UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE TELECOMUNICACIÓN

MÁSTER UNIVERSITARIO EN INGENIERÍA DE SISTEMAS ELECTRÓNICOS

MASTER'S THESIS

IMPLEMENTATION OF A PCIe INTERFACE TO TRANSFER DATA AT HIGH SPEED BETWEEN A HOST AND AN ADVANCED FPGA

SIMÓN PORTELA QUEIMAÑO

SEPTEMBER 2022

MÁSTER UNIVERSITARIO EN INGENIERÍA DE SISTEMAS ELECTRÓNICOS

MASTER'S THESIS

Title: IMPLEMENTATION OF A PCIe INTERFACE TO TRANSFER DATA AT HIGH SPEED BETWEEN A HOST AND AN ADVANCED FPGA

Author: SIMÓN PORTELA QUEIMAÑO

Supervisor: PEDRO JOSÉ MALAGÓN MARZO

Co-supervisor: MARIO GARRIDO GÁLVEZ

Department: DEPARTAMENTO DE INGENIERÍA ELECTRÓNICA

EVALUATION COMMITTEE

President: ....

Vocal: ....

Secretary: ....

The evaluation committee members agree to grant the grade: ....

Madrid, on ...., 2022

UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE TELECOMUNICACIÓN

MÁSTER UNIVERSITARIO EN INGENIERÍA DE SISTEMAS ELECTRÓNICOS

MASTER'S THESIS

IMPLEMENTATION OF A PCIe INTERFACE TO TRANSFER DATA AT HIGH SPEED BETWEEN A HOST AND AN ADVANCED FPGA

Author: SIMÓN PORTELA QUEIMAÑO

Supervisor: PEDRO JOSÉ MALAGÓN MARZO

Co-supervisor: MARIO GARRIDO GÁLVEZ

SEPTEMBER 2022

A mi familia y amigos, pero en especial a Paula.

"Put yourself in situations where you are not comfortable", Lebrom James.

Abstract

One of the main bottlenecks for algorithm acceleration in hardware is data transfer between different systems. Field-programmable gait arrays (FPGAs) have several high-speed interfaces that can be used for data communication. Among them is the Peripherial Component Interconnect Express (PCIe), a serial bus that can have up to 16 parallel channels. To efficiently implement communication between a computer (Host) and an acceleration card (FPGA), it is necessary that all the elements in the chain are as fast as possible: design of the PCIe interface in the FPGA, driver support for the Operating System and application access in user space.

In this Master's Thesis, two different projects have been implemented using two different drivers, one developed by Xilinx and the other one by CERN, so that they can be compared objectively taking into account factors such as the transfer rate achieved, the resources used in the FPGA and limitations at the level of parallelism of the structures that can be implemented to take advantage of this interface, such as a fast Fourier transform (FFT).

The obtained results reach 50% of the theoretical values in terms of transfer rate performance. Thus, the objective of implementing an interface between the computer and the FPGA through the PCIe port has been achieved, reaching values close to 80 Gbps.

Key Words

PCIe, AXI, DMA, FPGA, Wupper, XDMA.

Resumen

Uno de los principales cuellos de botella para la aceleración de algoritmos en hardware es la transferencia de datos entre los distintos sistemas. Las *Field − programmable gait arrays* (FPGAs) cuentan con varias interfaces de alta velocidad que se pueden utilizar para la comunicación de datos. Entre ellos destaca el *Peripherial Component Interconnect Express* (PCIe), un bus serie que puede tener hasta 16 canales en paralelo. Para implementar de forma eficiente una comunicación entre un ordenador (*Host*) y una tarjeta de aceleración (FPGA) es necesario que todos los elementos de la cadena funcionen lo más rápido posible: diseño de la interfaz PCIe en la FPGA, driver de soporte para el sistema operativo y aplicación de acceso en espacio de usuario.

En este Trabajo Fin de Máster (TFM) se han implementado dos proyectos distintos empleando dos drivers desarrollados uno por Xilinx y el otro por el CERN, de forma que se puedan comparar objetivamente teniendo en cuenta factores tales como la velocidad de transferencia alcanzada, los recursos empleados en la FPGA y limitaciones a nivel de paralelismo de las estructuras que se pueden implementar para sacarle partido a esta interfaz, como por ejemplo una *Fast Fourier Transform* (FFT).

Los resultados obtenidos alcanzan el 50% de los valores teóricos en cuanto a prestaciones de la velocidad de transferencia. Así se ha cumplido el objetivo de implementar una interfaz entre el ordenador y la FPGA a través del puerto PCIe, alcanzando valores cercanos a los 80 Gbps.

Palabras clave

PCIe, AXI, DMA, FPGA, Wupper, XDMA.

Acknowledgments

Thanks to Pedro and Mario, the two supervisors of this Master's Thesis, for having helped me during these months to carry out the project, for their patience and trust.

Contents

1 Introduction **1**

2 Background **3**

2.1 PCI Express 3

2.1.1 Communication between PCIe devices 4

2.1.2 PCI Utils 13

2.2 Drivers 16

2.2.1 XDMA by Xilinx 16

2.2.2 Wupper by CERN 17

3 Solution based on XDMA **19**

3.1 Proposed approach 19

3.1.1 Post-implementation results 23

3.2 Data transmission between the PC and the FPGA 24

3.2.1 Testing tools 24

3.3 Measurements 25

3.4 Technical challenges 28

4 Solution based on Wupper **31**

4.1 Proposed approach 31

4.1.1 Post-implementation results 33
4.2 Data transmission between the PC and the FPGA 34
4.2.1 Testing tools 34
4.3 Measurements 35
4.4 Technical challenges 38
4.5 Comparisons 39

5 Conclusions and future work **41**
5.1 Conclusions 41
5.2 Future work 41

A Budget **43**

B Ethical, social, economic and environmental aspects **45**
B.1 Ethical and social impacts 45
B.2 Economic impact 45
B.3 Environmental impact 46

C User guide **47**
C.1 Adaptation of the top model 47
C.2 Verify implementation 47
C.3 Data transmission 48
C.3.1 XDMA-based data transmission 48
C.3.2 Wupper-based data transmission 48
C.4 Recommendations 49

List of Figures

2.1 Interconnection between two PCIe devices with 16 lines. 4

2.2 Different wire inside of a PCIe line. 5

2.3 Transaction Layer Packet. 6

2.4 Full connection between two devices. 6

2.5 Efficiency of the transaction vs size of the data payload. 7

2.6 Example of a write operation. 8

2.7 Example of a read request from the CPU. 10

2.8 Example of a compliter's response. 11

2.9 Example of a connection between PCIe devices. 13

2.10 Structure of the Wupper. 18

3.1 Final RTL design based on XDMA. 20

3.2 Waveform of the transmission of the *valid_in* signal. 22

3.3 Valid-ready verifier module. 23

3.4 Input signal with $k = 2$ and amplitude equal to 0.1 25

3.5 Amplitude adapted to 16 bit 26

3.6 FFT of the signal with $k = 2$ and $N = 16$ 26

3.7 Measurement of average transfer speed 27

3.8 Transaction speed vs data payload size 28

4.1 Final model with the Wupper's driver 32
4.2 Waveform of the transmission of the *valid_in* signal 33
4.3 Input signal with $k = 2$ and amplitude equal to 0.1 35
4.4 Amplitude adapted to 16 bit 36
4.5 FFT of the signal with $k = 2$ and $N = 32$ 36
4.6 Transaction speed vs data payload size 37

C.1 Running data transmission with XDMA 48
C.2 Running data transmission with XDMA 48

List of Tables

2.1 PCI Express link performance 4

2.2 Efficiency depending on the size of the data payload 7

2.3 PCI header 14

2.4 PCI BARs 15

3.1 XDMA post-implementation results 23

3.2 Values of transfer rate as a function of data payload 29

4.1 Wupper post-implementation results 34

4.2 Values of transfer rate as a function of data payload 38

4.3 Transfer rate of each solution 39

A.1 Budget 43

Chapter 1

Introduction

In the world of research on advanced digital architectures such as FFTs (Fast Fourier Transform) or neural networks, in FPGAs (Field Programmable Gate Arrays), it is essential to have a test bench where designs can be tested, through which data can be sent and received between a computer (host) and an FPGA (card). Hence the need to carry out this project in order to have a test bench through which to verify the operation of these architectures as well as to measure transfer speeds, among others.

The FPGA VCU128 UltraScale+ [1] is used in this Master's Thesis. Although this card has different interfaces through which to send and receive data, such as Ethernet or UART, this project focuses on the development of the interface through the PCIe (Peripherial Component Interconnect Expres) port [2, 3, 4].

Therefore, the objective of this Master's Thesis is the design of an interface to send and receive data between the FPGA and the computer through the PCIe, thus having a test bench where advanced and high throghput digital architectures can be tested.

To this end, a search for possible existing solutions that could be adapted for this project was carried out [5, 6, 7, 8, 9], and two possible drivers were found. One is called XDMA [4] and has been developed by Xilinx, which allows for the implementation of a PCIe Gen3 interface with 16 channels. The other driver, called Wupper [10, 11, 12] and developed by CERN, allows a PCIe Gen4 interface with 16 channels. This, as explained in the following chapters, has advantages in terms of transfer speed.

In addition, communications between the different modules of the system, such as the PCIe IP and the architecture under test, is done via the AXI bus [13, 14], and going deeper, the digital architecture used to verify the operation of the system has been two FFTs, one 32-point 32-parallel and the other one 16-point 16-parallel [15].

This Master Thesis is organized as follows. In Chapter 2, the theoretical concepts necessary to understand the proposed approach are explained, as well as all the tools or resources that have been used to carry out the project, such as the XDMA and Wupper drivers. Following this, chapters 3 and 4 explain the proposed approach and the results obtained for the two solutions using XDMA and Wupper as a starting point, respectively. After presenting the results obtained with the two proposals, they are compared at the end of Chapter 4. And finally, Chapter 5 presents the future work that can be done to continue this project together with the conclusion. In addition, an user guide has been included as Annex C for those who use the work developed in this project to implement their own models.

Chapter 2

Background

2.1 PCI Express

PCI Express means ”Peripherial Component Interconnect Express” (PCIe) [16], and is a high-speed serial and differential computer expansion bus standard [17, 18]. It is a point-to-point communications bus and improves on the previous parallel and shared PCI bus. Therefore, to allow multiple cards to be connected, motherboards include switches that form a network of PCIe devices.

In addition, PCIe supports full duplex communications and has a certain number of PCI lines, which can be x1, x2, x4, x8 or x16. Full duplex communication means that information can be sent and received at the same time, since each PCI line or channel has a Tx port (for transmitting) and an Rx port (for receiving).

In terms of PCIe applications, it is intended to be used as a local bus, having a PCIe controller that communicates with the different PCIe Endpoints. It is mostly used to connect graphics cards and as a port for connecting high-performance solid-state drives (SSDs). As far as this project is concerned, many FPGAs have a PCIe connector for high-speed communication between the PC and the FPGA.

Additionally, depending on the generation and the number of PCIe lines, the throughput is not the same. Therefore, Table 2.1 shows the performance variation in relation to the number of lines and the PCIe generation.

This means that the maximum throughput that is possible to achieve with the VCU128 UltraScale+ FPGA, is 31.508 GB/s (Gen4 x16 channels) as it will be explained in the following chapters.

This chapter explains the characteristics of PCIe necessary to understand the project carried out in this Master's Thesis, leaving aside concepts such as interruptions in the communication between PCI Express devices, as this is surplus

Table 2.1: PCI Express link performance

Gen.	Transf. rate per line (GT/s)	Throughput (GB/s) x1	x2	x4	x8	x16
1.0	2.5	0.250	0.500	1.000	2.000	4.000
2.0	5.0	0.500	1.000	2.000	4.000	8.000
3.0	8.0	0.985	1.969	3.938	7.877	15.754
4.0	16.0	1.969	3.938	7.877	15.754	31.508

information and the objective is not to reach such a low level.

2.1.1 Communication between PCIe devices

PCIe does not behave as a standard bus, if not as a network, with each card connected to a network switch via a dedicated set of wires. Each line of a PCIe port consists of four wires of data transmission, two of differential pairs in each direction, transmitting (Tx^+, Tx^-) and receiving (Rx^+, Rx^-); accompanied by two other wires representing clock reference as a differential pair (Clk^+ and Clk^-) [2]. This is shown in Figure 2.1, where the communication between two PCIe devices is reoresented, and in Figure 2.2, where it is possible to appreciate the principal wires within each line of the PCIe interconnection.

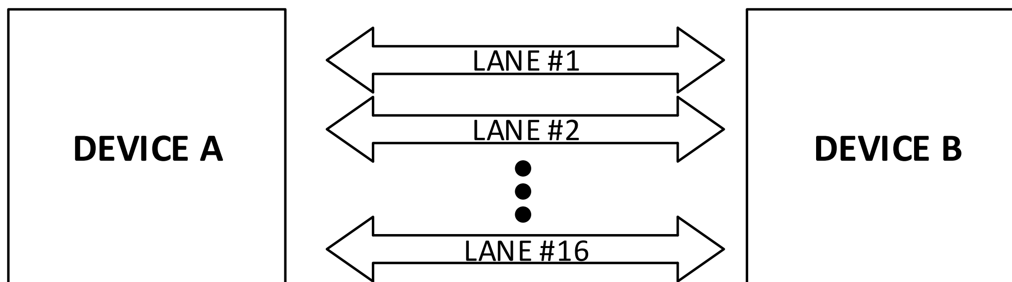

Figure 2.1: Interconnection between two PCIe devices with 16 lines.

