POLITECNICO DI MILANO
Scuola di Ingegneria Industriale e dell'Informazione
Dipartimento di Elettronica, Informazione e Bioingegneria

Master of Science in
Electronics Engineering

High Speed USB Interface for FPGA Devices

Supervisor:

DR. NICOLA LUSARDI

Co-Supervisors:

DR. FABIO GARZETTI
DR. NICOLA CORNA

Master Graduation Thesis by:

ANA ALONSO RÁMILA
Student Id n. 915753

Academic Year 2019-2020

Contents

Abstract

Currently, one of the most relevant issues present in electronics is the communications field, i.e., how to exchange data among different components. To illustrate this fact, one example would be how to connect an FPGA with the external world, like, for instance, a PC.

The task carried out during this thesis was developing part of the communication path between FPGA and PC, using as intermediate point the FT2232H device. The project consists in the implementation of the FT245 Synchronous FIFO interface supported by this chip between the PC and the FPGA. The connection with the PC is done by means of a USB cable.

Apart that, the FT2232H chip allows other sorts of communication modes, like the FT245 Asynchronous FIFO interface already done. The aim of this project was to substitute this protocol with a synchronous solution in order to achieve a higher data transfer rate.

Abstract

Al giorno d'oggi, uno dei problemi più rilevanti presenti nell'elettronica è il campo delle comunicazioni, ovvero lo scambiare dati tra diversi componenti; uno degli esempi principali è il collegamento dei sistemi a logica programmabile FPGA con il mondo esterno, come, ad esempio, un PC.

Il lavoro di tesi svolto consiste nello sviluppare parte del percorso di comunicazione tra FPGA e PC, utilizzando come punto intermedio di interconnessione il dispositivo FT2232H; il progetto consiste nell'implementazione dell'interfaccia FIFO sincrona FT245 lato FPGA e della relativa gestione software lato PC. Fisicamente PC ed FPGA sono connessi tramite un cavo USB.

Il chip FT2232H, oltre al meccanismo FIFO sincrona FT245 che sarà l'oggetto principale di questa tesi, consente altri modalità di comunicazione, come ad esempio l'interfaccia FIFO asincrona FT245; quest'ultima già implementata e testata. Lo scopo di questo progetto è la modifica, lato firmware e software, del protocollo asincrono, meno efficiente, con quello sincrono al fine di ottenere una velocità di trasferimento dati più elevata.

1 Introduction. The problem of communications

Nowadays, in the electronics' field, it arises the problem of exchanging data among the different devices managing them. Electronic devices are made of diverse components which operates with data and its flow is really relevant for their correct operation in terms of reliability, speed and efficiency. There is a wide variety of communication interfaces aiming to solve this problem and, as a first approach, it is possible to classify them regarding the distance that they have to cover.

- Intrachip buses. This type of buses are used to exchange data in the short distance domain, for communications inside the chip. The order of magnitude of the distance traveled is about mm, with a maximum of 1 cm approximately. Protocols for this gaps are, for example, the AXI and the AXI Stream.
- Interchip protocols. They are for communicating different elements inside the board. They cover spaces until around 10 cm. The protocols offered by the FT2232H chip fit in this category, for instance, the FT245 Synchronous FIFO mode. Also the SPI enters in this group.
- Outside the board. For exchanging data with the external world. For instance, to share the data with a PC. In this category it is possible to find the USB (until 5

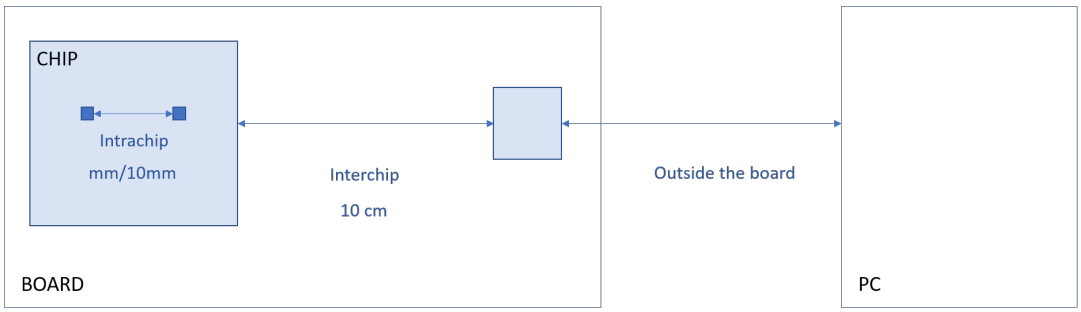


Figure 1.1: Buses overview.

meters), RS232 (until 10 m) and the Ethernet (until 100 m).

It is important to point out that not all buses are suitable for all distances: there is no point in using an AXI Stream for connecting two PCs: each one is specially design for a specific purpose and works in its range. Each bus works following a protocol and uses a physical layer.

For this thesis, a FT245 Synchronous FIFO mode has been implemented to communicate a board with the PC via USB.

1.1 Parallel and Serial architectures

Regarding to the topology, buses can be classified into serial and parallel structures:

In the **parallel communication**, the bits are sent simultaneously, one wire is used for each bit: for example, if the data width to be transmitted is 8 bits, the interface will have 8 wires plus the control signals, which are also necessary. This type of interface presents some problems related to its complexity:

- It requires a lot of wiring.
- Routing. The routing becomes complex because the higher the number of wires, the more difficult is to find a specific path among them [1].
- Crosstalk. When a current flows through a wire, it creates an electromagnetic field around it. But also a magnetic field can create a current into a wire, so the magnetic field created by each wire in the parallel structure can affect the wires besides them introducing some noise in the signals, thus affecting the quality of the transfer [1].
- Skew. It can happen that the wires constituting this sort of communication do not introduce the same delay (for example, if they have different lengths). In this way, the bits forming a word would not arrive at the same time, making the receptor wait until the last bit arrives.
- There is only one path between transmitter and receiver, so the same bus is used for receiving and sending data, being impossible to do both operations at the same time.

The advantage of the parallel communication is that it does not introduce the delay of storing and decoding the data as the serial communication does [2].

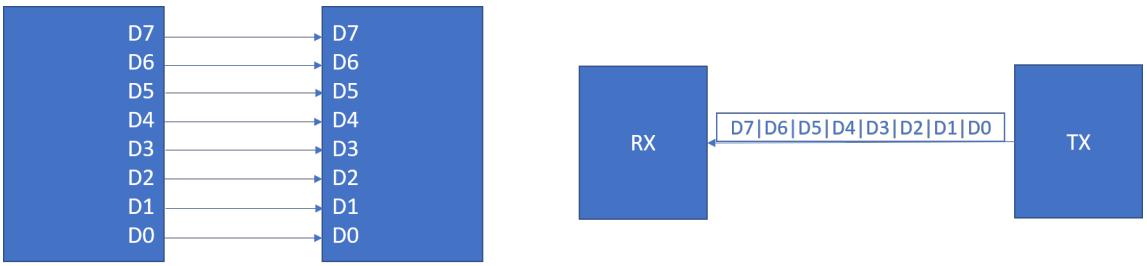


FIGURE 1.2: Parallel (left) vs serial communication (right).

On the other side, the **serial communication** only uses a wire to send the data one bit after the other. This eliminates the complexity introduced in the parallel interface and solves all its drawbacks:

- Easier routing as there is only a wire for all the data.
- There are less interferences among wires introducing less noise.
- There are not different delays for each bit because they arrive one after the other. The problem here is that it is necessary to wait until the end of the transmission [3].
- In the serial communication there is a wire for sending data and another wire for receiving it, thus it is possible to send and receive simultaneously.
- Less space is needed.
- Serial communication is also better suited for fully-differential configurations because it uses less wires than the parallel one [4].

At last, even though the serial communication may seem slower than the parallel one because it does not send the bits simultaneously, it turns out to be even faster: as it does not present the skew issue, it can be clocked much more faster than the parallel one, resulting in a higher transfer speed.

In the subsequent lines some interfaces that use these topologies are exposed:

I2C, SPI, USB, UART, PCI-Express and the Ethernet are examples of serial communication, whereas the PCI, IDE or GPBIO are based in parallel structures.

1.2 Modes of transmission

Regarding the modes of transmission of the data, another classification can be done. It refers to the way the devices exchange information among them, which can be unidirec-

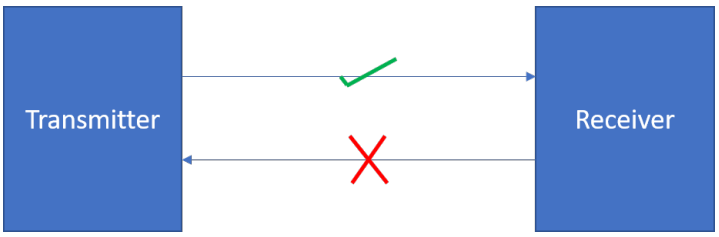


Figure 1.3: Simplex mode.

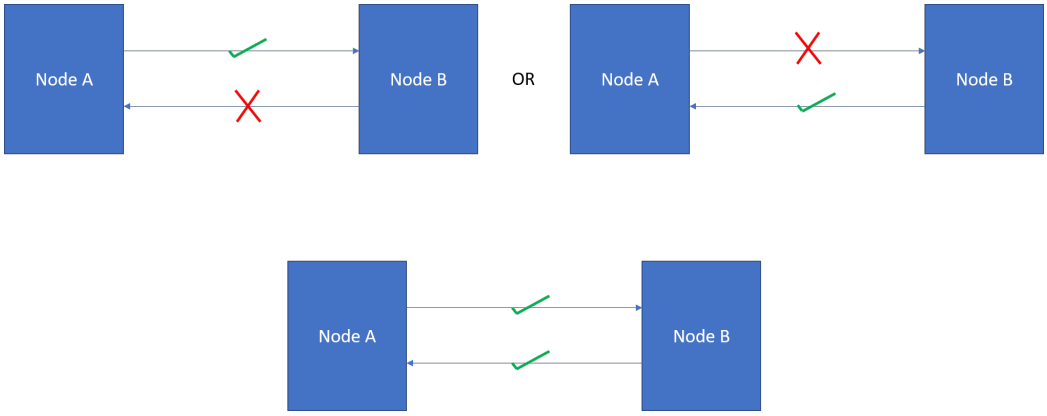


Figure 1.4: Half duplex (top) and full duplex (bottom) modes.

tional (simplex) or bidirectional (duplex) [5, 6].

The simplest one is the **simplex** communication system. It is a mode in which the flow of data occurs only in one direction, it goes from the sender to the receiver and cannot be reversed [6, 7, 8].

In the **half duplex (HDX)** mode, the stream of information is bidirectional, each party can send or accept data, however, it cannot be done simultaneously. While one node is transmitting data, it cannot receive and viceversa, it can be seen as two simplex in opposite directions. One example of this is the Walkie-talkie [5, 8, 7].

On the other hand, the **full duplex (FDX)** system also allows the flow of information in both directions, the difference with respect to the half duplex is that two or more stations can exchange data at the same time. It is not necessary to wait until the end of the reception to start sending new data, it can be done simultaneously. For instance, the phone works following this principle: both people can listen and speak at the same time [5, 7, 8].

2 Elements involved

The problem faced in this thesis, as said before, is the development of a synchronous connection between the FPGA and the FT2232H chip. This chapter will address an overview of the different elements involved in the project as well as an explanation of their main features.

The elements considered are either physical devices or communication protocols.

2.1 USB connection

The USB (Universal Serial Bus) is a sort of serial communication which is mainly used as a way to allow the connectivity of a wide range of peripherals to a computer and exchange data with them. It was created in 1994 by several companies that arrange together for this purpose. According to the classification done in the previous chapter, the USB protocol is a half-duplex interface used for communications outside the board.

First of all, it is a hierarchical protocol with only one master and a variable number of slaves. The master of the protocol is called "host" and it is generally the computer. To the host it is possible to connect either peripherals directly (also called functions) or hubs, which are points where more than one device can be connected (or even more hubs), they act as ports for more peripherals. In this way, a structure with different levels is obtained, as shown in Figure 2.1.

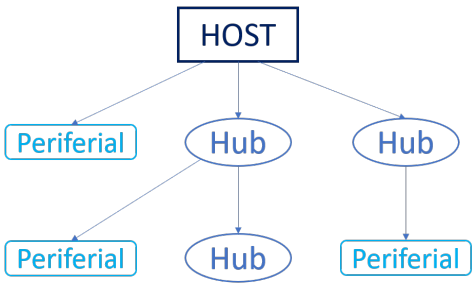


Figure 2.1: USB network structure.

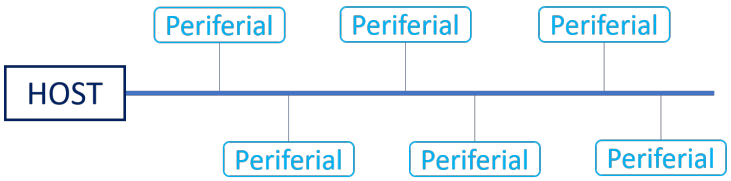


Figure 2.2: USB bus data structure.

Even if it appears like a tree layout, for practical purposes it behaves like a bus along which the different signals can travel (Figure 2.2).

With these arrangement, each device connected to the host needs a way of identification in order to the host to recognize it and exchange data properly. To this aim, every time a new device is attached to the network, it gets an address, which is its identification number. This address can be different each time the device is connected and it is unique, i.e., two attached devices cannot have the same address.

