PROYECTO FIN DE GRADO

TITLE: Design of high-speed data acquisition systems based on the JESD204 interface with Xilinx ZynqMP devices for MicroTCA platforms

AUTOR: Alejandro Piñas Higueruela

TITULACIÓN: Grado en Ingeniería Electrónica de Comunicaciones

TUTOR: Mariano Ruiz González

DEPARTAMENTO: Ingeniería Telemática y Electrónica

VºBº TUTOR

Miembros del Tribunal Calificador:

PRESIDENTE: César Briso Rodríguez

TUTOR: Mariano Ruiz González

SECRETARIO: Antonio Carpeño Ruiz

Fecha de lectura:

Calificación:

El Secretario,

Agradecimientos

Primero, gracias a Mariano y Antonio por abrirme una puerta que ha marcado mi vida y seguirá marcando mi futuro. Gracias también al resto de miembros y compañeros del I2A2: Guillermo, José Carlos, César, Víctor, Nacho, Raúl, Julián y Eduardo, por ayudarme cuando lo he necesitado y amenizar los días de trabajo.

Gracias a Pablo, Pedro Luis, Juan Manuel, Luka y Conrado por las risas y por compartir conmigo estos últimos años.

A Claudia por estar ahí siempre y ser mi mayor apoyo.

Finalmente, gracias a mi familia por permitirme estudiar una carrera que realmente he disfrutado, con la que he madurado y con la que he descubierto a lo que me quiero dedicar.

Resumen

MicroTCA (Micro Telecommunications Computing Architecture) se utiliza ampliamente en investigación científica y experimentos a gran escala para implementar sistemas de adquisición de datos de alta velocidad. Las FPGA (Field Programmable Gate Array) y los SoC (System on Chip) se utilizan habitualmente como elemento de procesamiento de los módulos AMC (Advanced Mezzanine Card) que forman este tipo de sistemas ya que los dispositivos basados en FPGA ofrecen gran flexibilidad y rendimiento, que se aprovecha utilizando módulos FMC (FPGA Mezzanine Card) para conectar el dispositivo de procesamiento con convertidores de datos de alta velocidad. A medida que han aumentado la resolución y la velocidad de los convertidores, la interfaz serie JESD204 se ha establecido como la interfaz de elección para los FMC de adquisición de datos.

Este proyecto explora las posibilidades y opciones para implementar un sistema de adquisición de datos y procesamiento heterogéneo utilizando las últimas metodologías de aceleración de hardware. El sistema se desarrollará para un módulo AMC fabricado por NAT Europe, que está equipado con un Zynq MPSoC (Multi-Processor SoC) de Xilinx. Mediante las herramientas de desarrollo de Xilinx, se pueden utilizar lenguajes de síntesis de alto nivel para implementar funciones aceleradas en la región lógica programable del dispositivo utilizando el estándar de computación OpenCL (Open Computing Language).

El sistema desarrollado integra dos componentes diferentes. En primer lugar, un diseño hardware estático que se implementa en la FPGA; el cual proporciona tanto la infraestructura para la interfaz JESD204, como los recursos necesarios para ejecutar funciones aceleradas o *kernels*. En segundo lugar, una distribución Linux embebida que se ejecuta en los núcleos ARM (Advanced RISC Machine) presentes en el SoC; su función es configurar los periféricos implementados en la FPGA y gestionar el *runtime* de OpenCL.

Este documento explica además el procedimiento utilizado para verificar el correcto funcionamiento de las partes hardware y software del sistema. Asimismo, se desarrolla una aplicación de ejemplo de adquisición de datos utilizando OpenCL y síntesis de alto nivel para demostrar la utilidad del sistema en la adquisición de señales de conversores ubicados en un módulo FMC. Como resultado, se ha obtenido un sistema completamente funcional que puede ser utilizado para implementar aplicaciones de adquisición de datos de alta velocidad y que requieran altas capacidades de procesamiento. Todo ello con la ventaja de poder implementarlas utilizando lenguajes de síntesis de alto nivel reduciendo el tiempo de desarrollo y aumentando el nivel de abstracción.

Abstract

MicroTCA (Micro Telecommunications Computing Architecture) is widely used in scientific research and large-scale experiments to implement high-speed data acquisition systems. FPGAs (Field Programmable Gate Array) and SoCs (System on Chip) are commonly used as the processing element of the AMC (Advanced Mezzanine Card) modules that build this type of system because FPGA-based devices provide great flexibility and performance, which is leveraged using FMC (FPGA Mezzanine Card) modules to connect the processing device with high-speed data converters. As the resolution and speed of converters have increased, the JESD204 serial interface has been established as the interface of choice for data acquisition FMCs.

This project explores the possibilities and options for implementing a data acquisition and heterogeneous processing system using the latest hardware acceleration methodologies. The target for developing such a system is an AMC module manufactured by NAT Europe, which is equipped with a Xilinx Zynq MPSoC (MultiProcessor SoC). Using Xilinx's development tools, high-level synthesis languages can be used to implement accelerated functions on the programmable logic region of the device using the OpenCL (Open Computing Language) computing standard.

The developed system integrates two different components. First, a static hardware design implemented on the FPGA provides both the infrastructure for the JESD204 interface and the resources required to support accelerated functions or *kernels*. Second, an embedded Linux distribution that runs on the ARM (Advanced RISC Machine) cores present in the SoC configures the peripherals implemented in the FPGA fabric and manages the OpenCL runtime.

This document explains the procedure used to verify the correct functioning of the system's hardware and software parts. In addition, an example data acquisition application is developed using OpenCL and high-level synthesis to demonstrate the system's usefulness for acquiring signals from a converter located in an FMC module. As a result, a fully functional system has been obtained that can be used to implement high-speed data acquisition applications requiring high processing capabilities. All this with the advantage of being able to implement them using high-level synthesis languages, reducing the development time and increasing the level of abstraction.

Table of contents

List of figures 7
List of tables 8
Acronym list 9
Chapter 1. Introduction 13
Chapter 2. Technological framework 15
2.1 Micro Telecommunication Computing Architecture 15
2.1.1 Chassis and Backplane 16
2.1.2 MicroTCA Carrier Hub 17
2.1.3 Advanced Mezzanine Cards 18
2.1.4 Power Modules and Cooling Units 20
2.2 JESD204 Converter Interface 20
2.2.1 Link Parameters 20
2.2.2 Functional Layers 21
2.2.3 Link Establishment 22
2.3 Xilinx Application Acceleration Methodology 23
2.3.1 Platform Types 23
2.3.2 Embedded Platform Components 24
2.3.3 Embedded Platform Application Acceleration Development Flow 25
2.3.4 Execution Model 26
Chapter 3. Design specifications and proposed solution 27
3.1 Hardware Setup 27
3.1.1 NAT-AMC-ZYNQUP-FMC Hardware 28
3.1.2 FMC Loopback Card 29
3.2 Design of the Hardware Platform 30
3.2.1 NAT-AMC-ZYNQUP-FMC Reference Design 30
3.2.2 JESD204B Interface 32
3.2.3 Support for acceleration kernels 35
3.2.4 Final Hardware Design 39
3.3 Design of the Software Platform 40
3.3.1 NAT-AMC-ZYNQUP-FMC Reference Petalinux Project 40
3.3.2 JESD204B Software Support 41
3.3.3 Xilinx Run Time 43
3.4 Project Structure 44
Chapter 4. Results 45
4.1 GitHub repository 45

4.2 JESD204B Interface Results 47
4.2.1 Linux Driver Initialization 47
4.2.2 JESD204B Status Utility 49
4.2.3 Hardware Analysis 52
4.3 PCIe Interface Results 55
4.3.1 XDMA Linux Driver 55
4.3.2 XDMA Test Application 57
4.4 OpenCL Applications 58
4.4.1 zocl Linux Kernel Module 58
4.4.2 Vector Addition 58
4.4.3 Data Acquisition 59
4.5 Hardware Resource Utilization 61
Chapter 5. Project impact, dissemination, conclusions, and future work 63
5.1 Project Impact 63
5.2 Dissemination 64
5.3 Conclusions 64
5.4 Future Work 65
References 67
APPENDIX I. Budget 71
APPENDIX II. JESD204B Device-tree 73

List of figures

Figure 1. Example of a typical MicroTCA system architecture....16
Figure 2. Example of a MicroTCA chassis with three AMC boards, PCIe, and GbE uplinks through MCH....16
Figure 3. MicroTCA.4 Backplane [6] 17
Figure 4. NAT-MCH-PHYS80 [8]....18
Figure 5. NAT-MCH-PHYS80 hardware block diagram [8]....18
Figure 6. Generic AMC block diagram....19
Figure 7. NAT-AMC-ZYNQUP-FMC, the image on the left shows a front view, while the one on the right shows a top-view where the FMC connector is visible 19
Figure 8. ADC3112, four channel 1 GS/s 12-bit ADC [12]....19
Figure 9. JESD204B single link example with L lanes and M converters.....20
Figure 10. JESD204 functional layers....22
Figure 11. Vitis application structure [16] 23
Figure 12. Xilinx platform types. On the left is a PCIe-based platform. On the right is an embedded platform. The red AXI interfaces are used for memory transfer, while the black ones are used for kernel control....24
Figure 13. Application acceleration development flow for embedded platforms....25
Figure 14. Hardware setup for the development of the project....28
Figure 15. Zynq UltraScale+ architecture block diagram 28
Figure 16. NAT-AMC-ZYNQUP-FMC block diagram [21]....29
Figure 17. On the left, the Terasic FMC loopback card. On the right, the FMC module is mounted on the NAT-AMC-ZYNQUP-FMC carrier....29
Figure 18. Terasic FMC Loopback Card Block Diagram [22]....30
Figure 19. NAT-AMC-ZYNQUP-FMC Hardware reference design....31
Figure 20. Implemented JESD204B Subsystem using Analog Devices IP blocks 34
Figure 21. Designed system hardware block diagram....39
Figure 22. NAT-AMC-ZYNQUP-FMC reference Petalinux project layers (simplified) 40
Figure 23. Petalinux project with Analog Devices layers 41
Figure 24. JESD204 Linux Kernel Framework implemented topology....43
Figure 25. jesd_status output for the RX data link core 51
Figure 26. jesd_status command output for the TX data link core....51
Figure 27.JESD204B CGS phase observed in ILA....52
Figure 28. First multiframe of the initial line alignment sequence observed in ILA 53
Figure 29. End of the first multiframe and the start of the configuration octets observed in ILA 53
Figure 30. Transition from ILAS phase to DATA phase observed in ILA....54
Figure 31. Ramp data sent and received through JESD204B observed in ILA 54
Figure 32. XDMA driver overview 55
Figure 33. Hardware setup for XDMA test application 57
Figure 34. Acquisition system block diagram....59
Figure 35. JESD204B sample mapping in the example acquisition application....60

List of tables

Table 1. MicroTCA Revisions [5] 15
Table 2. JESD204B Link parameters 21
Table 3. Control Characters used in JESD204 22
Table 4. Hardware interfaces required by Vitis for an acceleration platform 35
Table 5. Custom embedded platform kernel interfaces. 36
Table 6. JESD204B Linux device drivers 41
Table 7. FPGA Resource Utilization 61
Table 8. Complete project budget 71

Acronym list

ADC	Analog-to-Digital Converter
AMC	Advanced Mezzanine Card
API	Application Programming Interface
ARM	Advanced RISC Machine
ASIC	Application Specific Integrated Circuit
ATCA	Advanced Telecommunication Computing Architecture
AXI	Advanced eXtensible Interface
C2H	Card to Host
CCI	Cache Coherency Interconnect
CGS	Code Group Synchronization
CMA	Contiguous Memory Allocator
CML	Current Mode Logic
CMOS	Complementary Metal-Oxide Semiconductor
CMS	Compact Muon Solenoid
cPCI	Compact PCI
DAC	Digital-to-Analog Converter
DDR	Double Data Rate
DMA	Direct Memory Access
DTB	Device Tree Blob
ELF	Executable and Linkable Format
EMMC	Enhanced Module Management Controller
FMC	FPGA Mezzanine Card
FPGA	Field Programmable Gate Array
GbE	Gigabit Ethernet

GEM	Gigabit Ethernet MAC
GPL	GNU General Public License
H2C	Host to Card
HLS	High Level Synthesis
I2C	Inter-Integrated Circuit
ILA	Integrated Logic Analyzer
ILAS	Initial Lane Alignment Sequence
IO	Input / Output
IP	Intellectual Property
IPMB	Intelligent Platform Interface Bus
IPMI	Intelligent Platform Management Interface
JEDEC	Joint Electron Device Engineering Council
LHC	Large Hadron Collider
LUT	LookUp Table
MAC	Medium Access Controller
MCH	MicroTCA Carrier Hub
MCMC	MicroTCA Carrier Management Controller
MicroTCA/MTCA	Micro Telecommunication Computing Architecture
MMC	Module Management Controller
MMCM	Mixed-Mode Clock Manager
OpenCL	Open Computing Language
PCI	Peripheral Component Interconnect
PCIe	PCI express
PICMG	PCI Industrial Computer Manufacturers Group
PL	Programmable Logic
PLL	Phase Locked Loop
PS	Processing System

RAM	Random Access Memory
RFS	Root File System
RTL	Register Transfer Level
SATA	Serial Advanced Technology Attachment
SDG	Sustainable Development Goal
SRIO	Serial RapidIO
VME	Versal Module Eurocard
XDMA	Xilinx DMA
XFEL	X-Ray Free-Electron Laser
XSA	Xilinx Support Archive

Chapter 1. Introduction

High-speed data acquisition is needed in big-science and scientific research because scientists need to capture fast phenomena. To extract meaningful insights from this data, it is necessary to acquire, store, and process it quickly and efficiently. Over decades the platforms of choice to implement data acquisition systems were mainly based on VME (Versa Module Eurocard) and cPCI (compact PCI). However, the obsolescence of some key components and the development of new technologies led to adopting and developing new computing standards [1]. Such is the case for MicroTCA (Micro Telecommunications Computing Architecture), released in 2006 and derived from ATCA (Advanced Telecommunications Computing Architecture) to provide a smaller, cost-effective, and scalable architecture.

The key processing component of MicroTCA are AMCs (Advanced Mezzanine Card) which are interconnected through the backplane, usually included in the chassis. For data acquisition applications, the core of these boards is an FPGA (Field-Programmable Gate Array) or an SoC (System-on-Chip) that integrates the programmable logic and an ARM-based processing system. To take advantage of the processing capabilities of FPGA devices, the AMC cards are connected to an FMC (FPGA Mezzanine Card) that usually contains one or more high-speed data converters. With the increase in resolution and speed of converters, there has been a growing need for a more efficient interface. The JESD204 standard was introduced to fulfill this demand, as it defines a serial interface for FPGA and ASICs (Application Specific Integrated Circuit), which provides various benefits over its previous counterparts, CMOS (Complementary Metal-Oxide Semiconductor) and LVDS (Low Voltage Differential Signaling), in terms of speed, size, and cost [2].

Traditionally, FPGA applications have been developed using HDL (Hardware Description Languages), which requires a deep knowledge of hardware design principles. However, in recent years, FPGA manufacturers have integrated high-level synthesis languages into their design tools to reduce development time. Such is the case of OpenCL (Open Computing Language), which provides a common API (Application Programming Interface) to develop accelerated applications for FPGAs, GPUs (Graphical Processing Unit), and CPUs (Central Processing Unit).

This project's objective is to develop a high-speed data acquisition system for an AMC module in a MicroTCA chassis, which uses the JESD204B interface to connect with an FMC module and allows OpenCL-based application acceleration.

This document describes the development of the project, which is divided into the following chapters:

- **Chapter 2.** Describes the technological framework in which the project is framed and the technologies used.

- **Chapter 3.** Lists the specifications that the final system should meet. Then, explains the components of the designed system that is proposed as a solution.
- **Chapter 4.** This chapter describes the procedures used to verify the system's performance and the results obtained from these tests.
- **Chapter 5.** Discusses the contribution and dissemination of the project and its possible applications. Then, analyzes whether the proposed solution meets the objectives and identifies future lines of work.

Chapter 2. Technological framework

This chapter provides the necessary context for understanding the project's development. First, an overview of the MicroTCA standard is provided. Secondly, an emerging converter interface for FPGA and ASIC is introduced, the high-speed serial interface JESD204. Finally, Xilinx's proposed hardware acceleration methodology is explained, which is based on the OpenCL framework.

2.1 Micro Telecommunication Computing Architecture

MicroTCA is an open standard developed by PICMG (PCI Industrial Computer Manufacturers Group), a consortium of over 140 companies [3]. It provides a modular, scalable computing architecture to build robust and high-performance systems in a small form factor. MicroTCA was designed as a complementary standard to ATCA. While this last one is optimized for high-capacity and high-performance applications, MicroTCA addresses cost-sensitive and physically smaller applications with lower capacity [4].

There are five released MicroTCA revisions; a detailed view of the specifications is provided in Table 1.

Table 1. MicroTCA Revisions [5]

Revision	Description
MicroTCA.0	Base specification, defines electrical, mechanical, thermal, and management characteristics.
MicroTCA.1	Adds ruggedization features and forced-air cooling.
MicroTCA.2	Expands shock, vibration, and temperature operation, allowing both air and conduction cooling
MicroTCA.3	Adds more requirements about shock, vibration, and temperature conditions. Requires the use of conduction cooling.
MicroTCA.4	This revision is specifically designed for use in large-scale scientific facilities. Includes features such as precision timing and trigger signals.

A typical MicroTCA system is composed of up to twelve AMCs, an MCH (MicroTCA Carrier Hub), power modules, and cooling units, all connected through the backplane, as shown in Figure 1. A more detailed description of the system elements is provided in the following sections.

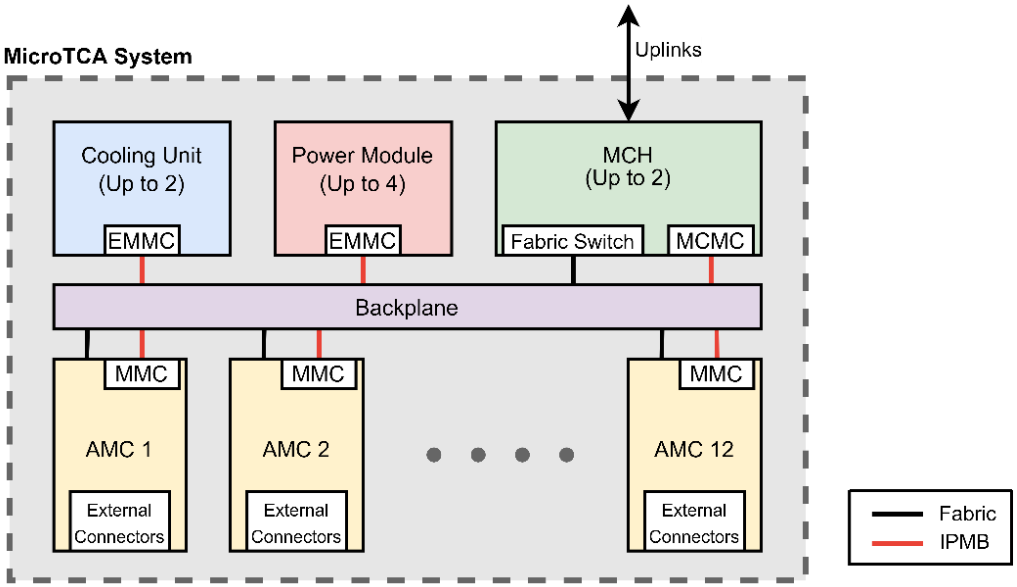

Figure 1. Example of a typical MicroTCA system architecture

2.1.1 Chassis and Backplane

The MicroTCA chassis is the enclosure that provides physical support to the components of the system, including processor modules, carrier modules, power modules, and cooling units. Different chassis sizes are available in the market, depending on the number of modules needed to meet the application requirements. A MicroTCA chassis with three AMC boards and PCIe/GbE uplinks is shown in Figure 2.

Figure 2. Example of a MicroTCA chassis with three AMC boards, PCIe, and GbE uplinks through MCH

It also includes the backplane, which serves as the system's backbone for communication and power distribution. The backplane includes connectivity for management over IPMB (Intelligent Platform Management Bus). It also provides data connectivity, as shown in Figure 3; the standard defines a specific set of interfaces to be used for each AMC port:

- **Common Options.**
  - Ports 0 and 1 implement the GbE (Gigabit Ethernet) interface.
  - Ports 2 and 3 implement the SAS (Serial Attached SCSI) or SATA (Serial Advanced Technology Attachment) interface. These are point-to-point connections used for storage.

- **Fat Pipes.**
  - Ports 4-7 implement PCIe (PCI express) or SRIO (Serial RapidIO). These are high-bandwidth interfaces capable of transmitting data at high rates.
- **Extended Fat Pipes.**
  - Ports 8-11 implement GbE or SRIO interfaces for more high-speed transfer options.
- **Extended Options.**
  - Ports 12-15 are used to implement LVDS point-to-point links between AMCs.
  - Ports 17-20 are used to implement multipoint LVDS clock, trigger, and interlock signals.
- **Clocks.**
  - The remaining ports are used for the distribution of low-jitter clocks.

