



**UNIVERSITA' DEL SALENTO**

Corso di Laurea Magistrale in  
"Communication Engineering and Electronic Technologies"

---

*Tesi di Laurea in Fisica Sperimentale*

*Processing of signals from a drift chamber with  
an FPGA*

*Elaborazione dei segnali da una camera a deriva  
mediante una FPGA*

**RELATORE:**

*Ch.mo Prof Marco Panareo  
Dr. Ing. Gianluigi Chiarello*

**STUDENTE:**

*Marco De Liso  
Matricola n° 20061541*

---

Anno Accademico 2020 - 2021

# Contents

<b>INTRODUCTION</b>	<b>4</b>
<b>1. DRIFT CHAMBERS AND CLUSTER COUNTING</b>	<b>6</b>
1.1 Drift chambers	6
1.2 Drift and diffusion of electrons in gases	9
1.3 Ionization by charged particle	9
<b>2. COMPONENTS AND PROTOCOLS</b>	<b>12</b>
2.1 Field Programmable Gate Array – FPGA	12
2.1.1 Artix-7 FPGA and Nexys Video board	16
2.2 The Microblaze core	17
2.3 Universal Asynchronous Receiver-Transmitter - UART protocol	19
2.4 Serial Peripheral Interface-SPI protocol	20
2.5 Vivado Design Suite	23
2.6 Advanced eXtensible Interface 4 (AXI4)	24
<b>3. ALGORITHM DESCRIPTION</b>	<b>28</b>
3.1 The CluTim algorithm	28
<b>4. FPGA IMPLEMENTATION</b>	<b>35</b>
4.1 Peak finding algorithm using the Microblaze Core	35
4.1.1 Vivado implementation	35
4.1.2 Xilinx SDK implementation	42
4.1.3 Hardware setup and results	47
4.2 Peak finding algorithm using Microblaze and a custom IP core	53
4.2.1 Vivado HLS implementation	54
4.2.2 Vivado implementation	57
4.2.3 Xilinx SDK implementation	59
4.2.4 Hardware setup and results	63
4.3 Peak finding algorithm using Microblaze and #pragma BRAMs	64
4.3.1 Vivado HLS implementation	65
4.3.2 Vivado implementation	66

4.3.3 Xilinx SDK implementation.....	69
4.3.4 Hardware setup and results.....	73
4.4 Peak finding algorithm using Microblaze and an AXI DMA.....	74
4.4.1 Vivado HLS implementation.....	74
4.4.2 Vivado implementation.....	76
4.4.3 Xilinx SDK implementation.....	79
4.4.4 Hardware setup and results.....	83
4.5 Peak finding algorithm implemented in hardware.....	84
4.5.1 Vivado HLS implementation.....	85
4.5.2 Vivado implementation.....	87
4.5.3 Xilinx SDK implementation.....	90
4.5.4 Hardware setup and results.....	92
4.6 Efficiency comparison.....	95
<b>CONCLUSIONS</b>	<b>97</b>

# INTRODUCTION

In this dissertation the processing of signals from a drift chamber with a Field Programmable Gate Array (FPGA) is presented. In order to do this, the first step consists in describing the operation of drift chambers. As we shall see, drift chambers are general-purpose detectors used in high energy physics experiments which can track and identify charged particles. A drift chamber consists of a large volume of gas with instrumented wires held at different voltages. When charged particles move through the chamber, they ionize the gas particles. The electrons from these primary ionizations drift towards the wires held at high positive voltage, while the ions drift towards the grounded wires. The sense wires are very thin ( $\sim 20\mu\text{m}$ ), such that the strong electric field accelerates the electrons enough to cause further ionization near the sense wire. The new electrons ionize further into an avalanche, which is registered as an electronic signal on the sense wire. The signal pulses from all the wires are then collected and the particle trajectory is tracked with the cluster counting/timing technique [1].

By identifying the number of ions clusters generated by a charged particle along its trajectory as well as the drift distance and time associated to each cluster, the cluster counting/timing technique allows to track the trajectory of a particle in a drift chamber.

The peaks in the signal coming from the drift chamber correspond to the clusters generated by the particle, while the time difference between two peaks coincides with the time difference between the generation of the two clusters.

The objective of this work is to use an FPGA in order to analyze in real-time the signals coming from a drift chamber. By finding the peaks and their timing, it is possible to reconstruct the trajectory of the particle in the drift chamber.

An ADC is used to convert the analog signals in the digital domain.

Variants of a fast readout algorithm (CluTim) for identifying the peaks (in the digitized input signals) are implemented on the Nexys Video board which mounts an Artix-7 FPGA.

FPGAs are integrated circuits, which are sets of circuits on a chip that are designed to be configured after manufacturing. An FPGA architecture consists of thousands of configurable logic blocks (CLBs) which includes look-up tables (LUTs), flip-flops (FFs) and multiplexers.

When using FPGAs, the hardware of the device is programmed rather than writing software to run on a predefined processor. FPGAs are primarily programmed using hardware description languages (HDLs).

A brief description of the hardware and software components used in this work is provided in chapter 2, while the peak-finding algorithm is described in the third chapter.

In chapter 4 different FPGA designs are presented. Each design exploits a different FPGA architecture in order to implement the peak-finding algorithm.

The architectures are created by using Vivado Design Suite, which is a software suite produced by Xilinx for synthesis and analysis of hardware description language (HDL) designs.

As we will see, each FPGA design will progressively use less software resources in order to implement the peak-finding algorithm. The efficiency of each design is provided.

A short final section contains the conclusions.

This work was done in collaboration with *I.N.F.N. Sezione di Lecce (via per Arnesano, I – 73100 Lecce, Italy)*.

# 1 DRIFT CHAMBERS AND CLUSTER COUNTING

In this chapter an introduction on drift chambers and cluster counting is presented.

## 1.1 Drift chambers

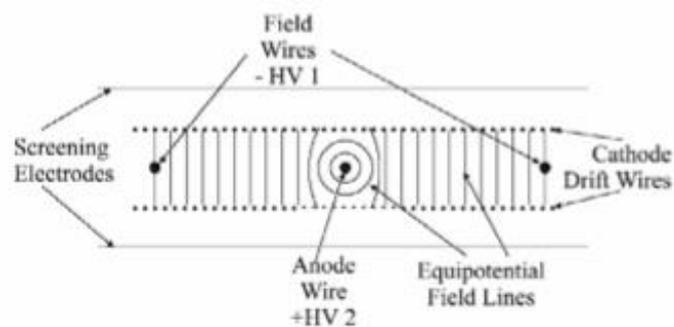
A drift chamber is a particle tracking detector that measures the drift time of ionization electrons in a gas to calculate the spatial position of ionizing particles. Spatial resolution is achieved by measuring the time it takes for the electrons to reach the anode wire, measured from the instant that the ionizing particle traversed the detector.

Drift chambers use an array of wires at high voltage (anodes), which run through a chamber with conductive walls held at ground potential (cathodes). The chamber is filled with a carefully chosen gas such that any ionizing particle that passes through the tube will ionize surrounding gaseous molecules.

When a charged particle passes through a gas, it will interact electromagnetically with gas molecules creating electron/ion pairs along the path of the particle. The number of such pairs depends on the energy of the particle and the type of gas in the drift chamber. If an electric field is applied, the electrons will start to drift towards the positive electrode (anode wire), undergoing repeated collisions with the gas molecules. If the electric field near the anode is strong enough, an electron can acquire enough energy between collisions to knock an additional electron free from a gas molecule. This additional electron can then go on to ionize more gas molecules; in this way, an avalanche is formed in which the number of electrons increases exponentially. When this avalanche reaches the positive electrode, it gives rise to a measurable current, the size of which is proportional to the original number of ions created. An electron sitting in the gas far away from the anode will see a much

smaller electric field and will drift towards the anode with a velocity roughly proportional to the field. When it gets close to the anode, the electric field will start to rapidly increase, and the electron will initiate an avalanche. The fact that an electron drifts with a predictable speed towards the anode can be exploited in order to turn a measurement of the time an electron took to drift to the anode into a measurement of the distance of the original source particle from the anode itself. In this way, the drift chambers are able to provide accurate measurement of the position of a charged particle.

Stacking several drift chambers permits to resolve a track of a charged particle [3].

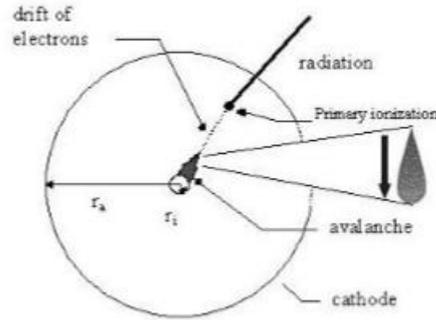


**Fig. 1** Schematic of a drift chamber. [2]

In Fig. 1 the schematic of a drift chamber is shown:

- the cathode drift wires carry a negative voltage which varies from ground (close to the anode wire) to a high negative voltage (close to the field wire);
- the anode wire is where electrons from ionized gas atoms will drift and be collected;
- the field wires are there to keep the equipotential lines linear over a larger range.

In order to track the particle, the time lag between the timing of the signal and the instant of particle passage is exploited. This time lag is related to the time the electrons from the ionization take to travel (drift) to the anode wire (Fig. 2).



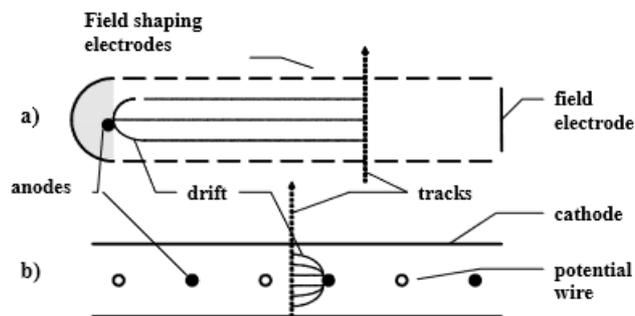
**Fig. 2** Tracking principle. [2]

So, if the timing of the particle passage is determined with high precision, the time lag can be used to determine the exact position of the ionization with respect to the anode wire. The drift time  $t_{drift}$  formula is:

$$t_{drift} = \int_{track}^{anode} \frac{ds}{v_{drift}(x)}$$

There are two basic schemes for drift chambers:

- the method of constant drift field (Fig. 3a) giving a simple dependence for the drift distance  $d = t_{drift} \cdot v_{drift}$  with  $v_{drift} = const$ ;
- variable drift field (Fig. 3b) where the gas is chosen such that the drift velocity is independent of the drift field.



**Fig. 3** a) constant b) variable drift field. [2]

## 1.2 Drift and diffusion of electrons in gases

The motion of an electron in the gas is due to two physical phenomena: drift and diffusion.

The velocity due to drift can be determined from the mobility of a charge in a gas, which is given by  $v_{\text{drift}} = \mu E$ , where  $v_{\text{drift}}$  is the drift velocity,  $\mu$  is the mobility and  $E$  is the electric field strength. In a uniform electric field, the electrons will drift with a constant velocity because the electrons collide frequently slowing them down, while the electric field accelerates them.

Diffusion consists in the random motion caused by the thermal energy of the electron. The result is that after a time  $t$ , a collection of electrons originally distributed in a straight line at  $x = 0$  will be distributed according to a Gaussian distribution.

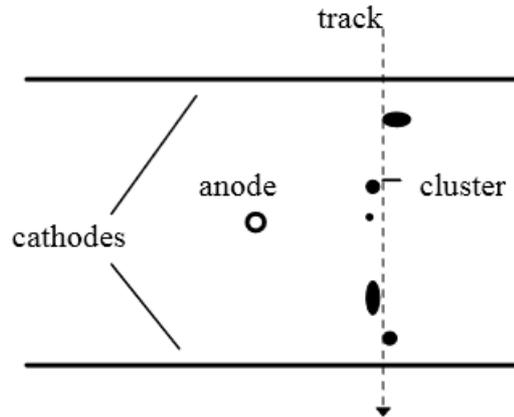
## 1.3 Ionization by charged particle

The first ion that a charged particle creates in the gas is called the primary ionization. When the electron from the primary ionization is accelerated by the electric field, it will eventually gain enough energy to ionize other molecules of the gas. Then, these secondary ionizations can gain enough energy to cause further ionizations, and so on. This effect is called avalanche multiplication. The electric field is constant through most of the chamber, except very near to the anode wire, where its value is  $\propto 1/r$ , where  $r$  is the radius of the wire. Thus, most avalanche multiplication takes place near the anode wire.

When a charged particle goes through a gas, it loses energy because of two phenomena: excitation and ionization:

- in excitation, the particle passes a specific amount of energy to a gas molecule;
- in ionization, the particle knocks an electron off the gas molecule, and leaves a positively charged ion.