Apart from that, each peripheral also contains what is called endpoints. An endpoint can be either a source or an end of a communication path and they can act as input or output. Each peripheral device always has an endpoint 0, used for the connection and configuration of the device.

Thus, the way of selecting the correct path (pipe) for the information to follow is done through the combination of address and the endpoints [9].

In addition, there are four types of transfer a pipe can carry out:

- Control: This sort of transfer is used for configuration and control purposes.
- Interrupt: It used for small transmissions with priority.
- Isochronous: In this type, it does not matter if some errors occur, the priority here is to maintain a constant transfer rate. Before configuring a pipe like this, a control transfer takes place to check if it is possible, i.e, to verify that the USB can manage the amount of information needed.
- Bulk: For sending large amounts of data. It has less priority since it requires to have enough available bandwidth to transfer [10].

2.1.1 USB Operation

A USB port is made of four contacts, two of them for the positive and negative power supply (V+ and V-) and the other two carry out the transmission of data: the data +

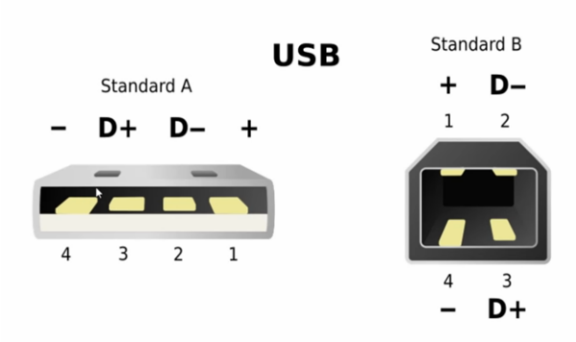


Figure 2.3: USB port [11].

and data - pins (D+ and D-).

The data pins for the exchange of information are differential: what matters is not the separate voltages, it is the difference between them. The main reason for having two data wires is to get rid of the noise. As the USB is a sort of serial communication, the information is sent bit by bit along the wire.

At last, the USB connection has been developed along the years with an improvement in the data transfer rate. So the following classification applies regarding the speed [12]:

- Low Speed (USB 1.0). They operate at a maximum speed of 1.5 Mbps.
- Full Speed (USB 1.1). Speeds up to 12 Mbps.
- High Speed (USB 2.0). It can reach until 480 Mbps.
- Super Speed (USB 3.0). It reaches a higher data transfer rate (5 Gbps).

Among these, the type of the connection used in this thesis is an USB 2.0.

2.2 FT2232H chip

The FT2232H is a USB 2.0 High Speed (480 Mbps) to UART/FIFO integrated circuit. It can be configured in a wide range of serial and parallel protocols [13]. For this thesis, the ones which matter are the asynchronous and the synchronous FT245 FIFO protocols.

The main difference between a synchronous and an asynchronous protocol is that the synchronous interfaces use a clock to, as the name says, synchronize the signals involved in the process, whereas the asynchronous modes do not follow any clock signal.

In the paragraphs below a panoramic of the FT245 Asynchronous and Synchronous FIFO modes offered by this chip will be presented, as well as their basic working prin-

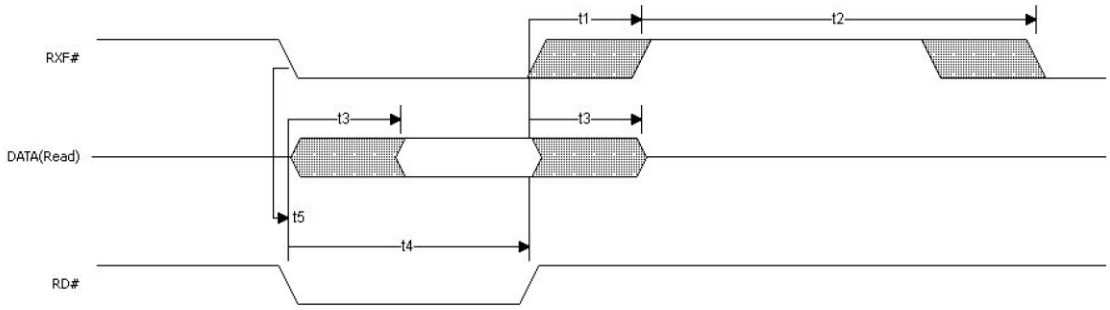


FIGURE 2.4: FT245 Asynchronous FIFO mode waveforms for reading [13].

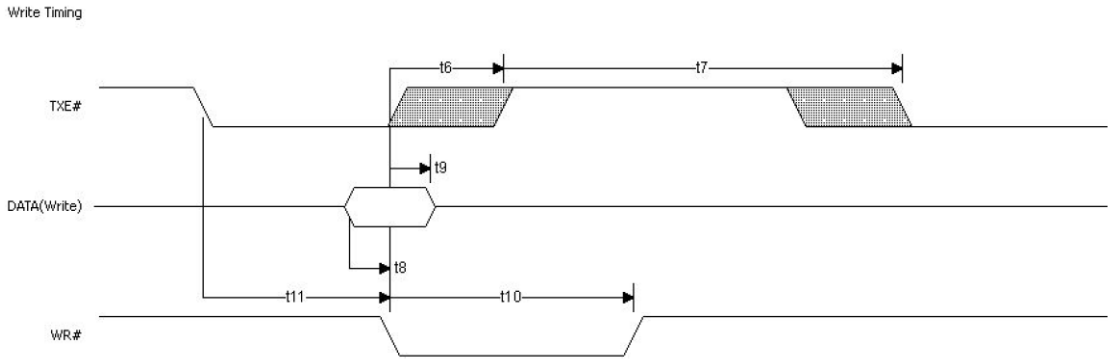


FIGURE 2.5: FT245 Asynchronous FIFO mode waveforms for writing [13].

ciples. This will help the reader to identify the pros and cons of each protocol and the reason for implementing the synchronous one.

2.2.1 FT245 Asynchronous FIFO Interface Mode

This is the protocol already configured and implemented in the board used in the laboratory, here, in building 24. It is an interchip, serial, half-duplex protocol. The reasons for programming it first are:

- Easier to implement.
- Shorter wires.

The operation of this mode can be seen in the Figures 2.4 and 2.5. Also the Table 2.1 with the corresponding timings is included for a better comprehension of the signals.

Regarding them, the transmission process begins with the falling edge of the signals RXF or TXE, which are provided by the chip and indicate whether the chip can read or

Name	Minimum	Typical	Maximum	Unit	Description
t1	1		14	ns	RD inactive to RX
t2	49			ns	RXF inactive after RD cycle
t3	1		14	ns	RD to DATA
t4	30			ns	RD active pulse width
t5	0			ns	RD active after RXF
t6	1		14	ns	WR active to TXE inactive
t7	49			ns	TXE inactive after WR cycle
t8	5			ns	DATA to WR active setup time
t9	5			ns	DATA hold time after WR inactive
t10	30			ns	WR active pulse width
t11	0			ns	WR active after TXE

Table 2.1: FT245 Asynchronous FIFO mode signal timings [13].

has data to be sent, respectively. The exchange of data starts when the external device drives down RD or WR, for the reading or writing operations respectively.

But, unfortunately, the asynchronous protocol has some disadvantages that should be avoided:

- It is slow
- It has sensitive connections

2.2.2 FT245 Synchronous FIFO Interface Mode

The FT245 Synchronous FIFO Interface Mode (serial, interchip and half-duplex) is a single channel protocol that can be configured using one of the channels of the FT2232H chip. It is the interface implemented during this project to overcome some of the problems mentioned about the FT245 Asynchronous FIFO mode.

On the one hand, the main advantages of this interface over the asynchronous one are:

- It is faster
- It has stronger wire connections

On the other hand, however, the FT245 Synchronous FIFO mode is more complex and difficult to implement. That is why the asynchronous mode was done in first place.

The behavior of this interface, i.e., how the FT2232H chip exchanges information with another device using this mode, is shown in Figure 2.6, where it is possible to see the

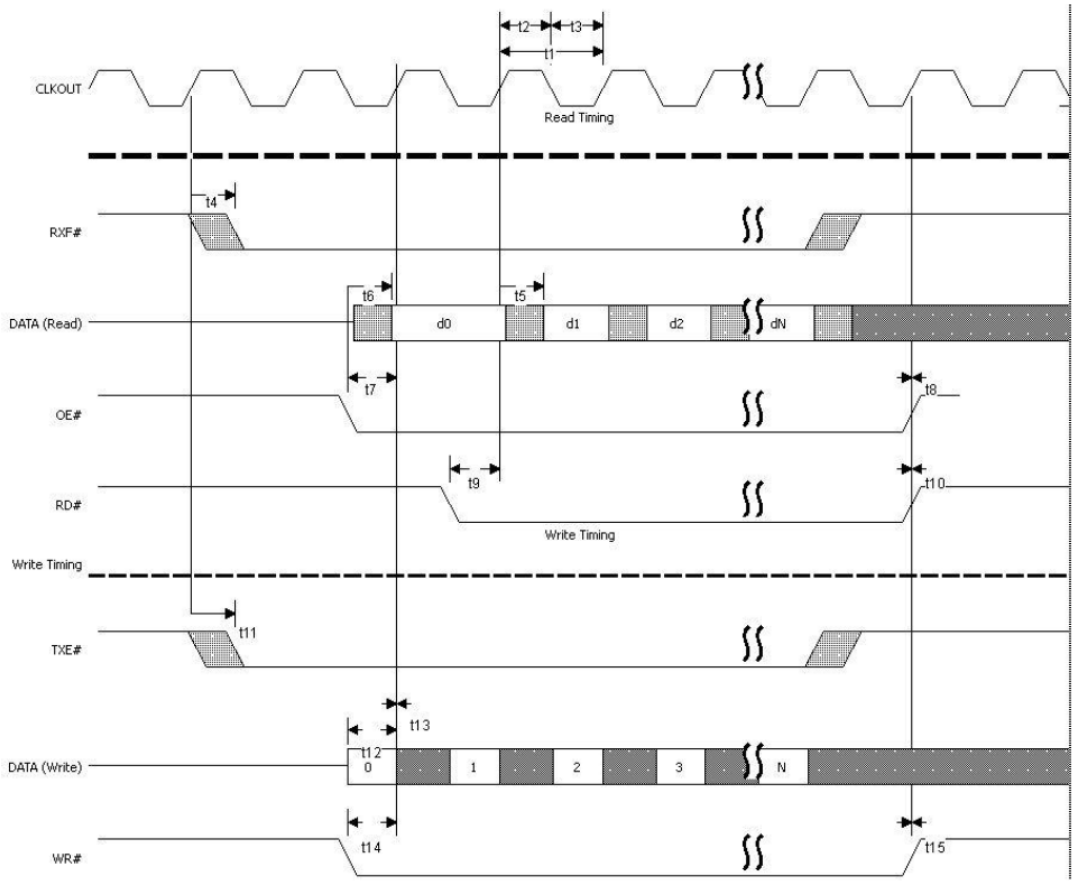


FIGURE 2.6: FT245 Synchronous FIFO mode waveforms [13].

waveforms of the different signals managed by the chip. There is also included Table 2.2 with the timing corresponding to those waveforms: it will be useful in the next chapters.

The signals are:

- For the reading operation:
  - RXF. Output of the chip. It indicates that the chip has some information that can be read by an external device.
  - OE: Input of the chip. This signal is managed by the receiving system indicating that it is ready to accept the data. Once it is asserted to 0, the chip charges the first data byte into the data bus, which is waiting until RD is driven low.
  - RD: Input of the chip. It is put down by the external device. It must be asserted at least one clock period after OE is driven low.
- For the writing operation:

Name	Minimum	Typical	Maximum	Unit	Description
t1		16.67	16.67	ns	CLKOUT period
t2	7.5	8.33	9.17	ns	CLKOUT high period
t3	7.5	8.33	9.17	ns	CLKOUT low period
t4	1		7.15	ns	CLKOUT to RXF
t5	1		7.15	ns	CLKOUT to read DATA valid
t6	1		7.15	ns	OE to read DATA valid
t7	8		16.67	ns	OE setup time
t8	0			ns	OE hold time
t9	8		16.67	ns	RD setup time to CLKOUT (RD low after OE low)
t10	0			ns	RD hold time
t11	1		7.15	ns	CLKOUT to TXE
t12	8		16.67	ns	Write DATA setup time
t13	0			ns	Write DATA hold time
t14	8		16.67	ns	WR setup time to CLKOUT (WR low after TXE low)
t15	0			ns	WR hold time

TABLE 2.2: FT245 Synchronous FIFO mode signal timing [13].

  - TXE: Output of the chip. When the chip puts down this signal, it means that it is ready to read information from the other device.
  - RD: Input of the chip. Driven by the external device, it is the starting point of the transfer of data.

All of these signals are active low. Apart from them, there are also:

- Clock 60: Output of the chip. It synchronizes all the process. All the signals involved refer to it, it is an output in order for the external system to use it and to change the values of the different signals according to it.
- SIWU: Send Immediate or Wake Up signal.
- Data bus: It acts as input/output data bus. It is a three states buffer: it acts as input or output depending on a third signal which acts as enable.

In the reading operation the chip first puts down the RXF signal. When the device is ready, it will assert OE and, a clock period later, RD. The reading operation takes place when both, RXF and RD are low and it finishes when one of these signals is driven high.