The simplest transaction

In order to understand how two PCIe devices talk, the easiest way to get started is with a small data transaction as an example where device A (the CPU of the PC) wants to write a 32-bit word into device B (the FPGA). The CPU generates a memory write packet that consists of 3 or 4 32-bit words along with the 32-bit word of the data. These 3 or 4 words that are not the data, which means something like

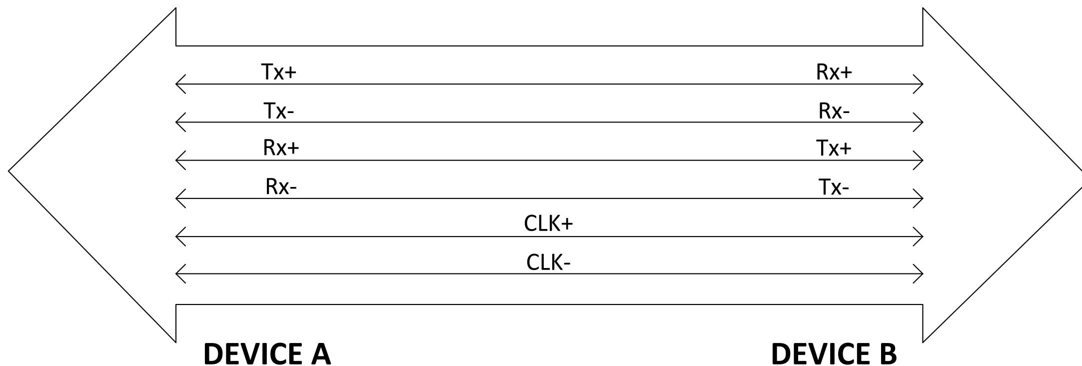

Figure 2.2: Different wire inside of a PCIe line.

"write this date into this address" and are considered overflow, a term that will be explained later.

Underlying communications mechanism

The underlying communications mechanism consists of three layers:

- Transaction layer: The description of the packet explained in the previous paragraph defines a transaction tayer packet (TLP). This packet consists of the header made up of 3 or 4 data words (DW) depending on the version of the PCIe, followed by the data payload, wich can go from 1 to 1024 DW, and at the end it could be an cyclic redundancy check (CRC) (1 DW).
- Data link layer: The data link layer is responsible for ensuring that each TLP reaches its intended recipient correctly. It wraps the TLP with its own header called *sequence* and a link CRC or LCRC, so that the integrity of the TLP is ensured. A flow control mechanism ensures that each packet is sent only when the link partner is ready to receive it. Finally, the transmitter of a memory write TLP does not receive an indicator that the packet arrived successfully, as this is not necessary and saves sending redundant information.
- Phyisical layer:In generations 1 and 2 of the PCIe, the physical layer adds one byte called *start* at the beginning of the data stream and another byte called *end* at the end of it. However, in later generations like 3 and 4, the physical layer only adds 4 bytes at the beginning as a *start* stream.

In Figure 2.3 it is possible to see the entire transaction layer packet or TLP with the different DWs of each layer, while in Figure 2.4 is shown the full connection between two devices.

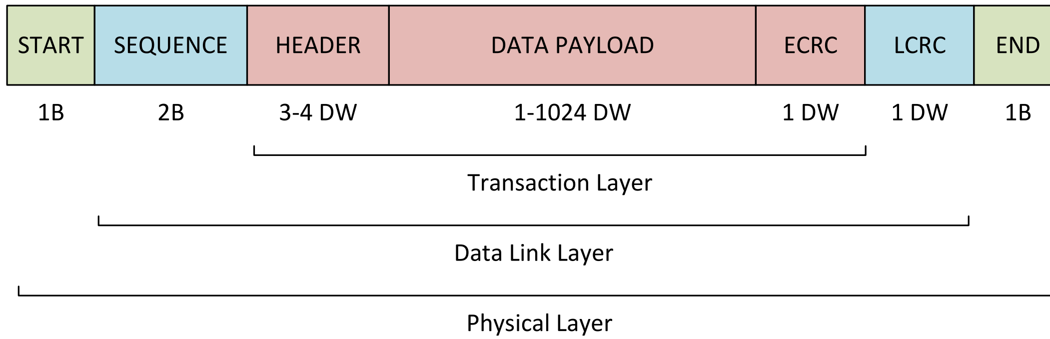

Figure 2.3: Transaction Layer Packet.

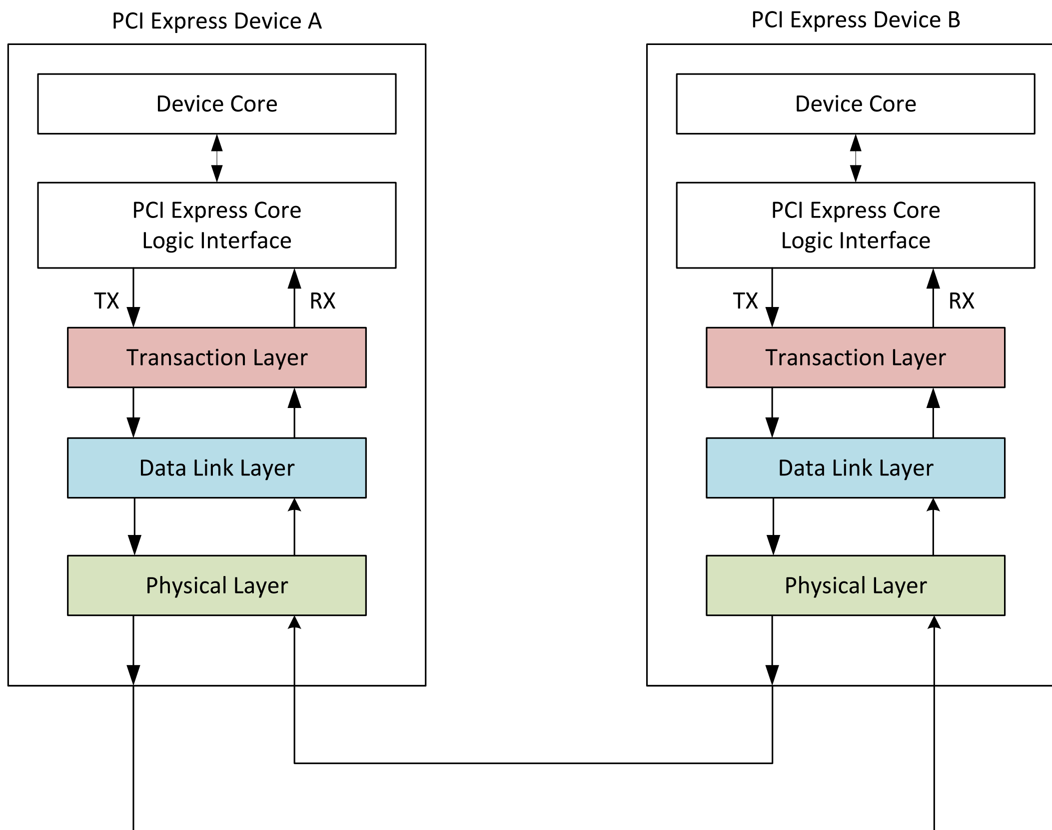

Figure 2.4: Full connection between two devices.

After knowing the different parts that make up a complete data frame, it can be observed that, within it, there are several DW that do not provide information as they do not belong to the data payload. All these bytes are called overflow

and it is important to take them into account to determine the efficiency of a data transmission. Depending on the number of Bytes that make up the data payload, the percentage of bytes that contain information within each TLP varies, since the larger the data payload, the higher the efficiency.

Knowing that the minimum data payload is 1 DW (4 bytes) and the maximum is 1024 DW (4096 bytes), Table 2.2 shows the value of the transmission efficiency in % depending on the size of the data payload.

Table 2.2: Efficiency depending on the size of the data payload

Data payload (DW)	Data payload (Bytes)	Efficiency (%)
1024 DW	4096	99.27%
512 DW	2048	98.56%
64 DW	128	81.01%
8 DW	32	51.61%
1 DW	4	11.76%

Figure 2.5 shows a graph with the variation of the efficiency in percent per one, as a function of the size of the Data Payload in Bytes.

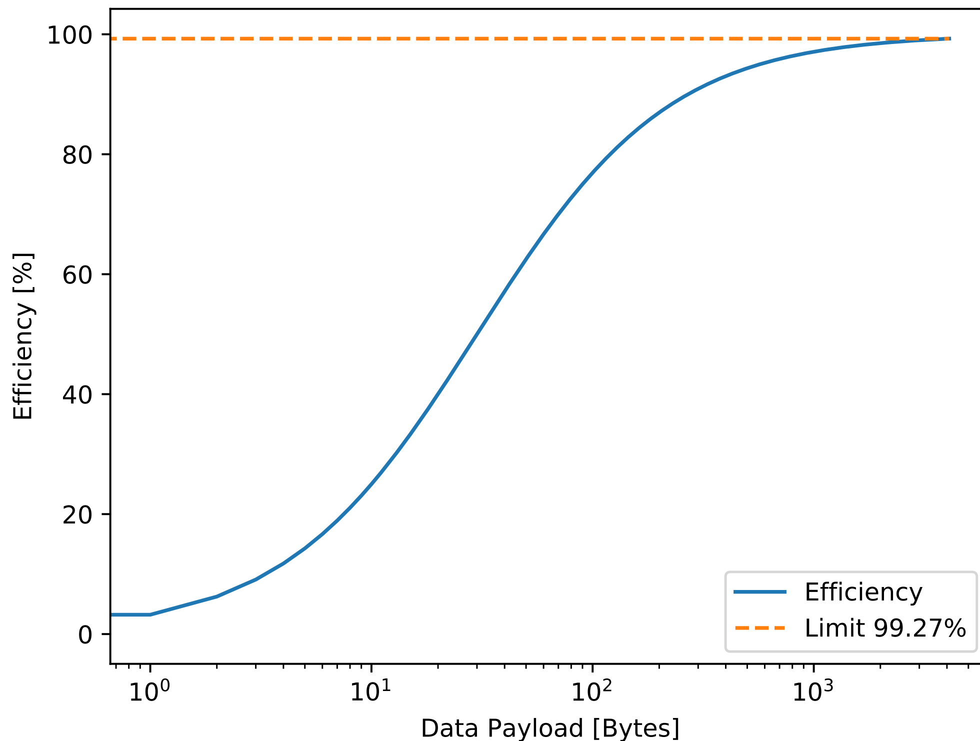

Figure 2.5: Efficiency of the transaction vs size of the data payload.

Writing operation

In order to explain better and more clearly the writing operations, the following example is provided. Assume that the CPU used 32-bit addressing to write the value 0xf0e1f2c3 to the physical address 0xfdaff040. In that case, the packet might be made up of four 32-bit words (four DWs, or double words), as shown in Figure 2.6.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00

DW 0	R 0	Fmt 0x2	Type 0x00	R 0	TC 0	R 0	TD 0	EP 0	Attr 0	R 0	Lenght 0x001
DW 1	Requester ID 0x0000						Tag (unused) 0x00			Last BE 0x0	1st BE 0xF
DW 2	Address [31:2] 0x3f6bfc10										R 0
DW 3	Data DW 0 0xF0E1D2C3										

Figure 2.6: Example of a write operation.

As a result, the packet is sent as 0x40000001, 0x0000000f, 0xfdaff040, and 0xf0e1f2c3. A brief explanation of the colour coding and the valid fields in Figure 2.6 is provided:

- Since reserved fields are gray, the sender must enter zeroes there (and the receiver ignores them). Some gray fields include the symbol "R" to indicate that they are always reserved, while others have names to indicate that they are reserved due to the characteristics of this particular packet.
- Although green fields can contain nonzero values, endpoint peripherals do not frequently use them.
- The values for the particular packet are highlighted in red.
- The Fmt field and the Type field indicate that this is a memory write request.
- There is no additional CRC on the TLP data because the TD bit is set to zero. If the hardware is expected not to corrupt the TLPs, the extra CRC is not necessary, as the link layer has its own CRC to make sure that the transmission is done correctly.
- This TLP contains one DW (32-bit word) of data, as shown by the value of the Length field, which is 0x001.
- According to the Requester ID field, the root complex is the packet's sender and may be identified by its ID of zero (the PCIe port closest to the CPU). Even though it is required, this field only serves in a write request to indicate failures.

- In this instance, the Tag field is empty. All other components should ignore anything the sender enters in this field.
- Selecting which of the first data DW's four bytes are acceptable and should be written is possible by using the 1st BE field (1st double-word byte enable). In this situation, setting it to 0xf means that all four bytes have been written.
- Since the 1st DW and the last DW are the same when Length equals unity, the Last BE field must be 0.
- The address in which the first data DW is written is all that is contained in the Address field. The bits 31-2 of this address, to be more precise. It should be noted that DW 2 really reads the write address itself because the two LSBs of DW 2 in the TLP are zero. The result of multiplying 0x3f6bfc10 by four is 0xfdaff040.
- Finally, the last DW is the Data DW. Notice that PCIe operates big endian and that would be a problem with its software representation since Intel processors use little endian. The software representation of 0xf0e1d2c3 would be 0xc3d2e1f0.

Reading operation

After explaining the writing operations, below, it is going to be examined what occurs when the CPU attempts to read from a peripheral. Inevitably, read operations involve two packets: one TLP from the CPU to the peripheral, instructing it to do a read operation, and one TLP returning with the data. This makes read operations a little more challenging. In this case, there is a requester (the CPU in this example) and a completer in PCIe words (the peripheral that is going to be an FPGA in this project).

For this example, it is going to be asumed that the CPU requires a single DW (32-bit word) from the same address as before 0xfdaff040. As previously, it is probable that it starts a read operation on the bus that it shares with its memory controller, which is home to the Root Complex, and that this causes the Root Complex to produce a TLP that is then sent over the PCIe bus. An example of read request is shown in Figure 2.7.

The three DWs 0x00000001, 0x00000c0f, and 0xfdaff040 make up this packet. The peripheral is instructed to read one complete DW at location 0xfdaff040 and return the information to the bus entity with ID 0x0000.