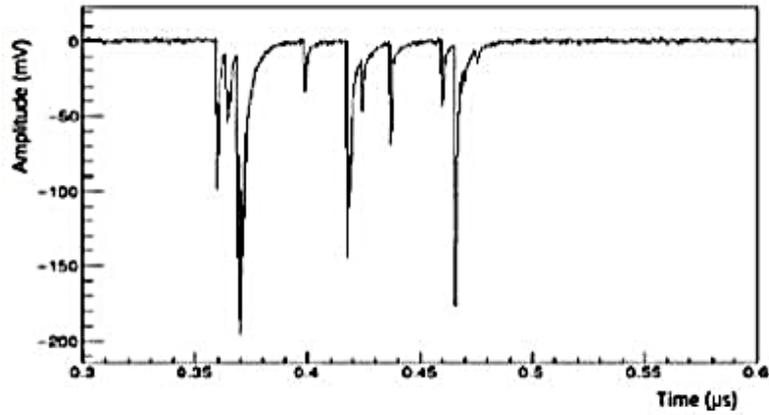
However only ionization generates signals in the drift chamber. In fact, the tracking of the particles is made using the ionization clusters left behind by the particle itself (Fig. 4).



**Fig. 4** Ionization clusters distribution along track. [2]

The ionization clusters are produced by the coupling between the electrical field of the passing particle and the electrons of the gas molecules. In particular, the electrical field of the passing particle may have enough impact to kick out the electron of an atomic shell. This free electron may ionize again, and, in this case, it is considered as a new charged particle and is called  $\delta$ -ray. The ionization left behind by these individual encounters, is called a cluster and the set of clusters represents the footprint of the charged particle, which will be detected in the drift chamber.

As already mentioned, the peaks in the signal generated by the drift chamber correspond to the clusters generated by the particle, while the time difference between two peaks represents the time difference between the creation of the two clusters. A typical simulated waveform coming from the drift chamber is represented in figure 5.



**Fig. 5** Typical simulated waveform. [1]

As shown in figure 5, the time interval between two peaks is  $\sim 1$  ns and the overall duration of the signal is about 100 ns. In order to track the trajectory of a particle it is crucial to detect and store the timing of the peaks. To do this, a resolution in the order of the nanosecond is needed.

## **2 COMPONENTS AND PROTOCOLS**

This chapter focuses on the hardware components and software tools that are used in the project.

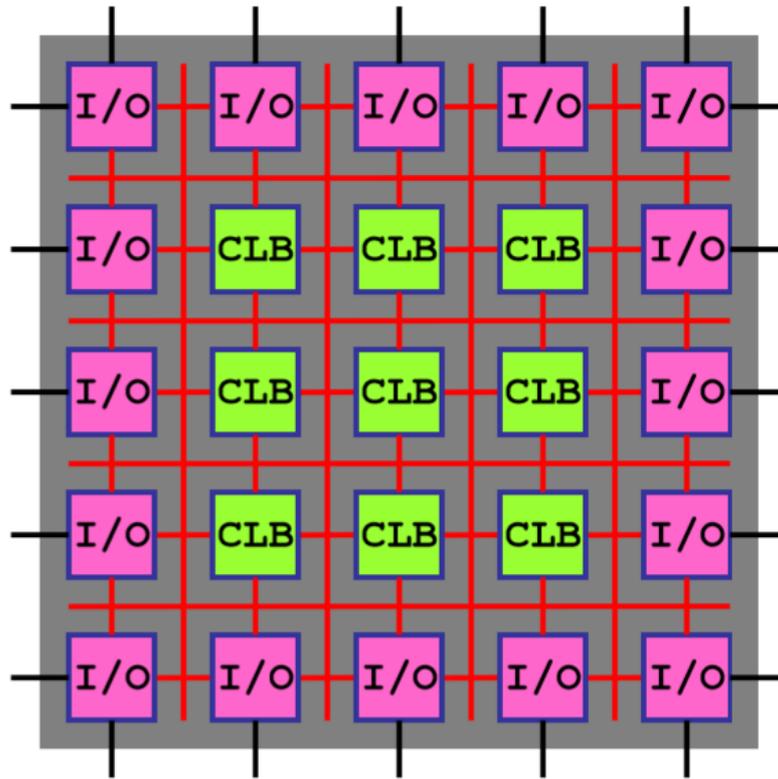
### **2.1 Field Programmable Gate Array - FPGA**

Since in this work an FPGA is used in order to process the signals coming from a drift chamber, this section presents the internal structure and the functioning of an FPGA.

The Field Programmable Gate Array (FPGA) technology was first developed by Ross Freeman in 1985 who was among the founders of the company Xilinx. Today, Xilinx and Altera are the two largest FPGA manufacturers in the world.

FPGAs are integrated circuits that a user can program to carry out one or more logical operations. The FPGA configuration is generally specified using a hardware description language (HDL), like that used for an application-specific integrated circuit (ASIC). FPGAs can be reprogrammed to desired application or functionality requirements after manufacturing. This feature distinguishes FPGAs from ASICs, which are custom manufactured for specific design tasks.

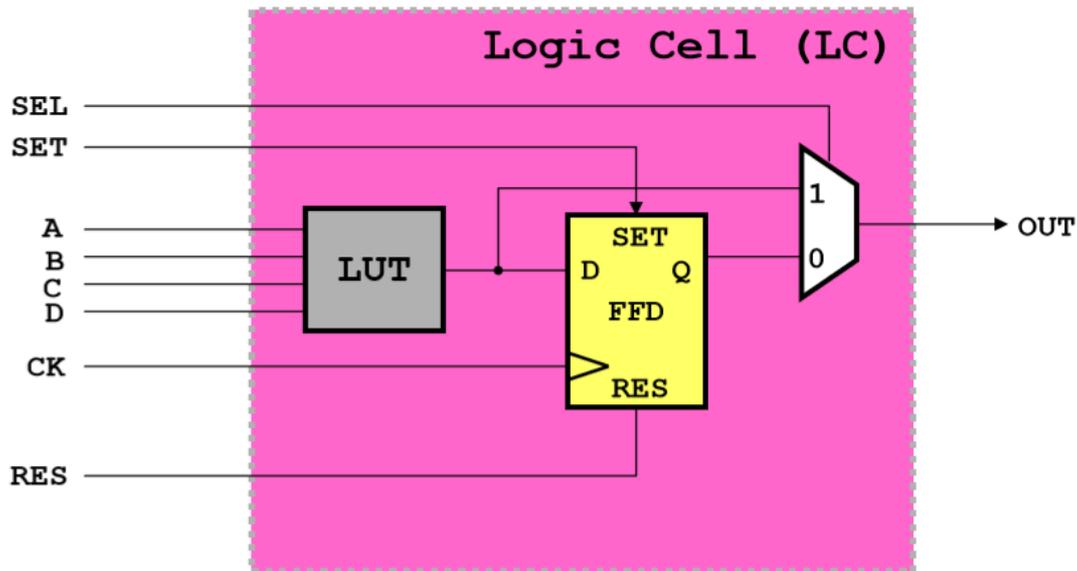
FPGAs contain an array of configurable logic blocks (CLB) and a hierarchy of reconfigurable interconnects allowing blocks to be wired together. Logic blocks can be configured to perform complex combinational functions and they also include memory elements, which may be simple flip-flops or more complete blocks of memory.



**Fig. 6** Internal structure of an FPGA. [10]

As it is shown in figure 6, CLBs make up the main structure of the FPGA and they implement the user logic. Interconnects provide direction between the logic blocks to implement the user logic and the switch matrices provide switching between interconnects. Finally, I/O pads, used for the outside world to communicate with different applications, are implemented.

In figure 7 the schematic of a logic cell (LC) inside a CLB is represented.



**Fig. 7** Internal structure of logic cells (LC). [10]

In a logic cell, usually there are one Lookup Table (LUT), one D Flip-Flop and one 2-to-1 Mux (Fig. 7). The LUTs in the logic cell are small memories (RAM) that perform logic operations. Complex and massive programs are created as a result of the combination of thousands of Logic Cells. According to the program loaded on the FPGA, the interconnection of the logic cells is provided with matrix-shaped data paths and programmable switches [3].

FPGAs are usually mounted on evaluation boards which consist of a printed circuit containing the support logic needed in order to simulate the behavior of the FPGA. On the FPGA board there are dedicated pins and user pins. There are 3 types of dedicated pins, divided according to the specific functions they perform in the FPGA:

- configuration pins: used for downloading program onto the FPGA;
- clock pins: special pins reserved for Clock signals;
- power pins: which provide power and ground connection for the FPGA.

User pins are standard I/O pins, and they are configurable. These pins are divided into three categories: Input, Output and Input/Output. Each I/O pin is connected to a single I/O cell in the FPGA. Power to I/O cells is provided by VCCIO (this voltage is used for feeding all I/O pins, except memory-related pins) [3].

One or more LCs are grouped together in order to form a slice (Fig. 8).

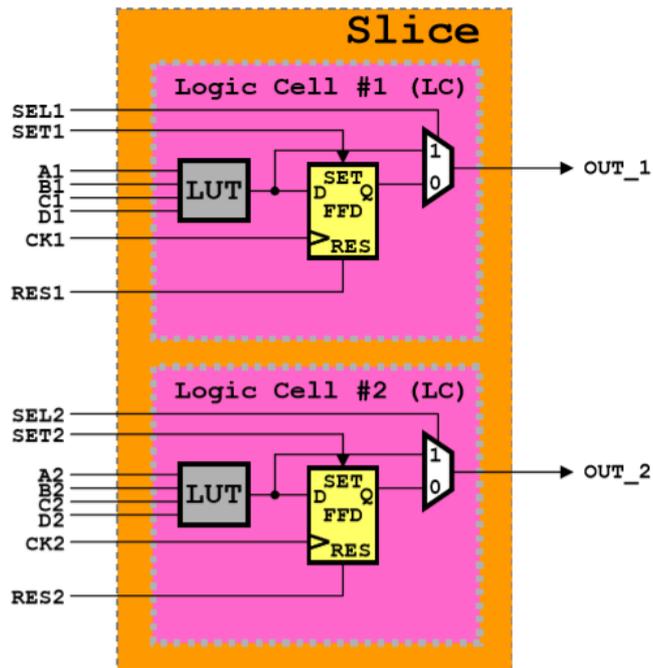


Fig. 8 Internal structure of a slice. [10]

Finally, one or more slices are grouped together to form a CLB (Fig. 9).

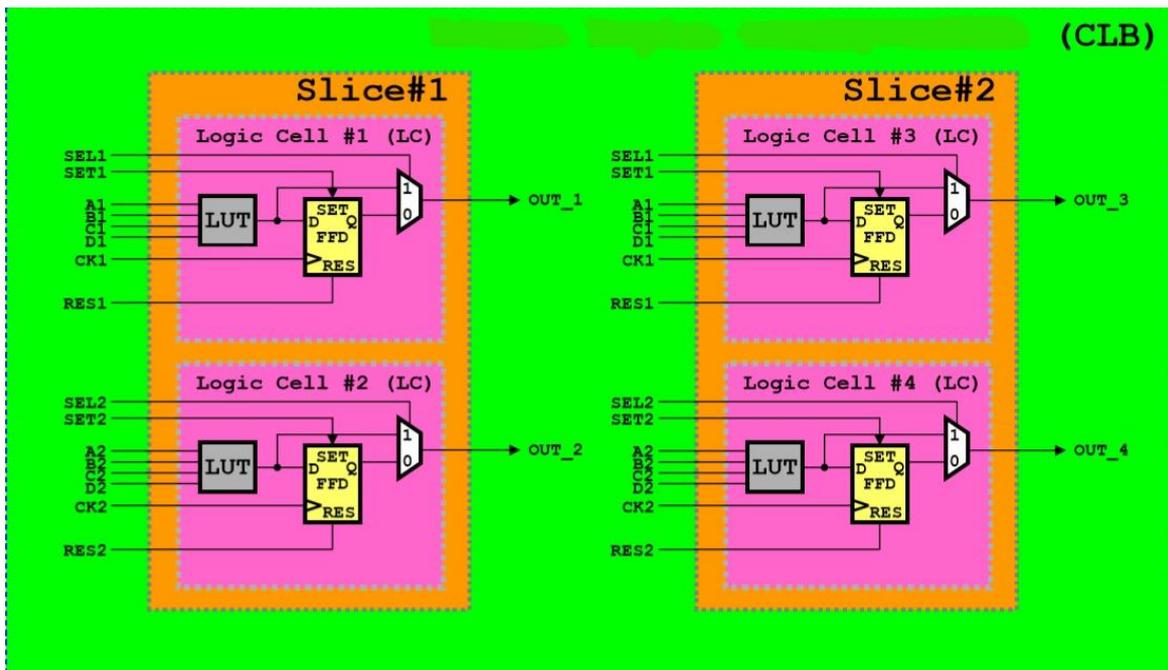


Fig. 9 Internal structure of a CLB. [10]

The processing inside the FPGA is synchronous, this means that the FPGA designs are ‘clock’ based and the D flip-flops in the FPGA change their state following the clock signal. Moreover, the FPGAs can operate in parallel processing.

