TELECO.

CAMPUS SUR·UPM

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA Y SISTEMAS DE TELECOMUNICACIÓN

POLITÉCNICA

PROYECTO FIN DE GRADO

TITLE: Design and implementation of a signal acquisition and processing architecture over mPCIe on a Xilinx Artix 7

AUTOR/A: David Andrino Izquierdo

TITULACIÓN: Grado en Ingeniería Electrónica de Comunicaciones

TUTOR/A: Antonio Carpeño Ruiz

DEPARTAMENTO: Departamento de Ingeniería Telemática y Electrónica

VºBº TUTOR/A

Miembros del Tribunal Calificador:

PRESIDENTE/A: Sofía Di Sarno García

TUTOR/A: Antonio Carpeño Ruiz

SECRETARIO/A: Mariano Ruiz González

Fecha de lectura:

Calificación:

El Secretario/La Secretaria,

This work was supported by Grant PID2022-137680OB-C33 funded by MCIN/AEI/ 10.13039/501100011033 and by “ERDF A way of making Europe”.

Resumen

Las tarjetas de adquisición de datos basadas en Field-Programmable Gate Array (FPGA) se utilizan habitualmente en proyectos de gran ciencia como International Thermonuclear Experimental Reactor (ITER) o Axially Symmetric Divertor EXperiment (ASDEX) Upgrade. Estos proyectos requieren características de alto rendimiento como baja latencia o alta fiabilidad.

Este trabajo presenta una arquitectura sobre una tarjeta con una FPGA, un Analog-to-Digital Converter (ADC) y un Digital-to-Analog Converter (DAC) para adquisición y procesado de datos. El diseño cuenta con un bloque de comunicación Peripheral Component Interconnect Express (PCIe) que puede ser utilizado para configuración o para transferencia de datos en flujo. La arquitectura está implementada en un Hardware Description Language (HDL) sobre una placa Micro PCIe con factor de forma pequeño. La placa cuenta con una FPGA Artix 7 con un ADC y un DAC externos que se comunican con una Serial Peripheral Interface (SPI).

La arquitectura permite la implementación de algoritmos en High-Level Synthesis (HLS) (o en HDLs), haciendo que cualquier científico los pueda implementar fácilmente sin conocimientos avanzados de modelado. Un equipo de expertos también se puede beneficiar de esta arquitectura ya que no necesita diseñar ni implementar la lógica de gestión de adquisición ni de comunicación.

El proyecto incluye además una interfaz gráfica en Python para la configuración manual y pruebas rápidas y una librería en C++ para su gestión automática e integración. El flujo de trabajo del proyecto está automatizado con herramientas estándar como Git y Make, e incluye un manual de desarrollo para facilitar su uso. Incluye filtros Finite Impulse Response (FIR) e Infinite Impulse Response (IIR) como ejemplo de algoritmos fácilmente implementables.

El diseño permite adquisición y procesado continuo a una frecuencia de hasta 100 kSPS o adquisición y procesado hasta 200 kSPS sin el DAC.

La capacidad de transmisión en flujo a través de PCIe y la facil implementación de algoritmos permite que este dispositivo se utilice para aceleración hardware de algoritmos, un caso de uso importante para campos como transcodificación de imagen y vídeo o inteligencia artificial.

Abstract

Data acquisition devices based on Field-Programmable Gate Arrays (FPGAs) are widely used on big science projects like International Thermonuclear Experimental Reactor (ITER) or Axially Symmetric Divertor EXperiment (ASDEX) Upgrade. These projects require high-performance characteristics like low latency or high reliability.

In this work, an architecture for Data Acquisition (DAQ) based on an FPGA with an Analog-to-Digital Converter (ADC) and Digital-to-Analog Converter (DAC) for data acquisition and processing is presented. The design includes a Peripheral Component Interconnect Express (PCIe) communication block that can be used for configuration or data streaming. The architecture is implemented in a Hardware Description Language (HDL). The device used is a Micro PCIe board with a small form factor that includes an Artix 7 FPGA with an external ADC and DAC with a Serial Peripheral Interface (SPI).

It allows for the implementation of algorithms with High-Level Synthesis (HLS) (or HDLs) so that any scientist can implement them easily without advanced modelling knowledge. An expert team can also benefit from this architecture, as it removes the need to implement the acquisition management and the communication logic.

The project includes a complementary Python Graphical User Interface (GUI) for manual management of the configuration and fast testing, and a C++ library for automatic management and integration. The project's development cycle is automated with standard tools such as Git or Make, and a development manual is included to ease its usage. It includes Finite Impulse Response (FIR) and Infinite Impulse Response (IIR) filters as examples of basic implementable algorithms.

The system supports continuous sampling, processing, and generation up to 100 kSPS, or sampling and processing up to 200 kSPS if the DAC is not used.

The PCIe information streaming capabilities and easy algorithm implementation enable this device to be used as a hardware accelerator, a valuable tool in various fields such as image and video transcoding or artificial intelligence.

Agradecimientos

Quiero dar las gracias a todos mis compañeros del I2A2, especialmente a mis tutores Antonio y Mariano, por darme la oportunidad, la confianza y la compañía para realizar estos proyectos.

A todos los que me han acompañado en la carrera: Estela, Diego, Agus, Fernando, Jorge, Hugo y Osvaldo por apoyarme cada día y superar juntos los malos momentos.

A mi familia y amigos, en especial a mi madre, por vuestro apoyo y cariño incondicional.

A Olalla y Sergio, porque sin vosotros no hubiera podido llegar hasta aquí.

Muchas gracias a todos.

Contents

Resumen **V**

Abstract **VII**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Project objectives . . . 2
1.3 Document structure . . . 3

2 State of the Art **5**
2.1 Nuclear Fusion . . . 5
2.2 FPGA devices . . . 8
2.3 DAQ solutions . . . 10
2.3.1 RIO devices . . . 10
2.3.2 MicroTCA . . . 11
2.4 Communication interfaces . . . 13
2.4.1 USB . . . 13
2.4.2 PCIe . . . 13
2.4.3 PXIe . . . 14
2.5 System implementation languages . . . 16
2.5.1 Hardware Description Languages . . . 16
2.5.2 High level languages . . . 17
2.6 On-Chip communication . . . 20
2.6.1 AXI . . . 20
2.6.2 AXI Stream . . . 22
2.7 Related work . . . 23

3 Implementation **25**
3.1 Resources . . . 25
3.1.1 Hardware resources . . . 25
3.1.2 Software resources . . . 27
3.2 Design architecture . . . 28
3.2.1 ADC module . . . 30
3.2.2 DAC module . . . 32
3.2.3 Configuration registers module . . . 34
3.2.4 DSP module . . . 36
3.2.5 Sampling frequency generator module . . . 37
3.2.6 Reset generator module . . . 37
3.2.7 XDMA IP core . . . 37
3.3 HLS algorithm implementation . . . 39
3.3.1 Width adapter . . . 40

3.3.2 FIR filter . . . 40
3.3.3 IIR filter . . . 42
3.4 Simulation . . . 44
3.4.1 ADC VHDL module test . . . 44
3.4.2 DAC VHDL module test . . . 45
3.4.3 Configuration registers VHDL test . . . 46
3.4.4 Width adapter HLS test . . . 47
3.4.5 IIR filter HLS test . . . 48
3.4.6 FIR filter HLS test . . . 48
3.5 Python GUI . . . 50
3.6 C++ Library . . . 53
3.7 Project management . . . 55

4 Results **57**
4.1 Design verification . . . 57
4.1.1 ADC VHDL test results . . . 57
4.1.2 DAC VHDL test results . . . 57
4.1.3 Configuration registers VHDL test results . . . 58
4.1.4 HLS test results . . . 58
4.2 Design validation . . . 65
4.2.1 FIR filter validation . . . 65
4.2.2 DAQ frequency results . . . 67
4.2.3 C++ Library results . . . 71
4.3 Design resource usage . . . 73

5 Budget **77**

6 Project impact **79**
6.1 Sustainable Development Goals . . . 80

7 Conclusions and next steps **81**
7.1 Conclusions . . . 81
7.2 Next Steps . . . 82

8 References **83**

9 Bibliography **89**

Appendix A Vivado block design **91**

Appendix B HLS implementations **93**

Appendix C FPGA implementation floor plan **99**

Appendix D Python Bode filter **101**

Appendix E User manual **103**

List of Figures

2.1 Simplified model of tokamak and stellarator [3] 6
2.2 ITER Tokamak [1] 7
2.3 ITER coil structure [10] 7
2.4 Generic FPGA architecture [11] 8
2.5 Comparison between ASIC, CPU and FPGA [11] 9
2.6 Example of PXIe chassis with FlexRIO device[21] 11
2.7 Example of µTCA chassis [25] 12
2.8 USB connectors [28] 13
2.9 PCIe connectors [31] 14
2.10 PXIe architecture example [34] 15
2.11 OpenCL architecture [40] 18
2.12 HLS methodologies [41] 19
2.13 Examples of AXI information transfers [43] 20
2.14 Read and write sequences on AXI interface [43] 21

3.1 TMPE627 picture and block diagram [52] 25
3.2 Other hardware components used 26
3.3 TMPE627 plugged into the motherboard 27
3.4 Block design of the architecture 28
3.5 Data flows of the architecture 30
3.6 Generic ADC SPI transaction [54] 31
3.7 ADC state diagram 32
3.8 DAC state machine diagram 33
3.9 Content of the configuration registers 34
3.10 CDC register handshake 36
3.11 Frequency response of the designed FIR filter 41
3.12 HLS loop unrolling [67] 42
3.13 Block diagram of the ADC module test bench 45
3.14 Block diagram of the DAC module test bench 46
3.15 Best and worst case for CDC handshake synchronization 47
3.16 Input and expected output on HLS IIR test 48
3.17 Input and expected output on HLS FIR tests 49
3.18 Python application stack 50
3.19 Python GUI for register access 51
3.20 Example C2H reading from Python GUI 52
3.21 Stack of a generic application implemented with the library 54
3.22 Design language distribution 55

4.1 Simulation of one ADC read 59
4.2 Zoom on the ADC read start and finish 60
4.3 Simulation of one DAC transaction 61

4.4 Simulation of AXI Lite transaction 62
4.5 Simulation of CDC data transfer 63
4.6 HLS completion reports 64
4.7 Block diagram of the FIR verification 65
4.8 FIR filter results on time domain 66
4.9 Bode diagram of FIR filter amplitude on Red Pitaya 67
4.10 Filtered bode diagram of FIR filter amplitude 67
4.11 Sampling clock at 100 kHz 68
4.12 ADC busy signal at 100 kSPS 69
4.13 ADC maximum communication frequency 69
4.14 DAC busy signal at 100 kSPS 70
4.15 DAC maximum communication frequency 70
4.16 PCIe C2H streaming result 71
4.17 PCIe H2C streaming result 72
4.18 Design implementation usage histogram 73
4.19 FIR filter shift register schedule viewer 75
4.20 FIR filter convolution schedule viewer 76
4.21 Loop pipelining diagram [82] 76

A.1 Block design of the FPGA implementation 91

C.1 Floor plan of the FPGA design implementation 99

List of Tables

2.1 PCIe generation speed [30] 14

3.1 Configuration parameters of ADC module 30
3.2 TMPE627 ADC pin mapping 31
3.3 DAC configuration parameters 33
3.4 Configuration registers map 34
3.5 Range translation in registers 35
3.6 Languages and line count of the project 55

4.1 Implementation resource usage 73
4.2 HLS resource usage 74
4.3 Approximated usage of empty base design 74

5.1 Project budget 77

List of Listings

2.1 Example counter on VHDL . . . 16
2.2 Example counter on Verilog . . . 17
3.1 HLS algorithm top prototype . . . 39
4.1 C++ Library read example . . . 72
4.2 C++ Library write example . . . 72
B.1 Bit width adapter implementation in HLS . . . 93
B.2 HLS implementation of the FIR filter . . . 94
B.3 HLS implementation of the IIR filter . . . 96
D.1 Python Bode filter . . . 101

Acronyms

ADC Analog-to-Digital Converter

AMBA Advanced Microcontroller Bus Architecture

AMC Advanced Mezzanine Card

ASDEX Axially Symmetric Divertor EXperiment

ASIC Application Specific Integrated Circuit

AXI Advanced eXtensible Interface

C2H Card to Host

CDC Clock Domain Crossing

CODAC Control, Data Access and Communication

CPU Central Processing Unit

DAC Digital-to-Analog Converter

DAQ Data Acquisition

DSP Digital Signal Processor

EOC End of Count

EPICS Experimental Physics and Industrial Control System

FF Flip-Flop

FIFO First In, First Out

FIR Finite Impulse Response

FPGA Field-Programmable Gate Array

FSM Finite State Machine

FSR Full Scale Range

GPU Graphics Processing Unit

GUI Graphical User Interface

H2C Host to Card

HDL Hardware Description Language

HDVL Hardware Description and Verification Language

HLS High-Level Synthesis

HVL Hardware Verification Language
I&C Instrumentation and Control
I/O Input/Output
IIR Infinite Impulse Response
IP Intellectual Property
ITER International Thermonuclear Experimental Reactor
JET Joint European Torus
JTAG Joint Test Action Group
LAN Local Area Network
LUT Look-Up Table
LVDS Low-Voltage Differential Signaling
MCF Magnetic Confinement Fusion
NRE Non-Recursive Engineering
PCIe Peripheral Component Interconnect Express
PLL Phase-Locked Loop
PXIe PCIe eXtensions for Instrumentation Express
RAII Resource Acquisition Is Initialization
RAM Random Access Memory
RIO Reconfigurable I/O
RTL Register-Transfer Level
SDG Sustainable Development Goal
SPI Serial Peripheral Interface
SoC System-on-Chip
TCL Tool Command Language
TPU Tensor Processing Unit
USB Universal Serial Bus
µTCA Micro Telecommunications Computing Architecture
VHDL VHSIC Hardware Description Language
VHSIC Very High-Speed Integrated Circuit
VISA Virtual Instrument Software Architecture

WEST Tungsten Environment in Steady-state Tokamak

XDMA Xilinx Direct Memory Access

XPM Xilinx Parameterized Macros

Chapter 1

Introduction

1.1 Motivation

Data Acquisition (DAQ) boards are a crucial piece of electronic instrumentation. These devices are used to acquire and process samples from physical signals. They are widely used in industry and big science projects worldwide.

International Thermonuclear Experimental Reactor (ITER) is a big science project aiming to research nuclear fusion for energy generation in a clean, limitless, and safe manner. The consortium involved in its construction is composed of 33 countries with a budget of € 20 billion [1].

ITER's main objective is the construction of a tokamak, a magnetic containment device with millions of sensors. Data from these sensors needs to be captured and processed in real time, using DAQ boards. This necessity has impelled the instrumentation industry significantly in the last years.

The DAQ instrumentation market for test and measurement applications is mainly dominated by National Instruments, with device families such as Reconfigurable I/O (RIO). It provides a very complete environment for data acquisition and processing (among other use cases) using proprietary hardware and software.

This project aims to provide an alternative signal capture, processing, and generation framework without any National Instruments (Emerson) tools. It only uses the AMD Vivado Design Suite, a requirement when working with Xilinx devices, and open tools such as Xilinx drivers or the Python programming language.

This project is motivated by the deprecation of the Compact RIO model NI-9159, a widely used chassis at ITER for Instrumentation and Control (I&C) and Interlock applications that National Instruments recently delisted.

It was not an original objective, but the system can also be used for hardware algorithm acceleration thanks to the high-speed data transfer to and from the host computer. This is a very important use case in fields such as video encoding, blockchain, or artificial intelligence.

1.2 Project objectives

The technical specifications of the project are:

- The system must provide a framework for data acquisition and signal generation.
- The system must provide a generic block for algorithm implementation.
- The system must allow algorithm implementation without Hardware Description Languages (HDLs), but they can be used.
- The system must include bidirectional communication with the computer.
- The system must be configurable from the computer.
- The system must have a configurable acquisition rate.
- The system must include configurable ranges for acquisition and generation.

The restrictions of the project are:

- The system is implemented on the TEWS TMPE627 Mini Peripheral Component Interconnect Express (PCIe) board.
- The system is developed with the Vivado design suite (version 2021.1).
- The system is developed on a Rocky Linux 9.5 (Kernel version 5.14.0-427.36.1).
- The system includes Python scripts with version 3.9.21, using Matplotlib and Numpy.
- The system depends on the Xilinx XDMA drivers (Commit `03ac7f3`, March 4th, 2025)

From an academic point of view, the following aptitudes are necessary:

- Field-Programmable Gate Array (FPGA) design and implementation.
- Hardware acceleration on FPGA
- Computer-to-chip interfacing.
- Direct memory transfer methodologies.
- Linux driver management.
- Integrated circuit interfacing.
- Linux library programming.
- Technical documentation elaboration.
- Xilinx Design Suite usage.

1.3 Document structure

Chapter 2 presents a brief summary of the current state of the art in fusion technology, acquisition hardware, and related work that was researched before the design of this project.

Chapter 3 presents the designed system with a block diagram and an explanation of each module. It also shows the implementation of the auxiliary tools in Python and C++, as well as the build automation procedure.

Chapter 4 shows the results of the implemented design. It includes usage statistics, result plots and diagrams, and design parametrization.

Chapter 5 contains the project budget divided into different categories.

Chapter 6 shows the predicted impact of the project and its relation to Sustainable Development Goals.

Chapter 7 draws some conclusions and suggests some next steps for this project.

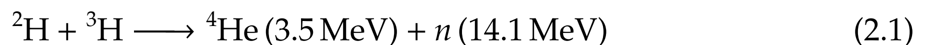
Regarding the appendices, Appendix A contains the Vivado block design developed, Appendix B shows the implemented algorithms, Appendix C contains an image of the design footprint on the FPGA, and Appendix D contains a Python utility used to improve result clarity.

Chapter 2

State of the Art

2.1 Nuclear Fusion

Nuclear fusion is the physical process of joining two light atoms into a heavier one. It can be used to generate energy in a safe and clean way, as opposed to fossil fuels. This process causes stars like the Sun to generate energy continuously. This process is generally performed with the D-T Reaction, composed of the fusion of Deuterium and Tritium into Helium with an exothermic reaction [2]. The reaction can be found in Equation 2.1.



$$^{2}\mathrm{H} + {}^{3}\mathrm{H} \longrightarrow {}^{4}\mathrm{He}\,(3.5\,\mathrm{MeV}) + n\,(14.1\,\mathrm{MeV}) \tag{2.1}$$

The reaction takes place when the atomic nuclei collide. Therefore, atoms need to be very close and at high temperatures. Because of this, the materials need to be heated and pressurized until they reach the plasma state. Then, this plasma needs to be confined in order to increase the probability of collision and avoid interaction with the vessel wall.

There are two approaches to plasma containment: Magnetic and Inertial Confinement. The Magnetic Confinement Fusion (MCF) uses toroidal magnetic fields to contain the plasma into a defined shape. Magnetic fields can be generated with an electrical current induced through the plasma (tokamak) or with external electromagnets (stellarator). Figure 2.1 shows a simplified diagram of both devices.

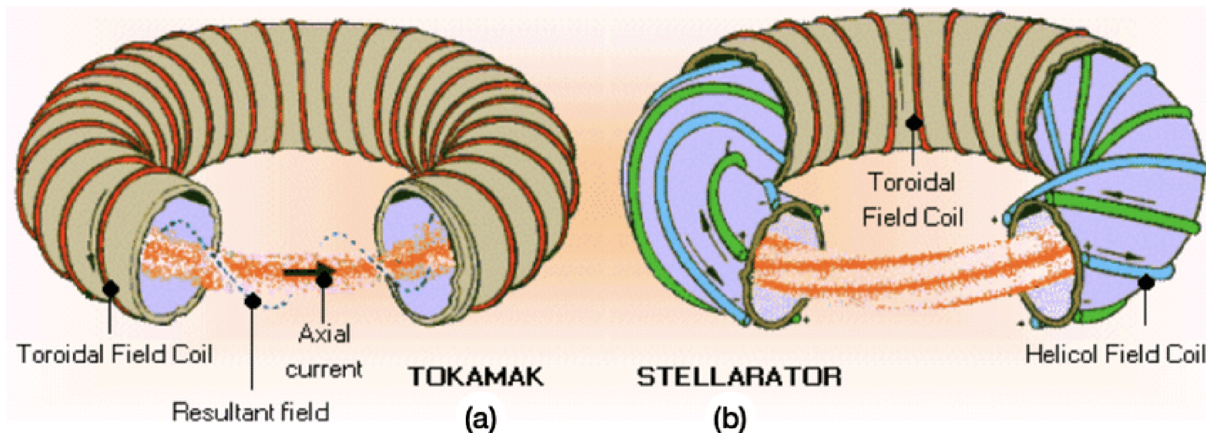


Figure 2.1: Simplified model of tokamak and stellarator [3]

The biggest nuclear fusion project is ITER [1], a project led by 33 science-leading countries with the objective of building an experimental tokamak in France. The machine aims to demonstrate that efficient fusion is possible. The Q factor is the ratio of produced energy to consumed energy of the process. ITER aims to generate up to ten times more energy than used to start the process, $Q = 10$ [4]. It could lead to a revolution in energy production, as it could be used for clean, secure, efficient, and (practically) unlimited generation [5].

Regarding fusion history, the first nuclear fusion experiment with the D-T reaction was performed on the Joint European Torus (JET) in 1997, reaching a Q factor of 0.67 [6]. Recently, the Tungsten Environment in Steady-state Tokamak (WEST) tokamak in France achieved stable fusion during 22 minutes, establishing a record in the nuclear science field [7].

In these kinds of experiments, millions of signals need to be monitored, processed, and acted upon in real time. Therefore, state-of-the-art acquisition hardware and software are designed and manufactured specifically for this project. Some examples include acquisition boards, the Control, Data Access and Communication (CODAC) software [8], the Experimental Physics and Industrial Control System (EPICS) system [9], or real-time applications.

Figure 2.2 shows a section of the ITER tokamak assembly, and Figure 2.3 shows the metal structure of the Tokamak coils, with a person for size reference.

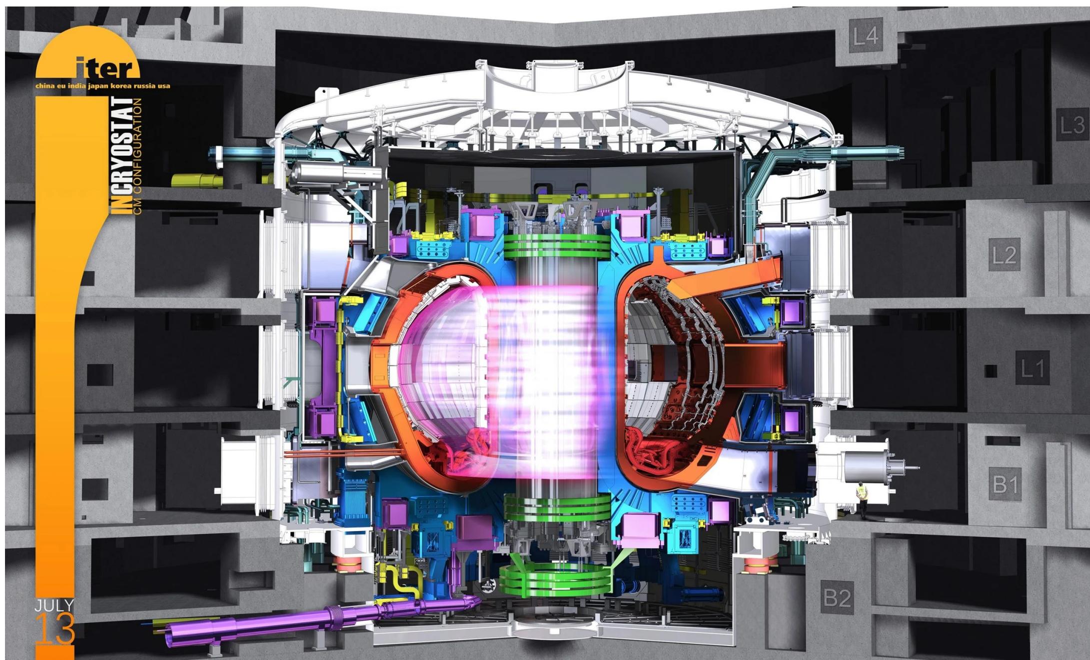


Figure 2.2: ITER Tokamak [1]

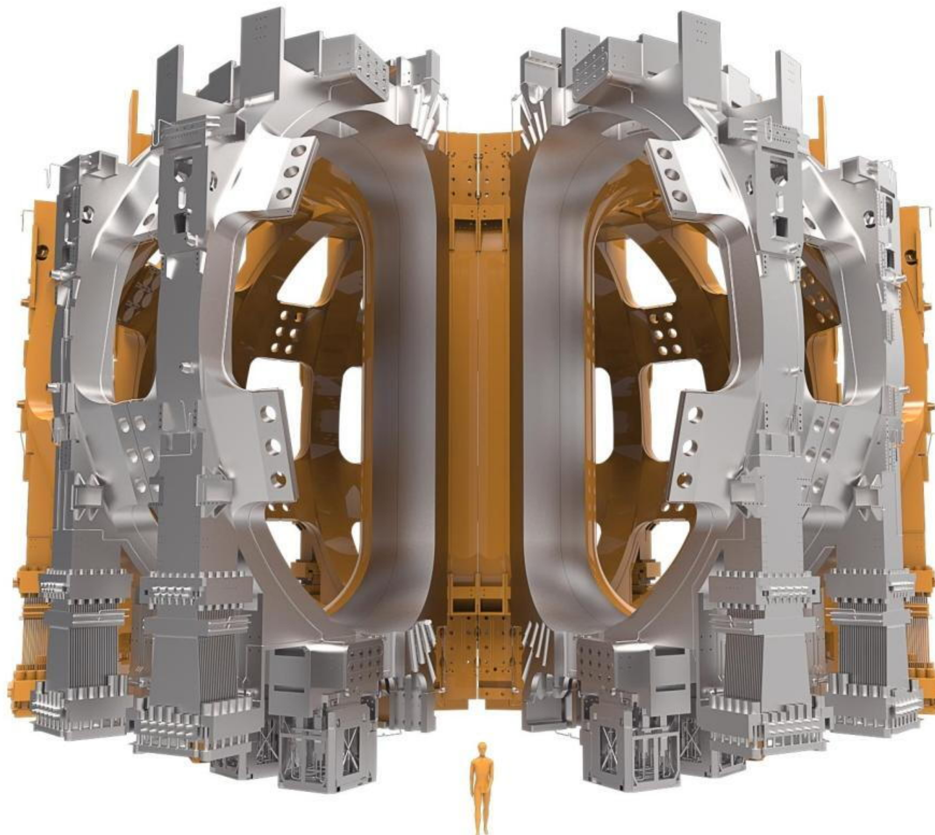


Figure 2.3: ITER coil structure [10]

2.2 FPGA devices

An FPGA is a programmable logic device. It is composed of discrete logic cells that can be configured to perform any digital logic function. These logic cells have an interconnection layer to combine them into complex logic designs. It also includes auxiliary blocks, such as multipliers, Input/Output (I/O) blocks, and memory blocks.

Each logic cell is composed of a Look-Up Table (LUT) to generate combinational logic and Flip-Flops (FFs) to implement sequential logic or memory elements. Figure 2.4 shows an image of its generic architecture, with a zoomed portion of the interconnection matrix and one combinational unit.

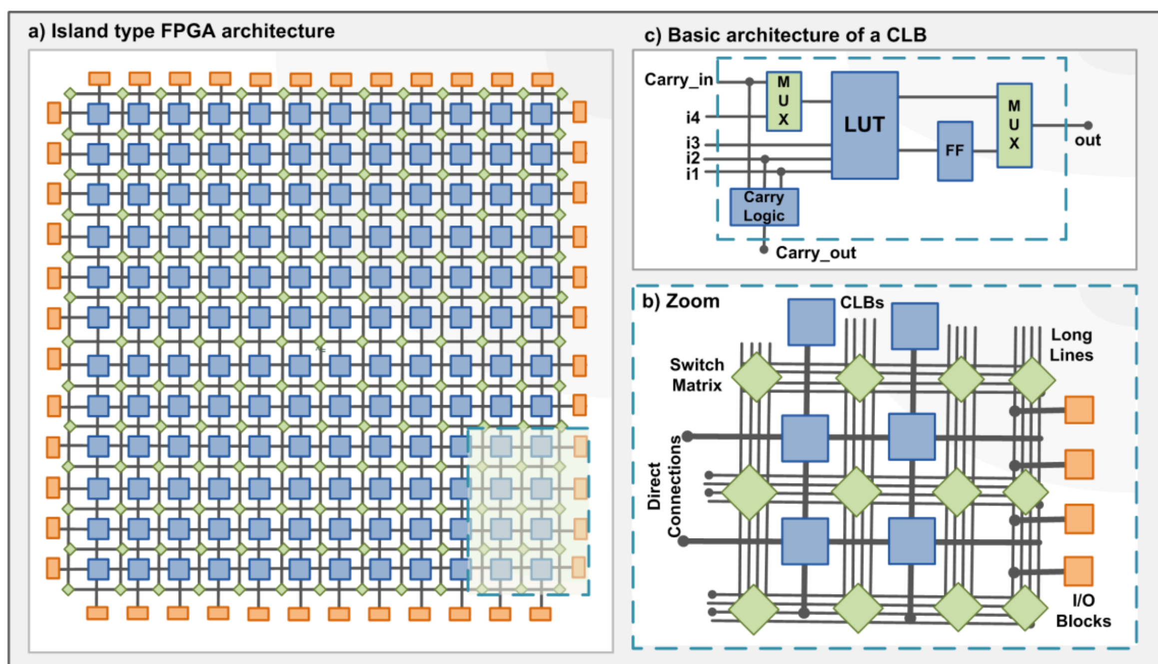


Figure 2.4: Generic FPGA architecture [11]

Compared to a Central Processing Unit (CPU), an FPGA offers increased parallelization capabilities and lower power consumption, resulting in higher performance in parallelizable algorithms. It also provides reconfiguration capabilities, as opposed to an Application Specific Integrated Circuit (ASIC) [11]. Figure 2.5 shows a compares the three types of devices based on their flexibility, performance, power consumption, programmability, and Non-Recursive Engineering (NRE).