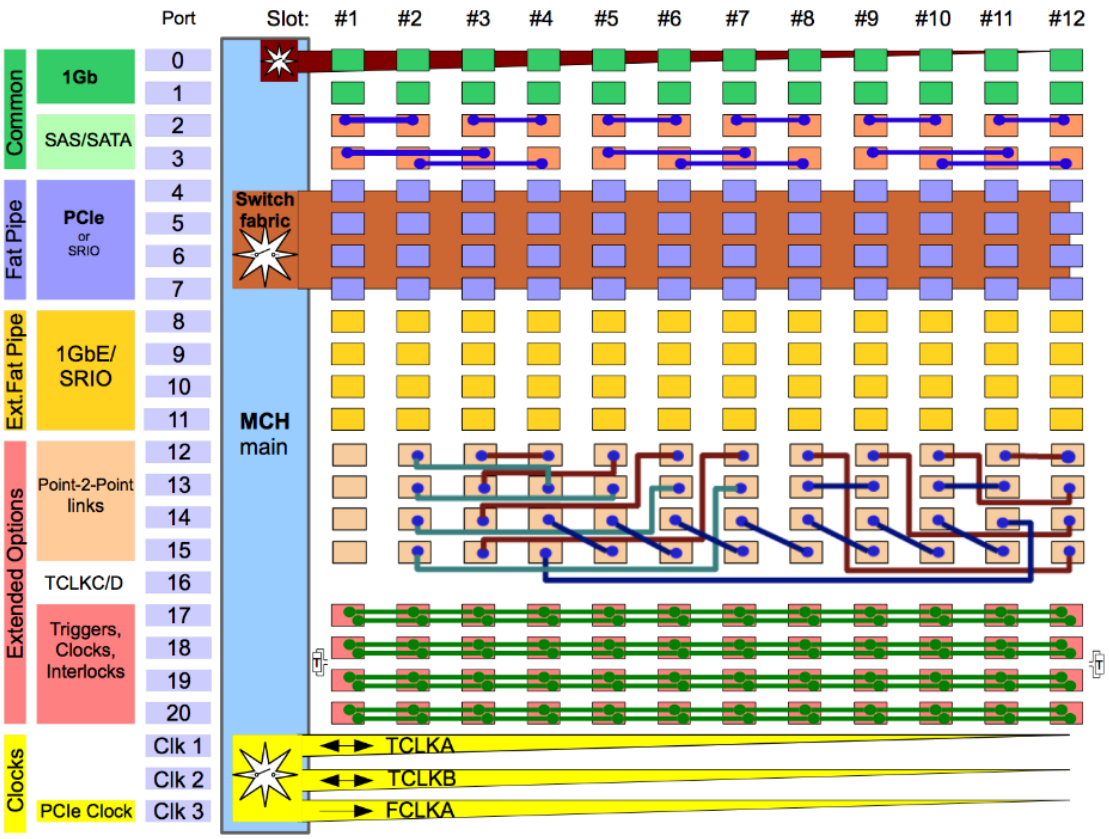

Figure 3. MicroTCA.4 Backplane [6]

2.1.2 MicroTCA Carrier Hub

The MCH is responsible for overall management, including AMCs, power, and cooling units. It uses the IPMI (Intelligent Platform Management Interface) over IPMB to communicate the MCMC (MicroTCA Carrier Management Controller) with the MMCs (Module Management Controller) and the EMMCs (Enhanced MMC) in the other system modules [7].

Another of the key functions of the MCH is switching. MicroTCA uses a passive backplane for interconnection, so the MCH is responsible for switching data between AMCs.

The specification defines a maximum of two MCHs per system to ensure the highest availability. The MCH is usually available in AMC form factor; however, it can be integrated into the chassis. In addition, it may provide other functions, such as uplinks to other systems or clock generation. Figure 4 shows the NAT-MCH-PHYS80, which is designed for physics applications providing PCIe optical uplink and switching, and a specialized clocking module. Figure 5 shows a detailed block diagram of the hardware of the device.

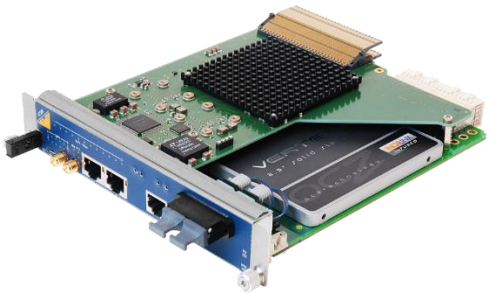

Figure 4. NAT-MCH-PHYS80 [8]

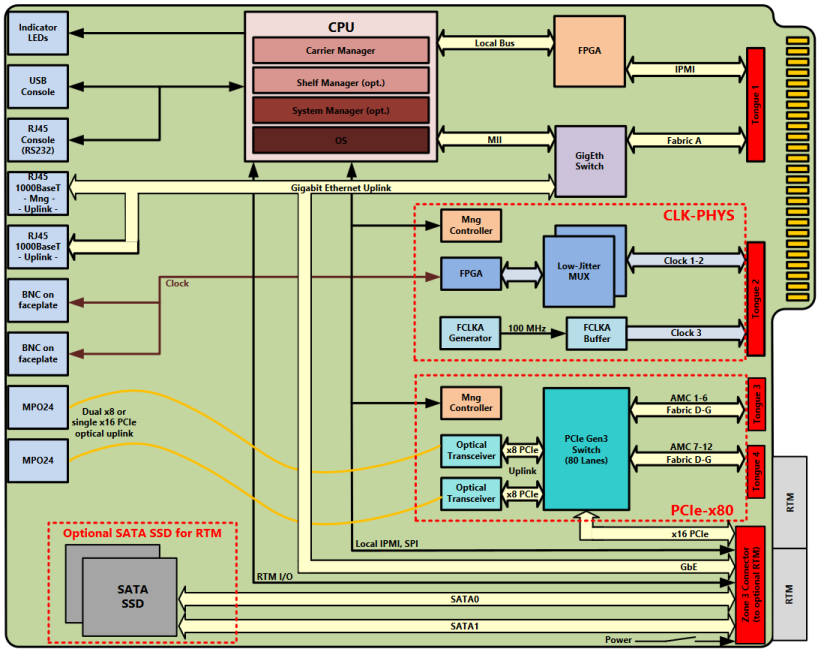

Figure 5. NAT-MCH-PHYS80 hardware block diagram [8]

2.1.3 Advanced Mezzanine Cards

AMCs are one of the key components of the MicroTCA architecture, and they are connected directly to the backplane. These devices implement modular functionality allowing to develop a wide range of applications. In addition, AMCs are hot-swappable, providing high availability, modularity, and scalability. The most common types of AMC are:

- Processor AMCs.
- Graphics AMCs.
- Storage AMCs.
- Networking AMCs.
- FPGA or SoC-based AMCs.

As shown in Figure 6, a generic AMC module has a processing element, usually an FPGA, a CPU, or a switch, accompanied by a memory bank and other memory resources. The processing device interfaces with the "outside world" through front panel connectors and communicates with the rest of the MicroTCA system using the AMC connector plugged into the backplane. Debugging interfaces such as JTAG or UART are usually available, as well as status LEDs. Also, following the standard, an MMC is present for management.

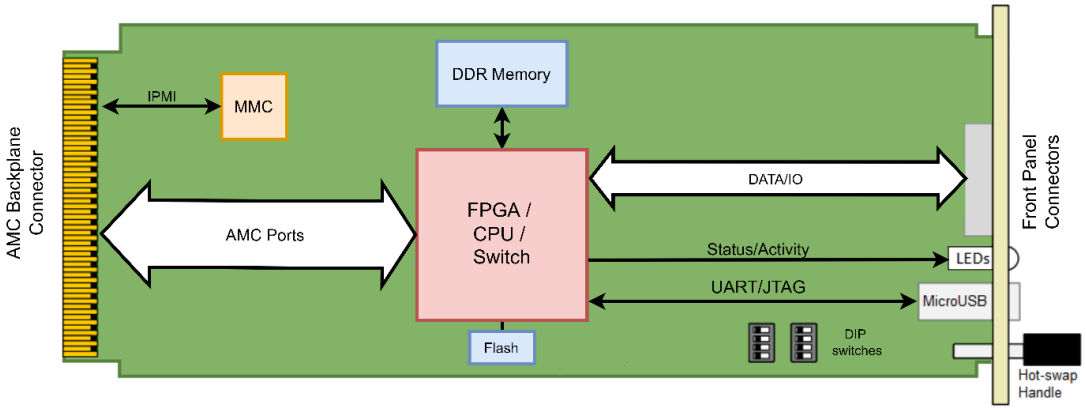

Figure 6. Generic AMC block diagram

In data acquisition applications, using FPGA-based AMCs is common due to the flexibility and scalability provided by reconfigurable hardware [9] [10]. These boards usually interface with an ADC/DAC in FMC (FPGA Mezzanine Card) form factor. Figure 7 shows an FPGA-based AMC with an FMC slot available.

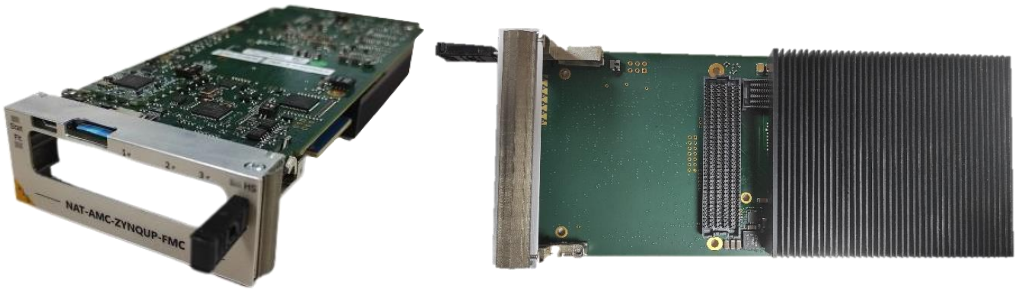

Figure 7. NAT-AMC-ZYNQUP-FMC, the image on the left shows a front view, while the one on the right shows a top-view where the FMC connector is visible

FMC is a standard defined by ANSI/VITA 57.1, which defines a standard mezzanine card form factor, connector, and modular interface to FPGAs [11]. The FMC connector has many pins for interfacing with the high pin out of FPGAs; these pins have different functions, e.g., module power supply, gigabit data and clocks, user data and clocks, I2C (Inter-Integrated Circuit) … Figure 8 shows the ADC3112, an FMC module with four ADC channels at 1 GS/s.

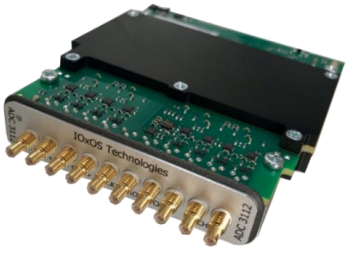

Figure 8. ADC3112, four channel 1 GS/s 12-bit ADC [12]

2.1.4 Power Modules and Cooling Units

Power modules provide regulated power for all the different components of the MicroTCA system. Up to four power modules can be in the system for increased reliability. These modules are responsible for monitoring the voltage and current levels of the system components and taking appropriate corrective actions if necessary.

Cooling units are used to dissipate this heat and maintain a safe operating temperature for the system components and may include monitoring and management features, such as temperature sensors and fan speed control.

2.2 JESD204 Converter Interface

The JESD204 standard is created by JEDEC (Joint Electron Device Engineering Council) and defines a serial interface for connecting high-speed converters and logic devices, typically FPGAs and ASICs. The motivation behind the development of the standard was to reduce the number of pins required to connect converters and processing devices in an efficient and cost-saving manner [13].

JESD204B is the second revision of the standard. In this revision, links of multiple synchronized lanes are defined, with lane rates of up to 12.5 Gbps. Figure 9 shows a multi-lane link example. This revision also defines requirements to achieve deterministic latency, ensuring the link latency is repeatable between power cycles and link resynchronization. Devices are classified into Subclass 0 if it does not support determinist latency, Subclass 1 if the device achieves deterministic latency using the SYSREF signal, and Subclass 2 if deterministic latency is achieved using the SYNC~ signal.

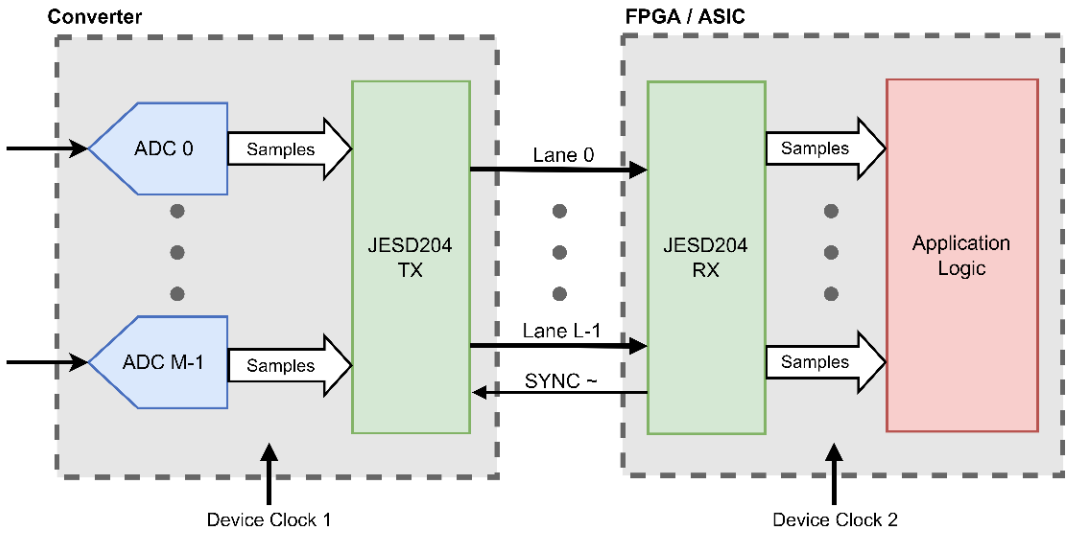

Figure 9. JESD204B single link example with L lanes and M converters.

2.2.1 Link Parameters

JESD204B links can be defined according to parameters specified by the standard. These parameters are presented in Table 2.

Table 2. JESD204B Link parameters

Parameter	Description
L	Number of lanes per link.
M	Number of converters per device.
F	Number of octets per frame.
N	Converter resolution.
CS	Number of control bits per sample.
N’	JESD204B word size. Composed of converter resolution and control bits, tail bits (T) can be added to complete the word.
K	Number of frames per multiframe.
S	Number of samples per single converter per frame cycle.
HD	High Density mode.
CF	Number of control words per frame cycle per device.

2.2.2 Functional Layers

The specification is divided into different layers based on functionality; a detailed diagram is provided in Figure 10.

- **Application Layer.** This is where the data processing logic is implemented. The functionality is application specific.
- **Transport Layer.** It is responsible for mapping/de-mapping the samples into/from the JESD204B frames. These frames can include optional control bits, which provide additional information about the transmitted data, or tail bits can be included to complete the frame. It is important to note that the transport and data link layers exchange parallel data.
- **Data Link Layer.** This layer works with the parallel framed data and applies 8b/10b encoding. Encoding the frames adds some overhead but allows clock recovery by the receiver to obtain DC balance in lane and provides built-in error checking [14]. In addition, the data can be optionally scrambled. The layer is responsible for generating and monitoring the control characters used for maintaining lane alignment; Table 3 shows the symbols used by JESD204. Also, it is responsible for synchronizing the link establishment process.
- **Physical Layer.** Consists of the circuitry that serializes/deserializes the 8b/10b encoded data. The drivers and receivers use CML (Current Mode Logic) over differential pairs, which consumes less power at higher working frequencies [2].

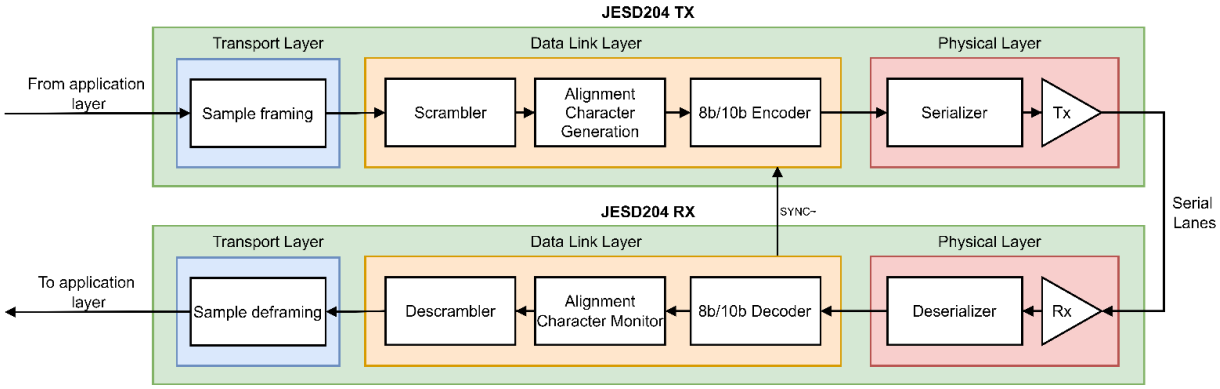

Figure 10. JESD204 functional layers

Table 3. Control Characters used in JESD204

Abbreviation	Control Symbol	Hexadecimal 8-bit Value	Function
/R/	/K28.0/	0x1C	Start of multi-frame
/A/	/K28.3/	0x7C	Lane alignment
/Q/	/K28.4/	0x9C	Start of link configuration
/K/	/K28.5/	0xBC	Code Group Synchronization
/F/	/K28.7/	0xFC	Frame alignment

2.2.3 Link Establishment

The JESD204B specification defines three phases to establish and monitor link synchronization:

- **CGS (Code Group Synchronization).**
    1. The receiver issues a synchronization request by driving the SYNC~ signal low.
    2. The transmitter sends a stream of /K/ symbols (10 bits/symbol) without scrambling.
    3. The receiver synchronizes and waits for the reception of at least consecutive correct /K/ symbols.
    4. The receiver drives the SYNC~ signal high.
- **ILAS (Initial Lane Alignment Sequence).** The ILAS consists of exactly four multiframes sent by the transmitter, which contains the link parameters it uses to communicate. This allows the receiver to align lanes, as this sequence contains /A/ control characters, and to verify the link parameters. The transmitter and receiver do not negotiate the link configuration.

- **Data phase.** Once the link is synchronized, the lines are aligned, and it is verified that both transmitter and receiver use the same configuration, the data phase begins. In this phase, lane alignment is monitored by the receiver using the control characters inserted by the transmitter.

2.3 Xilinx Application Acceleration Methodology

Xilinx presents an application acceleration development flow based on the Vitis tool, which provides a framework for developing FPGA-accelerated applications using standard programming languages [15]. An application program is divided into a software and a hardware component.

The software component, or host program, is written in C/C++ and uses the OpenCL API (Application Programming Interface). This program runs on the host processor, either an x86 server or an embedded processor.

The hardware component, or *kernel*s, are developed using C/C++, OpenCL, or RTL (Register Transfer Level) and are implemented on the PL (Programmable Logic) region of the device. Kernels can be understood as hardware-accelerated functions.

The XRT (Xilinx Run Time) manages the OpenCL API calls to process transactions between the host program and the hardware accelerator, as shown in Figure 11.

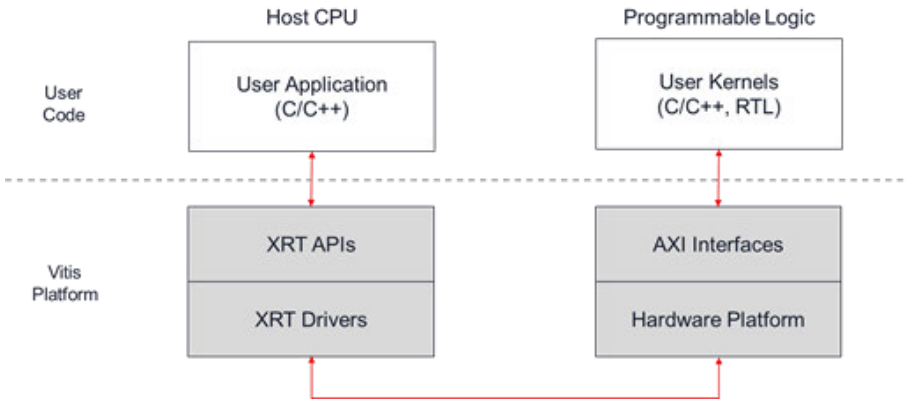

Figure 11. Vitis application structure [16]

2.3.1 Platform Types

As mentioned, the host application can run on an x86 computer or an embedded ARM processor. Xilinx distinguishes between PCIe-based and embedded platforms [17].

In the first case, the communication between the host and the hardware occurs over the PCIe bus. This solution is only available for Xilinx Alveo cards and is not open for custom board implementation. In the second case, the communication between the embedded processor and the kernels occurs across an AXI (Advanced eXtensible Interface) bus. This solution is available for Xilinx Zynq and Versal devices, which are SoCs (System on Chip) that integrate

one or more ARM processors, a programmable logic region, and hard cores such as GPUs, video codecs [18]… In this case, **users can develop their own platform for custom boards**. Figure 12 shows the architecture for each type of platform.

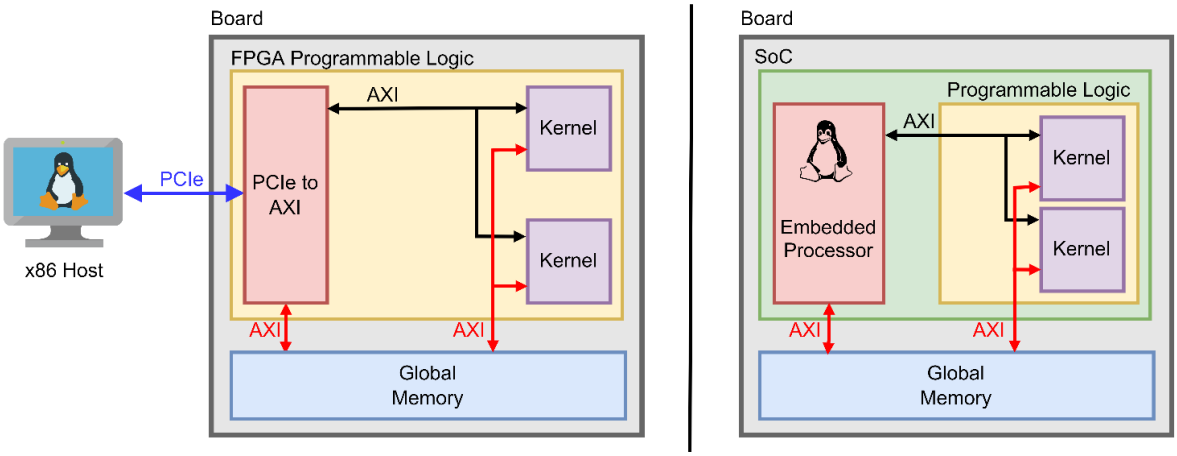

Figure 12. Xilinx platform types. On the left is a PCIe-based platform. On the right is an embedded platform. The red AXI interfaces are used for memory transfer, while the black ones are used for kernel control

2.3.2 Embedded Platform Components

An embedded platform consists of a hardware platform and a software platform. The hardware platform is the static part of the FPGA. It contains all the hardware resources needed to run acceleration kernels; these are input and output interfaces, AXI buses, interrupt controllers, clocks, and resets. This part of the project is developed in the Vivado Design Suite, allowing quick design using IP (Intellectual Property) blocks. This process outputs an XSA (Xilinx Support Archive) file containing the hardware description.