This is nearly identical to the write request illustration above. The differences are explained below:

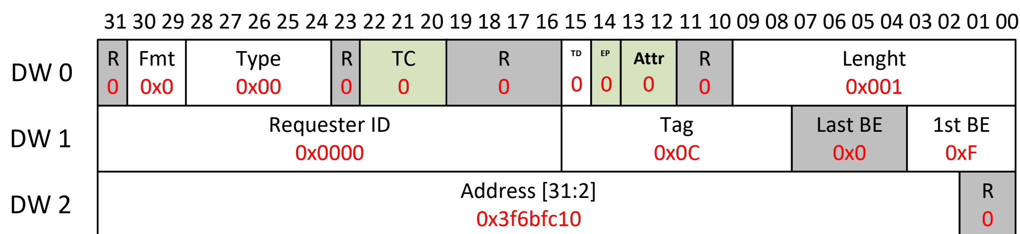

Figure 2.7: Example of a read request from the CPU.

- To signal that this is a read request, the Fmt/Type attributes have changed (really, only Fmt).
- The Requester ID field once again indicates that this packet's sender has ID zero. It is the same field as previously, but in a read request it is significant since it indicates to which address the completer must send its reply.
- In read requests, the Tag is important. It is critical to understand that while it serves as a tracking number, it itself means nothing: This value must be copied to the completion TLP by the completer when it responds. As a result, the requester can compare the completion responses to its request. After all, a bus allows many requests from a single device. As long as the Tags of all pending requests are distinct, the standard does not specify an enumeration method for this Tag, which is set by the requester for its own purposes. Even though there are 8 bits available, only the 5 LSBs can be used; the other bits must always be zero. This permits a pair of bus entities to have up to 32 open requests at once. Depending on the application, in-standard extensions might support up to 2048 requests.
- One DW shall be read, according to the Length field, and the Address field specifies which address. The two BE fields choose which bytes to read rather than write, but they keep the same meaning and constraints as with a write request.

Completion operation

Even if it is unable to perform the requested action, the peripheral is required to provide a Completion TLP in response to a Read Request TLP. An example where everything went well it is going to be examined: The peripheral read the relevant data from its internal storage and now has to deliver the outcome to the requester (the CPU in this case).

Figure 2.8 shows an example of a completion TLP. The TLP is made up of 0x4a000001, 0x01000004, 0x00000c00, and 0xf0e1f2c3. The essence of the packet is

to tell the bus entity 0x0000 that the response to its request to entity 0x0100, which was tagged 0x0c, is 0xf0e1f2c3. Now that it knows what that request was about, the CPU (or, more precisely, the memory controller, or root complex), can complete the necessary bus cycle. As with the write and read operations, each part of the TLP in the figure above is explained below:

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00

DW 0	R 0	Fmt 0x2	Type 0x0a	R 0	TC 0	R 0	TD 0	EP 0	Attr 0	R 0	Lenght 0x001
DW 1	Completer ID 0x0100						Status 0x00			BCM 0	Byte Count 0x004
DW 2	Requester ID 0x0000						Tag 0x0C			R 0	Lower Address 0x40
DW 3	Data DW 0 0xF0E1D2C3										

Figure 2.8: Example of a compliter's response.

- This is a completion packet with data, according to the Fmt field and Type field.

- This TLP contains one DW (32-bit word) of data, as shown by the value of the Length field, which is 0x001. However, the requester ought to be aware of that, so this seems to be redundant. The explanation is that a TLP has a length limit that may be smaller than the required number of DWs. Several Completion TLPs are then returned as a result of that. Therefore, the Length field indicates the number of DWs in this particular packet.

- In addition, there is the Byte Count field. The number of valid payload bytes present in the packet in this example of a single-TLP completion is all that matters. There are four valid bytes, because the first DW BE field in the request contained all ones, as specified in the field. The actual definition of this field, is the total number of bytes still available for transmission, including those in the current packet.

- The Lower Address field is the next. The first byte of this TLP was read from the address's seven least significant bits. In this case, it is 0x40, derived from 0xfdaff040's lower bits. This field is relevant when there are several TLP completions.

- The completer ID, which is 0x0100, identifies the sender of this packet.

- The Receiver ID, which is zero ID, identifies the receiver of this packet (Root Complex). This serves as the destination address if there are any PCIe switches to route through.

- Because there is a zero in the Status field, the Completion was successful. Other values represent various kinds of rejections.

- Except when a packet comes from a bridge with PCI-X, the BCM field is always zero. So it is 0 now.

- Finally, the last DW is the Data DW.

Posted and non-posted transactions

Requests that receive a Completion TLP from the Completer, which shows that the transaction was successfully handled, are considered non-posted transactions. The device that receives the request as its final destination, rather than an intermediary device along the packet's journey like a switch, is known as the Completer.

The Completer produces a Completion with Data TLP in response to read requests. The Completer provides a Completion without Data TLP with a message indicating that the transaction was unsuccessful if it is unable to meet the request. The Completer delivers a Completion without Data TLP for non-posted write requests that shows if the write was successful.

Messages and memory writes are posted transactions. The ultimate destination of the packet is not required to provide a Completion TLP for these operations. Instead, because the ACK/NAK DLLP protocol ensures that the packet will be successfully transmitted through the system, the transmitting device believes that the submitted request was successful. The requester is not notified via a Completion TLP in the event that the Completer runs into an issue while completing a posted request. To tell the root complex that anything is incorrect, the Completer typically generates an error message.

Bandwidth calculation

It is possible to compute or at least obtain credible estimations of a system's performance when taking into account the factors already covered. In general, bandwidth is calculated by dividing the total quantity of data transferred by the data transfer time. Overall bandwidth takes into account both writes and reads and represents the value that will last over time rather than the peak performance.

$$\text{Bandwidth (bytes/s)} = \frac{\text{Total transfer data size (bytes)}}{\text{Transfer time (s)}} \tag{2.1}$$

2.1.2 PCI Utils

In order to study the different configurations within a PCIe device such as its Header or Base Address Registers (BARs), [19] provides a series of very useful functions to represent these different registers, which are explained next [20, 21].

PCI Express device types

The first thing to be clear about is that there are different types of PCI Express devices depending on the number of connections to other PCIe devices and the position of these connections in the data flow hierarchy. For example, if they only have one connection through a downstream port (DS), then that device is an endpoint. In addition, if the connection is made between several devices, this is done via buses, which have their own ID.

It is important to know what the device ID means, as each device ID is of the form XX.YY.W, where XX represents the bus ID, YY the device number based on the order on the bus itself and W represents the Function Number.

Figure 2.9 represents a common connection between several PCIe devices where the three different types are included: Root Complex (the CPU), bridges (connection between buses) and endpoints.

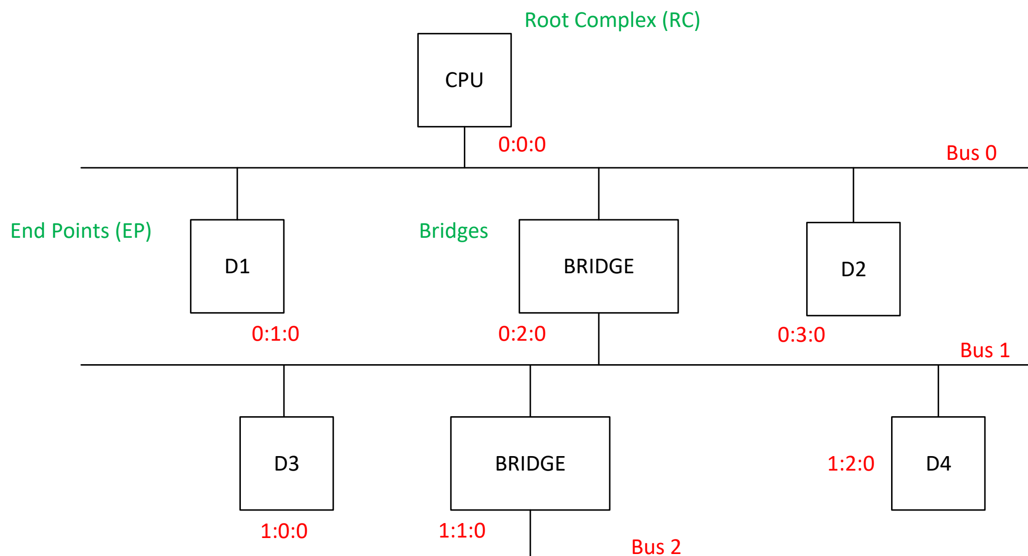

Figure 2.9: Example of a connection between PCIe devices.

PCI header

Most of the information about the device can be found in the header and address spaces. There are two types of addres spaces, one is the I/O or memory space, which is a type of memory over which is it possible to control the device. For example, if there is a PCI card with serial port on it over the I/O space it is possible to access the register to control it like setting support rate or sending/writing the bits/bytes required, out of the serial port.

In addition, the configuration space gives a lot of information about the kind of the device and it enables to set some settings like mapping the PCI devices memory space to the CPU's address space.

Finally, Table 2.3 shows all the information contained in the PCI header. The most important values are:

Table 2.3: PCI header

<table>
<tr><td colspan="2">31   16</td><td colspan="2">15   0</td><td></td></tr>
<tr><td colspan="2">Device ID</td><td colspan="2">Vendor ID</td><td>00h</td></tr>
<tr><td colspan="2">Status</td><td colspan="2">Command</td><td>04h</td></tr>
<tr><td colspan="3">Class Code</td><td>Revision ID</td><td>08h</td></tr>
<tr><td>BIST</td><td>Header Type</td><td>Lat. Timer</td><td>Cache Line S.</td><td>0Ch</td></tr>
<tr><td colspan="4" rowspan="6">Base Address Registers</td><td>10h</td></tr>
<tr><td>14h</td></tr>
<tr><td>18h</td></tr>
<tr><td>1Ch</td></tr>
<tr><td>20h</td></tr>
<tr><td>24h</td></tr>
<tr><td colspan="4">Cardbus CIS Pointer</td><td>28h</td></tr>
<tr><td colspan="2">Subsystem ID</td><td colspan="2">Subsystem Vendor ID</td><td>2Ch</td></tr>
<tr><td colspan="4">Expansion ROM Base Address</td><td>30h</td></tr>
<tr><td colspan="3">Reserved</td><td>Cap. Pointer</td><td>34h</td></tr>
<tr><td colspan="4">Reserved</td><td>38h</td></tr>
<tr><td>Max Lat.</td><td>Min Gnt.</td><td>Interrupt Pin</td><td>Interrupt Line</td><td>3Ch</td></tr>
</table>

- Device and vendor ID: Unique ID for each vendor. It will tell what device it is.
- Class code: It indicates the type of device.
- Header type: It indicates whether it is an endpoint or a bridge. EP = 0x00, Bridge = 0x01.
- Subsystem and subsystem vendor ID: More info about the device.

Base Address Registers (BARs)

The base address registers (BARs) are the link between the configuration space and the memory or I/O space. They have, essentially, three purposes:

- Tell the CPU which kind of address space is available on a BAR (memory or I/O).
- Indicate to the CPU how much space is available on a BAR.
- The CPU can map the PCI device's memory or I/O space into its own address space by assigning a memory or I/O address to the base address.

Table 2.4, shows the description of the bits of each BAR.

Table 2.4: PCI BARs

Bits	Description	Values
For all PCI BARs		
0	Region Type	0 = Memory 1 = I/O
For Memory BARs		
2-1	Locatable	0 = any 32-bit 1 = <1 MB 2 = any 64-bit
3	Prefetchable	0 = No 1 = Yes
31-4	Base Address	naturally 16-Byte aligned
For I/O BARs		
1	Reserved	
31-2	Base Address	naturally 4-Byte aligned

It is important to take the following considerations into account: The lower bits are hardwired and the upper bits (base address) are readable and writable. Also, whether a BAR is prefetchable or not is important to CPU's caches, because if it is not, it can not use cache memory for this data.

It is important to make sure that the bottom bits are masked while attempting to acquire a BAR's true base address. For 16-bit memory space BARs it is calculated (BAR[x] AND 0xFFFFFFF0); for 32-bit Memory Space BARs, (BAR[x] AND 0xFFFFFFF0); for 64-bit Memory Space BARs, ((BAR[x] AND 0xFFFFFFF0) + ((BAR[x + 1] AND 0xFFFFFFFF) $\ll$ 32)); for I/O Space BARs, (BAR[x] AND 0xFFFFFFFC) [22].

Finally, the amount of memory can be determined by masking the information bits, by writing all 1's to the register upper bits (the ones that are not hardwired),

then cleaning the lower bits and finally inverting the BAR and adding '1'. The result is the size in bytes.

This is easier to understand with an example. Consider a 32-bit BAR where bits 11 downto 0 are hardwired, so it is impossible to write there. The rest of the bits are all set to '1', so the address is 0xFFFFF000. Now the BAR is inverted (0x00000FFF) and '1' is added, so the result is 0x00001000 which represents 4k bytes.

2.2 Drivers

When implementing the PCIe interface between the PC and the FPGA, a Linux driver is needed to handle the low-level communications. For this Master's Thesis, two different solutions have been found and will be taken as a starting point. One of them is provided by Xilinx [4] and [23], but only includes the driver and some indication of how to use it and which IP to use. The other solution has been developed by CERN and is called Wupper [11], which is somewhat more complete, but it is still necessary to implement functions that perform the data transfer between PC and FPGA, as well as to adapt the project to the objectives of this Master's Thesis.

2.2.1 XDMA by Xilinx

The Xilinx driver, which will be called XDMA from now on, works with the Xilinx IP "DMA/Bridge Subsystem for PCI Express" [4], which works with AXI4-Stream [14]. The Xilinx IP is basically an intermediary between the PCIe and the model implemented in the FPGA. It is connected directly to the PCIe signals and communicates with the rest of the FPGA via a slave-master interface with AXI4 Stream. This will be better explained in later chapters when discussing implementation.