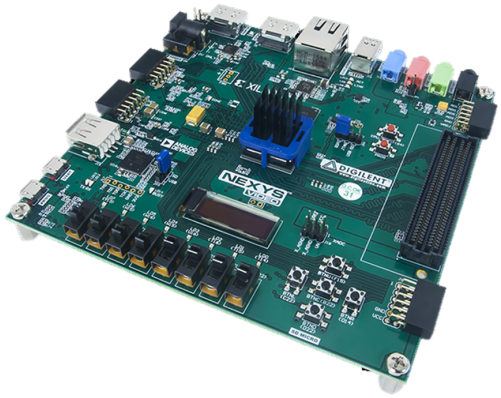


Figure 2.7: Nexys Video [15].

Besides that, the writing operation begins with the assertion of TXE to 0 by the chip. Then, if the device can accept the data transfer, it will put down WR. The writing operation takes place while these two signals are low [13].

2.3 FPGA

A Field Programmable Gate Array (FPGA) is a semiconductor device formed by configurable logic blocks which are linked using programmable interconnections [14]. Thus, it is an integrated circuit that can be reprogrammed by the final user to adapt it to the desired functionality. Due to their reconfigurable nature, they are suitable for a wide range of applications in different fields.

The board used for the development of this project was the Nexys Video (Figure 2.7), a FPGA board from the Nexys family including a Xilinx Artix 7 FPGA chip [15]. It is equipped with a wide variety of peripherals, like the high speed USB, and a huge range of features, among which is the FT2232H chip needed for the USB to FT245 Synchronous FIFO mode connection object of this thesis.

For programming a FPGA two languages can be used: Verilog or VHDL, then, the code is subjected to a sequence of processes in order to translate it into a logic circuit and implementing it on the FPGA:

1. Synthesis. It converts the design (described by the code) into a gate-level representation (netlists) [16].

2. Implementation. It takes the netlists generated during the synthesis and maps it on the resources of the device. It also manages the design to comply with the

timing constraints imposed [17, 18].

3. Bitstream generation. It translates the implemented design into a programming file that can be downloaded onto the FPGA [19].

All these operations are performed in Vivado environment.

2.4 AXI4 Stream Protocol

The AXI4 Stream protocol is a standard interface which allows the exchange of data among different components in the intrachip domain. It is of type Simplex, only allows the data stream in one direction. It can connect only a single master-slave pair or even a higher number of them.

This protocol is the way the FPGA used in this project will be connected with the module implemented in this thesis. This type of interface, as said above, has two sides which exchange the data:

1. Master. It is the part which sends the data

2. Slave. It receives the data

For the task carried out in this project, two pairs master-slave will be used, one for receiving information and other one for sending. The signals involved in this type of interface are [20]:

- TVALID: It is driven by the master entity indicating that that the master can proceed to send data.

- TREADY: Asserted by the slave side. When its value is 1, it means that the slave can accept the transfer of data.

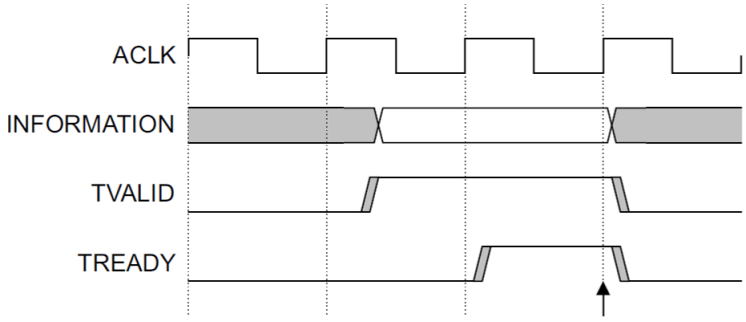
- TDATA: It is the data bus transmitted from the master to the slave.

- TSTRB: It provides information about the data to be transferred: each bit of this signal is associated with a byte of Tdata indicating if it is a position byte or an information byte.

- ACLK: Clock signal.

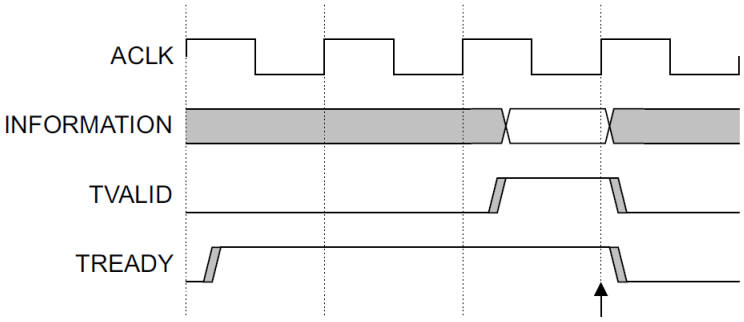
- ARESETn: Active low reset.

The operation of the protocol is simple: the transfer of data only occurs while Tvalid and Tready are high at the same time, i.e., when a handshake takes place, it does not matter which of them is asserted first or if both of them are asserted in the same clock cycle, as shown in Figure 2.8.

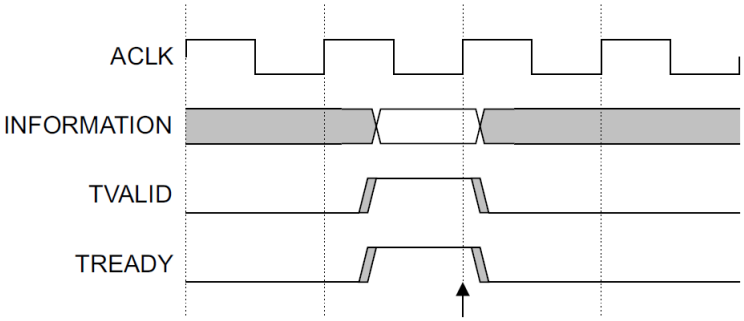


a) Tvalid before Tready.

3



b) Tready before Tvalid.



c) Tvalid and Tready at the same time.

Figure 2.8: Handshake examples [20].

3 Implementation

As it was already said, the purpose of this thesis is the development of a FT245 Synchronous protocol for a USB connection. The general structure of the whole data path is represented in Figure 3.1

First of all, data is sent from the PC to the chip by using a program in C: the communication between them is done by means of specific libraries which contain some functions to read/write in the chip and to configure its different parameters. In the following paragraphs it will be explained how it was done for this thesis.

Then, there is the connection between the chip and the FPGA. Here, the FT245 Synchronous FIFO mode is used.

3.1 Communication PC-FT2232H

As seen in Figure 3.1, the first part of the communication path is constituted by the PC, the FT2232H chip and the USB connection between them. A program in C was written and executed from the PC side for exchanging data with the chip: it makes use of some specific functions of the FTDI chips to communicate with the FT2232H [21]:

- FT_openEx. This opens the device to establish a communication with it. It is done by means of the identification number with which the device is connected to the computer via USB.
- FT_SetBitMode: To specify the behavior of the chip. In this case, the FT245 Synchronous FIFO mode was selected.
- FT_GetStatus: By calling this function it is possible to obtain some information about the device in that moment. It gives the number of bytes in the transmit and receive queues.
- FT_read: For reading data from the chip. It is necessary to specify how many bytes are going to be read and where they are going to be stored. Once executed, it returns how many bytes were read. Before FT_read, it is useful to call

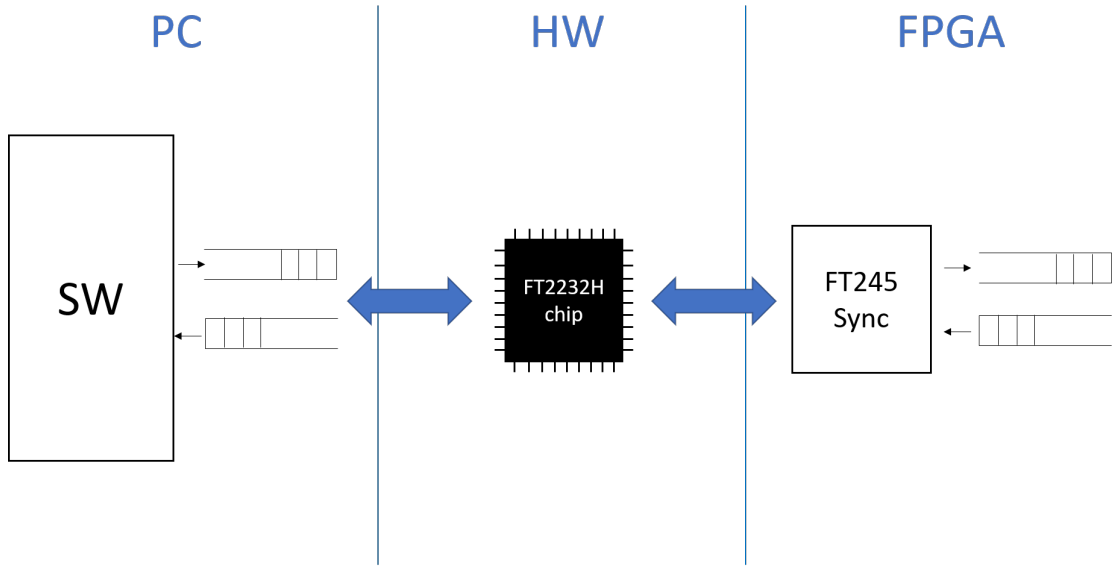


Figure 3.1: Structure of the data path.

FT_GetStatus in order for the chip to tell the computer how many bytes has ready to be sent.

- FT_write: For writing data in the chip. In the arguments there are the number of bytes that the user wants to write in the chip, the array containing them and a pointer for returning the number of bytes actually written in the external device.
- FT_Close: For closing the device after the communication has finished.

3.2 FT245 Synchronous FIFO Mode

The right half of Figure 3.1 represents the second part of the communication path: the connection between the FT2232H chip and the FPGA by means of a block called "FT245 Sync" (the object of this thesis). This module exchanges data with the chip and with the FPGA and it can be divided in three different parts (Figure 3.2):

1. FT245 module, where there is the logic of all the process: it manages all the signals in order to follow the behavior of the Synchronous interface described in Figure 2.6.
2. RX and TX FIFOs, which are the receiving and transmission FIFOs for the FPGA. They follow the AXI4 Stream Protocol explained in Section 2.4. The RX FIFO is

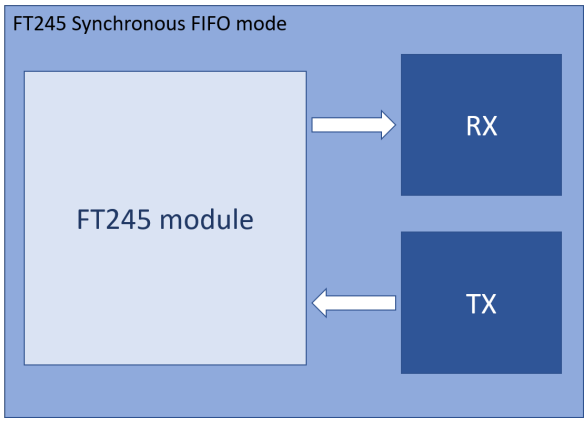


FIGURE 3.2: Block structure of the FT245 Synchronous FIFO interface.

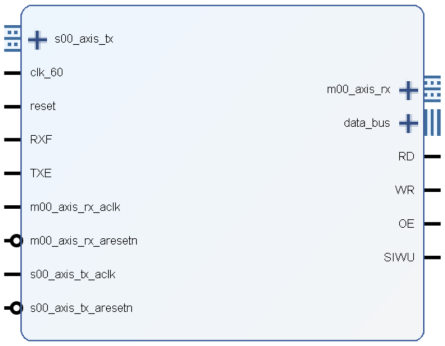


FIGURE 3.3: IP Core Implemented.

of type master and would have its symmetric slave in the FPGA side. In the same way occurs for the TX, which is an slave having its master couple in the FPGA side.

Data coming from the chip passes through the FT245 module and then, it is stored in the RX FIFO before entering the FPGA. Similarly, data coming out the FPGA enters the TX FIFO before passing through the FT245 module in its way to the FT2232H chip.

This interface has been implemented by programming in VHDL: the code done is treated by a software called Vivado and configured in the FPGA, i.e, the VHDL code, by means of this tool, is translated into a physical logic circuit that is implemented in the FPGA, as explained in Section 2.3.

The whole code can be seen like an independent block which has some inputs and, as a function of them, provides some outputs following the behavior of a FT245 Synchronous FIFO mode, as shown in Figure 3.3. In Vivado, a block of this characteristics is called IP-Core.

The input and output signals of this Ip Core (in the Image 3.3, the inputs appear on the left and the outputs on the right side of the block) are the ones in charge of, on the one hand, communicating with the chip and, on the other hand, exchanging data with the FPGA using the AXI4 Stream Protocol. All of them are explained in section 2.4 and subsection 2.2.2. The signals referred to the master RX FIFO are specify as m00_axis_rx and the ones referred to the slave TX FIFO, as s00_axis_tx.

3.2.1 Program structure and functioning

The block in Figure 3.3 represents the outer level of the FT245 Synchronous FIFO interface, but actually, it is structured in the three blocks explained before which are implemented in four VHDL files:

1. The FT245 module. Basically, it was done as a state machine: it has a waiting state and when the chip drives low RXF or TXE, it enters in the reading or writing modes respectively. It follows the behavior explained in Subsection 2.2.2, exchanging data with the chip and with the RX and TX FIFOs: when it is in the reading mode, the data coming from the chip is stored in the RX FIFO, whereas in the writing mode, the bytes coming from the TX FIFO are written in the chip.