The software platform is an embedded Linux that runs on the target processor and provides an environment to run the software to control acceleration kernels. This part of the project contains:

- **Boot components.** Boot loader and system initialization executables. This includes the device-tree, which provides a way to describe non-discoverable hardware [19] to the bootloader and the Linux kernel.
- **Kernel Image.** Compiled Linux kernel.
- **Root File System.** Contains the files and directories needed for system operation, such as binaries, libraries, and modules. Contains XRT to manage kernel execution.
- **Sysroot.** Includes the libraries to cross-compile applications that run on the target platform.

The software project is configured and compiled using Petalinux, a Yocto-based tool to develop custom embedded Linux for Xilinx devices.

Xilinx provides the source code for the embedded platforms for its evaluation boards on GitHub [20], so the user can adapt it to custom boards.

2.3.3 Embedded Platform Application Acceleration Development Flow

Once the embedded platform is generated, acceleration applications can be developed using the workflow shown in Figure 13. The steps for developing such applications in Xilinx embedded systems are defined below:

- **Kernel compilation.** The kernel source code written in C/C++ or OpenCL is compiled for implementation in the programmable logic region of the device. This process generates an XO (Xilinx Object) file.
- **Hardware link.** The kernel objects are linked to the static part of the hardware platform, and a device binary file (.xclbin) is created.
- **Host program compilation.** The host application, which uses OpenCL API, is compiled using the GNU Arm cross-compiler and generates an ELF (Executable and Linkable Format) file that can be run on the target processor.

Finally, all platform files and the results of the hardware and software compilation are packaged in an image that can be written to an SD card to run the application on the target device.

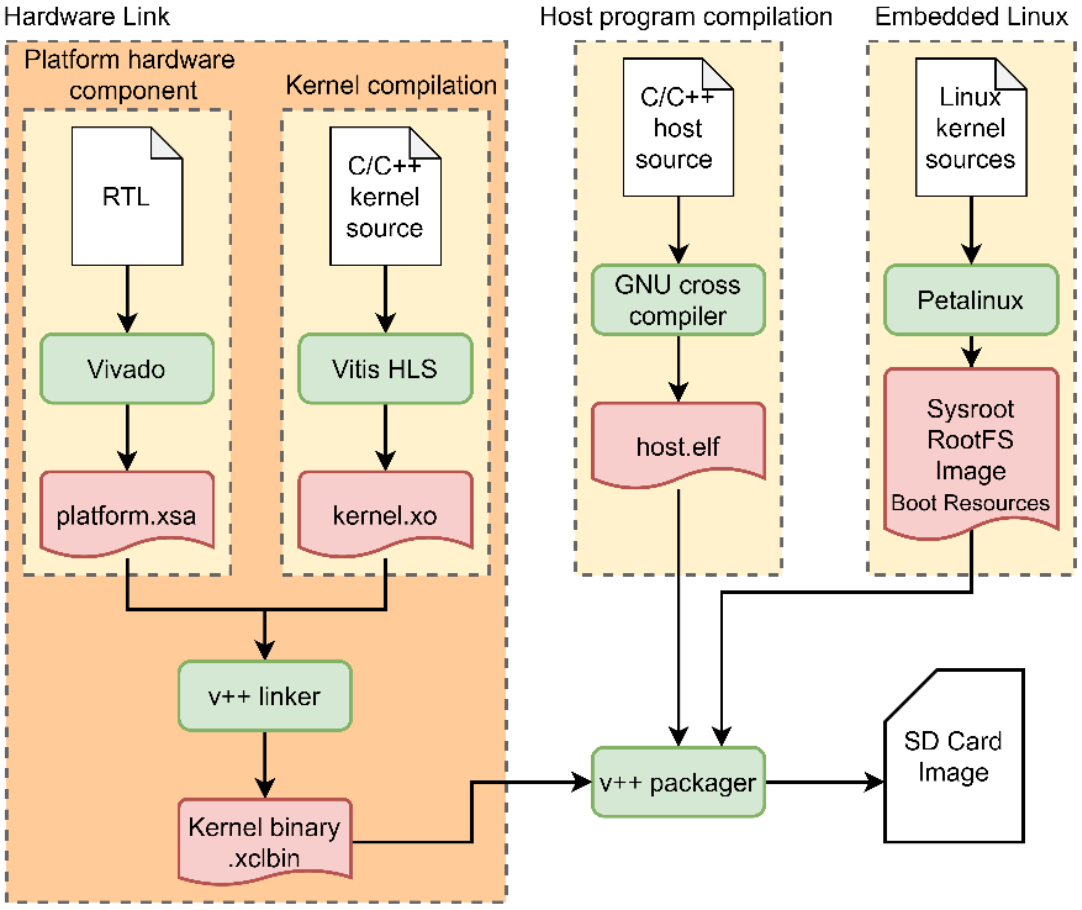

Figure 13. Application acceleration development flow for embedded platforms

2.3.4 Execution Model

The typical execution model for an accelerated application that uses the OpenCL framework is as follows:

1. The host program writes the data to be processed by a kernel into the device's global memory.
2. The host configures the kernel with input parameters.
3. The host launches the execution of the kernel on the FPGA.
4. The kernel begins processing by reading data from the global memory.
5. When the kernel has completed its task, it writes the processed data back to the global memory and notifies the host.
6. The host reads the output data from global memory and continues its processing.

Chapter 3. Design specifications and proposed solution

This project's objective is to design and develop a high-speed data acquisition system based on the JESD204B interface, which can be implemented on a given FPGA AMC module. Therefore, the following specifications and restrictions are defined for the design:

- The system must be implemented for the NAT-AMC-ZYNQUP-FMC board.
- The system must implement the JESD204B interface for transmission and reception so that a link can be established with either an ADC or a DAC.
- The JESD204B interface must be implemented through the VITA 57.1 FMC connector.
- An FMC loopback card must be used to verify the functionality of the JESD204B link.
- The system must be compatible with Xilinx's OpenCL-based framework.
- The system must connect to the MicroTCA backplane through the PCIe bus.
- For the development, version 2021.1 of Xilinx tools must be used.
- Git must be used for version control.
- There are not specific requirements defined for the JESD204B interface to implement in terms number of lanes and lane rate, but it should be flexible enough to fit current available converters on the market.

The following sections describe the implemented solution. First, the hardware setup used is explained. Then, it is explained how the embedded acceleration platform for the NAT-AMC-ZYNQUP-FMC is developed, separating the hardware and software parts, and how the available reference designs were adapted. Finally, the directory structure used to develop the project is explained.

3.1 Hardware Setup

The used hardware setup is shown in Figure 14. An x86 host computer with an NPCIe-UPLINK-O card provides an x16 PCIe optical link to the MCH in the MicroTCA chassis and, thus, to the MicroTCA backplane. In addition, a single AMC module is inserted in the chassis, the NAT-AMC-ZYNQUP-FMC.

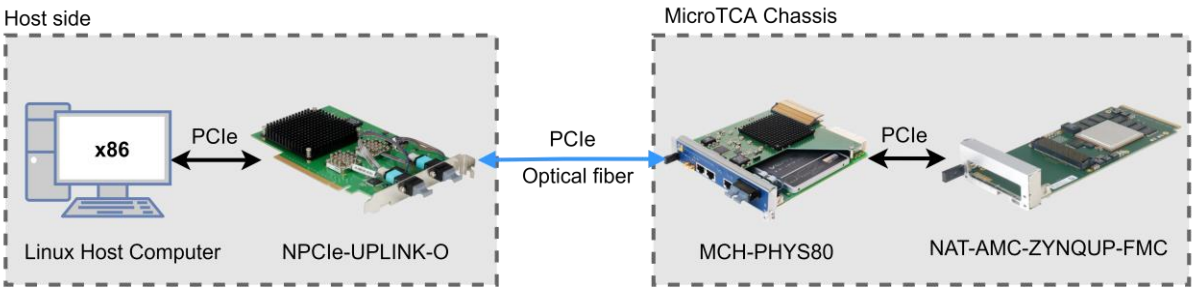

Figure 14. Hardware setup for the development of the project

3.1.1 NAT-AMC-ZYNQUP-FMC Hardware

The NAT-AMC-ZYNQUP-FMC is an FPGA-based FMC carrier AMC; its key component is a Xilinx Zynq UltraScale+ MPSoC that incorporates the following elements to provide programmable acceleration and heterogeneous processing:

- Quad-core ARM Cortex-A53 for application processing.
- Dual-core ARM Cortex-R5 for real time processing.
- ARM Mali-400MP2 GPU.
- FPGA Programmable Logic.

The UltraScale+ architecture also includes hard blocks for system management, memory interfaces, multi-gigabit transceivers, PCIe cores, GbE… An overview of the architecture is shown in Figure 15.

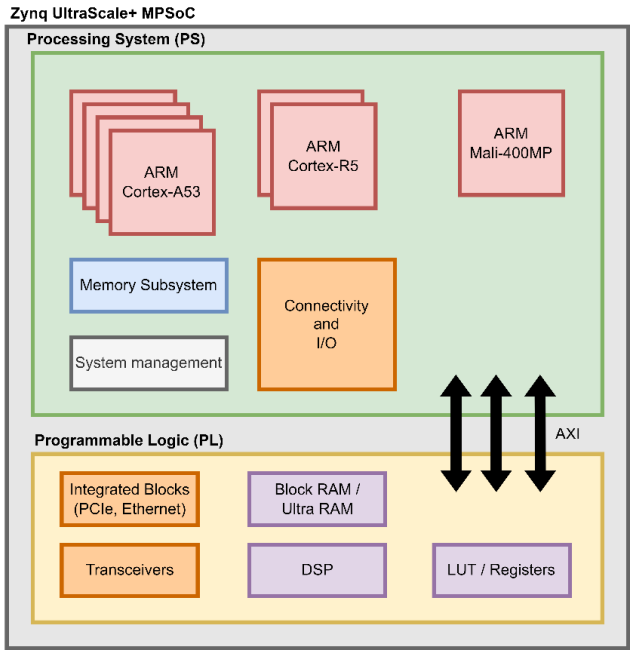

Figure 15. Zynq UltraScale+ architecture block diagram

To complement the processing capabilities of the MPSoC, the NAT-AMC-ZYNQUP-FMC includes the following elements, as shown in Figure 16:

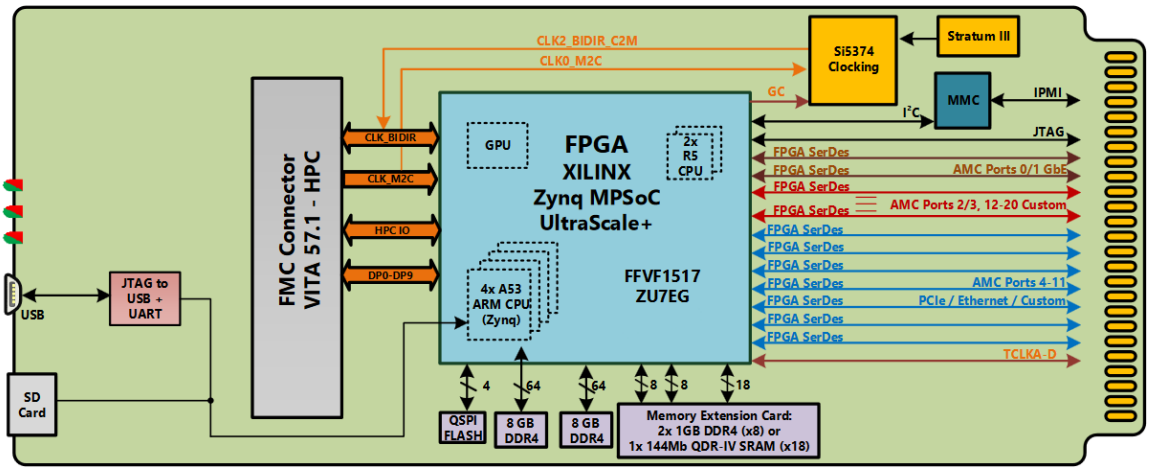

Figure 16. NAT-AMC-ZYNQUP-FMC block diagram [21]

- **Memory.** Two 8GB DDR4 (Double Data Rate) memory banks, one for the PL (Programmable Logic) and one for the ARM-based PS (Processing System).
- **Backplane Connector.** Connectivity with PCIe, GbE, clocks, and LVDS I/O.
- **Boot memory.** QSPI flash and Micro-SD card slot for system boot.
- **FPGA Programming/Debug interface.** JTAG over the front panel MicroUSB connector or through backplane connection.
- **FMC Slot.** HPC (High Pin Count) FMC slot VITA57.1 compliant, with differential pairs routed to the FPGA.
- **Clocking resources.** Si5374 PLL (Phase-Locked Loop) with 8 clock outputs.
- **Application LEDs.**

3.1.2 FMC Loopback Card

During system development, a Terasic FMC loopback card is used, which picture is shown in Figure 17.

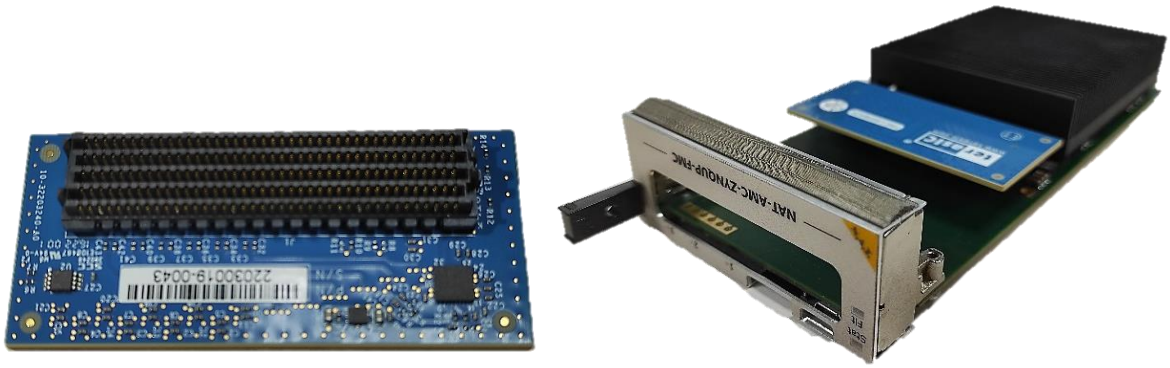

Figure 17. On the left, the Terasic FMC loopback card. On the right, the FMC module is mounted on the NAT-AMC-ZYNQUP-FMC carrier

This type of card loops the FMC pins coming out of the FPGA back into the FPGA, as shown in Figure 18, to test the functionality and transmission speed of the carrier I/O. The Terasic loopback card also provides an EEPROM that can be used to test the I2C interface and an ultra-low jitter clock multiplier that takes a clock output from the FPGA and distributes it to the remaining clock pins of the FMC connector.

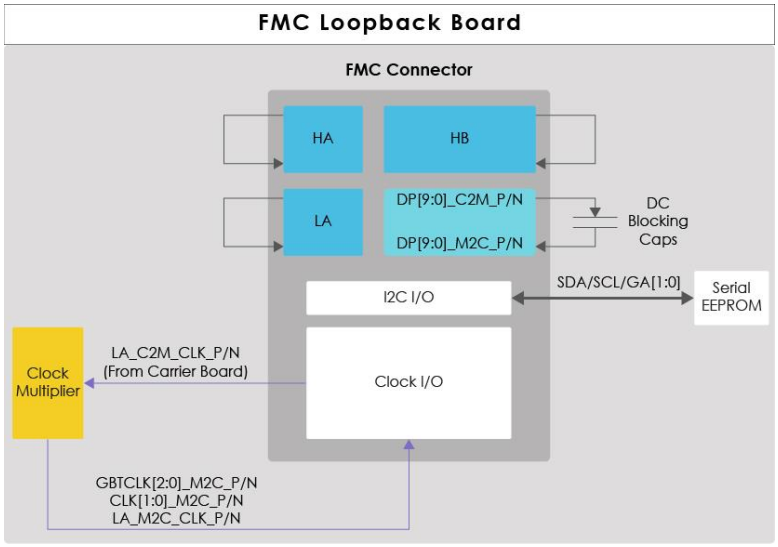

Figure 18. Terasic FMC Loopback Card Block Diagram [22]

3.2 Design of the Hardware Platform

This section describes how the hardware for the custom embedded platform is designed, starting with the AMC reference design, and adding the resources needed to run acceleration kernels and the JESD204B functionality. The hardware component of the platform is the static logic implemented on the FPGA fabric, designed using the Vivado software. The tool allows to create the design using HDL sources or by integrating pre-designed IP blocks. The use of IP reduces development time and allows reusability; these IP cores can be manufactured by Xilinx, third-party vendors, or user created.

3.2.1 NAT-AMC-ZYNQUP-FMC Reference Design

The manufacturer of the AMC, NAT, provides a reference hardware design that provides a basic infrastructure and can be used as a starting point for custom systems. Figure 19 shows the most relevant blocks of this design. These elements are:

- **Zynq UltraScale+ PS.** This block is a wrapper that instantiates the PS in the design and contains all its configuration and interconnection to the FPGA fabric. This includes:
    - Clocks and resets generated by the PS and used by the PL.
    - Memory interfaces that allow data transfers between the ARM cores and the PL.
    - Configuration of the PS DDR4 memory controller that enables interface with the on-board DDR4 8GB memory chip.

- I/O configuration, specifically this design enables an I2C master, the UART, the SD memory interface, and two embedded GEM (Gigabit Ethernet MAC).

- **PL DDR4 Controller.** There is no hard-block in the SoC to interface the PL with its dedicated DDR4 memory bank (unlike to what happens with the PS, in which the memory controller is a hard-block in the SoC), so the memory controller is instantiated as a soft-block in the FPGA fabric.

- **Ethernet Interface.** The 1G Ethernet PCS/PMA IP block, which provides the Physical Coding Sublayer and Physical Medium Attachment sublayers, interfaces the embedded GEM to the FPGA transceivers. Together, this implements two GbE interfaces.

- **PCIe Interface.** An IP block called XDMA (Xilinx Direct Memory Access) provides the PCIe interface to the backplane. This IP is based on the PCIe Gen4 hard-blocks present in the UltraScale+ architecture and implements a DMA engine that allows data movement between the PCIe address space and the AXI address space [23].

  In the reference design, the XDMA allows data to be transferred from both the PS and PL dedicated memory banks to the x86 computer over the PCIe bus. Xilinx provides a Linux kernel module that facilitates data transfers using character devices [24].

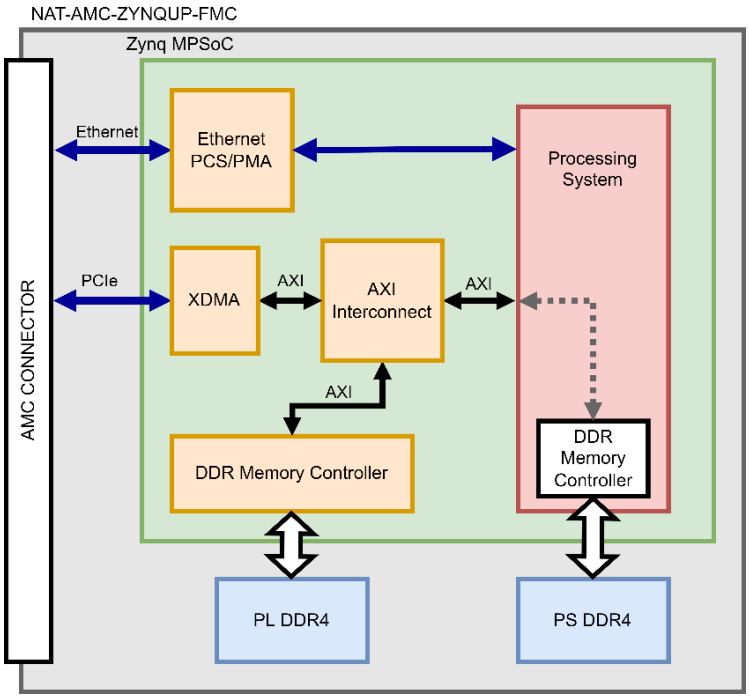

Figure 19. NAT-AMC-ZYNQUP-FMC Hardware reference design

To meet the objectives of the project, the following functionality must be added to this reference design:

- **JESD204B Interface.**

- **Support for application acceleration.** The reference design alone does not support Xilinx's application acceleration development flow because it does not provide the resources needed to run acceleration kernels.

3.2.2 JESD204B Interface

Different possibilities for implementing the JESD204 interface were considered during the initial phase of the project. The first one is to use the JESD204 IP cores developed by Xilinx that are available in the Vivado IP Catalog. Although, at first, this seems the option to go. Xilinx provides two IP cores that allow implementing a JESD204B link:

- **JESD204_PHY.** This core implements the physical layer for JESD204B with up to twelve lanes and up to 12.5 Gb/s data rate. It configures the high-speed transceivers and PLL to achieve the desired lane rate. The IP is included in the standard Vivado license.

