansion bus standard. In this project PCIE is used to achieve high speed data transmission.

1.2 Problem Statement

Number of channels that are supported by one ZC706 board is 136 which is quite large and the number of channels that are supported by one chopper module is 3. If three ZC706 boards are used simultaneously so the number of channels will be 3*136 i.e. 408 and with these much ZC706 board channels 136 around chopper modules can be connected. For establishing this Communication of multiple ZC706 boards with host PC, There is a need to use PCI Express to increase the channel count and

to achieve high speed data transmission. I have implemented ZC706 PCIE Targeted Reference Design to establish communication.

1.3 Overview

The study of Zynq all programmable System on Chip and ZC706 board architecture is required for developing the understanding of the complete hardware. According to the problem statement let say data coming from the receive channel of the 136 chopper modules will be sent to the host PC and after setting some parameters in the host PC application (power supply logic will be implemented here, not a part of my project) the modified or processed data will be again sent back to the chopper module using PWM channel of the chopper module.I have implemented ZC706 PCIE Targeted Reference Design to establish communication between host PC and ZC706 board using PCIE extender cable. Through this implementation, studied the hardware design created using Vivado design suite. studied whole hardware architecture and software architecture implemented in this ZC706 PCIE Targeted Reference Design. This study helped me in understanding that how data is passed from host PC to ZC706 board by using PCI Express extender cable and what are the hardware components inside the ZC706 board that are responsible for processing this data. Also understand how this data can be again sent back to the host system. I will use this knowledge in designing hardware block of my project by using Vivado design suite. The integration of PCI Express IP in the hardware design is most essential part of my project.

1.4 Thesis Organization

The rest of the report is organized as follows.

Chapter 2 describes the details of Zynq SOC architecture with Processing System, details of ARM Cortex A9, programmable logic its Special features as BRAM,XADC,

GTX and AXI interconnects with its three variants as AXI Full, AXI Lite, AXI stream.

Chapter 3 gives details about PCI Express technology, its features, terminology, topology, PCI Express transactions, PCI Express device layers, TLP routing and PCI Express configuration.

Chapter 4 gives introduction to PCI Tree. It is a windows utility.It also gives the explanation of each and every window of this utility.

Chapter 5 Describes the 7 Series FPGAs Integrated Block for PCI Express and its interfaces. It also gives explanation and simulation results for the example design for 7 Series FPGAs Integrated Block for PCI Express.

Chapter 6 Describes how to generate x4 Gen2 PCIE core using Vivado Design Suite 2015.4 and the use of PCI Tree in accessing the data from computer memory for the PCIe Endpoint block.

Chapter 7 In this chapter I have mentioned about the implementation of ZC706 PCIE targeted Reference Design. Through this design I successfully established communication between the host PC and ZC706 board by using PCIE extender cable.

Chapter 8 gives introduction to Aurora Protocol.Explains how data is transmitted and received over channels.Also gives explanation of the sublayers like Physical Coding Sublayer (PCS) and Physical Medium Attachment (PMA).

Chapter 9 gives the conclusion of work.

Chapter 10 gives the future scope.

Chapter 2

Introduction to Zynq Architecture

2.1 Introduction

A latest generation of all programmable System on Chip is known as zynq. Zynq connects a dual core ARM Cortex A9 processor with FPGA logic fabric. Here, the ARM Cortex-A9 is an application grade processor which is eligible to run full operating system like Linux. Whereas the programmable logic part is dependent on Xilinx 7-Series FPGA structure. The overall architecture is accomplished with industry standard AXI interfaces, it facilitates provide great bandwidth, minimum latency connections between the programmable logic part and processing system part of the zynq. Along with all these facilities benefits also arises by modifying the system to a single chip which leads to lower physical size and overall price of the device. A System on a Chip (SoC) can also be considered as an integrated IC as it combines many components of a desktop or various electronic system to a single chip. This System on a Chip (SoC) may have radio-frequency features, analog signal, digital signal and mixed signal that too on an individual chip substrate. A System on a Chip (SoC) can integrate all the features of digital system like processing of data, logical interfacing, memory and so forth. Benefits of a System on a Chip (SoC) solutions are less price, capable of speedy and high secure data transfer among many

system elements, it also has more overall speed of the system, less power utilization, occupies less physical dimensions and it is highly trustworthy. Drawbacks of ASICs kind of SoC is that it requires high manufacturing time and budget and they are less flexible. This ASIC based SOC preferred only when large bunch of devices need to be manufactured where in future there is no need of any modification[1].

Zynq also facilitates Fully integrated development tool environment. An integrated development environment (IDE) is a software suite, it contains the general tools that software programmer requires to write and verify software. Generally, a code editor, a compiler or interpreter and a debugger that the designer utilizes with the help of a single graphical user interface (GUI) is contained by an IDE. This IDE can be an individual application, or it can be integrated like a part of many other available and suitable applications[1].

Zynq devices are targeted platform reference designs. Xilinx Targeted Design Platforms (TDPs) are accomplished with boards, IP cores, Design Suite tools, reference designs and FPGA Mezzanine Card (FMC) support, ISE(Integrated Software Environment) making developers to finish application projection as soon as possible[1].

As we know the traditional embedded design flow in which software and hardware design flows are separated. In the embedded design flow using The Zynq-7000 All Programmable SoC separated design flows integrate to a specific target device. Tightly coupled (die-level) PS-PL interface gives high level of performance which cant be achieved by two-chip solutions. The advantages of this SoC are as follows: Low power consumption, less cost, small form factor and many more.Figure 1 shows the Zynq-7000 All Programmable SoC Block Diagram[1].

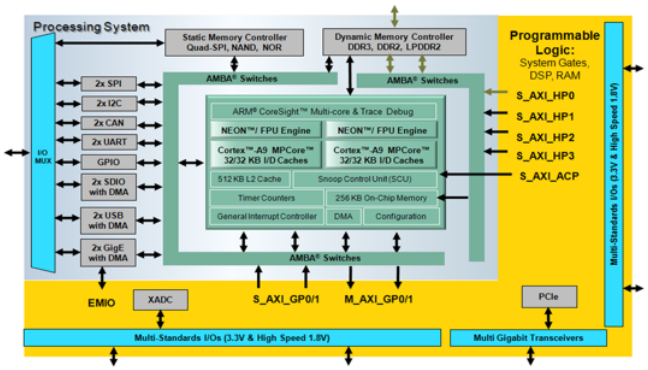


Figure 2.1: Zynq-7000 All Programmable SoC Block Diagram [1]

2.2 Embedded Design Flow using The Zynq-7000 All Programmable SoC

Vivado HLS and System Generator are tools that makes possible the synthesis of hardware logic from higher level coding languages. The Vivado HLS tool enables user to complete his task in C/C++/System C, whereas System Generator permits user to work in MATLAB /Simulink. The Zynq AP SoC is more than just a device. This SoC is a device with an Operating System, applications and libraries. Software tools are present to develop software and IP, whereas hardware development tools can enhance Zynq All Programmable SoC. Reference designs and board support packages also makes SoC suitable to provide a comprehensive solution[1].

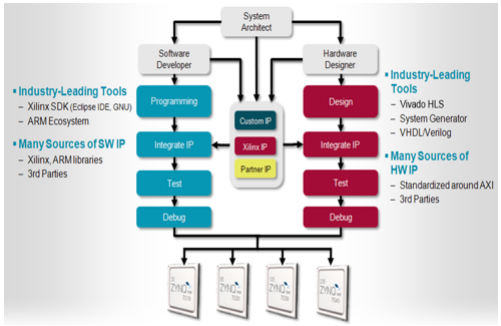


Figure 2.2: Embedded Design Flow using The Zynq-7000 All Programmable SoC[1]

2.3 Detail Study of Processing System

Processing System consists of Dual Core ARM Cortex A9 processor which is a Hard Processor. This processor can operate upto 1 GHz, this operating frequency depends on the particular Zynq device. It is present as a devoted and optimised silicon element on the device. Their performance is also very have high. The Processing System is generally classified into four major functional units[1]:

1. Application Processor Unit (APU)
2. I/O Peripherals (IOP)
3. Datapath (interconnects and interfaces)
4. Memory resources[1]

Other more minor units consists of clock generation, device configuration, reset system. The components of the processing System remain as it is as for all devices in the Zynq AP SoC family, the difference only observed in the speed grade of the processing system. Xilinx Software Development kit (SDK) helps the programmer by providing support for ARM instruction. This SDK contains all important components to write software for deployment on the ARM processor. Compiler Support for the ARM Thumb instruction sets (16 or 32 bit) along with 8-bit Java byte code (for java virtual machine) is provided by the compiler. It is not compulsory that there must be direct correspondence between ARM and Xilinx documentation. For example: ARM Cortex A9 can have processing cores between 1 and 4, while developing Zynq device Xilinx have mentioned a configuration which contains 2 cores. There are many other elements that can be programmed (eg. L1 cache memory size can be specified as 16 KB, 32 KB or 64 KB and Xilinx have selected 32 KB)[1].

1.APU

Application Processing Unit consists of 2 ARM processor cores (CPUs). Each processor core is attached with computational units[1]:

1. NEON Media Processing Engine (MPE) It facilitates to single instruction multiple data (SIMD) to make possible proper acceleration of media and DSP type algorithms. Actually NEON instructions are the advanced version of the standard ARM instruction set. This type of computation is utilized in applications such as image processing, that works on large number of data samples (pixels) simultaneously and inherently parallel, generic signal processing features such as FIR filters and FFTs. NEON provides help by supporting a large number of data types including signed and unsigned integers, single precision floating point, half precision floating point. For double precision computation floating Point unit is required[1]

2. Floating Point Unit (FPU)

3. Memory Management Unit (MMU) This helps in translation of virtual addresses and physical addresses.

4. Level 1 Cache Memory (2 section for instruction and data) 32 KB each

5. Level 2 Cache Memory 512 KB data and instruction

6. On Chip Memory (OCM) 256 KB

7. Snoop Control Unit (SCU) It creates a bridge between the ARM cores and Level 2 cache and OCM memories. This unit also possess some functionality for interfacing with the Programming Logic. SCU also enables the maintenance of memory coherency between processor data cache memories L1 (D) and shared L2 cache memory. It also maintains transaction that occurs between Processing System and Programmable Logic with the help of the Accelerator Coherency Port (ACP)[1].

8. Direct Memory Access Unit (DMA)

9. Cache Controllers

10. General interrupt controller (GIC)

11. Timers (public and private)

2.IOP

The Zynq All Programmable SoC facilitates a complete embedded PS with built-in support for the most common I/O interfaces with the help of these peripherals. Generally there are two interfaces of each peripheral but GPIO is an exception as

it has four interfaces. Here the Static Memory Controller (SMC) is mentioned as a PS memory interface but it can also be with the IOPs as it shares the same MIO. The USB, Ethernet and SD/SDIO blocks each have their own DMA[1].

Multiplexed input/output (MIO): Multiplexed output of peripheral and static memories. There are 54 package pins that are serviced by the MIO. This is not a fully populated mux (not all connection possible). There are two I/O banks that provides help by supporting large number of I/O standard and Voltages .These are majorly divided as High performance (HP) that are limited to 1.8v and used only for high speed interfaces to memory and other chips and High Range (HR) that are used only for 3.3v and mainly used for wider variety of I/O standard. Peripheral ports to the MIO are also present to use while the processor boots, before the PL is programmed. The MIO is controlled and monitored with the help of software-accessible, system-level registers. The peripheral ports that are mapped to the MIO pins are GUI selected in the tools. This selection of tool creates the suitable software code (later exported to and used in SDK) to properly program the registers controlled by MIO[1].

Extended MIO: This makes possible the use of Select IO interface with PS peripherals. Some ports can only use EMIO. Some peripherals have additional capabilities through EMIO. EMIO provided facilities such as[1]:

1.connection to peripheral in programmable logic

2.Utilization of general I/O pins to accomplish MIO pin usage

3.Enables extra signals for the other peripherals

4.Encourages competition for MIO pin usage because most IOP competition for MIO pins is with the flash configuration memory devices (NOR, SPI, NAND) that should utilizes MIO pins[1]

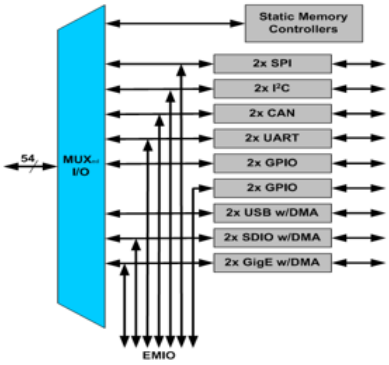


Figure 2.3: I/O Peripheral Interfaces[1]

3.Datapath and Memory

Here the meaning of word datapath is the ARM AMBA architecture which makes possible to move data between end points. This PS datapath is completely inside to the Zynq AP SoC. The datapath can be classified into two important features[1]:

1.AMBA interconnect switches

2.AMBA AXI(interface that is defined by AMBA specification and stands for Advanced eXtensible Interface) ports of the PS-PL interface.

Three interconnects can manage only data internal to the PS:

1. Central interconnect: makes possible other interconnects to communicate

2.IOP master/slave interconnects: makes possible transaction of data to or from specific I/O peripherals

3.OCM interconnect: It is considered as a part of the APU and helps in accessing OCM.

Three other interconnects manage PS-PL data:

1.Memory interconnect: Capable of transaction of data between PL and PS memory resources and PS-PL interface given by four AXI ports.

2.Master interconnect (PS master): Capable of transaction of data between PS master and PL slave endpoints and PS-PL interface given by two AXI ports.

3.Slave interconnect (PS slave): Capable of transaction of data between PS slave and PL master endpoints and PS-PL interface provided by two AXI ports

AXI ports of the PS-PL Interface:

The AMBA ports of the PS-PL interface facilitates the important procedure for the flow of data between the PS and PL.

The ports are grouped into four categories:

1.General purpose master ports

2.General purpose slave ports

3.Accelerator coherency (cache access) port

4.High performance (memory access) ports

This interface also facilitates the main procedure for PL access to main memory (DDR).

4.PS Memory Resources

Memory resources in the PS contains memory interfaces and embedded memory. There are two memory interfaces for the external memory:

1.DDRx dynamic memory controller: it utilizes dedicated package pins to con nect directly to the memory devices. Supports LPDDR2, DDR2,DDR3

2.Flash/ static, memory controller: The physical memory devices for this kind of controller must utilize MIO pins so that the memory is utilized during pro cessor booting. Supports SRAM, QSPI, NAND/NOR FLASH

Two embedded memory components are:

1.On-chip memory: 256KB SRAM

2.ROM for boot code (BootROM)[1]

Worth nothing that the PS can use the PL to enhance the embedded memory resources(BRAM).There is 256K of on-chip RAM and 128k of boot ROM. Their mapping of address can be located at the high or low ends of the 4 GB address space with the help of special control registers. The boot ROM contents are hidden from the user; they cannot be configured or modified and are only utilized while the processors boot[1].

2.4 Detail Study of Programmable Logic

The second major part of the Zynq architecture is the programmable logic. This is based on the Artix-7 and Kintex-7 FPGA fabric. The PL is mainly consists of general purpose FPGA logic fabric, which consist slices and Configurable Logic Blocks (CLBs), and there are also Input/ Output Blocks (IOBs) for interfacing[1].

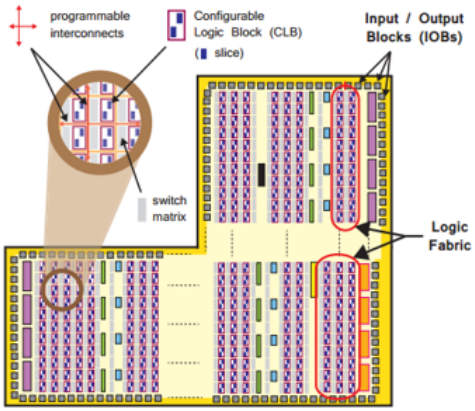


Figure 2.4: Programmable Logic in Zynq[1]

The logic fabric and its constituent element:

1.Configurable Logic Block (CLB): CLBs are very short, continous groupings of logic elements that are arranged in a two-dimensional array on the PL, and combined to many other same resources via programmable interconnects.Each CLB is arranged after another switch matrix and possess two logic slices, as shown in Figure[1].

2.Slice: A sub-unit inside the CLB, which possess resources for implementing combinatorial and sequential logic circuits. As indicated in Figure, Zynq slices are combination of 4 Lookup Tables, 8 Flip-Flops, and other logic[1].

3.Lookup Table (LUT): A lenient resource which makes possible to implement (i) a logic function of up to six inputs; (ii) a small Read Only Memory (ROM); (iii) a small Random Access Memory (RAM); or (iv) a shift register. LUTs can be integrated together to create higher logic functions, memories, or shift registers, as needed[1].

4.Flip-flop (FF): A sequential circuit element utilizing a 1-bit register along with reset functionality. One of the FFs can optionally be utilized to implement a latch.

5.Switch Matrix: A switch matrix is arranged after each CLB, and gives a lenient routing facility for making connections (i) between elements inside a CLB; and (ii) from one CLB to other resources on the PL[1].

Composition of a configurable logic block (CLB):

1.Carry logic: Arithmetic circuits needs internal signals to be propagated within the adjacent slices, and this is gained with the help of carry logic. The carry logic consists of a chain of routes and multiplexers to connect slices in a vertical column.

2.Input / Output Blocks (IOBs): IOBs are resources that facilitates the interfacing within the PL logic resources, and the physical device pads used to combine to external circuitry. Each IOB can manage a 1-bit input or output signal. IOBs are generally present around the perimeter of the device[1].

The Xilinx tools will automatically infer the needed LUTs, FFs, IOBs etc. from the design, and map them accordingly.

Special Resources:

DSP48E1s and Block RAMs are extra addition to the normal fabric, there are two special purpose components:

1.Block RAMs for dense memory requirements; and

2.DSP48E1 slices for high-speed arithmetic.

Both of these resources are combined into the logic array in a column arrangement, embedded into the fabric logic and generally they are close to each other (the reason being that intensive computation and storage of data in memory are often closely associated operations). The Block RAMs in the Zynq-7000 are same as the Xilinx 7 series FPGAs, and they can utilize Random Access Memory (RAM), Read Only Memory (ROM), and First In First Out (FIFO) buffers, that also supporting Error Correction Coding (ECC). Every Block RAM can save up to 36Kb of data, and may be programmed either as one 36Kb RAM, or two individual 18Kb RAMs. The default word size is 18 bits, and in this programming each RAM consists of 2048

memory elements. The shape of the RAM can also be modified such that it possess more, shorter elements (for example 4096 elements x 9 bits, or 8192 x 4 bits), or alternatively, very less, larger elements (e.g. 1024 elements x 36 bits, 512 x 72 bits). By integrating two or more Block RAMs together larger capacity memories can be created. Utilizing a Block RAM means that a great number of data can be saved in a shorter area on the device, inside a devoted and optimised memory element; this can be replaced by the Distributed RAM that is manufactured from the LUTs inside the logic fabric. A suitable number of LUTs (spanned over a larger area) are needed to create a memory of comparable size to a Block RAM, and the resulting implementation goes through restricted timing performance due to the enhanced logic and routing delays. On the other side, it is also beneficial to apply small memories utilizing distributed RAM, both for resource efficiency, and as their arrangement is more lenient (distributed memories can be located close to the components that interact with them, which can result in fast timing performance too). Block RAMs can generally be clocked at the highest clock frequency provided by the device[1].