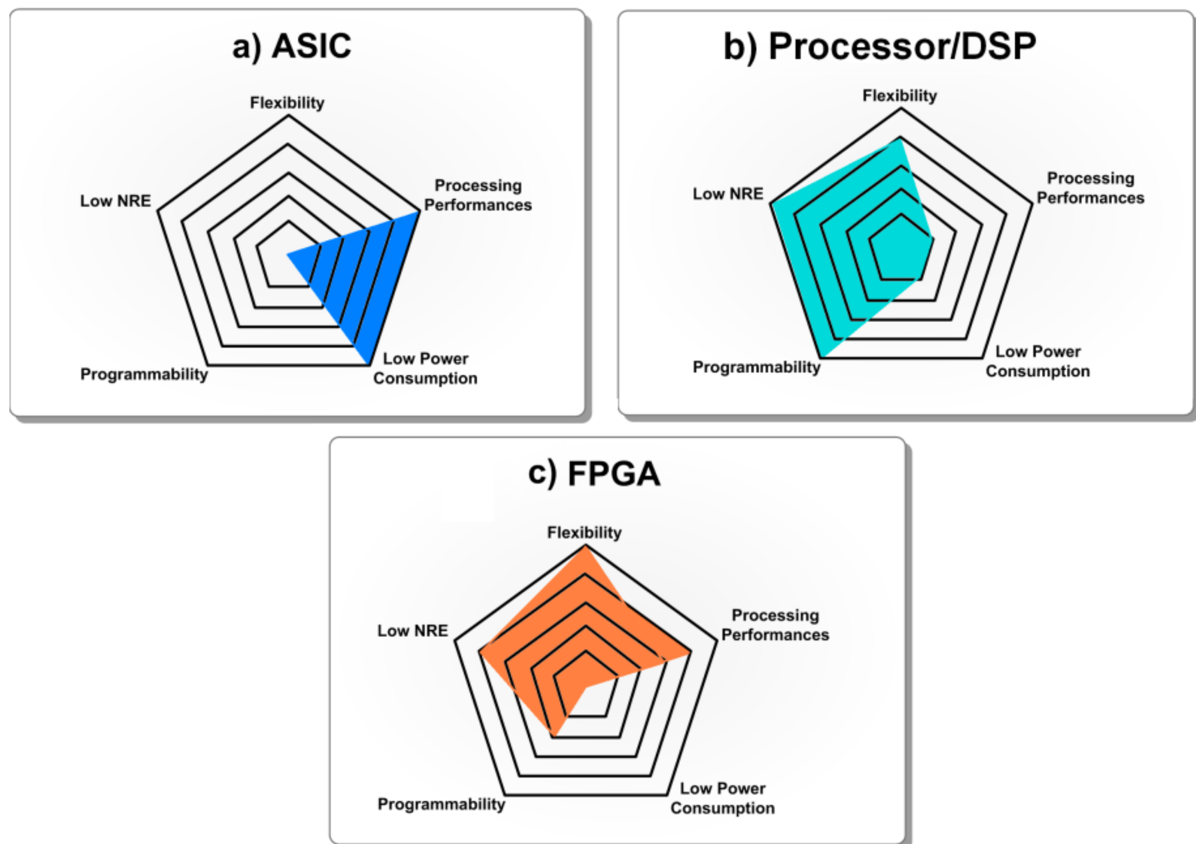


Figure 2.5: Comparison between ASIC, CPU and FPGA [11]

There are two leading brands in the FPGA market. On the one hand, Altera offers products from the low-cost Cyclone to the high-performance Agilex family. On the other hand, Xilinx ranges from the low-cost Artix family to the high-performance Versal. Each brand offers its own closed-source development tools under commercial license for the whole development cycle. Intel purchased Altera in 2015 [12] and Xilinx was purchased by AMD On 2022 [13]. Other brands on the market include Lattice Semiconductors [14], Microchip [15], or GOWIN Semiconductor [16].

Regarding Xilinx, the Artix 7 family contains the cost and transceiver-optimized FPGAs. This family of devices comprises eight low-cost devices (compared to other Xilinx products) with logic cell counts from 12800 to 215360. They include gigabit transceivers for operation with high-speed interfaces such as PCIe [17].

2.3 DAQ solutions

DAQ solutions offer a unified architecture for signal sampling, processing, and generation. These solutions are generally used to acquire signals, perform computations, and send the results to either a computer or an analog output.

These devices are widely used in applications such as Big Science, where signals must be sampled and processed with high frequencies and precision. This produces an enormous amount of information that traditionally has been stored and processed *off-line*. DAQ products allow real-time processing and data discard or compression that, although possible on a traditional CPU, would carry big power consumption and high latencies due to information transfer.

DAQ solutions are widely used in fusion experiments such as ITER in order to acquire millions of signals around the tokamak and react as fast as possible to any event produced in the plasma.

2.3.1 RIO devices

Some industrial examples of DAQ solutions are the National Instruments RIO family. This family of devices is composed of an FPGA with reconfigurable I/O capabilities. These FPGA-based devices allow the implementation of any kind of measurement logic with the card's peripherals. These devices are connected to a computer with a variety of interfaces, such as PCIe. [18]

This product family is divided into two lines. The CompactRIO line offers a flexible approach to board peripherals with a chassis and peripheral modules [19] while the FlexRIO line offers faster communication and processing thanks to the PCIe eXtensions for Instrumentation Express (PXIe) interface [20]. Figure 2.6 shows an example of a PXIe chassis with a FlexRIO device.

Figure 2.6: Example of PXIe chassis with FlexRIO device[21]

One problem with National Instruments products is the closed nature of their products. The products are not interoperable with any third-party hardware and the firmware must be designed with their own closed-source paid tools, such as LabVIEW. Although some open-source abstractions have been developed [22], the base drivers remain closed source.

This closed-source approach has become a problem from the obsolescence point of view because of the deprecation of the NI-9159 chassis, a CompactRIO device widely used in the industry, deprecated by National Instruments [23].

2.3.2 MicroTCA

Micro Telecommunications Computing Architecture (µTCA) is an open standard that defines a modular architecture for computer systems. It is based on modular cards that provide specific functionality and the infrastructure to interconnect and govern them [24].

A µTCA system is composed of a set of Advanced Mezzanine Cards (AMCs) that are interconnected with a PCIe network. These cards are controlled by a µTCA Carrier Hub and powered with a central power module.

This standard does not define the specific implementation of the cards, just the mechanical and electrical aspects of them. Therefore, specific boards for DAQ applications, such as digital and analog inputs and outputs or FPGA modules can be found. Figure 2.7 shows an example of a µTCA chassis.

Figure 2.7: Example of µTCA chassis [25]

2.4 Communication interfaces

DAQ devices generally acquire samples, perform some kind of computation over them, and send the result to a computer. This communication with the computer can be performed with different types of interfaces.

2.4.1 USB

Universal Serial Bus (USB) is the most widespread electronic device interface, used for both power and communications. Figure 2.8 shows the most common USB connectors. It also defines versions from 1.0 (1996) to USB4 2.0 (2022) based on the speed and capabilities of the connections [26].

It is a hot-swappable interface, meaning that devices can be added and removed at runtime without rebooting. USB 2.0 presents a maximum of 60 MB/s, which is used on the National Instruments USB devices [27].

Figure 2.8: USB connectors [28]

2.4.2 PCIe

PCIe is the protocol used on all computer motherboards to interconnect the CPU with its peripherals. It is an evolution over the old PCI interface. The devices are organized with switches and point-to-point connections, creating a network structure. This protocol is implemented with PCIe lines, with a scalable number of lines per connection [29].

The interface has been developed in generations, with each generation doubling the bandwidth of the previous one. Generation 1.0 started with a bandwidth of 500MB/s up to Generation 7.0 with a 32GB/s speed on an x1 connector. [30].

Table 2.1: PCIe generation speed [30]

Specifications	x1	x2	x4	x8	x16
2.5 GT/s (PCIe 1.x +)	500 MB/s	1 GB/s	2 GB/s	4 GB/s	8 GB/s
5.0 GT/s (PCIe 2.x +)	1 GB/s	2 GB/s	4 GB/s	8 GB/s	16 GB/s
8.0 GT/s (PCIe 3.x +)	2 GB/s	4 GB/s	8 GB/s	16 GB/s	32 GB/s
16.0 GT/s (PCIe 4.x +)	4 GB/s	8 GB/s	16 GB/s	32 GB/s	64 GB/s
32.0 GT/s (PCIe 5.x +)	8 GB/s	16 GB/s	32 GB/s	64 GB/s	128 GB/s
64.0 GT/s (PCIe 6.x +)	16 GB/s	32 GB/s	64 GB/s	128 GB/s	256 GB/s
128.0 GT/s (PCIe 7.x +)	32 GB/s	64 GB/s	128 GB/s	256 GB/s	512 GB/s

Figure 2.9 shows typical PCIe x4 and x16 connectors on a computer motherboard.

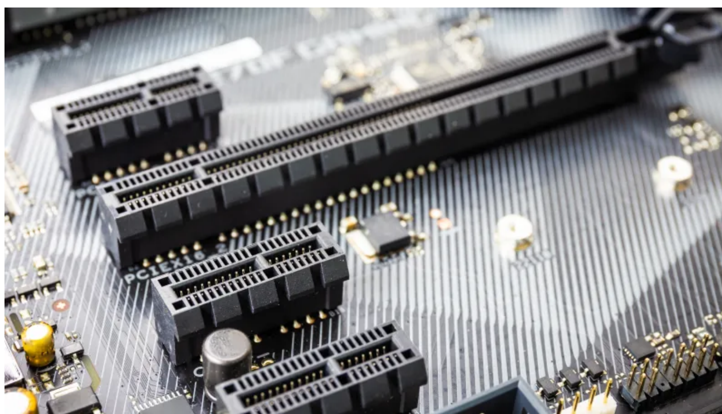


Figure 2.9: PCIe connectors [31]

2.4.3 PXIe

PXIe is not only an interface but a platform specification by the PXI Systems Alliance used to define systems for measurement, test, and data acquisition. It is composed of a set of standards for devices that include communication via CompactPCI Express. It is based on PCIe and it shares both the maximum speed and the non-hot-swappable characteristics [32].

The hardware architecture is based on three elements [33]:

- Chassis: Housing for the rest of the components. It contains a motherboard that interconnects all of them and performs some timing and synchronization operations.
- Controller: Embedded computer or link to an external computer to run an operating system and communicate with the other modules.
- Modules: Generic cards that connect to the chassis to provide functionality to the device. For example, digital I/O cards and analog input cards.

The standard also defines the software tools to be used with these devices. Some examples are Virtual Instrument Software Architecture (VISA) drivers, LabVIEW, MATLAB, etc.

Figure 2.10 contains an example of a PXIe device, where the three elements are identified.

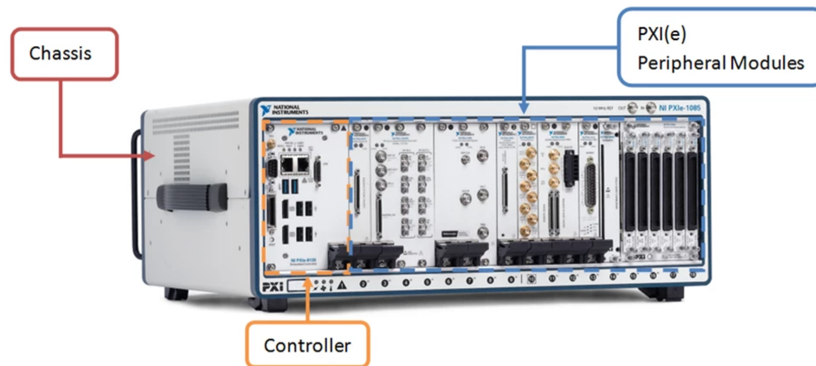


Figure 2.10: PXIe architecture example [34]

2.5 System implementation languages

2.5.1 Hardware Description Languages

Reprogrammable digital logic systems are modelled in Hardware Description Languages (HDLs). The two most popular ones are VHSIC Hardware Description Language (VHDL) and Verilog. VHDL is a language designed by the United States Defense Department in 1981 with a syntax based on the Ada programming language [35]. On the other hand, Verilog was standardized in 1991 by Cadence [36] based on the C programming language.

These languages are used to design systems on the Register-Transfer Level (RTL). This is a very low modelling level, very close to direct logic modelling. The logic is modelled as an information flow between registers, with the corresponding operations in between. The most common tool used in this design is the Finite State Machine (FSM), which can be directly mapped to register states. The desired behavior is mapped to some state set, and the desired output is derived from that state (and optionally the inputs of the circuit) [37].

Both languages can also be used on the verification stage of the workflow, creating test benches where the correct behavior of the models is checked.

However, these languages only allow for directed tests, where the test procedure is manually generated, depending on the designer to find and test all design functionality and edge cases.

To avoid this situation, Hardware Verification Languages (HVLs) such as Vera were created. These languages allow for a higher level of abstraction with tools such as object orientation or randomness to improve the verification procedures [38]. Multiple vendors such as Altera or Xilinx created their own independent tools. These tools were combined and standardized in a modern revision of Verilog called SystemVerilog, the first Hardware Description and Verification Language (HDVL) [39].

Listing 2.1 and Listing 2.2 show examples of 8-bit counters in both languages.

Listing 2.1: Example counter on VHDL

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_unsigned.all;
4
5  entity counter is
6      port (
7          clk: in std_logic;
8          cnt: out std_logic_vector(7 downto 0);
9      );
10 end counter;
11
```

```
12 architecture rtl of counter is
13 begin
14    process (clk) is
15    begin
16       if rising_edge(clk) then
17          cnt <= cnt + 1;
18       end if;
19    end process;
20 end rtl;
```

Listing 2.2: Example counter on Verilog