- **JESD204.** Implements the data link layer, providing an AXI4-Lite interface for link parameter configuration and an AXI4-Streaming interface for data transfers. Requires a specific license to use. However, a trial license can be requested to allow implementation on the FPGA with time limitation.

Example designs and simulation resources are provided with the cores. However, the fact that a special license is required to implement the link and that Xilinx does not provide any software resource to control the hardware makes this option to be **discarded**.

Because of this, it has been chosen to use the Analog Devices JESD204 Interface Framework. Analog Devices supplies a complete stack that supports JESD204 and allows the development of a fully integrated system. All the Analog Devices JESD204 sources are free to use under the GPL2 license [25]. This framework provides the HDL resources to implement a JESD204B Subclass 0 or Subclass 1 interface and software to configure the hardware implemented on the FPGA and the converter devices. This section covers the integration of the HDL source to implement the JESD204B link in the NAT-AMC-ZYNQUP-FMC, while the software integration is covered in section 3.3.

The Analog Devices HDL library provides IP blocks to implement the physical, data link, and transport layers. However, for this design, the transport and application layers defined by the JESD204B specification are going to be implemented using OpenCL kernels, so only the physical layer and data link layer IP blocks are used.

Figure 20 shows the block diagram of the designed subsystem, where the red block contains the cores that implement the physical layer, and the yellow ones the data link layer. These IP cores are:

- **UTIL ADXCVR.** This physical layer peripheral interfaces and configures the GTH4 high-speed transceivers available in the Zynq MPSoC UltraScale+ architecture. It is configured to implement a transmit and receive JESD204B link with **four lanes** each. The IP uses a reference clock, typically provided by a dedicated clock chip, and generates the clock that drives the transceivers at the desired lane rate. It also generates the link reference clock for the TX/RX data link layer peripherals.

  For the TX path, the core accepts the data output from the transmit data link peripheral containing pre-processed physical layer data and performs the final 8b/10b encoding. This data is then serialized and sent through the TX serial lanes. For the RX path, the

core is responsible for de-serializing the data that arrives on the RX serial lanes as well as 8b/10b decoding.

- **AXI ADXCVR.** This core provides an AXI-Lite register interface to control and configure the transceivers instantiated in the UTIL ADXCVR core. Features a reconfiguration interface for the dynamic reconfiguration of the transceivers. Each core only controls one link, so two are instantiated, one for the RX link and one for the TX link.

- **JESD204B Transmit Peripheral.** This core implements the JESD204B data link layer transmit logic. This includes controlling the link state machine, performing scrambling and character replacement, and handling the SYSREF signal to ensure deterministic latency and the SYNC~ signal for link synchronization.

  The data to transmit to the physical layer core, which are the framed samples, are read from an AXI Stream Slave interface to which OpenCL kernels can connect to. It is worth mentioning that this AXI Stream interface does not implement the *t_valid* control flow signal, therefore it will transmit the data present at the AXI Stream port, regardless of whether this data is valid or not.

- **AXI JESD204B Transmit Peripheral.** Provides a register interface based on AXI-Lite to configure the transmit peripheral parameters and monitor link status.

- **JESD204B Receive Peripheral.** Implements the JESD204B data link layer receive logic. As well as the transmit peripheral, this includes controlling the link state machine, performing de-scrambling and character replacement, and handling the SYSREF signal to ensure deterministic latency and the SYNC~ signal for link synchronization.

  The framed samples, received from the physical layer core, are sent through an AXI Stream Master interface to which OpenCL kernels can connect to. It is worth mentioning that this AXI Stream interface does not implement the *t_ready* flow control signal, therefore it will output data regardless of whether the AXI Stream Slave is ready to receive data or not.

- **AXI JESD204B Receive Peripheral.** Provides a register interface based on AXI-Lite to configure the transmit peripheral parameters and monitor link status.

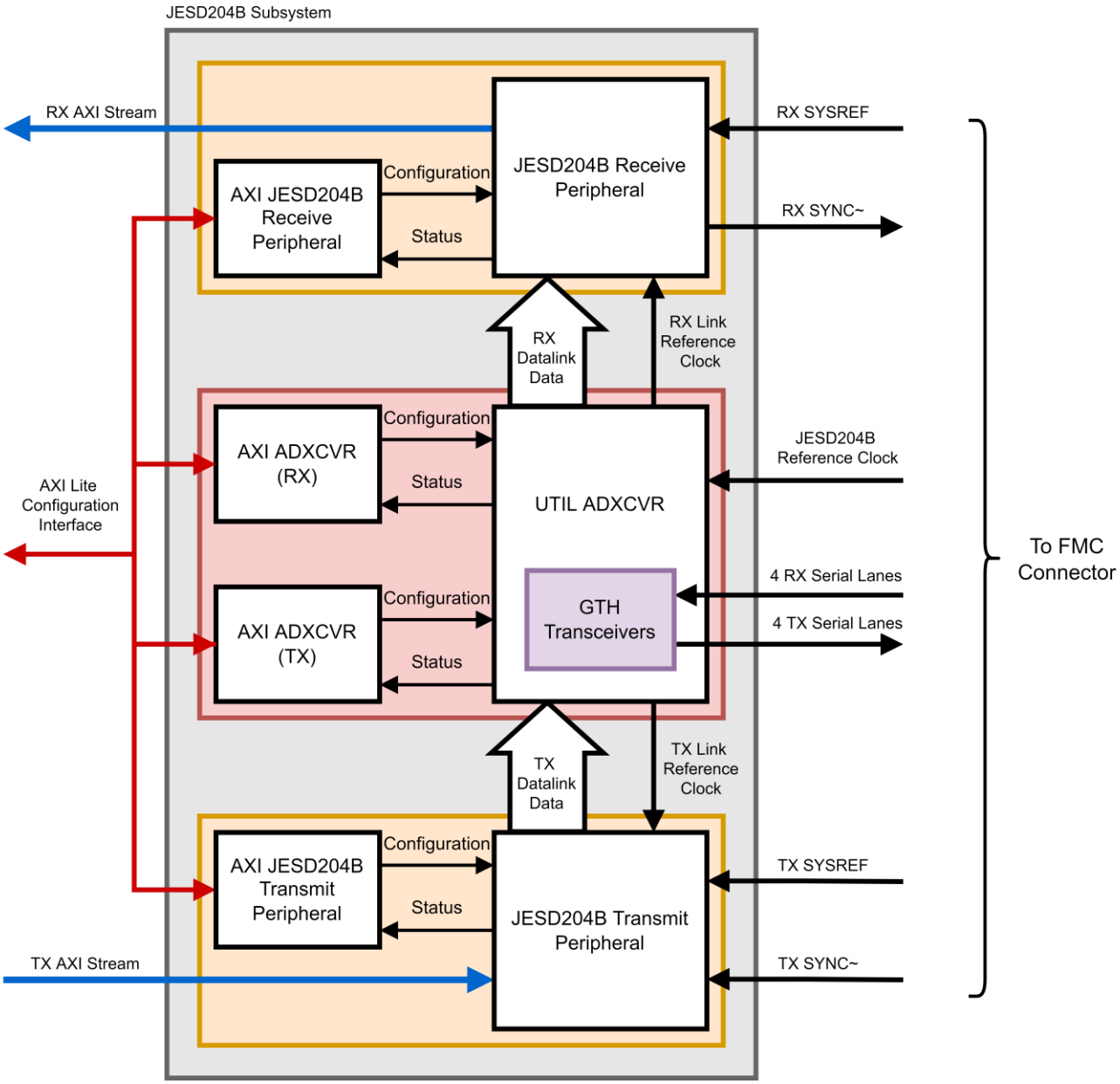

Figure 20. Implemented JESD204B Subsystem using Analog Devices IP blocks

3.2.3 Support for acceleration kernels

Vitis requires some hardware elements in the static part of an embedded platform to link the acceleration kernels. These requirements are presented in Table 4.

Table 4. Hardware interfaces required by Vitis for an acceleration platform

Type	Interface	Details
Control Interfaces	AXI Master interfaces from the PS or from AXI Interconnect IP	These are the interfaces from which the parameters are passed to the kernels. Used by the host to manage kernel execution.
Memory Interfaces	AXI Slave Interfaces	Interfaces to global memory used by kernels and the host to exchange data.
Streaming Interfaces	AXI Stream Interfaces	Kernels can use AXI Stream interfaces for data transfers. This is useful for kernels that do some part of the overall processing and do not need to transfer data to the global memory.
Clocks	-	Vitis uses the platform reference clocks to run the kernels.
Interrupts	-	Kernels can have an interrupt port to notify the host when it has finished executing.

Xilinx provides the source code of the embedded platforms for its evaluation boards [20]. These projects can be used as a reference to create custom designs, as they contain all the resources needed to run the acceleration kernels.

Based on these designs, the following resources are added to the NAT-AMC-ZYNQUP-FMC hardware platform:

- **Clocking Wizard.** The Clocking Wizard IP is used to generate clocks for different clock domains in the system using the clock resources available in the FPGA. It simplifies the clock design process by allowing the user to specify parameters such as the input frequency, output frequency, and jitter tolerance. The tool automatically configures the PLLs or MMCMs (Mixed-Mode Clock Managers) to generate the desired output frequencies.

- **Processor System Reset Module.** This IP provides resets for a processing subsystem. In this case, it is used to generate the reset signal for the acceleration kernels. The generated reset signals are synchronous to the clocks that drive the kernels.

- **AXI Interrupt Controller.** The AXI Interrupt Controller IP collects multiple interrupt signals from peripheral devices and combines them into a single interrupt output sent to the system processor. It is used to issue the kernel-generated interrupts to the PS.

In addition to these modules, the AXI interfaces for data transfer need to be exported.

- All the M_AXI interfaces available in the PS are enabled for kernel control except for M_AXI_HPM0_FPD. This interface is reserved for data transfer between the PS and the PL DDR4.

- As memory interfaces, the S_AXI interface of the PL DDR4 controller is exported, along with all the available S_AXI interfaces of the PS which provides access to its DDR4 memory. This allows the kernels to use both the PS and PL memory depending on the application requirements.

- The implemented JESD204B data link layer outputs a data stream containing the samples. This stream can be mapped to an AXI Streaming interface and exported as a platform interface so that data acquisition operations can be performed using kernels.

- In the NAT-AMC-ZYNQUP-FMC reference platform, the XDMA is configured to perform AXI memory-mapped transactions between the PCIe host and the global memory. However, when using Xilinx's acceleration model, the global memory **should not** be accessed from an external source. This is because the XRT running on the embedded processor manages the buffer allocation and memory transfers, and when reading/writing to the global memory from an external source such as XDMA, there is no defined method to know where the buffers are allocated or to ensure coherency. Therefore, the XDMA configuration is modified to perform AXI Streaming transfers that do not require global memory access. These AXI Streaming interfaces are set as platform interfaces so the kernels can connect to them and process data coming from the PCIe or send processed results to the external host.

Table 5 lists all the platform interfaces that the kernels can use. The SP Tag property can be used during application development to tell the v++ hardware linker where to connect the kernels. It does not apply to control interfaces, clocks, or interrupts.

Table 5. Custom embedded platform kernel interfaces.

Type	Source IP / signal	SP Tag	Details
Control Interfaces	Zynq UltraScale+ MPSoC/ AXI Peripheral Interconnect / M07_AXI to M14_AXI	-	Outputs from an AXI Interconnect connected to the M_AXI_HP0_LPD. Other peripherals occupy the M00_AXI to M06_AXI.

Type	Source IP / signal	SP Tag	Details
	Zynq UltraScale+ MPSoC/ M_AXI_HPM1_FPD	-	PS AXI Master output.
Memory Interfaces	Zynq UltraScale+ MPSoC/ S_AXI_HPC0_FPD	HPC0	Slave interfaces that provide access to the PS memory range. Ensures coherency using the CCI (Cache Coherent Interconnect)
	Zynq UltraScale+ MPSoC/ S_AXI_HPC1_FPD	HPC1	
	Zynq UltraScale+ MPSoC/ S_AXI_HP1_FPD	HP1	Slave interfaces that provide access to the PS memory range.
	Zynq UltraScale+ MPSoC/ S_AXI_HP2_FPD	HP2	
	Zynq UltraScale+ MPSoC/ S_AXI_HP3_FPD	HP3	
	AXI DDR4 SmartConnect/ S01_AXI to S15_AXI	MIG0	Slave interfaces that provide access to the PL DDR4 memory range. An AXI SmartConnect provides this connectivity to the DDR4 controller.
Streaming Interfaces	JESD204 Receive / rx_data	AXIS_JESD204_RX	JESD204 RX data link layer output. It is a 128-bit stream from which the mapped samples of the ADC come.
	JESD204 Transmit / tx_data	AXIS_JESD204_TX	JESD204 TX data link layer input. It is a 128-bit stream with the mapped samples to be sent to the DAC.
	XDMA / M_AXIS_H2C_0	AXIS_H2C_0	Host to card channels. 256-bit streams that contain the data sent by the external host through PCIe.
	XDMA / M_AXIS_H2C_1	AXIS_H2C_1	
	XDMA / M_AXIS_H2C_2	AXIS_H2C_2	

Type	Source IP / signal	SP Tag	Details
	XDMA / M_AXIS_H2C_3	AXIS_H2C_3	
	XDMA / S_AXIS_C2H_0	AXIS_C2H_0	Card to host channels. 256-bit streams that contain the data sent to the external host through PCIe.
	XDMA / S_AXIS_C2H_1	AXIS_C2H_1	
	XDMA / S_AXIS_C2H_2	AXIS_C2H_2	
	XDMA / S_AXIS_C2H_3	AXIS_C2H_3	
Clocks	Clocking Wizard/ clock_out1	-	100 MHz clock.
	Clocking Wizard / clock_out2	-	200 MHz. This clock is selected as default for kernels.
	Clocking Wizard / clock_out3	-	300 MHz clock.
	Clocking Wizard / clock_out4	-	400 MHz clock.
Interrupts	AXI Interrupt Controller / intr	-	Interrupt interface for the AXI Interrupt Controller.

3.2.4 Final Hardware Design

After these steps, the platform contains the hardware required to meet the project specifications. The designed system block diagram is shown in Figure 21.

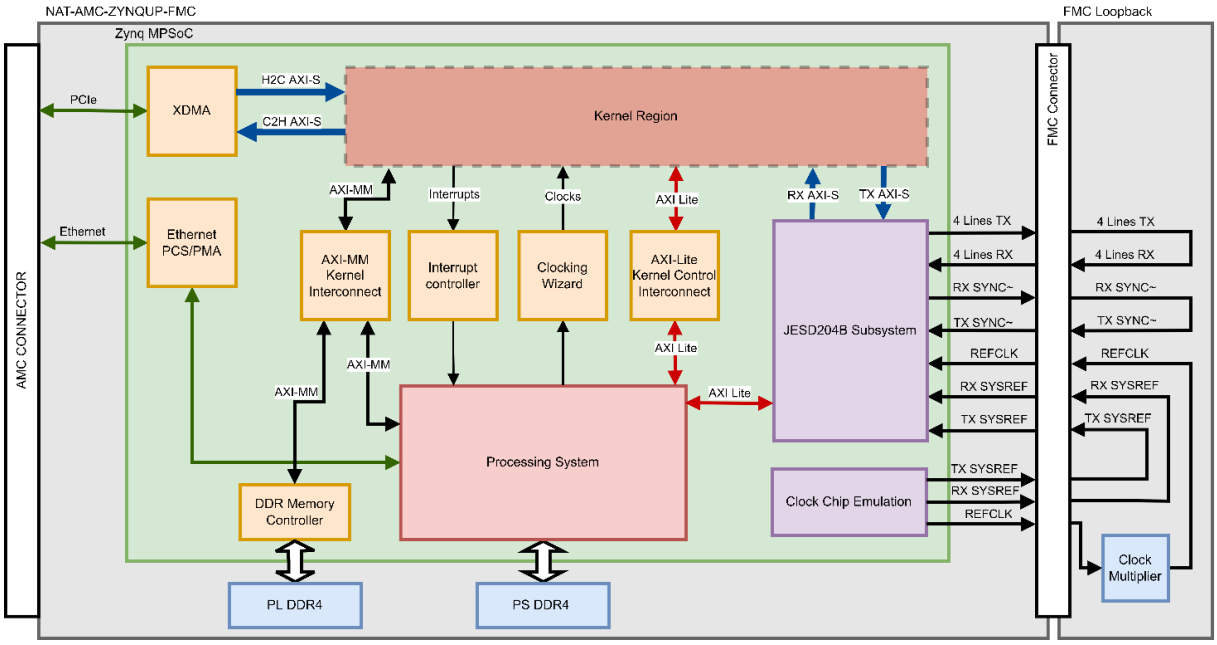

Figure 21. Designed system hardware block diagram

There are some things to pay attention to. The first one is the use of the Terasic FMC loopback card to loop the JESD204B transmit signals with the receiver signals; specifically, the four serial lanes are looped, as well as the SYNC~ signals. This allows the transmitter and receiver to establish a link with each other to verify the correct operation of the implemented JESD204B subsystem.

The second thing to note is the Clock Chip Emulation block. Typically, FMCs using JESD204B links have a dedicated clock chip that provides the reference clock for both the converter and the logic device and the SYSREF signals if the link supports deterministic latency. However, since there is no dedicated clock chip in the design, an FPGA PLL is used to generate a 200 MHz reference clock, and basic logic implements the generation of the SYSREF signals synchronous to the reference clock. Note that the Ultra-Low Jitter Clock Multiplier distributes the reference clock.

Finally, it is worth mentioning the standard that the signals outside the FPGA use. Except for the JESD204B serial lanes, which use the OIF-CEI-11G standard [26], the signals looped through the FMC use the LVDS standard.

3.3 Design of the Software Platform

Once the hardware platform is designed, the next step is to prepare the software platform. This section describes the resources that are part of the Petalinux project created to generate an embedded Linux that supports the designed hardware. First, the base NAT-AMC-ZYNQUP-FMC Petalinux project is described. Second, the Analog Devices sources are added to this project, and the device-tree is modified to include the required nodes for the JESD204B interface. Finally, the software resources required to run OpenCL kernels are included.

In this section, the term *kernel* refers to two different things. When referring to the Linux core, the term *Linux kernel* is used, while when referring to OpenCL-based accelerated functions, only the term *kernel* is used.

3.3.1 NAT-AMC-ZYNQUP-FMC Reference Petalinux Project

NAT provides a reference Petalinux project that generates an embedded Linux to support the reference hardware project. Petalinux uses recipes to configure and compile the project, and these recipes are organized into layers. Layers are ordered by priority from generic layers to more specialized ones.

The reference project adds the **meta-nat-amc-zynqup-fmc** layer on top of the default layers used by Petalinux. This layer contains the recipes, patches, and device-tree for the embedded Linux to use all the AMC board hardware correctly. However, it does not contain the files to support user-added logic in the PL region of the device, as this is application specific. To add support for specific applications, the top layer of the project is a user-defined layer called **meta-user**, which allows for custom configuration of the Linux distribution, including adding new software packages and modifying the configuration of existing components. Users can add nodes to the device-tree by adding them to the *system-user.dtsi* file included in this layer. Figure 22 shows the layers of the reference project, where some common and chip support layers are omitted for simplicity.

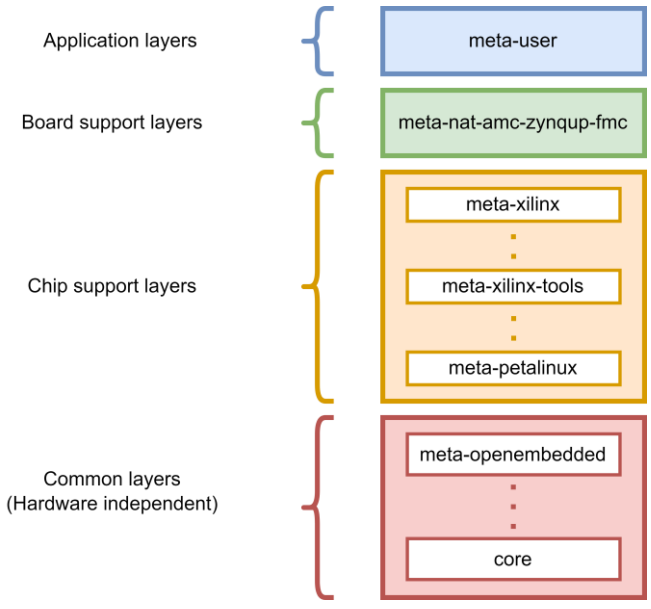

Figure 22. NAT-AMC-ZYNQUP-FMC reference Petalinux project layers (simplified)

3.3.2 JESD204B Software Support

Analog Devices provides software support for their JESD204B cores. Linux device drivers are provided to handle the high-speed transceivers using the AXI_ADXCVR IP, as well as drivers for the transmit and receive cores. All these resources are included in the **meta-adi-core** and **meta-adi-xilinx** layers which are available on the Analog Devices GitHub [27]. To add the resources to the project, these layers are included below the **meta-nat-amc-zynqup-fmc** layer, as shown in Figure 23.

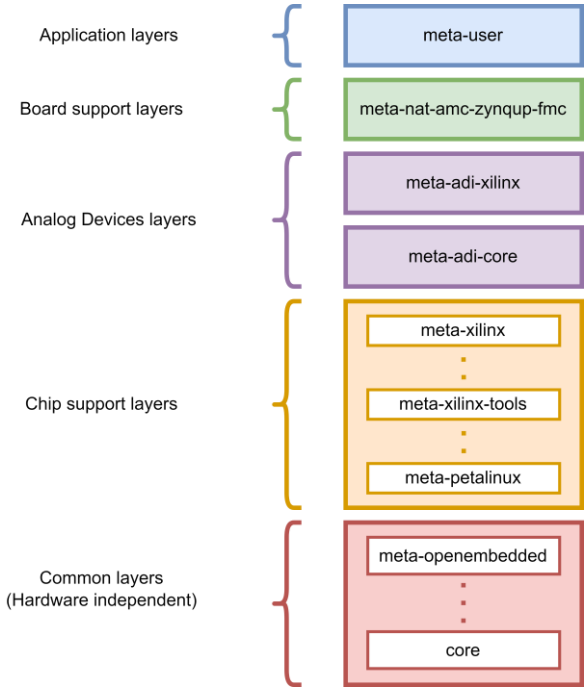

Figure 23. Petalinux project with Analog Devices layers

This configuration changes the Linux kernel that the built distribution uses; instead of using the Xilinx Linux kernel, it will use the Analog Devices Linux kernel, which includes all the drivers. These drivers need to be enabled in the Linux kernel configuration, so they are compiled later into the final kernel image. The required drivers are presented in Table 6.