The logic fabric and its constituent elements:

To implement arithmetic operators of various length The LUTs in the logic fabric can be utilized, but these are most suitable for arithmetic operators along with the short word lengths (arithmetic circuits for long word lengths can have a large footprint in slice logic, with placement and routing factors resulting in sub-optimal clock frequencies). DSP48E1s are best slices for applying high-speed arithmetic on signals with moderate to long arithmetic word lengths. These are devoted silicon resources, and majorly consists of a pre-adder/subtractor, multiplier, and post-adder/subtractor with logic unit. The DSP48E1 make possible the utilization of multiplexing circuitry to allow lenient utilization of registers, and supports dynamic variation in the computation (i.e. the function can be changed on a cycle by-cycle basis as required). Many calculations are feasible, including one, two or many of these arithmetic operators, and these are chosen with the help of an OPMODE input that programs the internal multiplexers (not fully shown in the diagram) and

identifies the arithmetic functionality applied. It also makes possible processing of SIMD, applies 2 or 4 shorter addition/subtraction/accumulation operations of 24 or 12 bits, respectively. The post-adder consist extra functionalities like a logic unit. While utilized in logic mode, it can handle logical functions rather arithmetic, and provides the support for many basic Boolean operations: bit-wise NOT, AND, OR, NAND, NOR, XOR, and XNOR. It is also observed that the pattern detector (not shown in the diagram) that adds the ability to identify overflow, implements rounding on the basis of the selection of schemes, and also performs many related functions[1].

Arithmetic capabilities of DSP48E1 slice:

Together with the benefit of high frequency operation (just like Block RAMs, DSP48E1s can be clocked at the maximum clock frequency of the device) and less power consumption, these DSP48E1 slices fascinates the designer to implement calculatedly demanding arithmetic circuits. Because of these properties, DSP48E1s are good to a various applications in signal processing and so forth. One of their important utilization is to apply symmetric type Finite Impulse Response filters that are generally utilized in DSP and digital communications. The pre-adder make possible that each DSP48E1 can apply two filter taps, and whole filters can be created by combining DSP48E1s together, without the need to utilise any logic from the normal fabric. This facilitates a high performance, highly efficient application for one of the basically useful calculations in DSP. While developing with Zynq, it makes sense to determine deterministic, calculatedly parallel features and apply them in the PL portion of the device, importantly targeting DSP and Block RAM resources wherever possible. In this manner, the PL can be utilized to accelerate algorithms available in the PS. There are large number of suitable examples in which the presence of PL directly proportional to the processor, and the chance to provide specific system functions to the PL, can provide suitable advantages to all the system application[1].

2.5 The AXI Standard

AXI stands for Advanced eXtensible Interface, and the current version is AXI4, which is part of the ARM AMBA 3.0 open standard. Lots of devices and IP blocks developed from the third party developers and manufacturers are dependent on this standard. The AMBA standard was basically produced by ARM for utilization in microcontrollers, with the first version being released in 1996. After then, the standard has been updated and extended, and it is now explained by ARM as the de facto standard for on-chip communication. The emphasis is right now on System-on-Chip, involving SoCs dependent on FPGAs or, in the situation of Zynq, a device that involves FPGA fabric. Help for AXI was first described into the Xilinx tool flow in release 12.3 of the ISE Design Suite, and valuable help is currently present in the Vivado Design Suite. AXI buses can be utilized leniently, and in the original sense are utilized to combine the processor(s) and other IP blocks in an embedded products. There are three kind of AXI4, each type signifies a variant bus are described below. The selection of AXI bus protocol for a specific connection based on the required characteristics of that connection[1].

AXI4: This is used for memory-mapped links, and gives the best performance: an address is transferred followed by a data burst transfer of up to 256 data words (or data beats)[1].

AXI4-Lite: It is a simplified link that supports only one data transaction per connection (no bursts). AXI4-Lite is also memory-mapped: here an address and single data word are transferred.

AXI4-Stream: This is used for high-speed streaming data, provides support for burst transaction with no limit in size. Here there is no mechanism of address; this bus type is suitable for directing flow of data between source and destination (non memory mapped)[1].

If user is using a memory mapped protocol, an address is described along with the transaction decided by the master (read or write), that based in to an address

in the system memory space. When we talk about AXI4-Lite, that supports a single data transfer per transaction, data is then written to, or read from, the described address; when we choose AXI4 bursts, the address described is for the first data word to be transferred, and the slave must then compute the addresses for the data words that follow[1]. AXI Interconnects and Interfaces The first interface between the PS and PL is with the help of a set of nine AXI interfaces, each of this nine AXI interfaces are composed of large number of channels. It results in the devoted connections between the PL, and interconnects inside the PS. It is advantageous to shortly describe these two significant terms[1]:

Interconnect: It is an interestingly a switch that handles and directs burst between connected AXI interfaces. There are many interconnects inside the PS, a few that are directly interfaced to the PL and some that are for internal utilization only. The connections between these interconnects are also produced utilizing AXI interfaces[1].

Interface: It is a point-to-point connection for moving data, addresses, and handshaking signals between master and slave clients inside the system[1].

By the figure in which all of the interfaces are specially connected to AXI interconnects present inside the PS, with the exception of the ACP interface, that is combined directly to the Snoop Control Unit within the APU. Inside the processing system, AXI interfaces are utilized along with both the ARM APU (making connections between the processing cores and SCU, cache memory and OCM), and more originally to combine the different interconnects inside the PS. These connections are extra to those at the PS-PL outer area. Specifically, the three interconnects described in the diagram (the Memory, Master and Slave Interconnects) are internally combined to the Central Interconnect, that is not described here. Table gives a brief of the interfaces described by the arrows in diagram. A short definition of every interface is provided, and the master and slave are shown (in accordance with convention, the master is in control of the bus, and initiates transactions, while the slave responds)[1].

Interfaces between PS and PL:

More explanation of the roles of these various kind of PS-PL AXI interfaces:

General Purpose AXI: It is generally a 32-bit data bus, that is best suited for less and medium rate data transfer between the PL and PS. The interface is straight forward and does not involve any buffering. There are four general purpose interfaces in totality: the PS is the master of two, and the PL is the master of the other two.

Accelerator Coherency Port: It is an single asynchronous connection between the PL and the SCU inside the APU, with a bus width of 64 bits. This port is utilized to gain coherency between the APU caches and elements inside the PL. The PL is the master[1].

High Performance Ports: It is the four high performance AXI interfaces including FIFO buffers to achieve bursty read and write behaviour, and provides support for high rate data transfer between the PL and memory elements in the PS. The width of the data is either 32 or 64 bits, and the PL is the master of all four interfaces[1].

Each and every bus is manufactured by the group of signals, and data transfer on these buses occurs according to the defined bus standard, AXI4[1].

2.6 Zynq-7000 and ARM Trust Zone Technology

Best property of Zynq devices that can avoid vulnerabilities is the Zynq-specific implementation of ARM Trust Zone technology. The Trust Zone architecture makes possible the trusted calculation inside the embedded systems by manufacturing a hardware architecture that can enable the spreading of the security framework throughout the manufacturing of the system. This is achieved by running particular subsystems in either a normal world or a secure world, instead of saving the entirety of the systems assets in a single, devoted hardware resource. From operating in this way, and integrating with software which is makes possible full utilization of the

offered benefits, Trust Zone provides a security solution that can operate from one end of a system to the other. For Zynq devices, a common world is described as a subset of hardware includes memory and L2 cache regions, and particular AXI devices. Non-trusted software can run in a common world, but its use and awareness of extra hardware will be very limited, as it may be devoted to the Trust Zone architecture in the secure world. Software applications that are divided as trusted will run in the secure world that is an isolated, trusted environment separated from the main OS to avoid any malicious use to the embedded system[1]

Chapter 3

PCI Express System Architecture

3.1 Introduction

PCIE stands for Peripheral Component Interconnect Express[2].PCIE remains for Peripheral Component Interconnect Express[2].PCI Express is the third generation of I/O bus used to interconnect peripheral devices in applications. PCI Express is widely inclusive I/O peripheral interconnect bus that has applications in the mobile, desktop, workstation, server, installed registering and correspondence platforms.Data rate of 2.5 Gbps is implemented by the Generation1 of PCIe, Data rate of 5 Gbps is implemented by the Generation2 of PCIe and Data rate of 8 Gbps is implemented by the Generation3 of PCIe[2].A PCI Express interconnect that interfaces two devices together is alluded to as a Link.It is the physical association between two peripherals. It is a point to point Link comprises of either x1, x2, x4, x8, x12, x16 or, on the other hand x32 signals pairs in each direction.These Signals are alluded to as Lanes. A x1 Interface comprises of 1 Lane or 1differential signals combine toward every path for a sum of 4 signals[2]. A x32 Link comprises of 32 Lanes or 32 signals sets for every heading for an aggregate of 128 signals. PCI Express executes a double simplex Link which suggests that information is transmitted and gotten at the same time on transmit and get Lane.The total transfer speed expect similar

traffic in both directions.While initializing hardware, the Link is initialized for Link width and frequency of operation automatically by the devices on another side of the Link[2].No OS or firmware is included in Link level initialization. Each and every byte of data transmitted is first converted into a 10-bit code (via an 8b/10b encoder in the transmitter device).Due to which 25 percent additional overhead to send a byte of data[2].

PCIE Features:

1. PCIe is a serial bus therefor it requires lesser pins than PCI which is a parallel bus.

2.It is a scalable as it gives support for the multiple lane. This feature is significant for the bandwidth intensive applications.

3. It is a packet based transaction protocol. It allows to raise the transmission frequency thereby giving better throughput.

4. It has improved data integrity and error handling. It is reliable, available and serviceable[2].

3.2 PCIE Terminologies

1.Root Complex: It is the device that connects the CPU and memory subsystem to the PCI Express fabric. It may give support for one or more PCI Express ports.The root complex produces transaction requests for the CPU.It is eligible to initiate configuration transaction requests for the CPU.It produces both memory and IO requests. Root complex sends packets out of its ports and take packets on its ports that it forwards to memory[2].Root complex uses major resources such as: hot plug controller, interrupt controller, power management controller, , error detection and reporting logic.The root complex initializes with a bus number, device number and function number that are required to make a requester ID or completer ID.The root complex bus, device and function numbers starts at all 0s[2].

2.EndPoint: This are peripheral devices such as Ethernet, USB or graphics devices.

It can act as requester or completer. There are Two types of endpoints, PCI Express endpoints and legacy endpoints.(In my project I am working with PCI Express Endpoints so explaining this only)[2].PCI Express Endpoints does not support IO transactions and supports MSI style interrupt generation. It also provides support for 64-bit memory addressing capability in prefetchable memory address space, even though its non-prefetchable memory address space is only allowed to map the below 4G Byte boundary[2]. Both types of endpoints uses Type 0 PCI configuration headers.[2]. Each endpoint is initialized with a device ID (requester ID or completer ID) that posses a bus number, device number, and function number.Endpoints are always device 0 on a bus.

3.Requester: It is a device that initiates a transaction in the PCI Express fabric. Root complex and endpoints are requester type devices[2].

4.Completer: It is a device that is targeted by a requester. Root complex and endpoints are completer type devices[2].

5.Port: It is the interface between a PCI Express component and the Link. It consists of differential transmitters and receivers[2].

a.) Upstream Port (endpoint port port): It points to the direction of the root complex.

b.) Downstream Port (root complex port): It points to the direction of the Endpoint device.

c.) Ingress Port: It is a port that takes a packet from the transmitter.

d.) Egress Port: It sends a packet to the receiver[2].

6.Bridge: This is used to connect PCI or PCIX device to a PCIE root complex.

7.Switches: These are used to connect multiple PCIE devices to the root complex.It is generally used when there are not enough slots present in the board[**?**]. Diagram to show PCIE topology.

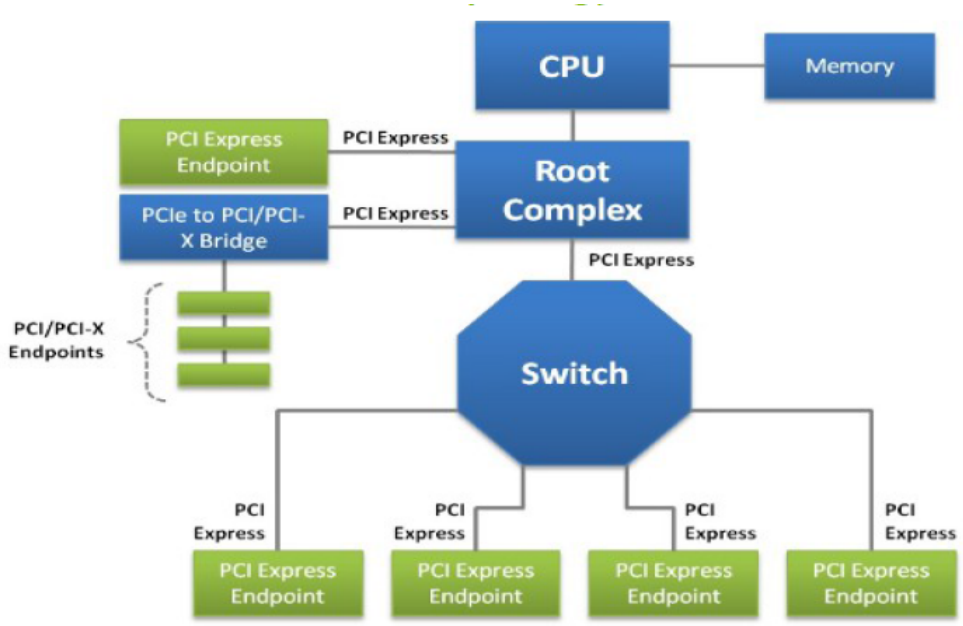


Figure 3.1: PCIE Topology[4]

3.3 Introduction to PCI Express Transactions

PCI Express uses packets for accomplishing data transfers between devices.An endpoint can communicate with a root complex.A root complex can communicate with an endpoint. [2]. An endpoint can communicate with another endpoint[2].Communication includes the transmission and reception of packets called Transaction Layer packets (TLPs)[2].PCI Express transactions can be grouped into four categories:1) memory, 2) IO, 3) configuration, and 4) message transactions.These transactions can be classified as non posted transactions and posted transactions.EP does not start configuration Read/Write requests. Completion without data will be produced by the completer if encountered with any error.For Non-posted transactions, a requester sends a TLP request packet to a completer. After that, the completer gives a TLP completion packet back to the requester[2]. Non-posted transactions can be handled as split transactions.Non-posted read transactions posses data in the completion TLP. Non-Posted write transactions posses data in the write request TLP.For

Posted transactions, a requester sends a TLP request packet to a completer[2]. The completer however does NOT gives back a completion TLP to the requester.As PCI Express Endpoint does not support IO transactions, So not discussed here. Memory Read, Configuration type 0 Read and Configuration type 0 Write transactions are non posted transactions while Memory Write and Message transactions are posted transactions[2].

3.4 PCI Express Device Layers

The PCI Express specification explains a layered architecture for device design.The layers consist of a Transaction Layer, a Data Link Layer and a Physical layer[2]. The layers can be again classified vertically in two as transmit side which processes outbound packets and a receive side which processes inbound Packets.Packets are created at a transmitting device and packet received and decoded at a receiving device.There are three types of packets here, for one of the three device layers. TLP for the Transaction Layer is the Transaction Layer Packet (TLP), TLP for the Data Link Layer is the Data Link Layer Packet (DLLP) and TLP for the Physical Layer is the Physical Layer Packet (PLP)[2].

3.5 Function of Each PCI Express Device Layer

This block diagram is used to explain the operation of each layer and gives the explanation of the function of each layer. The layers posses Device Core/Software Layer, Transaction Layer, Data Link Layer and Physical Layer[2].

Device Core / Software Layer:

The Device Core posses, for example, the root complex core logic or an endpoint core logic.

Transmit Side

The Device Core logic in combination with local software gives the important in-

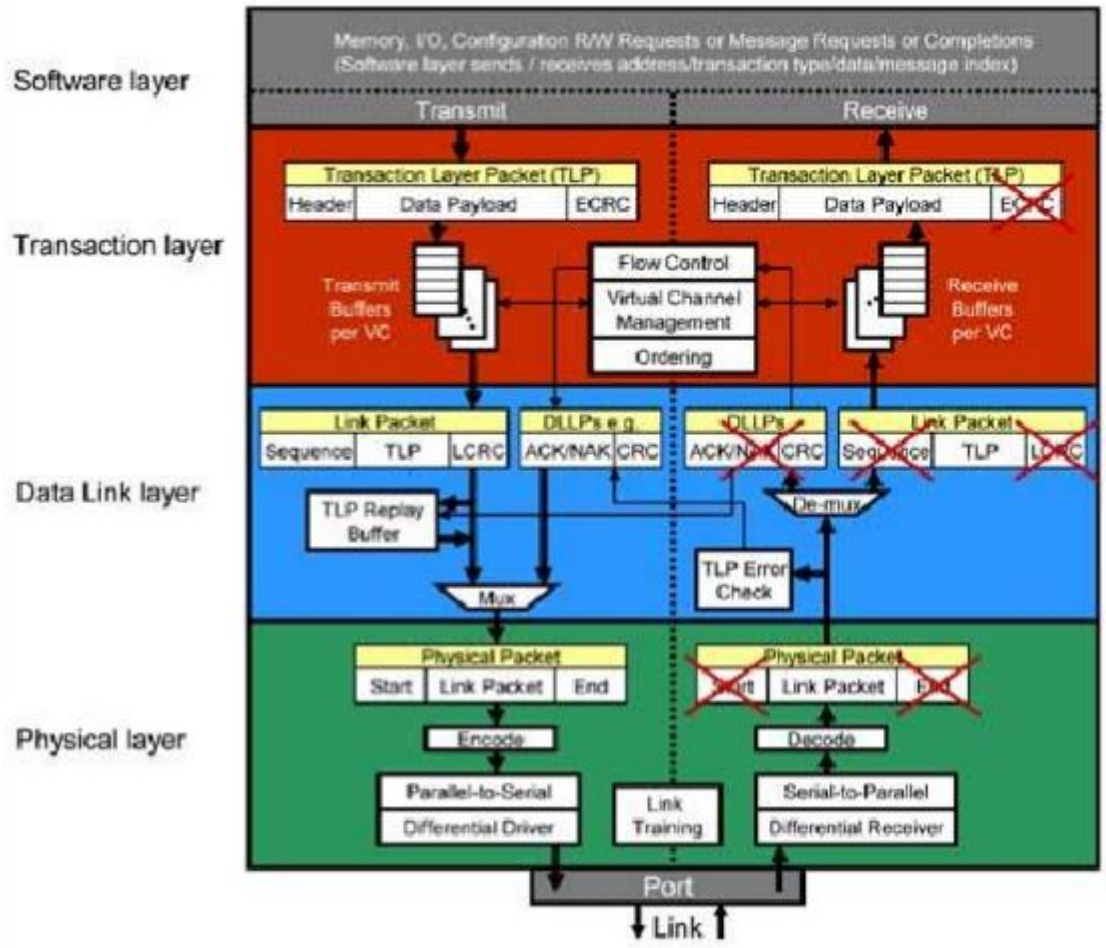


Figure 3.2: Block Diagram for PCI Express Device Layer [4]

formation needed by the PCI Express device to produce TLPs.This information is transmitted by the Transmit interface to the Transaction Layer of the device.