In addition, the XDMA can work with either Gen. 3, 16 channels or Gen. 4, 8 channels. This allows a theoretical maximum speed of 15.75 GB/s or 126 Gbps, without taking overflow into account.

In order to verify the performance of the driver, a project has been created, which will be explained later, where a 16-point fully parallel FFT [15] has been implemented with a 512-bit parallel input which matches the 16-channel PCIe Generation 3 interface.

2.2.2 Wupper by CERN

The driver developed by CERN, called Wupper, uses the Xilinx IP "UltraScale+ Devices Integrated Block for PCI Express" [3], which also works with AXI4 Stream, but with two interfaces, one for the completer and one for the requester.

The Wupper is more complex than XDMA because it implements PCIe using bifurcation [24], which is basically instantiating two independent PCIe entities running Gen4 with 8 channels, so that if you manage to coordinate the two entities, in the end what you get is a device performing PCIe Gen 4 with 16 channels. This means that it is possible to work at 31.51 GB/s or 252.08 Gbps (see Table 2.1) which theoretically doubles what can be achieved with XDMA.

As with XDMA, a fully parallel FFT has been implemented to verify the operation of the Wupper, in this case with the same FFT but with 32 points instead of 16 [15]. This is because each of the devices in the bifurcation has a 512-bit interface, which means that in the end there is an input/output interface of 1024 bits in parallel, which if is divided by 16 bits, it is equivalent to 32 inputs, which are those of the FFT.

To understand how the Wupper works from the inside, Figure 2.10 shows the basic structure provided by CERN. The model will be explained in Chapter 4 together with the proposed approach.

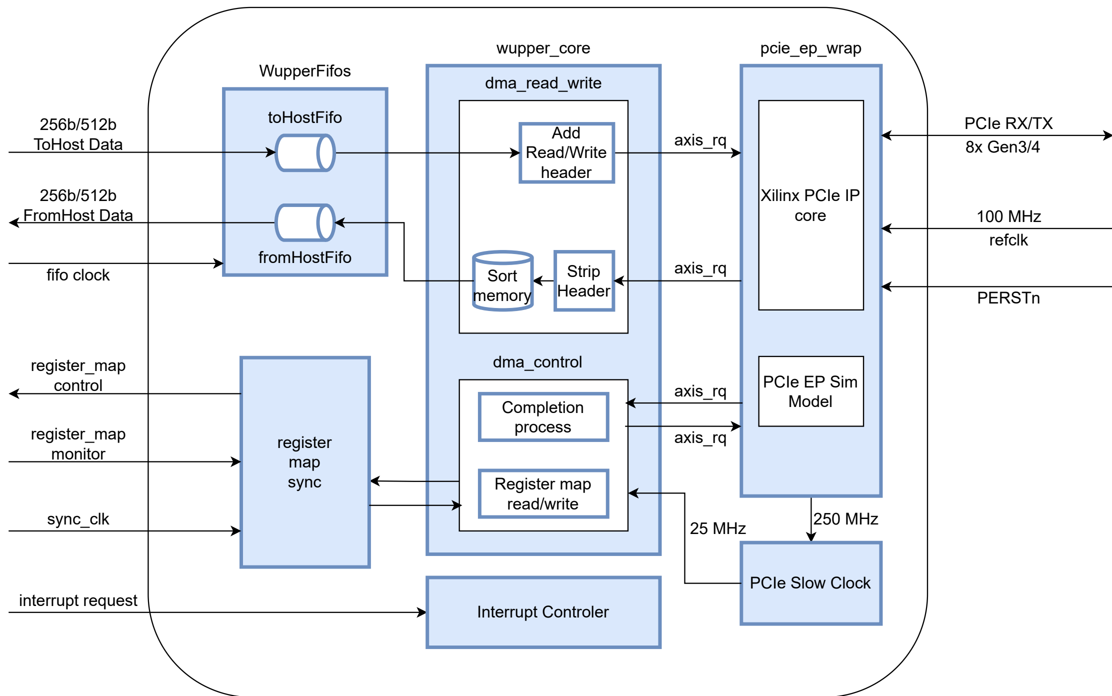

Figure 2.10: Structure of the Wupper.

Chapter 3

Solution based on XDMA

This chapter explains the final project starting from the XDMA driver made by Xilinx, where the final model in Vivado, the implementation results, functions developed to perform the data transfer, among others, will be discussed.

The objective of the implementation using the XDMA, is to be easy to use for the end user, since theoretically as explained in the Background, using this driver which requires PCIe Gen3 with 16 lines, is limiting the theoretical maximum speed to half compared to PCIe Gen4 and 16 lines, which is used in the model based on the Wupper explained in the next chapter.

In addition, the model implemented in the FPGA has been made from scratch, including the dual-clock FIFOs and AXI-Stream interface which are responsible for bridging between the XDMA and the model to be tested, in this case the 16-point 16-parallel FFT.

3.1 Proposed approach

Since it is based on a Xilinx IP, the project has been done as a block design, as it is much more visual and easier to implement.

The proposed approach is shown in Figure 3.1, taking into account that the four main blocks of the system are shown to facilitate the understanding of the design. Blocks such as clock wizards or clock buffers have been omitted, because they do not provide additional information to explain the model.

First is the "XDMA" block, which is the Xilinx "DMA/Bridge" IP [4]. As it can be seen, it has two channels with AXI4-Stream interface, one is "$S_AXIS_H2C_0$", which is the slave, and "$M_AXIS_C2H_0$", which works as the master. The acronym H2C stands for host to card, and C2H stands for card to host, as it refers

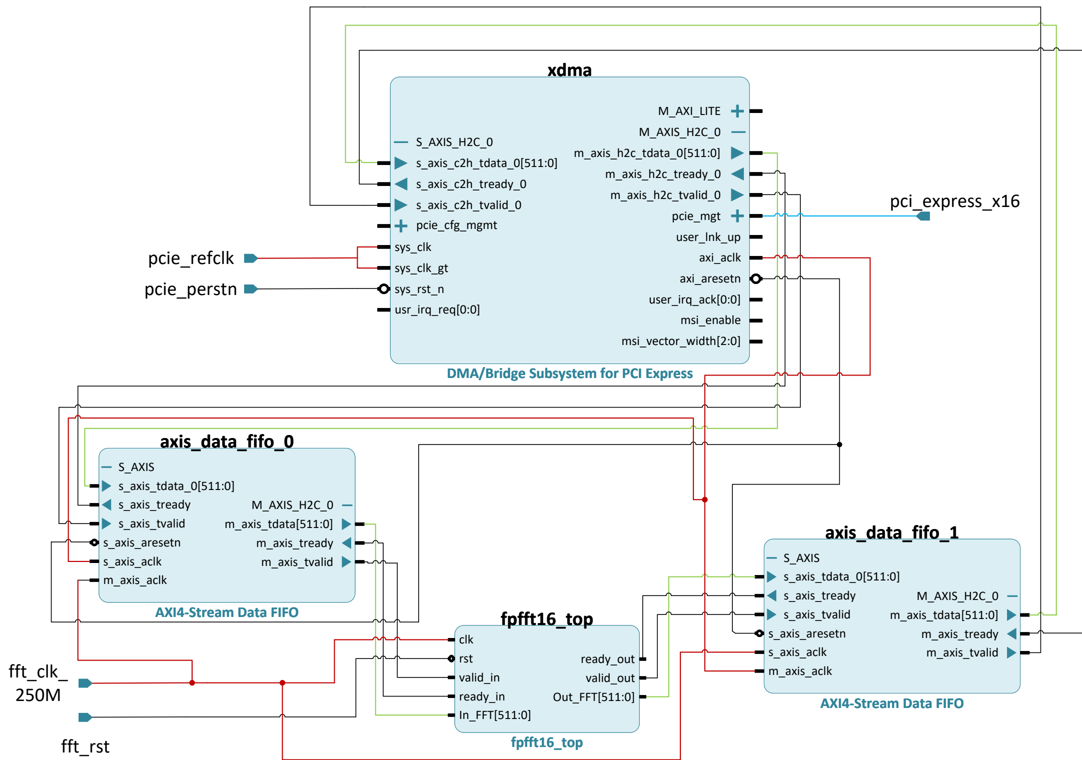

Figure 3.1: Final RTL design based on XDMA.

to the direction of the data flow with the PC as the host, and the FPGA as the card. The master interface on the XDMA is associated with the data input which, through one of the dual clock FIFOs [25], is connected to the input of the FFT module. While the output of the FFT module is connected to the Slave interface of the XDMA through the other FIFO. Also connected to this module are the 16 channels of the PCIe, the reset signal associated with the PCIe and the PCIe clock via a buffer.

The second block is the module to be tested that consists of a 16-point FFT[15], which has a total of 512 inputs and outputs in parallel. 256 are imaginary and the other 256 are real. In addition, it has a latency of 13 clock cycles, which will influence the results presented below, as well as the adaptation of the FFT itself in the system.

In addition, a couple of things need to be said about the FFT module, because a VHDL top file has been made to adapt the initial FFT model to the interface needed to communicate with the rest of the system blocks. The first thing is that both at the input and at the output, the 256 LSBs are associated with the real part and the other 256 bits with the imaginary part. These signals are divided into 16 real and 16 imaginary channels that enter the submodule that performs the Fully Parallel FFT.

Secondly, the CLK signal is limited by the clock of the XDMA and PCIe, as it cannot operate at more than 250 MHz. This is due to the fact that the bottleneck of the system is the PCIe which limits the theoretical throughput to just under 128 Gbps (Gen3 x16) and it makes no sense to operate at more than 250 MHz in the FFT, since as seen in the following equation, this is the limiting frequency at which a throughput of 128 Gbps is achieved in the FFT.

$$\text{Frequency (MHz)} = \frac{\text{Throughput (Gbps)}}{\text{Bits in parallel}} = \frac{128 \text{ Gbps}}{512 \text{ bits}} = 250 \text{ MHz} \tag{3.1}$$

Finally, it is necessary to synchronise this block with the XDMA and the FIFOs. For this purpose, there are the valid and ready signals both at the input and at the output. The input signal *ready_in* is always set to '1' as the FFT processes data faster than the XDMA so it is always ready to receive new data. As for the output *valid_out*, it is intended to be set to '1' at the moment when the data is fully processed and arrives at the FFT output. To do this, it is necessary to know the latency of the FFT and what has been done is to transmit the signal from *valid_in* to *valid_out* and that it takes as many cycles as the latency of the FFT indicates, which is 13 clock cycles.

The code that propagates the *valid_in* signal to *valid_out* depending on the value of the LAT constant (the latency of the FFT) is shown below.

```
constant LAT : integer := 13; -- LATENCY

process(clk,rst)
begin
    if rst = '0' then
        valid_reg <= (others => '0');
    elsif (clk'event and clk='1') then
         valid_reg(LAT-1 downto 1) <= valid_reg(LAT-2 downto 0);
         valid_reg(0) <= valid_in;
    end if;
end process;

valid_out <= valid_reg(LAT-1);
```

In order to be able to see what is happening in each of the signals, the waveform is shown below in Figure 3.2.

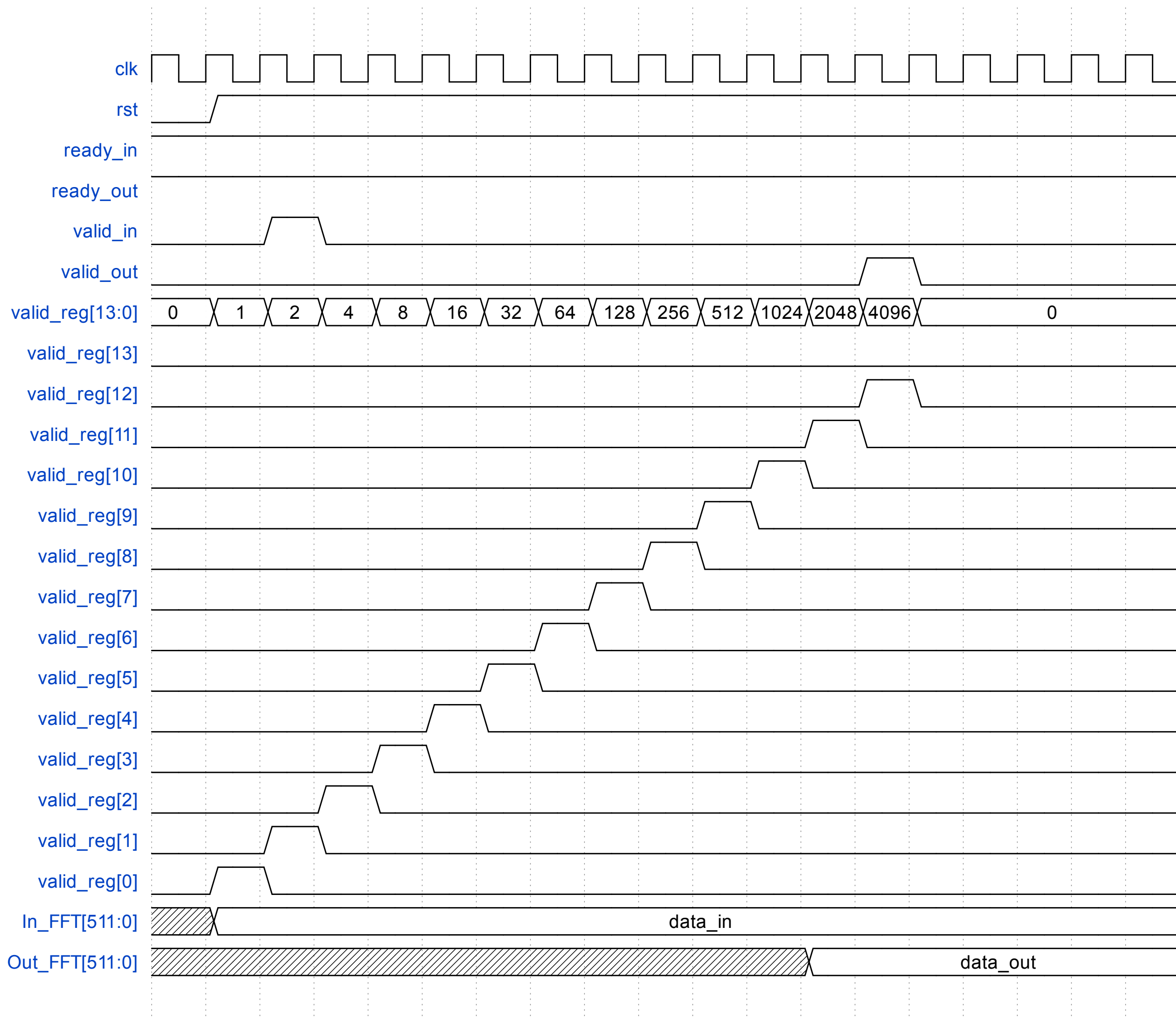

Figure 3.2: Waveform of the transmission of the *valid_in* signal.