Table 6. JESD204B Linux device drivers

Driver Name	Description	Kernel Dependencies
jesd204	JESD204 High-Speed Serial Interface Framework. Provides a unified model for supporting the initialization and operation of JESD204 transceivers, which aims at reducing complexity when managing these types of devices.	CONFIG_OF
axi-adxcvr	Support for Analog Devices AXI ADXCVR PHY. Reads the configuration from the corresponding device-tree node	CONFIG_IIO CONFIG_JESD204 CONFIG_COMMON_CLK

Driver Name	Description	Kernel Dependencies
	and applies it to the AXI ADXCVR peripheral.	
axi-jesd204-rx	Analog Devices AXI JESD204B TX Support. This driver reads the link configuration from the device-tree and configures the RX data link layer peripheral.	CONFIG_IIO CONFIG_JESD204 CONFIG_COMMON_CLK
axi-jesd204-tx	Analog Devices AXI JESD204B RX Support. This driver reads the link configuration from the device-tree and configures the TX data link layer peripheral.	CONFIG_IIO CONFIG_JESD204 CONFIG_COMMON_CLK
jesd204-top-device	JESD204 Top Device. The jesd204-top-device provides the link parameters for all other link devices. Used when a converter is not part of the topology.	CONFIG_JESD204

The JESD204B Interface Linux Kernel Framework is an FSM (Finite State Machine) that synchronizes all the Linux device drivers required to properly bring-up and manage JESD204B links [28]. The framework allows defining a topology of devices, ensuring that all devices pass through all the FSM states simultaneously. These topologies of devices are described to the framework through nodes in the device-tree, with a single top-level device that typically is the converter. Device-tree nodes are added to the *system-user.dtsi* file in the meta-user layer, where all the nodes for the PL implemented peripherals should be defined.

For the hardware implemented in section 3.2, there is a receiver and a transmitter which, in the final system, will establish a link with an ADC/DAC present on the FMC. However, to test the functionality of the JESD204B subsystem, the transmitter and receiver need to establish the link through the loopback card. Therefore, there is no converter and no top device. To use the JESD204B Linux Kernel Framework with this setup, the *jesd204-top-device* driver is used to provide the link parameters.

Figure 24 shows the implemented topology, where two JESD204B links are defined. Note that although two links are defined to the framework, only one link is being established in the implemented hardware. In addition, it is important to note that the connections in the figure do not represent data movement but rather relationships between devices that are required during link bring-up. A detailed description of the device-tree implementation is provided in Appendix II.

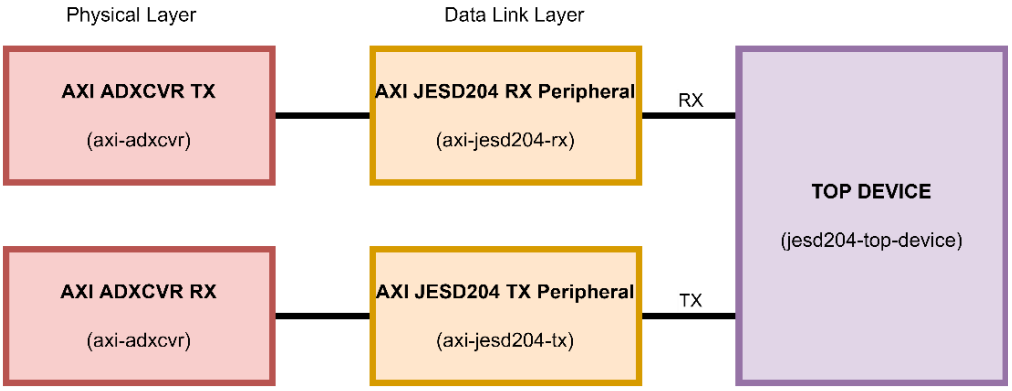

Figure 24. JESD204 Linux Kernel Framework implemented topology

3.3.3 Xilinx Run Time

For embedded platforms, the XRT performs memory management for OpenCL accelerated applications using the *zocl* Linux kernel module. This module supports both topologies implemented in the hardware, shared memory between the PL and PS, and dedicated memory in the PL using a soft memory controller. *zocl* supports CMA (Contiguous Memory Allocator) based memory management, where kernels in the PL region use physical addresses.

By default, XRT is not enabled in Petalinux projects, but the recipe is available in the **meta-xilinx** layer. To include the package in the Linux RFS (Root File System), the CONFIG_xrt option is enabled in the project configuration. Note that a device-tree node is required for the zocl driver to manage interrupt connection. This node is automatically created by Petalinux and can be revised in the *<project-directory>/components/plnx_workspace/device-tree/device-tree/pl.dtsi* file.

Finally, it is necessary to set the size for the CMA which is used to exchange data between the PS and PL kernels. This is added to the Linux kernel boot arguments with the parameter *cma=512M*. The size for the CMA is determined by the kernel requirements.

3.4 Project Structure

To facilitate version control using Git, the following directory structure has been used during the project's development. The system compilation process is automated using *Makefiles* and TCL scripts, which allows the system to be easily compiled from zero in almost any Linux environment.

```
├── hw
│   ├── build
│   ├── scripts
│   └── source
│
├─ platform
│
├─ platform_repo
│
└── sw
    ├── app_x86
    ├── petalinux
    ├── i2a2_sw
    └── ocl_sw
```

The content of each directory is as follows:

- The *hw* directory contains the source files related to the hardware design.
    - The *source* directory contains the HDL sources and IP blocks used in the design.
    - The *scripts* directory contains the TCL scripts that automate the hardware design building process.
    - The output files are found under the *build* directory, where the Vivado project is located.
- The *sw* directory contains the source files related to the software design and example applications.
    - *app_x86* contains the XDMA Linux kernel module to communicate with the FPGA using PCIe.
    - The *petalinux* directory contains the Petalinux project sources and the Makefile that automates the Linux compilation process.
    - The *i2a2_sw* directory contains the C/C++ source for example applications that use PCIe communication.
    - The *ocl_sw* directory contains the C/C++ source for OpenCL example applications.
- The *platform* directory contains the TCL script and files required to package both the hardware and software platform into the Vitis Embedded Acceleration Platform that the end user will use to develop applications. The platform is packaged using the XSCT (Xilinx Software Command-line Tool), which requires a TCL script defining the configuration. A reference script is available on the Xilinx Vitis Embedded Platform Source GitHub [20].
- The packaged platform and output files are stored in the *platform_repo* directory.

Chapter 4. Results

This chapter presents the results obtained with the designed system. First, the GitHub repository containing the project's source files is presented. Second, the operation of the implemented JESD204B subsystem and its integration with the Linux drivers is verified. Third, the PCIe interface is tested to verify that data can be sent and received from the external host PC. Then, some sample OpenCL applications are developed to test the platform's ability to run acceleration kernels. Finally, the FPGA resource utilization is analyzed.

4.1 GitHub repository

For this project, GitHub has been used as the hosting platform for version control during development. The repository of the project can be found in [29]. It is currently private; however, access can be granted on request. This repository contains a set of *readme* files that describe the system requirements for recreating the project, general information about the provided sources, and instructions for compiling and modifying the project. The building process is completely automated based on *Makefiles,* simplifying project recreation and deployment.

Once the building has finished, the Vitis Embedded Acceleration platform is installed in */opt/xilinx/platforms*, which is where the Vitis tool looks for installed platforms. The structure of the installation directory is as follows:

```
/opt/xilinx/platforms/namc_zynqup_jesd204_fmc/
├── hw
│   └── namc_zynqup_jesd204_fmc.xsa
├── namc_zynqup_jesd204_fmc.xpfm
├── sw
│   ├── namc_zynqup_jesd204_fmc
│   │   ├── boot
│   │   ├── qemu
│   │   └── xrt
│   └── namc_zynqup_jesd204_fmc.spfm
└── sysroot
```

- The *hw/* directory contains the *namc_zynqup_jesd204_fmc.xsa* file with the hardware design.
- The *sw/* directory contains the *namc_zynqup_jesd204_fmc.spfm* file which is the software platform metadata file. Under this directory are the boot sources in the *sw/boot/* directory, the Linux emulation files under sw/*qemu/* and the Linux kernel image as well as RootFS are located under *sw/xrt/*.
- The *sysroot/* directory contains the cross-compilation tools.
- The *namc_zynqup_jesd204_fmc.xpfm* contains the embedded platform metadata. The *platforminfo* utility can be used from the command line to get the platform properties:

```
i2a2@MTCA-Ubuntu:~$ platforminfo namc_zynqup_jesd204_fmc.xpfm
==========================
Basic Platform Information
==========================
Platform:           namc_zynqup_jesd204_fmc
File:
/opt/xilinx/platforms/nat_amc_zynqup_jesd204_fmc/namc_zynqup_jesd204_fmc.xpfm
Description:
A basic platform targeting the NAT AMC ZYNQUP FMC


====================================
Hardware Platform (Shell) Information
====================================
Vendor:                           i2a2
Board:                            namc_zynqup_fmc
Name:                             namc_zynqup_fmc
Version:                          1.0
Generated Version:                2021.1
Hardware:                         1
Software Emulation:               1
Hardware Emulation:               1
Hardware Emulation Platform:      0
FPGA Family:                      zynquplus
FPGA Device:                      xczu7eg

=================
Clock Information
=================
  Default Clock Index: 2
  Clock Index:         1
    Frequency:         100.000000
  Clock Index:         2
    Frequency:         200.000000
  Clock Index:         3
    Frequency:         300.000000
  Clock Index:         4
    Frequency:         400.000000

==================
Memory Information
==================
  Bus SP Tag: HP1
  Bus SP Tag: HP2
  Bus SP Tag: HP3
  Bus SP Tag: HPC0
  Bus SP Tag: HPC1
  Bus SP Tag: MIG0

============================
Software Platform Information
============================
Number of Runtimes:            1
Default System Configuration:  namc_zynqup_jesd204_fmc
System Configurations:
  System Config Name:                      namc_zynqup_jesd204_fmc
  System Config Description:               namc_zynqup_jesd204_fmc
  System Config Default Processor Group:   xrt
  System Config Default Boot Image:        standard
  System Config Is QEMU Supported:         1
  System Config Processor Groups:
    Processor Group Name:      xrt
    Processor Group CPU Type:  cortex-a53
    Processor Group OS Name:   linux
  System Config Boot Images:
    Boot Image Name:           standard
    Boot Image Type:
```

```
    Boot Image BIF:            namc_zynqup_jesd204_fmc/boot/linux.bif
    Boot Image Data:           namc_zynqup_jesd204_fmc/xrt/image
    Boot Image Boot Mode:      sd
    Boot Image RootFileSystem:
    Boot Image Mount Path:     /mnt
    Boot Image Read Me:        namc_zynqup_jesd204_fmc/boot/generic.readme
    Boot Image QEMU Args:
namc_zynqup_jesd204_fmc/qemu/pmu_args.txt:namc_zynqup_jesd204_fmc/qemu/qemu_args.txt
    Boot Image QEMU Boot:
    Boot Image QEMU Dev Tree:
Supported Runtimes:
  Runtime: OpenCL
```

4.2 JESD204B Interface Results

4.2.1 Linux Driver Initialization

The first thing to check is that the embedded Linux can parse the device-tree, bind the correct driver to the devices and start the JESD204 FSM. This is done at boot time, and if it is performed correctly, all the drivers that are part of the JESD204B interface must synchronize, and the link must be established. Once the embedded Linux has booted, the boot messages can be viewed using the *dmesg* command. What can be observed in the boot log is explained as follows.

First, the JESD204 framework recognizes the topology described by the nodes and properties in the device-tree and begins the initialization.

```
[3.194153] jesd204: created con: id=0, topo=0, link=1, /fpga-axi@0/axi_jesd204_rx_0@80050000 <->
/fpga-axi@0/fake-adc-dac
[3.204848] jesd204: created con: id=1, topo=0, link=0, /fpga-axi@0/axi_jesd204_tx_0@80060000 <->
/fpga-axi@0/fake-adc-dac
[3.215833] jesd204: created con: id=2, topo=0, link=0, /fpga-axi@0/axi_adxcvr_tx@80040000 <->
/fpga-axi@0/axi_jesd204_tx_0@80060000
[3.227684] jesd204: created con: id=3, topo=0, link=1, /fpga-axi@0/axi_adxcvr_rx@80030000 <->
/fpga-axi@0/axi_jesd204_rx_0@80050000
[3.239537] jesd204: /fpga-axi@0/fake-adc-dac: JESD204[0:0] transition uninitialized ->initialized
[3.248524] jesd204: /fpga-axi@0/fake-adc-dac: JESD204[0:1] transition uninitialized ->initialized
[3.257517] jesd204: found 5 devices and 1 topologies
```

Following this, the driver for each device is probed.

```
[3.942459] axi_adxcvr 80030000.axi_adxcvr_rx: AXI-ADXCVR-RX (17.05.a) using QPLL1 on GTH4 at
0x80030000. Number of lanes: 4.
[3.953004] axi_adxcvr 80040000.axi_adxcvr_tx: AXI-ADXCVR-TX (17.05.a) using QPLL1 on GTH4 at
0x80040000. Number of lanes: 4.
[3.963830] axi-jesd204-rx 80050000.axi_jesd204_rx_0: AXI-JESD204-RX (1.07.a) at 0x80050000. Encoder
8b10b, width 4/4, lanes 4, jesd204-fsm.
[3.976302] axi-jesd204-tx 80060000.axi_jesd204_tx_0: AXI-JESD204-TX (1.06.a) at 0x80060000. Encoder
8b10b, width 4/4, lanes 4, jesd204-fsm.
[3.989494] jesd204_top_device fpga-axi@0:fake-adc-dac: JESD204-GENERIC-TOP-DEVICE probed 2 links
```

Once the framework has ensured that the driver for each device part of the topology is running, the FSM starts. It is interesting to observe how the framework makes the two defined links go through the different states together. In the background, the device drivers involved in the JESD204B operation pass through the FSM together, making sure that everything is set up correctly.

```
[3.997339] jesd204: /fpga-axi@0/fake-adc-dac,jesd204:4,parent=fpga-axi@0:fake-adc-dac: JESD204[0:0]
transition initialized -> probed
[4.009269] jesd204: /fpga-axi@0/fake-adc-dac,jesd204:4,parent=fpga-axi@0:fake-adc-dac: JESD204[0:1]
transition initialized -> probed
[4.021208] jesd204: /fpga-axi@0/fake-adc-dac,jesd204:4,parent=fpga-axi@0:fake-adc-dac: JESD204[0:0]
transition probed -> idle
[4.032541] jesd204: /fpga-axi@0/fake-adc-dac,jesd204:4,parent=fpga-axi@0:fake-adc-dac: JESD204[0:1]
transition probed -> idle
[4.043872] jesd204: /fpga-axi@0/fake-adc-dac,jesd204:4,parent=fpga-axi@0:fake-adc-dac: JESD204[0:0]
transition idle -> device_init
[4.055632] jesd204: /fpga-axi@0/fake-adc-dac,jesd204:4,parent=fpga-axi@0:fake-adc-dac: JESD204[0:1]
transition idle -> device_init
[4.067400] jesd204: /fpga-axi@0/fake-adc-dac,jesd204:4,parent=fpga-axi@0:fake-adc-dac: JESD204[0:0]
transition device_init -> link_init
[4.079593] jesd204: /fpga-axi@0/fake-adc-dac,jesd204:4,parent=fpga-axi@0:fake-adc-dac: JESD204[0:1]
transition device_init -> link_init
[4.091793] jesd204: /fpga-axi@0/fake-adc-dac,jesd204:4,parent=fpga-axi@0:fake-adc-dac: JESD204[0:0]
transition link_init -> link_supported
[4.104245] jesd204: /fpga-axi@0/fake-adc-dac,jesd204:4,parent=fpga-axi@0:fake-adc-dac: JESD204[0:1]
transition link_init -> link_supported
[4.116741] jesd204: /fpga-axi@0/fake-adc-dac,jesd204:4,parent=fpga-axi@0:fake-adc-dac: JESD204[0:0]
transition link_supported -> link_pre_setup
[4.129590] jesd204: /fpga-axi@0/fake-adc-dac,jesd204:4,parent=fpga-axi@0:fake-adc-dac: JESD204[0:1]
transition link_supported -> link_pre_setup
[4.142480] jesd204: /fpga-axi@0/fake-adc-dac,jesd204:4,parent=fpga-axi@0:fake-adc-dac: JESD204[0:0]
transition link_pre_setup -> clk_sync_stage1
[4.155457] jesd204: /fpga-axi@0/fake-adc-dac,jesd204:4,parent=fpga-axi@0:fake-adc-dac: JESD204[0:1]
transition link_pre_setup -> clk_sync_stage1
[4.168431] jesd204: /fpga-axi@0/fake-adc-dac,jesd204:4,parent=fpga-axi@0:fake-adc-dac: JESD204[0:0]
transition clk_sync_stage1 -> clk_sync_stage2
[4.181489] jesd204: /fpga-axi@0/fake-adc-dac,jesd204:4,parent=fpga-axi@0:fake-adc-dac: JESD204[0:1]
transition clk_sync_stage1 -> clk_sync_stage2
[4.194553] jesd204: /fpga-axi@0/fake-adc-dac,jesd204:4,parent=fpga-axi@0:fake-adc-dac: JESD204[0:0]
transition clk_sync_stage2 -> clk_sync_stage3
[4.207613] jesd204: /fpga-axi@0/fake-adc-dac,jesd204:4,parent=fpga-axi@0:fake-adc-dac: JESD204[0:1]
transition clk_sync_stage2 -> clk_sync_stage3
[4.222692] jesd204: /fpga-axi@0/fake-adc-dac,jesd204:4,parent=fpga-axi@0:fake-adc-dac: JESD204[0:0]
transition clk_sync_stage3 -> link_setup
[4.233303] jesd204: /fpga-axi@0/fake-adc-dac,jesd204:4,parent=fpga-axi@0:fake-adc-dac: JESD204[0:1]
transition clk_sync_stage3 -> link_setup
[4.245934] jesd204: /fpga-axi@0/fake-adc-dac,jesd204:4,parent=fpga-axi@0:fake-adc-dac: JESD204[0:0]
transition link_setup -> opt_setup_stage1
[4.258647] jesd204: /fpga-axi@0/fake-adc-dac,jesd204:4,parent=fpga-axi@0:fake-adc-dac: JESD204[0:1]
transition link_setup -> opt_setup_stage1
[4.271365] jesd204: /fpga-axi@0/fake-adc-dac,jesd204:4,parent=fpga-axi@0:fake-adc-dac: JESD204[0:0]
transition opt_setup_stage1 -> opt_setup_stage2
[4.284601] jesd204: /fpga-axi@0/fake-adc-dac,jesd204:4,parent=fpga-axi@0:fake-adc-dac: JESD204[0:1]
transition opt_setup_stage1 -> opt_setup_stage2
[4.297835] jesd204: /fpga-axi@0/fake-adc-dac,jesd204:4,parent=fpga-axi@0:fake-adc-dac: JESD204[0:0]
transition opt_setup_stage2 -> opt_setup_stage3
[4.311067] jesd204: /fpga-axi@0/fake-adc-dac,jesd204:4,parent=fpga-axi@0:fake-adc-dac: JESD204[0:1]
transition opt_setup_stage2 -> opt_setup_stage3
[4.324303] jesd204: /fpga-axi@0/fake-adc-dac,jesd204:4,parent=fpga-axi@0:fake-adc-dac: JESD204[0:0]
transition opt_setup_stage3 -> opt_setup_stage4
[4.337536] jesd204: /fpga-axi@0/fake-adc-dac,jesd204:4,parent=fpga-axi@0:fake-adc-dac: JESD204[0:1]
transition opt_setup_stage3 -> opt_setup_stage4
[4.405267] jesd204: /fpga-axi@0/fake-adc-dac,jesd204:4,parent=fpga-axi@0:fake-adc-dac: JESD204[0:0]
transition clocks_enable -> link_enable
[4.415732] jesd204: /fpga-axi@0/fake-adc-dac,jesd204:4,parent=fpga-axi@0:fake-adc-dac: JESD204[0:1]
transition clocks_enable -> link_enable
[4.449291] jesd204: /fpga-axi@0/fake-adc-dac,jesd204:4,parent=fpga-axi@0:fake-adc-dac: JESD204[0:0]
transition link_enable -> link_running
[4.456146] jesd204: /fpga-axi@0/fake-adc-dac,jesd204:4,parent=fpga-axi@0:fake-adc-dac: JESD204[0:1]
transition link_enable -> link_running
[4.468606] jesd204: /fpga-axi@0/fake-adc-dac,jesd204:4,parent=fpga-axi@0:fake-adc-dac: JESD204[0:0]
transition link_running -> opt_post_running_stage
[4.482009] jesd204: /fpga-axi@0/fake-adc-dac,jesd204:4,parent=fpga-axi@0:fake-adc-dac: JESD204[0:1]
transition link_running -> opt_post_running_stage
```

The function of each state is:

- **LINK_INIT.** For each link, the *struct jesd204_link* is filled with link parameters defined by the top-level device.
- **LINK_SUPPORTED.** Checks that all the devices that are part of the link support the link configuration defined by the top-level device.
- **LINK_PRE_SETUP & CLK_SYNC_STAGE.** Clock chip configuration and synchronization stages.
- **LINK_SETUP.** All devices that are part of the link are configured based on the previously validated configuration. In the implemented hardware, all peripherals are configured using the AXI interface. OPT_SETUP_STAGEn allows having additional steps for more complex link setups.
- **CLOCKS_ENABLE & LINK_ENABLE.** The data link layer peripherals are taken out of reset, and the link is enabled.
- **LINK_RUNNING.** All links in the topology are running. Optional OPT_POST_RUNNING_STAGE states can be implemented if more configuration is required.