FPGAs have gained rapid growth over the past decade because they are useful for a wide range of applications. Applications of FPGAs include digital signal processing, bioinformatics, device controllers, software-defined radio, random logic, ASIC prototyping, medical imaging, computer hardware emulation, integrating multiple SPLDs, voice recognition, cryptography, filtering and communication encoding and many more.

### **2.1.1 Artix-7 FPGA and Nexys Video board**

The specific FPGA board used in this work is a Nexys Video board which mounts an Artix-7 FPGA (Fig. 10).



**Fig. 10** Nexys Video board. [4]

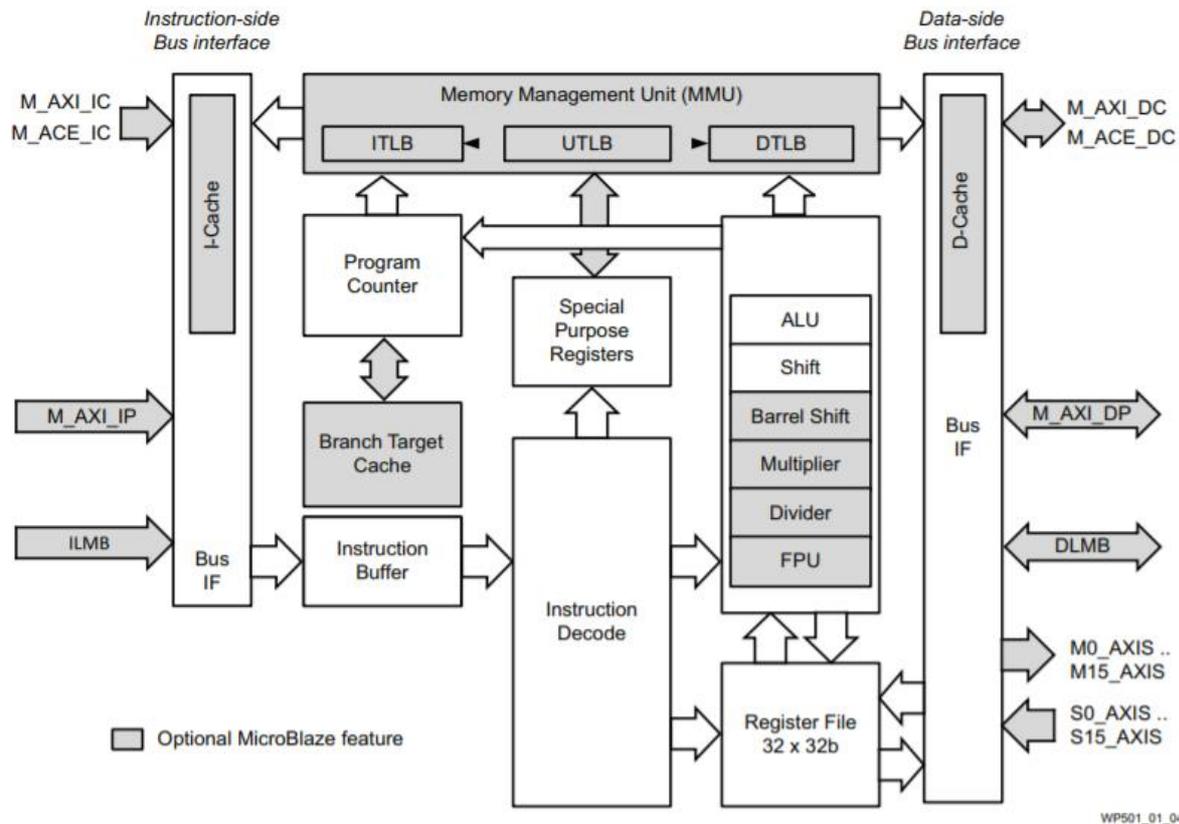
The Nexys Video board is a digital circuit development platform based on the latest Artix-7 Field Programmable Gate Array from Xilinx. It includes several built-in peripherals such as Ethernet, USB-UART, and a high bandwidth USB data transfer protocol. The Nexys Video also includes provisions for direct human interaction with a connector for USB-HID devices, an OLED display, and a vast assortment of switches, buttons, and LEDs. Moreover, the FMC and Pmod ports can be used to add any connection interface that the application demands. Nexys Video is programmed using Vivado Design Suite [4].

The Artix-7 FPGA is optimized for high performance logic, and offers more capacity, higher performance, and more resources than earlier designs [4]. Artix-7 200T features include:

- 33,650 logic slices, each with 4 6-input LUTs and 8 flip-flops;
- close to 13 Mbits of fast block RAM (3x more than the Nexys 4 DDR);
- 10 clock management tiles, each with phase-locked loop (PLL);
- 740 DSP slices;
- internal clock speeds exceeding 450 MHz;
- on-chip analog-to-digital converter (XADC);
- up to 3.75Gbps GTP transceivers.

## **2.2 The Microblaze core**

To enable the development of more powerful embedded solutions with the Xilinx FPGAs the company has provided a soft-core processor, Microblaze. The Microblaze CPU is a customizable 32-bit RISC microprocessor, optimized for implementation in Xilinx FPGAs [5]. Figure 11 shows a block diagram of the Microblaze processor core.



**Fig. 11** Microblaze Processor Block Diagram. [5]

The Microblaze processor uses Big-Endian or Little-Endian format to represent data, depending on the selected endianness and it has 32 general purpose registers each of which can be made up of 32-bit or 64-bit and up to 16 special purpose registers, depending on configured options.

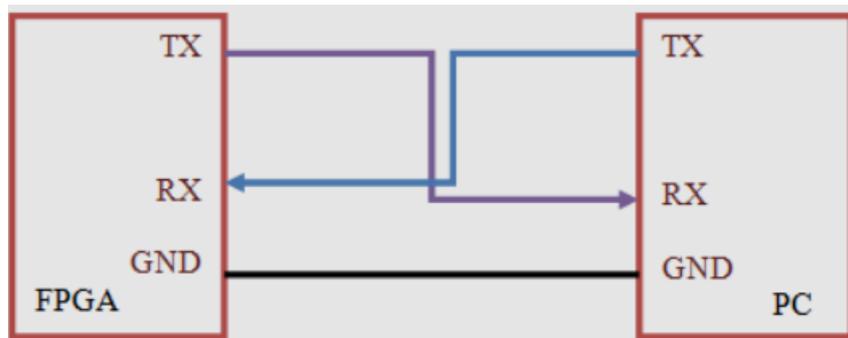
Microblaze instructions execution are pipelined. For most instructions, each stage takes one clock cycle to complete. Consequently, the number of clock cycles necessary for a specific instruction to complete is equal to the number of pipeline stages, and one instruction is completed on every cycle in the absence of data, control or structural hazards. It is possible to have a 3, 5 or 8 stage pipeline.

The Microblaze core is organized as a Harvard architecture with separate bus interface units for data and instruction accesses. Two memory interfaces are supported: Local Memory Bus (LMB), and the AMBA AXI4 interface or ACE interface. The LMB provides the access to the on-chip dual-port block RAM. The

AXI4 interface provides a connection to both on-chip and off-chip peripherals and memory. The ACE interface provides cache coherent connections to memory [6].

## 2.3 Universal Asynchronous Receiver-Transmitter - UART protocol

In this work the serial communication between the FPGA and the PC is handled by the UART protocol (Fig. 12).



**Fig. 12** UART protocol. [11]

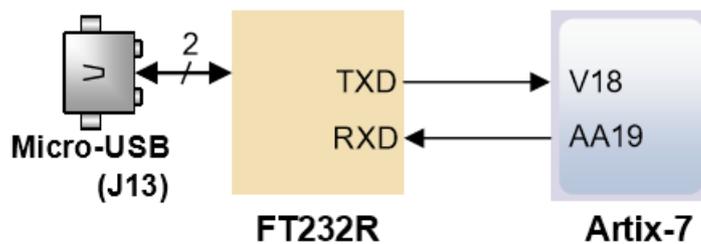
In UART communication, two UARTs communicate directly with each other. The transmitting UART converts parallel data from a controlling device into serial form and transmits it in serial to the receiving UART, which converts the serial data back into parallel data for the receiving device. Only two wires are needed to transmit data between two UARTs.

As one understands from their name, UARTs transmit data asynchronously, which means there is no clock signal to synchronize the output of bits from the transmitting UART to the sampling of bits by the receiving UART. Instead of a clock signal, the transmitting UART adds start and stop bits to the data packet being transferred. Both UARTs must operate at about the same baud rate.

The UART that will transmit data receives the data from a data bus. The data bus is used to send data to the UART by another device. Data is transferred from the data bus to the transmitting UART in parallel form. After the transmitting UART gets the parallel data from the data bus, it adds a start bit, a parity bit, and a stop bit, creating the data packet. Next, the data packet is put as output serially. The receiving UART reads the data packet bit by bit, then it converts the data back into parallel form and removes the start, parity and stop bits. Finally, the receiving UART transfers the data packet in parallel to the data bus on the receiving end.

The Nexys Video includes a FTDI FT232R USB-UART bridge (attached to connector J13) that allows one to use PC applications to communicate with the board using standard Windows COM port commands. Serial port data is exchanged with the FPGA using a two-wire serial port (TXD/RXD) with no handshake signals. Two on-board status LEDs provide visual feedback on traffic flowing through the port: the transmit LED (LD13) and the receive LED (LD12) [4].

In figure 13 the connection between the Artix-7 FPGA and the UART on the Nexys Video board is shown.



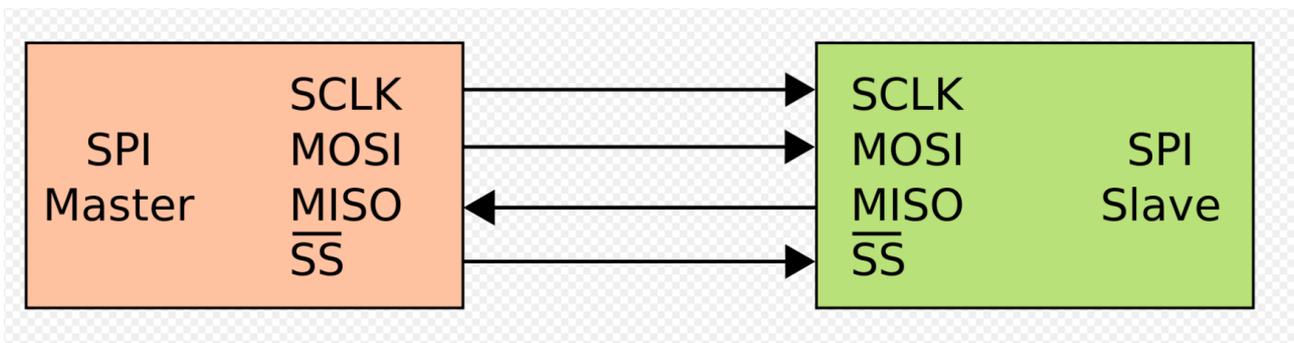
**Fig. 13** Nexys Video FT232R connections. [4]

## 2.4 Serial Peripheral Interface-SPI protocol

The ADC SPI allows the user to configure the analog to digital converter for specific functions or operations through a structured register space provided inside the ADC. The SPI gives the user added flexibility and customization, depending on the

application. Addresses are accessed via the serial port and can be written to or read from the port. Memory is organized into bytes that can be further divided into fields [7]. In this section the functioning of the SPI protocol is presented.

Serial peripheral interface (SPI) is one of the most widely used interfaces between microcontroller and peripheral ICs such as sensors, ADCs, DACs, shift registers, SRAM, and more. SPI is a synchronous, full duplex master-slave-based interface. Figure 14 represents the connections between the master and the slave in a SPI communication.