```
1 module counter (input logic clk, output logic[7:0] cnt);
2    always_ff @ (posedge clk) begin
3       cnt <= cnt + 1;
4    end
5 endmodule
```

2.5.2 High level languages

HDLs are very useful when the desired behavior can be directly implemented with the data-flow or behavioral methodology. However, complex algorithms are not easy to convert to these methodologies. Therefore, the high-level languages were designed as an abstraction over HDLs to ease the design and synthesis of algorithms.

They are designed to be very similar to regular programming languages, allowing even non-specialized people to adapt algorithms to synthesizable logic. Some examples of these languages are OpenCL and High-Level Synthesis (HLS).

OpenCL is a framework for application acceleration [40]. This standard is open and widely used by industry leaders. It is mainly used in hardware offloading of algorithms, where code running on the CPU transfers information to an FPGA. The FPGA contains a hardware implementation of an algorithm that processes the data and sends the results to the CPU.

OpenCL is based on C with a C++ extension, making it much more accessible to programmers than HDLs. The language can also be used for acceleration on other devices such as Graphics Processing Units (GPUs), Tensor Processing Units (TPUs) or even custom hardware. Figure 2.11 shows the main OpenCL use case architecture.

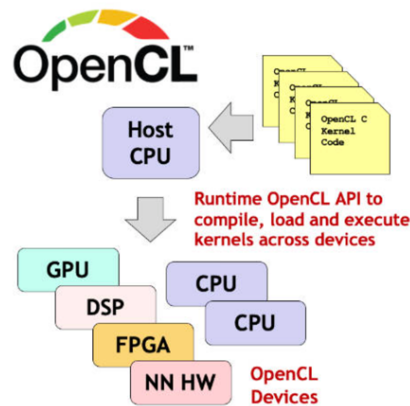
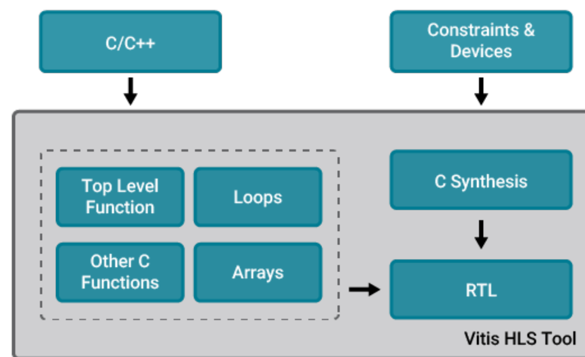


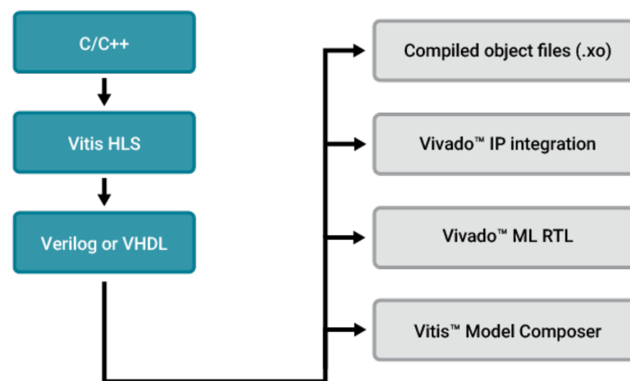
Figure 2.11: OpenCL architecture [40]

HLS is a programming model designed by Xilinx/AMD to allow conversion of C and C++ code to RTL code [41]. It also provides tools to simulate and verify implementation and generation of Intellectual Property (IP) cores to use them as parts of a larger project. This flexibility makes it a very versatile tool, but it can only be used for Xilinx devices. Figure 2.12 shows the workflows for two use cases.

The implementation of HLS is based on tasks (C functions) that take special data types as parameters (vectors or streams) and are synthesized into RTL processes. It includes some techniques for parallel programming, such as loop unrolling or pipelining, implemented as C preprocessor `pragmas`.



(a) HLS export to RTL tool chain



(b) HLS export to IP tool chain

Figure 2.12: HLS methodologies [41]

2.6 On-Chip communication

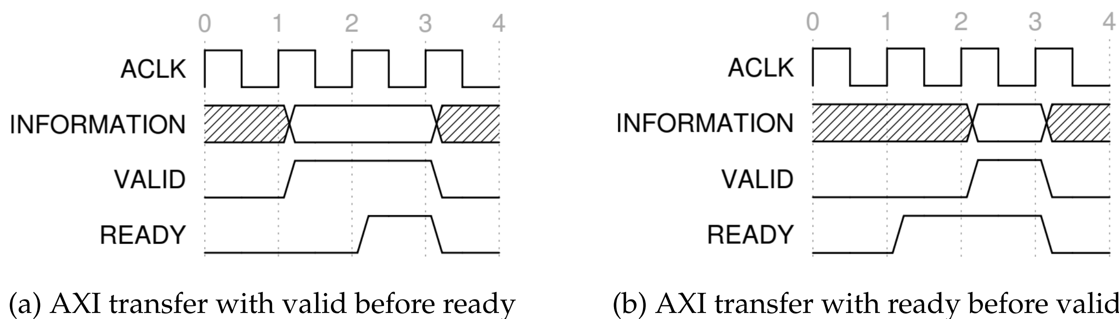
Generally, digital designs are divided into different hardware blocks. Each block performs an independent task, contributing to the desired global behavior. They can be implemented, as seen in section 2.5, with RTL languages, high-level languages, or even be third-party IP cores whose code is almost never available.

This variety of possibilities creates the need for standardized interfaces for information exchange between modules. ARM designed the Advanced Microcontroller Bus Architecture (AMBA), a family of open standards for the management of blocks in a System-on-Chip (SoC). They are royalty-free and platform-independent standards widely used in the industry [42]. It contains many data exchange protocols, such as the ones explained in the following subsections.

2.6.1 AXI

Advanced eXtensible Interface (AXI) is an on-chip communication standard for high-performance and high-bandwidth systems. It is a point-to-point protocol with a master-slave design [43]. It defines an interface based on different control and data channels and the protocol to send information on those channels.

The protocol defines a handshaking algorithm for unidirectional information transfer using three signals (apart from the system clock): `ready`, `valid` and `data`. The transmitter puts the information on the `data` signal and sets the `valid` signal high. Independently, the receiver sets the `ready` signal high whenever it can read information. If both `valid` and `ready` signals are active on the same clock cycle, the information is marked as transferred and the next data can be sent. Figure 2.13 shows two examples of information transfers with this handshake.



(a) AXI transfer with valid before ready (b) AXI transfer with ready before valid

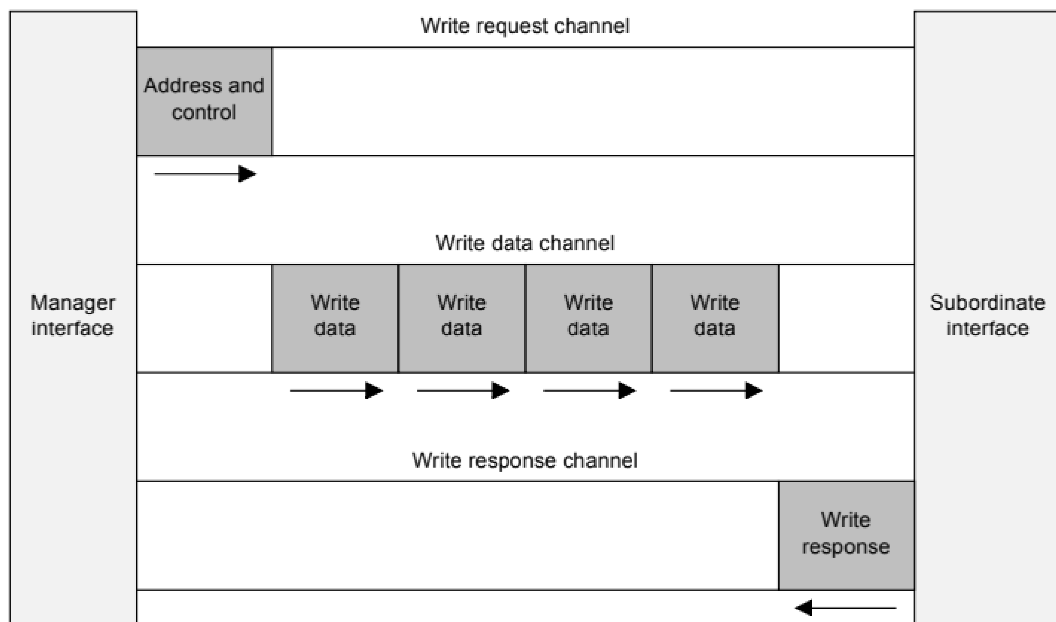
Figure 2.13: Examples of AXI information transfers [43]

The standard also defines the interface for memory-mapped transactions, based on five channels. Each one of these channels uses the handshaking algorithm presented previously to send information:

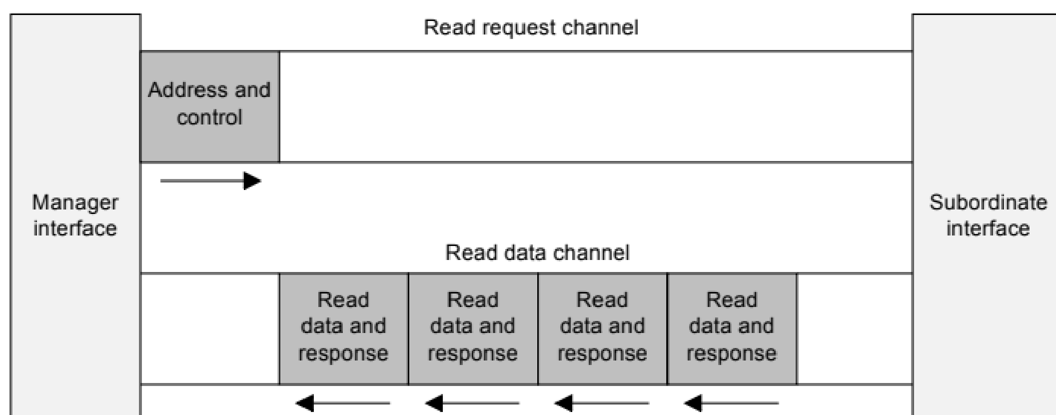
- Write request channel: Used by the master to select the address and send control in write operations.

- Write data channel: Used by the master to send the information in write operations.
- Write response channel: Used by the slave to send acknowledgments or errors in write operations.
- Read request channel: Used by the master to request the address to read and send control in read operations.
- Read data channel: Used by the slave to send the requested information.

Figure 2.14 shows the read and write sequences on this interface.



(a) AXI write sequence



(b) AXI read sequence

Figure 2.14: Read and write sequences on AXI interface [43]

The AXI standard defines burst memory accesses. AXI-Lite is a reduced standard

equivalent to AXI without burst accesses.

2.6.2 AXI Stream

AXI stream is a unidirectional data transfer protocol. It is based on the AXI information transfer handshake but only applied to one information channel without associated control channels. The information is transferred as a stream without addressing, *look-ahead* (reading data without consuming it) or *look-back* (reading already consumed data)[44].

It includes additional signals to control the data channel. For example, the `keep` signal is used to indicate which bits of the data sent contain real information and which ones are padding to the data size.

AXI stream is a very versatile protocol because it implicitly performs throughput adaptation. Both ends need to signal when data is valid and ready to be read, so transactions are automatically adjusted to the slower end. On the other hand, if both ends have the same speed, transactions can be performed up to clock speed if both control signals are kept high.

2.7 Related work

Ruiz *et al.* Ruiz *et al.* presented a methodology for using DAQ boards based on an FPGA for hardware acceleration, successfully accelerating a MATLAB algorithm to be able to run in real time, alleviating CPU load. The algorithm is a disruption predictor on the JET fusion device.

In [46], Astrain *et al.* presented a standard methodology for FPGA-based devices with OpenCL on FPGAs. This methodology is then implemented for applications such as neutron/gamma discrimination [47] or hot spot detection [48].

Patel *et al.* [49] developed a real-time measurement system with an FPGA to capture the electron density during plasma discharge inside a tokamak, with National Instruments RIO devices. The samples are acquired in the FPGA and sent to MATLAB to process and show the results to the user.

Regarding the high-level implementation of algorithms in real-time acquisition systems, Firmansyah and Yamaguchi [50] presented a similar architecture for data acquisition, processing, and generation using OpenCL for algorithm implementation and global memory for communications for radar and software-defined radio applications.

Ahmad *et al.* [51] showed an example of hardware acceleration with an FPGA and HLS to sample and process data from a pressure sensor matrix, showing the power of HLS to simplify algorithm implementation.

Chapter 3

Implementation

3.1 Resources

3.1.1 Hardware resources

The core of this project is the TMPE627 [52], a PCIe Mini card based on the Xilinx Artix-7 7A50T FPGA [53]. It also features an 8-channel Analog-to-Digital Converter (ADC), a 4 -hannel Digital-to-Analog Converter (DAC), a PCIe interface, and I/O pins. A picture of the board and its block diagram can be seen in Figure 3.1.

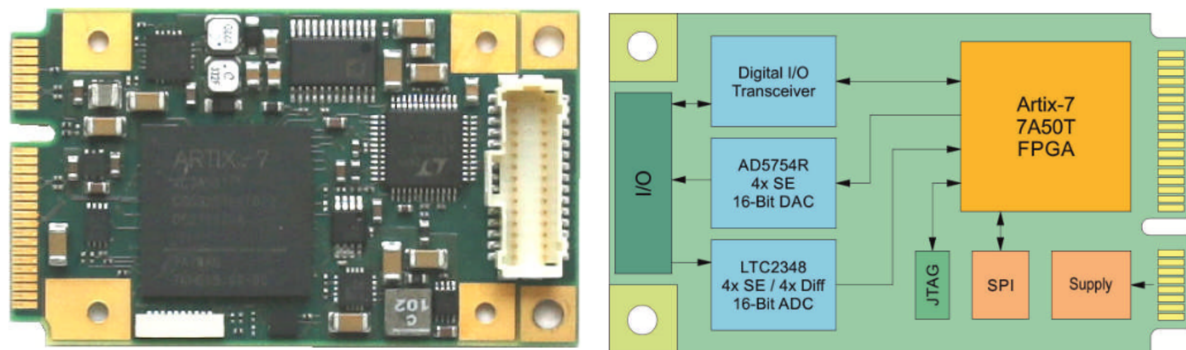


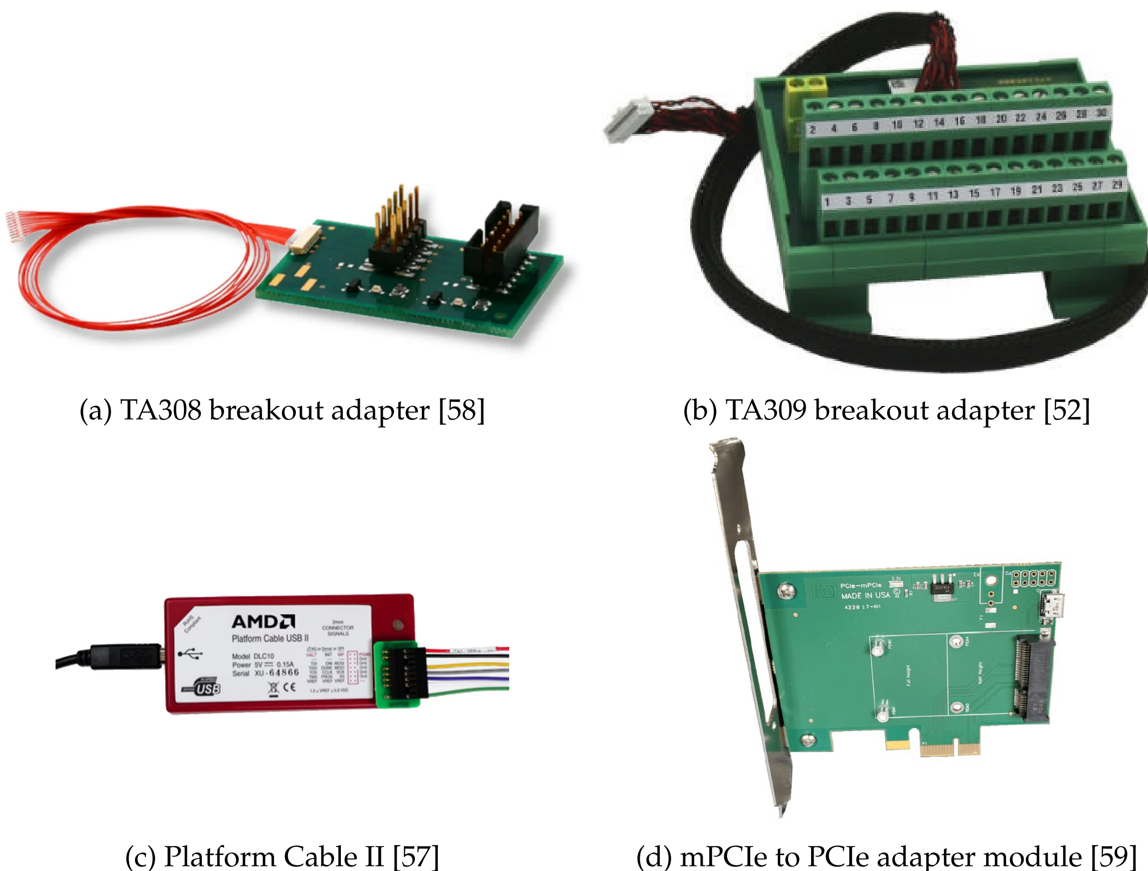
Figure 3.1: TMPE627 picture and block diagram [52]

This is a very versatile card because of its interface and peripherals. The FPGA device features around 50000 logic elements, 600 Kb of distributed Random Access Memory (RAM), 2700 Kb of block RAM, and 120 DSP slices, placing it in the middle of the Artix-7 family.

It features an LTC2348-18 [54], an 18-bit, 200ksps, differential, 8-channel ADC with Serial Peripheral Interface (SPI) or Low-Voltage Differential Signaling (LVDS) interface. The DAC is an AD5754R [55] that features 4 channels with 16-bit, 10 μs settling time and an SPI interface.

The board is installed inside the development computer with a Mini-PCIe to standard PCIe adapter, so to use the I/O pins of the board, the TA309 [56] breakout adapter must be used. It can be seen in Figure 3.2b. This board allows access to the ADC inputs, DAC outputs and some FPGA pins.

On the other hand, the design is loaded into the FPGA with a Joint Test Action Group (JTAG) interface. The interface is accessed with the Xilinx Platform Cable II USB to JTAG adapter [57] and the required TA308 Cable Kit for Modules with XRS Debug Connector [58]. The TA308 can be seen in Figure 3.2a and the Platform Cable II in Figure 3.2c. The adapter can be found in Figure 3.2d



(a) TA308 breakout adapter [58]

(b) TA309 breakout adapter [52]

(c) Platform Cable II [57]

(d) mPCIe to PCIe adapter module [59]

Figure 3.2: Other hardware components used

The project has been developed in a computer with an Intel Core i7-4790 (4 cores, 8 threads, 3.6 GHz) CPU, 32 GB of DDR3 RAM, and a 1 TB SSD. The board is plugged into one of its PCIe slots with the adapter board, as can be seen in Figure 3.3.

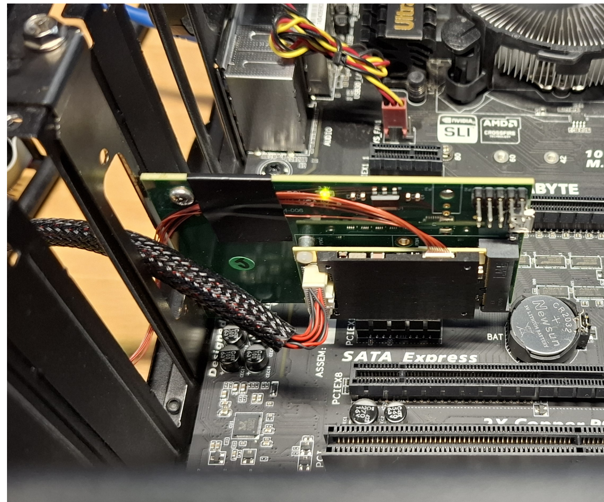


Figure 3.3: TMPE627 plugged into the motherboard

3.1.2 Software resources

This project has been developed with the Xilinx tool chain, composed mainly of Vivado and Vitis. The Vivado Design Suite has been used for HDL development while Vitis has been used for the implementation of the HLS algorithms. The software has been developed using Visual Studio Code, Python 3.9.21, and g++ 11.5.0, running on a Rocky Linux 9.5 operating system.

3.2 Design architecture

The system is designed to collect samples continuously from the ADC and the PCIe Host to Card (H2C) interfaces, process them in the Digital Signal Processor (DSP) block, and send the results to the DAC and the PCIe Card to Host (C2H) interfaces. This solution provides a generic framework for the implementation of DSP algorithms in the FPGA.

The architecture has been modelled as a collection of interconnected modules. Apart from the DSP algorithm, every module is an RTL module implemented in VHDL. Figure 3.4 shows a block design of the solution, where the connections between the modules are simplified. The complete design from Vivado can be seen in Appendix A.

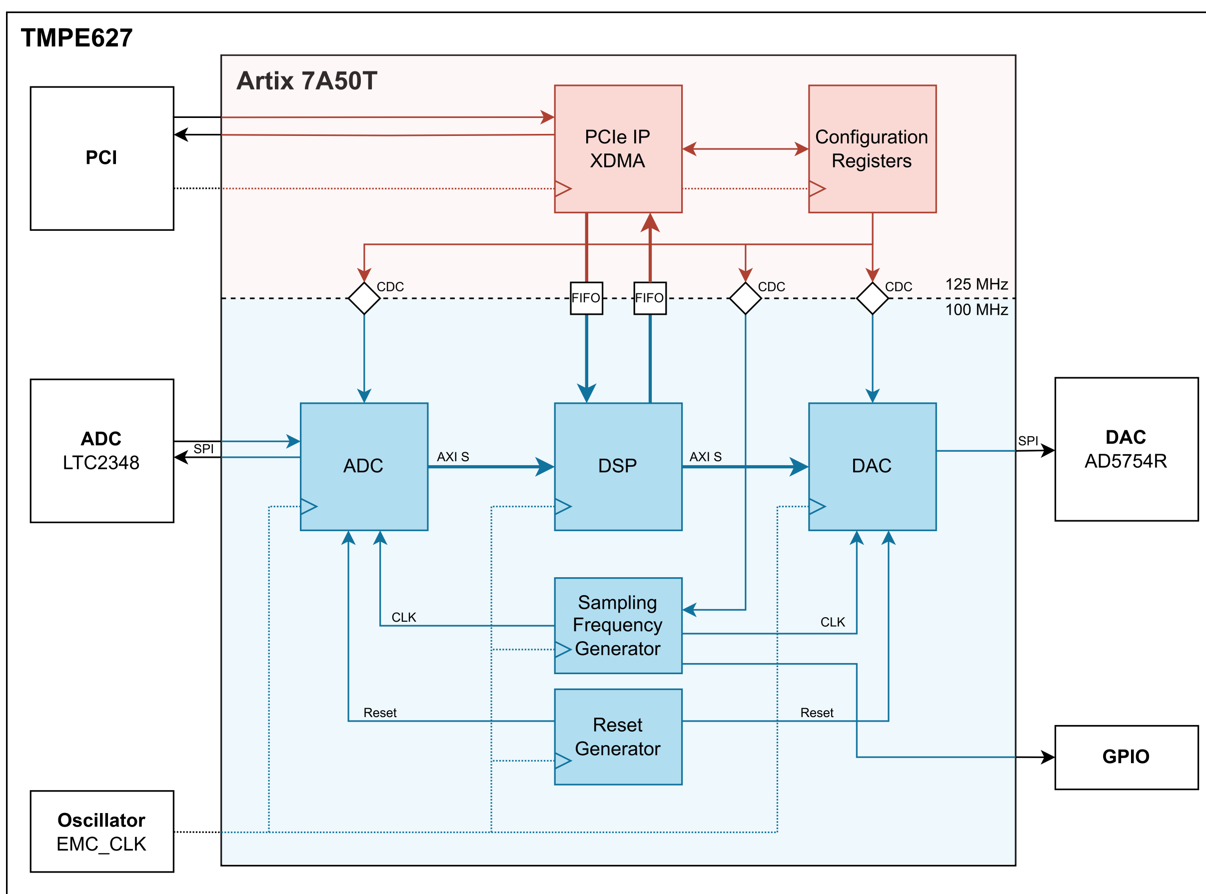


Figure 3.4: Block design of the architecture

The PCIe communication is managed by the Xilinx Direct Memory Access (XDMA) IP core (subsection 3.2.7), which interacts with the PCIe hard block via the Gigabit Transceivers of the FPGA. It communicates with the Configuration Registers module via an AXI Lite interface and with the DSP module (subsection 3.2.4) using two AXI Stream interfaces, providing Full-Duplex communication with the host computer.

The XDMA core receives the 100 MHz PCIe clock with a differential global buffer

(IBUFGDS), and uses a Phase-Locked Loop (PLL) to change it to a 125 MHz clock, which is used as the input clock for the configuration registers and on the output signals of the core. The other clock source of the design is the local oscillator of the board. It is a 100 MHz clock that is used on the rest of the modules of the design. It is used because it is a *Free-running* clock, allowing the design to keep working even when the host computer is turned off (provided that the auxiliary power is still on).

However, the division of the design into two clock domains increases its complexity, introducing the need for Clock Domain Crossing (CDC) techniques to avoid losing data and metastability. The data from the configuration registers is synchronized internally before sending it to the other modules (subsection 3.2.3), and the data from the C2H and H2C streams is synchronized with First In, First Out (FIFO) queues (subsection 3.2.7).

The ADC module (subsection 3.2.1) manages the configuration and sample acquisition of the ADC using its SPI interface, and sends the samples to the DSP module using an AXI Stream interface for each channel. It receives the configuration from the configuration registers, allowing *on-the-fly* reconfiguration.

The DAC module (subsection 3.2.2) interfaces with the DAC, sending the sample values and configuration over its SPI interface. It receives the configuration from the Configuration Registers module and the samples with an AXI Stream interface for each channel. It also allows *on-the-fly* reconfiguration.

The DSP module (subsection 3.2.4) offers a generic adapter for the implementation of DSP algorithms. It receives the ADC and H2C samples with AXI Stream interfaces, applies the processing, and outputs to the DAC and C2H AXI Stream interfaces.

The Sampling Frequency Generator module (subsection 3.2.5) generates the timing signal for both the ADC and DAC modules with a configurable frequency from the configuration registers. The Reset Generator module (subsection 3.2.6) is used to initialize the oscillator clock domain modules because it does not provide an accessible asynchronous reset signal. The output of the sampling generator is also connected to an external I/O pin to test its accuracy.

Figure 3.5 contains a summarized diagram of the data flows of the architecture.

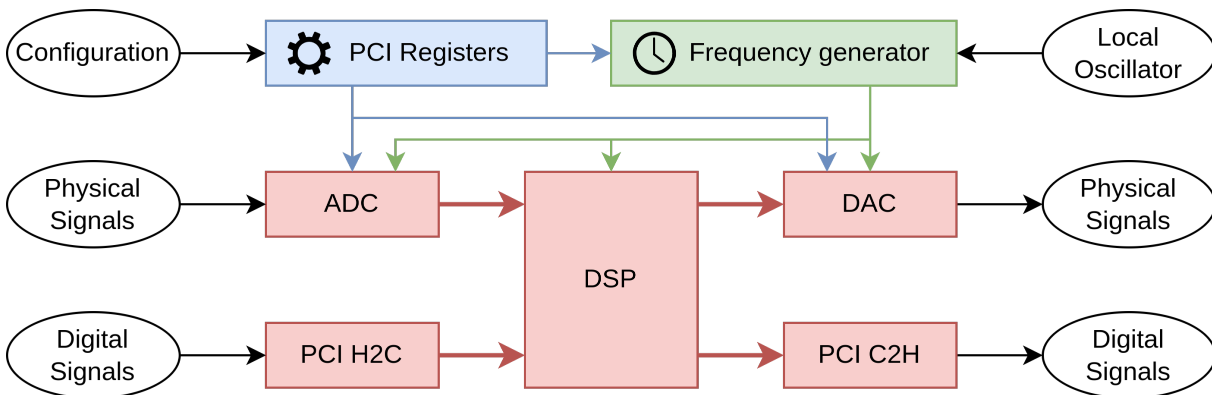


Figure 3.5: Data flows of the architecture

3.2.1 ADC module

The ADC module is responsible for managing the physical ADC on the board. It communicates with it via SPI to configure the acquisition, start the sampling, and read the results. The module takes the configuration parameters as digital signal inputs, receives the sampling clock from the Sampling Frequency Generator module, and sends the samples via AXI Stream interfaces.

The ADC offers 18-bit data, but due to the 16-bit size of the DAC output and the AXI stream data size, only the 16 most-significant bits are used. All configurable values can be seen in Table 3.1.

Table 3.1: Configuration parameters of ADC module

Configuration parameter	**Values**
Range (Softspan) CH 0	+5.12 V, +10 V, +10.24 V, ±5 V, ±5.12 V, ±10 V, ±10.24 V
Range (Softspan) CH 1	+5.12 V, +10 V, +10.24 V, ±5 V, ±5.12 V, ±10 V, ±10.24 V
Range (Softspan) CH 2	+5.12 V, +10 V, +10.24 V, ±5 V, ±5.12 V, ±10 V, ±10.24 V
Range (Softspan) CH 3	+5.12 V, +10 V, +10.24 V, ±5 V, ±5.12 V, ±10 V, ±10.24 V
Power Down	`ON`, `OFF`

The range or *Softspan* of the channels can be configured independently with unipolar or bipolar ranges of around 5 V to 10 V. It features one input SPI interface and up to eight SPI output interfaces with a common clock. Each transaction consists of a configuration write and eight sample readings at the same time. A generic transaction from the datasheet can be seen in Figure 3.6.

The TMPE627 board offers only four of these eight channels on the I/O ports. However, the datasheet presents a different numbering system that is not mentioned on the document. The channels on the ADC pins are numbered from 0 to 7, while the board user guide numbers the input (differential) pins from 1 to 4 [52]. Counterintuitively, the input pins are connected to the even pins of the ADC, as can be seen in Table 3.2, where NC stands for Not Connected.

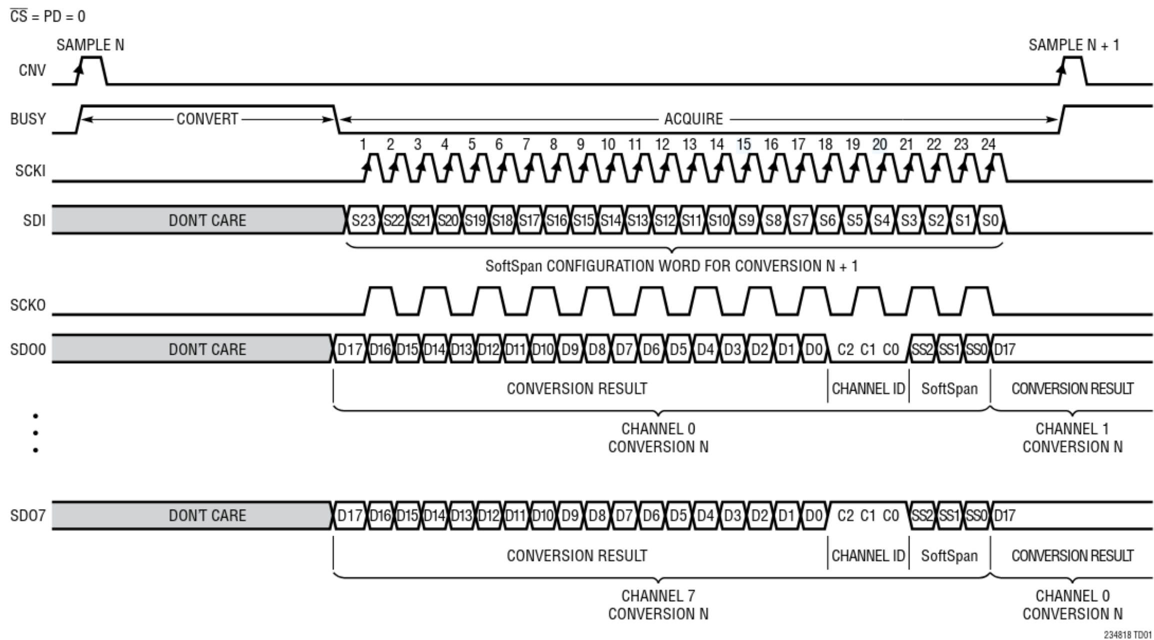


Figure 3.6: Generic ADC SPI transaction [54]

Table 3.2: TMPE627 ADC pin mapping

ADC Channel	Board pins
0	1+, 1−
1	NC
2	2+, 2−
3	NC
4	3+, 3−
5	NC
6	4+, 4−
7	NC

This module is based on a reference design provided by TEWS Technologies [60], but it has been heavily modified to accommodate it for this project. The original module is an SPI master that samples as fast as possible and only saves the last sample. It was modified to allow sample streaming, modularity, and precise timing.

The behavior is designed as an FSM that handles SPI reading and writing, sample temporization, and ADC power down. It has an initial `IDLE` state that waits until the ADC is not busy to start reading the last sample and sending the configuration for the next sample with the `TICK` and `TOCK` states, implementing SPI transactions. Then, it uses the `WAITING` state to wait for the sampling frequency generator signal (called `ena_conv` on this module) to start the conversion and wait in the `CONVERT` state for it to finish. Lastly, the `ADC_PD` state checks whether the device should be powered down and halted

in that state, or the cycle should be repeated from the beginning. Figure 3.7 shows the diagram of this state machine.

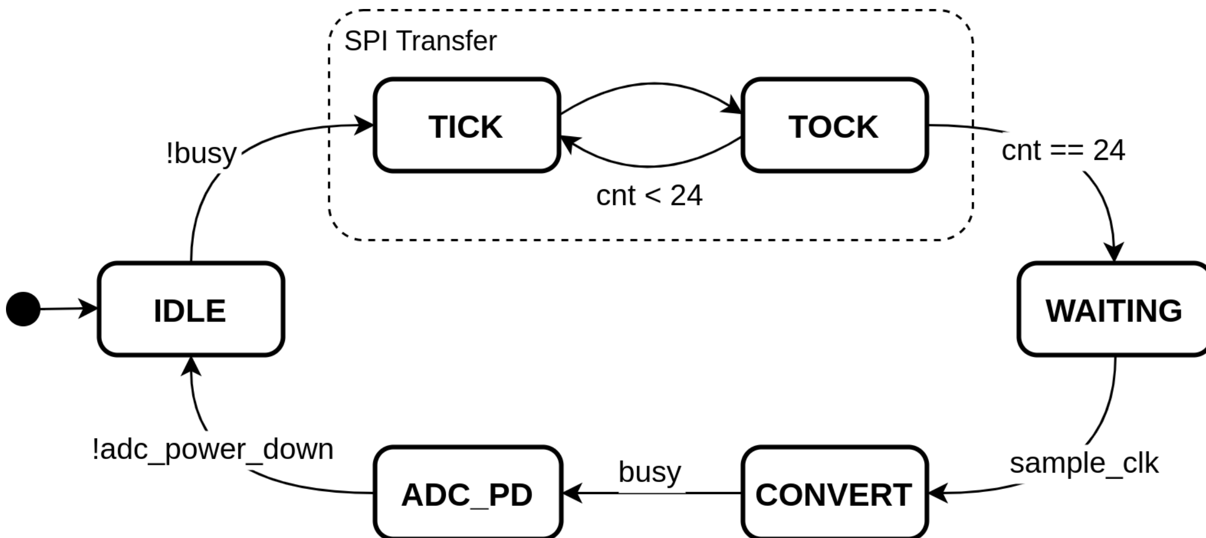


Figure 3.7: ADC state diagram

In order to increase the modularity of the design, the input SPI interfaces have been split into a smaller module called `adc_channel` that is instantiated a compilation-time configurable number of times (from 1 to 4). This reduces the design footprint if some channels are unused, increasing the free space for DSP logic. This module contains the shift register, packet processing, and AXI Streaming logic needed to read and send samples from one channel.

3.2.2 DAC module

The DAC module is responsible for communicating with the physical DAC device. The communication is based on the SPI protocol, featuring only one unidirectional communication from the FPGA to the DAC. The DAC offers a register-based structure for configuration and input of samples. As the DAC only has one SPI interface, the configuration and data transactions must happen sequentially, limiting the sampling frequency.

The DAC accepts 16-bit data, and the configurable parameters can be seen in Table 3.3. Each channel can be individually powered up or down, but the range is the same for all channels. It allows a unipolar or bipolar range with options from 5 V to 10.8 V.

Table 3.3: DAC configuration parameters

Configuration parameter	Values
Power down	`ON`, `OFF`
Negative number encoding	`2's Complement`, `Offset Binary`
Clear selection	`Clear to 0`, `Clear to half/bottom range`
Thermal Shutdown Enable	`ON`, `OFF`
Range	+5 V, +10 V, +10.8 V, ±5 V, ±10 V, ±10.8 V

This module is also based on the TEWS Technologies reference design [60], adding modularity and precise timing capabilities. It takes the configuration parameters as signal inputs and up to four AXI Streaming interfaces for the samples to be output through the channels.

The module is designed in two layers. The first one is a low-level physical layer that handles all the communication with the module using the SPI interface, abstracting the high layer from the transaction logic. This low-level layer was kept intact from the reference design because it proved complete and useful.

The other layer is a high-level layer that governs DAC configuration and sample writing. It was modified to adapt the connection to the different modules of the design, add precise timing, and channel modularity. It is based on an FSM that first reconfigures the device and then sends the sample data to be output.

Figure 3.8 shows a state diagram of the FSM. Each configuration state sends the information to the corresponding configuration register and proceeds with the configuration process. Then, the `WAIT_SAMPLING_CLK` state waits for a tick from the sampling frequency generator module to handle sample timing, and the `DATA` state sends the information to all data registers for the configuration channels.

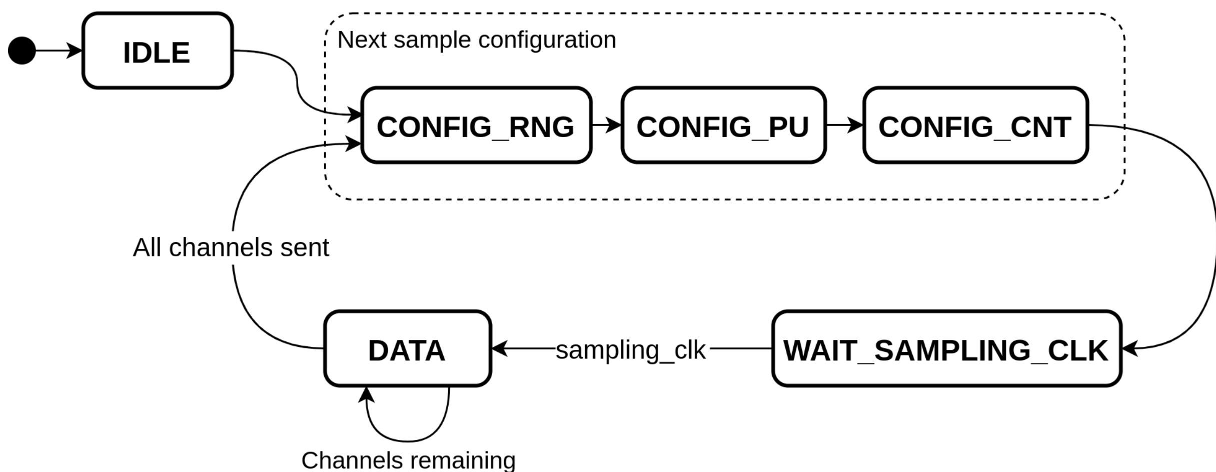


Figure 3.8: DAC state machine diagram

The number of channels is synthesis-time configurable, and although it does not affect

the design footprint, decreasing the number of channels can increase the transaction speed and allow for higher sampling frequencies.

3.2.3 Configuration registers module

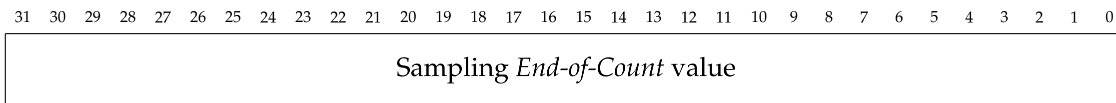
The Configuration Registers module is in charge of run-time reconfiguration of the acquisition parameters. It uses an AXI Lite memory-mapped interface to receive information from the host computer. The host computer can map this region to its memory to modify it easily.

This module has been generated with the *Create and Package new IP* option from Vivado, which generated the basic structure and most AXI Lite operations. Only the first four registers of the interface are used, but fifteen were generated in order to make this module extensible. The address map can be seen in Table 3.4.

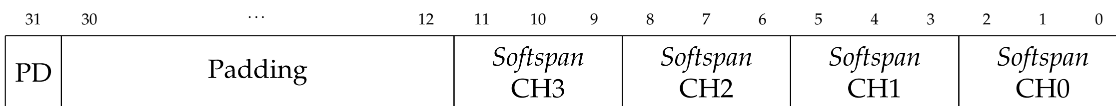
Table 3.4: Configuration registers map

Register	Address
Sampling End of Count	`0x00`
ADC configuration	`0x04`
DAC configuration	`0x08`
C2H enable	`0x0C`

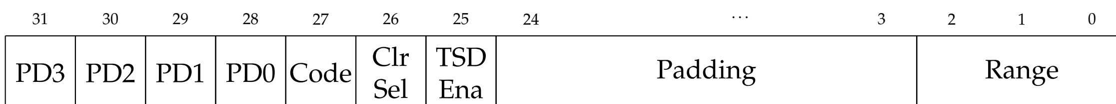
Figure 3.9 shows the content of each of these registers.



(a) Contents of the Sampling EOC register



(b) Contents of the ADC configuration register



(c) Contents of the DAC configuration register

(d) Contents of the C2H enable register

Figure 3.9: Content of the configuration registers

The ADC and DAC configuration parameters can be found in Table 3.1 and Table 3.3, respectively. They are equivalent to the homonymous parameters in the datasheets.

Both ADC and DAC present a three-bit range selection, but have different codes for each range. The configuration register module includes translation logic to unify the range codes between the two. The range selection bits can be seen in Table 3.5. The terms *Softspan* and range are equivalent.

It should be noted that the full range for both devices is not the same, as the ADC reaches 10.24 V and the DAC reaches 10.8 V. If both devices are used for the same signal type, either the 5.0 V or the 10.0 V ranges should be used.

Table 3.5: Range translation in registers

Register value	ADC Range	DAC Range	ADC value	DAC value
0	$+5.12\ V$	$+5\ V$	1	0
1	$+10\ V$	$+10\ V$	4	1
2	$+10.24\ V$	$+10.8\ V$	5	2
3	$\pm 5\ V$	$\pm 5\ V$	2	3
4	$\pm 5.12\ V$	$\pm 5\ V$	3	3
5	$\pm 10\ V$	$\pm 10\ V$	6	4
6	$\pm 10.24\ V$	$\pm 10.8\ V$	7	5
7	$\pm 10.24\ V$	$\pm 10.8\ V$	7	5

The data from this register is split into individual signals that are routed to the respective modules. However, this module is located on the PCIe clock domain because of the AXI interface, while the destination modules are on the *Free-running* clock domain. Therefore, CDC logic must be implemented to send the data from the fast clock domain to the slow one.

Xilinx offers the Xilinx Parameterized Macros (XPM) CDC Cores [61] to generate blocks for information transfer between clock domains. It provides seven possible variants of CDC techniques. The `async_rst`, `pulse`, `single` and `sync_rst` are designed for one-bit signals or non-data signals. The `gray` synchronizer can only handle one-bit changes on data, and the `array_single` synchronizer states that each bit of the array must be independent and have no relationship, but that is not the case. Therefore, the `handshake` synchronizer is the only one valid for this use case.

This synchronizer uses a handshaking protocol to send data between clock domains. In consequence, additional logic is needed to detect the acknowledgments and start the information transfers. As in this use case the information is sent by the host, overrides the previous information, and is not used until the next conversion, only the last valid information is required and no data streaming logic is needed. This module implements a simple logic procedure that sends the information, waits for the acknowledgment, and immediately starts another transfer upon receiving it.

The macro block is configured to use internal handshaking logic, making the implementation much easier. The design uses the `src_send` signal to start a transfer and waits for the pulse on the `src_rcv` signal, indicating a successful transmission. The input data must be registered on the transmission side to avoid data modification during transmission. Figure 3.10 shows the waveforms of this handshake, although the `data_out` signal would be synchronized with the output clock.

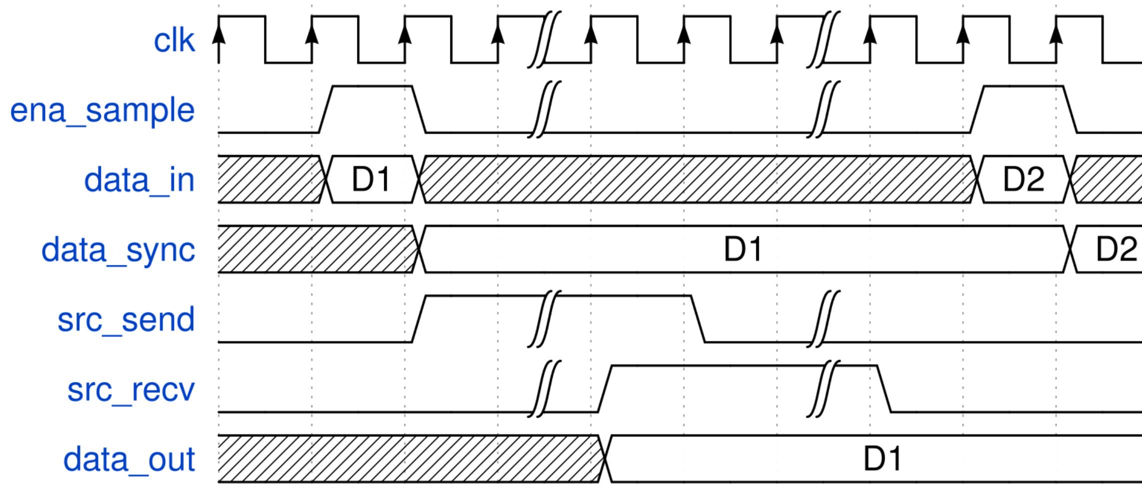


Figure 3.10: CDC register handshake

3.2.4 DSP module

The DSP module is a wrapper around the algorithm implementation. It mainly performs interface adaptation for the algorithm inputs and outputs. Some example algorithm implementations are presented in section 3.3.

To simplify the interconnection of the modules, the AXI Stream interfaces of both ADC and DAC are aggregated into one AXI Stream port. Therefore, this wrapper splits the input interface into the four channels that compose it and combines the four outputs into the aggregated stream that the DAC expects.

This module is parametrized at synthesis time with the same options as the ADC and DAC only to generate the splitting and aggregation logic necessary for implementation, allowing footprint optimization.

This module also connects the PCIe C2H and H2C AXI Stream interfaces to the algorithm, allowing data processing from and to the host. However, this module is on the *Free-running* clock domain and the XDMA IP core is on the PCIe domain. Consequently, CDC techniques must be applied to transfer information from both domains.

In order to simplify data stream synchronization, two FIFO IP cores are used [62]. They provide AXI Stream interfaces with independent clocks, a feature that makes them ideal for this application. As the AXI protocol already performs a *Valid-Ready* handshake, there is no need for memory depth in the FIFO queues. However, the minimum size allowed for instantiation is 16, so that is the one selected.

This queue depth imposes a limitation, as the C2H interface will not always be reading information, but the ADC will be generating samples continuously. This would cause the queue to fill with information from the beginning of the operation and only obtain useful information after reading 16 useless samples. In order to reduce the impact of this effect, a clock enable signal is used on the input clock of the C2H FIFO to disable it when no reading is being performed. This enable is controlled by the least significant bit of the `C2H configuration register` (subsection 3.2.3, Figure 3.9).

3.2.5 Sampling frequency generator module

The sampling frequency generator module is responsible for the acquisition timing of the ADC and DAC. It consists of an upwards, auto-resetting, 32-bit counter with a programmable maximum whose End of Count (EOC) signal is used as the trigger for conversions.

The value of EOC is selected with a configuration register, allowing for runtime frequency adjustments. The frequency in the operating region is calculated with Equation 3.1.

$$f_s = \frac{f_{CLK}}{EOC} = \frac{100\ MHz}{EOC} \tag{3.1}$$

The output of this module is also routed to an output pin of the FPGA to measure it externally, with the considerations explained in subsection 4.2.2.

3.2.6 Reset generator module

The reset generator module create a synchronous reset signal for the *Free-running* clock domain. It uses a non-resetting counter that counts up to 15 (`0xF`) and stays at that value. The EOC signal is then used as the synchronous reset because it starts low during those 15 cycles and then remains high during regular operation.

The reset time can be calculated with Equation 3.2. A feature of this module is the synchronous reset release, a requirement for some logic modules such as FIFO Queues.

$$t_{RST} = \frac{EOC + 1}{f_{CLK}} = \frac{15 + 1}{100 \cdot 10^6} = 160\ ns \tag{3.2}$$

3.2.7 XDMA IP core

The XDMA module is an instantiation of the *DMA/Bridge Subsystem for IP Express* IP core [63]. This module allows communication with the host computer through the PCIe lines using Gigabit Transceivers.

The TEWS reference design uses another PCIe IP that only handles low-level communication with the hard PCIe block of the board and requires proprietary drivers on the host, sold separately by TEWS Technologies. Therefore, the XDMA block was

selected to allow easier functionality implementation and open-source driver usage. Furthermore, the original reference design only allows memory-mapped transfers and does not support data streaming, an essential requirement of this project.

The IP core includes all the required logic to create a PCIe endpoint and communicate with the host. It receives the differential PCIe clock (after buffering it and converting it to unipolar with an `IBUFGDS`) and converts it to a 125 MHz clock for the AXI interfaces. It also generates a reset signal from the PCIe bus.

The core has been configured to use an AXI Stream interface mode, generating the C2H and H2C interfaces with an additional AXI Lite interface on another BAR for the configuration register access. The streaming interfaces are 64-bit wide, and the Lite interface is 32-bit long.

In order for this block to work with the host computer, this host must be running a Linux operating system (although non-official Windows implementations can be found) and install the XDMA IP drivers by Xilinx [64]

Once the drivers are installed, the configured interfaces can be found in the `/dev` directory on the host file system, where they can be used as character devices or memory-map them. The use of these files will be explained in section 3.6.

3.3 HLS algorithm implementation

This project offers a generic framework for the implementation of algorithms in HLS. This language is based on C++ and provides an abstraction over low-level HDL to make algorithm implementation easier. The logic is then exported as an IP core that is embedded into the wrapper (subsection 3.2.4). The requirements for the implementation are as follows:

- Four AXI Stream inputs with a 16-bit data channel (ADC inputs)
- Four AXI Stream output with a 16-bit data channel (DAC outputs)
- One AXI Stream input with a 64-bit data channel (H2C input)
- One AXI Stream output with a 64-bit data channel (C2H input)

Therefore, the top function of the algorithm code must have the prototype (the names are not mandatory) that can be found in Listing 3.1. Some directives must be used to set the interface mode of the ports and remove the control interface that is not used if the kernel is used on streaming mode (without external reset and temporization).

Listing 3.1: HLS algorithm top prototype