2. RX master. It behaves according to the AXI4 Stream protocol, managing the necessary signals to communicate with its couple in the other side. It does not store the data, but it decides when the data is taken out from the receiving FIFO RX and sent to the FPGA: it interacts with the RX FIFO and with its corresponding slave.

3. TX slave. It implements the AXI4 Stream slave interface, it handles signals with its master pair in the FPGA side and with the transmission FIFO TX, accepting or not the data coming from the FPGA.

4. The wrapper. This file instantiates a block of each type (a state machine, a RX master, a TX slave and two FIFOs for storing the incoming and outcoming data) and connects the signals among them.

The two FIFOs used (RX and TX) are asynchronous FIFOs of the XPM library of Vivado. The point of using asynchronous FIFOs for this mode lies in the fact that this is an interface using two clocks. On one side there is the chip, with its clock of 60 MHz, according to which the signals of the FT2232H chip side are managed and, on the other

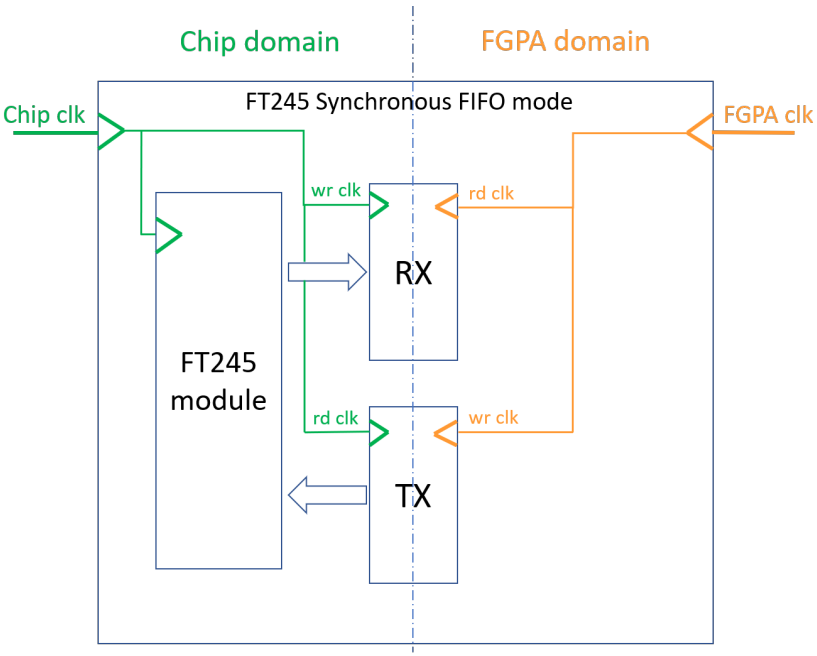


Figure 3.4: Clock domains.

side, there is the FPGA side, in which the clock could be anyone. The asynchronous FIFO acts as intermediate point for the data, separating the two clock domains.

For instance, looking at the RX FIFO (see Figure 3.4), it has its writing clock connected to the 60 MHz clock, i.e, when the RX FIFO has the write enable to 1, a byte of data is written in the RX asynchronous FIFO every rising edge of the clock. This makes things easier since the chip also sends a byte of data each clock period. Instead, the reading from the RX FIFO is performed following the clock of the FPGA side, i.e, the clock that runs the AXI4 Stream protocol.

The opposite occurs for the TX asynchronous FIFO: the writing clock is the one in the FPGA side (commonly 100 MHz) and the reading one is the 60 MHz clock.

In this way it is possible to communicate two different domains which use different clocks for their operation.

The basic functionality of the interface is the following: when the chip drives down the RXF signal, the state machine enters the reading mode. The bytes on the output of the FT2232H chip change each clock period (60 MHz) and they are sampled with the same frequency for being stored in the RX Asynchronous FIFO.

On the other side, when the the chip puts down TXE and the Synchronous protocol was in the waiting state, the interface enters the writing mode. The FT2232h chip reads a byte every clock period (60 MHz) the same frequency with which the TX Asynchronous

FIFO is putting out its data.

The FT245 interface comes back to the waiting state if it was in the reading or writing mode and RXF or TXE change from 0 to 1 respectively. It also happens when the FIFOs RX and TX are full or empty respectively.

Meanwhile, on the FPGA domain, the data is exchanged according to the AXI4 Stream protocol.

3.3 Vivado set up

Everything regarding the programming of the FPGA passes through the design tool Vivado. Using this software, the first thing to do is the block design: it allows to add new blocks (cores) to the design (apart from the FT245 Synchronous FIFO mode Ip core) in order to properly connect all the signals, characterize them and also to simplify the whole project before passing it to the FPGA. With this aim, some blocks were added to the design, as it can be seen in Figure 3.5:

- Clocking Wizard. There are two: one for the FPGA clock and other one for the clock of the chip. In this way, two different clocks are generated having as source the FPGA clock (from now on, it will be called *System clock*) and the FT2232H chip clock respectively. Thus, the delay introduced from the source clocks to the input ports where these signals are used is less than if the connection were done directly [22].

- Processor System Reset. It is for generating and customizing different types of reset. In this project, two types of reset are needed, active high and active low (for the AXI4 Stream) [23]. They are synchronized to the System clock and triggered by the reset button of the FPGA, which is an input of the processor system reset.

Then, after the validation of the block design, the synthesis is run.

Once this is done, it is time to insert the timing constraints. In a logic circuit, the signals do not propagate in zero time, as it would be ideally, it takes some time for them to change, starting from the signal that generates that change.

With this system it happens the same, as it is seen from Table 2.2, which is in correspondence with Figure 2.6 of the previous chapter, there are some delays in the signals and some hold and set up times that must be respected in order for the system to work properly. For instance, the temporal window in which the data going out of or coming in the chip can be sampled is restricted, not only by the chip timing, but also by the

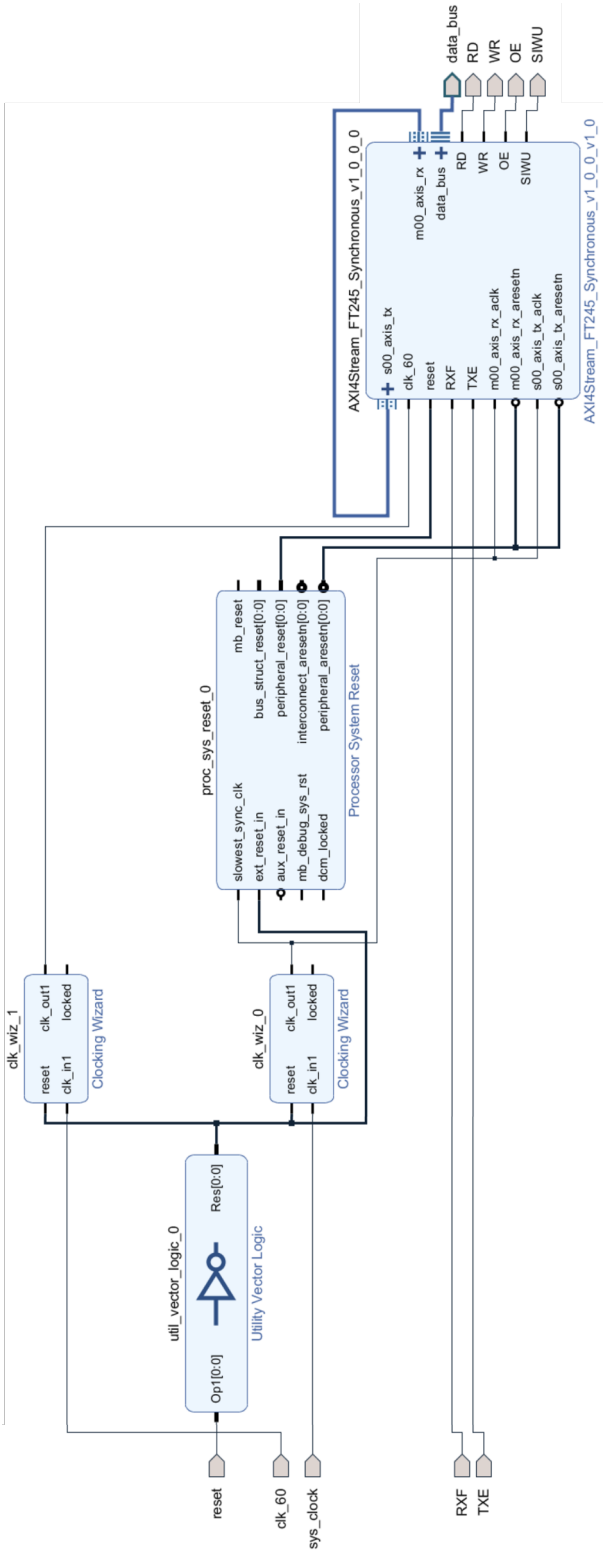


FIGURE 3.5: Block design.

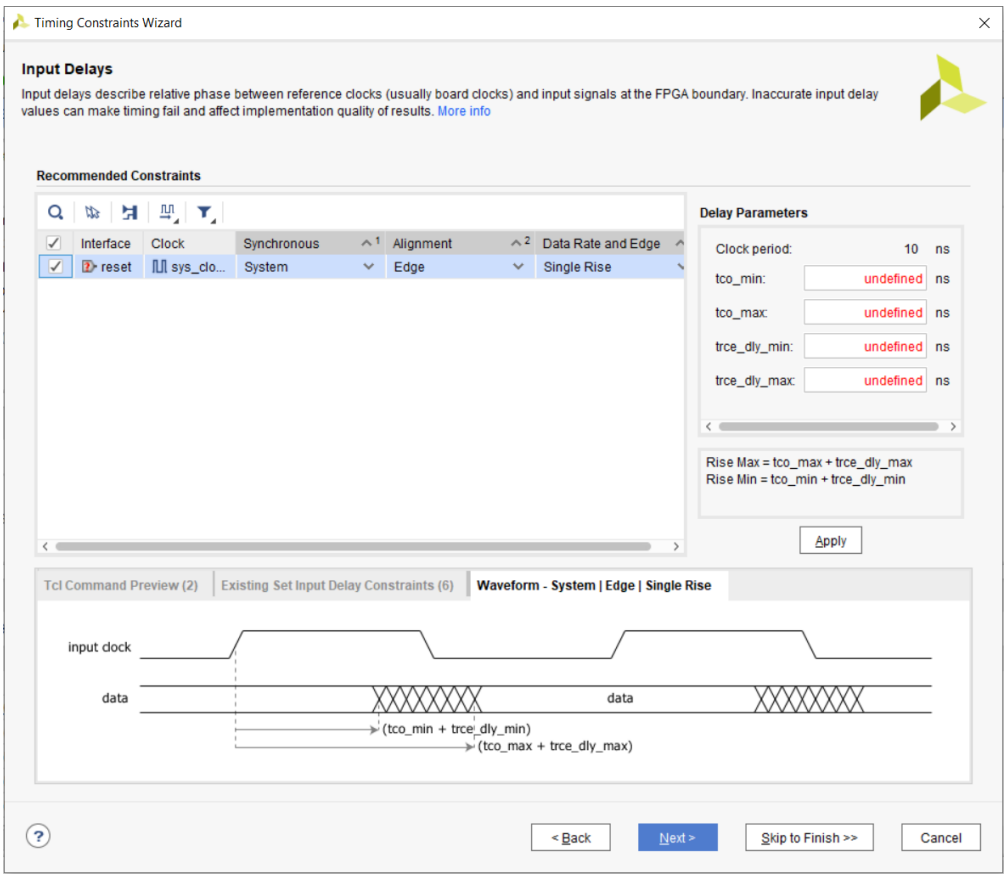


Figure 3.6: Constraints Wizard.

length of the traces in the board (the higher the length, the higher the delays because there is more space to be traveled).

All of this have been taken into account when programming the FPGA, either the restrictions of the datasheet of the chip or the delays introduced by the traces, in order to let Vivado manage the signals to acquire them when they are not changing.

For the trace delays, the material of the connections is the FR4, which has a propagation speed of around 15 cm/ns [24]. The length of the traces of the board used has not been found, so it has been supposed a delay of 0.5 ns for each trace, that would correspond to a length of 7.5 cm, which is plausible (of the order of magnitude of the dimensions of the board).

In Vivado, each timing constraint is introduced as an input or output delay using a special feature called "Constraints Wizard", from the synthesized design, which is shown in Figure 3.6 as example of how the input delay is included. This has been done for every external signal of the system exchanged with the FT2232H chip.

The tco_min and tco_max are the ones specified in Figure 2.2 and trce_dly_min

and trce_dly_max refer to the delay introduced by the traces in the board.

The next step after setting the timing constraints is running the implementation. Looking at the block design, there are some signals that start or finish in grey targets. In the implemented design, these signals are assigned to their corresponding pins in the board, they are signals that come from or end in the board, either from/to the chip or from/to the FPGA.

Lastly, the bitstream is generated and it is charged in the FPGA. At this point, everything is ready to run the program in C and test the FT245 Synchronous FIFO interface.

3.4 Connections

To test it, the Nexys Video is connected to the computer using two wires.

- One for programming and debugging the interface.
- One for exchanging data with the computer.