Receive Side

The Device Core logic also receives information coming from the Transaction Layer by the Receive interface.

Transaction Layer:

The transaction Layer generates outbound packets and receives inbound packets.The transaction layer posses virtual channel buffers (VC Buffers) to save outbound traffic which await transmission and also to save inbound traffic received from the Link.

Flow control protocol for this virtual channel buffers make sure that a nearby transmitter does not sends too many TLPs and cause the receiver virtual channel buffers to over flow.The Transaction Layer provides support for 4 address spaces: memory address, IO address, configuration address and message space. Message packets posses a message.

Transmit Side

The Transaction Layer takes data from the Device Core and produces outbound packets and completion TLPs which it saves in virtual channel buffers.This layer assembles Transaction Layer Packets (TLPs).Here, TLPs central components are Header, Data Payload and an optional ECRC (specification also uses the term Digest) field as shown in Figure.

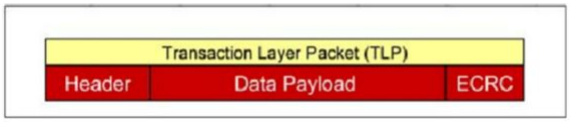


Figure 3.3: Structure of TLP at the Transaction Layer [4]

The Header is 3 double words or 4 double words in size and contains information such as; Address,requester ID/completer ID, TLP type, transfer size, tag, byte enables, completion codes, traffic class, and attributes The address is a 32-bit memory address or an extended 64-bit address for memory requests.A bit in the Header (TD = TLP Digest) shows that whether this packet posses an ECRC field also known as Digest. This field is 32-bits wide and contains an End-to-End CRC (ECRC).The ECRC field is produced by the Transaction Layer at time of creation of the outbound packets. The receiver device checks for an ECRC error that can be caused due to the packet goes through the fabric.

Receive Side

It saves inbound traffic in receiver virtual channel buffers.The receiver checks CRC errors by seeing ECRC field in the TLP.If no error is found, the ECRC field is

stripped and the remaining data in the TLP header as well as the data payload is transmitted to the Device Core.

- Flow Control

The Transaction Layer makes sure that it does not sends a TLP onto the Link to a nearby receiver device untill or unless the receiver device has virtual channel buffer space to accept TLPs . The protocol usded for completion of this mechanism is referred to as the "flow control" protocol. If the transmitter device does not observe this protocol, a sent TLP will cause the receiver virtual channel buffer to overflow. Flow control is automatically handled at the hardware level and is transparent to software. Software is only involved to add more buffers beyond the default set of virtual channel buffers.The default buffers are initialized automatically after Link training,that allows TLPs to flow through the fabric immediately after Link training. Configuration transactions implements the default virtual channel buffers.A receiver device sends DLLPs called Flow Control Packets (FCx DLLPs) to the transmitter device on a periodic basis.This posses flow control credit information that updates the transmitter showing how much buffer space is available in the receiver virtual channel buffer.The transmitter keeps track of this information and will only send TLPs out if it knows that the remote receiver has buffer space to accept the transmitted TLP.

Data Link Layer:

The main function of the Data Link Layer is to make sure data integrity while transmitting and receiving packets on each Link.If a transmitter device transmits a TLP to a nearby receiver device at the other end of a Link and a CRC error is detected, the transmitter device is noticed with a NAKDLLP[2]. The transmitter device automatically replays the TLP. This time hopefully no error occurs.[2].

Transmit Side

The Transaction Layer notices the flow control mechanism before forwarding outbound TLPs to the Data Link Layer.If required credits exist, a TLP in the virtual

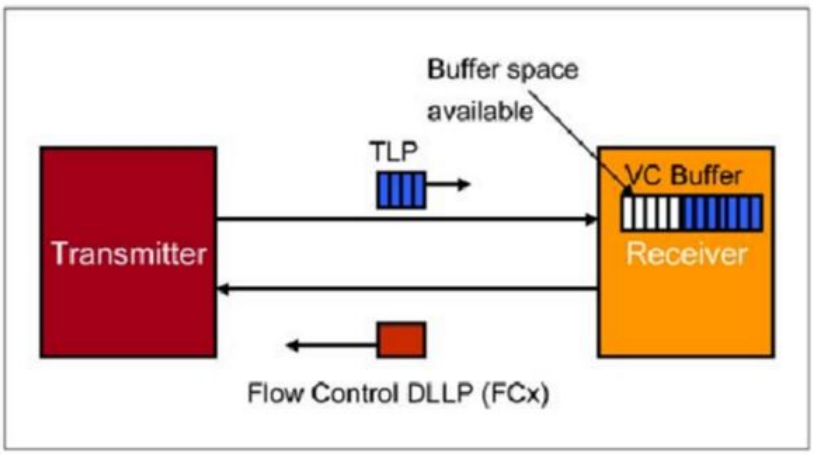


Figure 3.4: Flow Control Process [4]

channel buffer is moved from the Transaction Layer to the Data Link Layer for transmission[2].

Receive Side

The receive side of the Data Link Layer checks for LCRC error on inbound packets[2]. If there is no error, the device schedules an ACK DLLP for transmission back to the nearby transmitter device. The receiver strips the TLP of the LCRC field and sequence ID[2]. If there is a CRC error, it schedules a NAK to return back to the nearby transmitter. The TLP is discarded[2]. The receive side of the Data Link Layer also takes ACKs and NAKs from a remote device. If an ACK is received the receive side of the Data Link layer informs the transmit side to clear an associated TLP from the replay buffer. If a NAK is taken, the receive side causes the replay buffer of the transmit side to replay associated TLPs[2].

The 16-bitCRC is calculated using all 4 bytes of the DLLP. Received DLLPs that fail the CRC check are eliminated. ACK and NAK DLLPs posses a sequence ID field (shown as Misc. field in Figure 2-26) implemented by the device to associate an inbound ACK/NAK DLLP with a saved copy of a TLP in the replay buffer[2].

Physical Layer:

Both TLP and DLLP type packets are deleivered from the Data Link Layer to the

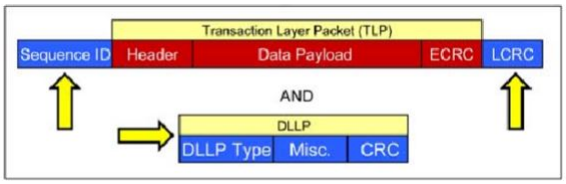


Figure 3.5: TLP and DLLP structure at Data Link Layer[4]

Physical Layer for transmission over the Link. Also, packets are accepted by the Physical Layer from the Link and transmitted to the Data Link Layer[2]. The Physical Layer is classified in two portions, the Logical Physical Layer and the Electrical Physical Layer. The Logical Physical Layer posses digital logic associated with processing packets before transmission on the Link, or processing packets inbound from the Link before sending to the Data Link Layer[2]. The Electrical Physical Layer is the analog interface of the Physical Layer which connects to the Link. It posses of differential receivers for each Lane.

Transmit Side

TLPs and DLLPs are clocked into a buffer in the Logical Physical Layer. The Physical Layer adds the Start and End character here. The symbol is a framing code byte that a receiver device implements to find the start and end of a packet[2].

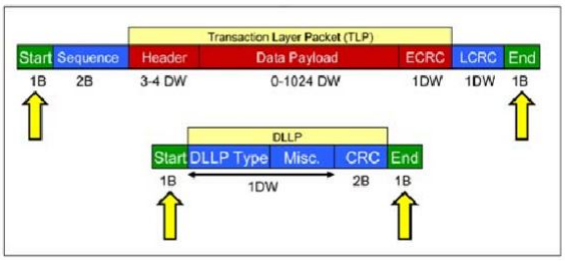


Figure 3.6: TLP and DLLP structure at Physical Layer[4]

Packets are byte striped on the available Lanes on the Link. The resultant bytes are encoded into a 10b code by the 8b/10b encoding logic. The major motive of encoding 8b characters to 10b symbols is to generate sufficient 1-to-0 and 0-to-

1transition density in the bit stream. The parallel-to-serial converter produces a serial bit stream of the packet on each Lane and transmits it differentially.

Receive Side

The receive Electrical Physical Layer clocks in a packet arriving differentially on all Lanes. The serial bit stream of the packet is converted into a 10b parallel stream by utilizing the serial-to-parallel converter. The receiver logic also involves an elastic buffer which stores clock frequency variation between a transmit clock with which the packet bit stream is clocked in a receiver and the receiver clock. The 10b symbol stream is decoded again into 8b symbol with the 8b/10b decoder. The 8b characters are de-scrambled. The Byte unstriping logic, regenerates the original packet stream sent by the remote device[2].

3.6 Transaction Layer Packet Routing Basics

Generally TLPs are utilized to Access Four Address Spaces. As transactions helds between PCI Express requesters and completers, four separate address spaces are utilized: Memory, IO, Configuration, and Message[2].

System Routing Strategy Is Programmed

Before transactions can be initialized by a requester, accepted by the completer, and forwarded by any devices in the path between the two, all devices must be programmed to enforce the system transaction routing scheme. Routing depends on traffic type, system memory and IO address assignments, etc. Each PCI express device is discovered, memory and IO address resources are given to them, and switch/bridge devices are configured to forward transactions for them[2]. Once routing is configured, bus mastering and target address decoding are initialized. Then. devices are prepared to create, accept, forward, or reject transactions as required.

Three Methods of TLP Routing

All the TLPs, targeting any of the four address spaces, are routed by utilizing one of the three possible schemes: Address Routing, ID Routing, and Implicit Routing.

Memory Read and Write TLP utilizes Address Routing method, Configuration Read and Write Type 0 TLP uses ID Routing method, Message and Message with data TLP uses both Address and ID Routing methods and Completion and Completion with data TLP uses ID routing method[2].

Header Fields Define Packet Format and Routing

As each TLP posses 3 or 4 double word (12 or 16 byte) header. It incldes two fields, Type and Format (Fmt), that define the format of the remainder of the header and the routing method to be utilized on the entire TLP as it passed between devices in the PCI Express topology.As TLPs enters at an ingress port, they are checked for errors at both the physical and data link layers of the receiver[2].Assuming there is no error, TLP routing is performed; basic steps include: The TLP header Type and Format fields in the first DWord are observed to find the size and format of the remainder of the packet.Then based on the routing method for the packet, the device will find if it is the recipient; if so, it will consume the TLP. If it is not the recipient, and it is a multi-port device, it will send the TLP to the suitable egress port[2].If it is neither the intended recipient nor a device in the path to it, it discards the packet as an Unsupported Request (UR)[2].

Applying Routing Mechanisms

Once configuration of the system routing strategy is finished and transactions are initialized, PCI Express devices decode inbound packet headers and utilizes corresponding fields in configuration space Base Address Registers to apply one of the routing method to the packet[2].

1.Address Routing

Address routing is utilized to transmit data to or from memory, memory mapped IO, or IO locations.Memory transaction contains either32 bit addresses by utilizing the 3DW TLP header format, or 64 bit addresses by utilizing the 4DW TLP header format[2]. The size of the system memory map depends on the range of addresses that devices are capable of generating [2].

TLPs with 3DW, 32-Bit Address

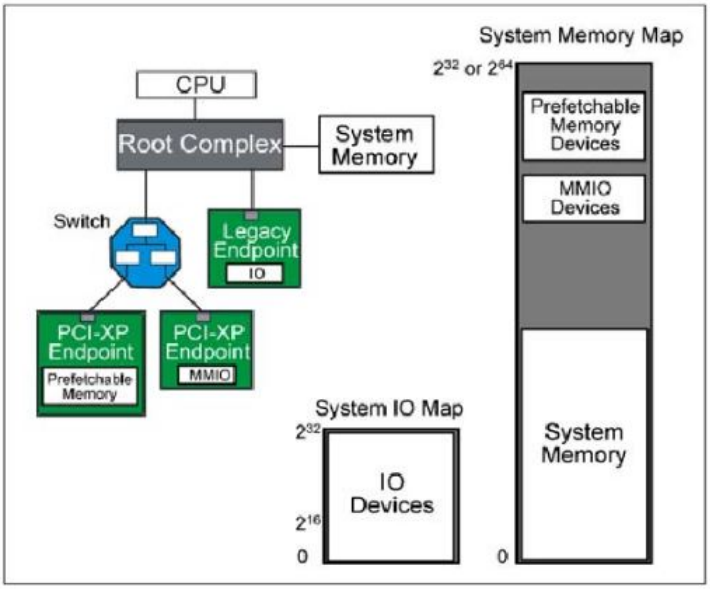


Figure 3.7: Generic System Memory And IO Address Maps[4]

For a 32-bit memory requests, only 32 bits of address are possessed in the header. TLPs With 4DW, 64-Bit Address.For 64-bit memory requests, 64 bits of address are possessed in the header.[2].

An Endpoint Checks an Address-Routed TLP

If the Type field in a received TLP shows address routing is to be utilized, then an endpoint device checks the address in the packet header against each of its utilized BARs in its Type 0 configuration space header[2].

2.ID Routing

ID routing is depends on the logical position (Bus Number, Device Number, Function Number) of a device function within the PCI bus topology.PCI Express provides support for a maximum of 256 busses/links in a system, a maximum of 32 devices per bus/link and maximum of 8 internal functions per device[2].

3.7 Plug-And-Play Configuration of Routing Options

PCI Express provides support for 256 byte PCI configuration space common to all compatible devices, involving the Type 0 and Type 1 PCI configuration space header formats utilized by non-bridge and switch/ bridge devices.[2].

Routing Registers Are Located in Configuration Header

As with PCI, registers for transaction routing are present in the first 64 bytes (16DW) of configuration space[2]. Base Address Registers (BARs) found in Type 0 and Type 1 headers. The first of the configuration space registers involving routing are the Base Address Registers (BARs) are used by all devices that needs system memory, IO, or memory mapped IO (MMIO) addresses assigned to them.A Type 0 configuration space header contain 6 BARs. After finding device resource requirements, system software configure each BAR with start address for a range of addresses the device acts as a completer.Set up of BARs includes[2]:

1.The device designer utilizes a BAR to hard-code a request for an allocation of one block of memory addresses in the system memory map.[2]

2.Hard-coded bits in the BAR involves a sign of the request type and the size of the request.During enumeration, all PCI-compatible devices are found and the BARs are checked by system software to decode the request. Once the system memory and IO maps are done, software configures upper bits in utilized BARs.

3.8 Introduction to the Packet-Based Protocol

All data moves across a PCI Express link in basic chunks known as packets.[2].The two major classes of packets are Transaction Layer Packets (TLPs), Data Link Layer Packets (DLLPs).Different TLPs and DLLPs permits two devices to perform memory, IO, and Configuration Space transactions reliably and utilize messages to start power management events, generate interrupts, report errors, etc[2].

A sample write packet

Lets take the case of data write TLP. Suppose that the CPU wrote the value 0x12345678 to the physical address 0xfdaff040 by utilizing 32-bit addressing. The packet could then posses of four 32-bit words (4 DWs, Double Words) as shown in figure[2]: So the packet sent as 0x40000001, 0x0000000f, 0xfdaff040, 0x12345678.

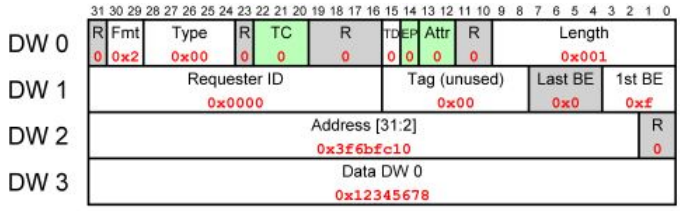


Figure 3.8: Example of Memory Write Request TLP[4]

Explanation of the valid fields:

1.Gray fields are reserved, indicates that the sender has to put zeros there. Some gray fields contain R means that the field is always reserved.[2].

2.Green fields are rarely utilized by endpoint peripherals[2].

3.The values of the particular packet are denoted in red.

4.The Fmt field and Type field shows that this is a Memory Write TLP.

5.The TD bit is zero, shows that there is no extra CRC on the TLP data.[2].

6.The Length field contain the value 0x001, showing that this TLP has one DW (32-bit word) of data. The Requester ID field shows that the transmitter of this packet contain ID zero its the Root Complex[2].

7.The Tag is an unused field here.

8.The 1st BE field (1st Double-Word Byte Enable) permits to select which of the four bytes in the first data DW are valid, and should be written. Set as 0xf in our case, it shows that all four bytes are written to[2].

9.The Last BE field must be zero when Length is unity, as the first DW and the last is the same one.

10.The Address field is the address to which the first data DW is written.[2].

Multiply 0x3f6bfc10 by four, and get 0xfdaff040.And finally, we have one DW of data[2].

A Read Request

Now, the CPU reads from a peripheral.It includes two packets: One TLP from the CPU to the peripheral, asking to perform a read operation, and one TLP going back with the data[2]. Suppose CPU wants a single DW from address 0xfdaff040.It initiates a read operation on the bus it shares with its memory controller, which posses the Root Complex, that in turn produces a TLP to be transmitted over the PCIe bus[2]. So this packet consists of the 3 DWs 0x00000001, 0x00000c0f,

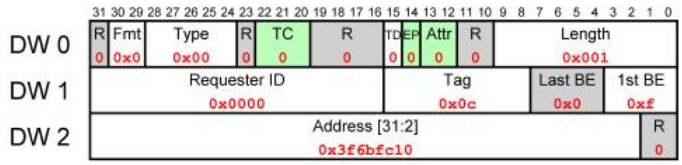


Figure 3.9: Example of Memory Read Request TLP[4]

0xfdaff040. It tells the peripheral to read one full DW at address 0xfdaff040, and to give result to the bus entity whose ID is 0x0000[2]. Its similar to the Write Request example shown above, so just focus on the differences:

1.The Fmt field has changed to show this is a Read Request[2].