```
1  typedef ap_axis<16, 0, 0, 0> data_t;
2
3  typedef ap_axis<64, 0, 0, 0> pci_data_t;
4
5  void top(
6          hls::stream<data_t>& in_adc1,
7          hls::stream<data_t>& in_adc2,
8          hls::stream<data_t>& in_adc3,
9          hls::stream<data_t>& in_adc4,
10         hls::stream<data_t>& out_dac1,
11         hls::stream<data_t>& out_dac2,
12         hls::stream<data_t>& out_dac3,
13         hls::stream<data_t>& out_dac4,
14         hls::stream<pci_data_t>& in_h2c,
15         hls::stream<pci_data_t>& out_c2h
16     ) {
17         #pragma HLS INTERFACE mode=axis
              port=in_adc1,in_adc2,in_adc3,in_adc4
18         #pragma HLS INTERFACE mode=axis
              port=out_dac1,out_dac2,out_dac3,out_dac4
19         #pragma HLS INTERFACE mode=axis port=in_h2c,out_c2h
20         #pragma HLS INTERFACE mode=ap_ctrl_none port=return
21         /* Algorithm implementation */
22     }
```

One crucial consideration on HLS implementation is function differentiation. By default, calls to the same function with different arguments will be synthesized to the same logic, multiplexing the inputs and sharing memory between calls. If this is not the desired implementation, in cases such as implementing multiple blocks of the same logic, C++ templates should be used with a generic integer argument. With them, the compiler defines different functions and therefore different and independent logic blocks.

To test the capabilities of this design, three different algorithms have been implemented that will be explained in the following subsections. The code for the implementations can be found in Appendix B

3.3.1 Width adapter

The width adapter is the simplest HLS model of this project. Its function is to change the width of the data elements between the ADC/DAC 16-bit data and the PCIe 64-bit data. This model has two variants, one for long-to-short and another for short-to-long conversion.

When performing the adaptation, it is essential to consider the bit representation. The data of each sample can be interpreted as a fixed-point fractional number without integer bits. This value represents the proportional part of the encoded data's Full Scale Range (FSR). Equation 3.3 contains the conversion of a fixed-point value with module CD and sign bit S (0 positive, 1 negative) into the corresponding voltage.

$$V = (-1)^S \cdot CD \cdot \frac{V_{FSR}}{2} \tag{3.3}$$

As a consequence, the bit conversion must prioritize the most significant bits, as they carry the most information in the fixed fractional representation. Therefore, conversions from 64 to 16 bits are performed by right-shifting the number, and 16 to 64 conversions are composed of a left shift while padding with zeros.

The conversion in HLS is implemented easily, as the C++ shift operators can be used to synthesize the conversion logic. In the case of left-shifting, a static cast must be used to change the value width before shifting, to avoid a possible shift before promotion, resulting in zero.

3.3.2 FIR filter

Finite Impulse Response (FIR) filters are a type of digital filter without feedback loops. They are very stable, and can be designed with a linear phase response. Because they lack feedback loops, they can be implemented as a dot product (inner product) between a coefficient array and the history of inputs [65]. Equation 3.4 shows the general equation for an FIR filter of order N.

$$y[n] = \sum_{i=1}^{N} b_i \cdot x[n-i] = \vec{b} \cdot \vec{x}, \qquad \vec{b} = \begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_N \end{pmatrix} \qquad \vec{x} = \begin{pmatrix} x[n] \\ x[n-1] \\ \vdots \\ x[n-N] \end{pmatrix} \tag{3.4}$$

Filter design is a mathematically complex task with lots of different approaches, so filter design tools are generally used to determine the optimal configuration. In this case, TFilter [66] was used to generate the coefficients for this design. The parameters used were:

- Sampling frequency of 100 kSPS
- 32 coefficients
- Passband from 0 Hz to 5 kHz with a desired ripple of 1 dB
- Stopband from 10 kHz with a desired attenuation of -40 dB

The filter design tool generates an array of floating-point coefficients that make up the desired filter. Figure 3.11 shows the Fast Fourier transform of the filter coefficients, where it can be seen that the number of coefficients is not enough to create a filter with that attenuation, but it is very close to the design parameters.

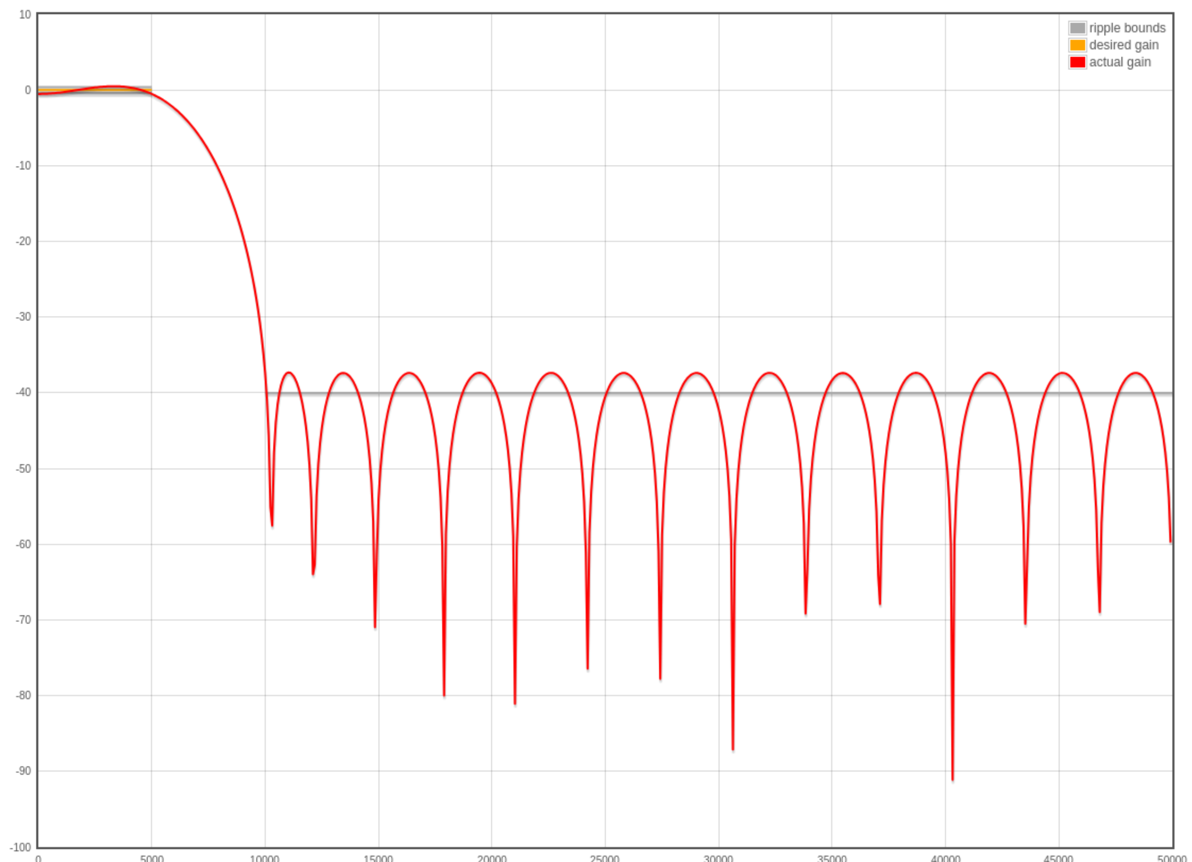


Figure 3.11: Frequency response of the designed FIR filter

This type of filter is easily modelled in HLS because it only needs a shift register to store the latest N samples and a weighted averager of the register contents. The FPGA used in this project has 25x18 multipliers [53], and the ADC and DAC data are 16 bits long ,so in order to improve precision without footprint impact, the filter coefficients used are 25-bit long fixed point numbers with one sign bit and 24 fractional bits, using one multiplier for each product.

The shift register is implemented with a delay loop, and the weighted average is implemented with a loop that accumulates the partial products that make up the scalar product of the vectors.

An important concept in HLS design is loop unrolling. A n iteration loop can be synthesized to recursive logic that takes at least n cycles to operate, but this can be optimized. A loop can be unrolled, meaning that the logic of the loop body is replicated to make loop iterations concurrent. The loop can be partially or completely unrolled, dealing with a footprint against latency tradeoff. Figure 3.12 shows different degrees of unrolling.

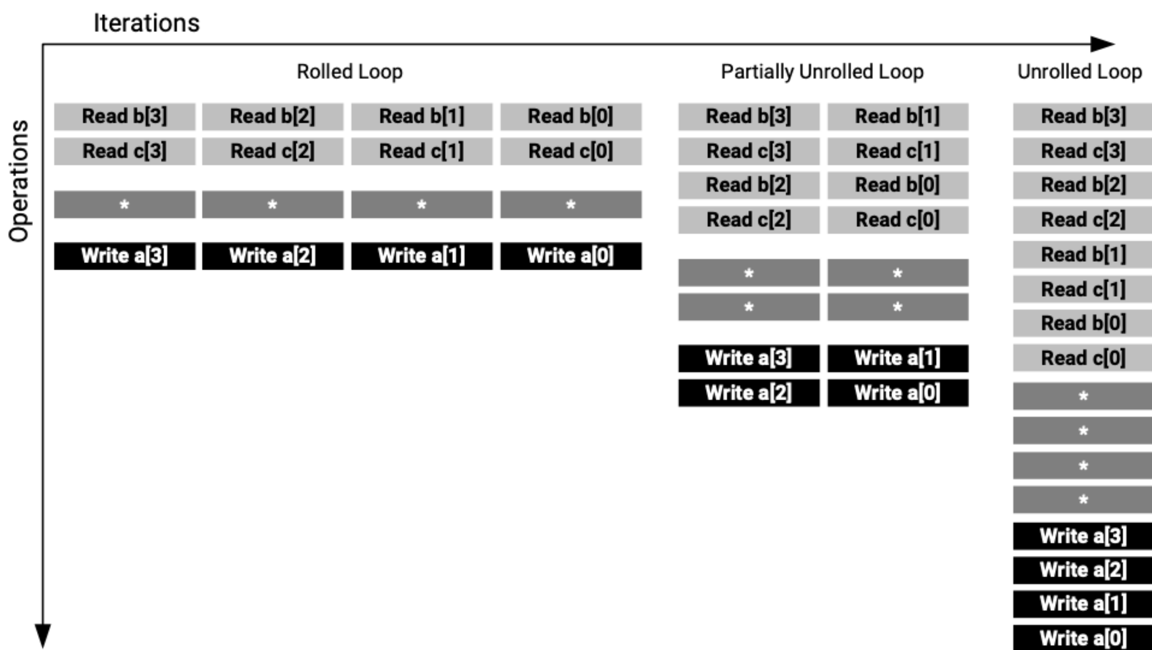


Figure 3.12: HLS loop unrolling [67]

This processing filter operates at the *Free-running* clock frequency of 100 MHz, while the ADC and DAC operate at the hundred kilohertz range. Therefore, 32 iterations of the loop do not affect the throughput, and footprint can be preserved by keeping the loop rolled.

3.3.3 IIR filter

Infinite Impulse Response (IIR) filters present feedback loops, which means that their output depends on values of current or past inputs as well as past outputs [68].

Equation 3.5 presents the general expression for the output. This type of filter provides better filter characteristics with a lower degree than FIR filters, but lacks linear phase and presents instability risks.

$$y[n] = \sum_{i=0}^{M} b_i x[n-i] - \sum_{j=1}^{N} a_j y[n-j] \tag{3.5}$$

The simplest low-pass IIR filter consists of a *Single-Pole IIR filter*, where the output depends only on the input and the previous output, weighted with a constant called α [69]. This filter follows the expression in Equation 3.6.

$$y[n] = \alpha x[n] + (1-\alpha) y[n-1], \qquad 0 < \alpha < 1 \tag{3.6}$$

The smaller α, the steeper the frequency response gets. This design uses a value of $\alpha = 0.25$. As this filter was designed to be used on PCIe samples, the data is 64 bits long. The multipliers in the FPGA are 25x18 bits, so three of them must be used for each data value if the coefficient is an 18-bit fixed-point constant with 1 sign bit and 17 fractional bits. However, in this case, there are no precision concerns as 0.25 and its complement to 1, 0.75, are directly divided into negative powers of two.

The filter is implemented in HLS as the sum of the coefficient times the input and the complement to one of the coefficient times the previous output. The previous output is stored in a static variable synthesized to local memory of the algorithm. The `HLS RESET` directive must be used to reset that memory to 0 when the global reset is triggered.

3.4 Simulation

Some simulation test benches were developed to aid during development. They were used to verify the correct model working in isolation. The next subsections explore all the tests designed in this project for both VHDL and HLS models.

The reference design provided by TEWS Technologies provided simulation agents for the ADC and DAC that mimic the external devices on the board, checking for correct timing and simulating the real element behavior. They were used without modifications to simplify test development.

Both HLS FIR and IIR tests use precalculated signals. These signals have been generated with a Python script that generates input sine signals and filters them. It automatically generates an HLS header file that contains all the samples for input and output on each channel of the tests.

All the tests are used to check the correct implementation of the isolated blocks. However, some test methodologies require a test that checks all the blocks' integration. This test would require instantiating the top design with the external device agents, implementing a PCIe agent to simulate the host computer, and generating an independent clock of 100 MHz.

3.4.1 ADC VHDL module test

The ADC VHDL test checks the correct operation of the ADC module. It uses the simulation agent provided by the TEWS Technologies reference design, using random data as input and checking if it was read correctly. The block diagram can be found in Figure 3.13, where the red elements are the ones being tested, and the blue ones are test blocks and interfaces.

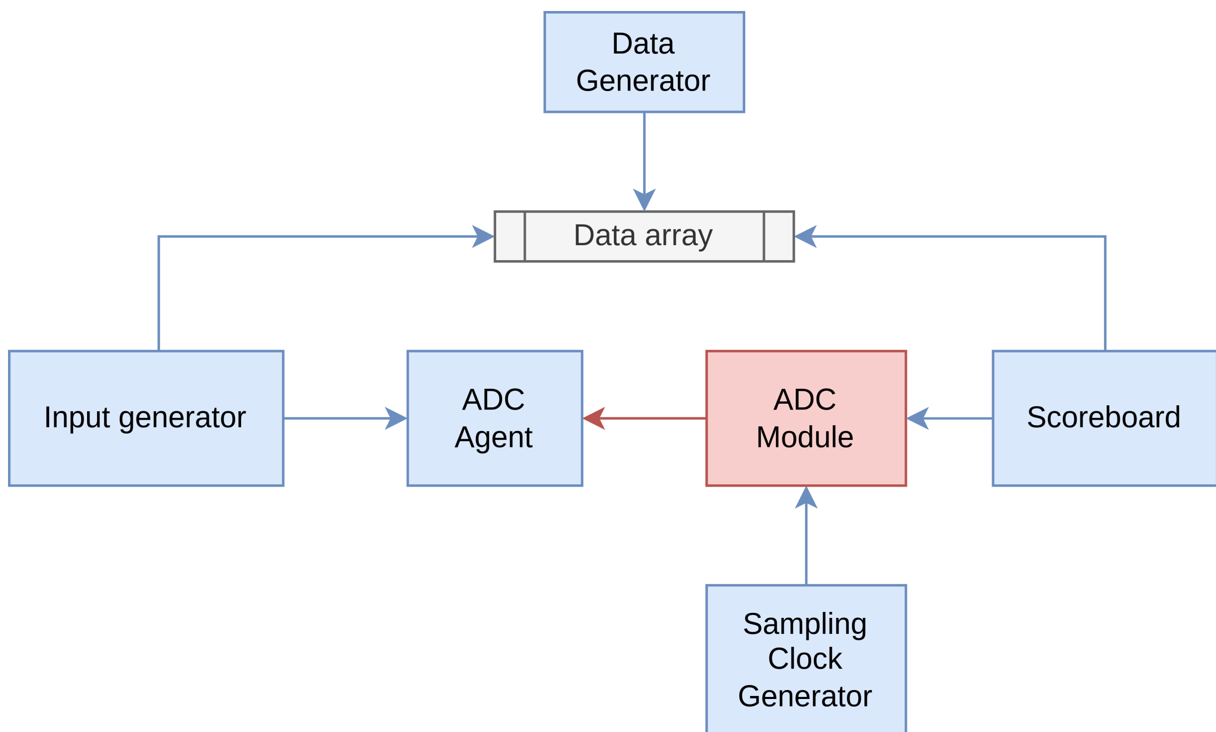


Figure 3.13: Block diagram of the ADC module test bench

The test is based on a sampling clock generator, an input generator, and a scoreboard to check the outputs. The clock frequency of the test is 100 MHz, and the sampling clock is set at 100 kHz. A data generator process fills an array with uniformly distributed random data (with a fixed seed for repeatability) at test start. Then, another process inputs the numbers from that array into the ADC agent input channels. The array position order is different for each channel to have distinct input sequences in each one.

On each sampling clock rising edge, the scoreboard process asserts the previous values, and the input process switches to the following data. This procedure is repeated several times to verify the correct working of the design. If every assertion is correct, the test passes successfully.

3.4.2 DAC VHDL module test

The DAC VHDL test checks the correct operation of the DAC module. It uses the simulation agent provided by the TEWS Technologies reference design, using random data as input and checking if it was written correctly. Figure 3.14 shows a block diagram of the test, where the red elements are the ones being tested, and the blue ones are simulation blocks.

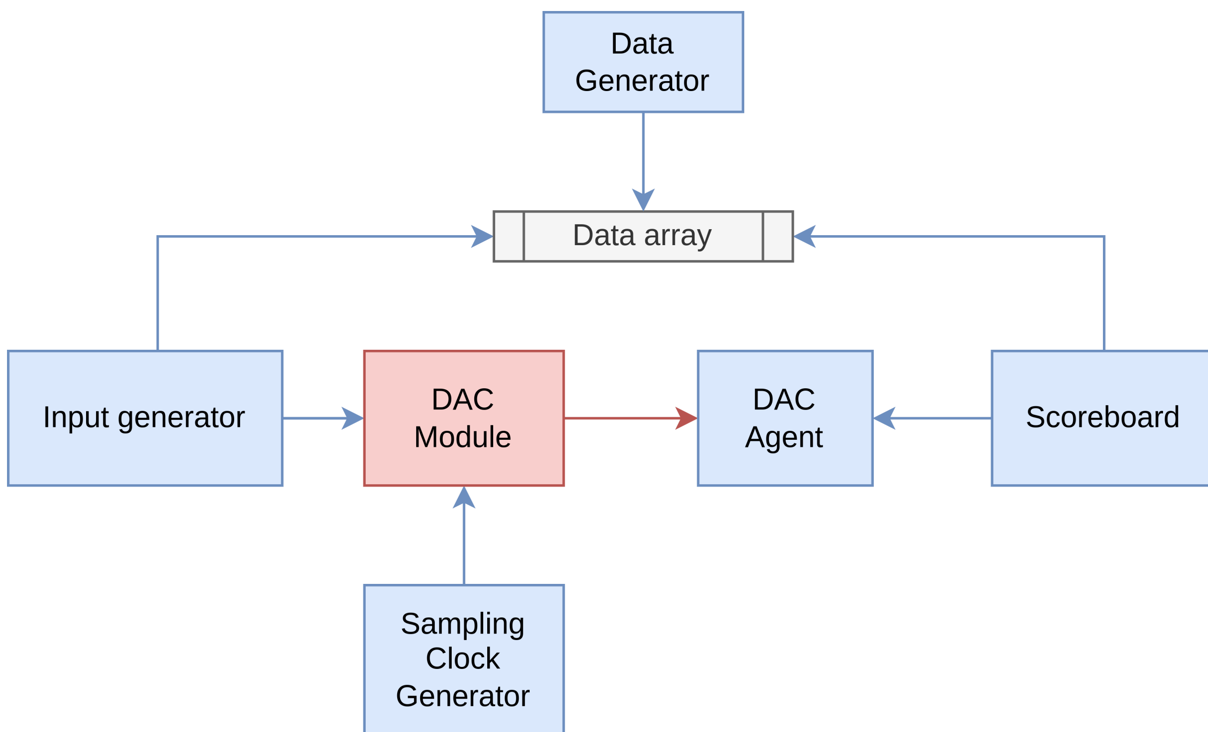


Figure 3.14: Block diagram of the DAC module test bench

The test generates random data from a uniform distribution. At the beginning of the test, the data is generated and stored in an array. The data from this array is sent to the AXI streaming interfaces of each channel in different orders. The test clock has a frequency of 100 MHz.

A 100 kHz sampling clock is generated and sent to the DAC module. On each clock edge, the output of each channel is asserted against the expected value from the array. This procedure is repeated a hundred times and the test finishes successfully if all assertions are correct.

3.4.3 Configuration registers VHDL test

The configuration registers test checks the correct implementation of the AXI registers module. A simulated AXI is used to write values to the registers, and the output individual values are checked.

This module operates on two different clock domains. This is simulated with an independent generation of both 100 MHz and 125 MHz. The inputs are synchronized to the PCIe (125 MHz) clock and the assertions are performed synchronized to the DAQ (100 MHz) clock.

In order to simplify the clock alignment logic when performing a synchronization, the worst-case scenario for CDC is calculated. Figure 3.15 contains a diagram with both the best and worst possible synchronization situations.

In the best-case scenario, data is changed just one clock tick before the registers are sampled, and the conversion takes only the CDC time. On the other hand, the worst case takes place when data is changed just one tick after the register sample. In this case, it takes a full conversion of the previous data and the CDC time for data conversion. This time has been checked with the logic analyzer and is estimated to be below 300 ns, so this time is used for the simulation tests.

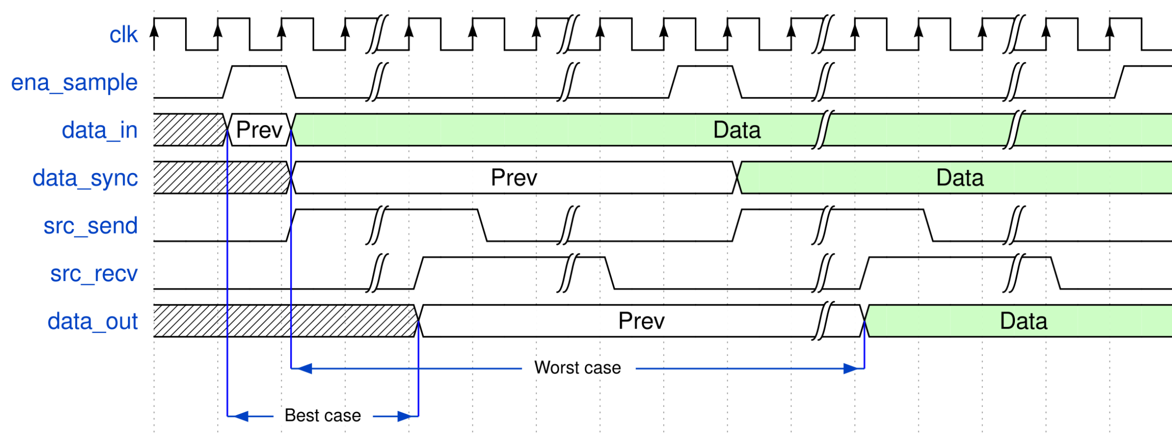


Figure 3.15: Best and worst case for CDC handshake synchronization

The timing procedure for one assertion of this test is:

1. Synchronize with PCIe clock (wait for rising edge)
2. Perform the AXI transactions
3. Wait for the CDC's worst-case conversion time
4. Synchronize with DAQ clock (wait for rising edge)
5. Assert the correct output state

First, the initial expected values for the outputs are tested, without previous transactions. Then, the four registers are tested sequentially twice, checking both high and low values for all possible outputs.

If all output asserts are correctly passed, the test ends successfully.

3.4.4 Width adapter HLS test

HLS tests are written in C++, and simulation is performed on the Vitis software. After checking the correct behavior and running synthesis, a cosimulation is performed, testing the synthesized logic.

The width adapter test checks the correct implementation of the width adapter in HLS, in both directions.

For that, a sequential series of numbers is sent in each conversion direction and the results are checked each iteration. The bit shifts are performed to keep the most significant bits to validate the correct padding and shifting.

3.4.5 IIR filter HLS test

The IIR filter test checks the correct implementation of the HLS filter. This test uses the precomputed signals from the generated header file to improve the test runtime.

The test inputs samples from the precomputed sine input and asserts the output with the precomputed expected value. Some tolerance is allowed because of the fixed-point operations performed in HLS and Python's arbitrary floating-point precision.

The ADC channels must be loaded with dummy data, and the DAC channels must be read after each algorithm cycle to avoid hanging the test.