There is one signal missing from the FFT block to explain, this is *ready_out*, which is an input that indicates when the output FIFO is not full so it can receive data. Therefore, the processed data will only be output when this signal is '1'. This can lead to data loss during the whole process, hence, a block has been designed which monitors the validation and ready signals. Figure 3.3 shows this block.

As can be seen in Figure 3.1, a slave type AXI-Lite [13] interface is used via the *S_AXI* bus. This is because the decision was taken to include 4 registers, 3 of them read-only and one write-only. The 3 read-only registers are 3 counters that are responsible for monitoring the number of data that are lost, as well as the number of packets that arrive and the number of data that are transmitted. A packet is understood as a complete transmission of several data of 512 bits each. In addition, the read register has two useful bits, one indicating START/STOP and the other one serving as a reset signal for the counters.

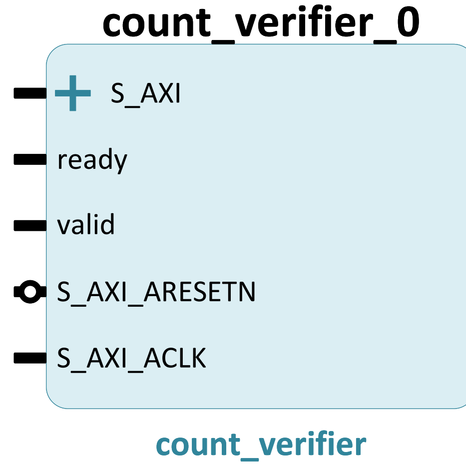

Figure 3.3: Valid-ready verifier module.

Moreover, it is important to make clear that the block shown in Figure 3.3 can be placed between any of the four main blocks shown in Figure 3.1, so it is shown separately, in order not to saturate the Figure and to explain each of the system modules separately. It should also be noted that its function is only to check that the data transfers between the modules are carried out correctly.

3.1.1 Post-implementation results

The results after implementation in Vivado are as shown in Table 3.1. They demonstrate that it takes up very little space to implement this model into the FPGA, in this case the Virtex UltraScale+ XCVU37P-L2FSVH2892E. This makes it easier to implement FFT models or larger neural networks in later work, if necessary.

Table 3.1: XDMA post-implementation results

Resource	Utilization	Avaliable	Utilization (%)
LUT	60212	1303680	4.62
LUTRAM	5727	600960	0.95
FF	69561	2607360	2.67
BRAM	113	2016	5.61
IO	1	624	0.16
GT	16	96	16.67
BUFG	27	1008	2.68
MMCM	1	12	8.33
PCIe	1	6	16.67

3.2 Data transmission between the PC and the FPGA

Once the project has been created and implemented inside the FPGA, it is only necessary to send and receive data through the Host, which in this case is a PC. For this purpose, a series of programs have been written in C language that communicate with the driver to send and receive data, based on a series of basic functions provided by Xilinx. Basically what needs to be done is, first, to make sure that there is a device to communicate with and that both channels, H2C and C2H, are detected. After that, simply by indicating the size of the frame to be sent (and therefore received) and providing a file with the data, the entire data transaction is performed, ending with the writing of the data processed by the FFT to another output file.

However, this leaves several questions unanswered. How is the data to be sent generated? How do we check that the received data has been correctly processed by the 16-point 16-parallel FFT? This will be discussed in the next subsection.

3.2.1 Testing tools

In order to verify that the values received at the output are correct, several different files have been generated as input. These files have been written as a binary file from the MATLAB tool, with the idea of representing mathematical functions of which we know the output when processed by a 16-point FFT. For example, if the input is a constant, the output has as its only frequency component the fundamental component, i.e. frequency 0. Similarly, if the input is a sinusoidal signal with $N = 16$ (number of points) and frequency $k = 2$, then the output is expected to have two components, one at frequency k (equal to two) and the other at $N - k = 14$. It is important to be clear that the value of k refers to the number of periods the sine has at N points.

To represent the output, the decision was made to make a Python script that translates the input data written in binary in order to represent it and compare the output signal with the expected signal.

First the vector X with an amplitude less than 1 is created, in this case $A = 0.1$, resulting in the function shown in Figure 3.4. This signal is then adapted for 16-bit words as in Equation (3.4) and as shown in Figure 3.5. The data taken as input is the vector Y. Finally, the output is the frequency representation of the Y signal shown in Figure 3.6.

$$N = 16,\ \ k = 2,\ \ A = 0.1 \tag{3.2}$$

$$X = A \cdot \sin(2 \cdot \pi \cdot \frac{k}{N} \cdot (0 : N - 1)) \tag{3.3}$$

$$Y = round(X \cdot 2^{15}) \tag{3.4}$$

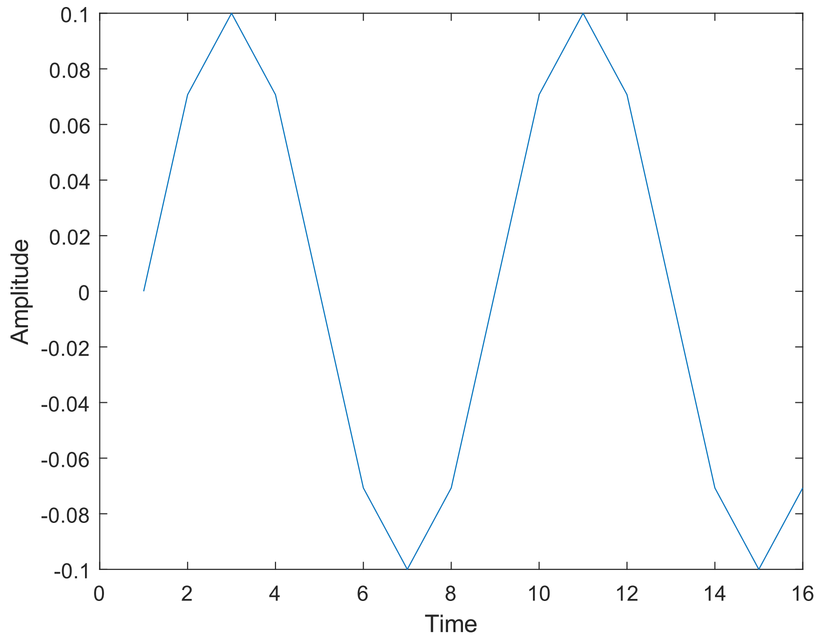

Figure 3.4: Input signal with $k = 2$ and amplitude equal to 0.1

3.3 Measurements

In order to measure the actual efficiency of the system in transmitting and receiving data, a program has been used. This program sends and receives a series of randomly generated data, so that the impact of the size of the data payload on the efficiency of the transaction can be studied.

To do this, the program measures the clock cycles that elapse since the transaction is initiated (send or receive) and compares this number with the clock cycles in which non-overhead data is being sent or received. This is thanks to the use of a structure that is part of the driver provided by Xilinx.

Furthermore, the data is presented as duty cyle, i.e., the clock cycles where the data payload is sent divided by the total clock cycles and multiplied by 100 to express it as a percentage. To know the transfer rate, when performing PCIe Gen4

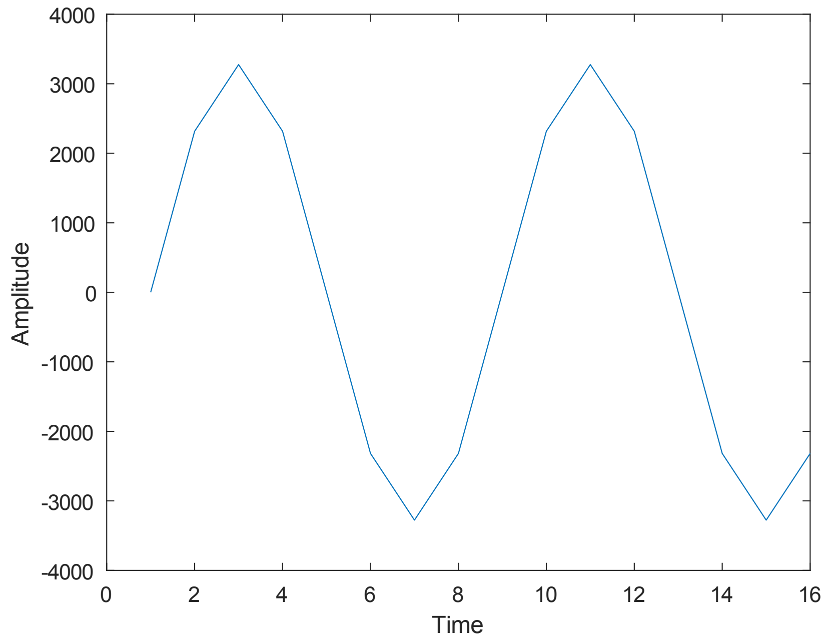

Figure 3.5: Amplitude adapted to 16 bit

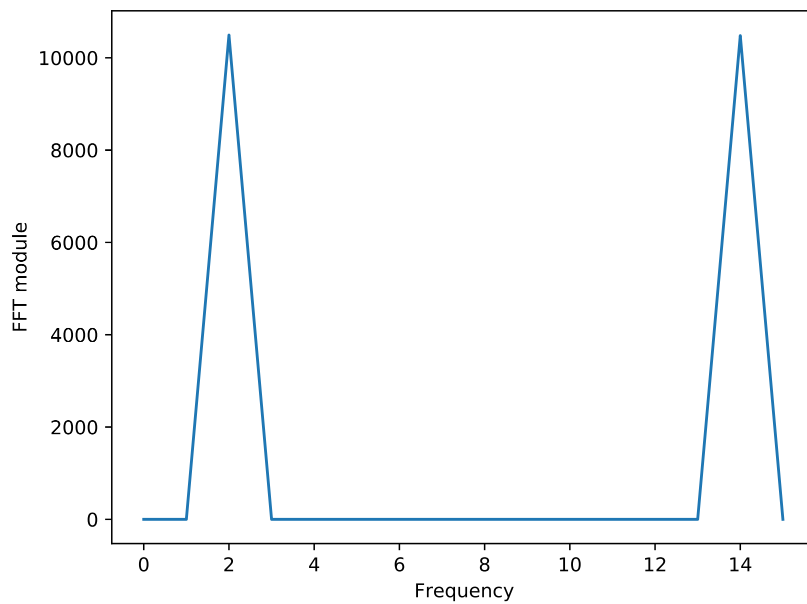

Figure 3.6: FFT of the signal with $k = 2$ and $N = 16$

with 16 lines, this duty cycle value must be multiplied by the maximum theoretical speed, which in this case is 16 GB/s = 128 Gbps.

To explain it better, an example where the measured values are clock_cycle_count = 499136320 and data_cycle_count = 18925272 is shown below.

$$\text{duty_cyle (\%)} = \frac{\text{data_cycle_count} \cdot 100}{\text{clock_cycle_count}} = \frac{18925272 \cdot 100}{499136320} = 3.79\% \tag{3.5}$$

$$\text{transfer_rate} = \text{max_rate} \cdot \text{duty_cycle} = 128 \text{ Gbps} \cdot 0.0379 = 4.85 \text{ Gbps} \tag{3.6}$$

Figure 3.7 shows a visual explanation of the duty cycle. It can be seen that when useful data is being sent or received, it is done at maximum speed, and when the overhead is being sent, it is done at zero speed. For this reason, it is necessary to take the average speed, which is calculated as explained above.

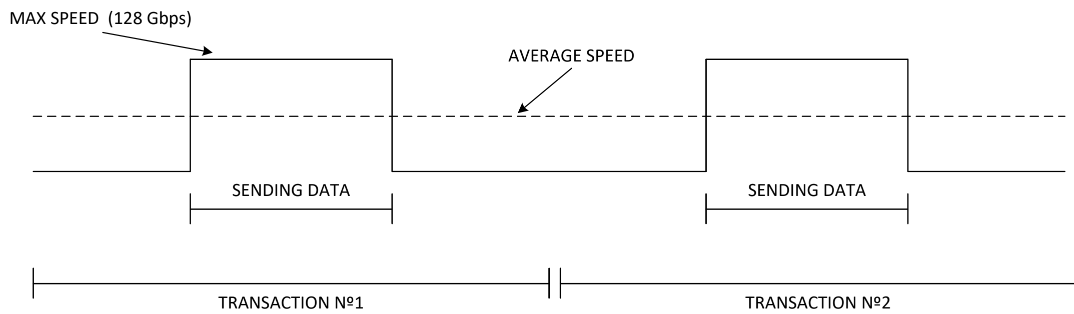

Figure 3.7: Measurement of average transfer speed

What is interesting now is to check that the percentage of overhead decreases as the size of the data being sent or received increases and so does the transaction speed. For this purpose, the transaction speed has been measured by progressively increasing the size of the data to be sent or received, as the same test has been performed independently for card-to-host and host-to-card operations.