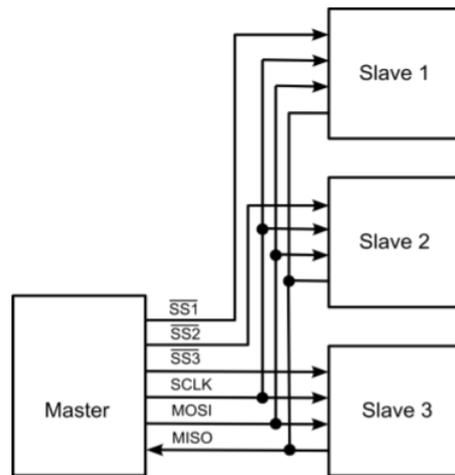


**Fig. 14** Connections between a master and a single slave. [8]

The SPI interface presents four lines:

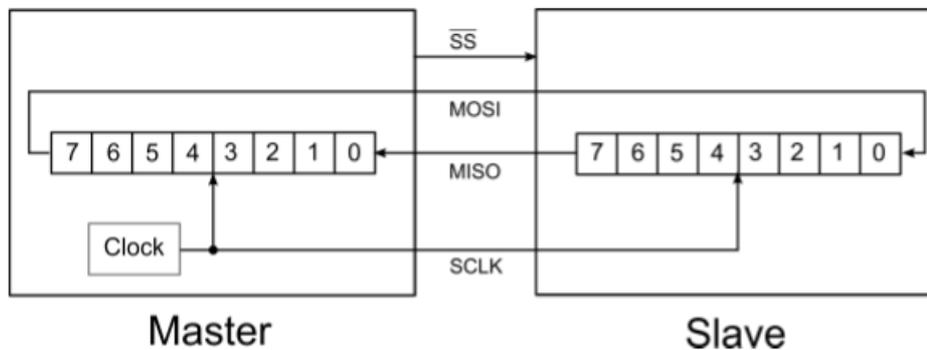
- SCLK (Serial clock) where the clock is implemented;
- MOSI (Master output slave input) where the master writes and the slave listens;
- MISO (Master input slave output) where the master receives bits from the slave;
- SS (slave select) used by the master in order to select a specific slave; the line is active low.

The number of SS lines is equal to the number of slaves. In fact, if more than one slave is present, the SCLK, MISO and MOSI lines are in parallel for every slave (Fig. 15).



**Fig. 15** SPI connections with more than one slave. [12]

The SPI interface can be represented as a shift register between the master and the slave (Fig. 16).



**Fig. 16** Connection between master and slave highlighting the internal shift registers. [12]

Figure 16 shows in detail how the communication between master and slave is managed in the SPI protocol. The MSB of the master is connected to the LSB of the slave on the MOSI line, whereas the MSB of the slave is connected to the LSB of the master on the MISO line. On every clock cycle the MSB of the master shifts towards the LSB of the slave and simultaneously the MSB of the slave shifts towards the LSB of the master. So, after eight clock cycles the byte inside the master will be completely transferred to the slave and vice versa.

The transmitting modality is handled with two control bits: CPOL (clock polarity) and CPHA (clock phase). The CPOL bit determines on which edge of the clock the operations are made, whereas the CPHA bit determines the phase of the clock.

## **2.5 Vivado Design Suite**

A key step in order to process the pulses coming from the drift chamber consists in developing and test the peak-finding algorithm on the Nexys Video board using the software Vivado Design Suite.

The Vivado Design Suite is a software suite produced by Xilinx for synthesis and analysis of HDL designs. Vivado was launched in April 2012, and it is an integrated design environment (IDE) with system-to-IC level tools built on a shared scalable data model and a common debug environment. Vivado includes electronic system level (ESL) design tools for synthesizing and verifying C-based algorithmic IP; standards-based packaging of both algorithmic and RTL IP for reuse; standards-based IP stitching and systems integration of all types of system building blocks; and the verification of blocks and systems. The Vivado software includes [9]:

- the Vivado High-Level Synthesis compiler, which enables C, C++ and SystemC programs to be directly targeted into Xilinx devices without the need to manually create RTL. Vivado 2014.1 introduced support for automatically converting OpenCL kernels to IP for Xilinx devices. OpenCL kernels are programs that execute across various CPU, GPU and FPGA platforms;
- the Vivado Simulator which is a compiled-language simulator that supports mixed-language, Tcl scripts, encrypted IP and enhanced verification;
- the Vivado IP Integrator which allows to quickly integrate and configure IP from the large Xilinx IP library;
- the Vivado Tcl Store which is a scripting system for developing additional components to Vivado and can be used to add and modify Vivado's

capabilities. Tcl is the scripting language on which Vivado itself is based. All Vivado's underlying functions can be invoked and controlled via Tcl scripts.

## **2.6 Advanced eXtensible Interface 4 (AXI4)**

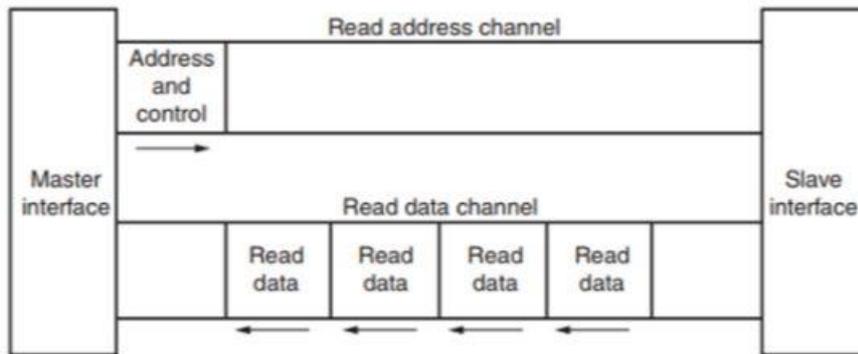
Advanced eXtensible Interface 4 (AXI4) is a family of buses defined as part of the fourth generation of the ARM Advanced Microcontroller Bus Architecture (AMBA) standard.

The AMBA specification defines 3 AXI4 protocols:

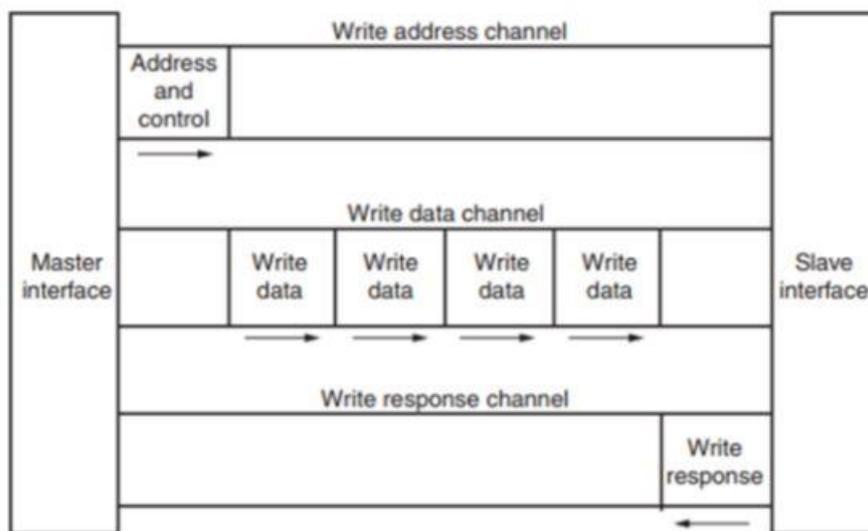
- AXI4: a high-performance memory mapped data and address interface capable of burst access to memory mapped devices;
- AXI4-Lite: a subset of AXI4, lacking burst access capability; has a simpler interface than the full AXI4 interface;
- AXI4-Stream: a fast unidirectional protocol for transferring data from master to slave.

Xilinx Vivado helps in the creation of custom IP with AXI4 interfaces. These can be connected to the Microblaze's Processing System or to other devices.

The AXI4 interface consists of five channels: Read Address (RA), Read Data (R), Write Address (AW), Write Data (W), and Write Response (B). An AXI4 read transaction using the Read Address and Data channels is shown in figure 17. Similarly, an AXI4 write transaction using the Write Address, Data, and Response channels is shown in figure 18 [14].



**Fig. 17** AXI4 read transaction [14].



**Fig. 18** AXI4 write transaction [14].

Any AXI component has two global signals: the clock  $ACLK$  and an active-low asynchronous reset,  $ARESETN$ . All AXI4 signals are sampled on the rising edge of the clock and all signal changes must occur after the rising edge. All five transaction channels use the same  $VALID/READY$  handshake process to transfer address, data, and control information. This two-way flow control mechanism means that both the master and slave can control the rate at which the information moves between master and slave. The information source generates the  $VALID$  signal to indicate when the address, data or control information is available. The information destination generates the  $READY$  signal to indicate that it can accept the information. The

handshake completes if both VALID and READY signals in a channel are asserted during a rising clock edge [14].

AXI4 is a burst-based protocol, meaning that there may be multiple data transfers for a single request. In AXI4, bursts can be of three types, selected by the signals ARBURST (for reads) or AWBURST (for writes) [15]:

- FIXED;
- INCR;
- WRAP.

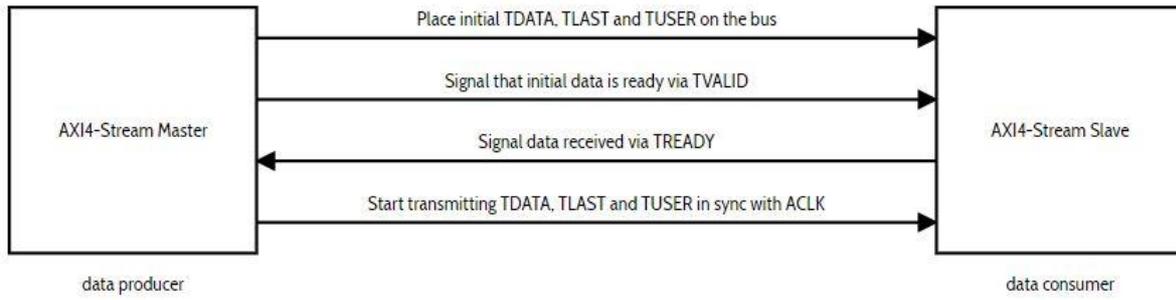
In FIXED bursts, each beat within the transfer has the same address. This is useful for repeated access at the same memory location, such as when reading or writing a FIFO.

In INCR bursts, on the other hand, each beat has an address equal to the previous one plus the transfer size. This burst type is commonly used to read or write sequential memory areas.

WRAP bursts are like the INCR ones, as each transfer has an address equal to the previous one plus the transfer size. However, with WRAP bursts, if the address of the current beat reaches the “Higher Address boundary”, it is reset to the “Wrap boundary”.

The AXI4-Lite is a subset of the AXI4 protocol, providing a register-like structure with reduced features and complexity.

AXI4-Stream is designed to transport data streams of arbitrary width in hardware. Usually 32-bit bus width is used, which means that 4 bytes get transferred during one cycle. At 100MHz of programmable logic frequency on FPGAs this can create a throughput of magnitude of hundreds of megabytes per second depending on memory capabilities. In AXI4-Stream TDATA width of bits is transferred per clock cycle. The transfer is started once sender signals TVALID, and the receiver responds with TREADY. TLAST signals the last byte of the stream (Fig. 19) [16].



**Fig. 19** Axi4-stream handshake. [16]

## **3 ALGORITHM DESCRIPTION**

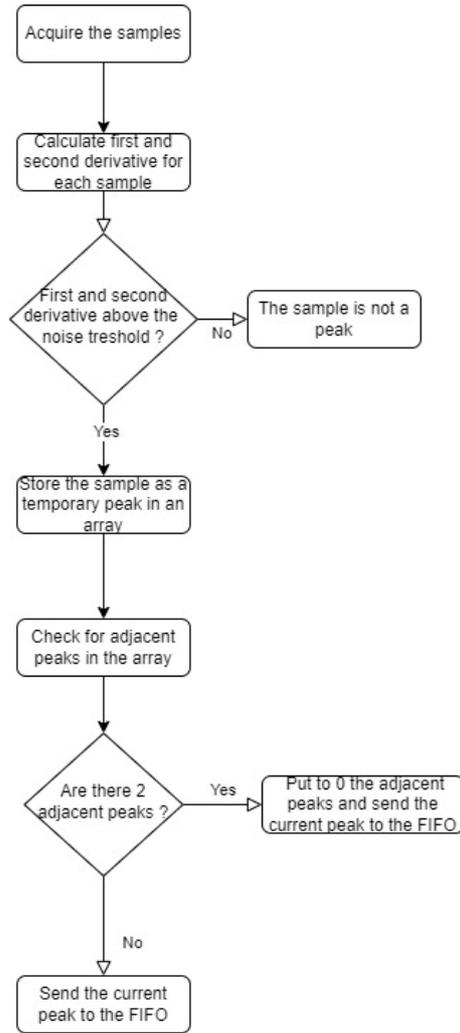
In order to process the data coming from the drift chamber and the ADC, a VHDL code is implemented on Vivado Design Suite. The objective of the code consists in applying the cluster counting/timing technique. The following chapter describes how the CluTim algorithm works.