The test passes successfully if the whole precomputed signal is written and asserted. An example of the input and expected output signal can be seen in Figure 3.16 (only the first 250 samples are shown).

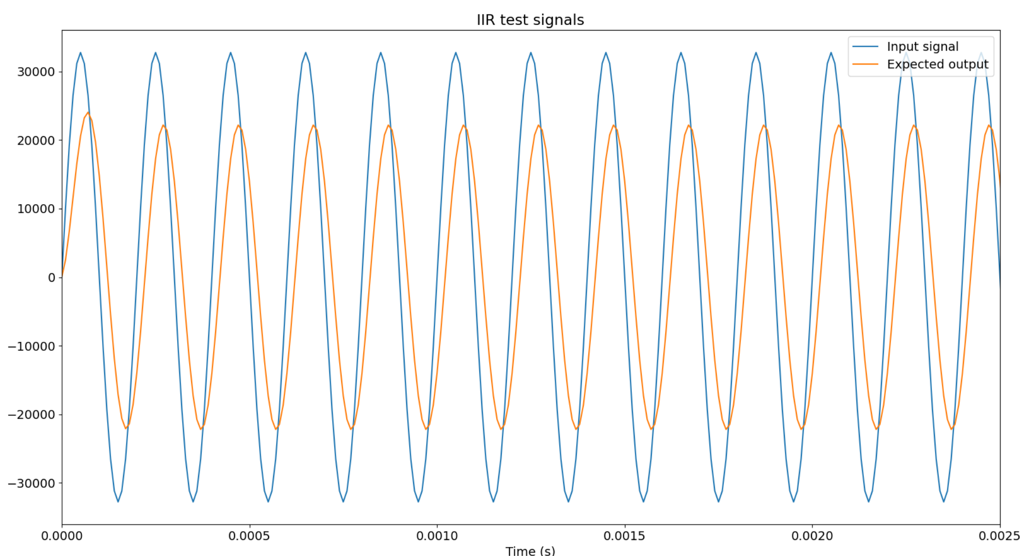


Figure 3.16: Input and expected output on HLS IIR test

3.4.6 FIR filter HLS test

The FIR filter HLS test checks the correct implementation of the FIR filter. To verify it, the test uses the precomputed samples from the Python utility. They contain the input and expected output for sine waves with four different frequencies.

The test takes a sine wave input array and iterates through its samples, inputting them into all ADC channels and reading the DAC output, asserting it to be close to the

expected value. A small tolerance is accepted to allow fixed and floating point precision differences. After iterating through one of the sine waves, the filter memory is reset by writing samples with zero amplitude until the shift register is completely filled with zeros. Then, the test is repeated with the rest of the input arrays.

Once all sine waves have been tested, the test has passed successfully if all the assertions have been correct. Figure 3.17 shows four signals used as inputs and their expected output.

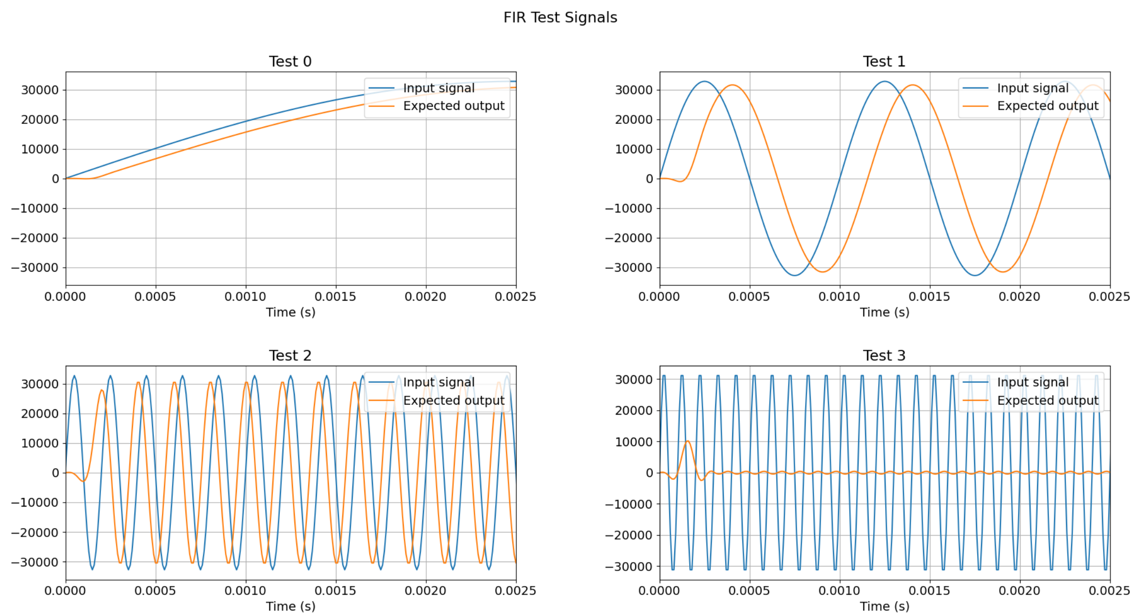


Figure 3.17: Input and expected output on HLS FIR tests

3.5 Python GUI

The XDMA IP core allows this application to use the XDMA Linux driver made by Xilinx [64]. This driver includes some tools devised for quick testing and development that enable reading and writing to the memory-mapped registers and data streaming.

In order to access the configuration registers easily, a Python application was developed. This application creates a Graphical User Interface (GUI) to interact with the DAQ device and change parameters such as ranges, frequencies, etc. The application converts the selected changes to register contents and uses the `reg_rw` utility to write them to the FPGA. Figure 3.18 shows the application stack, based on the XDMA tools and driver.

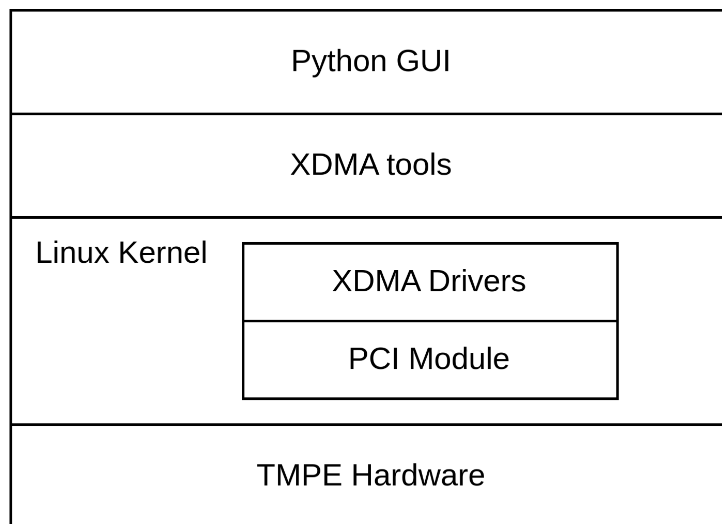


Figure 3.18: Python application stack

The interface has been developed with Tkinter, one of the most used frameworks for Python GUI development [70]. The interface, seen in Figure 3.19, is divided into four sections, one for each configuration register. On application startup, the register contents are read to set the application state to the current configuration and get the base register content onto which the requested modifications are performed.

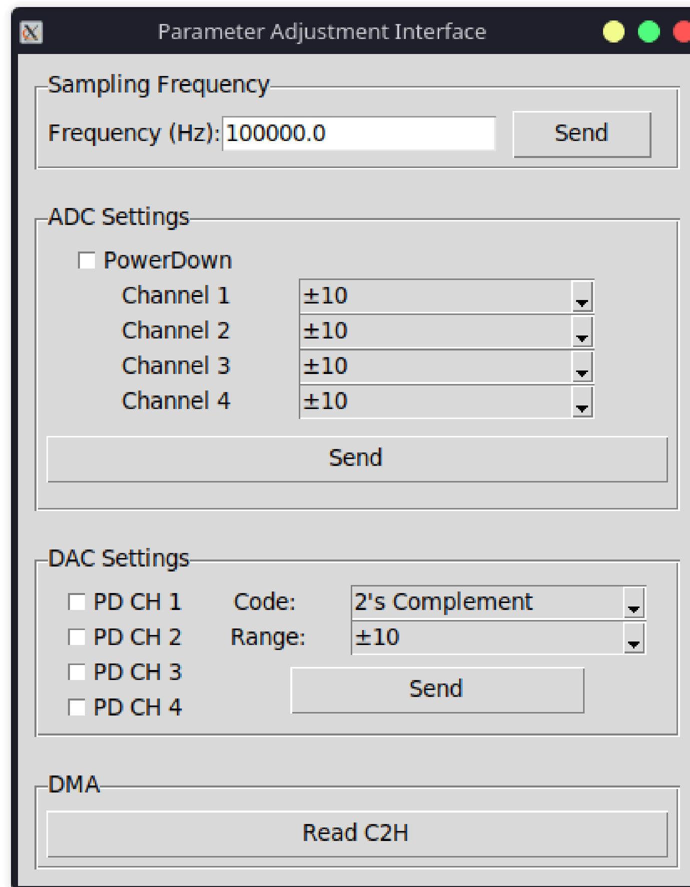


Figure 3.19: Python GUI for register access

The *Sampling Frequency* section can be used to change the acquisition frequency of the DAQ. As the DAQ clock is generated with a frequency divider, only whole divisions of the *Free-running* clock (100 MHz) are supported. The application is designed with this limitation in mind. If an invalid frequency is requested, the closest valid one is calculated, written to the register, and displayed on the GUI.

The *ADC Settings* section manages the ADC power down (not channel-independent) and the range for each channel. The range drop-down contains all possible ranges for each channel, not allowing invalid inputs.

The *DAC Settings* section allows the selection of DAC channel power down, negative number encoding (either 2's complement or offset binary), and the choice of the DAC range, which must be the same for all channels.

These three sections include a `Send` button. The changes made on the GUI are not sent immediately to avoid accidental configurations or non-synchronous changes to the same module. Once all desired changes are selected, pressing the button sends them simultaneously to the FPGA.

The *DMA* section allows the data streaming from the board to be tested. When the

button is pressed, the C2H streaming is enabled, 4096 samples are read to a file, the streaming is disabled, and the file contents are plotted to the user. This option uses the `dma_from_device` utility from the Xilinx driver. Figure 3.20 shows an example output for a square-wave input.

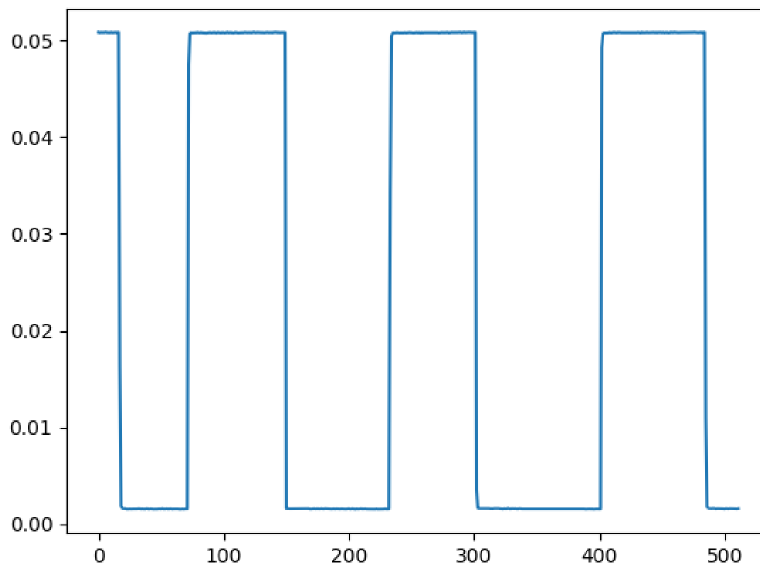


Figure 3.20: Example C2H reading from Python GUI

3.6 C++ Library

The Python GUI provides an accessible interface to interact with the device. However, a C++ library has been developed to provide a more extensible and automatable approach. This library allows interaction with the device via the device files generated by the Xilinx driver.

The register interface is accessed through the `/dev/xdma0_user`. This character device is memory-mapped with the `mmap` function, obtaining a pointer to the memory address where it has been located. The kernel automatically converts accesses to this memory address to AXI Lite transactions. Therefore, with a `static_cast`, the register interface can be accessed like an array.

However, the user of this library does not need to know the register layout or its contents. Therefore, the library does not offer public access to the register pointer; instead, it offers an interface based on getters and setters of every property inside it. This abstraction provides the library with ease of use and security, as the inputs can be checked before allowing only valid configurations. The complete list of library functions can be found in the manual in Appendix E.

The information streams are accessed on the `/dev/xdma0_c2h_0` and `/dev/xdma0_h2c_0` files. These device files are open on library startup and can be accessed with plain read and write C functions. However, the memory buffers used for reading and writing must be memory-aligned. This can be easily achieved with the use of the `posix_memalign` function.

This library also provides an abstraction from the use of plain file descriptors and offers the `read_c2h_dma` and `write_c2h_dma` functions to the user. Furthermore, the `read_c2h_dma` function allocates a memory-aligned buffer, so the user can use any buffer (the user is still required to use a memory-aligned buffer on `write_c2h_dma`) and the invalid samples from the beginning of the acquisition are automatically deleted.

The file descriptor for the device files should be kept open during library utilization to avoid concurrent access with other applications. Therefore, the descriptors should be opened at the application startup and closed when not used again. The Resource Acquisition Is Initialization (RAII) paradigm was used to simplify the management and increase the implementation's safety. [71].

With this paradigm, holding the file descriptor open becomes a class invariant. Therefore, the three descriptors are open on the library's class constructor and are only closed on the library's destructor. These file descriptors are kept private from the user to abstract them from their management.

To summarize, the library offers users a simple interface in which they should create one instance of the `tmpe_daq` object. This instance provides getters and setters for all register fields and read and write methods for both streamings. The resources are automatically freed on the destructor, so no special management is needed.

Figure 3.21 shows the stack of a generic application that uses the library to access the

design. The library uses the XDMA drivers loaded in the kernel to access the PCIe interface and communicate with the device.

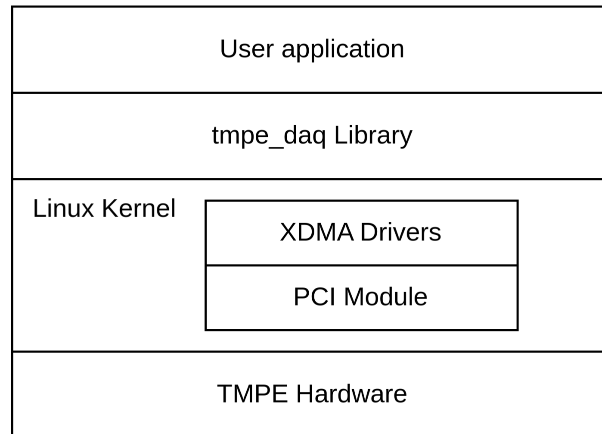


Figure 3.21: Stack of a generic application implemented with the library

3.7 Project management

This project has a lot of different parts that are compiled differently. In order to make the build process as streamlined as possible, some build automation tools were used. In addition, special practices were used to allow source control of the whole project.

The build automation is implemented using GNU Make [72]. This tool automates build procedures with a file called `Makefile`. This file includes recipes for creating the Vivado project, synthesizing and implementing the bitfile, loading the bitfile to the FPGA, and building the C++ library, among others.

This project's source control has been performed with Git [73] and GitHub [74], the two most used tools for this purpose. It can be found on the `David-Andrino/tmpe_daq` repository [75]. The Vivado project should not be source-controlled because it contains a lot of absolute paths and configurations that are not portable to other computers. Instead, the project contains a script in the Tool Command Language (TCL) scripting language that generates a new Vivado project with the correct configuration. The script is called `generate_project.tcl` and can be found in the `scripts` folder.

The TCL scripting language is also used for the creation of the bitfile and the loading to the FPGA, with scripts such as `generate_bitfile.tcl` or `load_bitfile.tcl`. The C++ library is built with another `Makefile` found inside the library folder.

The C++ library has been documented with Doxygen [76] format, and the Sphinx [77] tool has been used to combine the results with reStructured Text [78] documentation to create the user manual (Appendix E). This document and the associated presentation have been developed with the LaTeX markup language [79].

The default target of the root Makefile (`all`) creates the project, builds the bitfile, loads it to the FPGA, and compiles the library and the user manual.

This project has been developed using multiple programming and design languages. Table 3.6 shows a table of the different languages used and the lines written in each. Figure 3.22 shows a pie chart of the table contents.

Table 3.6: Languages and line count of the project

Language	Lines
`VHDL`	3195
`C++`	555
`HLS (C++)`	490
`Python`	363
`Makefile`	95
`TCL`	47

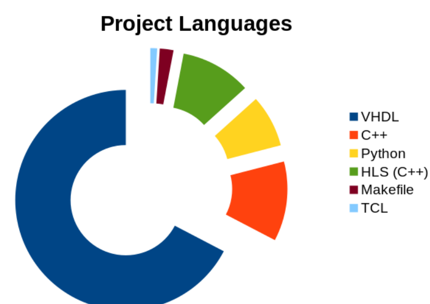


Figure 3.22: Design language distribution

Chapter 4

Results

4.1 Design verification

The VHDL and HLS models have been tested with simulation test benches in the same respective language, as explained on section 3.4. All models were designed with those tests as tools for correct working checks. Therefore, all models correctly pass their respective test benches. The VHDL test benches are performed on Vivado, allowing the inspection of internal signals for model debugging.

4.1.1 ADC VHDL test results

This test was explained in subsection 3.4.1. This test inputs data into the simulation agent of the ADC and asserts the correct reading of that data.

Figure 4.1 shows a complete conversion cycle on the ADC simulation. Figure 4.2 shows zooms on the transaction start (top) and finish (bottom). The red signals are physical ADC signals, the green ones are internal signals of the ADC module, and the pink ones are the AXI Stream interface used to output the results.

The transference starts with the `Sampling CLK` high level. This leads to a high level of the `CNV` signal, commanding a conversion. Then, the state changes to `IDLE`, where it waits for the falling edge on the `BUSY` signal that signals the end of conversion. When the conversion is finished, the SPI interface is used to send the configuration for the next sample and read the result of the previous one. The sampling result is sent through the AXI interface when the transference is complete.

This test is completed successfully after all values are correctly transferred.

4.1.2 DAC VHDL test results

As explained on subsection 3.4.2, this test writes data to the DAC channels and checks the outputs to match the selected input.

Figure 4.3 shows one cycle of DAC conversion. The red signals are the physical DAC ports, the pink signals are internals of the module, and the green ones are the simulated DAC outputs. A conversion is triggered by a high value of the sampling clock. This triggers a cycle on the DAC FSM (see Figure 3.8). As seen in the state signal, this cycle writes the data for the samples of the four channels and configures the three registers with the selected settings. The `LDACn` signal triggers a simultaneous update of the seven registers, as seen in the output channel signals that change simultaneously after the release of said signal.

This test is completed successfully after all values are correctly transferred.

4.1.3 Configuration registers VHDL test results

As explained on subsection 3.4.3, this test performs AXI Lite transactions on the configuration registers and checks the outputs aligned with the output clock.

Figure 4.4 shows an example of an AXI Lite transaction. In this case, the value `0x80000FFF` is being written to register `0x4`. The top pink signals are the address write interface, the green ones are the data write interface, and the red ones are the response interface. First, the address and data to be written are transferred, and then the slave acknowledges it through the response interface.

Figure 4.5 shows a simulated CDC data transfer. `0x000003E8` is being transferred to the DAQ domain from the AXI domain. The signal behavior is the expected handshake, explained on subsection 3.2.3. The blue signals represent the data to be transferred, and the pink ones are the control signals used on the handshake.

This test is successful after transferring all data and checking the output values.

4.1.4 HLS test results

The HLS test benches are implemented in C++ and run on the Vitis HLS tool. These tests are completely automatic, using standard comparisons and return codes as alternatives to asserts. All developed kernels have an associated test bench. The HLS test content is explained on subsection 3.3.1, subsection 3.3.2 and subsection 3.3.3.

The tests include progress printing, showing information about the successful completion of the tests, as seen in Figure 4.6.

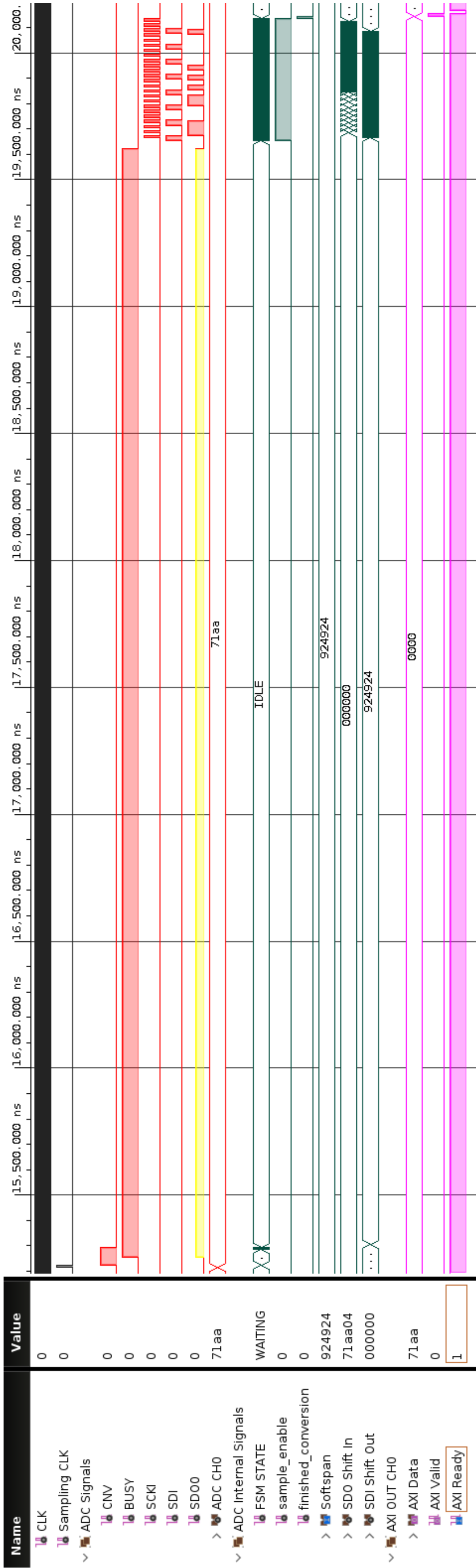


Figure 4.1: Simulation of one ADC read

Figure 4.2: Zoom on the ADC read start and finish

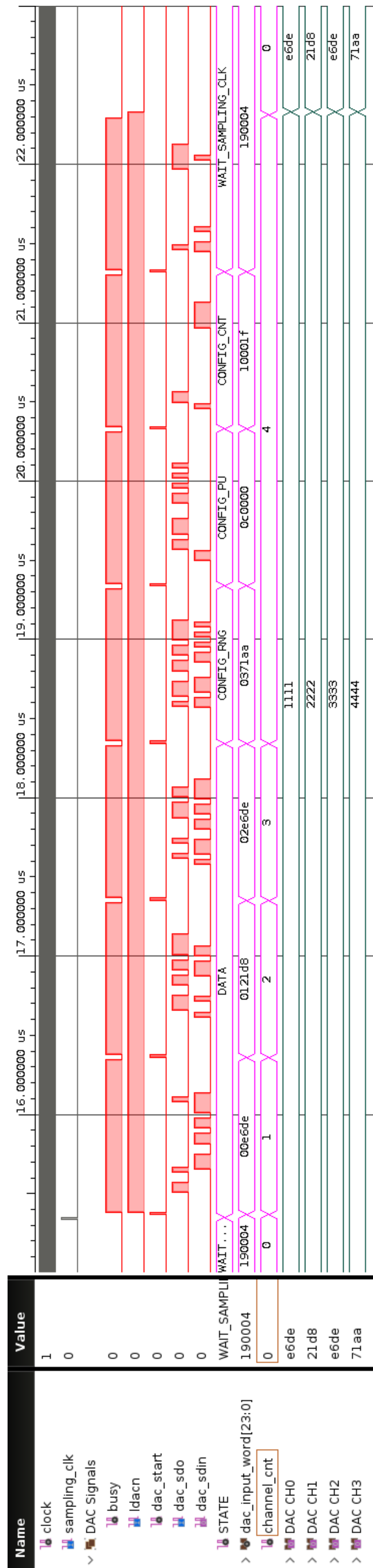


Figure 4.3: Simulation of one DAC transaction

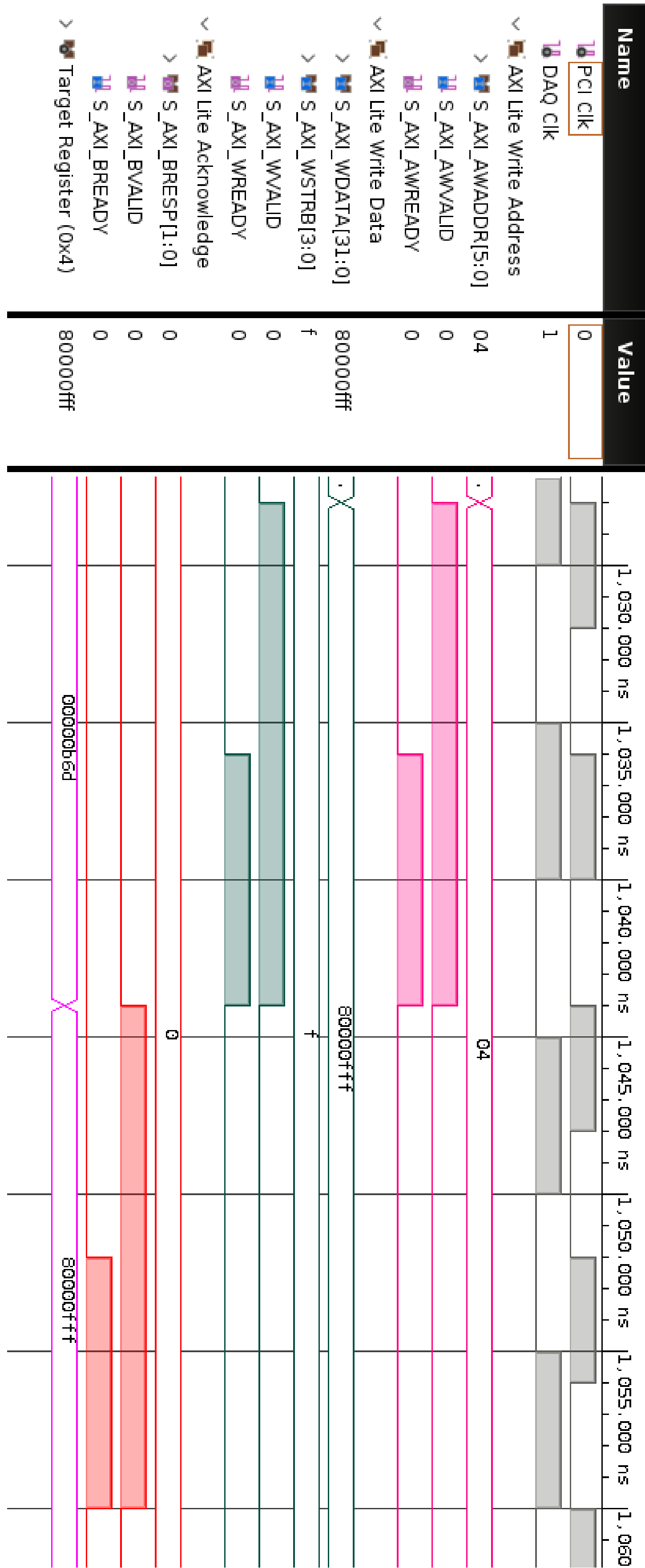


Figure 4.4: Simulation of AXI Lite transaction

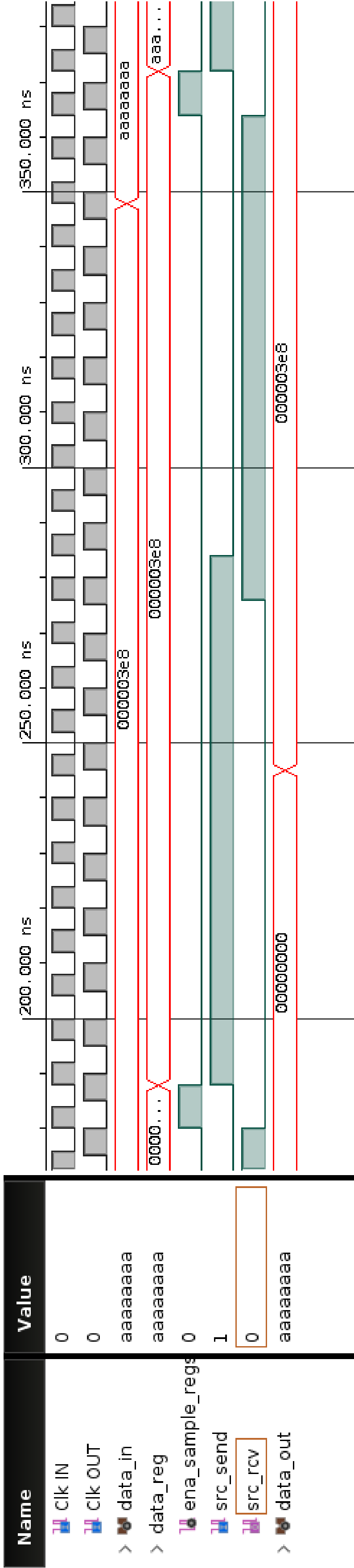


Figure 4.5: Simulation of CDC data transfer