Both cables are necessary for carrying out the two tasks. It is not possible to see the debug signals only with a wire if at the same time the chip is exchanging information with the computer.

4 Testing the Interface

At first, the preliminary tests to check the functionality of the interface were done using a loop-back configuration. The loop-back consists in connecting the RX and TX modules together rather than to the FPGA (they form a pair master-slave), so all the information that is sent from the PC goes back and is read by the computer after traveling along the whole data path (round trip). Testing the module with this configuration helps to identify if there are some errors in the code, because all the data sent must return.

After debugging the code and all the information sent was recovered successfully, another problem was found: the speed. Knowing that the FT245 Synchronous FIFO mode is driven by a 60 MHz clock, the theoretical maximum velocity achievable would be 60 Mbyte/s (in each rising edge of the clock a byte is sent). Thus, it was to be hoped at least a speed of some tens of Mbyte/s. However, when measuring (in the PC side) the time needed for the data to travel along the loop, the speed obtained was much slower than expected (of tenths of Mbyte/s).

4.1 Initial tests

In order to find out what was exactly happening, the subsequent proves were performed:

1. Sending data to the FPGA.
2. Receiving data from the FPGA.
3. Loop-back.

4.1.1 Sending data to the FPGA

When data was sent to the FPGA while the **chip** was **also receiving** from it, the speed was calculated either looking at the waveforms obtained with Vivado or directly inside the C program, measuring the time before and after calling the writing function.

Figure 4.1: Waveforms when sending data to the FPGA without receiving.

The program was done by means of a loop in which it executes the function FT_write, the user selects the value of the magnitude of the vector written in the chip. For caution issues, for the moment, a maximum of 4096 bytes was sent in each iteration of the loop. This is due to the fact that 4096 bytes is less than the maximum capacity of the FIFOs involved in the data path (2048 bytes belong to the RX FIFO and 4096 to the receiving FIFO of the FT2232H chip [13]). The signal Tready, input of the master module RX, was set to 1 in order for the FPGA to accept all the information coming from the PC.

For looking at the waveforms, the debugger of Vivado was activated and triggered when the chip started to send data to the FPGA. From that instant on, the waveforms of some signals are shown during a time interval. In this way, it is possible to see the different signals changing in parallel to the running C program, but only for a limited time window.

By means of the waveforms, the data rate that was obtained reached near 30 Mbyte/s. It was calculated by looking at the time window showed with the debug tool: the data sent during a time period divided by that time period. This test was done sending 4096, 2048 and 1024 bytes respectively with the C program and without introducing iterations in the loop. On the other side, by using the program, but with 2 iterations of 4096 bytes to send 8192 bytes, the resulting speed was 23 Mbyte/s and, in case of sending 65536 bytes in pieces of 4096 bytes, around 20 Mbyte/s. This means that some speed is lost between iterations.

So, the higher the number of iterations of the loop, the lower the speed.

Lastly, by only transferring data to the FPGA and **without the chip receiving** information from it, the speed reached was 32 Mbyte/s. A little bit higher than before, because, in this case, the chip was not alternating its resources between reading and

writing, it only wrote.

In Figure 4.1 the waveforms corresponding to the last test are shown. The image is divided into:

- The top part: it presents the behavior of the master of the AXI4 Stream protocol RX. It is driven by the System clock (100 MHz), so the time between two consecutive samples of the debug is 10 ns. The relevant signals are in the third and sixth rows respectively: Tvalid and Tdata (which coincides with the output of the RX FIFO).

- The bottom part: it follows the 60 MHz clock of the chip (samples of the debug every 16,67 ns) and represents some of the signals exchanged between the FT2232H chip and the interface implemented in this thesis. Those are, in order: RD, the data input of the RX FIFO (the data coming from the chip), the flag telling if RX is full, the states in the state machine, WR and the output of the TX FIFO (data going in the chip). As the chip was not receiving data, only sending, WR remains high and the output of TX does not change. Oppositely, RD is continuously changing. When it is low, a reading operation is being performed in the chip and the RX FIFO is storing data.

It is worth pointing out that when the state variable is 00, the module is in the waiting state and that there is a correspondence between the top and the bottom images: when state is 00, the master stops sending data to the FPGA because there is no more data to be sent.

4.1.2 Receiving data from the FPGA

For this test, the direction FPGA - PC was tested: from the PC no data was sent, so the FPGA was not receiving. Conversely, in the FPGA side a counter was attached to the TX slave module and the Tvalid of the slave was set to 1, to continuously send information from the FPGA to the PC.

To add the counter, a new block in the Vivado block design was introduced, as seen from Figure 4.2. This block was done by means of a VHDL file that behaves like a master following the AXI4 Stream protocol, it counts from 0 to 255 and starts again.

Thus, the program in C was reading data from the chip, that came from the counter, by calling the function FT_read. At first, it was difficult to measure the time with the program because when reading, the chip had already some data that came from the

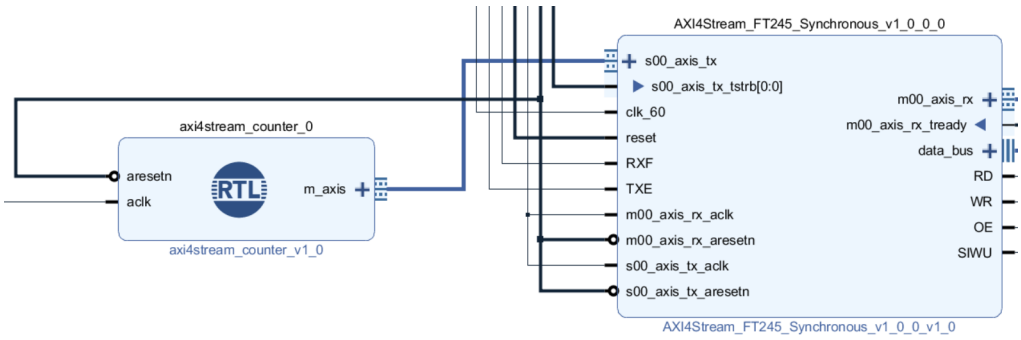


Figure 4.2: Block design to test the path FPGA-PC.

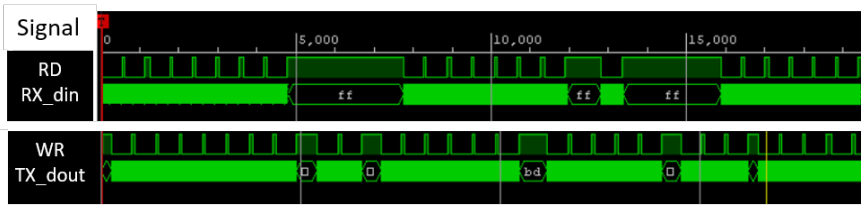


Figure 4.3: Comparison between sending (top) and receiving data (bottom).

chip. Thus, an estimation of the speed was done by looking at the waveforms, obtaining around 45 Mbyte/s. The same packages as in the point before were used (4096, 2048 and 1024 bytes respectively).

Comparing this speed (45 Mbyte/s) with the previous one (32 Mbyte/s), by symmetry, they should be equal. However, by looking at the waveforms, in the first case the chip inserted less waiting states (Figure 4.3). In the top part of the picture, the signal RD remains more time high than WR in the bottom part. This means that the chip spent more time when writing in the FPGA than reading from it.

4.1.3 Loop-back

Lastly, the loop-back, with which the lowest speed was obtained, was analyzed in order to discover why it was so slow. The program in C used consisted of a loop in which the write and read operations were performed in each iteration of, maximum, 4096 bytes. In each iteration of the loop, FT_write was called in first place and then, before executing FT_read, the program waited until the bytes ready to be read were the same as the bytes that were written.

The waveforms showed that the chip stopped to transmit or receive data between iterations (Figure 4.4): the debugger was triggered with the signal RD, signalizing the beginning of the transmission, and it can be seen that the transfer ends before the time

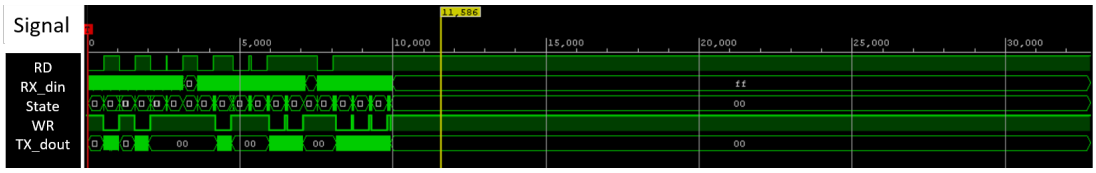


Figure 4.4: First test of the loop-back.

window finishes. Figure 4.4 is qualitative, the important thing to note from it is that the data exchanged corresponds only to one iteration of the loop: each time RD or WR are low, the FPGA reads or writes 512 bytes into the chip respectively (this is because in the FT245 Synchronous FIFO mode, the maximum USB 2.0 packet size is 512 bytes [25]). This happens 8 times for each signal (RD and WR), so the amount read and written is 4096 bytes in the image, corresponding to one iteration of the loop, then, the waveforms stop moving.

Thus, this shows that there was no continuity between iterations. The speed, calculated regarding the waveforms in the period in which the chip was exchanging data was 25 – 26 Mbyte/s.

4.1.4 Conclusions

To conclude this section, it is worth pointing out that the main purpose of these tests was checking the speed, controlling that the order of magnitude was acceptable. With this priority, these proves may not be rigorous in other aspects, as it was a first order approach, but they served, at least, to confirm that the speed with which the chip exchanges data via the FT245 Synchronous FIFO mode was in an plausible range.

In the next sections, a deeper analysis of the module will be performed, clarifying some issues that were not solved with this preliminary tests.

4.2 Exchanging 5 Gbytes

At this point, that the interface seems to be working properly and the velocity achieved is of the order of magnitude expected, it is time to check the FT245 Synchronous FIFO mode with a huge stream of information to, first of all, calculate the speed and, second, to verify that every single byte is transmitted successfully.

To this aim, the same configurations as in Section 4.1 were analyzed and the corresponding tests and results found are exposed in the subsequent points.

4.2.1 Receiving 5 Gbytes from the FPGA

The first test done to the interface was sending 5 Gbytes ($5{\cdot}2^{30}$) from the FPGA to the PC. There is information traveling only in one direction: the data goes from the FPGA to the PC, the PC does **not** write in the chip during the whole process. When the data arrives to the PC it is verified if all the bytes are not mistaken. The test was configured in two parts:

1. The **FPGA side**. As can be seen from Figure 4.5 some changes were introduced with respect to Figure 3.5 of the previous chapter in order to adapt the block design to the new functionality:
   - A new block was inserted, it is a master interface of the AXI4 Stream protocol connected to the TX slave of the FT245 Synchronous FIFO mode. It has an 8 bits counter inside it: the data sent from the FPGA counts from 0 to 255 indefinitely. The Tvalid of this master is set to 1 to continuously send data to the PC.
   - In the master side RX nothing was changed because for not receiving data the only thing to do is not sending it from the PC.
   - There is also another block introduced that does not influence the functionality of the design, the one called System ILA, which is only an interface for debugging and being able to see the signals of the master module RX. This may not be useful in this point, but for the next one, it will.
2. The **PC side**. The program in C is done by means of a loop which calls the function FT_read in each iteration until reaching the 5 Gbytes of the experiment. It was found that the maximum number of bytes that the functions FT_write and FT_read can handle is 1048576 and 65536 each [25], so the highest number of bytes read when calling FT_read will be 65536 (64k). Coming back to the loop, before reading, the function FT_Getstatus is executed to know how many bytes are in the receiving queue and then, that number of bytes is read from the chip and its quantity stored in a variable, which in each iteration increments itself in the number of bytes that have been read until reaching or overcoming the value of 5 Gbytes. When this happens, the loop ends. For checking that all the bytes are received, the first value that arrives is picked up and by adding 1, each new byte coming in is verified. If the value 255 is achieved, the count for comparing the incoming bytes with starts again from 0.