### **3.1 The CluTim algorithm**

As already stated in chapter 1, when a charged particle passes through the gas inside a drift chamber, it leaves behind several clusters produced by the coupling between the electrical field of the passing particle and the electrons of the gas atoms. The set of clusters represents the footprint of the charged particle, which will be detected in the drift chamber.

In order to apply the Cluster Counting/Timing technique it is necessary to identify the different peaks of the signal coming from the drift chamber and their time of arrival. In this way, the trajectory of the particle in the drift chamber can be reconstructed.

The CluTim algorithm can be used in order to detect each peak and its timing information. In particular, the CluTim algorithm approach to find the peaks is based on derivatives, the first derivative calculates the singular points of the function, while the second derivative calculates whether the singular points are a maximum or a minimum based on the slope of the function. The flow chart of the algorithm is provided in figure 20.

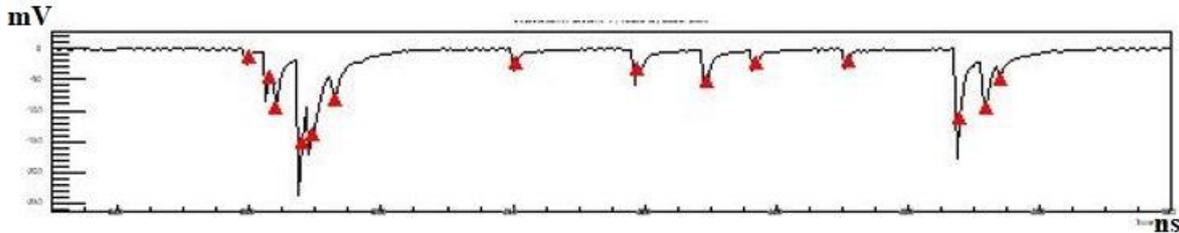


**Fig. 20** Flow chart of the CluTim algorithm.

A peak is found when  $f$ ,  $f'$  and  $f''$  are above a threshold level defined according to r.m.s. noise of the signal function  $f$ , and when the time difference between contiguous peaks is larger than the time bin resolution. The first and second derivative of the digitized signal function  $f$  are calculated, and their value is compared to some thresholds related to the signal noise level [13]. The algorithm calculates the first and second derivative for each time bin  $i$ :

$$f'(i) = \frac{f(i) - f(i - \Delta b)}{\Delta b}, \quad f'' = f'(i) - f'(i - 1)$$

where  $\Delta b$  is the number of bins over which the average value of  $i$  is calculated. The algorithm identifies the peaks corresponding to the ionization clusters, puts each peak amplitude and timing in a memory and sends the stored data when a trigger signal occurs [13]. A typical waveform with the found peaks is illustrated in figure 21.



**Fig. 21** A typical waveform where red markers identify the peaks recognized by the algorithm. [1]

As already mentioned, the hardware setup includes an ADC connected to an FPGA. The ADC receives the analog pulses from the drift chamber and, after converting them into the digital domain, it sends them to the FPGA. If the data from the ADC were stored without preprocessing, this would present several disadvantages, such as [1]:

- a time window of observation of the event greatly reduced, because the internal memory to the FPGA would be filled very quickly;
- a very long data transfer time from FPGA to PC for their postprocessing, due to the very large amount of data to be transferred;
- a very long time to postprocess data.

All these issues can be resolved by using the CluTim algorithm, which performs a reduction and only the data useful for experimental purposes are stored because the CluTim processes the data in real time. In fact, the CluTim algorithm:

- identifies, in the digitized signal, the peaks corresponding to the different ionization electrons;
- stores each peak amplitude and timing in an internal memory;
- sends the data stored to an external device when specific trigger signals occur.

In VHDL code the determination of a peak is done by relating the  $i$ th sampled bin to a number  $n$  of preceding bins [1].

At every time instant  $t_i$ , a string made of 16 samples  $S_{K,X}$  (each sample is made up of 12 bits which is the ADC resolution) is received;  $K$  is the sample number which ranges from 0 to 16 and  $X$  is the time at which the samples are present. The first step of the algorithm consists in calculating the functions  $D1_{K,X}$  and  $D2_{K,X}$  (which represent the first and second derivatives of the input signal, which are calculated as incremental differences) according to the following formulas:

$$D1_{K,X} = \left( \frac{2 * S_{K,X} - S_{K-1,X} - S_{K-2,X}}{16} * 3 \right)$$

$$D2_{K,X} = \left( \frac{2 * S_{K,X} - S_{K-2,X} - S_{K-3,X}}{16} * 5 \right)$$

The value of  $D1_{K,X}$  function provides an estimate of the variation of the amplitude of the  $i$ th sample compared to the  $(i-1)$ -th and  $(i-2)$ -th samples. Likewise, the  $D2_{K,X}$  function calculates the variation of the amplitude of the  $i$ th sample with respect to the  $(i-2)$ -th and  $(i-3)$ -th samples. The 3 and 5 coefficients in the formulas are experimentally obtained, they are used in order to increase the value of  $D1_{K,X}$  and  $D2_{K,X}$  with respect to the noise level. In this way the algorithm can better recognize the peaks.

The values of the  $D1_{K,X}$  and  $D2_{K,X}$  functions are stored in a 16-element vector. In order to calculate the first three elements of the vector ( $D1_{0,X}$ ,  $D1_{1,X}$ ,  $D1_{2,X}$ ,  $D2_{0,X}$ ,  $D2_{1,X}$ ,  $D2_{2,X}$ ) it is necessary to use the last three samples of the previous 16 input word ( $S_{13,X-1}$ ,  $S_{14,X-1}$ ,  $S_{15,X-1}$ ). Therefore, a temporary storage is used to this purpose.

In the second step, the values of  $D1_{K,X}$  and  $D2_{K,X}$  and the differences between  $D1_{K,X}$  and  $D1_{K-1,X}$  and between  $D2_{K,X}$  and  $D2_{K-1,X}$  are compared with respect to thresholds proportional to the level of noise present in the input signal. So, in order to calculate the first difference between  $D1_{K,X}$  and  $D1_{K-1,X}$  and between  $D2_{K,X}$  and  $D2_{K-1,X}$ , it is

necessary to temporarily store the last two elements of the previous 16-elements words ( $D1_{14,X}$ ,  $D1_{15,X}$ , and  $D2_{14,X}$ ,  $D2_{15,X}$ ).

If the  $D1_{K,X}$ ,  $D2_{K,X}$  and the respective differences are above the threshold, the corresponding sample  $S_{K,X}$  is temporarily stored in a  $M_{K,X}$  element of a 16-word vector. The latter contains all the detected peaks.

The third step consists in checking if there are no adjacent peaks. To this purpose, every element  $M_{K,X}$  of the vector is analyzed. If the element  $M_{K,X}$  contains a non-zero value, the two preceding locations ( $M_{K-1,X}$ ,  $M_{K-2,X}$ ) are assigned a zero value while the element  $M_{K,X}$  is transferred into the memory. The procedure continues by scrolling all locations and sending all the effective maxima to the memory. A schematic explanation of the algorithm is shown in figure 22.

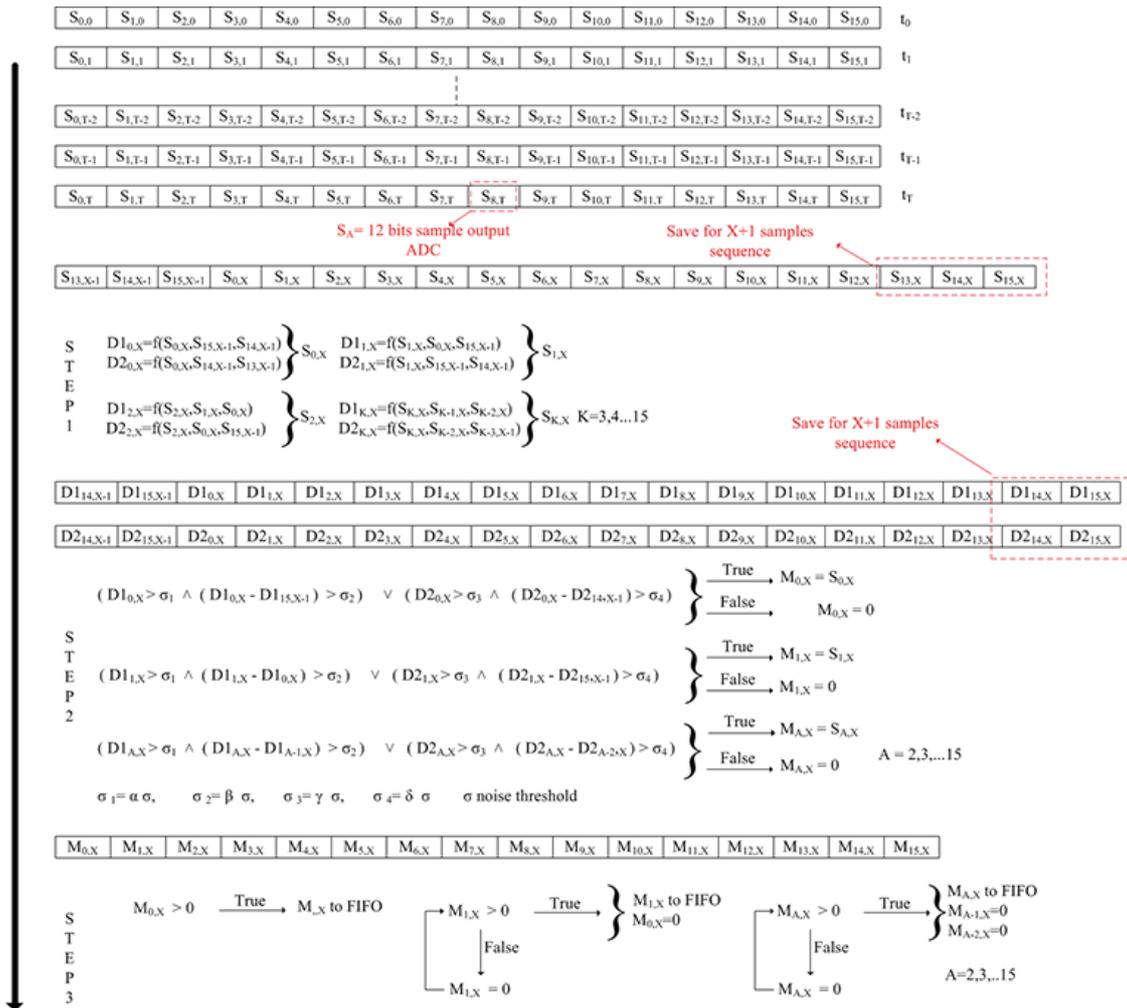
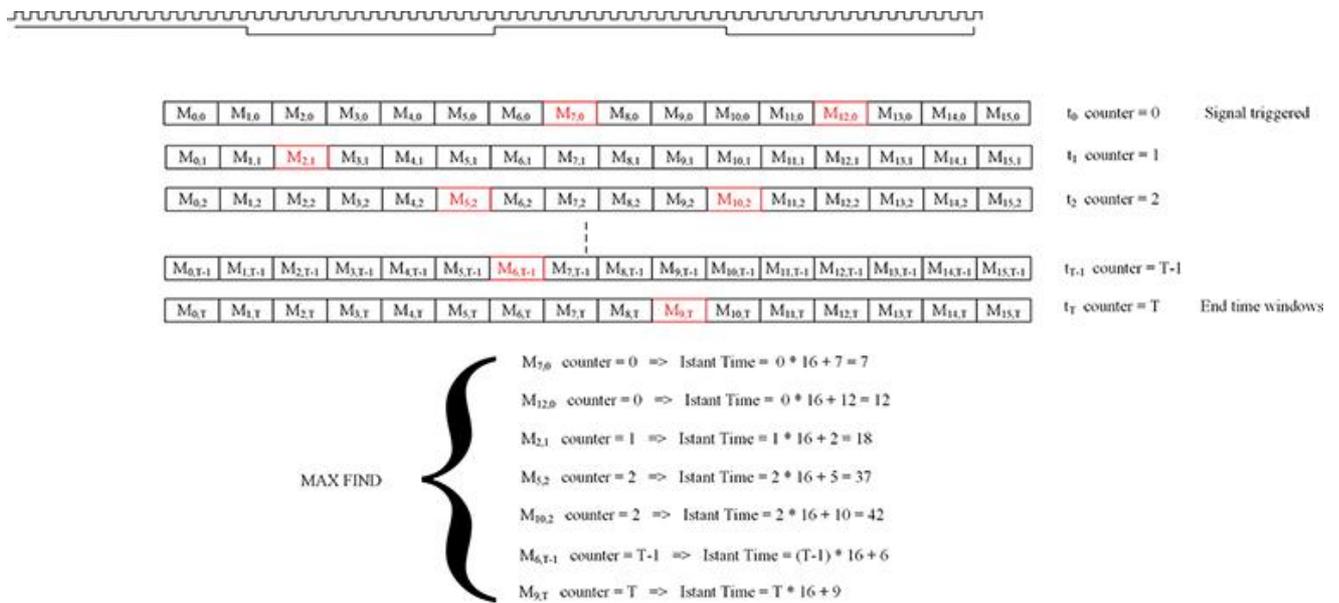


Fig. 22 The CluTim algorithm. [1]

As already mentioned, the other information to be stored in addition to the maxima are the times at which the pulses are detected. Therefore, the time information is provided with a counter, the value of which is stored in a FIFO every time a peak is found. At every time instant  $t_X$ , the value of the counter is updated. When a peak is found the *Instant Time* is calculated as follows:

$$\text{Instant Time} = \text{counter} * 16 + K\text{sample}$$

where the time to be stored (*counter*) is multiplied by 16 in order to consider the correct ADC sampling rate and added to the sample number corresponding to the maximum found [1]. Figure 23 shows how the CluTim algorithm handles the timing information.