2.The Requester ID field shows that the transmitter of this packet has ID zero. It shows the Completer where to transmit its response[2].

3.Tag field has the function of a tracking number: As the Completer responds, it must copy this value to the Completion TLP[2]. This permits the Requester to match Completion answers with its Request.

4.The Length field shows the one DW should be read, and the Address field from which address.

The Completion

When the peripheral get a Read TLP, it must return a Completion TLP. Suppose the peripheral read the chunk of data from its internal resources, and now requires

to send the result back to the Requester[2]. So the TLP consists of 0x4a000001,

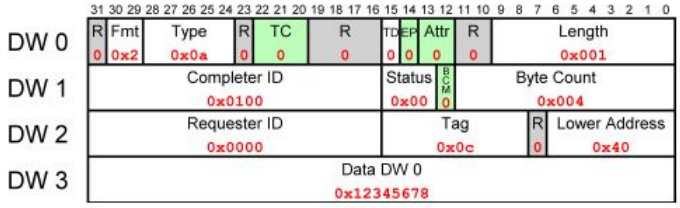


Figure 3.10: Example of Completion TLP[4]

0x01000004, 0x00000c00, 0x12345678. The packets basically indicates that tell bus entity 0x0000 that the answer to its Request to entity 0x0100, which was tagged 0x0c, is 0x12345678[2]. The CPU will look up in its internal records what that request was about, and complete the requires bus cycle.The different fields in packet indicates:

1.The Fmt field, together with the Type field say this is a Completion packet with data[2].

2.The Length field has the value 0x001, showing that this TLP has one DW (32-bit word) of data.

3.Byte Count is the number of valid payload bytes in the packet.

4.The Lower Address field is the 7 LSBs of the address, from which the first byte in this TLP was read. Its 0x40 in this case, from the lower bits of 0xfdaff040.

5.The Completer ID shows the sender of this packet, which is 0x0100[2].

6.The Requester ID shows the receiver of this packet, which is zero ID.

7.The Status field is zero, showing that the Completion was successful.[2].

8.The BCM field is always zero for PCI Express. And finally, we have one DW of data.

3.9 PCI Express Configuration

A device on a bus and posses one or more functions, it contains the PCI Express Configuration space

Topology Is Unknown At Startup

When the system is started, the configuration software has not yet scanned the PCI Express fabric to find the machine topology and how the fabric is populated.The configuration software is only known of the existence of the Host/PCI bridge within the Root Complex and that bus number 0 is directly connected to the downstream [2].The process of scanning the PCI Express fabric to find its topology is called as the enumeration process[2].

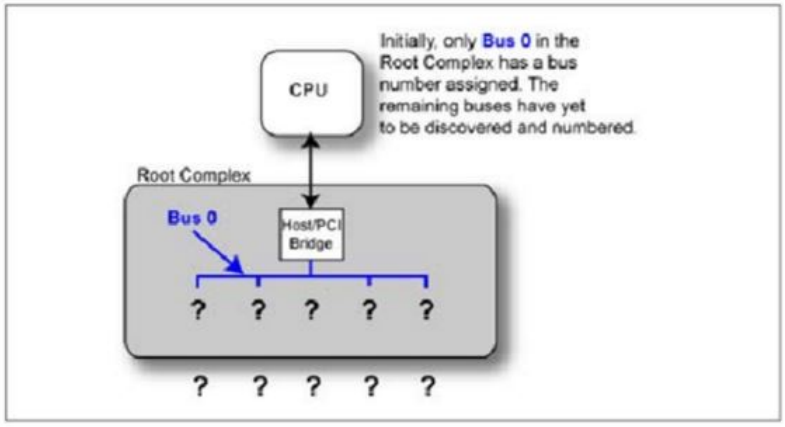


Figure 3.11: Topology View at Startup[4]

Each Function Implements a Set of Configuration Registers For the software running on the processor, the Root Complex starts configuration transactions to read from or write to a function's configuration registers.These registers are used to find the presenceof a function as well as to program it for normal operation[2].PCI Express explains a block of configuration space for each function within which its configuration registers are used.Each function's configuration space is 4KB in size[2].This area can be accessed by utilizing PCI Express Enhanced Configuration mechanism[2]. The first 16 dwords posses the PCI configuration header area, while

the remaining 48 dword area is reserved for function-specific configuration registers and PCI New Capability register sets.The remaining 3840 byte (960 dword) area is for the PCI Express Extended Configuration Space. It is utilized to implement the optional PCI Express Extended Capability registers[2] The Host/PCI bridge's configuration register set does not have to be accessed by utilizing any of the configuration mechanisms. As, it is mapped into a Root Complex design-specific address space that is known to the platform-specific BIOS firmware[2].However, its configuration register layout and usage must adhere to the standard Type 0.

Configuration Transactions Are Originated by the Processor

Only the Root Complex is permitted to originate configuration transactions[2]. The Root Complex behaves as the processor's surrogate to inject transaction requests into the fabric, and to pass completions back to the processor[2]. The configuration software running on the processor is responsible for finding and programming all devices in the system[2].

Type 0 Configuration Request

A configuration read or write has the form of a Type 0 configuration read or write when it reaches on the destination bus. On finding that it is a Type 0 configuration operation:

1. The devices on the bus decode the header's Device Number field to find which of them is the target device[2].

2. The selected device decodes the header's Function Number field to find the selected function within the device.

3. The selected function utilizes the connected Extended Register Number and Register Number fields to choose the target dword in the function's configuration space.

4. Finally, the function utilizes the First Dword Byte Enable field to choose the byte(s) to be read or written within the selected dword[2]. Figure 3-13 and Figure 3-14 shows the Type 0 configuration read and write request header formats.Here, the Type field = 00100, while the state of the Fmt field's msb shows whether it's a read or a write[2].

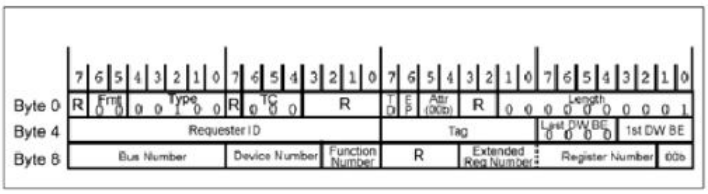


Figure 3.12: Type0 Configuration Read Request Packet Header[4]

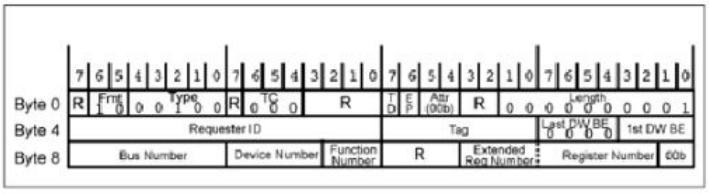


Figure 3.13: Type0 Configuration Write Request Packet Header[4]

Header Type 0: This is the header that is found in PCIE End Point devices. The registers within the Header are utilized to know the device, to control its functionality and to sense its status[2].

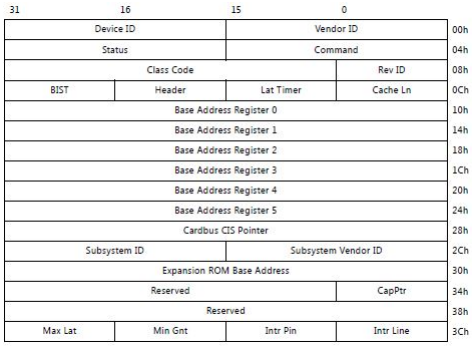


Figure 3.14: Type 0 PCI Configuration Space Header[4]

Registers Used in Header 0

1. Vendor ID Register: This 16-bit register indicates the manufacturer of the function.

2. Device ID Register: This 16-bit value is defined by the function manufacturer and shows the type of function.

3. Revision ID Register: This 8-bit value is defined by the function manufacturer and shows the revision number of the function[2].

4. Class Code Register: It is a 24-bit, read-only register classified in three fields: base Class, Sub Class, and Programming Interface. It shows the basic function of the function (e.g., a mass storage controller), a more specific function sub-class (e.g., IDE mass storage controller), and in some cases, a register-specific programming interface (such as a specific flavor of the IDE register set)[2].

5. Header Type Register:Bits [6:0] of this one byte register identifies the format of dwords 4-through-15 of the function's configuration Header. Bit seven shows the device as a single- (bit 7 = 0) or multifunction (bit 7 = 1) device[2].

6. BIST Register:This register is an Optional.

7. Capabilities Pointer Register: If the Capabilities List bit in the Status register is assign to one, the function uses the Capabilities Pointer register in byte zero of dword 13 in its PCI-compatible configuration space.This shows that the pointer posses the dword-aligned start address of the Capabilities List within the function's lower 48 dwords of PCI-compatible configuration space[2].

8. CardBus CIS Pointer Register:This register is an Optional register.

9.Command Register: Explanation of each bit in the Command register of a non-bridge function (i.e., one with a Type 0 Header format)[2].

To transfer TLPs onto he link, the Bus Master Enable bit that is bit 2 of the PCI Command register at address offset 0x04 in the configuration space must be set. To get memory or IO TLPs the memory or I/O enable bits, bits 0 and 1, must be set in the PCI Command register.

10.Cache Line Size Register: Only for legacy pcie endpoint devices.

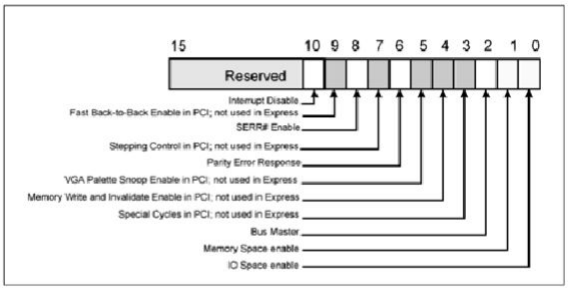


Figure 3.15: Command Register[4]

11.Master Latency Timer Register: This register is an Optional.This register does not apply to PCI Express.

12.Interrupt Line Register: This register is an Optional.

13.Interrupt Pin Register: This register is an optional

14.Base Address Registers: Utilized if a function needs memory and/or IO decoders.On power-up, the system must be automatically programmed so that each function's IO and memory functions uses different address ranges.For this, the system must be able to find how many memory and IO address ranges a function needs and the size of each. The system must then be able to program the function's address decoders for assigning non-conflicting address ranges to them.The Base Address Registers in function's configuration Header space are utilized to implement a function's programmable memory and/or IO decoders.Each register is 32-bits wide (or 64-bits wide).Bit 0 is a read-only bit and shows whether it's a memory or an IO decoder:

a.If bit 0 = 0, memory address decoder.

b.If bit 0 = 1, IO address decoder.

Memory Base Address Register

Decoder Width Field: In a Memory BAR, bits [2:1] shows whether the decoder is 32- or 64-bitswide:

a.If 00b = it's a 32-bit register.

b.If 10b = it's a 64-bit register. Pre fetchable Attribute Bit: Bit three indicates the

block of memory as Pre-fetchable or not[2].

15.Min Gnt/Max Lat Registers: These registers do not apply to PCI Express[2].

Chapter 4

Introduction to PCI Tree

4.1 Introduction

PCItree is a graphical Windows tool to view all the hardware devices of the PCIbus. The devices are viewed in form of a tree . Information about the devices and its vendors is taken from a seperate database. PCItree provides read and write access to the config registers of each device to the user and even to each device's memory provided by the BAR[3]. This tool also helps in figuring out problems with PC, or lets user debug user's custom PCI chip.

Important: There is a requirement of administrator rights to install and to run PCItree[3].

Main window of PCItree

The main window of PCItree indicates the structure of the PCIbus on the left side. The config registers of the selected device (here: VGA controller) are shown on the right lower side. On top of this Config space dump information of some registers is viewed .[3]. There is a field for modifying a dword of a config register. For that choose a config register change the value and click write to write that value back into the register[3].

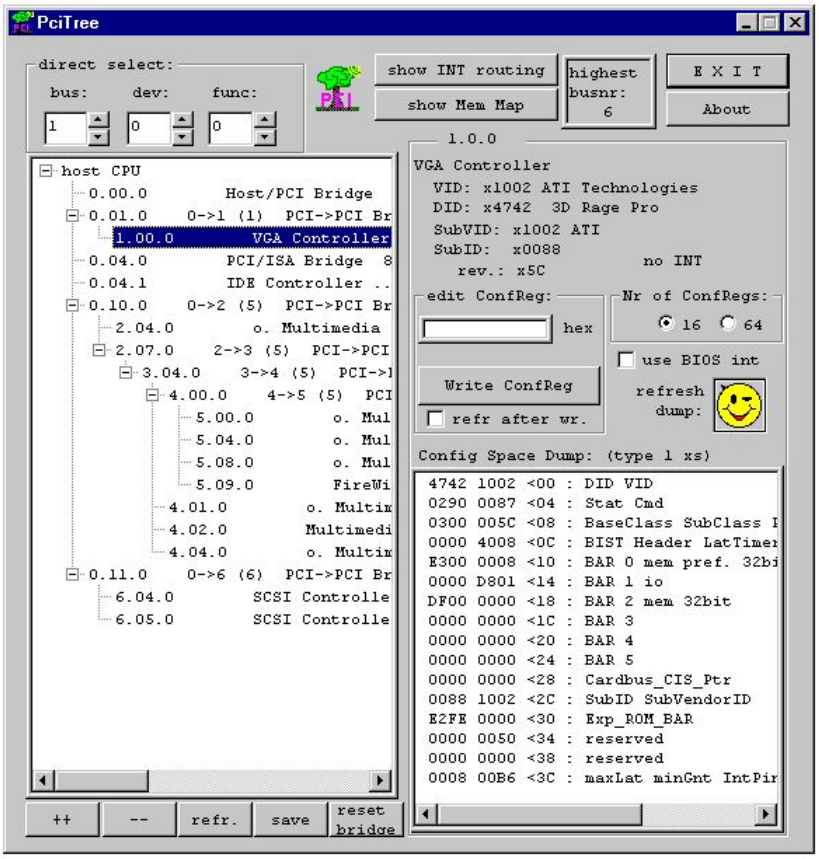


Figure 4.1: Main window of PCItree[4]

Memory window of PCItree

The memory window of PCItree is viewed when a BAR of the config register in the main window is double clicked. The memory window indicates a memory range of 1 kByte[3]. One can change one dword of the memory by choosing one address in left view. The value of that memory location can be modified in the edit box on the right side.

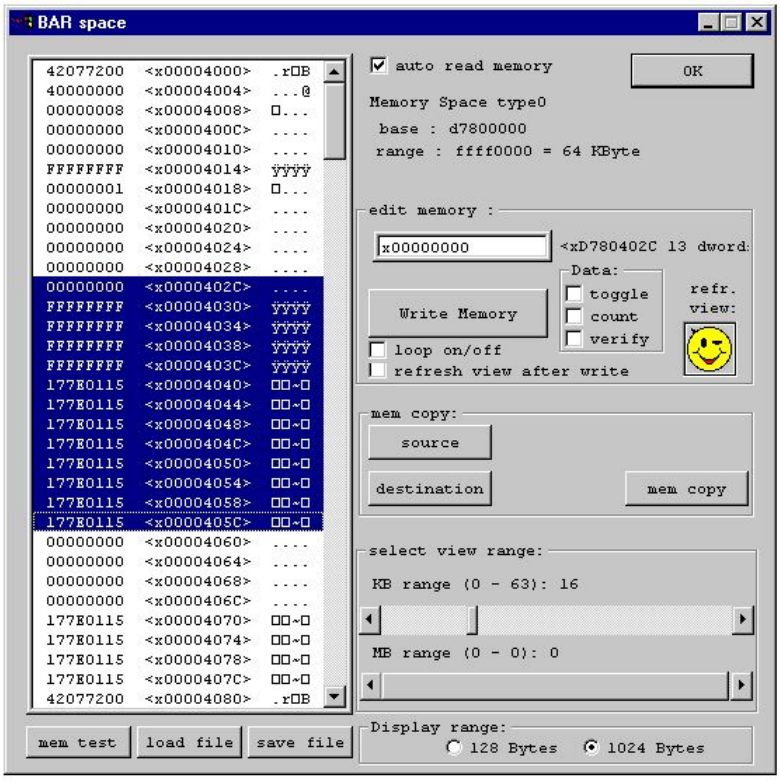


Figure 4.2: Memory window of PCItree[4]

Memory map view of the PCIbus

The memory map of the PCIbus is viewed with a sizeof 8 MByte. Displayed are the PCI to PCI bridges and the BARs which are mapped to this address range[3].

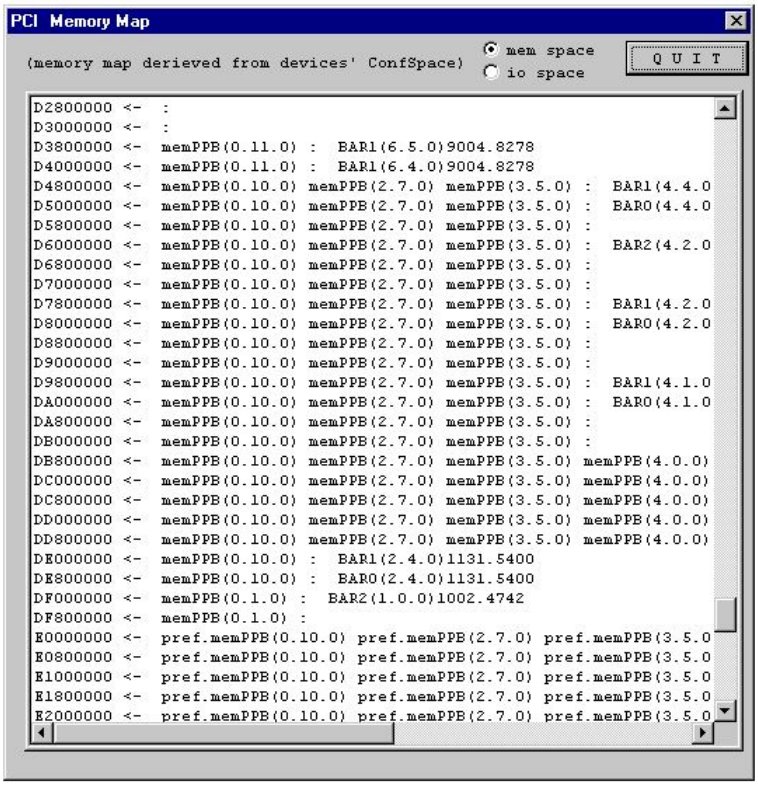


Figure 4.3: Memory map view of PCItree[4]

PCI INT routing of the PCIbus

The first 16 Interrupt lines of the host controller are displayed. The config registers of all devices are found for which INT pin is routed for which line of the controller[3]. The devices are shown with their INT pin (A to D), their b.d.f (bus.device.function number)and their VID and DID[3].