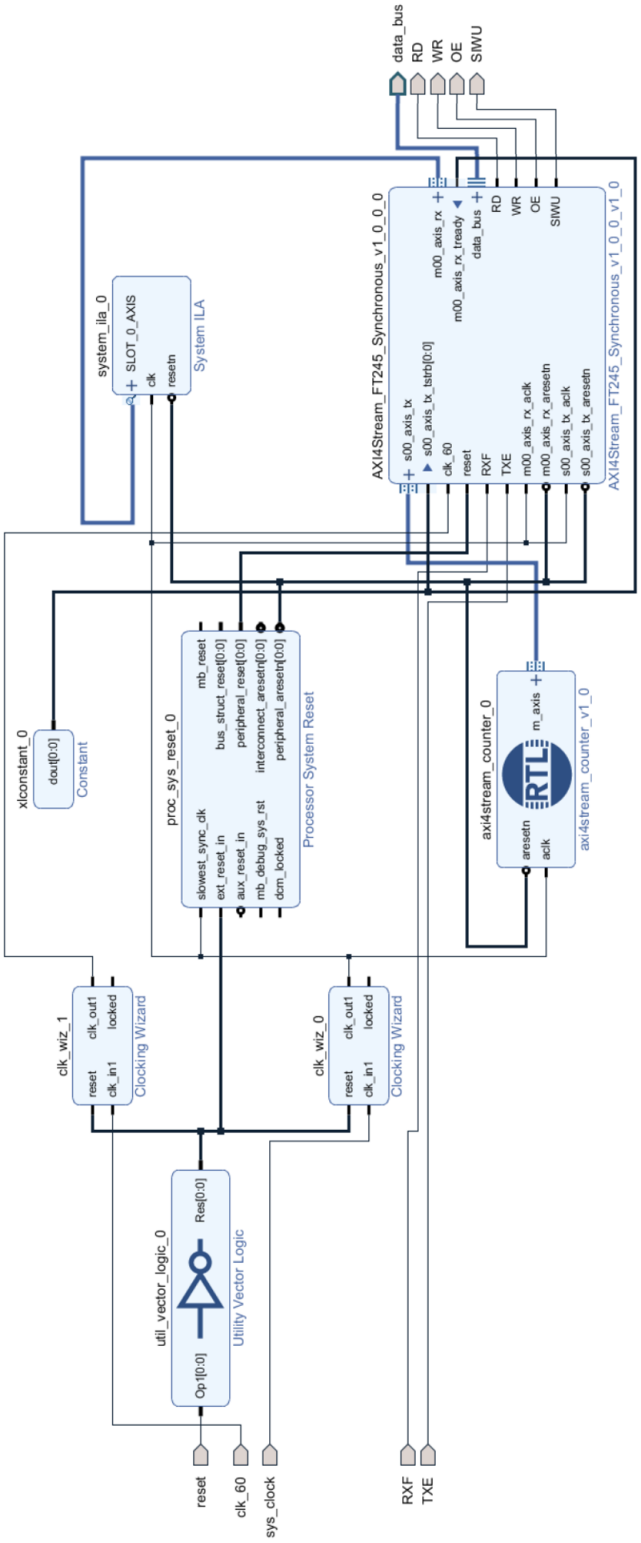


FIGURE 4.5: Receiving 5 Gbytes, block design.

Once this was done, the program in C was executed several times to prove that it worked properly. No wrong bytes were found and the speed was around 44 - 45 Mbyte/s. It also was done limiting the maximum number of bytes read each time, but no differences were obtained. The time spent in running the loop was not slower than the time the interface needs to have bytes to send to the computer.

The last thing to comment is that the time to calculate the velocity was measured in two different moments:

- Right after the first batch of bytes was read (this first packet is not added to the number of bytes that have been read). This is because when the program starts, as the FPGA is continuously sending data, the chip is also receiving, therefore, when running the program, there are already some bytes ready to be sent to the computer.

- When exiting the loop.

Then, the speed was calculated as the number of bytes read divided by the difference of the two times measured.

4.2.2 Sending 5 Gbytes from the PC to the FPGA

The second prove to which the module was subjected was sending 5 Gbytes ($5 \cdot 2^{30}$) from the PC to the FPGA. For this test, the flux of data was unidirectional: the information went from the PC to the FPGA, the FPGA was **not** transmitting at the same time. In addition, in the FPGA side it was checked that all the bytes were sent correctly.

The details of the experiment are exposed below, following the sequence of the data path:

1. The **program in C**. It consists in a loop calling the function FT_write in a way such that the number of iterations multiplied by the number of bytes written is 5 Gbytes. If one of the batches sent is not correctly written in the chip, an error will appear. The bytes written count from 0 to 255 indefinitely, until reaching the amount desired of bytes to send (5 Gbytes). The speed of the exchange of data is measured here: the time is picked up just before the beginning of the transmission and after sending the last byte using the function *gettimeofday*. The number of bytes sent is then divided by the difference of these two times to calculate the velocity.

Bytes per iteration	Iterations	Speed (Mbyte/s)
4096	1310720	23
8192	655360	28
65536	81920	36
1048576	5120	~ 40 (the max was 45)

TABLE 4.1: Send 5 Gbytes.

2. In the **FPGA side** two changes were applied:

   - A new module attached to the RX master of the FT245 Synchronous FIFO mode was introduced in the block design of Vivado, as shown in Figure 4.6. It serves to check that all the bytes that arrive from the PC are the ones expected, to verify that the interface developed in this thesis manages them successfully according with the requirements. With this purpose, the module is an 8 bits counter (from 0 to 255) which increments the count every clock pulse only if the Tvalid of the RX is asserted to 1 (if there is data coming in). The Tready of the counter is set to 1 to accept every incoming data and there is also a flag that changes from 0 to 1 if one byte does not match (if the byte just arrived from the computer is different from the counter value). So, if the flag changes to 1 means that there is a byte wrongly sent.
   - The signal Tvalid of the counter added in the previous point 4.2.1 was set to 0 to avoid the stream of data from the FPGA to the PC, i.e., to disable that first counter already included.

With this design in the FPGA, the program in C was run setting different number of bytes for each iteration and seeing what happened to the speed. Meanwhile, the debug of Vivado was opened using as trigger the flag for detecting wrong bytes. All the proves finished with the flag at 0, without triggering the debug, meaning that all the bytes were transmitted correctly.

The results for the speed found are reported in Table 4.1:

As mentioned in the previous Subsection 4.2.1, the limits for the FT_write and FT_read functions are 1048576 and 65536 bytes respectively [25], that is why the last value in the table is 1048576 bytes per iteration.

Apart from that, it is worth pointing out that the speed obtained with this configuration is not perfectly accurate: when measuring the time the second occasion (end of the transfer), it is collected right after the last byte is sent, while it should be picked up

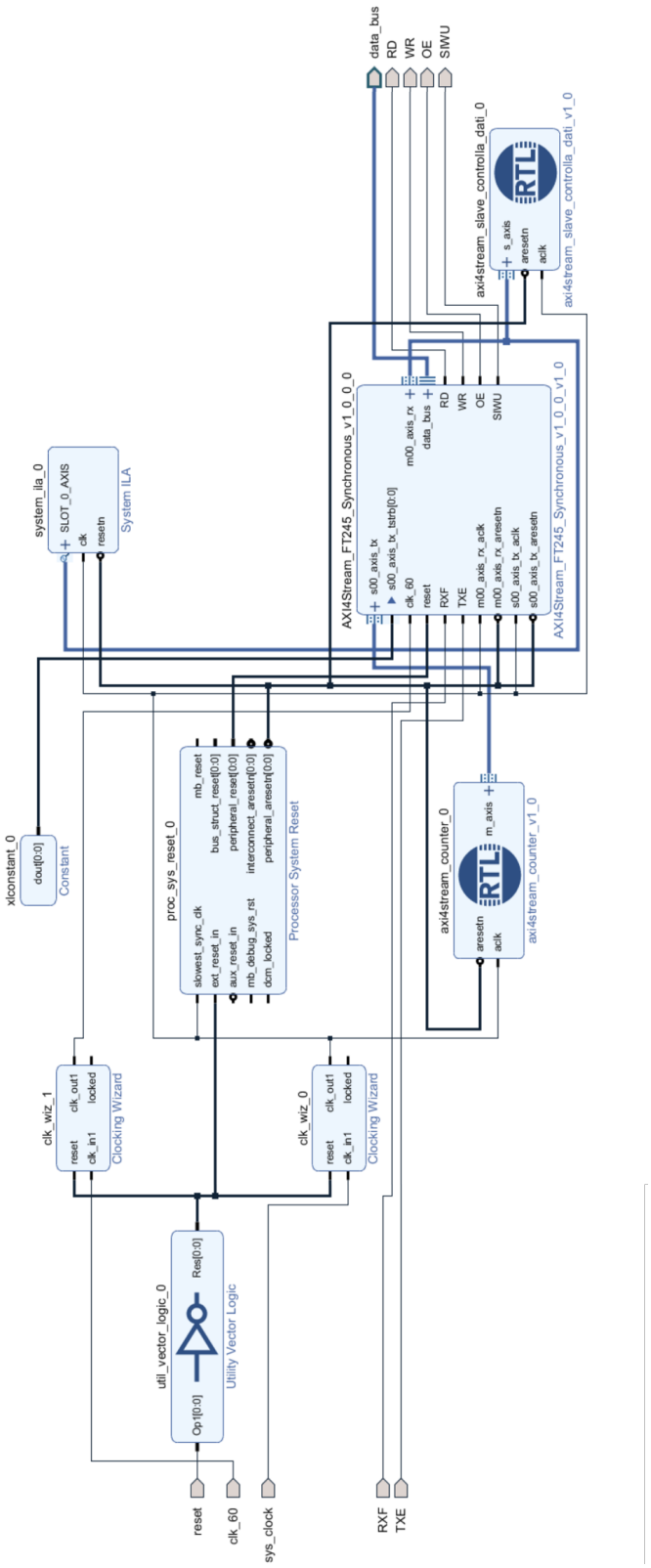
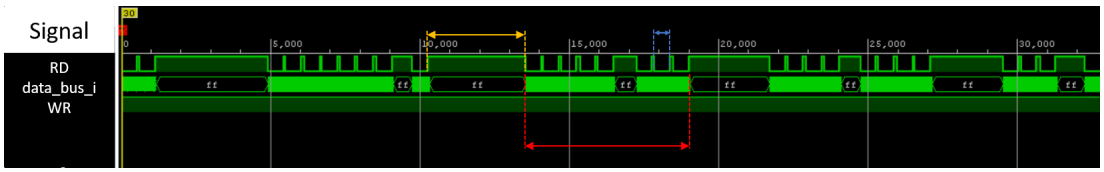
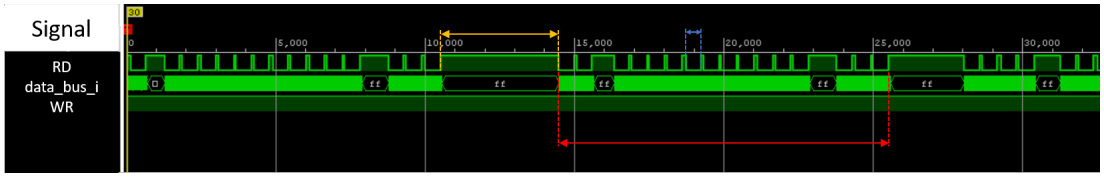


FIGURE 4.6: Sending 5 Gbytes, block design.



a) 4096 bytes per iteration.



b) 8192 bytes per iteration.

FIGURE 4.7: Waveforms with different bytes per iteration.

after the last byte enters the FPGA. That is the reason of sending such a big amount of information, to minimize the impact of that time that is not taken into account.

To conclude, it follows from the table that the higher the speed, the lower the number of iterations, so it seems that there is some time lost due to the execution of the loop, as Figure 4.7 suggests. It shows the waveforms when the bytes per iteration are 4096 and 8192 respectively. The arrows in different colors mark the relevant time periods:

- Blue arrow: during this time the chip sends 512 bytes [25] to the FPGA.

- Red arrow: it is the time of an iteration of the loop in the C program. The blue arrow is 512 bytes and within the red arrow it is possible to count 8 periods of blue arrows in the top part of the image, therefore, the total amount of bytes sent during the red arrow is $8 \cdot 512 = 4096$, corresponding to one iteration of the loop. The same applies for the second half of the picture, where there are 16 periods of 512 bytes.

- Yellow arrow: it is the time between iterations, because it separates amounts of 4096 and 8192 bytes respectively. During this period no operations of reading and writing are performed, thus, the module programmed is in the waiting state. This can be due to the fact that the chip is communicating with the computer (it follows the dynamics of the loop).

This issue will be deeply explained in Section 4.3.

Bytes per iteration	Speed (Mbyte/s)
4096	0.3
16 384	1
65 536	3 - 4
262 144	10
1 0480 576	17 - 18

Table 4.2: Loop-back 5 Gbytes.

4.2.3 Loop-back

The last configuration tested with 5 Gbytes was the loop-back layout, as seen from Figure 4.8. It connects together the RX master and the TX slave interfaces of the FT245 Synchronous FIFO mode, so all the information that enters the FPGA comes back to the PC.

- The **program in C** consists, again, of a loop: first of all, the FT_write function is called and a number of bytes are written in the chip. Afterwards, FT_read is executed until the all bytes previously sent have been read (as always done until now, FT_Getstatus was called before FT_read to know the amount of bytes that are ready to enter in the computer). Then, the loop restarts again. The time for calculating the speed is measured right before calling FT_write for the first time and after reading the last package sent. Lastly, for verifying that there are no wrong bytes recovered, after each iteration, the bytes received are compared with those sent, when one does not match, a variable is incremented by one.

- From the **FPGA side**, all the blocks added in the points before to the block design were removed and the pair RX master and TX slave were connected as said above.

The software was executed several times without mistaken any byte sending a random stream of data, writing batches of different sizes per iteration. The results are reported in Table 4.2.

One more time, it happens that the higher the number of iterations, the slower the transfer of data. The maximum speed obtained with this configuration was around 17 - 18 Mbyte/s.