4.2.2 JESD204B Status Utility

The Analog Devices Linux resources include the *jesd_status* application. This program reads the status registers of the data link layer peripherals from Linux sysfs [30] and displays the link and lane status information. Figure 25 and Figure 26 show the output from the application. Here is a breakdown of the information displayed:

- At the top, the found data link layer peripherals are listed. The application has found the TX and RX cores instantiated in the FPGA hardware design.
- The STATUS section provides information related to the JESD204B Link, including:
    - "Link is enabled" shows that the link is running.
    - "Link status" shows the link state. The program reports that the link has gone through synchronization and has successfully reached the DATA phase.
    - "Measured/Reported Link Clock" show respectively the link clock frequency measured by the hardware core and the link clock frequency reported by the Linux Common Clock framework.
    - "Measured/Reported/Desired Device Clock" show respectively the device clock frequency measured by the hardware core, the device clock frequency reported by the Linux Common Clock framework, and the required device clock rate defined by the lane rate and data path width. The required device clock of 200 MHz is achieved.

  - "Lane Rate" shows the lane rate. The desired 8 Gbps throughput is achieved.
  - "LMFC Rate" shows the LMFC (Local MultiFrame Clock) frequency. The K and F link parameters define this frequency.
  - "SYSREF captured" and "SYSREF alignment error" show that the SYSREF signal has been captured and no alignment errors are detected.
  - "SYNC~" reports the status of the synchronization request signal.

- The LANE STATUS section shows per-lane status. Only the RX core reports this information.
  - "Lane#" shows the number of lanes of the link. The four lanes defined in the hardware are being used in the link.
  - "Errors" shows the total number of errors detected. The reported errors include unexpected /K/ characters, not in table 8b/10b symbols detected, and disparity errors. No errors are detected after the link has been running for a significant amount of time.
  - "Latency" reports the lane latency as total multiframes and octets. This latency is measured from when the SYNC~ signal is de-asserted until the lane achieves frame synchronization. It is interesting to observe that despite the fact that not all lanes have the same latency, the interface is able to synchronize and provide a reliable data-transfer method.
  - "CGS State" shows the current state of the lane code group synchronization. DATA means that the lane is synchronized and ready to receive data. Note that this is a per-lane state; for the link to enter the DATA phase and start transmitting data, all lanes must have achieved synchronization.
  - "Initial Frame Sync" indicates that the lane has completed frame synchronization.
  - "Initial Lane Alignment Sequence" reports that the ILAS configuration data has been correctly received.

After monitoring and checking the information reported by the program, it is safe to state that the implemented link is reliable and can transmit data accurately and consistently.

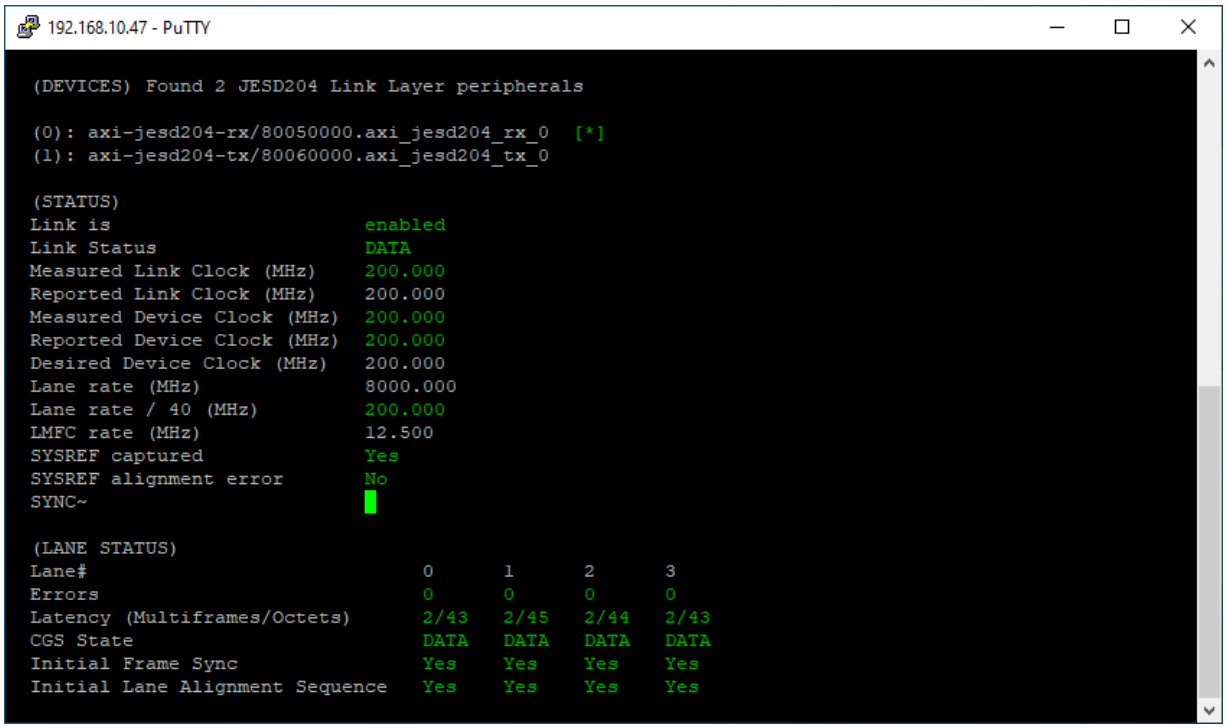

Figure 25. jesd_status output for the RX data link core

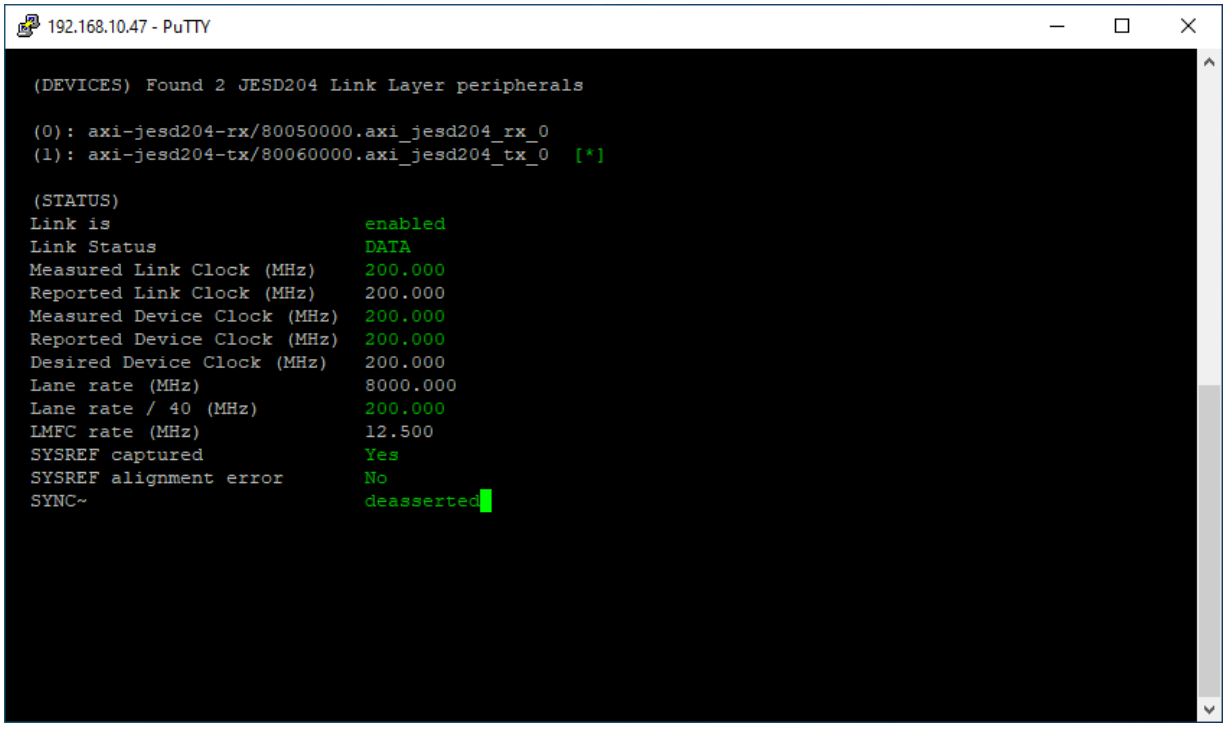

Figure 26. jesd_status command output for the TX data link core

4.2.3 Hardware Analysis

The Xilinx ILA (Integrated Logic Analyzer) is a powerful debugging tool commonly used in digital system design. The ILA allows designers to observe and analyze internal signals, providing valuable insight into hardware behavior. The ILA is available in the Vivado IP Catalog and can be instantiated in any hardware design. Using an ILA on the implemented system can provide interesting information about how the JESD204B interface works. Therefore, an ILA has been instantiated to monitor the most relevant signals from the implemented hardware, and its results are shown below.

The CGS process can be observed by setting a trigger on the rising edge of the SYNC~ signal. As shown in Figure 27, when the data link layer cores come out from reset, the RX core drives the SYNC~ signal low to request synchronization. In response to this, the TX core sends a stream of /K/ characters (0xBC in hexadecimal) through the four lanes, which in Figure 27 are labeled *tx_datalink_<n>*. Once the receiver has achieved synchronization, the SYNC~ signal is de-asserted (driven high). Then, the transmitter waits for the next LMFC boundary to send the ILAS.

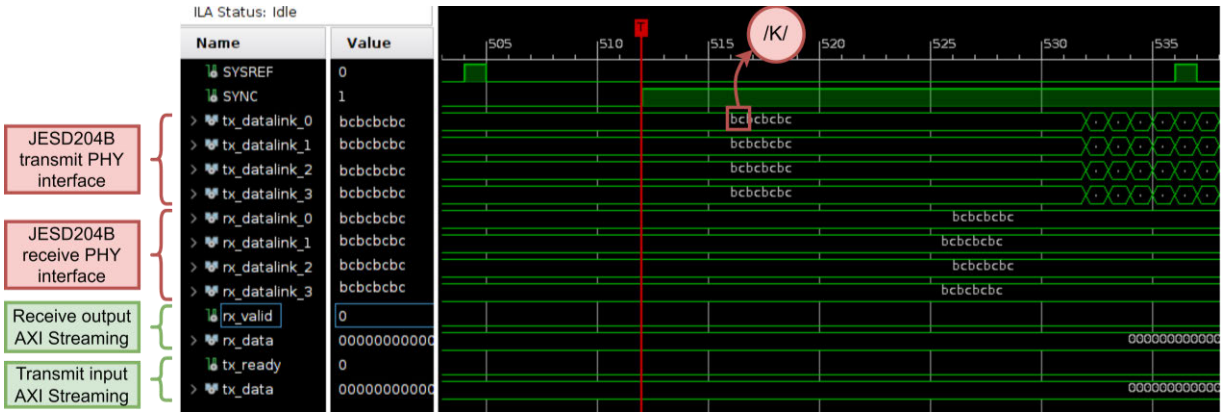

Figure 27.JESD204B CGS phase observed in ILA

The ILAS consists of four multiframes sent by the transmitter. The structure of each multiframes is as follows:

Multiframe 1. It starts with an /R/ character (0x1C in hexadecimal) and ends with an /A/ character (0x7C in hexadecimal). The rest of the multiframe consists of ramp data. Figure 28 shows the transition from CGS to ILAS.

Multiframe 2. It starts with an /R/ character followed by a /Q/ character (0x9C), and then 14 configuration octets are sent. Finally, the multiframe ends with an /A/ character. Figure 29 shows the end of the first multiframe and the beginning of the second multiframe, along with the configuration octets.

Multiframe 3. It starts with an /R/ character and ends with an /A/ character. The rest of the multiframe consists of ramp data.

Multiframe 4. It starts with an /R/ character and ends with an /A/ character. The rest of the multiframe consists of ramp data.

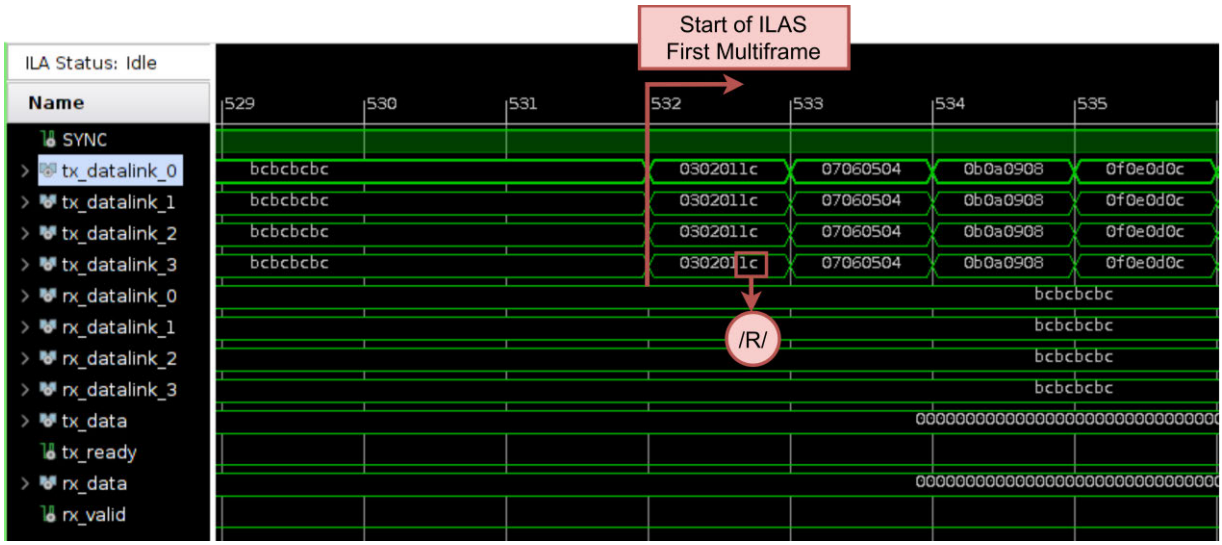

Figure 28. First multiframe of the initial line alignment sequence observed in ILA

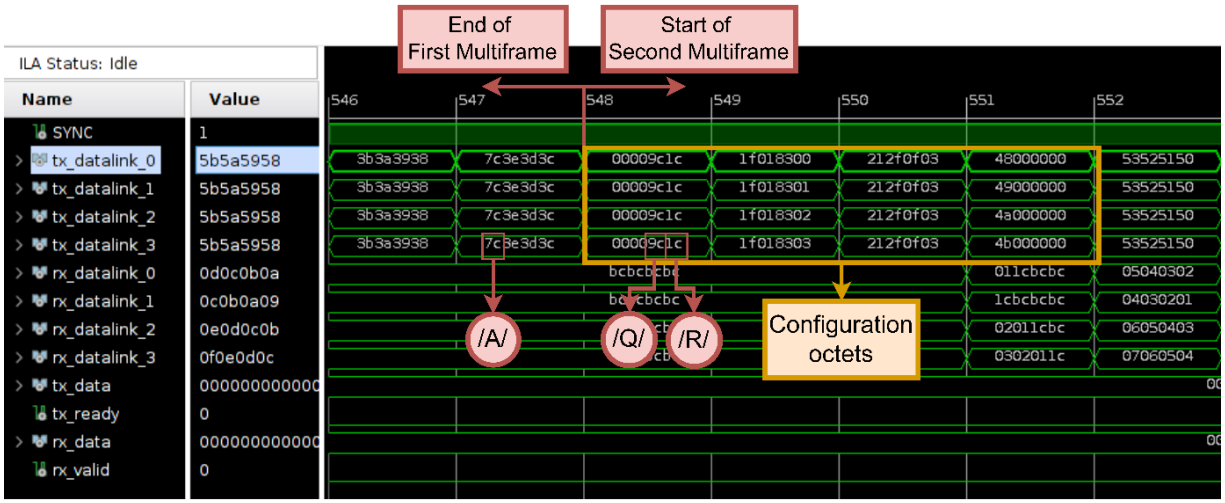

Figure 29. End of the first multiframe and the start of the configuration octets observed in ILA

After the last multiframe of the ILAS is transmitted, the link enters the DATA phase. In this phase, the transmitter drives the *tx_ready* signal high, indicating that the data stream provided on *tx_data* will be transmitted to the link. On the other side of the link, when the receiver receives the last ILAS multiframe, it drives the *rx_valid* signal high, indicating that the data stream coming from *rx_data* is valid. This means that the data sent through *tx_data* will arrive at *rx_data* after passing through the entire link. The transition from the ILAS phase to the DATA phase is shown in Figure 30.

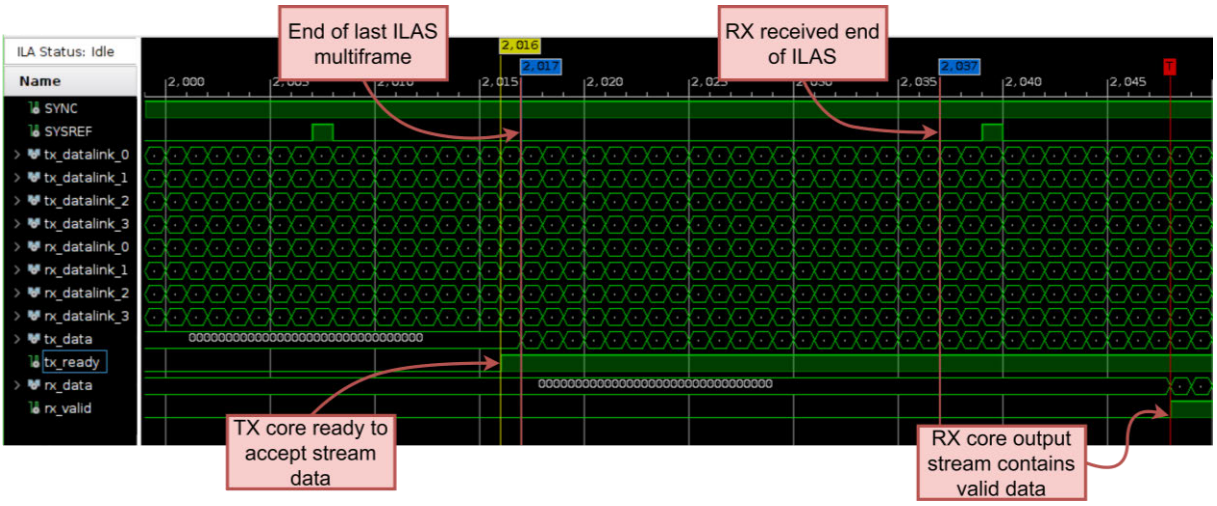

Figure 30. Transition from ILAS phase to DATA phase observed in ILA

To check the data transfer in the DATA phase, a ramp is generated using a binary counter and sent on each line through *tx_data*. The data must arrive through *rx_data*. Figure 31 shows the result obtained using the ILA; the received data exactly data matches the transmitted data. It is important to note that this test **does not consider** the mapping of the samples that must keep the data in *tx_data* and *rx_data* in the assumed system using four converters is not being considered since this functionality is implemented by the transport layer, and in this system, its implementation is planned for OpenCL kernels. The purpose of this test is to verify that the data sent by the transmitter arrives correctly at the receiver.

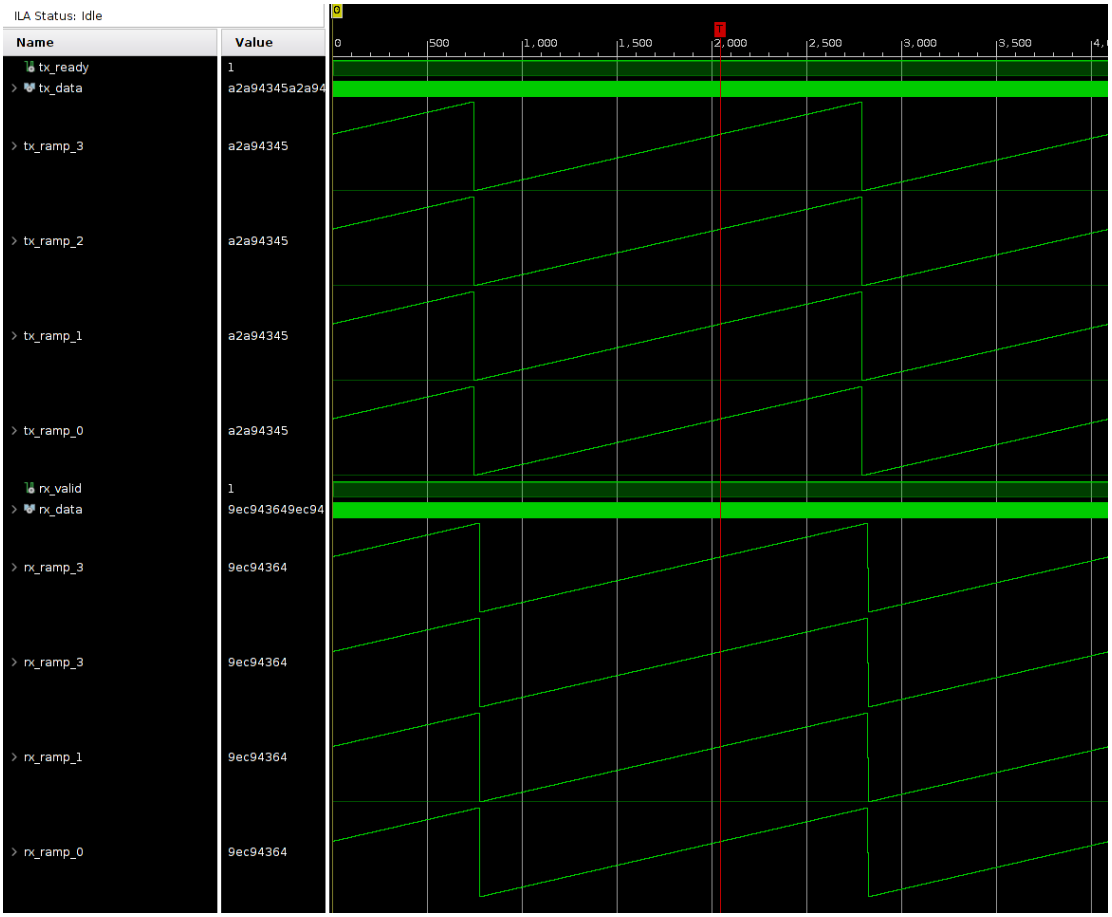

Figure 31. Ramp data sent and received through JESD204B observed in ILA

4.3 PCIe Interface Results

4.3.1 XDMA Linux Driver

The PCIe interface is implemented using the XDMA IP in the endpoint AMC module. In addition, Xilinx provides a Linux kernel module for the host computer that facilitates data transfers by using character devices. The kernel module is available on GitHub [24]. This allows the development of user space applications that communicate with the FPGA, as shown in Figure 32.