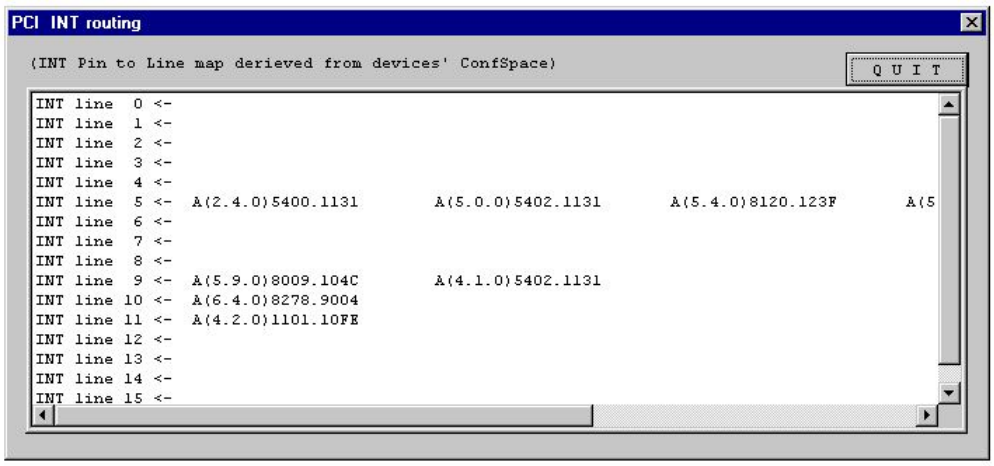


Figure 4.4: PCI INT routing of the PCIbus[4]

4.2 Explanation of various fields in the window

Tree list: pcitree scans the PCIbus of the host PC and shows the found components as a tree. components are found if a read of DID and VID doesn't return 0xffffffff[3]. last PCI bus number: This is the value of the highest PCIbus number.

b.d.f : Each and every PCI component has an integer number for bus, device and function[3].

direct select: If user thinks that not all components in the PCIbus system are shown[3]: any b.d.f is selected directly by the three spin controls in the direct select box. A config read is performed for this b.d.f and the result is shown in the Config Space Dump[3].

++ shows all levels of the tree

– collapses all levels of the tree

refreshtree rescans the whole PCIbus

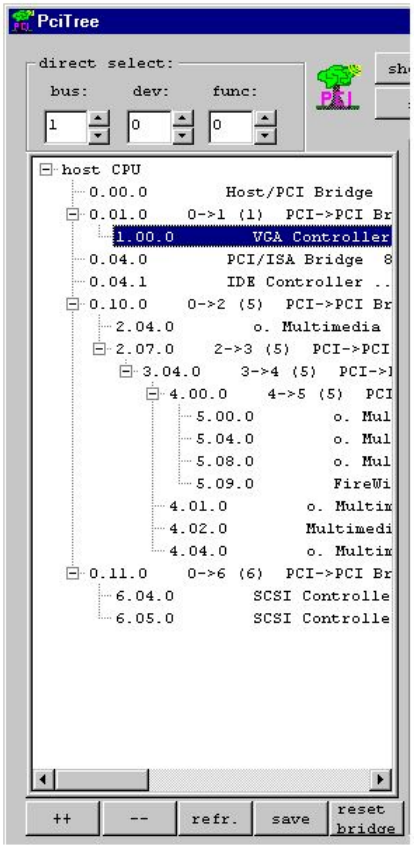


Figure 4.5: Tree list[4]

Config Space Dump:

use BIOS int checkbox: type 1 access utilizes IO ports to write/read the config space (default) type 0 access utilizes BIOS INT 13 calls

Nr of ConfRegs:

16 : config space from adress 0 to 0x3f is read (default)

64 :config space from adress 0 to 0xff is read

refresh dump: rereads the config space of the device and shows it if a component is chosen in the tree the register contents of its configuration space (16 or 64 dwords) are shown in the lower right window[3].

The following information is gained and decoded from the config space:

VendorID: value and decoded

DeviceID: value

Subsys VID: value and decoded

Subsys ID : value

Base/Sub Class code: value and decoded

Interrupt Pin/Line: decoded

revision number: value[3]

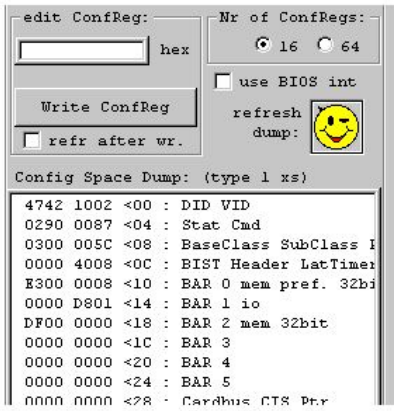


Figure 4.6: Config Space Dump[4]

BAR space: One can double click on a BAR in the Config Space list to see the memory/io space Only 32 bit adress space is supported.

BAR space detection: If one click YES in the message box the following happens:
the BAR in the ConfigSpace is read and saved
the BAR is written with 0xffffffff
the BAR is read. With the read value the size of the memory/io space is find[3]. The BAR is written with the saved BAR value a linear adress of 1Kbyte is mapped to the physical adress (the contents of the BAR).A new Dialog window with a list box is viewed.If one click NO then the explained procedure is not done. Instead a size of 1 MByte is assumed[3].

auto read memory: the content of the display range is not read when the Dialog window is opened. If you select an address in the list box the dword of the address

is read[3]. If one want to read the range of the memory check the button auto read memory and click on refresh view[3].

Display range:

128 Bytes: only the first 128 Bytes of the chosen 1Kbyte range are shown and refreshed

1024 Bytes: the whole 1K range is shown in the list box If the size of the memory/io space is smaller only this space is shown[3].

select view range: If the size of the memory/io space is bigger than 1Kbyte then you can select another 1Kbyte block to be displayed in the list box. Just move the scrollbars for KB range and MB range[3].

Write Memory writes the DWORD value of the edit box to the chosen range with the following options:

loop on/off. continously loops accesing the chosen range stops if button is not checked

toggle. toggles (inverts) every second DWORD to be written

count. raises each WORD of the DWORD for the next write access

verify. after the chosen range is written it is read and compared to the written values[3]. If an error occurs the false values are indicated in the list box

refresh view after write. refreshes the whole list box after the chosen range is written (and verified)

Chapter 5

7 Series FPGAs Integrated Block for PCI Express

5.1 Introduction

The 7 Series FPGAs Integrated Block for PCI Express core is a scalable, high-bandwidth, and reliable serial interconnect building block for use with Xilinx Zynq-7000 All Programmable SoC, and 7 series FPGA families[5]. The 7 Series Integrated Block for PCI Express (PCIe) provides support for 1-lane, 2-lane, 4-lane, and 8-lane Endpoint and Root Port configurations at up to 5 Gb/s (Gen2) speeds, all are implementing the PCI Express Base Specification, rev. 2.1[5].The 7 Series Integrated Block for PCIe applies the PCI Express Base Specification layering model, that posses the Physical, Data Link, and Transaction Layers. The protocol utilizes packets to exchange data between layers. Packets are generated in the Transaction and Data Link Layers to transmit data from the transmitting device to the receiving device. Required data is combined in the packet being transmitted, that is needed to manage the packet at particular layer.At the receiving end, each and every layer of the receiving element processes the incoming packet, adds the important data and sends the packet to the below and above layer[5]. Due to which, the received

packets are converted from their Physical Layer representation to their Data Link Layer and Transaction Layer representations[5].

5.2 Core Interfaces

The PCI Express core involves the top-level signal interfaces which have sub-groups for the receive and transmit direction, and signals common to both directions[5].

1.System Interface

The System (SYS) interface posses the system reset signal and the system clock signal. The system input clock must be 100 MHz, 125 MHz, or 250 MHz, as chosen from the Vivado IP catalog clock and reference signals.

2.PCI Express Interface

The PCI Express interface posses the differential transmit and receive pairs arranged in multiple lanes. A PCI Express lane posses a pair of transmit differential signals (pci-exp-txp, pci-exp-txn) and a pair of receive differential signals (pci-exp-rxp, pci-exp-rxn). The 1-lane core provides support for only Lane 0, the 2-lane core provides support for lanes 0-1, the 4-lane core provides support for lanes 0-3, and the 8-lane core provides support for lanes 0-7[5].

3.Transaction Interface

The transaction interface is used by the user design to create and accept TLPs[5].

Common Interface These signals include transaction Clock (user-clk-out), transaction Reset (user-reset-out), transaction Link Up (user-lnk-up) and flow control credit signals[5].

Transmit Interface:

These signals are used to transmit outbound packets.To transmit a TLP, the user application needs to perform this sequence of events on the transmit transaction interface[5]:

1. The user application logic assigns s-axis-tx-tvalid and puts the first TLP QWORD on s-axis-tx-tdata[63:0]. If the core is assign s-axis-tx-tready, the QWORD is ac-

cepted, otherwise, the user application must keep the QWORD putted until the core assigns s-axis-tx-tready.

2. The user application assigns s-axis-tx-tvalid and puts the remainder of the TLP QWORDs on s-axis-tx-tdata[63:0] for next clock cycles [5].

3. The user application assigns s-axis-tx-tvalid and s-axis-tx-tlast together with the last QWORD data. If all eight data bytes of the last data are valid, they are putted on s-axis-tx-tdata[63:0] and s-axis-tx-tkeep is reached to 0xFF; otherwise, the four remaining data bytes are putted on s-axis-tx-tdata[31:0], and s-axis-tx-tkeep is reached to 0x0F[5].

4. At the next clock cycle, the user application deasserts s-axis-tx-tvalid to indicate the end of valid transfers on s-axis-tx-tdata[63:0].

Receive Interface

These signals are utilized to receive inbound packets.This sequence of events must occur on the receive AXI4-Stream interface for the Endpoint core to put a TLP to the user application logic[5]:

1. When the user application is ready to receive data, it assigns m-axis-rx-tready.

2. When the core is ready to transmit data, the core assigns m-axis-rx-tvalid and puts the first complete TLP QWORD on m-axis-rx-tdata[63:0].

3. The core keeps m-axis-rx-tvalid asserted, and puts TLP QWORDs on m-axis-rx-tdata[63:0] on next clock cycles

4. The core then assigns m-axis-rx-tvalid with m-axis-rx-tlast and puts either the last QWORD on s-axis-tx-tdata[63:0] and a value of 0xFF on m-axis-rx-tkeep or the last DWORD on s-axis-tx-tdata[31:0] and a value of 0x0F on m-axis-rx-tkeep[5].

5. If no other TLPs are available at the next clock cycle, the core deasserts m-axis-rx-tvalid to signal the end of valid transfers on m-axis-rx-tdata[63:0].

4.Physical Layer Interface

The Physical Layer (PL) interface sets the user design to determine the status of the Link and Link Partner and control the Link State.

5.Configuration Interface

The Configuration (CFG) interface sets the user design to determine the state of the Endpoint for PCIe configuration space. The user design gives a 10-bit configuration address, which choses one of the 1024 configuration space doubleword (DWORD) registers. The Endpoint gives back the state of the chosen register over the 32-bit data output port[5].

5.3 Detailed Example Design

The example simulation design for the Endpoint device consists of two parts[5]:
The Root Port Model: A test bench which creates, consumes, and checks PCI Express bus traffic.Test bench is utilized to simulate the example design. F
The Programmed Input/Output (PIO) example design: A completer application for PCI Express. The PIO example design responds to Read and Write requests to its memory space and can be synthesized for testing in hardware.

For the simulation design, transactions are transferred from the Root Port Model to the core and processed by the PIO example design[5]. Source code for the example is provided with the core.

Programmed Input/Output: Endpoint Example Design

Programmed Input/ Output (PIO) transactions are utilized by a PCI Express system host CPU to access Memory Mapped Input/ Output (MMIO) and Configuration Mapped Input/ Output (CMIO) locations in the PCI Express logic[5]. Endpoints for PCI Express accept Memory and I/O Write transactions and respond to Memory and I/O Read transactions with Completion with Data transactions. The PIO example design (PIO design) is included with the core in an Endpoint configuration generated by the Vivado Integrated Design Environment (IDE)[5]. The PIO design is a simple target-only application that interfaces with the Endpoint for PCIe core Transaction (AXI4-Stream) interface[5].

BAR Support

PIO design involves four transaction-specific 2 KB FPGA block RAMs (providing

a total target space of 8192 bytes) indicated by a separate Base Address Register (BAR). This 32-bit target space is accessible by single DWORD Memory Read 64, Memory Write 64, Memory Read 32, and Memory Write 32 TLPs. The PIO design creates a completion with one DWORD of payload in response to a valid Memory Read 32 TLP, Memory Read 64 TLP, or I/O Read TLP request presented to it by the core[5]. The PIO design processes a Memory Write TLP with one DWORD payload by modifying the payload into the target address in the FPGA block RAM space. By Utilizing the default parameters, the Vivado IDE develops a core programmed to work with the PIO design that consists of:

One 64-bit addressable Memory Space BAR

One 32-bit Addressable Memory Space BAR Each of the four 2 KB address spaces shown by the BARs corresponds to one of four 2 KB address regions in the PIO design. Each 2 KB region is implemented using a 2 KB dual-port block RAM[5]. As transactions are taken by the core, the core decodes the address and finds which of the four regions is destined.The core puts the TLP to the PIO design and assigns the suitable bits of (rx-bar-hit[7:0]) m-axis-rx-tuser[9:2]

TLP Data Flow

The data flow of a TLP successfully processed by the PIO design. The PIO design successfully processes single DWORD payload Memory Read and Write TLPs.[5].

Memory and I/O Write TLP Processing

When the Endpoint for PCIe takes a Memory TLP, the TLP destination address and transaction type are compared with the values in the core BARs[5]. If the TLP passes this test, the core moves the TLP to the Receive transaction interface of the PIO design. Along with the start of packet, end of packet, and ready handshaking signals, the Receive transaction interface also assigns the suitable (rx-bar-hit[7:0]) m-axis-rx-tuser[9:2] signal to show to the PIO design the particular destination BAR that matched the entering TLP[5].On receiving, the RX State Machine of the PIO design processes the incoming Write TLP and filters the TLPs data and appropriate address fields so that it can move this along to the internal block RAM write

request controller of the PIO design.Depending on the specific rx-bar-hit[7:0] signal assigned, the RX State Machine shows to the internal write controller the appropriate 2 KB block RAM.[5]. When the write TLP is processing in the FPGA block RAM, the PIO design RX state machine deasserts the m-axis-rx-tready, due to which the Receive transaction interface to stops accepting other TLPs until the internal Memory Write controller finishes the write to the block RAM[5]. Deasserting m-axis-rx-tready in this way is not needed for all designs utilizing the corethe PIO design utilizes this method to simplify the control logic of the RX state machine[5].
Memory Read TLP Processing A noticable variation in managing memory write and read TLPs is the need of the receiving device to return a Completion with Data TLP in the case of memory or I/O read request[5].

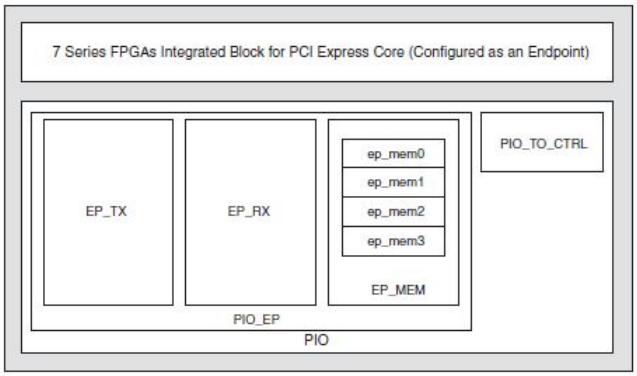


Figure 5.1: PIO Design Component[4]

64-bit PIO application top-level connectivity

The datapath width (32, 64, or 128 bits) based on which Endpoint for PCIe core is utilized[5]. The PIO-EP module posses the PIO FPGA block RAM modules and transmit and receive engines. The PIO-TO-CTRL module is the Endpoint Turn-Off controller unit, that responds to power turn-off message from the host CPU with an acknowledgment. The PIO-EP module connects to the Endpoint transaction and Configuration (cfg) interfaces[5].

Receive Path: The datapath of the module must match the datapath of the core

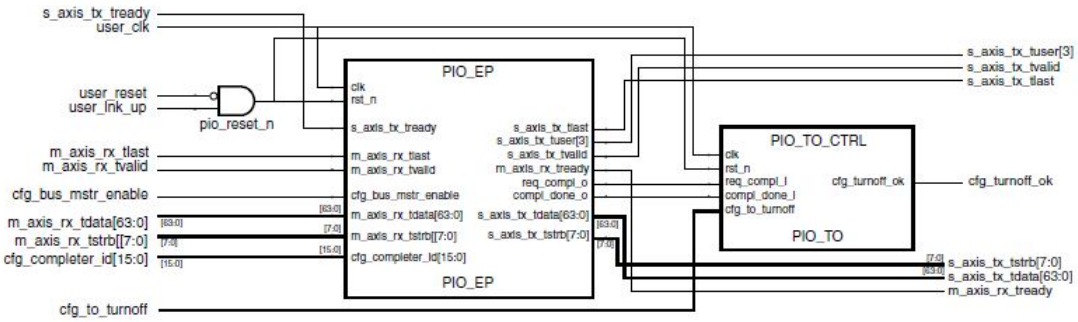


Figure 5.2: PIO 64-bit Application[4]

being utilized. These modules connect with Endpoint for PCIe Receive interface.The RX state machine filters required data from the TLP and moves it to the memory controller,The RX Engine passes one DWORD 32- and 64-bit addressable memory write requests. The RX state machine filters required data from the TLP and moves it to the memory controller, The read datapath stops taking new TLPs from the core when the application is processing the current TLP. This is done by m-axis-rx-tready deassertion[5]. For current Memory Read transaction, the module waits for compl-done-i input to be assigned before it takes the next TLP, when current Memory or I/O Write transaction is complete after wr-busy-i is deasserted.

Transmit Path: These module connects with the core Transmit interface. The PIO-TX-ENGINE module creates completions for received memory and I/O read TLPs. The PIO design does not create outbound read or write requests[5]. The PIO-TX-ENGINE module creates completions in response to one DWORD 32- and 64-bit addressable memory and I/O read requests. Data required to create the completion is moved to the TX Engine,After the completion is sent, the TX engine assigns the compl-done-i output showing to[5]: the RX engine that it can assign m-axis-rx-tready and continue getting TLPs.

Endpoint Memory: This module possesw the Endpoint memory space.The PIO-EP-MEM-ACCESS module uses data written to the memory from incoming Memory Write TLPs and gives data read from the memory for the Memory Read TLPs[5].

The EP-MEM module uses one DWORD 32- and 64-bit addressable Memory and I/O Write requests depending on the data received from the RX Engine. When the memory controller is computing the write, it assigns the wr-busy-o output showing it is busy.Both 32- and 64-bit Memory and I/O Read requests of one DWORD are computed. After the read request is computed, the data is returned on rd-data-o[31:0][5].