**Fig. 23** Timing information. [1]

The memories are continuously filled as new peaks are found.

For the algorithm to start, a trigger signal is needed. The trigger signal is generated by an external device when the particle is sent into the drift chamber. After the trigger signal is generated, the algorithm starts to compute the peaks. After some time, the

particle escapes from the drift chamber, so the algorithm stops and waits for another trigger signal.

So, when a trigger signal occurs at time  $t_0$ , indicating with  $T$  the observed time window, which coincides with the maximum drift time, the reading procedure is enabled and only the data related to the found peaks in the  $[t_0, t_0 + T]$  time interval are transferred to an external device [1].

## **4 FPGA IMPLEMENTATION**

In this chapter different FPGA designs are presented. Each design exploits a different architecture in order to acquire the external analog signal and to find the peaks. The designs are created using the Vivado IP integrator feature, which allows the user to create complex system designs by instantiating and interconnecting IPs from the Vivado IP catalog on a design canvas. The IP Integrator converts the designer's block design into a synthesizable RTL description (Verilog or VHDL) and automates the implementation of the embedded system (from RTL to the bitstream-file). The external analog signal is generated with a waveform generator, then is passed both to an ADC connected to the FPGA and to an oscilloscope. Moreover, in order to see if the FPGA has correctly acquired the external signal, also a DAC has been connected to the FPGA. The output of the DAC is connected to the oscilloscope.

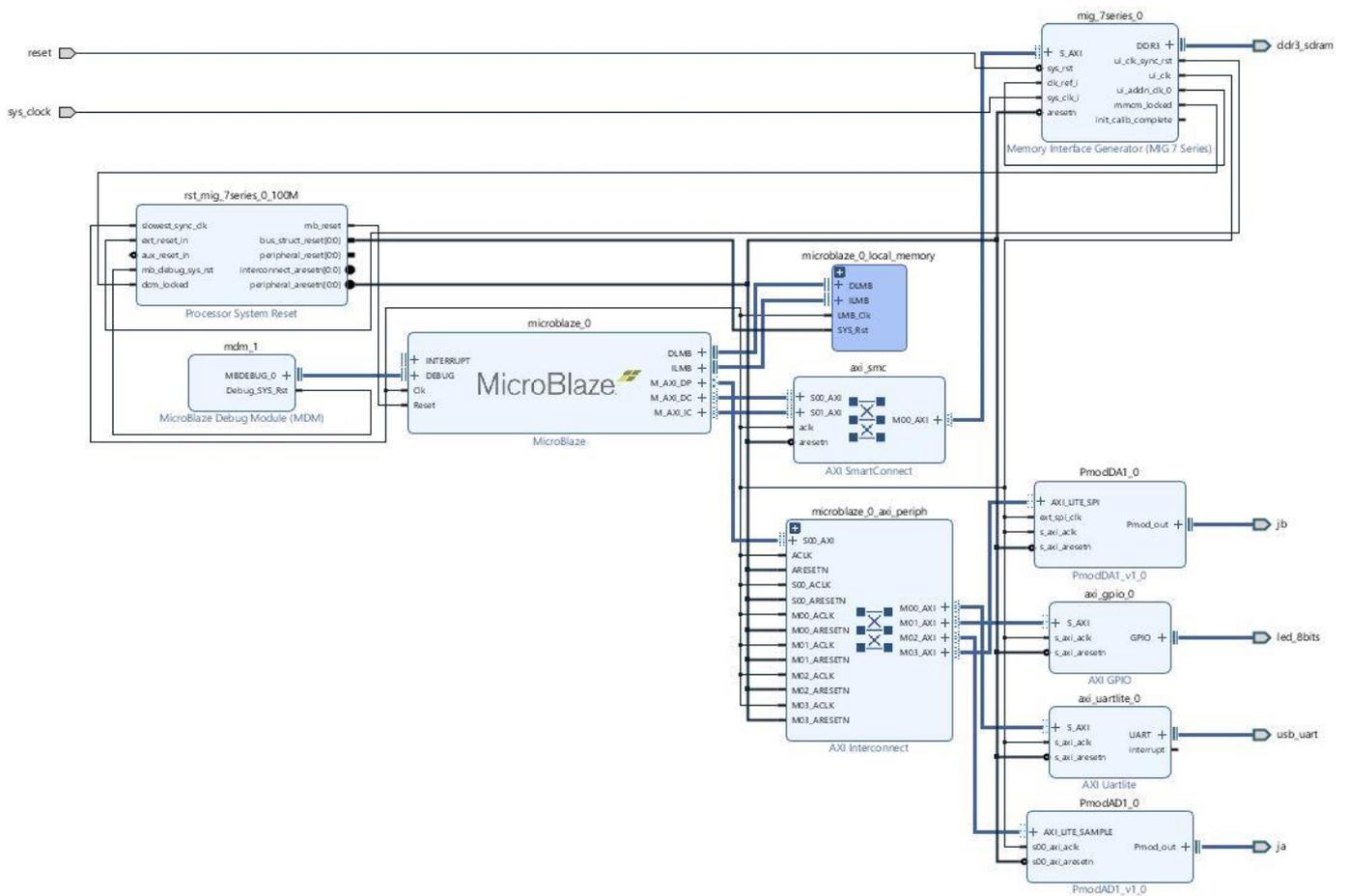
### **4.1 Peak finding algorithm using the Microblaze Core**

In this section the first FPGA design is presented. In this design a simplified peak finding algorithm is entirely implemented on the Microblaze Core. The ADC and the DAC used in the experiments carried out are, respectively, the Digilent Pmod AD1 and the Digilent Pmod DA1. In order to generate the external analog signal the HP 8112A 50 MHz Pulse Generator is used. In order to check the input and the output signals of the FPGA a Tektronix TDS 3014B 100MHz Digital Phosphor Oscilloscope is used.

#### **4.1.1 Vivado implementation**

In this section the implemented design in Vivado is presented in figure 24, while in figure 25 the memory addresses of each component in the design are illustrated. In

order to create the block design illustrated in figure 24, it is necessary to include the “board files” of the FPGA board (Nexys Video) in the Vivado project: in this way Vivado will automatically assign each pin instantiated in the block design to the real pins of the FPGA board. Moreover, in order to instantiate the Pmod AD1 and the Pmod DA1 blocks, it is necessary to include the specific libraries in the Vivado project. This is because the Pmod AD1 and the Pmod DA1 blocks are not implemented in the standard Vivado IP catalog.



**Fig. 24** Implemented design in Vivado.

Cell	Slave Interface	Base Name	Offset Address	Range	High Address
microblaze_0					
Data (32 address bits : 4G)					
PmodAD1_0	AXI_LITE_SAMPLE	Reg0	0x44A0_0000	64K	0x44A0_FFFF
PmodDA1_0	AXI_LITE_SPI	Reg0	0x0001_0000	64K	0x0001_FFFF
axi_gpio_0	S_AXI	Reg	0x4000_0000	64K	0x4000_FFFF
axi_uartlite_0	S_AXI	Reg	0x4060_0000	64K	0x4060_FFFF
microblaze_0_local_memory/dlmb_bram_if_cntlr	SLMB	Mem	0x0000_0000	8K	0x0000_1FFF
mig_7series_0	S_AXI	memaddr	0x8000_0000	512M	0x9FFF_FFFF
Instruction (32 address bits : 4G)					
microblaze_0_local_memory/ilmb_bram_if_cntlr	SLMB	Mem	0x0000_0000	8K	0x0000_1FFF
mig_7series_0	S_AXI	memaddr	0x8000_0000	512M	0x9FFF_FFFF

**Fig. 25** Memory addresses of the implemented design.

The Microblaze (microblaze\_0) block in the implemented design represents the Microblaze Core which has already been described in section 2.2. As shown in figure 24, the Microblaze IP core is connected to all the external peripherals via the AXI Interconnect block (microblaze\_0\_axi\_periph) and it is connected to the memory via the AXI SmartConnect block (axi\_smc). The Microblaze Core communicates with the AXI Interconnect IP via the M\_AXI\_DP interface which is the peripheral data interface, and it can be an AXI4-Lite or AXI4 interface. Conversely, the Microblaze IP communicates with the SmartConnect IP via the M\_AXI\_DC and the M\_AXI\_IC which are the data side cache AXI4 interface and the instruction side cache AXI4 interface, respectively.

The AXI Interconnect IP connects one or more AXI memory-mapped master devices to one or more memory-mapped slave devices. The AXI interfaces conform to the AMBA AXI version 4 specifications from ARM. The Interconnect IP is intended for memory-mapped transfers only; AXI4-Stream transfers are not applicable [19].

Also the AXI SmartConnect IP core connects one or more AXI memory-mapped master devices to one or more memory-mapped slave devices, but the SmartConnect IP is more tightly integrated into the Vivado design environment to automatically

configure and adapt to connected AXI master and slave IP with minimal user intervention [17].

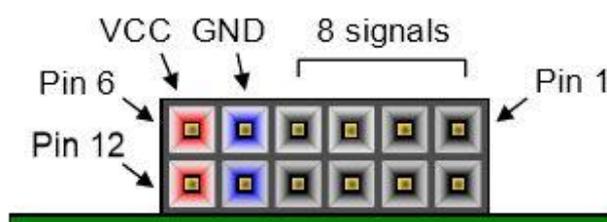
Moreover, the Microblaze Core is connected to the local memory block (microblaze\_0\_local\_memory) via the DLMB and the ILMB which are respectively the data and instruction interfaces. The local memory is used for data and program storage, and it is implemented using the block RAM. The size of the local memory is parametrized and can be between 4 KB and 128 KB [18], in this project the local memory size is 8KB.

Debug is facilitated by the Microblaze Debug Module (MDM).

The DAC is instantiated by the PmodDA1\_v1\_0 block which is connected to the jb pin that represents the JB Pmod port of the Nexys Video board. Moreover, the Pmod DA1 is connected to the AXI Interconnect block via an AXI4-Lite interface which implements the SPI protocol. In this way the Microblaze Core is able to communicate with the DAC via the AXI Interconnect IP core.

The ADC is instantiated by the PmodAD1\_v1\_0 block which is connected to the ja pin that represents the JA Pmod port of the Nexys Video board. The Pmod AD1 is connected to the AXI Interconnect block via an AXI4-Lite interface. So, the Microblaze can communicate with the ADC using the AXI Interconnect IP core.

The Pmod ports are arranged in a 2x6 right-angle, 100-mil female connectors that mate with standard 2x6 pin headers. Each 12-pin Pmod connector provides two power pins (6 and 12), two ground pins (5 and 11), and eight logic signals. The VCC and Ground pins can deliver up to 1A of current [4]. Pin assignments for the Pmod I/O connected to the FPGA are shown in Figure 26.



**Fig. 26** Pmod ports of the Nexys Video board.[4]

The AXI GPIO block (`axi_gpio_0`) provides a general-purpose input/output interface to an AXI4-Lite interface. The AXI GPIO can be configured as either a single or a dual-channel device. The width of each channel is independently configurable [20]. In the design the AXI GPIO block is connected to the `led_8bits` pin which is connected to the 8 LEDs of the Nexys Video board, and it is also connected to the AXI Interconnect IP core with an AXI4-Lite interface in order to communicate with the Microblaze Core. This block has been only used to do some tests on the FPGA before launching the peak-finding algorithm.

The last peripheral connected to the AXI Interconnect is the AXI UartLite IP core which instantiate an UART communication protocol, and it will be used to print the peaks on the UART port of the Nexys Video board. The AXI UartLite IP core is also connected to the `usb_uart` pin.

The Processing System Reset IP core (`rst_mig_7series_0_100M`) provides a mechanism to handle the reset conditions for a given system. The core handles numerous reset conditions at the input and generates appropriate resets at the output. This core generates the resets based upon external or internal reset conditions [21]. In the design the Processing System Reset IP core is used to control the Memory Interface Generator (MIG 7 Series) IP core.