Figure 3.8 shows the evolution of transaction speed in Gbps as a function of increasing data size. Two graphs are shown, one for host to card operations and one for card to host operations. In addition to the graph, Table 3.2 with the values of the transfer rate at each of the points on the graph is also included.

It can be clearly seen that both graphs present a part where they evolve exponentially and then a limit is reached where the transfer rate remains constant even if the data payload is further increased. The exponential part of the graph is due to the fact that the data size values also increase exponentially as $f(x) = x^2$, with $x_0 = 64$.

In addition, it is clearly faster to perform a host to card operation than a card

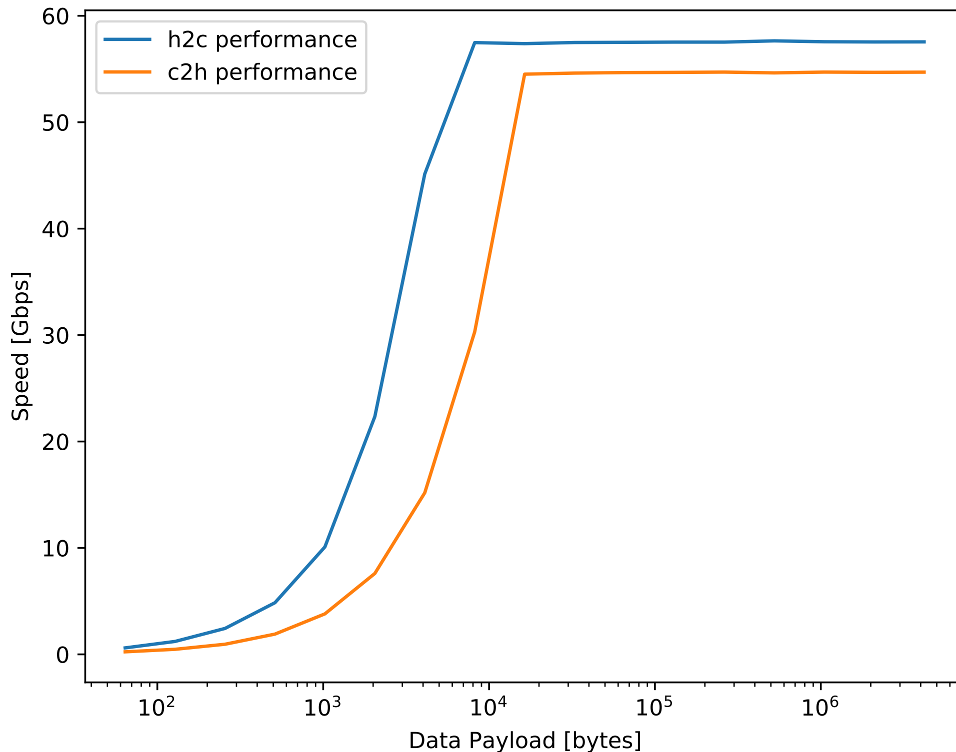

Figure 3.8: Transaction speed vs data payload size

to host operation, which is to be expected as write operations are generally faster than read operations.

Finally, it is also observed that the maximum value reached is 57.64 Gbps, which corresponds to 45% of the theoretical maximum speed. The fact that the theoretical maximum value is not reached is due to the fact that it has not been achieved that during transmission there is no waiting, i.e. that it is a continuous send/receive process. Therefore, when measuring the cycles in which data is actually sent or received and comparing them with the totals, these delays that affect the maximum transmission speed are being taken into account.

3.4 Technical challenges

During the implementation of this part of the project, several challenges were encountered and solutions had to be found. They are explained below.

In the first place, it was a challenge to meet the timming due to the use of the

Table 3.2: Values of transfer rate as a function of data payload

Data size (bytes)	Speed rate H2C (Gbps)	Speed rate C2H (Gbps)
64	0,6074	0,2376
128	1,2145	0,4750
256	2,4358	0,9503
512	4,8532	1,9026
1024	10,0973	3,8042
2048	22,3418	7,6008
4096	45,1354	15,1832
8192	57,4813	30,3050
16384	57,3843	54,5149
32768	57,4909	54,6157
65536	57,5072	54,6656
131072	57,5283	54,6830
262144	57,5279	54,7104
524288	57,6456	54,6359
1048576	57,5620	54,7105
2097152	57,5425	54,6876
4194304	57,5498	54,7048

tool provided by Vivado called "autoconnect", which interconnects modules of the block design as well as adding the IPs that the tool considers as clock wizards.

Although the tool "autoconnect" is very useful as it makes the work much easier, especially in models where the number of signals is high, it was decided to connect the signals by hand, in order to have more control over the connection between the different blocks, as well as to implement only the necessary.

In addition, the fact of having several clock domains, with synchronous and asynchronous reset signals, generated an implementation error in Vivado which does not allow the project to go ahead. As a solution, the processor system reset module block [26] was added which synchronises the reset signal for all modules in the system.

Finally, another problem encountered is that separately, the FFT in simulation processed the data correctly. However, when put together and implemented inside the FPGA, the FFT output received by the computer did not correspond to the data that should have been received.

In order to see where the error was, it was decided to set as input what were called "deltas", that is, as the FFT is divided into 16 parallel inputs, to go from one input to another, putting all 0's except in the corresponding input, which was put a 1 and the rest 0's. For example the first delta was 0x10...00, the second one was 0x00 in the first 16 bits followed by a 1 and the rest 0's.

After checking all the FFT entries, it was discovered that the program used to send the data was actually misstating the size of the data by confusing bytes with 16-bit words.

Chapter 4

Solution based on Wupper

As with XDMA, this chapter explains the project based on the Wupper. It will start by explaining what the final model consists of, showing the implementation results and ending with the measurements taken.

Unlike the XDMA, CERN provides a more complete project together with the driver, which is ported to VCU128, which is the FPGA used in this Master's Thesis. However, the Wupper was not initially designed for bifurcation, which requires more complexity and work to adapt the CERN project to the objectives of this Master's Thesis. Bifurcation consists of instantiating two PCIe devices and synchronising them to be able to implement the 32-point FFT and communicate with it.

In addition, the aim of implementing a Wupper-based model is to achieve a higher speed than that achieved in the model explained in Chapter 3. To this end, the project signals provided by CERN have been modified to include the FFT model in order to process the input data. Therefore, the main approach included in this chapter has been the adaptation of CERN's RTL project and the creation of programs to send and receive data, and measure the transfer rates.

4.1 Proposed approach

The intended 32-point 32-parallel FFT has a total of 1024 parallel input/output bits. Therefore, since there are two independent PCIe IPs, it makes sense to split these inputs and outputs of the FFT into 512 bits for each IP. Furthermore, the real part has been separated from the imaginary part, i.e. the 512 LSBs (real part) are associated with one PCIe IP and the other 512 (imaginary part) are associated with the other. In order to get a clearer idea of this, Figure 4.1 shows what the final model looks like when the FFT is included. The main parts of the system will be explained below.

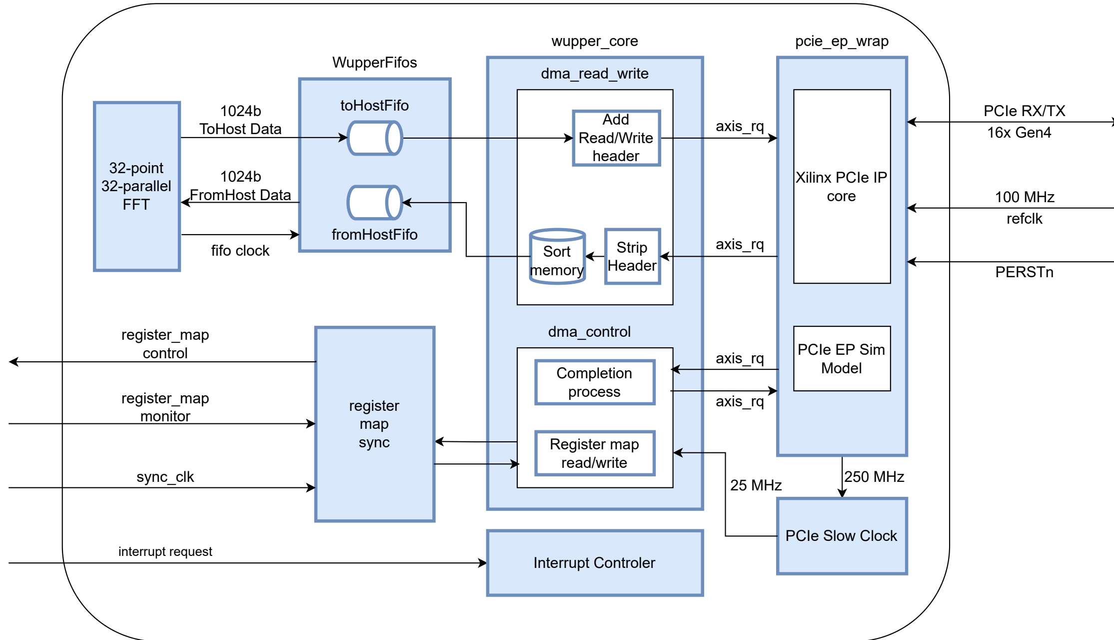

Figure 4.1: Final model with the Wupper's driver

Firstly, the *pcie_ep_wrap* consists of the PCIe interface mainly made up of the Xilinx IP of the PCIe. This block communicates with the Host (the PC) in the transmission of data.

Secondly, there is the *wupper_core*, which is made up of two main modules: the *dma_control* and the *dma_read_write*. The first entity in which the descriptors are parsed and supplied to the engine, and in which the Status register of each descriptor can be read back via PCIe. The pointer for the current address is handled by DMA Control and incremented each time a TLP completes, depending on the address range of the descriptor. The circular buffer DMA is also handled by DMA Control if the descriptor so requests. A register map for the user space register map for DMA control includes addresses to the descriptors, status registers, and external registers.

The *dma_read_write* contains two processes:

- *ToHost*: The descriptors are read and a header constructed in accordance with each descriptor during the first process. The payload data is inserted after the header if the descriptor is a ToHost descriptor and is read from the FIFO. This procedure also takes care of changing to the following active DMA descriptor, which leads to choosing the MUX on the ToHostFifo's output ports. However, in this case the MUX is not necessary as there is only one ToHostFifo (a single descriptor).
- *FromHost*: The second process involves removing the header and determining the length of the received data before shifting the payload into the FIFO.

Finally, there are the FIFOs mentioned above and lastly, there is the FFT module, which has been manually connected to the two FIFOs, as explained above.

Apart from this, in the same way as with the XDMA, to generate the *valid_out* signal, the *valid_in* signal has been transmitted as many clock cycles as the latency of the FFT, which in this case is 19 cycles. Figure 4.2 shows the waveform of the main signal in the 32-point 32-parallel FFT module.

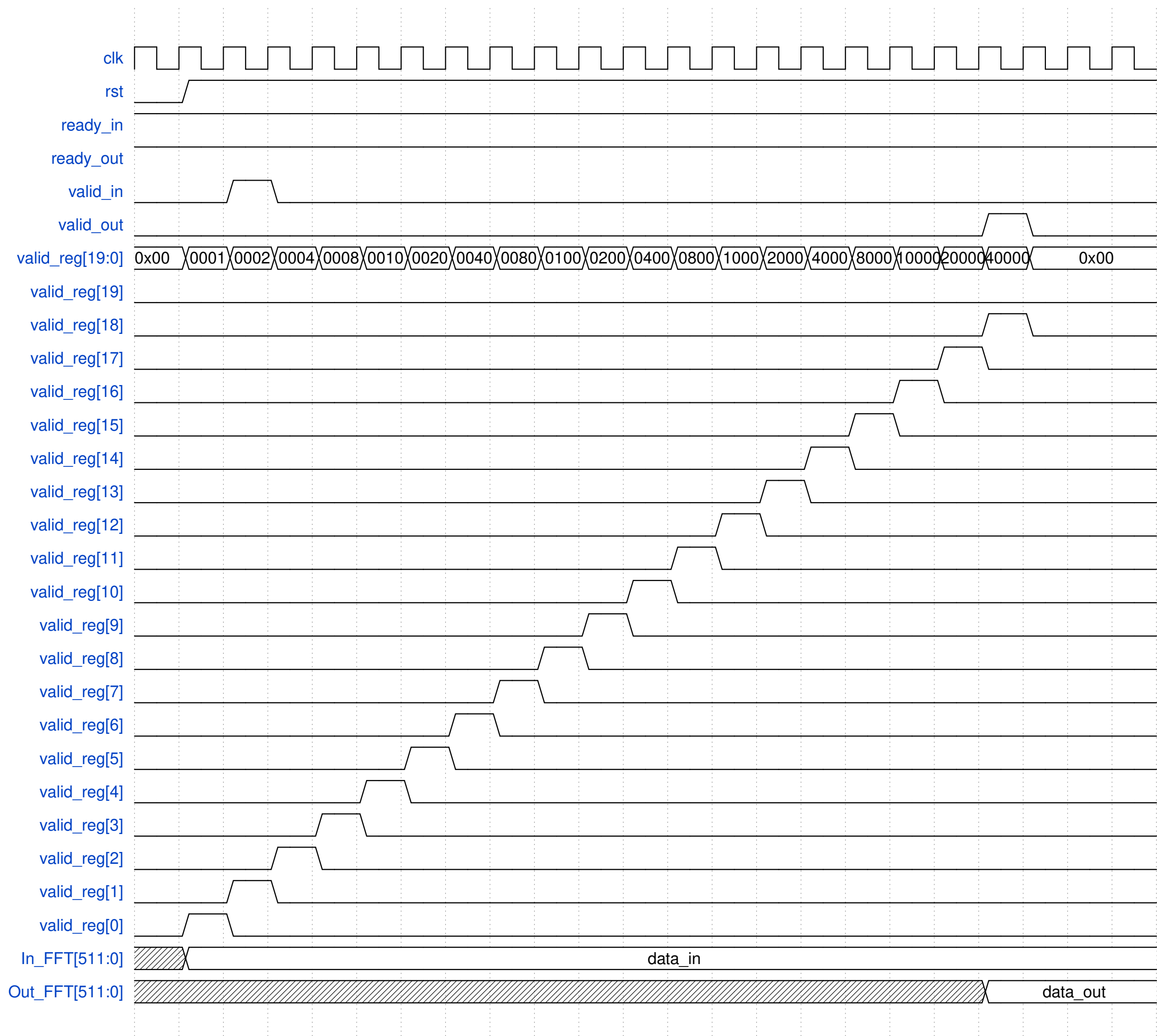

Figure 4.2: Waveform of the transmission of the *valid_in* signal

4.1.1 Post-implementation results

The results after implementation in Vivado are shown in Table 4.1. Similarly to the model explained in Chapter 3, although the FFT architecture is larger in this case, the amount of FPGA resources used is low, which allows the implementation of larger and more complex models in future tests.