Root Port Model Test Bench for Endpoint

The PCI Express Root Port Model is a test bench environment that supports a test program interface that can be utilized with the provided PIO design or with a user design[5].

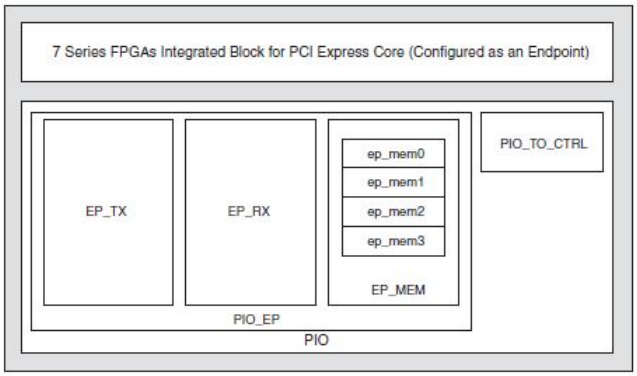


Figure 5.3: Root Port Model and Top-Level Endpoint[4]

Explanation of Architecture: The usrapp-tx and usrapp-rx blocks interface with the dsport block for transmission and reception of TLPs to/from the Endpoint Design Under Test (DUT). The Endpoint DUT posses of the Endpoint for PCIe and the PIO design (displayed) or customer design[5].The usrapp-tx block transmits TLPs to the dsport block for transmission across the PCI Express Link to the Endpoint DUT. In turn, the Endpoint DUT device sends TLPs across the PCI Express Link to the dsport block, which are moved to the usrapp-rx block. The dsport and core are responsible for the data link layer and physical link layer processing when communicating across the PCI Express logic. Both usrapp-tx and usrapp-rx uses

the usrapp-com block for shared functions, for example, TLP processing and log file outputting[5]. Transaction sequences or test programs are initialized by the usrapp-tx block to stimulate the logic interface of the Endpoint device. TLP responses from the Endpoint device are received by the usrapp-rx block[5]. Communication between the usrapp-tx and usrapp-rx blocks allow the usrapp-tx block to check correct behavior and act accordingly when the usrapp-rx block has taken TLPs from the Endpoint device[5].

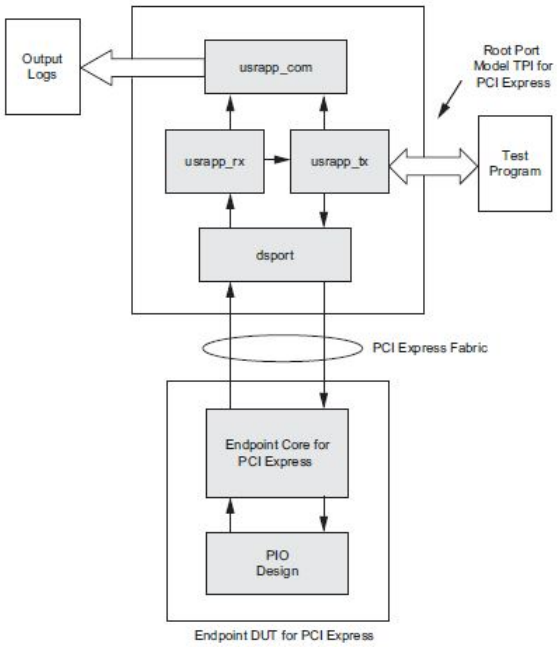


Figure 5.4: Root Port Model and Top-Level Endpoint[4]

Simulating the Example Design: The tests are provided with the Root Port Model to see how to utilize the test programming interface.pio-writeReadBack- test0 is the default test provided by the Root Port Model. **PIO Simulator Expected Output:** The PIO design simulation should give the output as follows: Running test pio-writeReadBack-test0.....

[0] : System Reset Asserted...

[4995000] : System Reset De-asserted...

[48743324] : Transaction Reset Is De-asserted...

[50471408] : Transaction Link Is Up...

[50535337] : TSK-PARSE-FRAME on Transmit

[53799296] : TSK-PARSE-FRAME on Receive

[58535316] : Check Max Link Speed = 2.5GT/s - PASSED

[58535316] : Check Negotiated Link Width = 01x - PASSED

[58583267] : TSK-PARSE-FRAME on Transmit

[60967220] : TSK-PARSE-FRAME on Receive

[66583220] : Check Device/Vendor ID - PASSED

[66631220] : TSK-PARSE-FRAME on Transmit

[69031328] : TSK-PARSE-FRAME on Receive

[69031328] : TSK-PARSE-FRAME on Receive

[74631328] : Check CMPS ID - PASSED

[74631328] : SYSTEM CHECK PASSED

[74631328] : Inspecting Core Configuration Space...

[74679316] : TSK-PARSE-FRAME on Transmit

[76327322] : TSK-PARSE-FRAME on Transmit

[77031308] : TSK-PARSE-FRAME on Receive

[78727272] : TSK-PARSE-FRAME on Receive

[84375277] : TSK-PARSE-FRAME on Transmit

[86023267] : TSK-PARSE-FRAME on Transmit

[86727220] : TSK-PARSE-FRAME on Receive

[88423220] : TSK-PARSE-FRAME on Receive

[94071220] : TSK-PARSE-FRAME on Transmit

[95719220] : TSK-PARSE-FRAME on Transmit

[96423288] : TSK-PARSE-FRAME on Receive

[98119322] : TSK-PARSE-FRAME on Receive

[103767322] : TSK-PARSE-FRAME on Transmit

[105415337] : TSK-PARSE-FRAME on Transmit

[106119316] : TSK-PARSE-FRAME on Receive

[107815316] : TSK-PARSE-FRAME on Receive

[113463267] : TSK-PARSE-FRAME on Transmit

[115111308] : TSK-PARSE-FRAME on Transmit

[115815207] : TSK-PARSE-FRAME on Receive

[117511220] : TSK-PARSE-FRAME on Receive

[123159220] : TSK-PARSE-FRAME on Transmit

[124807220] : TSK-PARSE-FRAME on Transmit

[125511308] : TSK-PARSE-FRAME on Receive

[127207296] : TSK-PARSE-FRAME on Receive

[132855337] : TSK-PARSE-FRAME on Transmit

[134503288] : TSK-PARSE-FRAME on Transmit

[135207316] : TSK-PARSE-FRAME on Receive

[136903316] : TSK-PARSE-FRAME on Receive

[142503316] PCI EXPRESS BAR MEMORY/IO MAPPING PROCESS BEGUN...

BAR 0: VALUE = 00000000 RANGE = fff00000 TYPE = MEM32 MAPPED

BAR 1: VALUE = 00000000 RANGE = 00000000 TYPE = DISABLED

BAR 2: VALUE = 00000000 RANGE = 00000000 TYPE = DISABLED

BAR 3: VALUE = 00000000 RANGE = 00000000 TYPE = DISABLED

BAR 4: VALUE = 00000000 RANGE = 00000000 TYPE = DISABLED

BAR 5: VALUE = 00000000 RANGE = 00000000 TYPE = DISABLED

EROM : VALUE = 00000000 RANGE = 00000000 TYPE = DISABLED

[142503316] : Setting Core Configuration Space...

[142551308] : TSK-PARSE-FRAME on Transmit

[144199316] : TSK-PARSE-FRAME on Transmit

[144903193] : TSK-PARSE-FRAME on Receive

[145847316] : TSK-PARSE-FRAME on Transmit

[146567204] : TSK-PARSE-FRAME on Receive

[147495316] : TSK-PARSE-FRAME on Transmit

[148199270] : TSK-PARSE-FRAME on Receive

[149143316] : TSK-PARSE-FRAME on Transmit

[149863267] : TSK-PARSE-FRAME on Receive

[150791328] : TSK-PARSE-FRAME on Transmit

[151495316] : TSK-PARSE-FRAME on Receive

[152439322] : TSK-PARSE-FRAME on Transmit

[153159316] : TSK-PARSE-FRAME on Receive

[154087296] : TSK-PARSE-FRAME on Transmit

[154791316] : TSK-PARSE-FRAME on Receive

[155735315] : TSK-PARSE-FRAME on Transmit

[156455316] : TSK-PARSE-FRAME on Receive

[158087322] : TSK-PARSE-FRAME on Receive

[171735277] : Transmitting TLPs to Memory 32 Space BAR 0

[171783193] : TSK-PARSE-FRAME on Transmit

[171991308] : TSK-PARSE-FRAME on Transmit

[174247296] : TSK-PARSE-FRAME on Receive

[179943316] : Test PASSED — Write Data: 01020304 successfully received

[180103267] : Finished transmission of PCI-Express TLPs

Time: 180103267 ps Iteration: 6 Instance: /board/RP/tx-usrapp[5]

Conclusion from simulaion: The Root Port Model TPI tasks include these tasks:

1.First task is initializing the system, for which root port model Waits for transaction interface reset and link-up between the Root Port Model and the Endpoint DUT[5].

2.Second task is to check that the options selected from the GUI are set correctly or not. Here, first task is to check link speed and link width. For that root port model sends Type 0 configuration read request TLP to the Endpoint and completion with data TLP is returned from the Endpoint. Similar check performed for the device ID and Vendor ID checks[5].

3.Next task is to scan PCI core's configuration registers. For this root port model

Performs a sequence of PCI Type 0 Configuration Writes and Configuration Reads using the PCI Express logic to determine the memory and I/O requirements for the Endpoint[5].

4.Next task is PCIE BAR Memory mapping. For this root port model looks at the range values read from configuration space and builds corresponding memory maps[5].

5.Next task is to program PCI core's configuration registers. Here, Base address register of user choice is programmed along with the command register.To transfer TLPs onto he link, the Bus Master Enable bit which is bit 2 of the PCI Command register at address offset 0x04 in the configuration space is set and to receive memory or IO TLPs the memory or I/O enable bits, bits 0 and 1, is set in the PCI Command register. If these bits are not set then the core will not accept the transfer[5].

6.Finally, the Memory Write TLPs are transmitted to the System Memory 32 space BAR 0 and this TLP contains data that is specified in the pio-writeReadBack-test0[5].

Chapter 6

ZC706 PCIe Design Creation

6.1 ZC706 PCIe Design Creation

This design is all about generating the X4 Gen 2 PCIe Core. For this design I used Xilinx Vivado Design Suite 2015.4, Design Edition and PciTree Bus Viewer[6].

Steps to implement the ZC706 PCIe Design:

1.First of all I downloaded and extracted the ZC706 PCIe Design Files (2015.4 C) zip file, in the C:/drive:of my computer. I downloaded this file from the Xilinx site[6].

2.To generate the X4 Gen 2 PCIe core first of all I opend vivado in my laptop and then I created a new project, then I selected zc706 board in board selection pane of the Vivado. By clicking on IP catalog icon I selected 7 Series Integrated Block for PCI Express, v3.2 under Standard Bus Interfaces then Under the Basic tab I set Component name to zc706-pcie-x4-gen2, Development Board to ZC706 ,Silicon to GES and Production, the Lane Width to X4, Set the Max Link Speed to 5 GT/s and the Ref Clock to 100 MHz. Under the BARs tab, set BAR 0 and Set to 1 Megabytes and then Click OK. Then I clicked on generate so the output products were generated and then PCIe design appeared in Design Sources, I Waited until checkmark appeared on zc706-pcie-x4-gen2-synth-1 then I right-clicked

on zc706-pcie-x4-gen2 and selected Open IP Example Design and set the location to C:/zc706-pcie and clicked OK[6].

3.A new project is created then in the XDC file, xilinx-pcie-7x-ep-x4g2-ZC706.xdc, I added these lines[6]:

set-property BITSTREAM.GENERAL.COMPRESS TRUE [current-design]

set-property CFGBVS VCCO [current-design]

set-property CONFIG-VOLTAGE 3.3 [current-design]

Explanations of the XDC constraints :

BITSTREAM.GENERAL.COMPRESS TRUE: Shrinks the bitstream

CFGBVS VCCO: Set to VCCO when CONFIG-VOLTAGE is either 2.5 or 3.3 V[6]

CONFIG-VOLTAGE 3.3: The ZC706 Configuration Bank (Bank 0) voltage is connected to 3.3 V

After this modification in the .xdc file I Clicked on Generate Bitstream, and then I opened and viewed the Implemented Design[6].

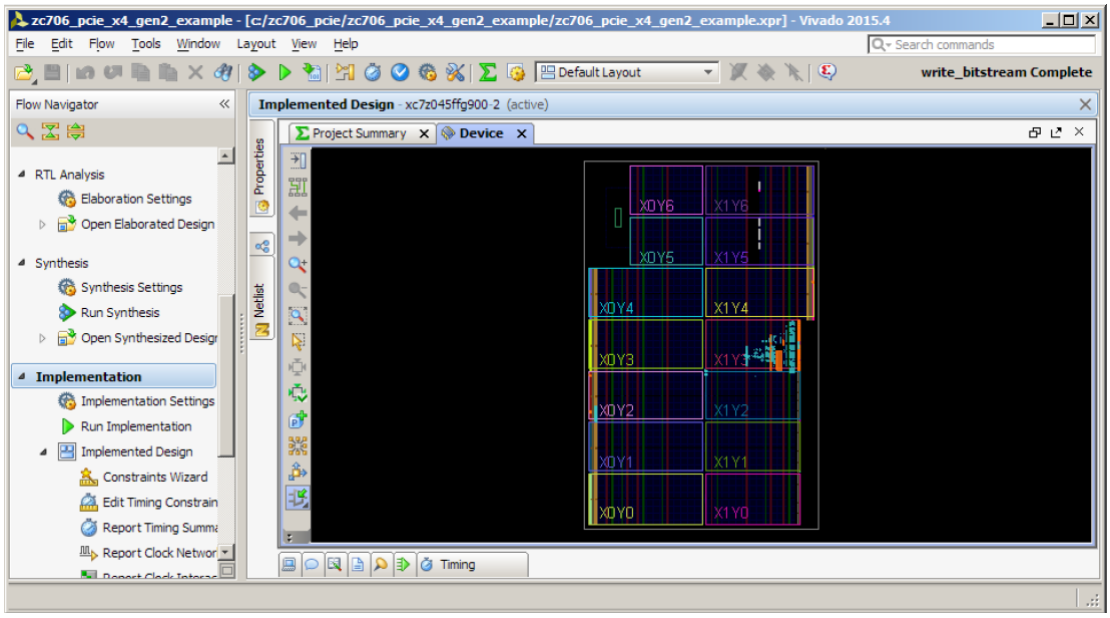
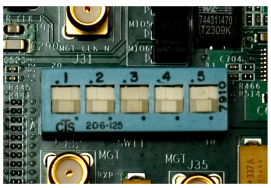


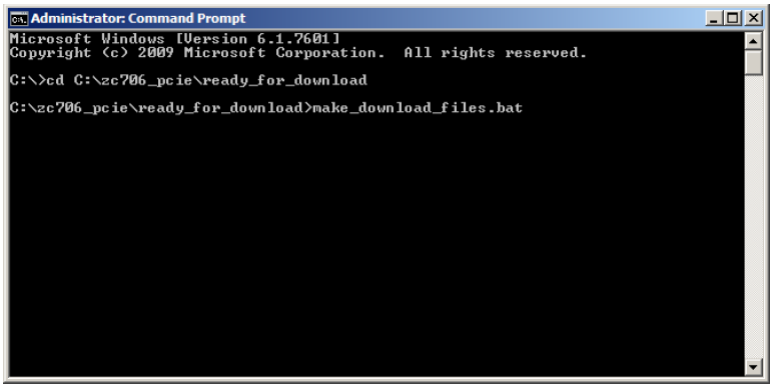
Figure 6.1: Compiled Example Design [6]

4.On the ZC706 board Set the SW11 DIP switches to: 00000

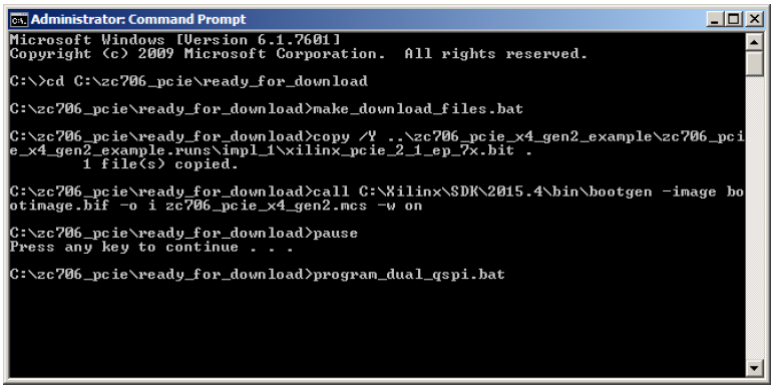


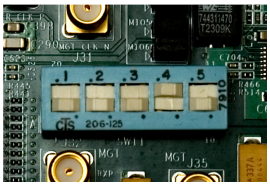
5.I Connected a USB Type-A to Micro-B cable to the USB JTAG (Digilent) connector on the ZC706 board and then Powered on the ZC706 board [6].

6.Then to generate PCIe MCS File from a Windows prompt I typed: cd C: zc706-pcie ready-for-download and make-download-files.bat [6]



7.To program Dual QSPI Flash with PCIe Design from a Windows prompt I typed program-dual-qspi.bat.[6]





8.Then I set the SW11 DIP switches to boot from QSPI: 00010[6]

9.Then I connected PCIE extender cable to the PCIE slot 4 on the motherboard of the host computer and to the ZC706 board[6].

For Running the PCIe x4 Gen 2 Design I have used PCItree software on the host computer. It is a graphical Windows tool to view each of the hardware devices of the PCIbus. The devices are visualized in a tree like view. Details about the devices and its vendors can be observed from a seperate database. PCItree provides us read and write access to the config registers of each device and even to each device's memory provided by the BAR. This tool also helps us to figure out problems with our PC, or lets us debug our custom PCI chip[6].

10.Now first of all I powered on zc706 board and then I powered on the host PC. Then, I started PCITree software tool and here I set number of configuration registers to 64 and then I clicked on refresh dump[6].