4.2.4 Conclusions

Regarding the results obtained for the speed, it seems that there is a dependence of the velocity on the number of times the loop in C was executed, as the amount of bytes

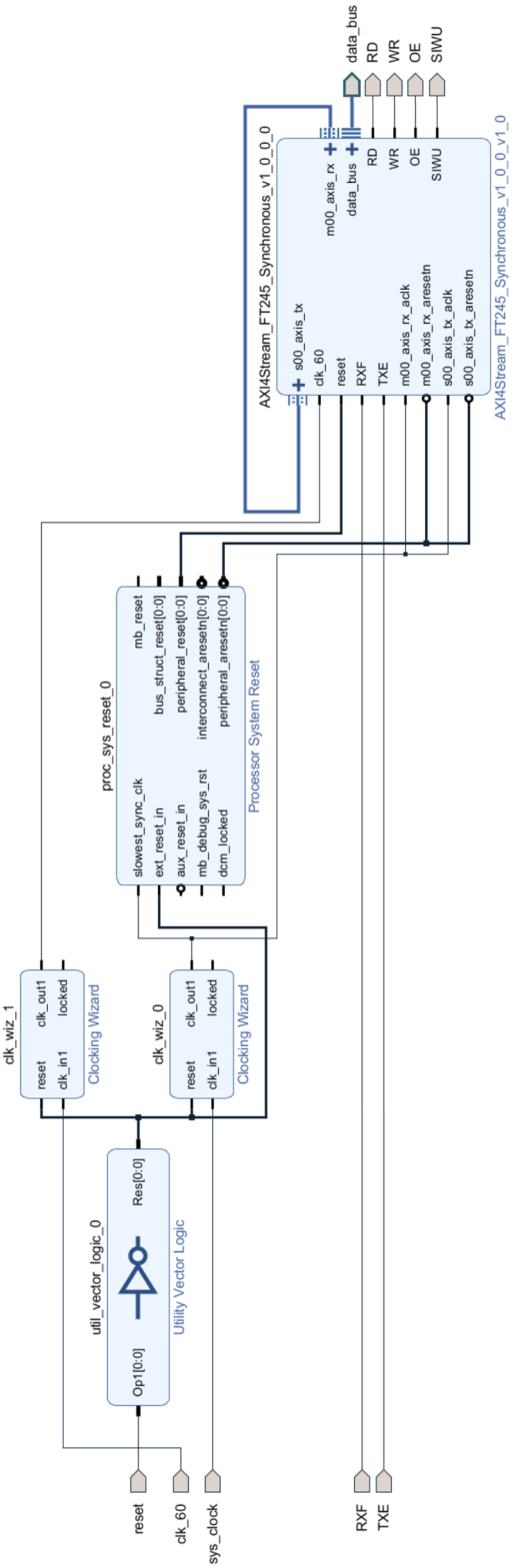


Figure 4.8: Loop-back at 5 Gbytes.

sent is always the same. The only case in which this does not happen is when receiving data. Here, the limiting factor is the chip providing the data to the computer, which goes slower than the loop asking for new bytes.

By looking at the numbers, the maximum speed for the loop-back configuration should be half of the average of the receiving and sending velocities, which is:

$$\frac{(44+44)}{2} \cdot \frac{1}{2} = 22\ Mbyte/s$$

If for the calculation, instead of using the maximum sending speed the average is employed, the loop-back speed should be around 20 - 21 Mbyte/s, but actually it is only 17 - 18 Mbyte/s, so there is something lost.

4.3 Reaching the maximum speed

Apart from the fact that considering the numbers the loop-back speed should be higher, another question arises: Why is the speed limited? Why is it not closer to 60 MHz or, in case of the loop-back, to 30 MHz? Because in each clock pulse a byte is transmitted. The explanation has to do with, on the one hand, how the chip manages the reading and writing operations and, on the other hand, the time lost due to iterations in the loop in C. The effect of these two factors is explained below.

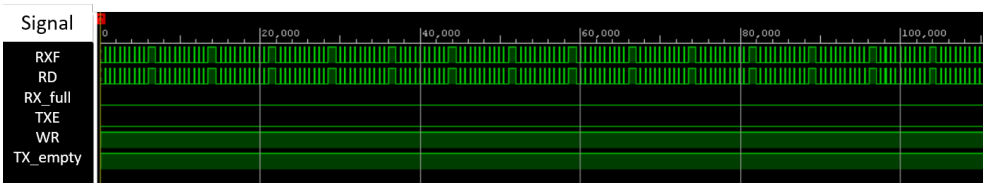
4.3.1 Sending data to the FPGA

The first test was done for the data path PC - FPGA using the scheme in Figure 4.6 and the same C program that in Section 4.2.2. The way to proceed was looking at the waveforms achieved with the debugging tool in order to see some sort of correlation between them and the speed obtained for different number of iterations.

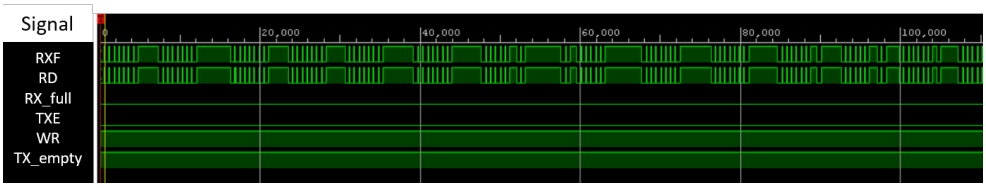
In the previous point, this issue was briefly introduced, but Figure 4.9 clearly shows a correspondence between the speed and the number of iterations in the program in C executed by the computer.

The signals that appear in the graphics (Figure 4.9) are divided in two:

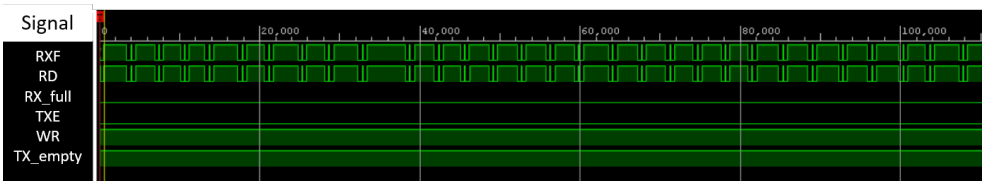
- The first three correspond to the reading operation, when the data is read from the chip by the FPGA. They are RXF, provided by the chip; RD, provided by the FT245 Synchronous FIFO interface implemented; and RX_full, which tells if the receiving FIFO is full (1) or not (0).



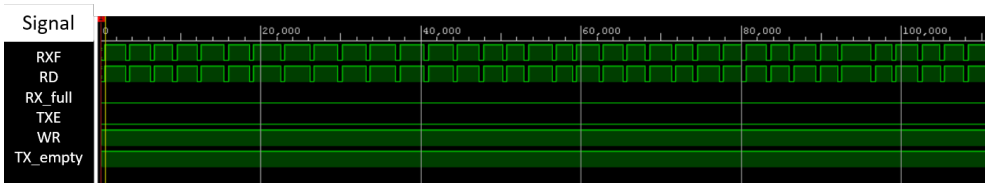
a) 1048576 bytes



b) 4096 bytes



c) 1024 bytes



d) 512 bytes

Figure 4.9: Speed vs iterations when sending data to the FPGA.

- The last three signals are related to the writing operation, when the data is written in the chip. They are TXE, output of the chip; WR, input of the chip and output of the FT245 Synchronous FIFO mode; and TX_empty, which is asserted to 1 when the transmitting FIFO is empty.

These signals have been selected because the issue under study is when the chip stops exchanging information and why, making the speed to get worse. For instance, RD gives information about whether the chip is performing a reading operation or not: when it is low, a reading operation is being executed. Although RD already indicates if the chip is in the reading mode, RXF and RX_full where also included because RD depends on them: if the reading process stops it is because either RXF is high (the chip is not available to send information) or RX_full is one (no more space in the FIFO to receive new data). The same reasoning applies to the writing mode.

From the Image 4.9, it follows that the fact that the exchange of data was slower when incrementing the iterations in the loop of the program in C is due to the chip, which drives RXF high more often. This can be because the chip needs to communicate more frequently with the computer and wait for it to send the corresponding data.

In Figure 4.9 a), the stop of the chip for communicating/waiting for the computer does not appear because the batch sent was so high. However, in the other three pictures of the figure, that pause can be seen clearly, especially in the last two images: as said in the previous section, during the time span that RD remains low, the chip sends 512 bytes [25]. Therefore, in c) and d) it is shown that between two big pauses, 1024 and 512 bytes are transmitted respectively.

When running the C program to obtain the waveform in a), the maximum speed achieved was between 44 and 45 Mbyte/s and by making calculations with the information of the waveforms, the resulting value was the same. Thus, in this case, the lost of speed is due to those intermediate gaps every 512 bytes sent and to the frequency with which the computer sends new data.

4.3.2 Receiving data to the FPGA

In this case there were no problems due to the iterations of the program in C, but the speed achieved, around 44 Mbyte/s, is not close to 60 Mbyte/s.

The speed calculated regarding the waveforms in Figure 4.10 is also around 44 Mbyte/s, corroborating the result and indicating that the loss is due to those waiting states introduced by the chip.

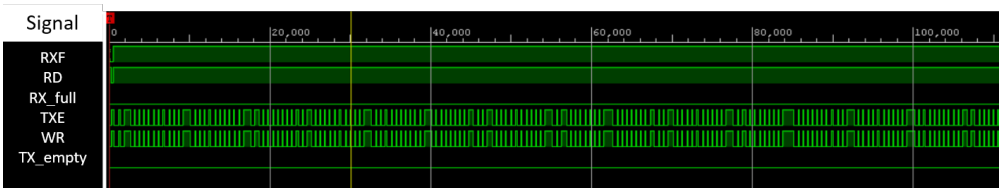


Figure 4.10: Receiving data.

4.3.3 Loop-back

For analyzing the loop-back, the same signals concerning the reading and writing mode were evaluated. The same program in C and design of Vivado as in Section 4.2.3 were used.

The first step was to look at the waveforms to find the pauses due to the iterations in the loop. Two different situations were found during the execution of the program:

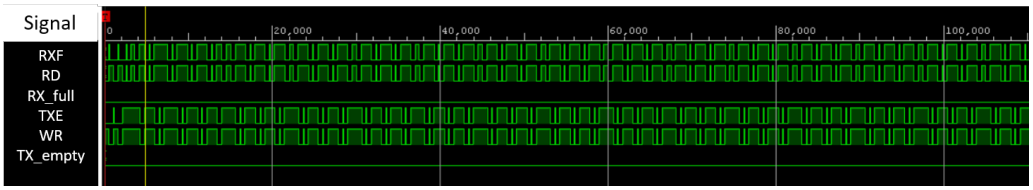
- Time periods in which the chip was alternating between the reading and writing operations.
- Long time spans in which the chip does not do anything: the signal TXE is low (the chip is ready to receive data) but the transmitting FIFO is empty. In the meanwhile, RXE is high (the chip does not have data to send).

This fact is illustrated in Figure 4.11: the chip is exchanging data with the FPGA in both directions, then, the reading operation ends first, driving RXF up to indicate that the chip does not have more data to send. After that, the writing operation continues for a while, until the transmission FIFO is empty, in that moment, the data flow stops. During this cut of transmission, the RXF is 1, so there is no more information to be sent to the FPGA (the computer has not share the next batch of data yet). On the other side, TXE is low and the TX FIFO empty meaning that the chip can receive data from the FPGA, but the FPGA has no bytes to provide.

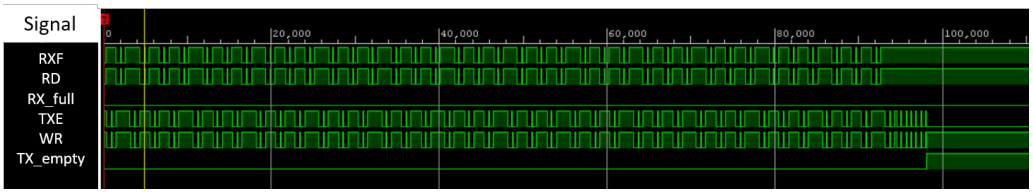
Viewing Figure 4.11 and already knowing that the communication with the computer makes the interface to lose some time, the first hypothesis is that this pauses have to do with how the program in C works.

The way to proceed was the following:

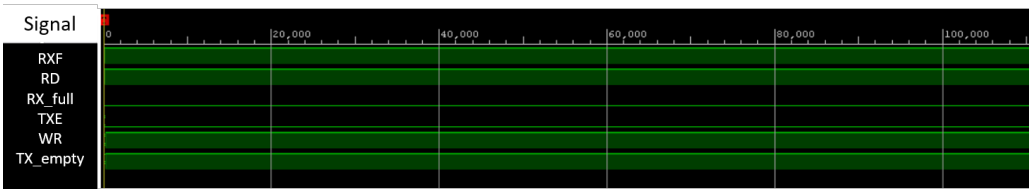
1. Calculate the speed with the data shown in the waveforms to see if it differs from the one obtained in Subsection 4.2.3.
2. Estimate the time lost in each iteration if possible.



a) Read and write



b) Read, write and pause



c) Pause

Figure 4.11: Loop-back: various waveforms while running the program.

Bytes per iteration	Iterations	Time obtained (ms)	Time estimated (ms)
1 048 576	1	60	-
524 288	2	75	-
262 144	4	104	105
65 536	16	288	285

TABLE 4.3: Time between iterations.

For the **point number 1**, the time window used was the one where the the chip is alternating its functions between the reading and writing. The calculated speed was 24,5 Mbyte/s. The pause was not considered because the point here is to estimate what would be the maximum speed achievable (the chip exchanging data continuously). This speed is somehow far from the 17 - 18 Mbyte/s obtained before.

It seems the problem is due that big waste of time, thus, the **second step** would be to estimate it and find an explanation.

As first, it was supposed that the time between iterations would be the same, independently of the amount of data sent in each of them, regarding the results obtained in Section 4.2.3. For the experiment, a fix quantity of information was exchanged (1 Mbyte) but using different number of iterations in the program in C (1, 2, 4 and 16 iterations respectively).