Table 4.1: Wupper post-implementation results

Resource	Utilization	Available	Utilization (%)
LUT	40689	1303680	3.12
LUTRAM	3414	600960	0.57
FF	89263	2607360	3.42
BRAM	319	2016	15.82
IO	13	624	2.08
GT	16	96	16.67
BUFG	35	1008	3.47
MMCM	2	12	16.67
PCIe	2	6	33.33

4.2 Data transmission between the PC and the FPGA

As with XDMA, along with the driver, CERN provides an API with functions that can be used to send and receive data over PCIe. However, in contrast to the XDMA, and although the implementation of the project on the FPGA is relatively simple, sending data from a text file, allocated in memory, is not so easy with the Wupper.

However, a program has been written that works in a similar way to the XDMA program, except that the Wupper API's functions work more at a low level by sending as arguments the pointers where the memory reads and writes are initiated, as well as the size of the data frame and the device ID, i.e. a '0' if it is a ToHost operation and a '1' if it is a FromHost operation. It is important to mention that the data size has to be a multiple of 32 bits, as the minimum data payload is 1 DW, which corresponds to 4 bytes, i.e. 32 bits.

Basically the program sends the data specified in a file and writes it to an output file and then checks whether the output corresponds to the theoretical output of the data when processed by a 32-point FFT.

4.2.1 Testing tools

As was done to verify that the output of the XDMA corresponded to the correct output, basically the same has been done with the Wupper both to generate the input data and to verify the output data after being processed by the FFT. The only difference is that in this case it is a 32-point 32-parallel FFT and, therefore, the input signal is different.

Figure 4.3 shows the input signal with $N = 32$, $k = 2$ and amplitude equal to 0.1. Meanwhile, in Figure 4.4, the same signal is shown but adapting the amplitude for 16-bit values. Finally, Figure 4.5 shows the signal in the frequency spectrum

obtained as the output of the FFT.

As can be seen in Figure 4.5, being a signal of $N = 32$ and $k = 2$, the frequency components that can be seen are the second component, and in $N - k$, that is, 30.

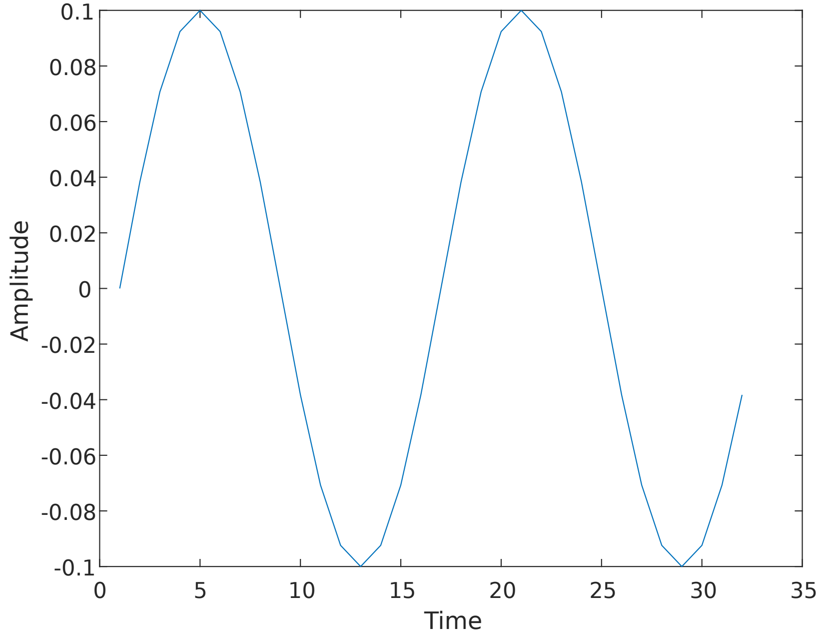

Figure 4.3: Input signal with $k = 2$ and amplitude equal to 0.1

4.3 Measurements

Similarly to the solution proposed in Chapter 3, measurements of the transfer rate as a function of the size of the data being sent have also been made for this model. To do this, in this case, API functions provided by the Wupper driver have been used to create a program that performs these measurements, which allow the user to read a specific number of 1 kB blocks from the FPGA and by means of the following equation, the transfer speed in MB/s can be known by measuring the time from the moment the first block is sent until the last one.

$$\text{speed} = \frac{\text{blocks_read} \cdot \text{blocksize}}{\text{delta} \cdot 1024 \cdot 1024} = \frac{10 \cdot 1024}{14.33\ \mu\text{s} \cdot 1024 \cdot 1024} = 681.5\ \text{MB/s} \tag{4.1}$$

Furthermore, what the driver does underneath is to allocate a 1 kB block in memory and what it does is to read from the FPGA and write to that memory space as many times as blocks are to be read.

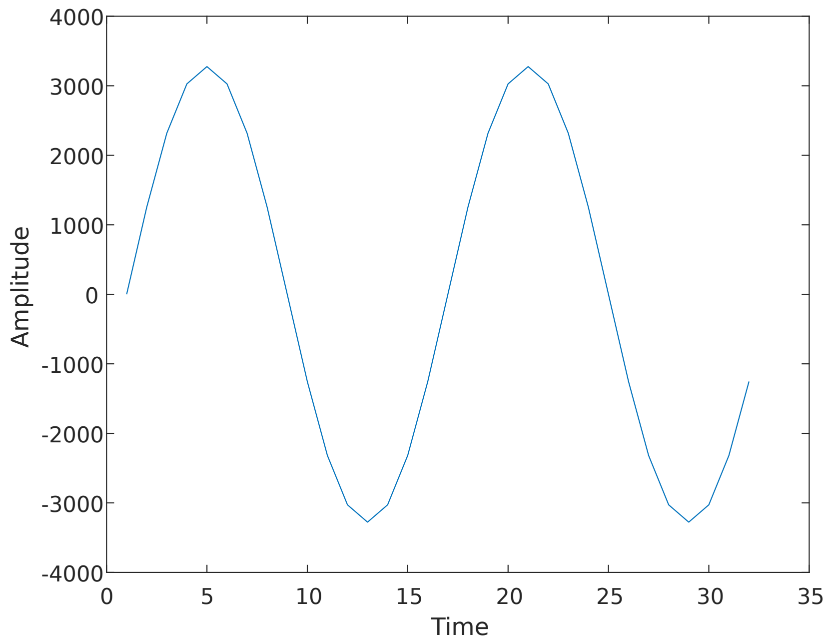

Figure 4.4: Amplitude adapted to 16 bit

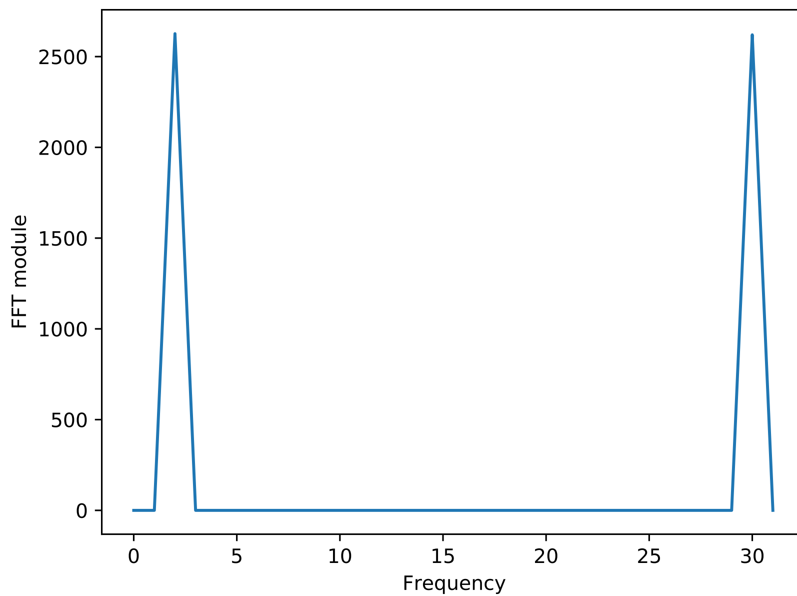

Figure 4.5: FFT of the signal with $k = 2$ and $N = 32$

In addition, it is also interesting to check here the evolution of speed as a function of data size and to see if it is also true in this case that the smaller the overhead, the higher the average speed.

For this purpose, the speed measurements as a function of the size of the data read from the FPGA are shown in Figure 4.6. In addition, Table 4.2 shows the values represented in the graph.

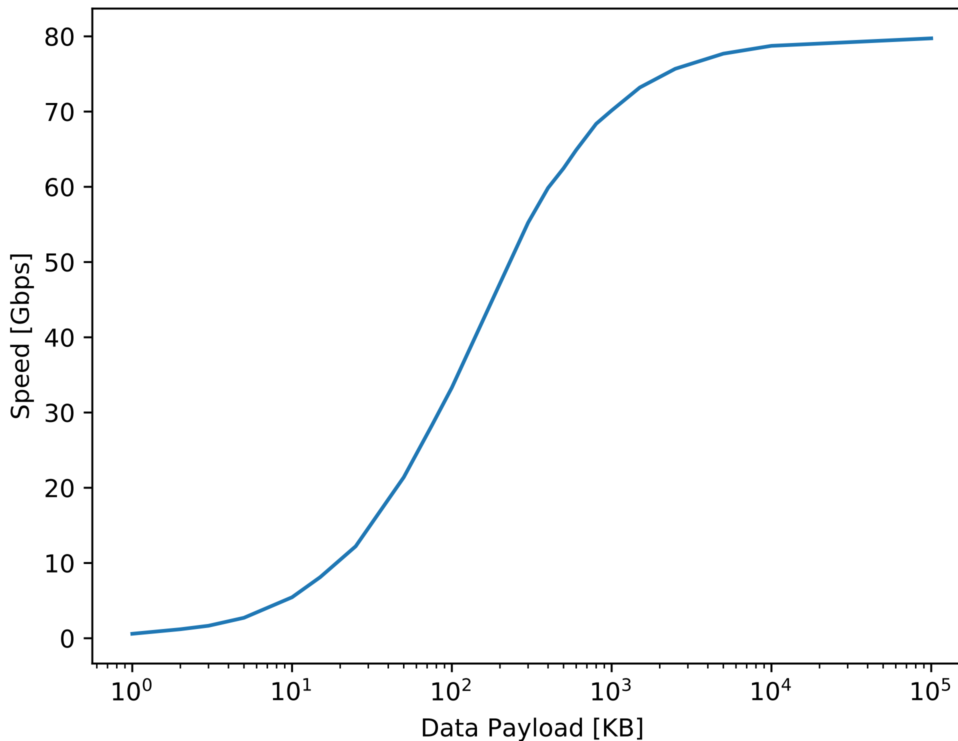

Figure 4.6: Transaction speed vs data payload size

It is indeed confirmed that as the size of the data payload increases, so does the transfer rate, reaching a limit of 79.83 Gbps, which falls well short of the theoretical maximum of 256 Gbps.

Although the transfer rate results are higher than those of the model in Chapter 3, it is still far from the theoretical maximum speed of 258 Gbps in this case. The results they provide as documentation with the Wupper do not reach the maximum speed either (less than 75%) even though the model they implement is simply hardwiring the input with the output.

As with the solution proposed in Chapter 3, the fact that the theoretical maximum speed is not reached is due to the fact that in the transmission of data there are delays in which no time is spent on sending or receiving data, but they affect the maximum transmission time and therefore reduce the average transmission speed.

Table 4.2: Values of transfer rate as a function of data payload

Nº of 1 kB blocks	Speed rate H2C (Gbps)
1	0,5948
2	1,1981
3	1,6615
5	2,7206
10	5,4499
15	8,1226
25	12,2191
50	21,3969
75	28,2959
100	33,3188
200	47,1601
300	55,2358
400	59,8572
500	62,4635
600	64,8948
800	68,3783
1000	70,1466
1500	73,2009
2500	75,6880
5000	77,6992
10000	78,7397
100000	79,7345
1000000	79,8320

4.4 Technical challenges

The main challenge in implementing this model was clearly to make the connection between the FIFOs and the 32-point FFT to be tested, because the Wupper does not provide any information about which signals have to be modified. In addition, the fact that the initial Wupper model generated the two PCIe instances using the "for ... generate" statement makes it difficult to route the signals for each of the two blocks.

In order to solve this, there were two options: Keep the "for" statement and try to connect everything correctly, or remove it and redo the whole Wupper project without the "for" statement. After serious consideration, the decision was made to keep the "for ... generate" statement and depending on the index used for the loop, the FFT signals were routed with one of the FIFOs or the other.