11.At PCI Tree I located to the Xilinx Device, there I observed Vendor ID is 0x10EE and the x4 Gen 2 configuration have a Device ID of 0x7024. Then I navigated to the linked list in configuration space to locate the PCIe Capabilities Structure. With the Xilinx device selected, I selected register 0x40 and it points to the next structure which is 0x48 is the address of the next structure. Then I Selected register 0x48 that points to the next structure 0x60 that is the address of the next structure. Then I observed that register 0x60 is a type 0x10, indicating PCIe Capabilities Structure that is the last Structure. Then I selected register 0x6C that is the link capabilities register which indicated the maximum number of lanes and speed (Gen 1, Gen 2) for device. The value 0x42 shows this is an x4 Gen 2 device (1). Register 0x70 is the Link Status Register which Shows the current link

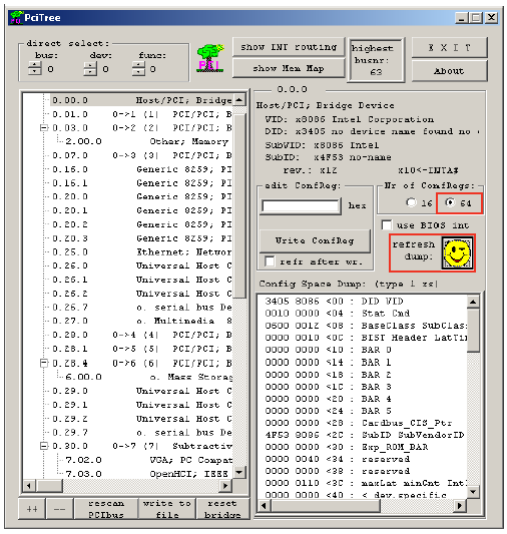
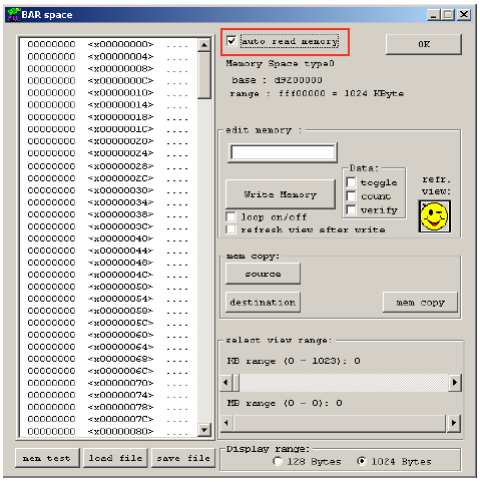


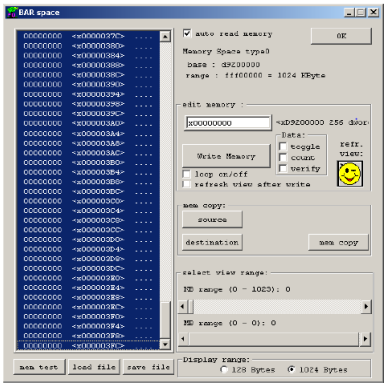
Figure 6.2: PCI Tree [6]

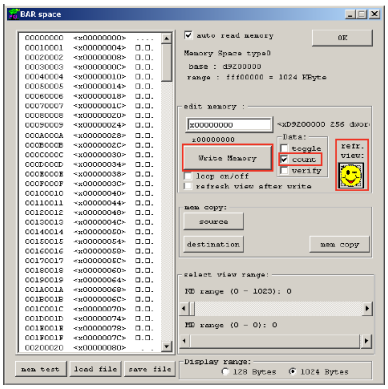
status. Then I double-clicked on BAR 0 this address is machine dependent, then I selected auto read memory as shown in figure[6].



then I clicked on the first memory location and typed ¡Shift-End¿ to select 1024 Bytes[6]

12.Then to write on memory I clicked on Select count, clicked on Write Memory then clicked on refr view then Viewed results counting up to FF as shown in figure[6].





13.Finally to restore memory I Deselected count, clicked on write memory, clicked on refr view and then memory is reset to zeros

Then I turned off PCIe host system and rthen I also turned off ZC706 board[6].

Chapter 7

ZC706 PCIe Targeted Reference Design

The overall design is a video processing card that demonstrates these capabilities[7]:

1.PCIe connectivity: It shows the utilization of the Zynq-7000 XC7Z045 AP SoC PCIe Endpoint block in x4 Gen2 configuration, PCIe bus-compatible high performance low latency multichannel DMA, Performance demonstration by using a traffic generator and checker running in Programmable logic (PL) and host software that contains a PCIe root port[7].

2.Cortex-A9 processing and offload: It shows the utilization of a XC7Z045 AP SoC to offload and process video data, The use of a Sobel filter in PL, HDMI-based display controller, Cortex-A9 multiprocessing core in the XC7Z045 AP SoC used as a video data coprocessor[7]

This design showing independent memory management in the host system and the Zynq-7000 Processing System (PS) This design demonstrates the operation of[7]:

1.PCIe Endpoint block (x4 Gen2)

2.High-speed GTX transceivers

3.High-speed multichannel DMA interface to a PCIe Endpoint

4.Zynq-7000 PS

5.Video DMA (VDMA) and Sobel filtering

6.HDMI-based display controller[7]

7.1 Data Flow

The Zynq-7000 PCIe TRD focuses on the utililization of the XC7Z045 AP SoC to offload video processing tasks from a PCIe host system[7].

1. Video Processing and Offload Demonstration on PCIe DMA Channel 0:

In the PCIe host system user application is running which generates 1920 x 1080 pixel video frames. The PCIe host system software handles channel 0 of the PCIe DMA to transmit the video stream over a x4 Gen2 PCIe link to the ZC706 board. A PCIe DMA converts the stream of PCIe video data packets into AXI streaming data, that is connected to a Video DMA (VDMA). Software that is running in the PS on the Cortex-A9 processor handles the AXI VDMA and passes the raw video frames into the PS DDR3 memory. The hardware-based Sobel filter reads the image by utilizing another VDMA and performs edge detection on the raw image and passes the data back to the PS DDR3. The processed data in PS DDR3 can either be sent back to the host system using channel 0 of the card-to-system (C2S) interface of the PCIe DMA or be visualised on the monitor using the LogiCVC display controller. Because of the limitation of the PS DDR3 bandwidth, the same data cannot be visualised and transfer back to the host system at the same time[7].

2. Generator and Checker Demonstration on PCIe DMA Channel 1:

A generator and checker on channel 1 of the PCIe DMA permits the RX and TX paths to run separately. The hardware generator in the PL produces data packets with an incremental sequence pattern. The PCIe host system software checker tests the incremental sequence pattern generated by the hardware generator. Separately, the PCIe host system driver produces a stream of incremental data that is sent via the PCIe link by the NWL PCIe DMA to the checker applied in the PL[7].

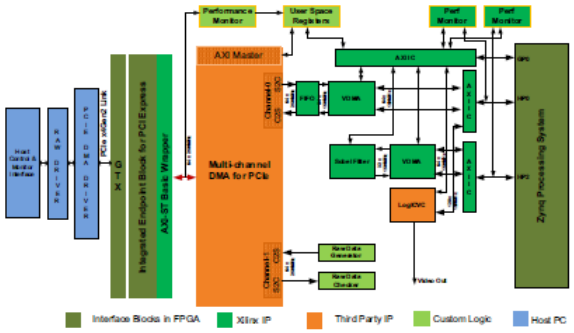


Figure 7.1: Zynq-7000 PCIe Targeted Reference Design Block Diagram [7]

7.2 Hardware Test Setup

This section represents how to set up ZC706 board, control computer, host computer, Monitor that supports 1080p and USB mouse. Here the USB mouse is required to utilize with the ZC706 board. The control PC must have Vivado Design Suite 2015.4 software installed in it. It must also have communications drivers and terminal program. Here as a terminal program I am using Tera Term Pro. Host computer must have a Fedora Core 16 Linux operating system. In this design I am using PCI Express extender cable which is connected to the host PC PCIe slot 4. This is required for establishing communication between host computer and the zc706 board[8].

Procedure to implement this design: first of all I downloaded the design files located in rdf0287-zc706-pcie-trd-2015-4.zip from the Xilinx forum in the working directory of my laptop (it is working as a control computer in this design) and then I unzipped it[8].

1. Now the first task is to program the ZC706 Board. The zynq SoC is programmed via the bitstream in a 2 cross 128 Mb Quad-SPI flash memory therefor this bitstream is first loaded in the QSPI flash memory via the SD card plugged into the J30 on the ZC706 board. For this we need to copy whole the content present in the rdf0287-zc706-pcie-trd-2014-3/ready-to-test/prog-qspi directory to the SD card

from the control PC. Now insert this SD card into zc706 board SD card receptacle. Then I program the QSPI flash memory from via files present in the SD card for running this design. For that purpose first I establish communication between control PC and zc706 board by connecting each other with UART port (shown in figure) and using the Tera Term Pro[8].

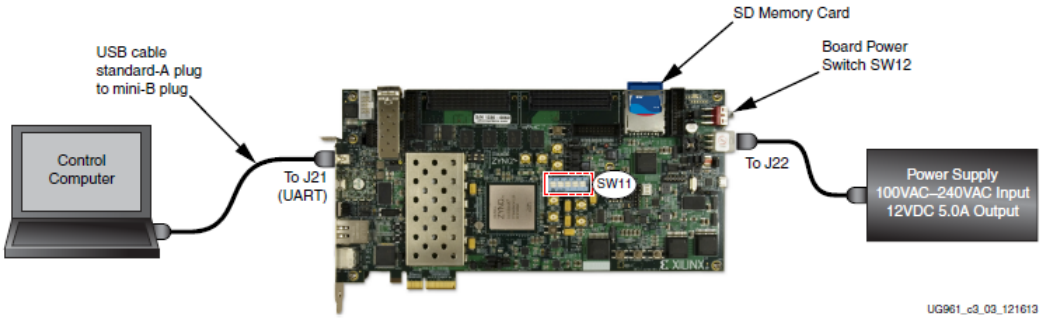


Figure 7.2: ZC706 Board Programming Setup [8]

I also set here the SW11 switch in SD boot mode. Then I power on my laptop and started the terminal program by setting baudrate to 115200 bits/s. Then I power on the board. Finally the init.sh script that is present in the SD card loads the Quad-SPI flash memory with zc706-pcie-trd.bin and the Linux kernel images and the Initialization progress is displayed on the Tera Term Pro display. After completion of the initialization process on the terminal program I turn off the board and then again I loaded whole the content of the rdf0287-zc706-pcie-trd-2015-4/ready-to-test/sd-image directory to the SD card and then I set DIP switch SW11 to Quad SPI boot mode [8].

2.Now I connected the HDMI monitor to the zc706 board and the PCIe extender cable to the host computer and the zc706 board. Now again I turn on the board and then the host PC[8].

3.This design also provides PCIe status on the GPIO LEDs near the ZC706 board power switch. The LEDs that should glow are[8]:

LED R and L should be ON and LED C should be OFF. The LEDs shows that

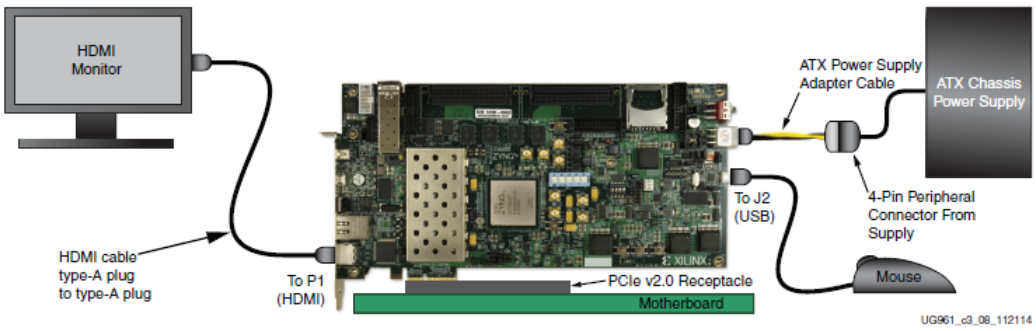


Figure 7.3: ZC706 Board TRD Setup in Host Computer [8]

Row	Labels					
Top Row	L	C	R	INIT	DONE	DS10
	Green *-BLINKING-*	Off	Green	Green	Green	Off
Bottom Row	LED1	LED2	LED3	LED4	LED5	LED6
	Green	Green	Green	Green	Green	Green

Figure 7.4: LEDS Showing PCIe Status [8]

[8]:

1.LED R- PCIe link up

2.LED C - User reset from PCIe IP

3.LED L: User clock heartbeat LED[8]

During the boot up on monitor of the host PC, the images as shown in figure below are displayed. On the HDMI monitor connected to the ZC706 board, a Qt-based application starts which gives temperature and power of the device[8].

Then, on the host pc again I downloaded rdf0287-zc706-pcie-trd-2015-4.zip file from the Xilinx site and then I copied it to the particular folder (/tmp) in the PCIe host. Then I changed permission by writing chmod 755 -R rdf0287-zc706-pcie-trd-2015-4 on a terminal to give execution permission to the files. Then inside it I double clicked to the quickstart.sh script (shown in figure) as this scripts is required to set

proper permission and initializes the driver installation GUI[8]. Then I clicked on

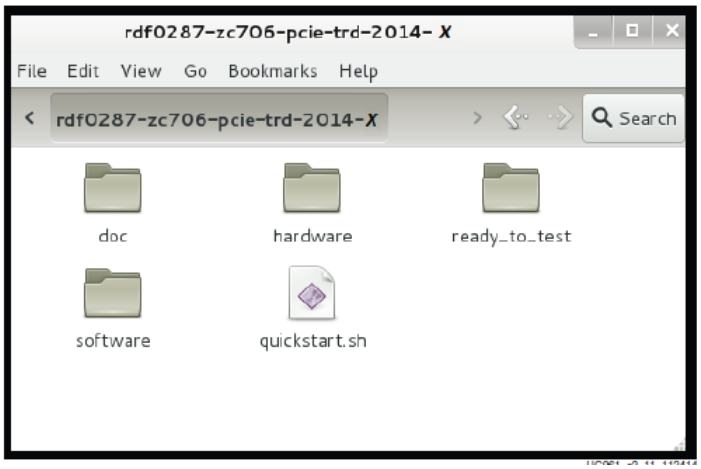


Figure 7.5: Directory Structure of the ZC706 PCIe TRD [8]

the Run in terminal option. Then the GUI with driver installation option comes out (as shown in figure). This step is necessary as it installs whole the required software important for the host system to control, transmit and receive PCIe data burst to and from the zc706 board and to monitor the performance[8].

6. Now it can be shown from the figure that the Driver installation GUI has two buttons to install various drivers related to Performance Demo and Video Demo. I Clicked the Performance Demo Install[8].

7.After installing the performance mode driver, the control and monitor user interface pops up. The control pane shows control parameters like Sobel Filter and Video Out selection modes[8].

8. Then I Clicked the Start button in the Video Path panel to initialize the PCIe host system that produces a 1080p60 video stream and transfer it over to the ZC706 board through PCIe. The video stream is processed and visualized on the HDMI monitor or transfer back to the host via PCIe, depending on the test mode chosen in the Video Out menu. The Performance Plots tab displays the system-to-card (S2C) and card-to-system (C2S) PCIe performance numbers. There are different test modes available in the Sobel Filter drop-down menu[8]:

Figure 7.6: Performance Mode GUI [8]

1.Select option None to display the frames on the monitor without Sobel.

2.Select option Sobel-HW to display the frames on the monitor with HW Sobel.

3.Select option Sobel-SW to display the frames on the monitor with SW Sobel[8].

There are also different types of test modes from the Video Out drop-down menu:

1.Select option HDMI to display Sobel data on HDMI monitor.

2.Select option PCIe Host to send data back to the PCIe host system[8].

For option Sobel Filter: None and Video Out: HDMI-Video data from PCIe host system is directly transfer to the display without being processed by the edge detection Sobel filter. A color bar pattern shows on the display as shown in figure for this option[8].

For the options Sobel Filter: Sobel-HW and Video Out: HDMI-, video data from the PCIe host system is directly processed by the edge detection Sobel filter in the PL depending on Max and Min threshold values selection provided through the host GUI, then sent to the display. Edges of the color bar pattern shows on the display as shown in Figure 3-16 for this option without invert option.

Figure 7.7: HDMI Display for Color Bar Display [8]

Optionally, the Sobel output video can be inverted by choosing Invert check box on the GUI[8].

Figure 7.8: HDMI Display for Sobel Output Display [8]

For option Sobel Filter: Sobel-SW and Video Out: HDMI-, video data from the PCIe host system is directly processed by the edge detection Sobel filter in the PS, then sent to the display. Edges of the color bar pattern shows on the display[8].

For option Video Out: PCIe Host-, video data from PCIe host system is

processed by Sobel filter in the PL or PS depending on mode selected in Sobel Filter, then sent back to the PCIe host system through PCIe. The data is not sent to the display[8].

Sobel Filter: None- is not a supported option while Video Out is set to PCIe Host.

The Qt GUI monitors the power of the device voltage rails and die temperature. The CPU utilization and PS HP port 0 and HP port 2 performance numbers are also periodically plotted. When the user selects Sobel Filter: None HP port 0 performance becomes 8 Gb/s and HP port 2 port performance becomes 0 Gb/s When the user selects Sobel Filter: Sobel-HW both HP port 0 and HP port 2 performance is close to 8 Gb/s[8]. When you select Sobel Filter: Sobel-SW, the CPU2 performance becomes 100 percent, HP port 0 performance becomes close to 8 Gb/s, and HP port 2 performance becomes 0.

As noted in the discussion above, because a single HD stream of video data is insufficient to saturate available PCIe x4 Gen2 bandwidth, datapath 1 can be turned on to add additional PCIe traffic. Click on the Start button in the Data Path-1 panel to generate additional traffic. On this path, the user can change packet sizes and see performance variation accordingly. Total PCIe BW is updated in the PCIe statistics panel and the performance plot. The user can choose Loopback, HW Generator, and the HW Checker option in the GUI for Data Path-1[8].

9.Now I installed the video mode driver and then I chose a video file present on the host file system that file is Big Buck Bunny,[8]

10.Then I click the Start button in the Video Path panel for sending the selected video file present in the Linux host system to the ZC706 board through the PCIe interface. The video stream is processed and displayed on the HDMI monitor. The Performance Plots tab shows the system-to-card (S2C) and card-to-system (C2S) PCIe performance numbers[8].

11.To select various test modes from the Filter drop-down menu: 1.Select None to display the frames on the monitor without sobel filtering.

Figure 7.9: Performance Mode Plots [8]

2.Select Sobel-HW to display the frames on the monitor with hardware sobel filtering.

3.Select Sobel-SW to display the frames on the monitor with software sobel filtering.

Then I clicked the Pause button (located below the Start button) to pause the video transmission[8].

The video being transmitted to the Zynq-7000 SOC is displayed on the host machine in it's original form through a vlc player window as shown in Figure[8].

Figure 7.10: Video running on Host [8]

When the filter option is set to None, video data from PCIe host system is sent directly to the display without being processed by the edge-detection sobel filter. The transmitted video appears on the display as shown in Figure.