```
INFO: [SIM 211-2] *************** CSIM start ***************
INFO: [SIM 211-4] CSIM will launch GCC as the compiler.
   Compiling ../../../../../hls/testbench_adapter.cpp in debug mode
   Generating csim.exe
[TEST] Starting Width adapter test
[TEST] Width adapter test completed successfully
[TEST] Starting FIR test
[TEST] FIR test completed successfully
INFO: [SIM 211-1] CSim done with 0 errors.
INFO: [SIM 211-3] *************** CSIM finish ***************
```

```
INFO: [SIM 2] *************** CSIM start ***************
INFO: [SIM 4] CSIM will launch GCC as the compiler.
   Compiling ../../../../../hls/testbench_adapter.cpp in debug mode
   Generating csim.exe
[TEST] Starting IIR test
[TEST] IIR test completed successfully
[TEST] Starting FIR test
[TEST] FIR test completed successfully
INFO: [SIM 1] CSim done with 0 errors.
INFO: [SIM 3] *************** CSIM finish ***************
```

Figure 4.6: HLS completion reports

4.2 Design validation

Design verification is performed on simulated models. It is a very important step in the design process, but it is not definitive. A model that fails the verification is incorrect, but a model that completes it is not guaranteed to work. Because of this, a validation step is necessary.

4.2.1 FIR filter validation

The results of the FIR filter have been checked with a Red Pitaya Diagnostic Kit [80]. It is a data acquisition tool in a small form factor. It includes two input and two output channels and offers a web interface to access its tools. Using the device's waveform generator and oscilloscope functions, the FIR filter capabilities were checked.

A block diagram of the connections for the validation can be found on Figure 4.7. In the diagram, the red elements represent the data flow elements, the blue elements are in charge of configuration, and the green elements are for the web interface and connection through the Local Area Network (LAN).

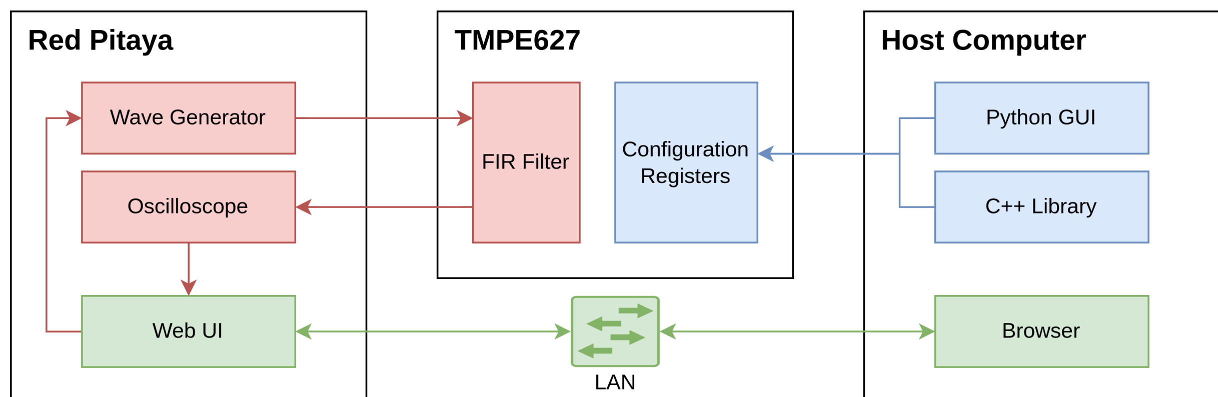
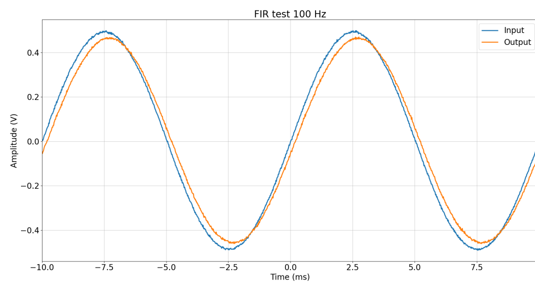
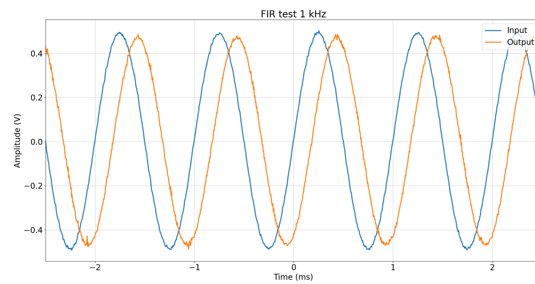


Figure 4.7: Block diagram of the FIR verification

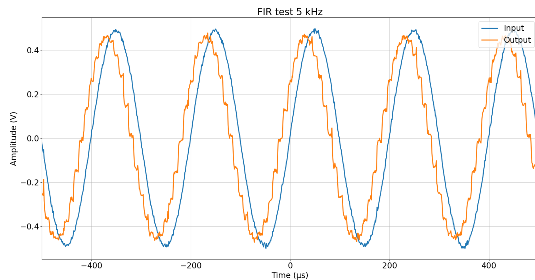
Figure 4.8 includes four different processing examples. It is important to note the lack of a reconstruction filter in the project's output. Therefore, the output is a sampled sine wave instead of a clean one. As can be seen in the figure, the 10 kHz frequency is almost completely attenuated, as was designed in subsection 3.3.2.



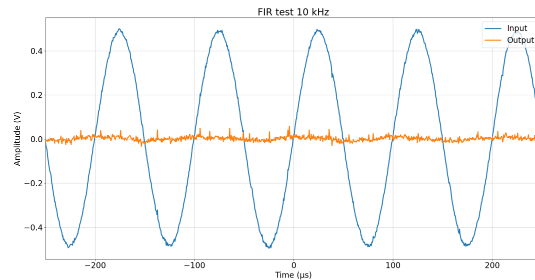
(a) Filtering of a 100 Hz sine wave



(b) Filtering of a 1 kHz sine wave



(c) Filtering of a 5 kHz sine wave



(d) Filtering of a 10 kHz sine wave

Figure 4.8: FIR filter results on time domain

In order to measure the complete frequency response of the filter, the Bode Diagram function of the Red Pitaya was used. This function generates sine waves, checks the output amplitude and phase for a discrete range of frequencies, and plots the results against the frequency. The sampling frequency for the DAQ was set to 100 kHz, so the diagram was plotted over 3000 frequencies in the 500 Hz to 50 kHz range.

The results can be found in Figure 4.9. However, the diagrams are very noisy, and the phase result does not make sense. This is probably because of the sampled sine output, as the Pitaya expects a clear sine wave like the one on the input. Therefore, a moving average filter was applied to the resulting amplitude Bode diagram with a Python script. The result can be found in Figure 4.10. The script used to filter the amplitude can be found in Appendix D.

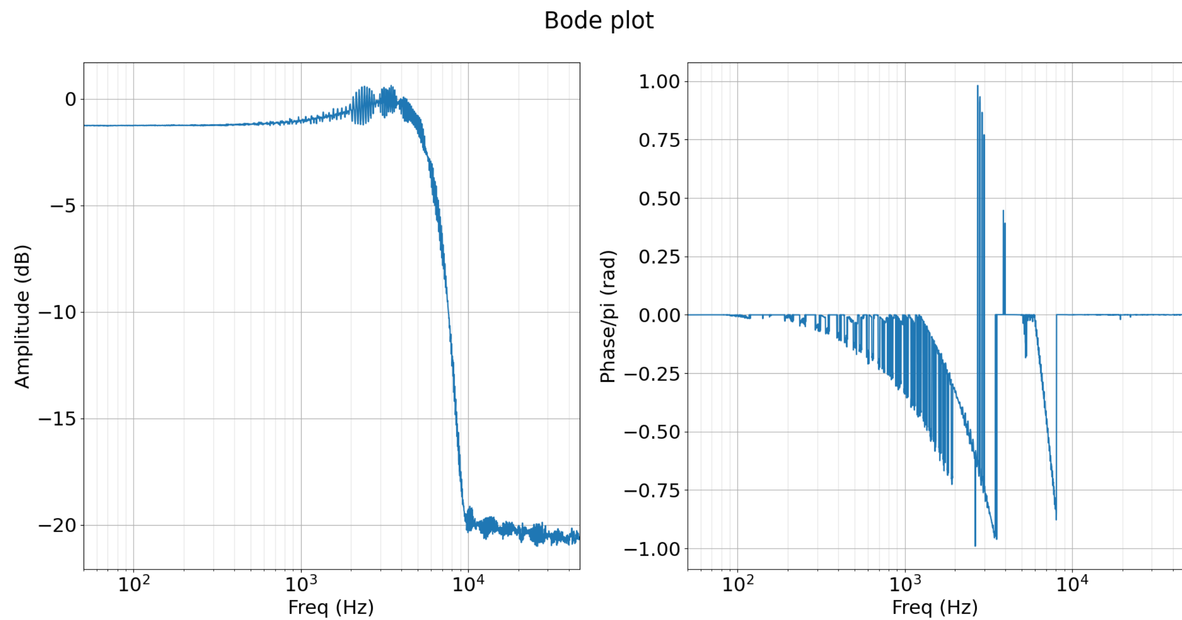


Figure 4.9: Bode diagram of FIR filter amplitude on Red Pitaya

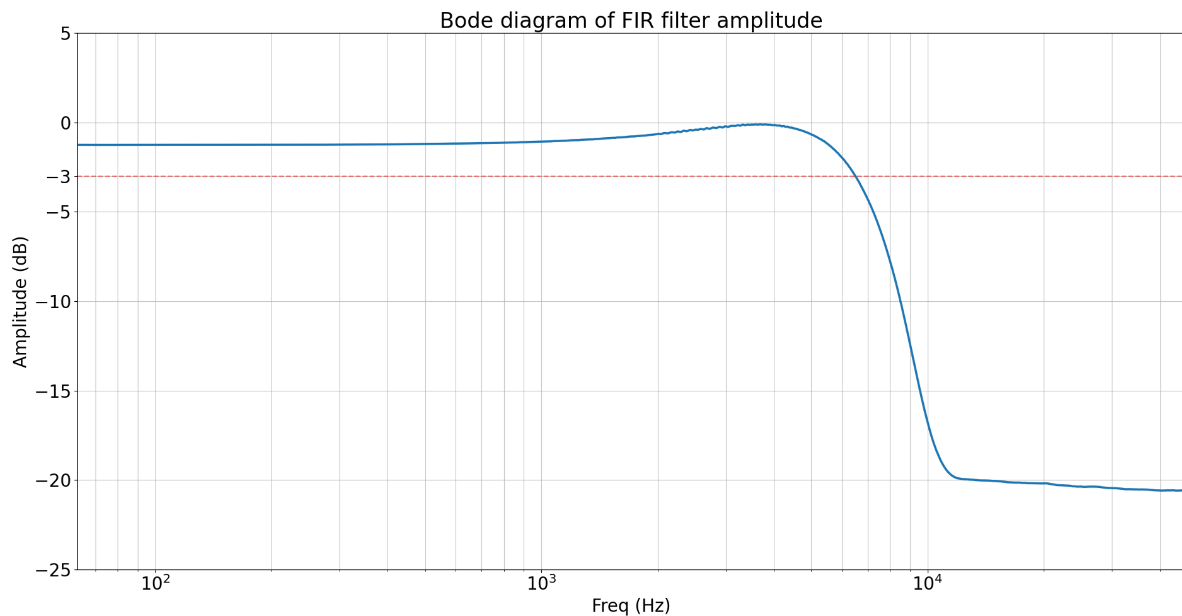


Figure 4.10: Filtered bode diagram of FIR filter amplitude

4.2.2 DAQ frequency results

The FPGA offers some general-purpose I/O ports. These ports have been used to verify and test the acquisition frequency.

The I/O port 0 has been connected to the output of the Sampling Frequency Generator module, allowing precise timing verification. However, the output of the Frequency

Generator module is one clock cycle long. Therefore, the signal is active for 10 nanoseconds on each sampling cycle, which is in the tens of microseconds range.

This kind of signal is challenging to measure with an oscilloscope. To make it easier to acquire, a T-type FF (or one-bit counter) has been used. This element toggles its input on each enabled clock cycle. As a consequence, a square signal with half the frequency is generated.

Figure 4.11 shows this signal when the sampling clock is configured at 100 kHz. It can be seen that, as expected, a square signal with a 50 kHz frequency is captured.

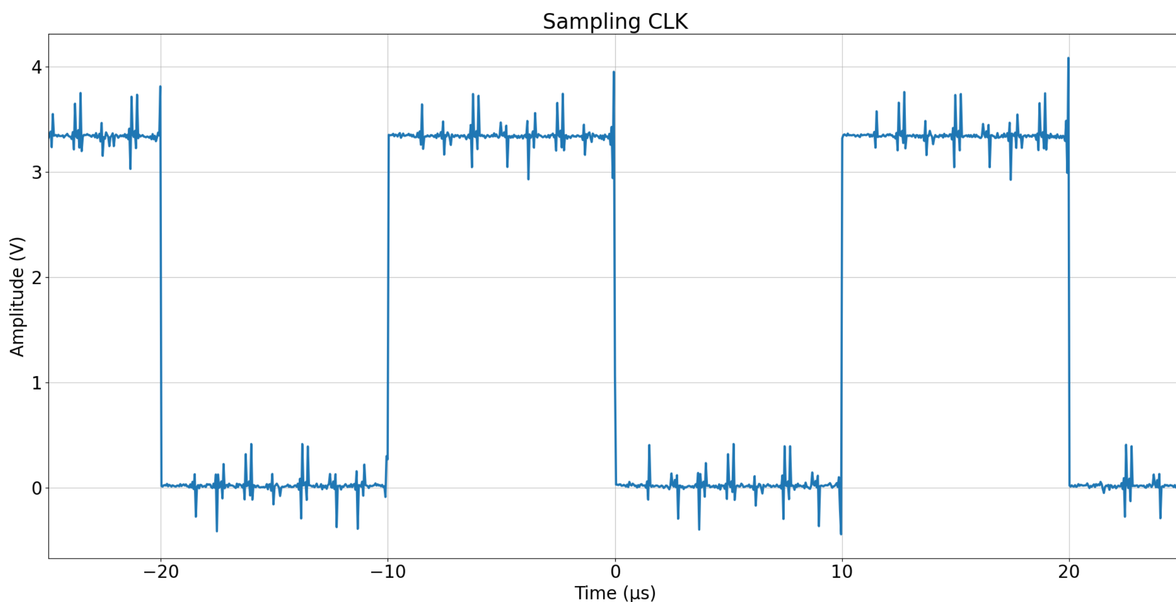


Figure 4.11: Sampling clock at 100 kHz

The I/O pin 1 is connected to the ADC busy pin. This connection checks the time it takes for the ADC to perform a conversion and the correct synchronization to the clock. The conversion takes 2.06 μs, needing some time after it to transfer the information to the FPGA.

Figure 4.12 shows the sampling clock (with toggle logic) and the ADC busy signal with a sampling frequency of 100 kSPS. The busy signal is triggered after every toggle of the sampling clock, that is, on every sampling clock edge. The maximum frequency of the module can be found in Figure 4.13, with an approximate value of 286 kSPS. Any frequency above the limit causes some clock edges not to trigger a conversion, making the timing irregular. However, the datasheet of the ADC states a maximum acquisition frequency of 200 kSPS. This means that any acquisition faster than that will probably contain inaccurate data. Therefore, this design can use the ADC to its maximum potential.

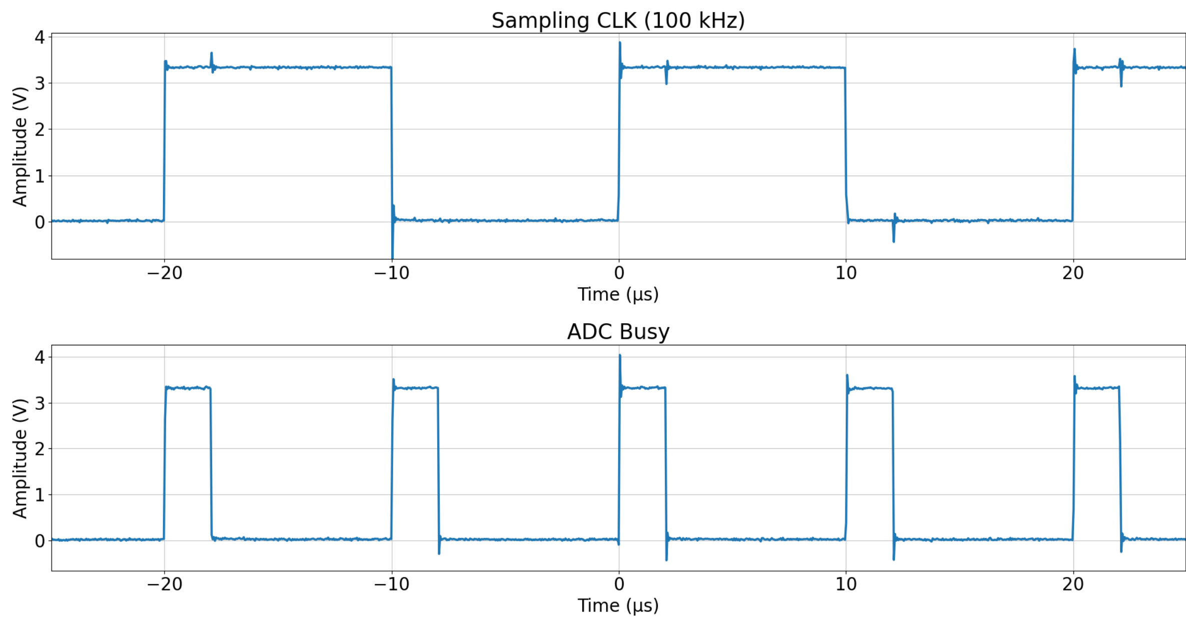


Figure 4.12: ADC busy signal at 100 kSPS

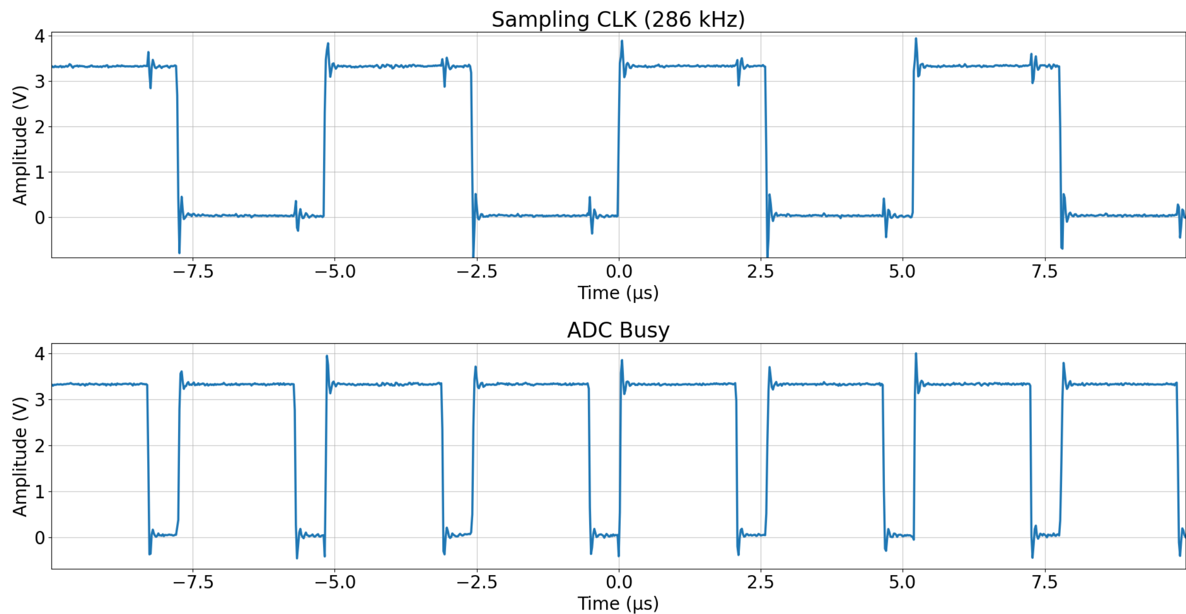


Figure 4.13: ADC maximum communication frequency

On the other hand, I/O pin 2 is connected to the `LDAC` pin of the DAC. This pin is held down during information transfers with the DAC, so it can be used as a busy signal, because the DAC does not include one. One register write takes 1.3 μs, with a complete cycle taking seven writes (3 configuration registers and 4 data registers).

Figure 4.14 contains the waveforms for a DAC cycle with a sampling frequency of 100

kSPS. The seven information transfers can be clearly seen on the seven low pulses on each sampling clock toggle. The maximum frequency achievable is captured on Figure 4.15, where the sampling clock arrives just as the seventh transmission is finished. Any faster frequency causes the module to miss some clock cycles.

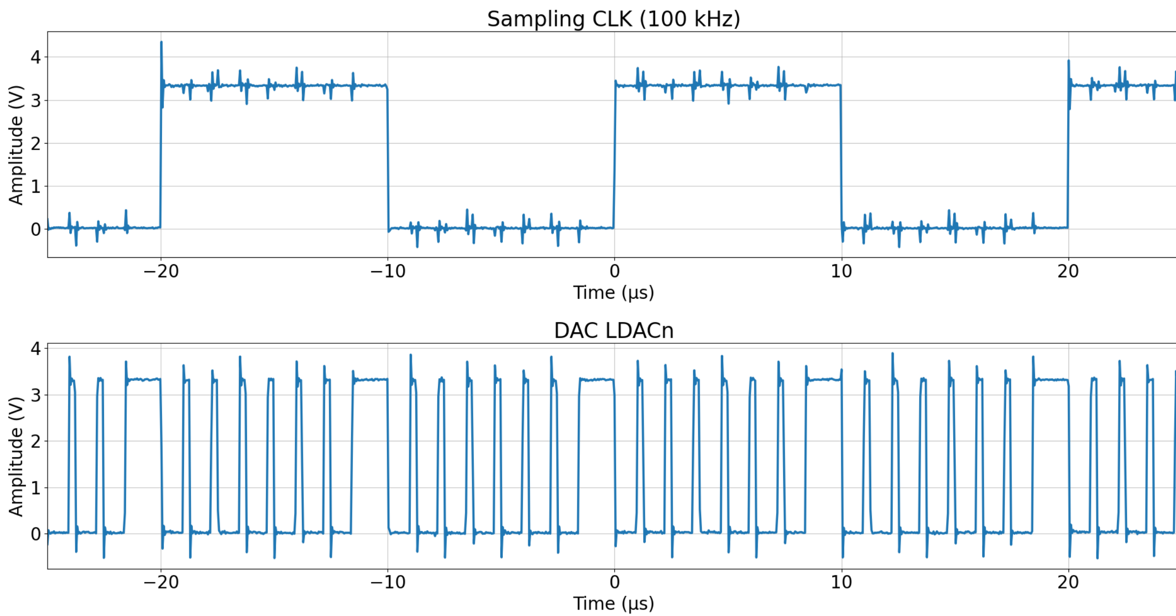


Figure 4.14: DAC busy signal at 100 kSPS

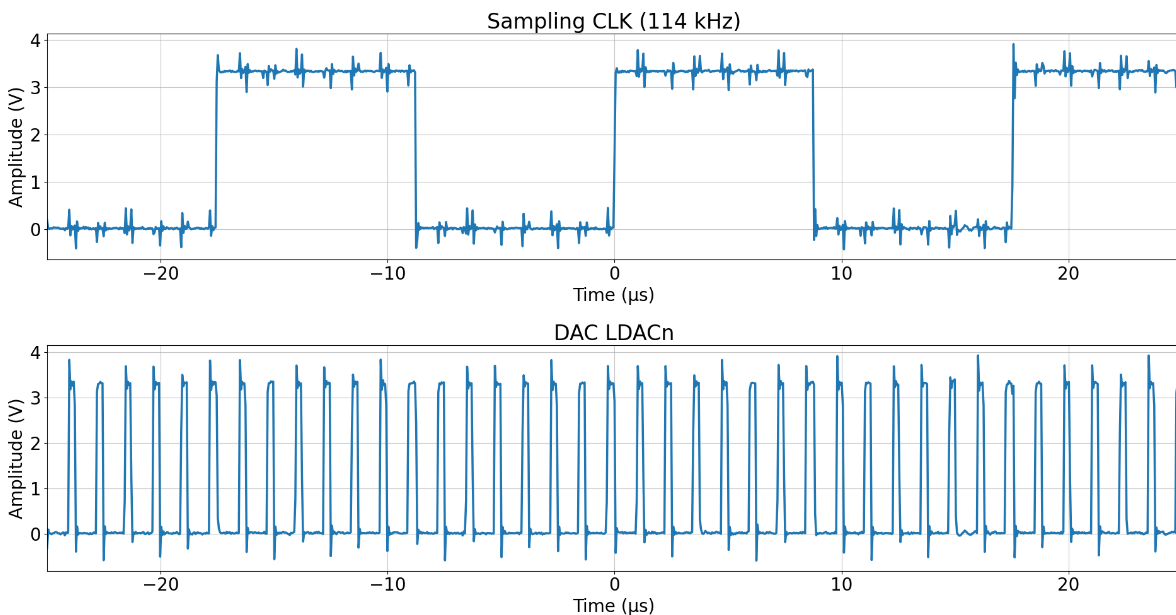


Figure 4.15: DAC maximum communication frequency

The maximum communication speed achievable is 114 kSPS, limited by the seven

transmissions needed for each conversion cycle. However, the DAQ has a typical 10 μs settling time [55], limiting the maximum generation speed as the slew rate will not be enough for signals over 100 kSPS. Therefore, the design is capable of reaching the maximum generation speed of the DAC successfully.

As a result, the maximum speed is limited by the characteristics of both devices, allowing a maximum acquisition, processing, and generation frequency of 100 kSPS or up to 200 kSPS if the samples are acquired, processed, and sent to PCIe without using the DAC.

4.2.3 C++ Library results

A C++ test application has been developed with the library. It is designed to be used with a width adapter from an ADC channel to the PCIe C2H interface and another width adapter from the PCIe H2C interface to a DAC channel.

This application reads 4096 samples from the C2H interface, displays them with a Python script, and writes 4096 samples of a square wave signal to the H2C interface. This can be used to test the library's streaming capabilities. For example, with a sine wave connected to the input, the application displays the results found on Figure 4.16 and outputs the signal found on Figure 4.17 (measured with Red Pitaya).

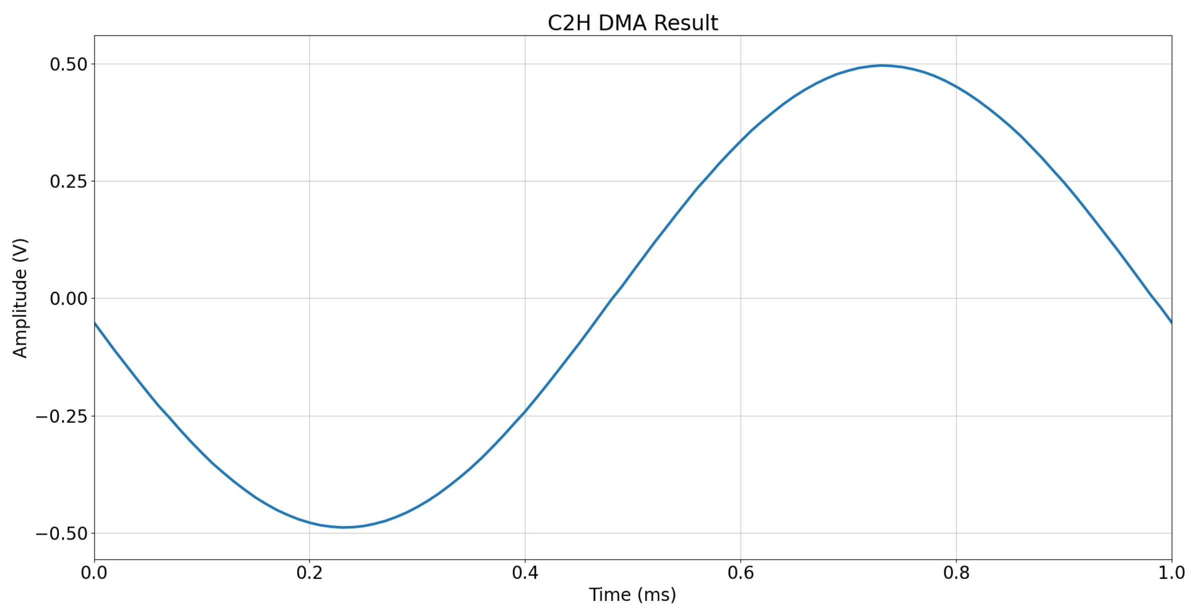


Figure 4.16: PCIe C2H streaming result

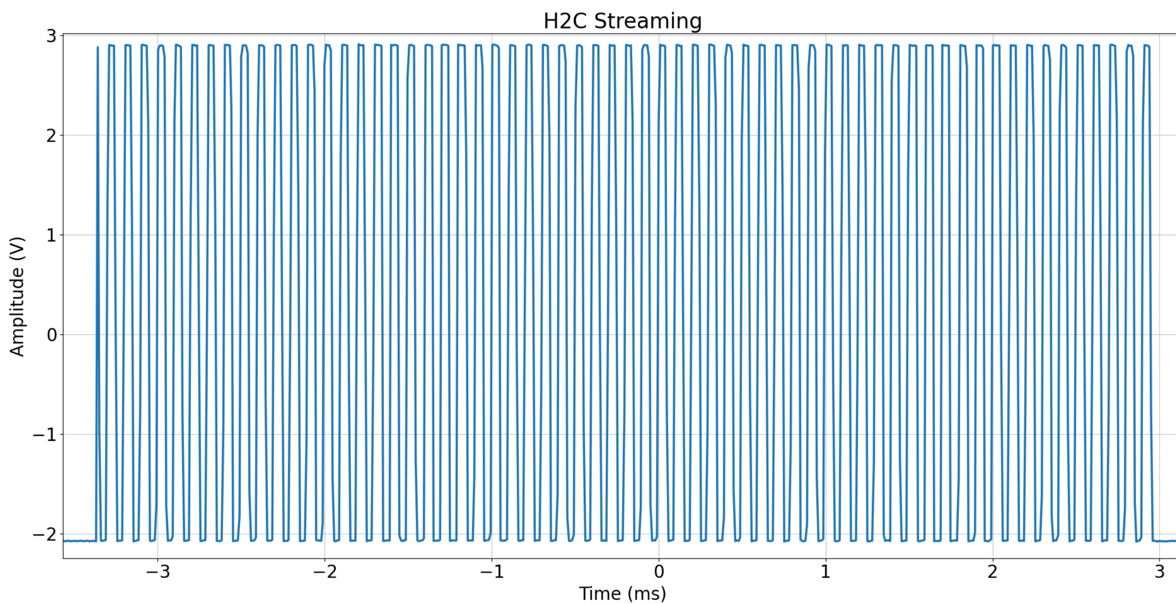


Figure 4.17: PCIe H2C streaming result

The implementation of this test shows the ease of use of the designed library. One C2H read can be performed with only three instructions, and a H2C write is performed in just one, with no need for file descriptor manipulation or memory management (apart from the origin and destination buffers). The code for these operations can be found in Listing 4.2 and Listing 4.1.

Listing 4.1: C++ Library read example