However, this caused another problem, which is how to route the valid and ready signals. Having two FIFOs for each operation (both read and write) means that the empty and full signals of both FIFOs have to be synchronised. The empty signals

are in charge of generating the *valid_in* and the full signals, the *ready_out*. This is best explained by the following code.

```
-- Inside of the for loop:
    valid_in_v(i) <= not fromHostFifo_empty;
    ready_out_v(i) <= not toHostFifo_prog_full;
    In_FFT(DATA_WIDTH*(i+1)-1 downto DATA_WIDTH*i)
    <= fromHostFifo_dout;
    toHostFifo_din <= Out_FFT(DATA_WIDTH*(i+1)-1 downto DATA_WIDTH*i);

-- Outside of the for loop:
    ready_out <= ready_out_v(0) and ready_out_v(1);
    valid_in <= valid_in_v(0) and valid_in_v(1);

valid_out <= valid_reg(LAT-1);
```

4.5 Comparisons

After obtaining the results of the two models, it is interesting to make a comparison in terms of performance. To this end, the Table 4.3 shows the maximum transfer speed values achieved in each case.

Table 4.3: Transfer rate of each solution

XDMA		Wupper	
reading	writing	reading	writing
54.70 Gbps	57.55 Gbps	79.74 Gbps	-

It can be seen that the speed achieved with the proposed solution based on the Wupper driver is clearly higher. However, by comparing the results achieved with the theoretical maximum values, the solution based on the XDMA driver is closer. With the XDMA, 45% of the theoretical maximum speed is achieved, while with the Wupper driver, only 31% of the theoretical maximum speed is achieved.

Moreover, in terms of implementation results, both proposals present a fairly similar use of resources with the big difference that the model explained in this chapter uses two PCIe entities due to the fact that two different IPs are being instantiated. The model explained in Chapter 3, however, only uses a single PCIe entity.

Chapter 5

Conclusions and future work

5.1 Conclusions

The objective of this Master's Thesis was to implement a test bench on which to verify the performance of complex architectures such as FFTs or neural networks on FPGAs. To this end, a study of possible starting points for adapting third-party solutions to the needs in terms of performance and FPGA used in this project (the VCU128) was initially carried out. From the study, two alternatives were found and then, implemented in this Master's Thesis. One based on a driver developed by Xilinx (XDMA), and the other based on a CERN project (Wupper).

Both proposals were able to achieve the objective of serving as a benchmark test, but the one explained in Chapter 4 (Wupper-based) achieves a notably higher transfer speed compared to the proposal explained in Chapter 3 (XDMA-based). Despite this, the XDMA-based model is much more modular and therefore much easier to implement for users who are going to use the work carried out in this Master's Thesis, in their projects.

In conclusion, the objectives of realising a test bench for advanced digital architectures such as FFTs and with a transfer rate close to 100 Gbps have been achieved. However, it was expected to achieve a transfer rate closer to 256 Gbps which is the theoretical maximum that can be achieved with a 16-channel PCIe Gen4 interface which is the one used in the solution proposed in Chapter 4.

5.2 Future work

As future work, it is proposed to use the interface developed in this Master's Thesis to implement state-of-the-art FFTs and neural networks and test them.

In addition, it is also proposed to investigate the possibility of applying bifurcation to the project based on the driver developed by Xilinx (the XDMA) and see if it is possible to implement a PCIe Gen4 interface with 16 channels and, theoretically, double the transfer speed. This is what has been implemented in the model explained in Chapter 4, so it is proposed to study the possibility of doing the same with the model explained in Chapter 3.

Finally, the FPGA used to implement the project carried out in this Master's Thesis also has an Ethernet port, so it is proposed for the future to develop an interface to communicate the FPGA and the computer through the Ethernet port and compare it with the results obtained in this thesis.

Appendix A

Budget

LABOUR COSTS		Hours	Cost/hour	TOTAL
Junior Telecommunication Engineer		400	15 €	**6.000 €**
MATERIAL COSTS	Cost (€)	Use (months)	Amort. (years)	TOTAL
Computer	700,00	9	5	**105,00 €**
VIVADO	2.995,00	9	1	**2246,25 €**
MATLAB	2.000,00	9	1	**1500,00 €**
FPGA Virtex Ultraescale+	10.262,00	9	5	**1539,30 €**
Python and Linux	0,00	9	5	**0,00 €**
SUBTOTAL DIRECT COSTS				**11390,55 €**
INDIRECT COSTS		15%	of DC	**1708,58 €**
INDUSTRIAL PROFIT		6%	of DC+IC	**785,95 €**
FUNGIBLE MATERIAL				
Office material				**15,00 €**
Printer paper				**5,00 €**
SUBTOTAL				**13.905,08 €**
IVA		21%		**2.920,07 €**
TOTAL BUDGET				**16.825,15 €**

Table A.1: Budget

Appendix B

Ethical, social, economic and environmental aspects

This annex discusses ethical, economic, social and environmental aspects related to this Master's Thesis. The general context of the proposed approach is the one that links electronic systems with telecommunication systems and signal processing.

B.1 Ethical and social impacts

The proposed approach on a test bench to verify the performance of advanced digital architectures such as FFTs or neural networks can help people who want to implement their models on an FPGA to easily verify that they work correctly.

B.2 Economic impact

Having a tool that facilitates the verification of complex digital architectures saves time and therefore money in the development of these architectures. For this reason, the project carried out in this Master's Thesis may have an economic impact on a larger or smaller scale in this sector.

In addition, the architectures to be tested will be used in 6G which will enable the development of many applications and businesses.

B.3 Environmental impact

In terms of environmental impact, power consumption is critical when working with FPGAs, so decreasing the development and verification time using these cards helps to reduce the environmental impact of FPGAs. With the work proposed in this Master's thesis, by saving time in the development of digital architectures and facilitating their verification, it is also helpful for the environment.

Appendix C

User guide

The aim of this annex is to explain the steps of the process that a person who wants to implement their models in the benchmark test carried out in this Master Thesis has to do. It is assumed that the software proposed in this project is available, as well as the drivers installed as explained in their manuals [11, 23].

C.1 Adaptation of the top model

The first thing to do is to adapt the model to be implemented to the size of the system buses and the type of synchronisation. That is, taking into account the operation of the valid and ready signals, as well as the 512-bit inputs and outputs in the case of the XDMA-based model and 1024-bit inputs and outputs in the Wupper-based model.

For architectures that do not have the same number of parallel inputs and outputs, it is recommended to parallelise those models that are serial and consider whether it is worth wasting inputs on those architectures with parallel but smaller interfaces.

C.2 Verify implementation

Once the top model has been created and the driver installed, the FPGA must be programmed. After that, reboot the system and verify that the computer detects the device through PCIe.

To check that the device is detected, run, in the command line, instruction *lspci* to get a detailed description of the devices, and $lspci\ -n$ to check the device ID. In

case the device is not detected, it is recommended to reboot the system and reinstall the drivers.

C.3 Data transmission

The user must generate a binary file with the data to be sent. Once this is done, depending on whether it is intend to use XDMA or Wupper, the procedure is differently.

C.3.1 XDMA-based data transmission

To transmit data using the model explained in Chapter 3, go to the *tests* folder in the project directory and put all the files you want to send for testing in the *data* folder. Then run the file $./test_fft16.sh\ \ < file > .bin$ as administrator as shown in Figure C.1. The output data is saved in the *data* folder as $output_ < file > .bin$.

```
simonportela@VCU128-server:~/simon/git/dma_ip_drivers_VIVADO/XDMA/linux-kernel/tests$
simonportela@VCU128-server:~/simon/git/dma_ip_drivers_VIVADO/XDMA/linux-kernel/tests$
simonportela@VCU128-server:~/simon/git/dma_ip_drivers_VIVADO/XDMA/linux-kernel/tests$ pwd
/home/simonportela/simon/git/dma_ip_drivers_VIVADO/XDMA/linux-kernel/tests
simonportela@VCU128-server:~/simon/git/dma_ip_drivers_VIVADO/XDMA/linux-kernel/tests$ sudo ./test_fft16.sh sine2k.bin
```

Figure C.1: Running data transmission with XDMA

If the user wants to test the system's transmission speed, running the file called $perform_hwcount.sh$, which performs the test explained in Chapter 3 to measure read and write speeds, is located in the same folder.

C.3.2 Wupper-based data transmission

The case of the model explained in Chapter 4 is very similar. In this case, in the $software/wupper_tools$ folder, in the project directory, run the file $./wupper - dma - transfer.sh\ \ < file > .bin$ indicating the file to be transferred, which must be in the *data* folder, as shown in Figure C.2. In the same folder as with the input data, the files with the output data are written.

```
simonportela@VCU128-server:~/simon/git/wupper_CERN/software/wupper_tools/build$ pwd
/home/simonportela/simon/git/wupper_CERN/software/wupper_tools/build
simonportela@VCU128-server:~/simon/git/wupper_CERN/software/wupper_tools/build$ sudo ./wupper-dma-transfer sine2k.bin
```

Figure C.2: Running data transmission with XDMA

In addition, if the user wants to run a test to measure the transfer rate based on the size of the data being sent, the $wupper - throughput.bin$ file must be run.

C.4 Recommendations

As a conclusion to this appendix, in case the user only wants to verify the behaviour of the system in the simplest possible way, it is recommended to use the model explained in Chapter 3, which is much easier to adapt for different architectures.

However, if the aim is to achieve the highest possible transfer speed, it is recommended to implement the model in Chapter 4, the one based on the Wupper driver.

Finally, the user should feel free to modify any of the functions provided for the transfer of data or the measurement of transfer speeds.

Bibliography

[1] VCU128 evaluation board User Guide, April 2021.
https://www.xilinx.com/support/documentation/boards_and_kits/vcu128/ug1302-vcu128-eval-bd.pdf.

[2] How PCI express devices talk.
http://xillybus.com/tutorials/pci-express-tlp-pcie-primer-tutorial-guide-1.

[3] UltraScale+ devices integrated block for PCI express v1.3 Product Guide, December 2021.
https://www.xilinx.com/support/documentation/ip_documentation/pcie4_uscale_plus/v1_3/pg213-pcie4-ultrascale-plus.pdf.

[4] DMA/Bridge subsystem for PCI Express v4.1 Product Guide, April 2021.
https://www.xilinx.com/support/documentation/ip_documentation/xdma/v4_1/pg195-pcie-dma.pdf.

[5] Wassim Mansour, Nicolas Janvier, and Pablo Fajardo. FPGA Implementation of RDMA-Based Data Acquisition System Over 100 GbE. *IEEE Transactions on Nuclear Science*, PP:1–1, 03 2019.

[6] Hiroki Nakamura, Hirotaka Takayama, Yoshiki Yamaguchi, and Taisuke Boku. Thorough analysis of PCIe Gen3 communication. In *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6, 2017.

[7] Hossein Kavianipour and Christian Bohm. High performance FPGA-based scatter/gather DMA interface for PCIe. In *2012 IEEE Nuclear Science Symposium and Medical Imaging Conference Record (NSS/MIC)*, pages 1517–1520, 2012.

[8] Jian Gong, Tao Wang, Jiahua Chen, Haoyang Wu, Fan Ye, Songwu Lu, and Jason Cong. An efficient and flexible host-FPGA PCIe communication library. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–6, 2014.

[9] L. Rota, M. Caselle, S. Chilingaryan, A. Kopmann, and M. Weber. A new DMA PCIe architecture for Gigabyte data transmission. In *2014 19th IEEE-NPSS Real Time Conference*, pages 1–2, 2014.

[10] Carsten Dulsen. A high data rate readout system for particle detectors based on FPGA-to-server ethernet connections and the eXpress Data Path technology, Feb 2021. Presented 07 May 2021.

[11] Wupper - a Xilinx Virtex-7 PCIe engine, September 2021. https://gitlab.nikhef.nl/franss/wupper/-/blob/master/documentation/wupper.pdf.

[12] Soo Ryu and. FELIX: The new detector readout system for the ATLAS experiment. *Journal of Physics: Conference Series*, 898:032057, October 2017.

[13] AXI4 Lite interface specification by ARM. https://developer.arm.com/documentation/ihi0022/e/AMBA-AXI4-Lite-Interface-Specification.

[14] AXI4 Stream protocol by ARM. https://developer.arm.com/documentation/ihi0051/a/Introduction/About-the-AXI4-Stream-protocol.

[15] M. Garrido, K. Möller, and M. Kumm. World's fastest FFT architectures: Breaking the barrier of 100 GS/s. *IEEE Trans. Circuits Syst. I*, 66(4):1507–1516, April 2019.

[16] PCI SIG oficial website. https://pcisig.com/specifications.

[17] PCI Express. https://en.wikipedia.org/wiki/PCI_Express.

[18] J. Lawley. Understanding Performance of PCI Express Systems. *Xilinx White Paper*, October 2014.

[19] PCI Utils. https://https://github.com/pciutils/pciutils.

[20] D. A. Rusling. The Linux Kernel, Chapter 6: PCI, 1996-1999. https://https://tldp.org/LDP/tlk/dd/pci.html.

[21] D. A. Rusling. Linux command 'lspci' and 'setpci', 2018-2022. https://www.programmersought.com/article/6973544207/.

[22] The PCI bus by OSDev.org. https://wiki.osdev.org/PCIBase_Address_Registers.

[23] DMA ip drivers by xilinx. https://github.com/Xilinx/dma_ip_drivers.

[24] Costa, Filippo, Chapeland, Sylvain, Alexopoulos, Konstantinos, and Fuchs, Ulric. Assessment of the alice o2 readout servers. *EPJ Web Conf.*, 245:01013, 2020.

[25] AXI4-Stream FIFO. https://docs.xilinx.com/v/u/4.1-English/pg080-axi-fifo-mm-s.

[26] Processor system reset module.
https://docs.xilinx.com/v/u/en-US/pg164-proc-sys-reset.