The Memory Interface Generator (MIG 7 Series) IP core creates memory controllers for Xilinx's FPGAs. MIG creates complete customized VHDL RTL source code, pin-out and design constraints for the Nexys Video board, and script files for implementation and simulation. As figure 24 shows, the MIG 7 series block is connected to the `ddr3_sdram` pin which is connected to the external DDR3 memory of the Nexys Video. In fact, the Nexys Video board contains two external memories: a 512MByte volatile DDR3 memory and a 32MiByte nonvolatile Serial Flash device. The Nexys Video includes one Micron MT41K256M16HA-187E DDR3 memory component creating a single rank, 16-bit wide interface. It is routed to a 1.5V-powered high range (HR) FPGA bank with 50 ohm controlled single-ended trace impedance. 50-ohm internal terminations in the FPGA are used to match the trace

characteristics. The MIG 7 Series IP core is also connected to the AXI SmartConnect with an AXI4-Lite interface in order to communicate with the MicroBlaze Core.

The sys\_clock and reset pins are connected respectively to the sys\_clk\_i and sys\_rst ports of the MIG 7 Series block. The ui\_clk port of the MIG 7 Series IP core, which represents a 100 MHz clock, is connected to all the clock ports of the other blocks. Before launching the synthesis, a constraints.xdc file has been created. An .xdc file allows the user to manually assign a specific port or a specific standard to a pin instantiated in the design. The constraints.xdc file contains the following line:

```
set_property IOSTANDARD LVCMOS33 [get_ports sys_clock]
```

which states that LVCMOS33 standard must be assigned to the sys\_clock pin. This constraint was added because both the JB and the system clock pins are in the bank 34 of the FPGA and all the components in the same bank must have the same LVCMOS standard.

After creating the block design and the constraints file, an HDL wrapper must be created for the block design. This process translates the block design into a source file that can be read by the Vivado tools and is used to build the actual design.

Once the HDL wrapper is created, the Vivado synthesis is launched. Synthesis is the process of transforming an RTL-specified design into a gate-level representation. After the Synthesis, all the pins in the block design are assigned to the real pins of the Nexys Video board (Fig. 27, 28, 29).

Name	Direction	Board Part Pin	Board Part Interface	Neg Diff Pair	Package Pin	Fixed	Bank	I/O Std	Vcco	Vref	Drive Strength	Slew Type
↳ All ports (76)												
↳ CLK_SYS_CLOCK_54576 (1)	IN					☑	34	LVCMOS33*	3.300			
↳ Scalar ports (1)												
↳ sys_clock	IN				R4	☑	34	LVCMOS33*	3.300			
↳ ddr3_sdr3ram_54576 (48)	(Multiple)					☑	35	(Multiple)*	1.500	(Multiple)	(Multiple)	FAST
↳ ddr3_sdr3ram_addr (15)	OUT					☑	35	SSTL15*	1.500	0.750		FAST
↳ ddr3_sdr3ram_ba (3)	OUT					☑	35	SSTL15*	1.500	0.750		FAST
↳ ddr3_sdr3ram_ck_p (2)	OUT			ddr3_sdr3ram_ck		☑	35	DIFF_SSTL15	1.500			FAST
↳ ddr3_sdr3ram_cke (1)	OUT					☑	35	SSTL15*	1.500	0.750		FAST
↳ ddr3_sdr3ram_dm (2)	OUT					☑	35	SSTL15*	1.500	0.750		FAST
↳ ddr3_sdr3ram_dq (16)	INOUT					☑	35	SSTL15*	1.500	0.750		FAST
↳ ddr3_sdr3ram_dqs_p (4)	INOUT			ddr3_sdr3ram_dc		☑	35	DIFF_SSTL15	1.500			FAST
↳ ddr3_sdr3ram_odt (1)	OUT					☑	35	SSTL15*	1.500	0.750		FAST
↳ Scalar ports (4)												
↳ ddr3_sdr3ram_cas_n	OUT				K3	☑	35	SSTL15*	1.500	0.750		FAST
↳ ddr3_sdr3ram_ras_n	OUT				J4	☑	35	SSTL15*	1.500	0.750		FAST
↳ ddr3_sdr3ram_reset_n	OUT				G1	☑	35	LVCMOS15*	1.500		12	FAST
↳ ddr3_sdr3ram_we_n	OUT				L1	☑	35	SSTL15*	1.500	0.750		FAST

Fig. 27 I/O pinout for the clock and the DDR3 RAM.

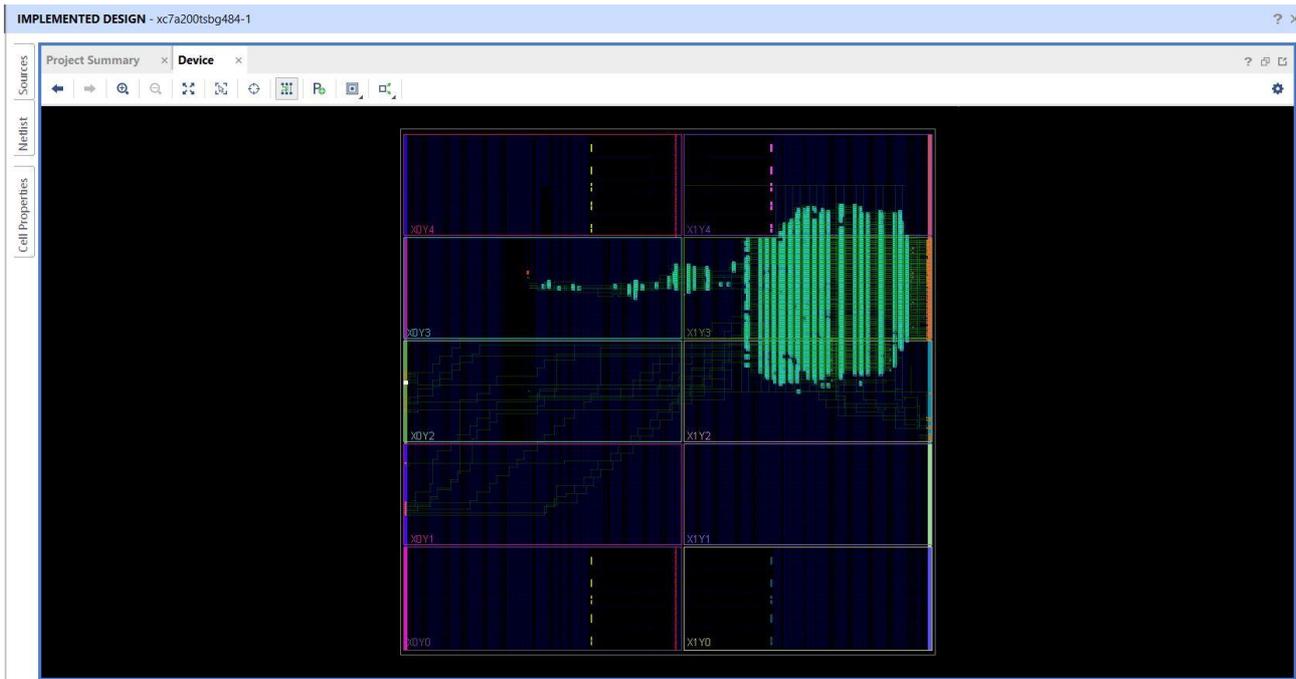
Name	Direction	Board Part Pin	Board Part Interface	Neg Diff Pair	Package Pin	Fixed	Bank	I/O Std	Vcco	Vref	Drive Strength	Slew Type
ja_54576 (8)	INOUT					<input checked="" type="checkbox"/>	14	LVC MOS33*	3.300		12	SLOW
Scalar ports (8)												
ja_pin1_io	INOUT	JA1			AB22	<input checked="" type="checkbox"/>	14	LVC MOS33*	3.300		12	SLOW
ja_pin2_io	INOUT	JA2			AB21	<input checked="" type="checkbox"/>	14	LVC MOS33*	3.300		12	SLOW
ja_pin3_io	INOUT	JA3			AB20	<input checked="" type="checkbox"/>	14	LVC MOS33*	3.300		12	SLOW
ja_pin4_io	INOUT	JA4			AB18	<input checked="" type="checkbox"/>	14	LVC MOS33*	3.300		12	SLOW
ja_pin7_io	INOUT	JA7			Y21	<input checked="" type="checkbox"/>	14	LVC MOS33*	3.300		12	SLOW
ja_pin8_io	INOUT	JA8			AA21	<input checked="" type="checkbox"/>	14	LVC MOS33*	3.300		12	SLOW
ja_pin9_io	INOUT	JA9			AA20	<input checked="" type="checkbox"/>	14	LVC MOS33*	3.300		12	SLOW
ja_pin10_io	INOUT	JA10			AA18	<input checked="" type="checkbox"/>	14	LVC MOS33*	3.300		12	SLOW
jb_54576 (8)	INOUT					<input checked="" type="checkbox"/>	34	LVC MOS33*	3.300		12	SLOW
Scalar ports (8)												
jb_pin1_io	INOUT	JB1			V9	<input checked="" type="checkbox"/>	34	LVC MOS33*	3.300		12	SLOW
jb_pin2_io	INOUT	JB2			V8	<input checked="" type="checkbox"/>	34	LVC MOS33*	3.300		12	SLOW
jb_pin3_io	INOUT	JB3			V7	<input checked="" type="checkbox"/>	34	LVC MOS33*	3.300		12	SLOW
jb_pin4_io	INOUT	JB4			W7	<input checked="" type="checkbox"/>	34	LVC MOS33*	3.300		12	SLOW
jb_pin7_io	INOUT	JB7			W9	<input checked="" type="checkbox"/>	34	LVC MOS33*	3.300		12	SLOW
jb_pin8_io	INOUT	JB8			Y9	<input checked="" type="checkbox"/>	34	LVC MOS33*	3.300		12	SLOW
jb_pin9_io	INOUT	JB9			Y8	<input checked="" type="checkbox"/>	34	LVC MOS33*	3.300		12	SLOW
jb_pin10_io	INOUT	JB10			Y7	<input checked="" type="checkbox"/>	34	LVC MOS33*	3.300		12	SLOW

Fig. 28 I/O pinout for the JA and JB connectors.

Name	Direction	Board Part Pin	Board Part Interface	Neg Diff Pair	Package Pin	Fixed	Bank	I/O Std	Vcco	Vref	Drive Strength	Slew Type
jb_pin3_io	INOUT	JB3			V7	<input checked="" type="checkbox"/>	34	LVC MOS33*	3.300		12	SLOW
jb_pin4_io	INOUT	JB4			W7	<input checked="" type="checkbox"/>	34	LVC MOS33*	3.300		12	SLOW
jb_pin7_io	INOUT	JB7			W9	<input checked="" type="checkbox"/>	34	LVC MOS33*	3.300		12	SLOW
jb_pin8_io	INOUT	JB8			Y9	<input checked="" type="checkbox"/>	34	LVC MOS33*	3.300		12	SLOW
jb_pin9_io	INOUT	JB9			Y8	<input checked="" type="checkbox"/>	34	LVC MOS33*	3.300		12	SLOW
jb_pin10_io	INOUT	JB10			Y7	<input checked="" type="checkbox"/>	34	LVC MOS33*	3.300		12	SLOW
led_8bits_54576 (8)	OUT					<input checked="" type="checkbox"/>	13	LVC MOS33*	3.300		12	SLOW
led_8bits_tri_o (8)	OUT					<input checked="" type="checkbox"/>	13	LVC MOS33*	3.300		12	SLOW
led_8bits_tri_o[7]	OUT	led_8bits_tri_o_7			Y13	<input checked="" type="checkbox"/>	13	LVC MOS33*	3.300		12	SLOW
led_8bits_tri_o[6]	OUT	led_8bits_tri_o_6			W15	<input checked="" type="checkbox"/>	13	LVC MOS33*	3.300		12	SLOW
led_8bits_tri_o[5]	OUT	led_8bits_tri_o_5			W16	<input checked="" type="checkbox"/>	13	LVC MOS33*	3.300		12	SLOW
led_8bits_tri_o[4]	OUT	led_8bits_tri_o_4			V15	<input checked="" type="checkbox"/>	13	LVC MOS33*	3.300		12	SLOW
led_8bits_tri_o[3]	OUT	led_8bits_tri_o_3			U16	<input checked="" type="checkbox"/>	13	LVC MOS33*	3.300		12	SLOW
led_8bits_tri_o[2]	OUT	led_8bits_tri_o_2			T16	<input checked="" type="checkbox"/>	13	LVC MOS33*	3.300		12	SLOW
led_8bits_tri_o[1]	OUT	led_8bits_tri_o_1			T15	<input checked="" type="checkbox"/>	13	LVC MOS33*	3.300		12	SLOW
led_8bits_tri_o[0]	OUT	led_8bits_tri_o_0			T14	<input checked="" type="checkbox"/>	13	LVC MOS33*	3.300		12	SLOW
Scalar ports (0)												
RST.RESET_54576 (1)	IN					<input checked="" type="checkbox"/>	35	LVC MOS15*	1.500			
Scalar ports (1)												
reset	IN	reset			G4	<input checked="" type="checkbox"/>	35	LVC MOS15*	1.500			
usb_uart_54576 (2)	(Multiple)					<input checked="" type="checkbox"/>	14	LVC MOS33*	3.300			
Scalar ports (2)												
usb_uart_rxd	IN	usb_uart_rxd			V18	<input checked="" type="checkbox"/>	14	LVC MOS33*	3.300			
usb_uart_txd	OUT	usb_uart_txd			AA19	<input checked="" type="checkbox"/>	14	LVC MOS33*	3.300		12	SLOW
Scalar ports (0)												

Fig. 29 I/O pinout for the GPIO, reset and the UART.