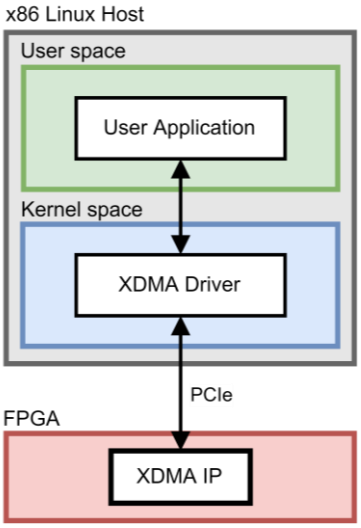

Figure 32. XDMA driver overview

The host computer scans the PCIe bus at boot time to detect any attached devices. Then, it uses the information stored in the endpoint PCIe configuration space to select the appropriate device driver to use based on the device vendor ID, device ID, and class code. The host computer correctly identifies the AMC as a Xilinx device and therefore loads the *xdma* driver; this can be observed in the boot log:

```
i2a2@MTCA-Ubuntu:~$ dmesg | grep xdma
[2.879155] xdma: loading out-of-tree module taints kernel.
[2.892416] xdma: module verification failed: signature and/or required key missing -
tainting kernel
[2.893640] xdma:xdma_mod_init: Xilinx XDMA Reference Driver xdma v2020.2.2
[2.893641] xdma:xdma_mod_init: desc_blen_max: 0xffffff/268435455, timeout: h2c 10 c2h
10 sec.
[2.893684] xdma:xdma_device_open: xdma device 0000:09:00.0, 0x00000000c715ea5d.
[2.893802] xdma:map_single_bar: BAR0 at 0xdc800000 mapped at 0x000000005e81cbb7,
length=8388608(/8388608)
[2.893810] xdma:map_single_bar: BAR2 at 0xdd000000 mapped at 0x00000000b7ea66b7,
length=65536(/65536)
[2.893814] xdma:map_bars: config bar 2, pos 1.
[2.893814] xdma:identify_bars: 2 BARs: config 1, user 0, bypass -1.
[2.894022] xdma:pci_keep_intx_enabled: 0000:09:00.0: clear INTX_DISABLE, 0x406 -> 0x6.
[2.894042] xdma:probe_one: 0000:09:00.0 xdma0, pdev 0x00000000c715ea5d, xdev
0x0000000029f33aa6, 0x00000000160fc5a0, usr 16, ch 4,4.
[2.894543] xdma:cdev_xvc_init: xcdev 0x00000000a98930fa, bar 0, offset 0x40000.
```

The *lspci* utility provides deeper information about the PCIe device, such as the physical slot used, available memory resources, and the device's capabilities. This also confirms that the *xdma* driver is used to manage the device.

```
i2a2@MTCA-Ubuntu:~$ sudo lspci -s 09:00.0 -v
09:00.0 Serial controller: Xilinx Corporation Device 9038 (prog-if 01 [16450])
        Subsystem: Xilinx Corporation Device 0007
        Physical Slot: 5
        Flags: bus master, fast devsel, latency 0, IRQ 148
        Memory at dc800000 (32-bit, non-prefetchable) [size=8M]
        Memory at dd000000 (64-bit, non-prefetchable) [size=64K]
        Capabilities: [40] Power Management version 3
        Capabilities: [48] MSI: Enable+ Count=1/1 Maskable- 64bit+
        Capabilities: [70] Express Endpoint, MSI 00
        Capabilities: [100] Advanced Error Reporting
        Capabilities: [1c0] Secondary PCI Express
        Kernel driver in use: xdma
        Kernel modules: xdma
```

The driver creates the character devices in the */dev* directory. These character devices can be treated as files, and standard system calls such as read() and write() can be used to interact with the FPGA. The functionality of each character device is as follows:

- **xdma0_c2h_<n>.** This character device provides access to the C2H (Card to Host) DMA channel number <n> of the Xilinx XDMA IP core. It can be used to transfer data from the FPGA to the host system memory.
- **xdma0_h2c_<n>.** This character device provides access to the H2C (Host to Card) DMA channel number <n> of the Xilinx XDMA IP core. It can be used to transfer data from the host system memory to the FPGA.
- **xdma0_control.** Gives access to the XDMA IP configuration registers.
- **xdma0_user.** Gives access to the AXI-Lite interface of the XDMA IP.
- **xdma0_events_<n>.** Used to recognize interrupts coming from the FPGA.
- **xdma0_xvc.** Xilinx Virtual Cable.

```
i2a2@MTCA-Ubuntu:~$ ls /dev/xdma*
/dev/xdma0_c2h_0      /dev/xdma0_events_12  /dev/xdma0_events_7
/dev/xdma0_c2h_1      /dev/xdma0_events_13  /dev/xdma0_events_8
/dev/xdma0_c2h_2      /dev/xdma0_events_14  /dev/xdma0_events_9
/dev/xdma0_c2h_3      /dev/xdma0_events_15  /dev/xdma0_h2c_0
/dev/xdma0_control    /dev/xdma0_events_2   /dev/xdma0_h2c_1
/dev/xdma0_events_0   /dev/xdma0_events_3   /dev/xdma0_h2c_2
/dev/xdma0_events_1   /dev/xdma0_events_4   /dev/xdma0_h2c_3
/dev/xdma0_events_10  /dev/xdma0_events_5   /dev/xdma0_user
/dev/xdma0_events_11  /dev/xdma0_events_6   /dev/xdma0_xvc
```

4.3.2 XDMA Test Application

To test DMA operation, the C2H and H2C channels are looped as shown in Figure 33 so that the data sent from the host to the FPGA can be read back to verify communication in both directions.

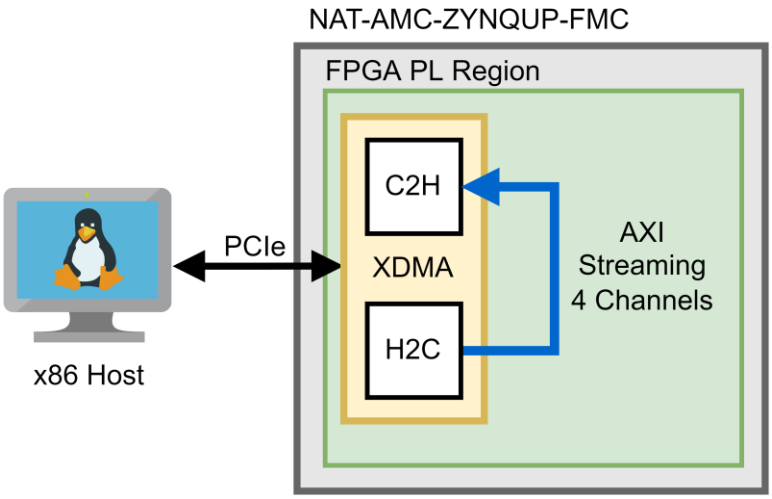

Figure 33. Hardware setup for XDMA test application

The *AXIS_xdma.py* script has been developed to run on the host computer and performs the following operations to test the communication:

1. Open *xdma_c2h* and *xdma_h2c* files.
2. Create an output buffer in host memory and fill it with data.
3. Set up the reading process on the *xdma_c2h* file. The process waits until data arrives at the C2H channel.
4. Start the write process of the output buffer to the *xdma_h2c* file. This writes the data through the H2C channel. Then, this data is looped to the C2H channel.
5. Wait for the read operation on the *xdma_c2h* file to complete; the input data is stored in a buffer in host memory.
6. Close *xdma_c2h* and *xdma_h2c* files.
7. Compare output data and input data.

```
i2a2@MTCA-Ubuntu:~/Documents/scripts $ sudo python3 AXIS_xdma.py 0
Opened file /dev/xdma0_c2h_0
Set up read process
Opened file /dev/xdma0_h2c_0
Set up write process
Wrote 1024 bytes (1024 should have been written) to the H2C channel. Took 0.00003
seconds, at 33.0323 MB/s
Read 1024 bytes from the C2H channel. Took 0.0071 seconds, at 0.13718 MB/s
All processes have completed
Total time: 0.0080919 seconds
Input and output data match! Test succesful
```

The test is run on all available channels and is **successful** because the input data matches the output data in all cases. Therefore, it can be concluded that using the XDMA IP in streaming mode provides a high-performance and reliable method for transferring data blocks between the FPGA and the host computer, relieving CPU use.

4.4 OpenCL Applications

This section demonstrates that the system can run OpenCL-based applications with two sample applications, a vector addition, and a data acquisition example.

4.4.1 zocl Linux Kernel Module

The zocl Linux kernel module should be running to run OpenCL accelerated applications. This can be checked in the boot messages and by running the *lsmod* command.

```
[5.778604] zocl: loading out-of-tree module taints kernel.
[5.786388] [drm] Probing for xlnx,zocl
[5.790641] zocl-drm amba_pl@0:zyxclmm_drm: IRQ index 32 not found
[5.796979] [drm] FPGA programming device pcap founded.
[5.802222] [drm] PR Isolation addr 0x0
[5.803565] [drm] Initialized zocl 0.0.0 00000 for amba_pl@0:zyxclmm_drm on minor 0

-------------------------------------------------------------------------------

root@namc_zynqup_jesd204_fmc:/media/sd-mmcblk0p1# lsmod
Module                  Size  Used by
zocl                  147456  0
uio_pdrv_genirq        16384  0
```

The module is loaded, and the system is ready to run hardware-accelerated applications.

4.4.2 Vector Addition

Vector addition is one of the simplest acceleration kernels, commonly used as the “Hello World!” of application acceleration. The *vadd* kernel, which is written in C/C++, reads the input vectors and adds them to the output vector. The operations performed by the host program are:

1. Initialize the runtime environment. The host detects the attached FPGA device, loads the device binary (.xclbin) into it, and creates a kernel object.
2. Create buffers in global memory to share data with the kernel. Three buffers are defined, two for the kernel input vectors and one for the output.
3. Map the buffers to host memory. The kernel input buffers are filled with test data.
4. Set the kernel argument and trigger its execution.
5. The kernel output data is retrieved from global memory.
6. Validate the results.

```
root@namc_zynqup_jesd204_fmc:/media/sd-mmcblk0p1# ./vadd binary_container_1.xclbin
INFO: Reading binary_container_1.xclbin
Loading: 'binary_container_1.xclbin'
TEST PASSED
```

The device successfully passes the test, verifying that the system can use the OpenCL API to run accelerated applications.

4.4.3 Data Acquisition

The purpose of this application is to implement a single-event data acquisition system. This requires two kernels, one to provide data to the JESD204B transmit peripheral and the other to capture the data provided by the JESD204B receive peripheral. This emulates the transmit kernel sending samples to a DAC and the acquire kernel receiving samples from an ADC.

The transmit kernel takes the samples to transmit from the device's global memory and sends them through the AXI streaming interface available in the JESD204B TX core. The receive kernel acquires the samples from the AXI Streaming interface of the RX core and stores them in the global memory. Both kernels are written in C/C++. Figure 34 shows the details of the hardware implementation.

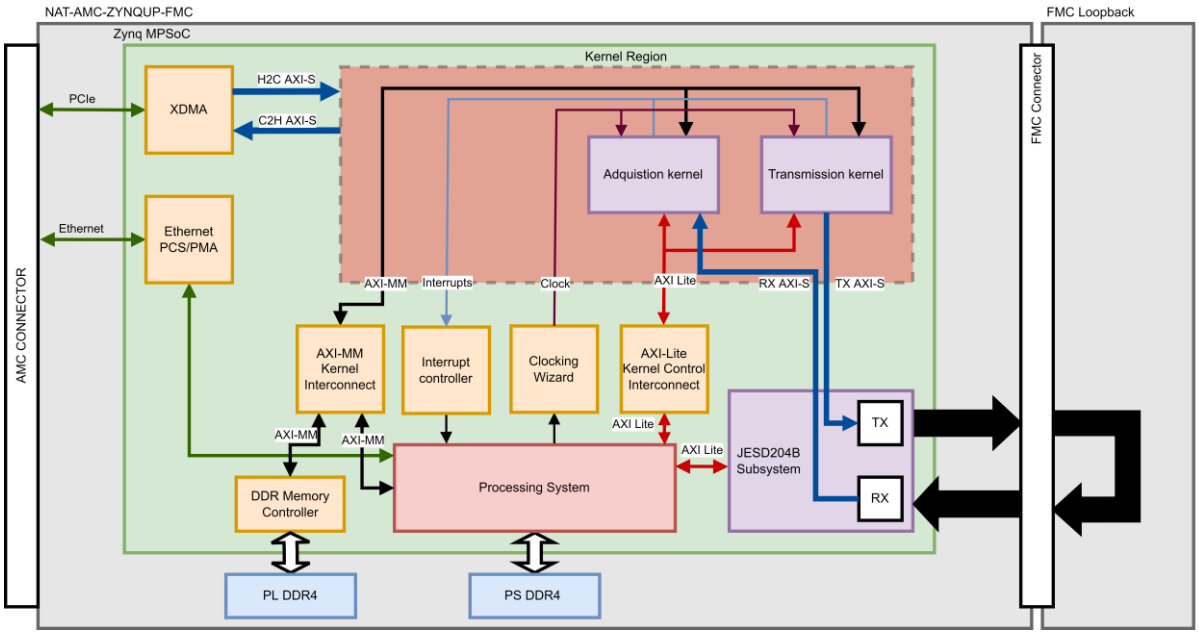

Figure 34. Acquisition system block diagram

Note that the JESD204B link configuration defined in the embedded Linux device-tree assumes that the TX/RX device interfaces with a four-channel, 400 MS/s, 16-bit resolution DAC/ADC. The AXI Streaming interfaces run at the device clock at 200 MHz, and its width is 32 bits per lane for a total of 128 bits. Therefore, the sample mapping is shown in Figure 35, with two samples per converter in each data beat. The signal samples to be sent are an ascending ramp of integer values to facilitate later data evaluation.

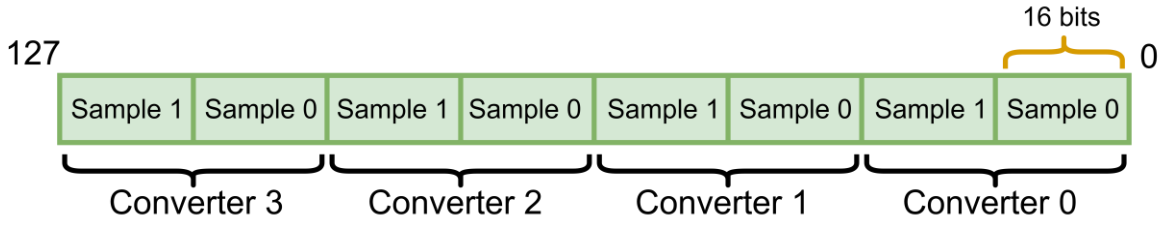

Figure 35. JESD204B sample mapping in the example acquisition application

The host program operation is as follows:

1. Initialize the runtime environment. The host detects the attached FPGA device, loads the device binary (.xclbin) into it, and creates the kernel objects.
2. Allocate host memory to store samples. The output buffer *host_ptr_w* is filled with ramp data, and *host_ptr_r* is the input buffer.
3. Create buffers in global memory to share data with the kernels. The buffer *d_buffer_w* will contain the samples to be sent over the JESD204B link, while the *d_buffer_r* will contain the acquired samples.
4. Copy ramp data from host memory to global memory (*host_ptr_w* to *d_buffer_w).*
5. Set the kernel arguments. For the transmission kernel, the arguments are the number of samples to send and the global memory buffer containing the samples (*d_buffer_w)*. For the acquisition kernel, the arguments are the number of samples to acquire and the global memory buffer to store them (*d_buffer_r)*.
6. Launch acquisition kernel.
7. Launch transmission kernel.
8. When both kernels have finished executing, migrate acquired data from global memory to host memory (*d_buffer_r* to *host_ptr_r*).
9. Check that the acquired samples have the expected ramp values and save the samples to files.
10. Release resources.

The application works as expected, as can be seen in the command line:

```
root@namc_zynqup_jesd204_fmc:/media/sd-mmcblk0p1# ./adq binary_container_1.xclbin
Found Platform
Platform Name: Xilinx
INFO: Reading binary_container_1.xclbin
Loading: 'binary_container_1.xclbin'
Trying to program device[0]: namc_zynqup_jesd204_fmc
Device[0]: program successful!
Allocated 65536 bytes of host memory to store the output signal.
Allocated 2097152 bytes of host memory to store the input signal.
Output signal global buffer created.
Input signal global buffer created.
Kernel arguments set.
Launched rx kernel!
```

```
Launched tx kernel!
Migrated acquired samples to host memory.
Test successful! No errors found.
Channel 0 data available in file: /home/petalinux/channel_0.dat
Channel 1 data available in file: /home/petalinux/channel_1.dat
Channel 2 data available in file: /home/petalinux/channel_2.dat
Channel 3 data available in file: /home/petalinux/channel_3.dat
```

It is worth mentioning that for the acquisition kernel to work correctly, it should be run at a higher frequency than the default 200 MHz clock; in this case, it was tested at 400 MHz. This is because at 200 MHz, the simple acquisition kernel developed cannot store the data in the global memory fast enough, resulting in sample loss.

4.5 Hardware Resource Utilization

One aspect to pay attention to when developing Vitis Platforms is the utilization of FPGA resources. Any resources used to implement the static hardware part cannot be used by the acceleration kernels. Table 7 shows the total resources used by the implemented design. Although the platform hardware includes complex interfaces such as JESD204B, Ethernet, and a memory controller, most FPGA resources are available for implementing hardware-accelerated applications.

Table 7. FPGA Resource Utilization

Resource	Utilization (Units)	Utilization (%)	Available (Units)	Available (%)
LUT (LookUp Table)	62963	27,3	167418	72,7
LUTRAM	6333	6,2	95484	93,8
Flip Flops	85434	18,5	375375	81,5
Block RAM	110,5	35,4	201	64,6
Ultra RAM	16	16,7	80	83,3
Digital Signal Processors	3	0,2	1762	99,8
IO	136	29,3	328	70,7
Gigabit Transceivers	14	58,3	10	41,7
BUFG	29	5,3	515	94,7
MMCM	2	25	6	75
PLL	4	25	12	75

Chapter 5. Project impact, dissemination, conclusions, and future work

5.1 Project Impact

The project proposes an innovative high-speed data acquisition system for an FPGA-based AMC board that is part of a MicroTCA chassis, that allows the development of hardware-accelerated applications using the OpenCL framework and high-level synthesis languages. The solution provides the following features:

- A GitHub repository has been prepared, which automates the building process of a complete data acquisition system.
- The solution avoids the complexity of using and managing a JESD204B link. This allows the user to focus on the development of the processing algorithm without being aware of the complexity of the underlying hardware.
- The designed system allows users to evaluate their own processing algorithm developed using high-level synthesis on real hardware.

All these features facilitate the implementation of high-speed data acquisition and processing systems in the following areas:

- **Big Science.** MicroTCA systems are used in scientific research applications requiring high performance and reliability, particularly in fusion environments and accelerators, which can benefit from the developed system. Some examples are the control of the European XFEL (X-Ray Free-Electron Laser Facility) [31] and the CMS (Compact Muon Solenoid) experiment at the LHC (Large Hadron Collider) at CERN [32], which are already using MicroTCA; or ITER, which is evaluating the implementation of MicroTCA.
- **Telecommunications.** High-speed data transfer is critical in telecommunications applications such as network monitoring, wireless, and satellite communications. In addition, the market availability of FMC modules with specialized radio-frequency transceivers, such as the Analog Devices ADRV9009 [33], facilitates system implementation.
- **Medical Imaging.** Medical imaging involves the acquisition, analysis, and manipulation of digital images for medical diagnosis and treatment, and it often requires

the acquisition and processing of large amounts of image data. MicroTCA and FPGAs provide the required processing capabilities for such applications [34].

Therefore, the developed system has the potential to have a significant impact on the field of data acquisition and processing, contributing to the **SDG** (Sustainable Development Goal) **9: Industry, Innovation, and Infrastructure,** especially to **target 9.5**, enhance scientific research, improve the technological capabilities of industrial sectors in all countries [35].

5.2 Dissemination

Part of this work is involved in publishing a paper titled *"Using FPGA-based AMC Carrier Boards for FMC to Implement Intelligent Data Acquisition Applications in MTCA Systems using OpenCL"* in the *IEEE Transactions on Nuclear* journal [9]. In particular, it has contributed to exploring the possibilities of Xilinx's hardware acceleration methodology, obtaining comparative results with Intel's methodology, and implementing data acquisition and processing applications using Xilinx's reference designs.