```
1  int64_t c2h_buf[dma_bufsize_i64];
2  tmpe_daq t("/dev/xdma0_user", "/dev/xdma0_c2h_0",
       "/dev/xdma0_h2c_0");
3  t.enable_c2h_dma();
4  t.read_c2h_dma(c2h_buf, dma_bufsize_i64);
5  t.disable_c2h_dma();
```

Listing 4.2: C++ Library write example

```
1  int64_t* h2c_buf;
2  tmpe_daq t("/dev/xdma0_user", "/dev/xdma0_c2h_0",
       "/dev/xdma0_h2c_0");
3  posix_memalign(reinterpret_cast<void **>(&h2c_buf),
       page_alignment, dma_bufsize_bytes);
4  t.write_h2c_dma(h2c_buf, dma_bufsize_i64);
```

4.3 Design resource usage

To generate an estimate of resource utilization of this project, the HLS implementation used contains two width adapters (ADC to PCIe and PCIe to DAC) and three FIR filters on the rest of the ADC and DAC channels. After synthesis and implementation, the usage results can be seen in Table 4.1 and Figure 4.18. The meaning and purpose of each resource can be found in the Configurable Logic Block User Guide [81].

The utilization percent does not exceed 40 % in any category (except `PCIe` because there is only one). This means that many device resources are unused and can be dedicated to implementing a more advanced processing kernel.

Table 4.1: Implementation resource usage

Resource type	Utilization	Available	Utilization %
`LUT`	12218	32600	37.48
`LUTRAM`	1212	9600	12.63
`FF`	16846	65200	25.84
`BRAM`	24	75	32.00
`DSP`	3	120	2.50
`IO`	49	148	33.11
`GT`	1	4	25.00
`BUFG`	5	32	15.63
`MMCM`	1	5	20.00
`PCIe`	1	1	100.00

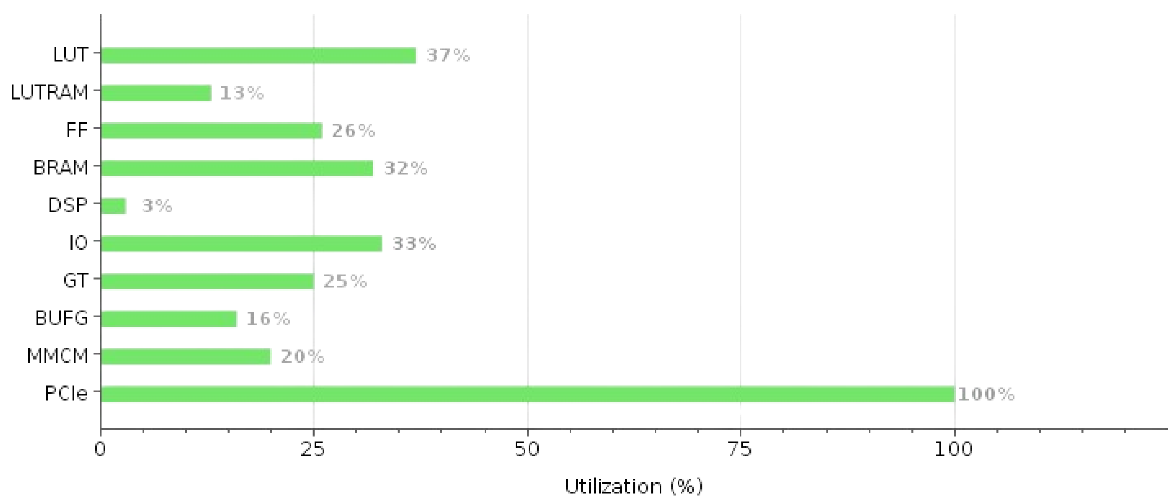


Figure 4.18: Design implementation usage histogram

Appendix C includes the resulting design floor plan, containing a top-down view of the implemented results.

The Vitis HLS tool reports usage of each independent algorithm implementation after synthesis. The results of each algorithm can be found in Table 4.2. Both rolled and unrolled implementations of the FIR filter are included. The rolled version uses fewer resources but takes 41 cycles to process one sample, while the unrolled version uses more resources but takes only six cycles to complete the processing.

Table 4.2: HLS resource usage

Algorithm	**Latency (cycles)**	**DSP**	**FF**	**LUT**
FIR filter (rolled loop)	41	1	1211	405
FIR filter (unrolled loop)	6	32	850	2591
IIR filter	6	3	694	644
Width adapter	1	0	3	44

With these values, the usage of the base design can be calculated by subtracting the algorithm usage from the values on Table 4.1, obtaining the values from Table 4.3

Table 4.3: Approximated usage of empty base design

Resource type	**Utilization**	**Available**	**Utilization %**
`LUT`	10915	32600	33.48
`FF`	13207	65200	20.25
`DSP`	0	120	0

The Vitis HLS tool also offers the Schedule Viewer tool that can be used to analyze the implementation of the algorithm's parallelism. Taking the rolled FIR filter as an example, Figure 4.19 shows the shift register performing an information displacement. The shift is divided into the 32 individual information transfers, but as seen in the Schedule Viewer, they are performed simultaneously.

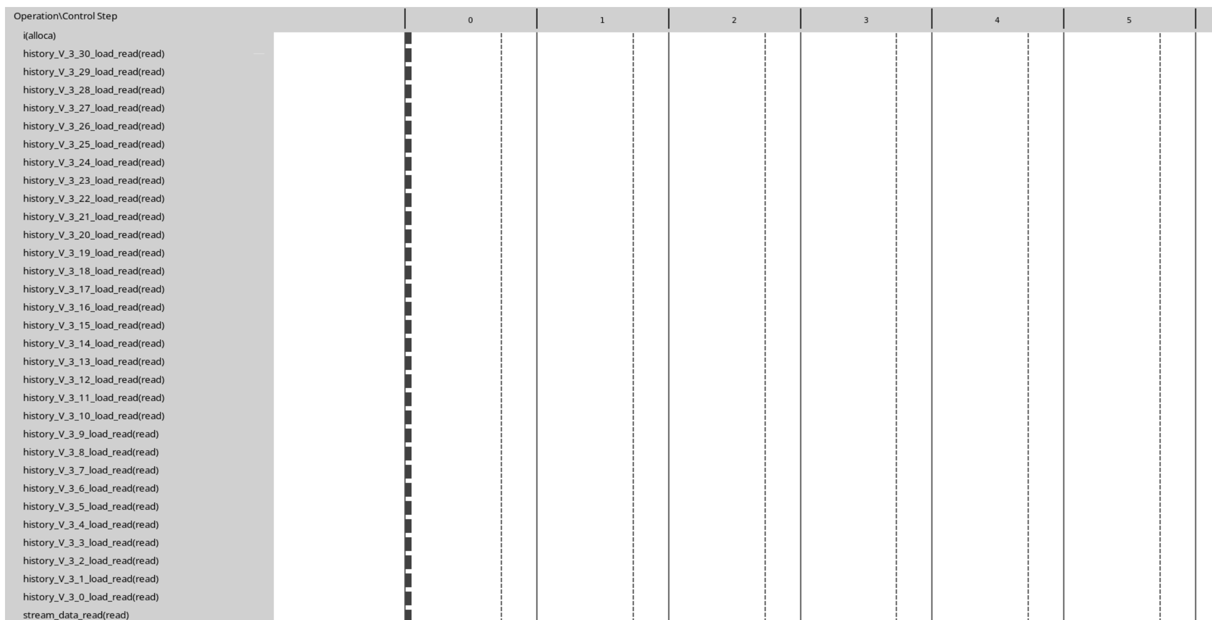


Figure 4.19: FIR filter shift register schedule viewer

On the other hand, Figure 4.20 shows the convolution operation, composed of some sequential steps. These steps are repeated once for each coefficient sequentially, as seen on the approximate staircase shape.

However, the iterations are pipelined as seen by the `ii=1` annotation on the top. This means that one iteration can start before the previous one has completed, overlapping different substeps. Pipelining is only possible because none of the iterations depend on the results of the previous one. This optimization speeds up the operations by a significant factor and is performed automatically by the HLS synthesizer.

Figure 4.21 shows a simplified diagram of loop pipelining. The loop initialization interval (`ii`) is the delay in clock cycles between loop iterations, measured between two consecutive iteration starts.

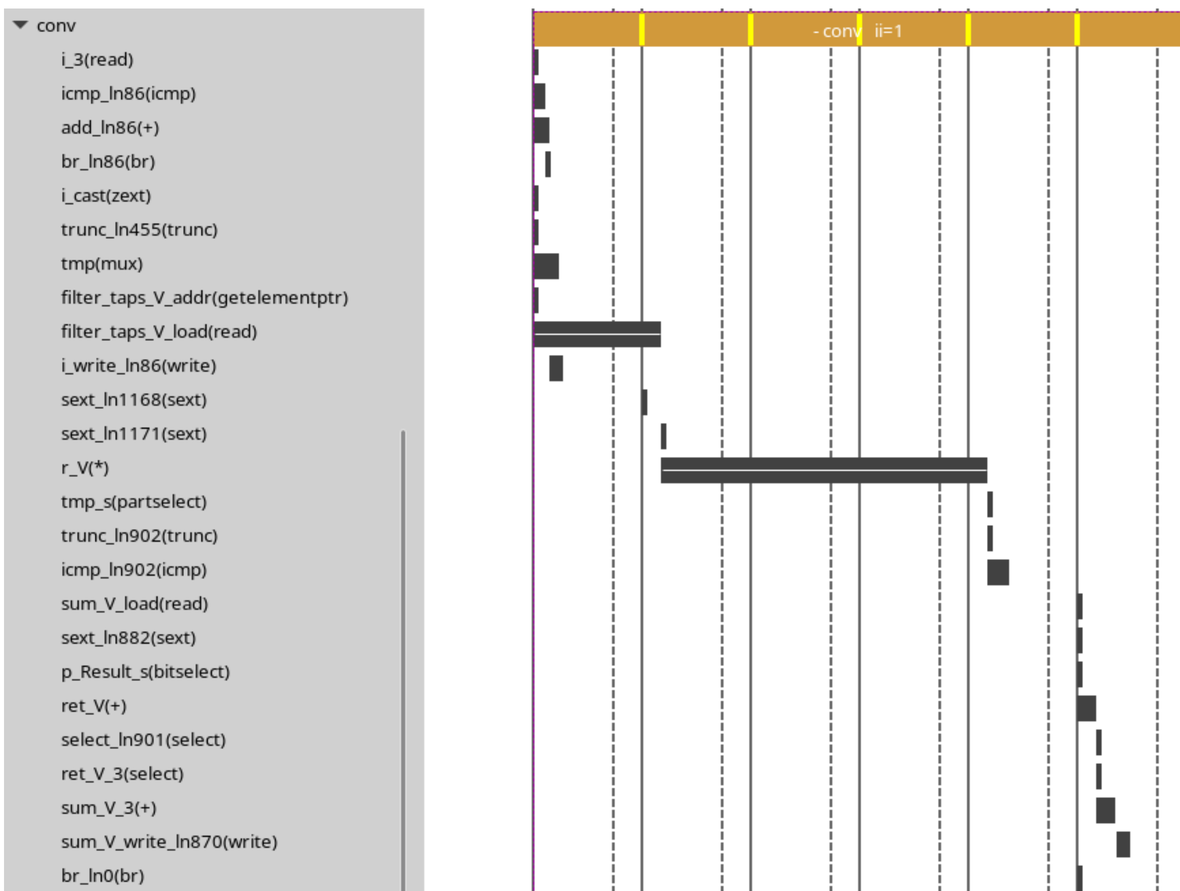


Figure 4.20: FIR filter convolution schedule viewer

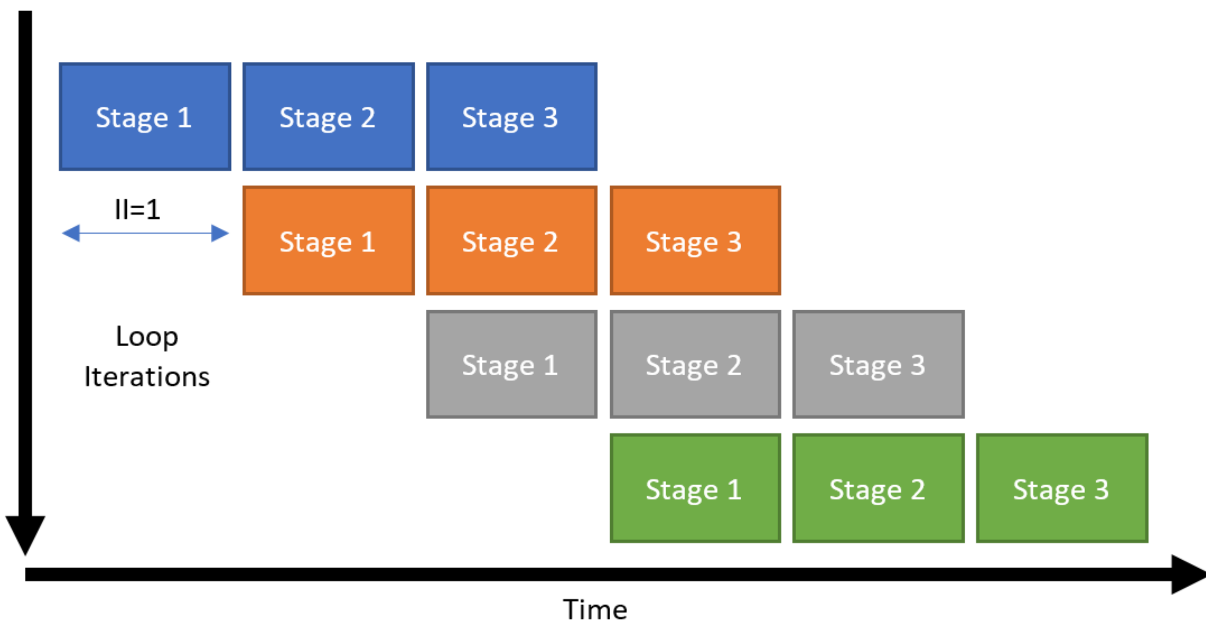


Figure 4.21: Loop pipelining diagram [82]

Chapter 5

Budget

This project was developed over 300 work hours by a final-year Electronics Engineering student. The average salary for a junior Electronic Engineer in Spain is around €24000 per year [83], equivalent to approximately €11.54 per hour.

With an approximate 33 % of Social Security upcharge and a 50 % of enterprise cost overhead, the result is approximately €23.00 per hour.

The development of this project used the hardware elements presented in subsection 3.1.1 and the Red Pitaya Diagnostic Kit for debugging and result verification [80].

The Vivado Software Suite has been provided by the university department. It is acquired on a license pack with other software. The proportional part of the Vivado Design Suite sums up to €700.

The final budget for this project is summarized in Table 5.1.

Table 5.1: Project budget

Item	Quantity	Unit Cost (€)	Total (€)
Human costs			
Junior Electronics Engineer Hour	300	23.00	6900.00
Hardware			
TMPE627-10R	1	1156.67	1156.67
TA308-10R	1	92.50	92.50
TA309-10R	1	372.50	372.50
Platform Cable USB II	1	294.24	294.24
mPCIe adapter	1	70.07	70.07
Red Pitaya Diagnostic Kit	1	641.30	641.30
Software			
Vivado Design Suite	1	700.0	700.0
Total			10227.28

Chapter 6

Project impact

This project has a clear industrial scope. It aims to present an open, academic alternative to a closed enterprise family of products. It has been developed with the scope of being integrated into big science projects, especially ITER.

The impact of the project can be classified into the following aspects:

- Environmental impact: Nuclear fusion has a near-zero carbon footprint, and its success would reduce the need for fossil fuels and fission to a historical minimum, greatly contributing to the reduction of the greenhouse effect and global warming. The generation of tritium is an environmentally critical task with some risk that accurate sensors can help to decrease.
- Health impact: Unlike fossil fuels or nuclear fission, fusion does not generate air pollutants or permanent radioactive waste. This could seriously improve the health of the people living near power generation plants and avoid the necessity of nuclear entombment.
- Security impact: In big science projects, security is always a concern because of the past catastrophes related to nuclear science. However, precise and fast technologies like the one presented in this project can help reduce the risk present in these experiments.
- Technological impact: The designed architecture allows for a very flexible and easy implementation of algorithms. This allows non-specifically trained people to implement algorithms in FPGAs, which are usually inaccessible.
- Economic impact: The implementation of this project could help decrease the investment in the tokamak, as the current technologies are very expensive due to the proprietary nature of the products and the low variety available.

6.1 Sustainable Development Goals

This project is directly correlated to two Sustainable Development Goals (SDGs):

- SDG 7 - Affordable and Clean Energy. Ensure access to affordable, reliable, sustainable, and modern energy for all.
- SDG 9 - Industry, Innovation and Infrastructure. Build resilient infrastructure, promote inclusive and sustainable industrialization, and foster innovation.

It can also be indirectly related to two more SDGs:

- SDG 3 - Good Health and Well-being. Ensure healthy lives and promote well-being for all at all ages.
- SDG 13 - Climate Action. Take urgent action to combat climate change and its impacts.

SDG 7 is a direct target of the ITER project, aiming to eliminate the need for fossil and fission fuels and provide a clean way to produce energy globally. It also impacts SDG 9 because of the industrial revolution that nuclear fusion could mean, and the technological impulse it provides to the state of the art of instrumentation and physics.

SDGs 3 and 13 are a consequence of the zero carbon footprint and no-waste energy generation, improving the air quality of cities and reducing the greenhouse effect.

Chapter 7

Conclusions and next steps

7.1 Conclusions

The development of this project offers an open and accessible solution for the implementation of signal acquisition, processing, and generation with FPGA devices. The project is designed with a generic implementation in mind, allowing most algorithms to be programmed. It also allows communication with the computer for both configuration and data transfer.

The design is accessible in two ways: A person without HDL knowledge can use it to implement acquisition, processing, and generation with only HLS. On the other hand, an FPGA engineer can also use it as a base for design, removing the need to study and design the communication with the devices and the computer, synchronization, etc.

The project's design phase proved crucial to its success, where selecting the correct architecture helped avoid iterations on the integration phase. The implementation phase was very streamlined thanks to the design, training material, and documentation provided by Xilinx. Lastly, the results phase was trivial thanks to the tools provided by the research group.

Therefore, this project could be used for both data acquisition and processing or hardware acceleration of algorithms, making it a versatile approach to computer peripherals. The design has been implemented on a micro PCIe board installed inside a computer, and some example algorithms have been designed. It only depends on the open-source Linux drivers made by Xilinx. Because of this, the project is highly maintainable as they should update these drivers with each kernel release.

The device allows acquisition, processing, and generation up to 100 kSPS and algorithm acceleration with the PCIe interface or 200 kSPS acquisition and processing. Around 60% of the resources of the board are not used and are free for the implementation of HLS modules, offering high flexibility.

Some simple algorithms were implemented and validated to check the HLS imple-

mentation, being able to implement filters in less than 20 lines of code. A Python tool and a C++ library have been designed to further simplify the usage of the device, just needing to implement the algorithm, install the drivers, and use the tools to start the processing.

This device and its companion tools can provide a very useful tool for Big Science projects like ITER, where precise and real-time acquisition is a must, both for performance and safety concerns.

7.2 Next Steps

This project defines the generic framework for algorithm implementation, but only some simple filtering algorithms have been developed. Some more complex algorithms should be implemented to test the limits of the design in terms of footprint and performance.

Regarding the verification methodology, only basic VHDL tools were used. The system could benefit from formal verification tools like those in SystemVerilog. This kind of verification is generally a requirement in fields like safety or investment protection.

This project is highly integrable thanks to the flexible algorithm implementation and the I/O. Therefore, it would be interesting to implement the system with Real-time Linux features (`PREEMPT_RT` patch) or Time-Sensitive Networks (TSN). With these integrations, accurate latency and performance measurements could be implemented to test the device's real-time characteristics.

These integrations would prove very useful in the Big Science environment. It could be used, for example, to monitor real-time values with accurate timestamping or for fault detection with immediate notification and even mitigation actions. The system could also be integrated into a control environment like CODAC [8] and EPICS [9].

Chapter 8

References

[1] *ITER - the way to new energy*. Accessed: Mar. 21, 2025. [Online]. Available: https://www.iter.org/node.

[2] Eduardo Oliva Gonzalo *et al.*, *Curso básico de fusión nuclear*. Madrid: Senda, 2017, ISBN: 978-84-697-5718-5.

[3] N. Mandell, *Magnetic Fluctuations in Gyrokinetic Simulations of Tokamak Scrape-Off Layer Turbulence*, 2021. DOI: 10.48550/ARXIV.2103.16062. Accessed: Mar. 27, 2025. [Online]. Available: https://arxiv.org/abs/2103.16062.

[4] *What will ITER do?* Nov. 2023. Accessed: Mar. 27, 2025. [Online]. Available: https://www.iter.org/fusion-energy/what-will-iter-do.

[5] ITER, *Advantages of fusion*, Nov. 2023. Accessed: Mar. 21, 2025. [Online]. Available: https://www.iter.org/fusion-energy/advantages-fusion.

[6] *History of Fusion*. Accessed: Apr. 21, 2025. [Online]. Available: https://euro-fusion.org/fusion/history-of-fusion/.

[7] David Szondy, *France runs fusion reactor for record 22 minutes*, Feb. 2025. Accessed: Mar. 27, 2025. [Online]. Available: https://newatlas.com/energy/france-tokamak-cea-west-fusion-reactor-record-plasma-duration/.

[8] *CODAC*, Jun. 2023. Accessed: Mar. 27, 2025. [Online]. Available: https://www.iter.org/machine/supporting-systems/codac.

[9] Argonne National Laboratory, *EPICS - Experimental Physics and Industrial Control System*. Accessed: Mar. 27, 2025. [Online]. Available: https://epics.anl.gov/.

[10] *Magnets*, Jun. 2023. Accessed: Apr. 23, 2025. [Online]. Available: https://www.iter.org/machine/magnets.

[11] G. Ochoa-Ruiz, "A High-level Methodology for Automatically Generating Dynamically Reconfigurable Systems using IP-XACT and the UML MARTE Profile," Ph.D. dissertation, Nov. 2013. Accessed: Mar. 25, 2025. [Online]. Available: https://www.researchgate.net/publication/265125404_A_High-level_Methodology_for_Automatically_Generating_Dynamically_Reconfigurable_Systems_using_IP-XACT_and_the_UML_MARTE_Profile.

[12] *Intel Completes Acquisition of Altera*, Dec. 2015. Accessed: Mar. 25, 2025. [Online]. Available: https://www.intc.com/news-events/press-releases/detail/302/intel-completes-acquisition-of-altera.

[13] *AMD Completes Acquisition of Xilinx*, Feb. 2022. Accessed: Mar. 25, 2025. [Online]. Available: https://ir.amd.com/news-events/press-releases/detail/1047/amd-completes-acquisition-of-xilinx.

[14] *Lattice Semiconductors*. Accessed: Apr. 21, 2025. [Online]. Available: https://www.latticesemi.com/.

[15] *Microchip*. Accessed: Apr. 21, 2025. [Online]. Available: https://www.microchip.com/.

[16] *GOWIN Semiconductor*. Accessed: Apr. 21, 2025. [Online]. Available: https://www.gowinsemi.com/en/.

[17] *AMD Artix 7 FPGAs*. Accessed: Mar. 25, 2025. [Online]. Available: https://www.amd.com/en/products/adaptive-socs-and-fpgas/fpga/artix-7.html.

[18] *NI RIO Platform*. Accessed: Mar. 21, 2025. [Online]. Available: https://wireflow.com/ni-rio-platform/.

[19] *CompactRIO Systems (cRIO)*. Accessed: Mar. 21, 2025. [Online]. Available: https://www.ni.com/en/shop/compactrio.html.

[20] *What Is FlexRIO?* Accessed: Mar. 21, 2025. [Online]. Available: https://www.ni.com/en/shop/electronic-test-instrumentation/flexrio/what-is-flexrio.html.

[21] *Introduction to the New FlexRIO Modules with Xilinx Kintex UltraScale*. Accessed: Apr. 24, 2025. [Online]. Available: https://www.ni.com/en/shop/electronic-test-instrumentation/flexrio/what-is-flexrio/introduction-to-the-new-flexrio-modules-with-xilinx-kintex-ultra.html.

[22] V. Costa, D. Andrino, and M. Ruiz, *I2a2/irioCoreCpp*, I2A2 instrumentation R&D group, Feb. 2025. Accessed: Mar. 21, 2025. [Online]. Available: https://github.com/i2a2/irioCoreCpp.

[23] National Instruments, *NI-9159*. Accessed: Apr. 4, 2025. [Online]. Available: https://www.ni.com/es-es/support/model.ni-9159.html.

[24] *MicroTCA*. Accessed: Apr. 24, 2025. [Online]. Available: https://www.picmg.org/openstandards/microtca/.

[25] e. Europe, *US microTCA maker sets up European headquarters*, Dec. 2012. Accessed: Apr. 24, 2025. [Online]. Available: https://www.eenewseurope.com/en/us-microtca-maker-sets-up-european-headquarters/.

[26] USB-IF, *USB 2.0 Specification*, Spec Ification, Sep. 2024. Accessed: Mar. 25, 2025. [Online]. Available: https://www.usb.org/document-library/usb-20-specification.

[27] National Instruments, *How to Choose the Right Bus for Your Measurement System*, Shop, Jul. 2024. Accessed: Mar. 25, 2025. [Online]. Available: https://www.ni.com/en/shop/data-acquisition/how-to-choose-the-right-bus-for-your-measurement-system.html.

[28] V. Quesada, *Tipos de Conectores USB: Una Guía Completa*, Apr. 2023. Accessed: Mar. 25, 2025. [Online]. Available: https://ecoportatil.es/blog/tipos-conectores-usb.

[29] T. V. Wilson, *How PCIe Works*, Aug. 2005. Accessed: Mar. 25, 2025. [Online]. Available: https://computer.howstuffworks.com/pci-express.htm.

[30] *The PCIe 7.0 Specification, Version 0.9 is Now Available to Members | PCI-SIG*. Accessed: Apr. 30, 2025. [Online]. Available: https://pcisig.com/specifications/pcie-70-specification-version-09-now-available-members.

[31] HP, *PCIe Slots: Everything You Need to Know*. Accessed: Mar. 25, 2025. [Online]. Available: https://www.hp.com/us-en/shop/tech-takes/what-are-pcie-slots-pc.

[32] National Instruments, *PXI Specification Standards Explained*, Jan. 2025. Accessed: Mar. 25, 2025. [Online]. Available: https://www.ni.com/en/shop/pxi/pxi-specification-standards-explained.html.

[33] SPECTRUM Instrumentation, *Introduction to PXIe and PXI*, Product Note. Accessed: Mar. 25, 2025. [Online]. Available: https://spectrum-instrumentation.com/applications/product_notes/PN_Introduction_to_PXIe_and_PXI.php.

[34] National Instruments, *Introducción a la arquitectura PXI*. Accessed: Mar. 25, 2025. [Online]. Available: https://www.ni.com/es/shop/pxi/introduction-to-the-pxi-architecture.html.

[35] Doulos, *A Brief History of VHDL*. Accessed: Mar. 26, 2025. [Online]. Available: https://www.doulos.com/knowhow/vhdl/a-brief-history-of-vhdl/.

[36] Deepkak Kumar Tala, *History Of Verilog*. Accessed: Mar. 26, 2025. [Online]. Available: https://www.asic-world.com/verilog/history.html.

[37] F. Vahid, *Digital Design with RTL Design, VHDL, and Verilog*. John Wiley & Sons, Mar. 2010, ISBN: 978-0-470-53108-2.

[38] *Vera*. Accessed: Apr. 24, 2025. [Online]. Available: https://semiengineering.com/knowledge_centers/languages/vera/.

[39] S. Sutherland and D. Mills, "HDVL += (HDL & HVL) SystemVerilog 3.," 2003.

[40] *OpenCL - The Open Standard for Parallel Programming of Heterogeneous Systems*, Jul. 2013. Accessed: Mar. 26, 2025. [Online]. Available: https://www.khronos.org/opencl/.

[41] *AMD Vitis™ HLS*. Accessed: Mar. 26, 2025. [Online]. Available: https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vitis/vitis-hls.html.

[42] *AMBA 5*, 2013. Accessed: Mar. 26, 2025. [Online]. Available: https://developer.arm.com/Architectures/AMBA.

[43] ARM, *AMBA® AXI Protocol Specification*, Sep. 2023. [Online]. Available: https://developer.arm.com/documentation/ihi0022/latest/.

[44] *AMBA AXI-Stream Protocol Specification*, AMBA, Apr. 2021.

[45] M. Ruiz *et al.*, "Real Time Plasma Disruptions Detection in JET Implemented With the ITMS Platform Using FPGA Based IDAQ," *IEEE Transactions on Nuclear Science*, vol. 58, no. 4, pp. 1576–1581, Aug. 2011, ISSN: 1558-1578. DOI: 10.1109/TNS.2011.2158552. Accessed: Mar. 26, 2025. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/5954198.

[46] M. Astrain, M. Ruiz, A. Carpeño, S. Esquembri, E. Barrera, and J. Vega, "A methodology to standardize the development of FPGA-based high-performance DAQ and processing systems using OpenCL," *Fusion Engineering and Design*, vol. 155, p. 111 561, Jun. 2020, ISSN: 0920-3796. DOI: 10.1016/j.fusengdes.2020.

111561. Accessed: Apr. 24, 2025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0920379620301095.

[47] M. Astrain *et al.*, "Real-Time Implementation of the Neutron/Gamma Discrimination in an FPGA-Based DAQ MTCA Platform Using a Convolutional Neural Network," *IEEE Transactions on Nuclear Science*, vol. 68, no. 8, pp. 2173–2178, Aug. 2021, ISSN: 1558-1578. DOI: 10.1109/TNS.2021.3090670. Accessed: Apr. 24, 2025. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9459756.

[48] S. Esquembri *et al.*, "Application of Heterogeneous Computing Techniques for the Development of an Image-Based Hot Spot Detection System Using MTCA," *IEEE Transactions on Nuclear Science*, vol. 68, no. 8, pp. 2151–2158, Aug. 2021, ISSN: 1558-1578. DOI: 10.1109/TNS.2021.3087124. Accessed: Apr. 24, 2025. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9448106.

[49] K. Patel *et al.*, "LabVIEW-FPGA-Based Real-Time Data Acquisition System for ADITYA-U Heterodyne Interferometry," *IEEE Transactions on Plasma Science*, vol. 49, no. 6, pp. 1891–1897, Jun. 2021, ISSN: 1939-9375. DOI: 10.1109/TPS.2021.3082159. Accessed: Mar. 26, 2025. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9442107.

[50] I. Firmansyah and Y. Yamaguchi, "OpenCL Implementation of FPGA-Based Signal Generation and Measurement," *IEEE Access*, vol. 7, pp. 48 849–48 859, 2019, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2910391. Accessed: Mar. 26, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/8686071/?arnumber=8686071.

[51] A. M. Ahmad, P. Lukowicz, and J. Cheng, "FPGA based hardware acceleration of sensor matrix," in *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct*, Heidelberg Germany: ACM, Sep. 2016, pp. 793–802, ISBN: 978-1-4503-4462-3. DOI: 10.1145/2968219.2968289. Accessed: Mar. 26, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/2968219.2968289.

[52] TEWS Technologies, "TMPE627," Tech. Rep. Issue 1.0.5, Feb. 2020. [Online]. Available: https://www.tews.com/wp-content/uploads/TMPE627_DS_1.0.5.pdf.

[53] XILINX, "7 Series FPGAs Data Sheet: Overview," Tech. Rep. DS180, 2020. [Online]. Available: https://docs.amd.com/v/u/en-US/ds180_7Series_Overview.

[54] Linear Technology, "LTC2348-18," Datasheet Rev. A, Feb. 2016.

[55] Analog Devices, "AD5724R-AD5734R-AD5754R," Datasheet Rev. G, Feb. 2017.

[56] TEWS Technologies, *TA309 | Cable Kit for Modules with Pico-Clasp Connector*. Accessed: Mar. 12, 2025. [Online]. Available: https://www.tews.com/products/accessories/ta309/.

[57] AMD, "Platform Cable USB II," Tech. Rep. Accessed: Mar. 12, 2025. [Online]. Available: https://www.amd.com/en/products/adaptive-socs-and-fpgas/board-accessories/hw-usb-ii-g.html.

[58] TEWS Technologies, "TA308 | Cable Kit for Modules with XRS Debug Connector," Tech. Rep. Accessed: Mar. 12, 2025. [Online]. Available: https://www.tews.com/products/accessories/ta308/.

[59] *PCIe-mPCIe Adapter*. Accessed: May 6, 2025. [Online]. Available: https://accesio.com/product/pcie-mpcie/.

[60] TEWS Technologies, "TMPE627 FPGA Board Reference Design User Manual," Tech. Rep. Issue 1.0.2, Apr. 2018.

[61] XILINX, "XPM CDC Generator v1.0 LogiCORE IP," Tech. Rep. PG382, Feb. 2021. Accessed: Mar. 13, 2025. [Online]. Available: https://docs.amd.com/r/en-US/pg382-xpm-cdc-generator.

[62] XILINX, "LogiCORE IP FIFO Generator," Tech. Rep. DS317, Apr. 2012. Accessed: Mar. 14, 2025. [Online]. Available: https://docs.amd.com/api/khub/documents/W2Dj6hQtp1T6Ia1AHc1MYA/content?Ft-Calling-App=ft%2Fturnkey-portal&Ft-Calling-App-Version=5.0.61#.

[63] XILINX, "DMA/Bridge Subsystem for PCI Express v4.1 Product Guide," Tech. Rep. PG195, Dec. 2024. [Online]. Available: https://www.amd.com/content/dam/xilinx/support/documents/ip_documentation/xdma/v4_1/pg195-pcie-dma.pdf.

[64] XILINX, *Dma_ip_drivers*, Xilinx, Mar. 2025. Accessed: Mar. 14, 2025. [Online]. Available: https://github.com/Xilinx/dma_ip_drivers.

[65] A. V. Oppenheim, A. S. Willsky, and S. H. Nawab, "Nonrecursive Discrete-Time Filters," in *Signals and Systems*, ser. Prentice-Hall Signal Processing Series, 2. ed, Upper Saddle River, NJ: Prentice Hall, 1997, pp. 245–249, ISBN: 978-0-13-814757-0.

[66] *TFilter - Free online FIR filter design*. Accessed: Mar. 14, 2025. [Online]. Available: http://t-filter.engineerjs.com/.

[67] N. Zhang, *Loop Optimization in HLS*, Oct. 2020. Accessed: Mar. 14, 2025. [Online]. Available: https://www.zzzdavid.tech/loop_opt/index.html.

[68] A. N. Sloss, D. Symes, and C. Wright, "DIGITAL SIGNAL PROCESSING," in *ARM System Developer's Guide*, Elsevier, 2004, pp. 258–314, ISBN: 978-1-55860-874-0. DOI: 10.1016/B978-155860874-0/50009-5. Accessed: Mar. 14, 2025. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/B9781558608740500095.

[69] WaveWalkerDSP, *Single Pole IIR Filter: Substitute for Moving Average Filter*, Aug. 2022. Accessed: Mar. 14, 2025. [Online]. Available: https://www.wavewalkerdsp.com/2022/08/10/single-pole-iir-filter-substitute-for-moving-average-filter/.

[70] *Tkinter — Python interface to Tcl/Tk*. Accessed: Mar. 17, 2025. [Online]. Available: https://docs.python.org/3/library/tkinter.html.

[71] B. Stroustrup, *The C++ Programming Language: C++ 11*, 4. ed., 4. print. Upper Saddle River, NJ: Addison-Wesley, 2015, ISBN: 978-0-321-56384-2.

[72] *Make - GNU Project - Free Software Foundation*. Accessed: Mar. 18, 2025. [Online]. Available: https://www.gnu.org/software/make/.

[73] *Git*. Accessed: Mar. 18, 2025. [Online]. Available: https://git-scm.com/.

[74] *GitHub*. Accessed: Mar. 18, 2025. [Online]. Available: https://github.com.

[75] Andrino, David, *Tmpe-daq*. Accessed: Apr. 21, 2025. [Online]. Available: https://github.com/David-Andrino/tmpe-daq.

[76] *Doxygen*. Accessed: Apr. 1, 2025. [Online]. Available: https://www.doxygen.nl/.

[77] *Sphinx*. Accessed: Apr. 1, 2025. [Online]. Available: https://www.sphinx-doc.org/en/master/.
[78] *reStructuredText*, Oct. 2024. Accessed: Apr. 1, 2025. [Online]. Available: https://docutils.sourceforge.io/rst.html.
[79] *LaTeX - A document preparation system*. Accessed: Apr. 21, 2025. [Online]. Available: https://www.latex-project.org/.
[80] *STEMlab 125-14 Diagnostic Kit - Red Pitaya*. Accessed: Mar. 14, 2025. [Online]. Available: https://redpitaya.com/product/stemlab-125-14-diagnostic-kit/.
[81] XILINX, "7 Series FPGAs Configurable Logic Block User Guide," Tech. Rep. UG474, 2016.
[82] Intel, *Pipeline Loops*. Accessed: Mar. 19, 2025. [Online]. Available: https://www.intel.com/content/www/us/en/docs/programmable/683259/19-1/pipeline-loops.html.
[83] *Sueldo: Ingeniero Electronico Junior en España 2025*. Accessed: Mar. 14, 2025. [Online]. Available: https://www.glassdoor.es/Sueldos/ingeniero-electronico-junior-sueldo-SRCH_KO0,28.htm.

Chapter 9

Bibliography

[84] P. J. Ashenden, *The Designer's Guide to VHDL* (The Morgan Kaufmann Series in Systems on Silicon), 3rd ed. Amsterdam Boston: Morgan Kaufmann Publishers, 2008, ISBN: 978-0-12-088785-9.

[85] J. Corbet, A. Rubini, G. Kroah-Hartman, and A. Rubini, *Linux Device Drivers*, 3rd ed. Beijing ; Sebastopol, CA: O'Reilly, 2005, ISBN: 978-0-596-00590-0.

[86] Fabrizio Tappero and Bryan Mealy, *Free Range VHDL 2019 Edition*. Jul. 2019. Accessed: Nov. 12, 2024. [Online]. Available: `http://archive.org/details/free_range_vhdl_2019`.

[87] *Down to the TLP: How PCI express devices talk (Part I)*. Accessed: Oct. 2, 2024. [Online]. Available: `https://xillybus.com/tutorials/pci-express-tlp-pcie-primer-tutorial-guide-1`.

[88] S. H. last updated, *What Is PCIe? A Basic Definition*, May 2022. Accessed: Apr. 30, 2025. [Online]. Available: `https://www.tomshardware.com/reviews/pcie-definition,5754.html`.

[89] XILINX, *Vivado Design Suite 7 Series FPGA and Zynq 7000 SoC Libraries Guide*, 2024. [Online]. Available: `https://docs.amd.com/viewer/book-attachment/Lz7t3FLJuzYlv9pBdksO6Q/ob7lIrXtxRMJLY4I6UEprg`.

Appendix A

Vivado block design

Figure A.1: Block design of the FPGA implementation

Appendix B

HLS implementations

Listing B.1: Bit width adapter implementation in HLS