In Table 4.3, the time measured with the program in C when sending 1 Mbyte and the estimated one based on the two first rows are exposed. The extra time was calculated as follows:

The difference between one and two iterations will be the time wasted per iteration, because the amount of data managed is the same and it has been supposed that the paused time is independent of how much information is sent in each iteration. So,

$$t = 75 - 60 = 15\,ms$$

With this number, the time running the loop with 4 will be,

$$t_{4iterations} = 60 + 15 * 3 = 105\,ms$$

And for 16 iterations of 65 536 each,

$$t_{16iterations} = 60 + 15 * 15 = 285\,ms$$

Regarding the results, it seems that the approximation done was right. Moreover, with this data, the speed of the loop-back was evaluated supposing that there were no waiting states, i.e., subtracting to the time of one iteration the time per iteration:

$$t_{effective} = 60 - 15 = 45\,ms$$

Therefore, the speed removing those pauses would be the number of bytes sent divided by the time taken:

$$v = \frac{1048576}{45\,ms} = 23,3\ Mbyte/s$$

Which is close to the 24,5 Mbyte/s obtained when evaluating the waveforms in Figure 4.11.

However, another approach was provided achieving a similar result.

This second approach takes the speed calculated by looking at the waveforms (around 24 Mbyte/s) and the one obtained in the previous Section (17 - 18 Mbyte/s) to evaluate how much would be the time lost due to each iteration whether all the time wasted were because of that.

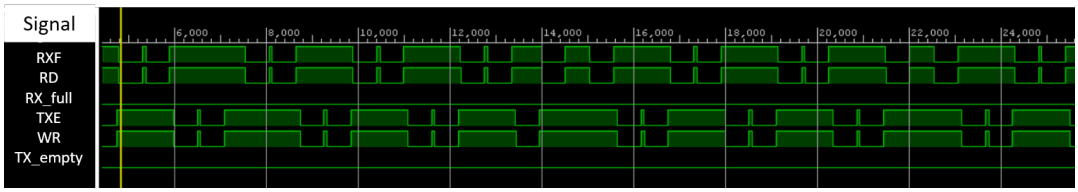
This calculation was performed when sending 5 Gbytes, with a time of 306 s (corresponding to 17,5 Mbyte) and 5120 iterations of the loop. The total time wasted T is:

$$\frac{5\,Gbyte}{306 - T} = 24,5\ Mbyte/s$$

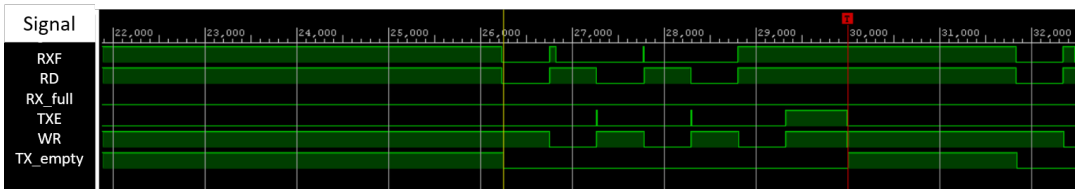
Then, the time for each iteration (T/5120) would be around 16,9 ms, which is close to the approximately 15 ms got above.

The **conclusion** of all these estimations and calculations is that there are two main losses of speed which make it differ from 30 Mbyte/s:

- From 30 Mbyte/s to around 24 Mbyte/s: even though the chip, in its effective times, is alternating between reading and writing, as shown in Figure 4.11, it is not perfect and there are some overlaps of RD and WR (both high) in which the chip is not reading neither writing (Figure 4.12 a). Furthermore, there is also some waste of time when either the RX FIFO is full or the TX FIFO is empty, as illustrated in Figure 4.12 b: TX is empty, but TXE continues low for a moment while RXF remains high.



a) Alternating between reading and writing.



b) TX FIFO empty.

FIGURE 4.12: Details of the loop-back waveform.

- From 24 Mbyte/s to 17/18 Mbyte/s: it is due to those wasted times between iterations.

At this point, the next thing to do was trying to reduce the pauses between iterations to speed up the whole interface. To this aim, the program in C was modified.

The idea was to read and write in the chip in a more dynamic way: instead of writing a big amount of data (1 Mbyte), then reading it and repeating the same operation, the plan was to write and read without waiting for FT_read to read all the bytes previously sent. So, the new program consisted in a loop which first calls FT_write, then FT_Getstatus and, finally, FT_read, which reads the bytes the chip has ready to send back to the PC. Then, the loop starts again. This continues until the bytes recovered reach 5 Gbytes or more.

Knowing that the maximum batch that the FT_read can read at a time is 65 536 and that it is actually what FT_read almost read in each iteration, the bytes to send with FT_write were set to 65 536 for having some symmetry.

In this way, the pauses were reduced as can be seen from Figure 4.13, and therefore, the speed increased to around 21 - 22 Mbyte/s, even reaching 23 Mbyte/s sometimes. Theses values correspond now to the average of sending and receiving divided by two.

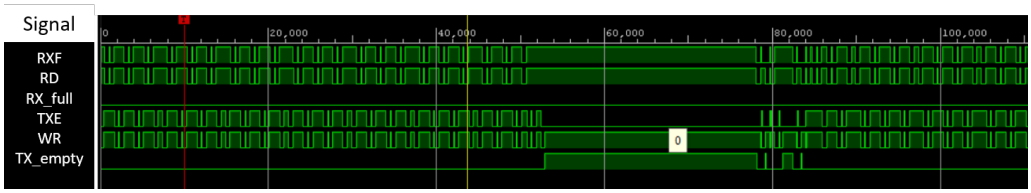


FIGURE 4.13: Increasing the speed.

Conclusions

In this thesis, an overview of all bus typologies was introduced from a theoretical point of view. In particular, the FT245 Synchronous FIFO protocol was developed for a USB connection between the PC and the FPGA. The communication path is constituted of the PC, the USB connection, the FT2232H chip, the interface done and the FPGA.

For implementing the mode, it was divided in three blocks: the FT245 module, which exchanges data with the chip and the FPGA and the RX and TX FIFOs, which are the receiving and transmitter FIFOs of the FPGA respectively.

The interface was, then, programmed and transferred to the FPGA for testing it. The maximum ideal expected speed would be 60 Mbyte/s in each direction and 30 Mbyte/s in the loop-back configuration because the clock with which the chip exchanges data with the FPGA is 60 MHz (a byte of information is sent every clock pulse).

The first results showed an important lack of speed when using the loop-back layout, however, after a second deeper analysis, it could be seen that the the speed was of an acceptable order of magnitude (tens of Mbyte/s).

Then, the interface was tested in both directions and with the loop-back configuration by exchanging 5 Gbyte of data for checking the velocity and verifying that every single byte was sent correctly. The results obtained were satisfactory, achieving speeds of 44 - 45 Mbyte/s for the direction FPGA-PC, around 40 Mbyte/s for the PC-FPGA way and 17 - 18 Mbyte/s for the loop-back, which was finally increased to 21 - 22 Mbyte/s after including some changes in the program in C with which the data was sent from the computer.

Furthermore, the difference between the maximum reachable speed and the obtained one is due to how the chip manages the signals of the reading and writing operations (sometimes there are overlap between signals in which the chip is not reading neither writing) and to the iterations in the C program, which introduce some pauses. This last effect was mainly manifested in the loop-back configuration and minimized considerably in the last steps of this thesis.

To contrast the values for the speed of the FT245 Synchronous FIFO mode pro-

grammed, a quick research on the internet was done. For the same chip and interface, similar values for the velocity were found in both directions: they ranged from 30 Mbyte/s to around 40 Mbyte/s.

To conclude, the goal of this project was to substitute the already existing FT245 Asynchronous FIFO mode with the FT245 Synchronous FIFO interface developed, which can provide a better performance in terms of speed in the exchange of data. In the view of the results exposed in Chapter 4, this target was fulfilled, achieving the velocities specified above for the synchronous interface and knowing that the speed reached with the asynchronous mode was around 8 Mbyte/s, five times less than the one obtained with the module implemented during this thesis.

Bibliography

[1] Howard Johnson and Martin Graham. *High-Speed Signal Propagation: Advanced Black Magic*. Prentice Hall, 2003. 8

[2] Wikipedia. *Parallel communication*, 2018. https://en.wikipedia.org/wiki/Parallel_communication. 8

[3] Wikipedia. *Serial communication*, 2020. https://en.wikipedia.org/wiki/Serial_communication. 9

[4] Howard Johnson and Martin Graham. *High-Speed Digital Design: A Handbook of Black Magic*. Prentice Hall, 1993. 9

[5] Wikipedia. *Duplex (telecommunications)*, 2019. https://en.wikipedia.org/wiki/Duplex_(telecommunications). 10

[6] Wikipedia. *Simplex communication*, 2018. https://en.wikipedia.org/wiki/Simplex_communication. 10

[7] GeeksforGeeks. *Transmission Modes in Computer Networks (Simplex, Half-Duplex and Full-Duplex)*, 2020 accesed February 2020. https://www.geeksforgeeks.org/transmission-modes-computer-networks/. 10

[8] Teach Computer Science. *Simplex, Half Duplex, Full Duplex*, 2020. https://teachcomputerscience.com/simplex-half-duplex-full-duplex/. 10

[9] Geoff Knagge. *The Universal Serial Bus: How it Works and What it Does*, 2020 accesed February 2020. https://www.geoffknagge.com/uni/elec101/essay.shtml. 12

[10] Electronics Notes. *USB Data Transfer & Protocol*, 2020 accesed February 2020. https://www.electronics-notes.com/articles/connectivity/usb-universal-serial-bus/protocol-data-transfer.php. 12

[11] Augmented Startups. *Fun and Easy USB - How the USB Protocol Works*, 2016. https://www.youtube.com/watch?v=F7NlCaaL3yU. 13, 61

[12] Robert Murphy. *USB 101: An Introduction to Universal Serial Bus 2.0*. Cypress, 2020accesed February 2020. https://www.cypress.com/file/134171/download. 13

[13] FTDI chip. *FT2232H Dual High Speed USB to Multipurpose UART/FIFO IC*, 2019 accesed 2019. https://www.ftdichip.com/Support/Documents/DataSheets/ICs/DS_FT2232H.pdf. 13, 14, 15, 16, 17, 18, 34, 61, 63

[14] Xilinx. *Field Programmable Gate Array (FPGA)*, 2020 accesed February 2020. https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html. 18

[15] Diligent. *Nexys Video FPGA Board Reference Manual*, 2017. https://reference.digilentinc.com/_media/reference/programmable-logic/nexys-video/nexysvideo_rm.pdf?_ga=2.65983662.1256648763.1584713870-1730722626.1569573918. 18, 61

[16] Xilinx. *Vivado Design Suite User Guide: Synthesis*, June 2019. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug901-vivado-synthesis.pdf. 18

[17] Xilinx. *Vivado Design Suite User Guide: Implementation*, Dec 2019. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug904-vivado-implementation.pdf. 19

[18] Xilinx. *Implementation Overview for FPGAs*, 2020 accesed February 2020. https://www.xilinx.com/support/documentation/sw_manuals/xilinx11/ise_c_implement_fpga_design.htm. 19

[19] Real Digital. *Synthesize, Implementation and Generate Bitstream*, 2020 accesed February 2020. https://www.realdigital.org/doc/bd6a53089056fc9e2888deabdfcb2a66. 19

[20] ARM. *AMBA 4 AXI4-Stream Protocol*, March 2010. https://static.docs.arm.com/ihi0051/a/IHI0051A_amba4_axi4_stream_v1_0_protocol_spec.pdf. 19, 21, 61

[21] FTDI chip. *D2XX Programmer's Guide*, June 2019. https://www.ftdichip.com/Support/Documents/ProgramGuides/D2XX_Programmer%27s_Guide(FT_000071).pdf. 23

[22] Xilinx. *Clocking Wizard v6.0*, Feb 2020. https://www.xilinx.com/support/documentation/ip_documentation/clk_wiz/v6_0/pg065-clk-wiz.pdf. 28

[23] Xilinx. *Processor System Reset Module v5.0*, Nov 2015. https://www.xilinx.com/support/documentation/ip_documentation/proc_sys_reset/v5_0/pg164-proc-sys-reset.pdf. 28

[24] EDN. *Signal speed on an interconnect: Rule of Thumb 3*, Dec 2013. https://www.edn.com/rule-of-thumb-3-signal-speed-on-an-interconnect/. 30

[25] FTDI chip. *FT2232H Used in an FT245 Style Synchronous FIFO Mode*, Nov 2015. https://www.ftdichip.com/Support/Documents/AppNotes/AN_130_FT2232H_Used_In_FT245%20Synchronous%20FIFO%20Mode.pdf. 37, 38, 41, 43, 48

List of Figures

List of Tables

Acknowledgments

First of all, I would like to thank prof. Angelo Geraci and the members of the Digilab 1 Nicola L., Fabio G., and Nicola C., for supervising my work and guiding me during this project.

This work could not have been possible without my family, which gave me this enrichment opportunity of studying abroad and supported me whenever I needed.

I'd like to thank my friends, for their advice and for keeping in touch even if we are spread all over the world.

I am also very grateful to Carlos, for being my main support here in Milan cheering me up during the dark exam periods. For giving me hope.

Finally, to all members of the Digilab 2, for their kindness and for the good enviroment created.