5.3 Conclusions

In this project, the integration of a complex system consisting of many different components was achieved. An SoC integrating an FPGA with an embedded processor running Linux was used, on which the hardware was designed to implement a JESD204B interface for data acquisition as well as integration with the OpenCL framework to run hardware-accelerated applications, all for an AMC form factor board that is part of a MicroTCA system. The following conclusions can be drawn from the use of all these technologies:

- SoCs with integrated FPGA fabric, such as Zynq MPSoC devices, offer great flexibility and customization for hardware and software to meet specific application requirements. This is further enhanced by running an embedded Linux on the system, which benefits from the large open-source code and community support.
- Analog Devices provides a valid and open-source solution for implementing the JESD204B interface on FPGAs and software support for link bring-up and management.
- It is proven that using Xilinx's OpenCL-based application acceleration methodology for high-speed data acquisition systems on MicroTCA platforms is possible and viable.
- The use of a single common Linux driver and API facilitates the development of OpenCL-based accelerated applications.
- Using high-level synthesis languages over traditional low-level hardware description languages provides a faster design process, better abstraction, and overall better design optimization.

5.4 Future Work

The designed system is fully functional and meets the objectives set for the project's development. However, there is room for improvement in the project:

- The implemented JESD204B interface was verified using an FMC loopback card. However, it remains to integrate the developed system with an FMC module with converters using the JESD204B interface to implement a data acquisition system capable of performing real-world measurements.
- A simple script has been developed to test the PCIe communication from the host PC to the AMC module and vice versa. Developing OpenCL applications that use kernels to communicate with the host PC would be interesting.
- The developed OpenCL applications serve as examples. More complex and intensive applications can be developed that use the processing capabilities of the Zynq MPSoC.
- The project may be useful to the MicroTCA community, among others. A more refined version could be released as open source.

References

[1] NAT, "MTCA - The proven successor for VME and CPCI," [Online]. Available: https://www.wb-ip.com.au/uploads/3/8/8/4/38841341/it_is_about_time_(cover_story).pdf. [Accessed 25 April 2023].

[2] J. Harris, "What is JESD204 and Why Should We Pay Attention to It?".

[3] PICMG, "About PICMG," [Online]. Available: https://www.picmg.org/about-picmg/. [Accessed 20 February 2023].

[4] PICMG, "MicroTCA Short Form Specification," 2006.

[5] PICMG, "The evolution never ends: PICMG announces new MicroTCA specifications," [Online]. Available: https://www.picmg.org/the-evolution-never-ends-picmg-announces-new-microtca-specifications/. [Accessed 21 February 2023].

[6] K. Rehlich, "MicroTCA Communication Links," [Online]. Available: https://indico.desy.de/event/35560/contributions/133040/attachments/78899/102531/MicroTCA_CommunicationClocksTriggers2022.pdf. [Accessed 22 March 2023].

[7] Vadatech, "Vadatech MicroTCA Overview," January 2016. [Online]. Available: https://www.vadatech.com/media/pdf_MicroTCA_Overview.pdf. [Accessed 23 February 2023].

[8] NAT, "NAT-MCH-PHYS80," [Online]. Available: https://nateurope.com/product/nat-mch-phys-phys80/. [Accessed 28 February 2023].

[9] M. Ruiz, A. Carpeño, D. Rivilla, M. Astrain, A. Piñas and V. Costa, "Using FPGA-based AMC Carrier Boards for FMC to Implement Intelligent Data Acquisition Applications in MTCA Systems using OpenCL," *IEEE Transactions on Nuclear Science.*

[10] Jie Zhang, Hangxu Li, Jingzi Gu and Xiaoshan Jiang, "The MicroTCA fast control board for generic control and data acquisition applications in HEP experiments," *IEEE TRANSACTION ON NUCLEAR SCIENCE,* vol. 10, no. 10, 2018.

[11] VITA, "ANSI/VITA 57.1 FPGA Mezzanine Card (FMC) Standard," 2019.

[12] IOxOS, "ADC_3112," [Online]. Available: https://www.ioxos.ch/produit/adc_3112/. [Accessed 28 February 2023].

[13] G. Diniz, "JESD204B vs. Serial LVDS Interface Considerations for Wideband Data Converter Applications".

[14] J. Harris, "Understanding Layers in the JESD204B Specification - A High Speed ADC Perspective," 2017.

[15] Xilinx, "Vitis Unified Software Platform Documentation," [Online]. Available: https://docs.xilinx.com/v/u/en-US/ug1416-vitis-documentation. [Accessed 01 April 2023].

[16] Xilinx, "Vitis Getting Started Tutorial," [Online]. Available: https://github.com/Xilinx/Vitis-Tutorials/tree/2022.2/Getting_Started/Vitis. [Accessed 18 April 2023].

[17] Xilinx, "XRT and Vitis Platform Overview," [Online]. Available: https://xilinx.github.io/XRT/2021.1/html/platforms.html. [Accessed 07 March 2023].

[18] Xilinx, "Xilinx Adaptative SoCs," [Online]. Available: https://www.xilinx.com/products/silicon-devices/soc.html. [Accessed 07 March 2023].

[19] "Device Tree Reference," [Online]. Available: https://elinux.org/Device_Tree_Reference#Introduction. [Accessed 07 April 2023].

[20] Xilinx, "Vitis Embedded Platform Source," [Online]. Available: https://github.com/Xilinx/Vitis_Embedded_Platform_Source. [Accessed 08 March 2023].

[21] NAT, *NAT-AMC-ZYNQUP-FMC Technical Reference Manual.*

[22] Terasic, "Terasic FMC/FMC+ Loopback," [Online]. Available: https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=64&No=1232&PartNo=2#contents. [Accessed 31 March 2023].

[23] Xilinx, "DMA/Bridge Subsystem for PCI Express Product Guide," [Online]. Available: https://docs.xilinx.com/r/en-US/pg195-pcie-dma/Overview?tocId=O_EMX26J5IsdubL4i3XJ_w. [Accessed 25 March 2023].

[24] Xilinx, "XDMA IP drivers," [Online]. Available: https://github.com/Xilinx/dma_ip_drivers. [Accessed 25 March 2023].

[25] Analog Devices, "JESD204 Interface Framework," [Online]. Available: https://wiki.analog.com/resources/fpga/peripherals/jesd204. [Accessed 16 February 2023].

[26] Xilinx, "GTH Transceiver Protocol List," [Online]. Available: https://www.xilinx.com/content/dam/xilinx/support/documents/data_sheets/ds922-kintex-ultrascale-plus.pdf#page=50. [Accessed 16 April 2023].

[27] Analog Devices, "meta-adi Layer Source," [Online]. Available: https://github.com/analogdevicesinc/meta-adi. [Accessed 09 April 2023].

[28] Analog Devices, "JESD204B(FSM) Interface Linux Kernel Framework," [Online]. Available: https://wiki.analog.com/resources/tools-software/linux-drivers/jesd204/jesd204-fsm-framework. [Accessed 09 April 2023].

[29] A. Piñas, "NAT-AMC-ZYNQUP-FMC Acceleration Platform," [Online]. Available: https://github.com/i2a2/namc_zynqup_fmc_bsp/tree/ad_jesd204_2021.1.

[30] "sysfs(5) Linux Manual Page," [Online]. Available: https://man7.org/linux/man-pages/man5/sysfs.5.html. [Accessed 13 April 2023].

[31] DESY, "MicroTCA technology lab," [Online]. Available: https://innovation.desy.de/technologies/microtca/index_eng.html. [Accessed 24 April 2023].

[32] M. Joos, "A comparative discussion of bus/crate standards and their use at CERN," [Online]. Available: https://indico.cern.ch/event/673073/contributions/2770356/attachments/1553050/2440986/COMPASS_JOOS.ppt. [Accessed 24 April 2023].

[33] Analog Devices, "ADRV9009," [Online]. Available: https://www.analog.com/en/products/adrv9009.html#product-overview. [Accessed 24 April 2023].

[34] C. Bestler and S. Rupp, "MicroTCA for Medical Applications," [Online]. Available: https://www.kontron.com/download/download?filename=/downloads/white_papers/microtca_for_medical_-applications.pdf&product=88594. [Accessed 24 April 2023].

[35] United Nations, "SDG Goal 9: Build resilent infrastructure, promote sustainable industrialization and foster innovation," [Online]. Available: https://www.un.org/sustainabledevelopment/infrastructure-industrialization/. [Accessed 24 April 2023].

[36] GNU, "GNU General Public License," [Online]. Available: https://www.gnu.org/licenses/gpl-3.0.html. [Accessed 16 February 2023].

[37] XIlinx, "ZYNQ UltraScale+ MPSoC," [Online]. Available: https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html#productAdvantages. [Accessed 21 March 2023].

[38] Analog Devices, "DAQ2 HDL Project for Xilinx," [Online]. Available: https://wiki.analog.com/resources/fpga/peripherals/jesd204/tutorial/hdl_xilinx. [Accessed 31 March 2023].

[39] Analog Devices, "AD-FMCDAQ2-EBZ Evaluation Board," [Online]. Available: https://www.analog.com/en/design-center/evaluation-hardware-and-software/evaluation-boards-kits/eval-ad-fmcdaq2-ebz.html#eb-overview. [Accessed 31 April 2023].

[40] Xilinx, "JESD204 v7.2 Product Guide," [Online]. Available: https://docs.xilinx.com/v/u/en-US/pg066-jesd204. [Accessed 25 March 2023].

[41] PICMG, "MicroTCA® Overview," [Online]. Available: https://www.picmg.org/openstandards/microtca/. [Accessed 20 February 2023].

[42] Analog Devices, "AD9680 Datasheet".

[43] PICMG, "MicroTCA Base Specification," 2020.

[44] JEDEC, "Serial Interface for Data Converters JESD204B," 2011.

[45] R. Waldt, "MTCA.4 Tutorial - Introduction in xTCA," in 11th MicroTCA Workshop, December 2022.

APPENDIX I. Budget

The project's total cost can be divided into three categories: hardware costs, software costs and human resources costs. The Instrumentation and Applied Acoustic Research Group provided all necessary resources for the project's development. A detailed breakdown can be observed in Table 8.

Table 8. Complete project budget

Type	Details	Units	Value
Hardware	NAT-AMC-ZYNQUP-FMC	1	6780,00 €
	PowerBridge MTCA-6P-80R	1	12000,0 €
	Terasic FMC loopback	1	186,29 €
	Development PC	1	800,0 €
	NPCIe-UPLINK-O	1	5636,68 €
	Optical cable	2	346,7 €
Software	Xilinx Vivado	1	0,0 €
	Xilinx Vitis	1	0,0 €
	Xilinx Petalinux	1	0,0 €
	Ubuntu 20.04	1	0,0 €
	Analog Devices JESD204 IP Cores	1	0,0 €
	Analog Devices JESD204 Drivers	1	0,0 €
Human resources	Junior Engineer	320 hours	23,42 €/h
Total	**33244,07 €**		

Note: The license obtained for Vivado serves academic purposes; therefore, it does not add any cost to the project. Also, Vitis is license-free; however, a Vivado license is needed when the project requires hardware compilation.

An important aspect to highlight is the reduction of software costs due to using **free and open-source components**. In specific, Analog Devices resources are available under GPL2 license, which allows the end user the freedom to run, study, share, and modify the source [25] [36]. Furthermore, Petalinux can be used for free, and many of its components are open-source such as u-boot and the Linux kernel itself. Finally, the Ubuntu operating system (used for development) can also be run for free but uses a mixed license as it uses open-source components and proprietary drivers.

APPENDIX II. JESD204B Device-tree

The device tree is a data structure used to describe the hardware components of a system in a standardized way. It consists of a hierarchical tree structure, with each node representing a hardware component and its properties. The device tree is used by the operating system to understand the hardware configuration and to configure device drivers and other system components accordingly.

This section details the device-tree implementation so the embedded Linux can configure the Analog Devices JESD204 Drivers to manage the hardware implemented in the PL region.

Nodes present in the *"system-user.dtsi"* file present in the meta-user layer are added to the final DTB (Device Tree Blob) which is the file used by the system at boot-time to find non-discoverable hardware. Petalinux automatically generates device-tree nodes for all memory mapped peripherals included in the hardware design. These nodes can be found in "*<project-directory>/components/plnx_workspace/device-tree/device-tree/pl.dtsi"*. The first thing to do is delete them because they do not provide a valid hardware description.

```
/*Delete unnecesary auto-generated nodes*/
/delete-node/ &axi_adxcvr_tx;
/delete-node/ &axi_adxcvr_rx;
/delete-node/ &jesd204_tx_axi_jesd204_tx_0;
/delete-node/ &jesd204_rx_axi_jesd204_rx_0;
```

To add support for Analog Devices JESD204 Linux Kernel Framework, the first node to add describes the JESD204B reference clock. For this design, the reference clock is a 400 MHz fixed clock generated by a PLL.

```
/*JESD reference clock*/
clk_jesd: clk_jesd {
          #clock-cells = <0>;
          clock-frequency = <400000000>;
          compatible = "fixed-clock";
};
```

The next nodes define the configuration of the physical layer, defined by the AXI_ADXCVR IP. There are two nodes since one handles transmission transceivers and the other the reception transceivers. The required properties are:

- **compatible:** Should be "adi,axi-adxcvr-1.00" to bind the correct driver.
- **reg:** Physical address and range.
- **clock-names:** List of input clocks, should be "s_axi_clk" for the AXI Interface clock and "conv" for the reference clock.
- **clocks:** Phandles to clock source.

- **clock-output-names:** List of generated clocks, “tx_gt_clk” and “rx_gt_clk” are the Gigabit Transceiver reference clocks, while “tx_out_clk” and “rx_out_clk” is the reference clocks for the data link layer peripherals.

- **adi,sys-clk-select:** Selects the PLL reference clock source used by the transceivers, which can be CPLL, QPLL0, or QPLL1. For this design, the QPLL1 is used because its operating range is 8-13 GHz, and the desired lane rate is 8Gbps.

- **adi,out-clk-select:** Selects the output clock source to the data link layer peripherals. In this design, the reference clock divided by two is used, as this provides a 200 MHz clock required by the data link layer, which must be lane_rate/40.

- **adi,use-lpm-enable:** Selects the Low Power Mode.

- **jesd204-device:** Registers the core as a device for the JESD204 Linux Kernel Framework.

```
axi_adxcvr_rx: axi_adxcvr_rx@80030000 {
          compatible = "adi,axi-adxcvr-1.0";

          /*Address and range*/
          reg = <0x80030000 0x1000>;

          /*Input clock*/
          clocks = <&zynqmp_clk 73>, <&clk_jesd>;
          clock-names = "s_axi_aclk", "conv";

          /*Output Clocks*/
          #clock-cells = <1>;
          clock-output-names = "rx_gt_clk", "rx_out_clk";

          /*Configuration*/
          adi,sys-clk-select = <XCVR_QPLL1>;
          adi,out-clk-select = <XCVR_REFCLK_DIV2>;
          adi,use-lpm-enable;

          /*FSM support*/
          jesd204-device;
          #jesd204-cells = <2>;
};

axi_adxcvr_tx: axi_adxcvr_tx@80040000 {
          compatible = "adi,axi-adxcvr-1.0";

          /*Address and range*/
          reg = <0x80040000 0x10000>;

          /*Input clock*/
          clocks = <&zynqmp_clk 73>,  <&clk_jesd>;
          clock-names = "s_axi_aclk", "conv";

          /*Output Clocks*/
          #clock-cells = <1>;
          clock-output-names = "tx_gt_clk", "tx_out_clk";

          /*Configuration*/
          adi,sys-clk-select = <XCVR_QPLL1>;
          adi,out-clk-select = <XCVR_REFCLK_DIV2>;
          adi,use-lpm-enable;

          /*FSM support*/
          jesd204-device;
          #jesd204-cells = <2>;
};
```

Now, the nodes for the data link layer are defined. The configuration interface is implemented by the AXI JESD204B Transmit/Receive Peripheral. The required properties are:

- **compatible:** It should be “adi,axi-jesd204-rx-1.0” for the receive core and “adi,axi-jesd204-tx-1.0” for the transmit core.
- **reg:** Physical address and range.
- **interrupts:** Describes the interrupt number. The driver performs IRQ Handling; however, the IP core does not generate any interrupts at the moment.
- **clock-names:** The list of input clocks should be “s_axi_clk” for the AXI Interface clock, “device_clk” for the reference output clock from ADXCVR, and “lane_clk” for the transceiver reference clock provided by ADXCVR.
- **clocks:** Phandles to clock source.
- **clock-output-names:** Generated clocks.
- **jesd204-device:** Registers the core as a device for the JESD204 Linux Kernel Framework.
- **jesd204-inputs:** The first cell is the phandle to the PHY layer node; this defines a relationship as the data link layer depends on the physical layer to work. The second cell is the topology ID, and the third cell is the link ID inside that topology.

```
axi_jesd204_rx: axi_jesd204_rx_0@80050000 {
          compatible = "adi,axi-jesd204-rx-1.0";

          /*Address and range*/
          reg = <0x80050000 0x10000>;

          /*Interrupts*/
          interrupts = <0 107 IRQ_TYPE_LEVEL_HIGH>;

          /*Input clocks*/
          clock-names = "s_axi_aclk", "device_clk", "lane_clk";
          clocks = <&zynqmp_clk 73>, <&axi_adxcvr_rx 1>, <&axi_adxcvr_rx 0>;

          /*Output clock*/
          #clock-cells = <0>;
          clock-output-names = "jesd_rx_lane_clk";

          /* FSM support */
          jesd204-device;
          #jesd204-cells = <2>;
          jesd204-inputs = <&axi_adxcvr_rx 0 1>;
};

axi_jesd204_tx: axi_jesd204_tx_0@80060000 {
          compatible = "adi,axi-jesd204-tx-1.0";

          /*Address and range*/
          reg = <0x80060000 0x10000>;

          /*Interrupts*/
          interrupts = <0 106 IRQ_TYPE_LEVEL_HIGH>;

          /*Input clocks*/
          clock-names = "s_axi_aclk", "device_clk", "lane_clk";
          clocks = <&zynqmp_clk 73>, <&axi_adxcvr_tx 1>, <&axi_adxcvr_tx 0>;
```

```
        /*Output clock*/
        #clock-cells = <0>;
        clock-output-names = "jesd_tx_lane_clk";

        /* FSM support */
        jesd204-device;
        #jesd204-cells = <2>;
        jesd204-inputs = <&axi_adxcvr_tx 0 0>;

};
```

Finally, the top-level node is defined. In a final system, the top device would be a converter, but since both RX and TX are looped, the jesd204-top-device driver is used as the top device of the topology. This driver provides the link configuration. Its required properties are:

- **compatible:** Must be "adi,jesd204-top-device".
- **jesd204-device:** Registers the core as a device for the JESD204 Linux Kernel Framework.
- **jesd204-top-device:** Indicates that this is the top-level device of the topography described by the cell.
- **jesd204-link-ids:** Indicates the link IDs.
- **jesd204-inputs:** Declares a relationship with the data link layer. The first cell is the phandle to the data link layer node. The second cell is the topology ID, and the third cell is the link ID inside that topology.
- **adi,jesd-links:** Defines the link configuration. In this case, as both links must synchronize with each other they share the same configuration. This node describes a JESD204B subclass 1 and four-lane link, with four converters working at 400 MS/s with 16-bit resolution. The data link layer uses two octets per frame and 32 frames per multiframe. Scrambling is enabled.

```
fake-adc-dac{
        compatible = "adi,jesd204-top-device-1.0";
        /*FSM support*/
        jesd204-device;
        #jesd204-cells = <2>;
        jesd204-top-device = <0>; /* This is the TOP device */
        jesd204-link-ids = <0 1>;
        jesd204-inputs = <&axi_jesd204_rx 0 1>, <&axi_jesd204_tx 0 0>;

        adi,jesd-links {
        #size-cells = <0>;
        #address-cells = <1>;

                rx_jesd: link@1 {
                        reg = <1>;

                        adi,sample-rate = /bits/ 64 <400000000>;
                        adi,subclass = <1>; /* JESD SUBCLASS 0,1,2 */
                        adi,version = <1>;                      /* JESD VERSION*/
                        adi,dual-link = <0>;                    /* JESD Dual Link Mode */
                        adi,converters-per-device = <4>;        /* JESD M */
                        adi,octets-per-frame = <2>;             /* JESD F */
                        adi,frames-per-multiframe = <32>;       /* JESD K */
```

```
                        adi,converter-resolution = <16>;       /* JESD N */
                        adi,bits-per-sample = <16>;            /* JESD NP' */
                        adi,control-bits-per-sample = <0>;     /* JESD CS */
                        adi,lanes-per-device = <4>;            /* JESD L */
                        adi,samples-per-converter-per-frame = <2>; /* JESD S */
                        adi,high-density = <0>;                        /* JESD HD */
                        adi,scrambling = <1>;                          /*JESD SCR*/
                };

                tx_jesd: link@0 {
                        reg = <0>;
                        adi,sample-rate = /bits/ 64 <400000000>;
                        adi,subclass = <1>; /* JESD SUBCLASS 0,1,2 */
                        adi,version = <1>;                     /* JESD VERSION*/
                        adi,dual-link = <0>;                   /* JESD Dual Link Mode */
                        adi,converters-per-device = <4>;       /* JESD M */
                        adi,octets-per-frame = <2>;            /* JESD F */
                        adi,frames-per-multiframe = <32>;      /* JESD K */
                        adi,converter-resolution = <16>;       /* JESD N */
                        adi,bits-per-sample = <16>;            /* JESD NP' */
                        adi,control-bits-per-sample = <0>;     /* JESD CS */
                        adi,lanes-per-device = <4>;            /* JESD L */
                        adi,samples-per-converter-per-frame = <2>; /* JESD S */
                        adi,high-density = <0>;                        /* JESD HD */
                        adi,scrambling = <1>;                  /*JESD SCR*/
                };
        };
```