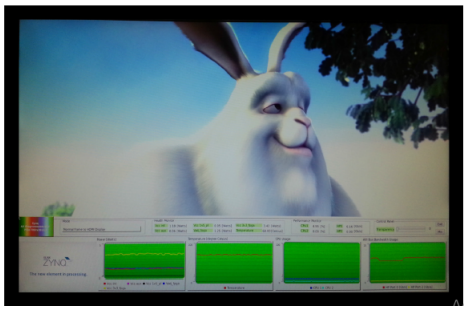


Figure 7.11: Video running on HDMI Monitor with SOBEL set to None [8]

When the filter option is set to Sobel-HW, video data from the PCIe host system is processed directly by the edge-detection Sobel filter in the PL based on Max and Min threshold values selected and provided through the host GUI, and then sent to the display. Edges of the objects appear on the display as shown in Figure when the invert option is not selected (not checked). Checking the invert option will invert the display. When the filter option is set to Sobel-SW, video data from the PCIe host system is directly processed by the edge-detection Sobel filter in the PS, then sent to the display as shown in Figure[8]. 12. Also, as mentioned in step 8, page 29,

Figure 7.12: Video running on HDMI Monitor with HW/SW SOBEL Filter [8]

data path 1 (PCIe DMA Channel 1) can be turned on in this mode to showcase the

PCIe x4 Gen2 bandwidth[8].

13.Exit the Qt GUI by clicking the Exit button in the GUI as shown in Figure[8]

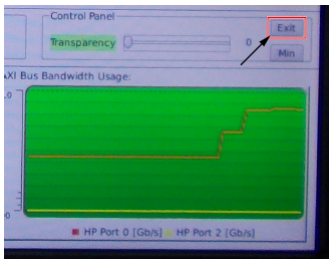


Figure 7.13: Exiting the Qt GUI [8]

Chapter 8

Introduction to Aurora Protocol

8.1 Introduction

The Aurora 8B/10B protocol is a scalable, lightweight, link-layer protocol that can be utilized to pass data point-to-point over one or more high-speed serial lanes[9]. The Aurora 8B/10B protocol is an open standard and is available for utilization by anyone without restriction. The Aurora 8B/10B protocol explains the transmission of user data over an Aurora 8B/10B channel[9]. An Aurora 8B/10B channel posses of one or more Aurora 8B/10B lanes. Each Aurora 8B/10B lane is a full-duplex serial data connection. The devices that communicate over the channel are called channel partners[9]. The Aurora 8B/10B protocol interfaces moves data and control to and from user applications by way of the user interface. Data flow posses the transmission of user PDUs and user flow control messages between the user application and the Aurora 8B/10B interface, and the transfer of channel PDUs, and flow control PDUs across the Aurora 8B/10B channel[9].

8.2 8B/10B Data Transmission and Reception

The minimum unit of data that is sent across an Aurora 8B/10B channel is two symbols, referred a symbol-pair. The data on an Aurora 8B/10B channel (or lane)

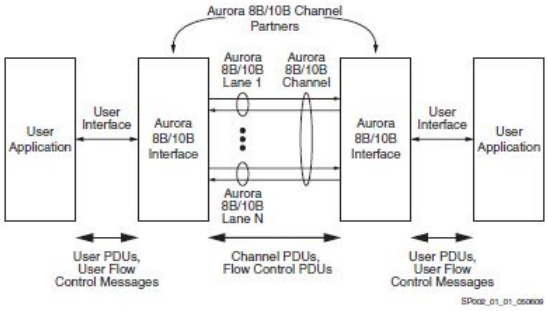


Figure 8.1: Aurora 8B/10B Channel Overview

always contains multiple symbol-pairs[9]. Utilization of the Aurora 8B/10B protocol takes a stream of octets from user applications and sent them over the Aurora 8B/10B channel as one or more streams of symbol-pairs[9].

Transmission Scheduling

The symbol six types of data are sent across an Aurora 8B/10B channel:

1.Clock Compensation Sequences:

These are the Sequences of control symbols utilized to save overrun of the receiver due to differences in clock rate between channel partners[9]. The Aurora 8B/10B protocol gives a compensating procedure for clock rate differences between the sender and receiver. This procedure, called clock compensation[9]. The Aurora 8B/10B protocol uses clock compensation by periodically inserting clock compensation sequences into idle patterns or user data[9]. Clock compensation sequences should not be putted into user flow control PDUs. The clock compensation sequence posses six copies of the clock compensation ordered set, /CC/. The clock compensation sequence sent at least every 10,000 code groups even when there are PDUs or other code groups available for transmission[9]. For multi-lane channels, the complete clock compensation sequence is sent across each lane.

2.Initialization Sequences:

These are the four ordered sets that are utilized with the idle sequence to prepare an Aurora 8B/10B channel for data transfer during initialization[9]. The initialization

procedures required to prepare an Aurora 8B/10B channel for data transmission and reception. Initialization of an Aurora 8B/10B channel is a three-stage process.

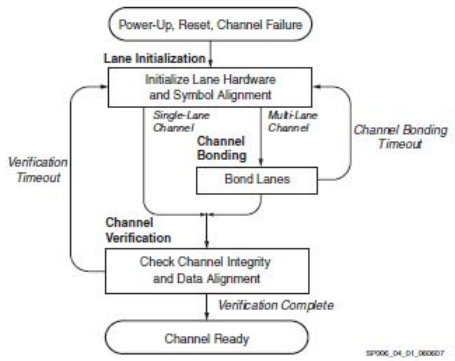


Figure 8.2: Initialization Overview

These stages are:

a.Lane Initialization: This procedure is performed individually on each lane to initiate the link. The individual lane initialization process synchronizes each lane transceiver with its lane partner upon reset or channel failure.A channel failure is described as any one of the following conditions[9]:

i.Excessive channel data errors:During the normal transfer of data, it is possible for the channel to fail.All lanes must be individually observed for errors.Before the serial transceiver properly byte aligns the data, many errors will be observed on the outputs. due to which, it is important that no error detection circuits is initialized till the lane is stable.This stable point is reached when the K counter reaches 3. Once the K counter=3, all error detection circuits should be initialized.

ii. Excessive protocol violations, implementation defined.

b.Channel Bonding: It bonds the individual lanes into a single data channel. If the function uses only single link, channel bonding is not needed and will be bypassed. Channel bonding rejects skew from different sources involving the trace lengths, connectors and IC variations. The channel bonding procedure aligns all data being received by each lane[9]. Channel bonding takes place in two phases. This first

phase, that corresponds to State CB0 posses the transceiver specific data alignment. The second phase,that corresponds to State CB1, checks that the transceivers have aligned correctly and are delivering the /A/ or-dered sets within the /I/ ordered set at the same time[9].

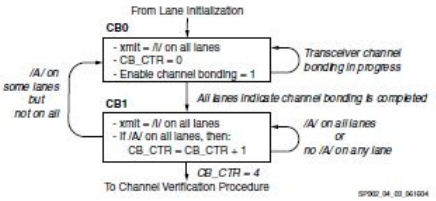


Figure 8.3: Channel Bonding Procedure

c.Channel Verification: This procedure performs any alignment required to map taken data to the user interface and checks the ability of the channel to send valid data. It essentially posses the channel verification sequence across all lanes in each direction.Using this known data pattern, the receiving channel partners can correctly align data across the user interface, and checks channel integrity. Native

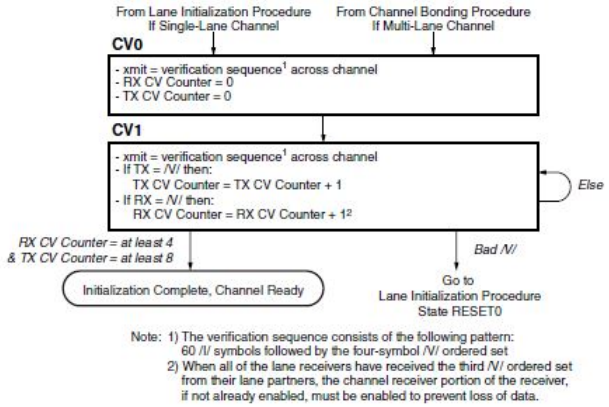


Figure 8.4: Channel Verification Procedure

Flow Control PDUs Link layer flow control PDUs created by and interpreted by Aurora 8B/10B interfaces.

3.Native Flow Control PDU Format:

Native flow control PDUs are two octets in length.The first octet is an SNF (start of native flow control) character and the second octet is a data character called the command octet.The command octet posses the PAUSE field, which indicates the number of clock cycles the idle characters must send in response to the PDU by the channel partner[9]. All native flow control PDUs send on the same cycle must carry the same PAUSE code and XOFF bit setting. When multiple native flow control PDUs reaches on the same clock cycle, the receiver must process only one PDU, selecting any of the PDUs for processing[9].

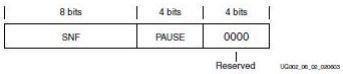


Figure 8.5: Native Flow Control Command Octet Format

4.User Flow Control PDUs: Flow control messages generated by, and interpreted by user applications, and encapsulated for transmission into user flow control PDUs[9]. User Flow Control PDU Format User flow control PDUs are 4 to 18 octets in length.The first octet is an SUF (start of user flow control) character, which is followed by a data character called the command octet.This command octet is immediately followed by 2 to 16 octets of the flow control message from the user application.The length of the user flow control message is specified in the SIZE field.The user flow control message size can be any even number of octets from 2 to 16[9].

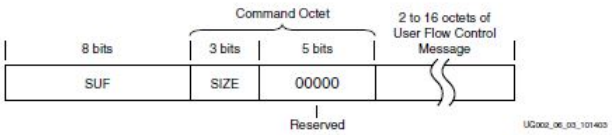


Figure 8.6: User Flow Control PDU Format

5.Channel PDUs: User PDUs encapsulated for transmission over the Aurora 8B/10B channel.

User PDU Transmission Procedures

Transmission of user PDUs requires the following procedures[9]:

Padding:The Aurora 8B/10B channel requires that all transmissions consist of an even number of symbols. To meet this requirement, all user PDUs that posses an odd number of octets must be padded with a single octet. This pad octet has a value of 0x9C and immediately follows the user PDU[9].

Encapsulation with channel PDU delimiters:The user PDU is encapsulated with control symbol sequences, called ordered sets, to produce the complete channel PDU. These ordered sets demarcate the beginning and end of channel PDUs within the serial data stream. The Aurora 8B/10B protocol uses an /SCP/ (/K28.2/K27.7/) ordered set to mark the start of channel PDUs, and an /ECP/ (/K29.7/K30.7/) ordered set to mark the end of channel PDUs[9].

8B/10B encoding of channel PDU payload:After padding, the resulting data structure is known to as the link layer payload. Prior to transmission the link layer payload is 8B/10B encoded by the physical coding sublayer (PCS)[9]. All characters, with the exception of the pad octet, are encoded as data symbols. The pad octet is encoded as a /P/ (/K28.4/) control symbol to facilitate stripping at the receiving end[9].

Serialization and clock encoding:After the complete channel PDU has been assembled and encoded, it is serialized for transmission. The serialized data stream is sent in differential non return to zero (NRZ) format[9].

User PDU Reception Procedures

Reception of user PDUs includes the following procedures:

Deserialization: The serial data stream is received in differential NRZ format. The receive logic deserializes this data into 10-bit data and control symbols. Symbol alignment within the stream is established during the lane initialization procedure and is not performed again during normal channel operation[9].

8B/10B decoding of channel PDU payload: After deserialization, the link layer payload is decoded into a stream of octets. During the decode process, the presence

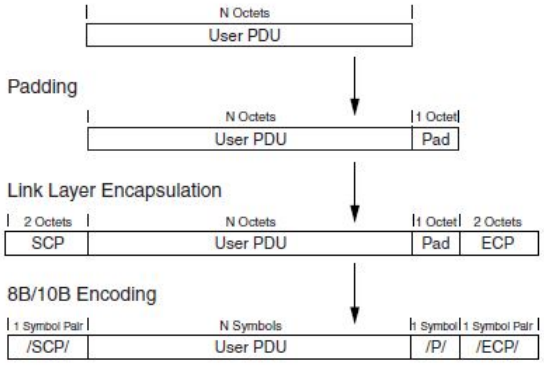


Figure 8.7: Transmission Procedures

of a /P/ (/K28.4/) control symbol followed by an/ECP/ (/K29.7/K30.7/) in the data stream must be flagged so that the pad octet can be stripped during the pad stripping procedure[9].

Link layer stripping: The link layer stripping procedure removes channel PDU encapsulation and any embedded idle ordered sets that may have been inserted during transmission[9]. Removal of channel PDU encapsulation includes the /SCP/ (/K28.2/K27.7/) ordered set to mark the start of channel PDUs, and an /ECP/ (/K29.7/K30.7/) ordered set to mark the end of channel PDUs. Removal of idle ordered sets involves the removal of /K/ (/K28.5/), /R/ (/K28.0/), and /A/ (/K28.3/) symbols. Any number of these symbols can appear at any point in the channel PDU with the following restrictions[9]: 1) An even number of idle symbols must have been inserted 2) The start of the idle sequence must begin after an even number of symbols in the channel PDU

Pad stripping: If a /P/ (/K28.4/) control symbol followed by /ECP/ (/K29.7/K30.7/) was detected during the 8B/10B decoding process, a pad octet was post-pended to the user PDU by the transmission process in order to meet channel alignment requirements[9]. This octet, which has a value 0x9C after decoding, shall be stripped from the end of the data stream before passing it to the user application[9].

6.Idle Sequences:

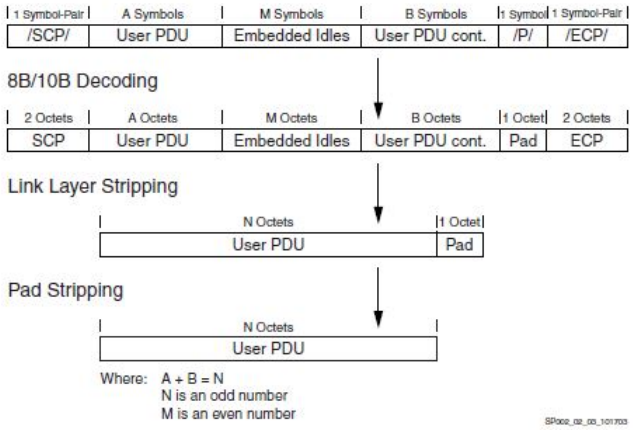


Figure 8.8: Receive Procedures

Sequences of control symbols that are transmitted whenever there is no data to send[9].

8.3 The physical coding sublayer (PCS)

The physical coding sublayer (PCS) function is responsible for idle sequence generation, lane striping, and encoding for transmission. Upon reception, the PCS is responsible for decoding, lane alignment, and destriping. The PCS in a standard implementation uses an 8B/10B encoding for transfer over the channel. for the source of the 8B/10B encoding scheme[9]. The PCS layer also provides mechanisms to detect lane states. It provides for clock difference tolerance between the sender and receiver without requiring flow control[9]. The The Aurora 8B/10B protocol does not include mechanisms for finding the number of lanes that make up the channel. Each channel partner must be programmed for operation over the same number of lanes[9].

The PCS layer performs the following list of transmit functions:

a.) Dequeues channel PDUs, native flow control PDUs, user flow control PDUs and delimited control symbols awaiting transmission as a character stream[9]

b.) Stripes the send character stream across the available lanes

c.) Generates the idle sequence and inserts it into the transmit character stream for each lane when no PDUs or delimited control symbols are available for transmission

d.) Encodes the character stream of each lane independently into 10-bit parallel code groups

e.) Moves the resulting 10-bit parallel code groups to the PMA[9]

The PCS layer performs the following receive functions:

a.) Decodes the received stream of 10-bit parallel code groups for each lane independently into characters

b.) Marks characters decoded from invalid code groups as invalid[9]

c.) Aligns the character streams to eliminate the skew between the lanes and reassembles (destripes) the character stream from each lane into a single character stream, if the channel is using more than one lane

e.) Delivers the decoded character stream of PDUs and delimited control symbols to the higher layers[9]

8.4 The physical Medium Attachment (PMA)

The physical medium attachment (PMA) function is responsible for serializing 10-bit parallel code groups to/from a serial bitstream on a lane-by-lane basis. Upon receiving data, the PMA function aligns the received bitstream into 10-bit code group boundaries, independently on a lane-by-lane basis[9]. It then gives a continuous stream of 10-bit code groups to the PCS, one stream for each lane. The 10-bit code groups are not observable by layers higher than the PCS[9].

Chapter 9

Conclusion

The SOC solution is provided by zynq architecture. SOC system combines processor and FPGA design technology together and facilitates fully integrated development tool environment. Zynq devices are targeted platform reference designs. PCI Express extends the capabilities of host system. PCI Express is more like a network than a bus. While implementing zc706 pcie targeted reference design, I successfully established communication between host PC and ZC706 bord.

Accessing of the data from computer memory for the PCIe Endpoint block is done by using PCITree software.Example design simulation helps in understanding the flow of PCIE transaction from system initialization to transmission of TLP between root port model and Endpoint DUT.

For communication between multiple Zynq boards, Aurora which is a scalable, lightweight, link-layer protocol that is used to move data across point-to-point serial links are used. Aurora provides a transparent interface to the physical layer, allowing upper layers of proprietary or industry-standard protocols to easily use high-speed transceivers.

Chapter 10

Future Scope

In the future, For communication between multiple Zynq boards, Aurora protocol can be used to move data across point-to-point serial links. Direct memory access method will be very helpful for establishing communication between host PC and zc706 board. It can be explored in future for this project.

Bibliography

[1] "Introduction to Zynq Architecture ",[Online],Website,July 2016,
$https://bookshelf.vitalsource.com$

[2] "PCI Express System Architecture ",[Online],Tutorial,July 2016,
$https://www.youtube.com/watch?v=uccPR6X8vy8$
$https://www.scribd.com/doc/23817387/Pci-Express-System-Architecture$
$http://xillybus.com/tutorials/pci-express-tlp-pcie-primer-tutorial-guide-1$

[3] "Introduction to PCI Tree ",[Online],Tutorial,July 2016,
$http://www.pcitree.de/$

[4] "Figure ",[Online],Figure,September 2016,
$https://www.google.co.in$

[5] "7 Series FPGAs Integrated Block for PCI Express",[Online],Website,August 2016,
$https://www.xilinx.com/support/documentation/ip_documentation/pcie_7x/v3_2/pg054-7series-pcie.pdf$

[6] "PCIE Design",[Online],Website,August 2016,
$https://secure.xilinx.com/webreg/clickthrough.do?cid=$

413467$and license = RefDesLicense and filename = xtp246 - zc706 - pcie - c - 2015 - 4.pdf and language ID$

[7] "ZC706 PCIE Targeted Reference Design",[Online],Website,September 2016, $https://www.xilinx.com/support/documentation/boards_and_kits/zc706/2015_4/ug963-zc706-pcie-trd-ug.pdf$

[8] "Main Document",[Online],Website,September 2016, $https://www.xilinx.com/support/documentation/boards_and_kits/zc706/2015_4/ug961-zc706-GSG.pdf$

[9] "Introduction to Aurora Protocol",[Online],Website,September 2016, $https://www.xilinx.com/support/documentation/ip_documentation/aurora_8b10b_protocol_spec_s$