```
1  /**
2   * @file width_adapter.h
3   * @author David Andrino - I2A2 UPM
4   * @brief Functions for AXI-Stream width adaptation
5   * @date 12-03-2025
6   */
7  #include "hls_dsp.h"
8  #include "hls_stream.h"
9  #include "ap_axi_sdata.h"
10 
11 /**
12  * @brief Adapt data from the data_t to the pci_data_t (longer)
        format
13  *
14  * @tparam N Integer to differentiate function calls
15  * @param in Input data stream
16  * @param out Output data stream
17  */
18 template <unsigned int N>
19 void adapt_longer(hls::stream<data_t>& in,
       hls::stream<pci_data_t>& out) {
20         data_t input;
21         pci_data_t output;
22         output.keep = -1;
23         output.strb = 1;
24         output.user = 1;
25 
26         input = in.read();
27         ap_int<16> data = input.data;
28         ap_int<64> data_long = (static_cast<ap_int<64>>(data) <<
               (64 - 16));
29 
30         output.data = data_long;
31 
32         out.write(output);
33 }
34 
35 /**
```

```
36   * @brief Adapt data from the pci_data_t to the data_t (shorter)
        format
37   *
38   * @tparam N Integer to differentiate function calls
39   * @param in Input data stream
40   * @param out Output data stream
41   */
42  template <unsigned int N>
43  void adapt_shorter(hls::stream<pci_data_t>& in,
       hls::stream<data_t>& out) {
44          pci_data_t input;
45          data_t output;
46          output.keep = -1;
47          output.strb = 1;
48          output.user = 1;
49
50          input = in.read();
51          ap_int<64> data = input.data;
52          ap_int<16> data_long = data >> (64 - 16);
53
54          output.data = data_long;
55
56          out.write(output);
57  }
```

Listing B.2: HLS implementation of the FIR filter

```
1   /**
2    * @file fir.h
3    * @brief HLS FIR implementation
4    * @author David Andrino - I2A2 UPM
5    * @date 12-03-2025
6    */
7   #pragma once
8
9   /*
10
11  FIR filter designed with
12   http://t-filter.appspot.com
13
14  sampling frequency: 100000 Hz
15
16  * 0 Hz - 5000 Hz
17    gain = 1
18    desired ripple = 1 dB
```

```
19   actual ripple = 1.0170495015247925 dB
20
21 * 10000 Hz - 50000 Hz
22   gain = 0
23   desired attenuation = -40 dB
24   actual attenuation = -37.38237348417459 dB
25
26 */
27
28 #include "hls_dsp.h"
29 #include "hls_stream.h"
30 #include "ap_axi_sdata.h"
31
32 #define FIR_LPFFILTER_TAP_NUM 32
33
34 // DSP48E1 slices have 25x18 multipliers
35 static ap_fixed<25, 1, AP_RND, AP_WRAP>
       filter_taps[FIR_LPFFILTER_TAP_NUM] = {
36     0.006108859412847285,
37     -0.002605323801273313,
38     -0.0062451173277574415,
39     -0.011510507812646715,
40     -0.017170968826967187,
41     -0.02153234146494143,
42     -0.022652396111731542,
43     -0.018679705670301708,
44     -0.00829571269626522,
45     0.008875747297207777,
46     0.032016397939266805,
47     0.05909097924462758,
48     0.08708755945670506,
49     0.11245700206595499,
50     0.1317082905053255,
51     0.14210760007481849,
52     0.14210760007481849,
53     0.1317082905053255,
54     0.11245700206595499,
55     0.08708755945670506,
56     0.05909097924462758,
57     0.032016397939266805,
58     0.008875747297207777,
59     -0.00829571269626522,
60     -0.018679705670301708,
61     -0.022652396111731542,
62     -0.02153234146494143,
```

```
63      -0.017170968826967187,
64      -0.011510507812646715,
65      -0.0062451173277574415,
66      -0.002605323801273313,
67      0.006108859412847285
68  };
69
70  template<unsigned int N>
71  void fir(hls::stream<data_t>& in, hls::stream<data_t>& out) {
72      static ap_int<16> history[FIR_LPFFILTER_TAP_NUM] = { 0 };
73      #pragma HLS reset variable=history
74      #pragma HLS array_partition dim=0 type=complete
           variable=history
75
76      data_t stream_data = in.read();
77      ap_int<16> data = stream_data.data;
78
79      delay: for (int i = FIR_LPFFILTER_TAP_NUM - 1; i > 0; i--) {
80          #pragma HLS unroll
81          history[i] = history[i - 1];
82      }
83      history[0] = data;
84
85      ap_int<16> sum = 0;
86      conv: for (int i = 0; i < FIR_LPFFILTER_TAP_NUM; i++) {
87          sum += history[i] * filter_taps[i];
88      }
89      stream_data.data = sum;
90
91      out.write(stream_data);
92  }
```

Listing B.3: HLS implementation of the IIR filter

```
1  static const ap_fixed<18, 1, AP_RND, AP_WRAP> iir_alpha = 0.25;
2  static const ap_fixed<18, 1, AP_RND, AP_WRAP>
      iir_one_minus_alpha = 0.75;
3
4  template<unsigned int N>
5  void lpf(hls::stream<pci_data_t>& in, hls::stream<pci_data_t>&
      out) {
6      static ap_int<64> prev = 0;
7      #pragma HLS RESET variable=prev
8
9      pci_data_t tmp = in.read();
```

```
10      ap_int<64> input = tmp.data;
11
12      prev = input * iir_alpha + prev * iir_one_minus_alpha;
13
14      tmp.data = prev;
15      out.write(tmp);
16  }
```

Appendix C

FPGA implementation floor plan

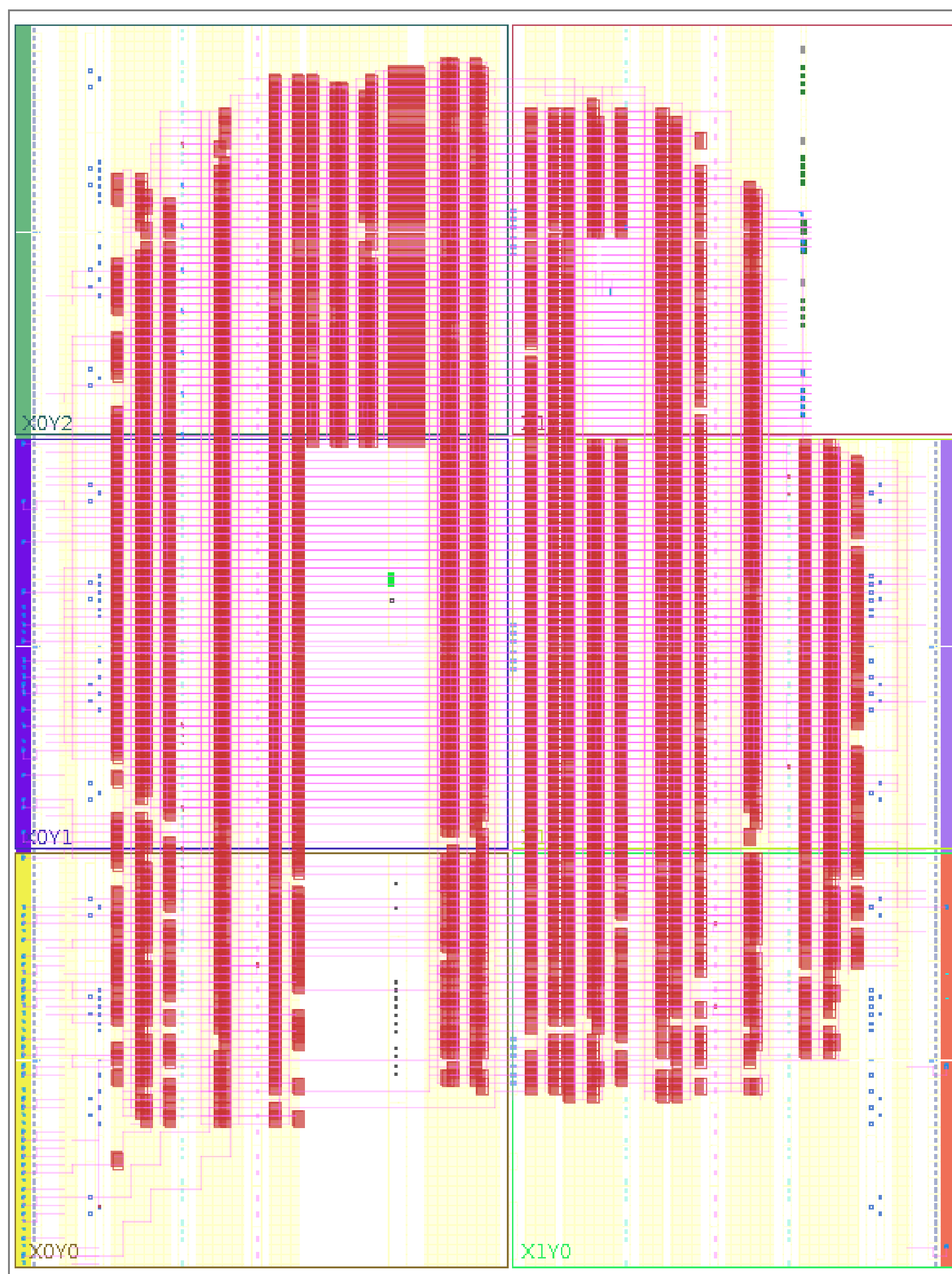


Figure C.1: Floor plan of the FPGA design implementation

Appendix D

Python Bode filter

Listing D.1: Python Bode filter

```
""" Python Bode filter

    This script takes the CSV data from the FILE and performs a moving average of AVERAGE_SAMPLES_AMPS
    samples. The results are plotted, saved to TARGET_FILE and shown on screen

    Author: David Andrino - I2A2 19/03/2025
"""
import csv
import numpy as np
from matplotlib import pyplot as plt
import math

FILE = 'Diagrams/pitaya.csv'
TARGET_FILE = 'images/Results/Pitaya/firBode.png'
AVERAGE_SAMPLES_AMPS = 100

with open(FILE, 'r') as f:
    data = list(csv.reader(f, delimiter=','))[1:]

freqs = [float(x[0]) for x in data]
amplitudes = [float(x[1]) for x in data]

amp_averager = 1 / AVERAGE_SAMPLES_AMPS * np.array([1] * AVERAGE_SAMPLES_AMPS)
freqs_amp = freqs[AVERAGE_SAMPLES_AMPS-1:len(amplitudes)]
filtered_amps = np.convolve(amplitudes, amp_averager, mode='valid')

plt.figure()
plt.plot(freqs_amp, filtered_amps, linewidth=2.5)
plt.title("Bode diagram of FIR filter amplitude")
plt.xlabel('Freq (Hz)')
plt.ylabel('Amplitude (dB)')
plt.xlim(freqs_amp[0], freqs_amp[-1])
plt.xscale('log')
plt.grid(True, which="both", ls="-", alpha=0.7)
```

```
35 plt.yticks(np.sort(np.append(plt.yticks()[0], -3)))
36 plt.axhline(y=-3, color='r', linestyle='--', alpha=0.5)
37
38 plt.savefig(TARGET_FILE, dpi=300)
39 plt.show()
```

Appendix E

User manual

TMPE DAQ

Version 1.0.0

David Andrino I2A2

2

Table of Contents

Contents

1. Installation 4
- 1.1. Create project and load bitfile 4
- 1.2. Xilinx XDMA drivers 4

2. HLS Algorithm implementation 6

3. TMPE DAQ C++ Library Documentation 7
- `tmpe_daq` 7

3

TMPE DAQ

Documentation for the `tmpe_daq` project.

Author: David Andrino

Group: I2A2

Date: April 2025

Installation

Create project and load bitfile

1. Clone the repository and `cd` into it.

```
$ git clone https://github.com/David-Andrino/tmpe-daq
$ cd tmpe-daq
```

2. Use `make` to create the project, compile the bitfile and load it. If Vivado 2021.1 is not installed on `/opt/Xilinx/Vivado/2021.1` , define the `VIVADO_DIR` environment variable to its location.

```
$ export VIVADO_DIR=~/Xilinx/Vivado/2021.1
$ make
```

Alternatively, use the `project` target to generate the project files and use Vivado standard workflow to implement and load the bitfile.

```
$ make project
$ vivado vivado/tmpe_daq.xpr
```

3. Reboot the computer to allow PCI configuration.

```
$ sudo reboot
```

Xilinx XDMA drivers

1. Clone the official Xilinx drivers

```
$ git clone https://github.com/Xilinx/dma_ip_drivers
$ cd dma_ip_drivers
```

2. Compile and install the kernel module

```
$ cd XDMA/linux-kernel/xdma
$ make install
```

3. Compile the tools and copy them to the `python` folder of the project

```
$ cd tools
$ make
$ cp reg_rw dma_from_device <ProjectFolder>/python
```

4. Load the kernel module

```
$ cd ../tests
$ sudo ./load_driver.sh
```

HLS Algorithm implementation

In order to implement an algorithm in HLS, the Vitis HLS tool should be used. The top level task for the implementation must have the following structure:

```
void top(
        hls::stream<data_t>& in_adc1,    hls::stream<data_t>&
in_adc2,
        hls::stream<data_t>& in_adc3,    hls::stream<data_t>&
in_adc4,
        hls::stream<data_t>& out_dac1,   hls::stream<data_t>&
out_dac2,
        hls::stream<data_t>& out_dac3,   hls::stream<data_t>&
out_dac4,
        hls::stream<pci_data_t>& in_h2c, hls::stream<pci_data_t>&
out_c2h
    ) {
    #pragma HLS INTERFACE mode=axis
port=in_adc1,in_adc2,in_adc3,in_adc4,
    #pragma HLS INTERFACE mode=axis
port=out_dac1,out_dac2,out_dac3,out_dac4
    #pragma HLS INTERFACE mode=axis port=in_h2c,out_c2h
    #pragma HLS INTERFACE mode=ap_ctrl_none port=return

    /* ALGORITHM IMPLEMENTATION */
}
```

Once implemented, the design must be exported as an IP in a ZIP file and imported into the Vivado project. Then, the `hls_dsp` IP must be deleted from the source tree and the new IP must be instantiated **with the same name** .

TMPE DAQ C++ Library Documentation

class tmpe_daq

Class for managing the TMPE DAQ.

Public Functions

tmpe_daq (const std :: string & user_dev , const std :: string & c2h_dev , const std :: string & h2c_dev)

Construct a new tmpe daq object. Acquires the file descriptors for the register interface and the DMA interfaces.

Parameters :

- **user_dev** – Register interface device (e.g. /dev/xdma0_user)
- **c2h_dev** – Card to host DMA interface device (e.g. /dev/xdma0_c2h_0)
- **h2c_dev** – Host to card DMA interface device (e.g. /dev/xdma0_h2c_0)

~tmpe_daq ( )

Destroy the tmpe daq object. Releases the file descriptors for the register interface and the DMA interfaces.

double get_sampling_rate ( ) const

Get the current sampling rate of the DAQ.

Returns :

double Sampling rate in Hz

double set_sampling_rate (double rate)

Set the sampling rate of the DAQ. Not all sampling rates are possible, only whole divisors of the FPGA clock (100 MHz). This function will set the closest possible rate.

Parameters :

rate – Desired sampling rate in Hz

Returns :

double Actual sampling rate in Hz

bool get_adc_pd ( ) const
Get the ADC power down status.

Returns :

true The ADC is powered down

Returns :

false The ADC is powered up

void set_adc_pd (bool pd)
Set the ADC power down status.

Parameters :

pd – true to power down the ADC, false to power it up

ADCRanges get_adc_range (uint8_t channel) const
Get the configured range for one ADC Channel.

Parameters :

channel – Channel number (0-3)

Returns :

ADCRanges Current range of the ADC

void set_adc_range (uint8_t channel , ADCRanges range)
Set the range for one ADC Channel.

Parameters :

- **channel** – Channel number (0-3)
- **range** – Channel range

bool get_dac_pd (uint8_t channel) const
Get the Power Down status of one DAC Channel.

Parameters :

channel – Channel number (0-3)

Returns :

true The channel is powered down

Returns :

false The channel is powered up

void set_dac_pd (uint8_t channel , bool pd)

Set the Power Down status of one DAC Channel.

Parameters :

- **channel** – Channel number (0-3)
- **pd** – true to power down the channel, false to power it up

bool get_dac_code ( ) const

Get the DAC code configuration.

Returns :

true Offset binary coding

Returns :

false 2's Complement coding

void set_dac_code (bool code)

Set the DAC code configuration.

Parameters :

code – false for 2's Complement coding, true for Offset binary coding

bool get_dac_clr_sel ( ) const

Get the DAC Clear Select configuration.

Returns :

true DAC clear set to mid-scale if unipolar or negative full-scale if bipolar

Returns :

false DAC clear set to zero

void set_dac_clr_sel (bool clr_sel)
Set the DAC Clear Select configuration.

Parameters :

clr_sel – true to set DAC clear to mid-scale if unipolar or negative full-scale if bipolar, false to set DAC clear to zero

bool get_dac_tsd_ena ( ) const
Get the DAC Thermal Shutdown Enable configuration.

Returns :

true Thermal shutdown enabled

Returns :

false Thermal shutdown disabled

void set_dac_tsd_ena (bool tsd_ena)
Set the DAC Thermal Shutdown Enable configuration.

Parameters :

tsd_ena – true to enable thermal shutdown, false to disable it

DACRanges get_dac_range ( ) const
Get the current range of the DAC.

Returns :

DACRanges Current range of the DAC

void set_dac_range (DACRanges range)

Set the range of the DAC.

Parameters :

range – Desired range of the DAC

void enable_c2h_dma ( )

Enable the Card to Host DMA.

void disable_c2h_dma ( )

Disable the Card to Host DMA.

int read_c2h_dma (int64_t * buf , size_t bufsize)

Read the Card to Host DMA. The buffer must be large enough to hold the data. The function will read bufsize 64-bit integers. An internal aligned buffer is allocated, read to and then copied, so the buffer does not need to be aligned. The first DAC_IGNORE_SAMPLES (32) samples are ignored, and the rest are copied to buf.

Parameters :

- **buf** – Buffer to hold the read samples
- **bufsize** – Size of the buffer in 64-bit integersj

Returns :

int 0 on success, -1 on error

int write_h2c_dma (int64_t * buf , size_t bufsize)

Write to the Host to Card DMA. The function will write bufsize 64-bit integers. The buffer must be aligned to the page size (see posix_memalign). The function will write the buffer to the DMA.

Parameters :

- **buf** – Buffer to write to the DMA
- **bufsize** – Size of the buffer in 64-bit integers

Returns :

int 0 on success, -1 on error