After the synthesis is completed, the implementation of the design is run. Vivado implementation is the process of mapping the synthesized design to physical resources of the target device in order to create the implemented design. In the device window of Vivado it is possible to see both the placement results and the routing resources in the FPGA in order to implement the synthesized design (Fig. 30).



**Fig. 30** Implemented design on the FPGA.

After the implementation is completed, in order to program the FPGA, it is necessary to generate and load the bitstream into the FPGA. An FPGA bitstream is a file that contains the programming information for an FPGA. A Xilinx FPGA device must be programmed using a specific bitstream for it to behave as an embedded hardware platform.

Once the bitstream file is generated the hardware is exported to Xilinx SDK in order to program the behavior of the Microblaze Core.

### **4.1.2 Xilinx SDK implementation**

In this section the Vivado SDK code is presented. The Xilinx Software Development Kit (SDK) is an Integrated Development Environment (IDE) for development of embedded software applications targeted towards Xilinx embedded processors. SDK works with hardware designs created with Vivado Design Suite. The following C code programs the behavior of the Microblaze Core in the block design.

```

#include <stdio.h>
#include "PmodAD1.h"
#include "sleep.h"
#include "xil_cache.h"
#include "xil_io.h"
#include "xil_types.h"
#include "xparameters.h"
#include "PmodDA1.h"

#define max_size 20 //size of the array which stores the maxima
#define samples 20 //number of samples on which the check is
performed

PmodDA1 myDevice1;
PmodAD1 myDevice;
const float ReferenceVoltage = 3.3;
int i=0,k=0,count=0;
float max[max_size];
float input_data;
float current_max = 0.0;

void DemoInitialize();
void DemoRun();
void DemoCleanup();
void EnableCaches();
void DisableCaches();

int main() {

    EnableCaches();

    AD1_begin(&myDevice, XPAR_PMODAD1_0_AXI_LITE_SAMPLE_BASEADDR);
    DA1_begin(&myDevice1, XPAR_PMODDA1_0_AXI_LITE_SPI_BASEADDR);

    // Wait for AD1 to finish powering on
    usleep(1); // 1 us (minimum)

    AD1_RawData RawData;
    AD1_PhysicalData PhysicalData;

    while (1) {
        if (count < samples){
            for(i = 0;i < samples; i++){
                AD1_GetSample(&myDevice, &RawData);
                AD1_RawToPhysical(ReferenceVoltage, RawData, &PhysicalData);
                DA1_WritePhysicalValue(&myDevice1, PhysicalData[0]);
                input_data = PhysicalData[0];

                if (current_max < input_data){
                    current_max = input_data;}
            }
            max[count] = current_max;
            current_max = 0.0;
            count++;
        } else {

```

```

        for (k = 0; k < samples; ++k) {
            printf("max[%d] = %f\r\n",k , max[k]);
            count = 0;
        }
    }

    DemoCleanup();
    DemoCleanupDA1();

    return 0;
}

void DemoCleanup() {
    DisableCaches();
}

void DemoCleanupDA1() {
    DA1_end(&myDevice);
    DisableCaches();
}

void EnableCaches() {
#ifdef __MICROBLAZE__
#ifdef XPAR_MICROBLAZE_USE_ICACHE
    Xil_ICacheEnable();
#endif
#ifdef XPAR_MICROBLAZE_USE_DCACHE
    Xil_DCacheEnable();
#endif
#endif
}

void DisableCaches() {
#ifdef __MICROBLAZE__
#ifdef XPAR_MICROBLAZE_USE_DCACHE
    Xil_DCacheDisable();
#endif
#ifdef XPAR_MICROBLAZE_USE_ICACHE
    Xil_ICacheDisable();
#endif
#endif
}

```

The header files "PmodAD1.h" and "PmodDA1.h" contain the specific variables and functions used by the PmodAD1 and the PmodDA1 in order to get the samples from the ADC and to send the samples to the DAC.

The header file "xparameters.h" contains the variables in which all the memory addresses are stored. As already shown in section 4.1.1 each IP core in the block

design has a memory address. The "xparameters.h" file contains all the memory addresses of the IP cores connected to the Microblaze Core.

The variables `PmodDA1 myDevice1;` and `PmodAD1 myDevice;` are respectively the DAC and the ADC.

The functions `AD1_begin(&myDevice, XPAR_PMODAD1_0_AXI_LITE_SAMPLE_BASEADDR);` and `DA1_begin(&myDevice1, XPAR_PMODDA1_0_AXI_LITE_SPI_BASEADDR);` initialize the PmodAD1 and the PmodDA1 devices and they take as first argument the object to start and as second argument the memory address of the PmodAD1 and of the PmodDA1, respectively.

The function `AD1_GetSample(&myDevice, &RawData);` captures the most recently read sample from the PmodAD1 IP core and stores it in the RawData variable.

```
void AD1_GetSample(PmodAD1 *InstancePtr, AD1_RawData *RawDataPtr) {
    u32 data;
    data = Xil_In32(InstancePtr->BaseAddress);
    (*RawDataPtr)[0] = data & AD1_DATA_MASK;
    (*RawDataPtr)[1] = (data >> 16) & AD1_DATA_MASK;
}
```

The function `AD1_RawToPhysical(ReferenceVoltage, RawData, &PhysicalData);` converts an AD1 sample to a float value.

```
void AD1_RawToPhysical(float ReferenceVoltage, AD1_RawData RawData,
    AD1_PhysicalData *PhysicalDataPtr) {
    float conversionFactor = ReferenceVoltage / ((1 << AD1_NUM_BITS) - 1);
    (*PhysicalDataPtr)[0] = ((float) RawData[0]) * conversionFactor;
    (*PhysicalDataPtr)[1] = ((float) RawData[1]) * conversionFactor;
}
```

The function `DA1_WritePhysicalValue(&myDevice1, PhysicalData[0]);` computes the integer value corresponding to the physical value by considering the reference value as the one corresponding to the maximum integer value (0xFF). If the integer value is within the accepted range (0 - 0xFF), this function writes the 12-bit value to the DA converter, by writing 8 bits to SPI, and returns the `DACSPI1_ERR_SUCCESS` message.

If the integer value is outside the allowed range, the function does nothing and returns the `DACSPI1_ERR_VAL_OUT_OF_RANGE` message. If the `dReference` function argument is missing, 3.3 value is used as reference value. Moreover, the `DA1_WriteIntegerValue(PmodDA1 *InstancePtr, u8 bIntegerValue)` function, if the value is inside the allowed range (0 - 0xFF), writes the 12 bits value to the DA converter, by writing 16 bits to SPI and returns the `DACSPI1_ERR_SUCCESS` status message. If the value is outside the allowed range, the function does nothing and returns the `DACSPI1_ERR_VAL_OUT_OF_RANGE` message.

```

u8 DA1_WritePhysicalValue(PmodDA1 *InstancePtr, float dPhysicalValue) {
    u8 status;
    float dReference = 3.3; // The value corresponding to the maximum converter
                            // value (reference voltage). If this parameter is
                            // not provided, it has a default value of 3.3.

    u8 wIntegerValue = dPhysicalValue * (float) (1 << DA1_SPI_NO_BITS)
                        / dReference;
    status = DA1_WriteIntegerValue(InstancePtr, wIntegerValue);
    return status;
}

u8 DA1_WriteIntegerValue(PmodDA1 *InstancePtr, u8 bIntegerValue) {
    u8 recv[2];
    u8 bResult = 0;

    recv[0] = DA1_SPI_CTRL_BYTE;
    recv[1] = bIntegerValue;

    if (bIntegerValue < 0 || bIntegerValue >= (1 << DA1_SPI_NO_BITS)) {
        bResult = DA1_SPI_ERR_VAL_OUT_OF_RANGE;
    } else {
        XSpi_Transfer(&InstancePtr->DA1Spi, recv, recv, 2);
    }
    return bResult;
}

```

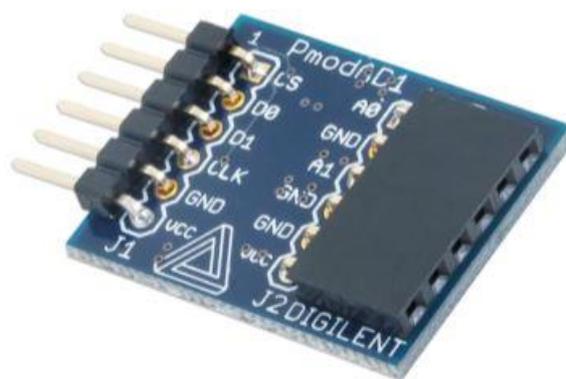
As it is shown in the code, the data transfer on the SPI interface is done with the `XSpi_Transfer(&InstancePtr->DA1Spi, recv, recv, 2);` function which transfers the specified data on the SPI bus. If the SPI device is configured to be a master, this function initiates bus communication and sends/receives the data to/from the selected SPI slave. If the SPI device is configured to be a slave, this function prepares the data to be sent/received when selected by a master. For every byte sent, a byte is received.

So, in the while(1) loop in the main(), each sample coming from the ADC is acquired and converted into a float variable. Every sample is stored in the input\_data variable and confronted with the current max until the array which contains all the peaks (max[max\_size]) is full. When the array max[max\_size] is full the first 'if' statement is not verified so all the peaks found are printed on the UART. Once the array is printed the 'count' variable is set to 0, so a new array is computed.

Once the SDK application is ready, it is launched on the hardware in order to test how the implemented design works.

### 4.1.3 Hardware setup and results

In this section the hardware setup and the experimental results are presented. As already mentioned, the FPGA board used in this project is a Nexys Video board which mounts an Artix-7 FPGA. The ADC used in this work is the Digilent PmodAD1 which is a two channel 12-bit analog-to-digital converter that features Analog Devices AD7476A. The AD7476A is a 12-bit successive-approximation analog to digital converter which contains a track-and-hold amplifier that can handle input frequencies in excess of 13 MHz. The PmodAD1 has a sampling rate of up to 1 million samples per second (Fig. 31) [22].

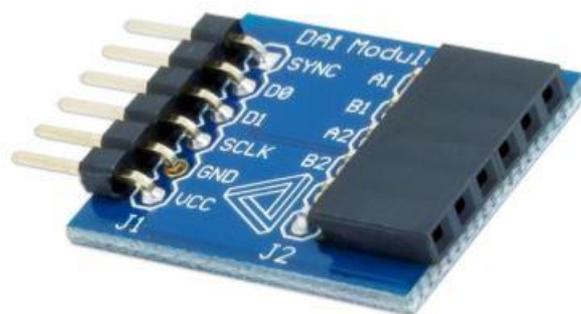


**Fig. 31** PmodAD1 [22]

The PmodAD1 communicates with the host board via an SPI-like communication protocol. The difference between the standard SPI protocol and this protocol is manifested in the pin arrangement on this Pmod. A typical SPI interface would expect a Chip Select, a Master-Out-Slave-In, a Master-in-Slave-Out, and a Serial Clock signal. However, with the two ADCs on this chip, both data lines (MOSI and MISO) are designed to operate only as outputs, making them both Master-In-Slave-Out data lines.

The PmodAD1 will provide its 12 bits of information to the system board through 16 clock cycles with the first four bits consisting of four leading zeroes and the remaining 12 bits representing the 12 bits of the data with the MSB first. The first leading zero is clocked out on the falling edge of the CS signal with all the subsequent bits clocked out on the falling edge of the serial clock signal [22].

The DAC used in this project is the Digilent PmodDA1 which is an 8-bit Digital-to-Analog Converter module that can output up to four different analog signals simultaneously (Fig. 32) [23].



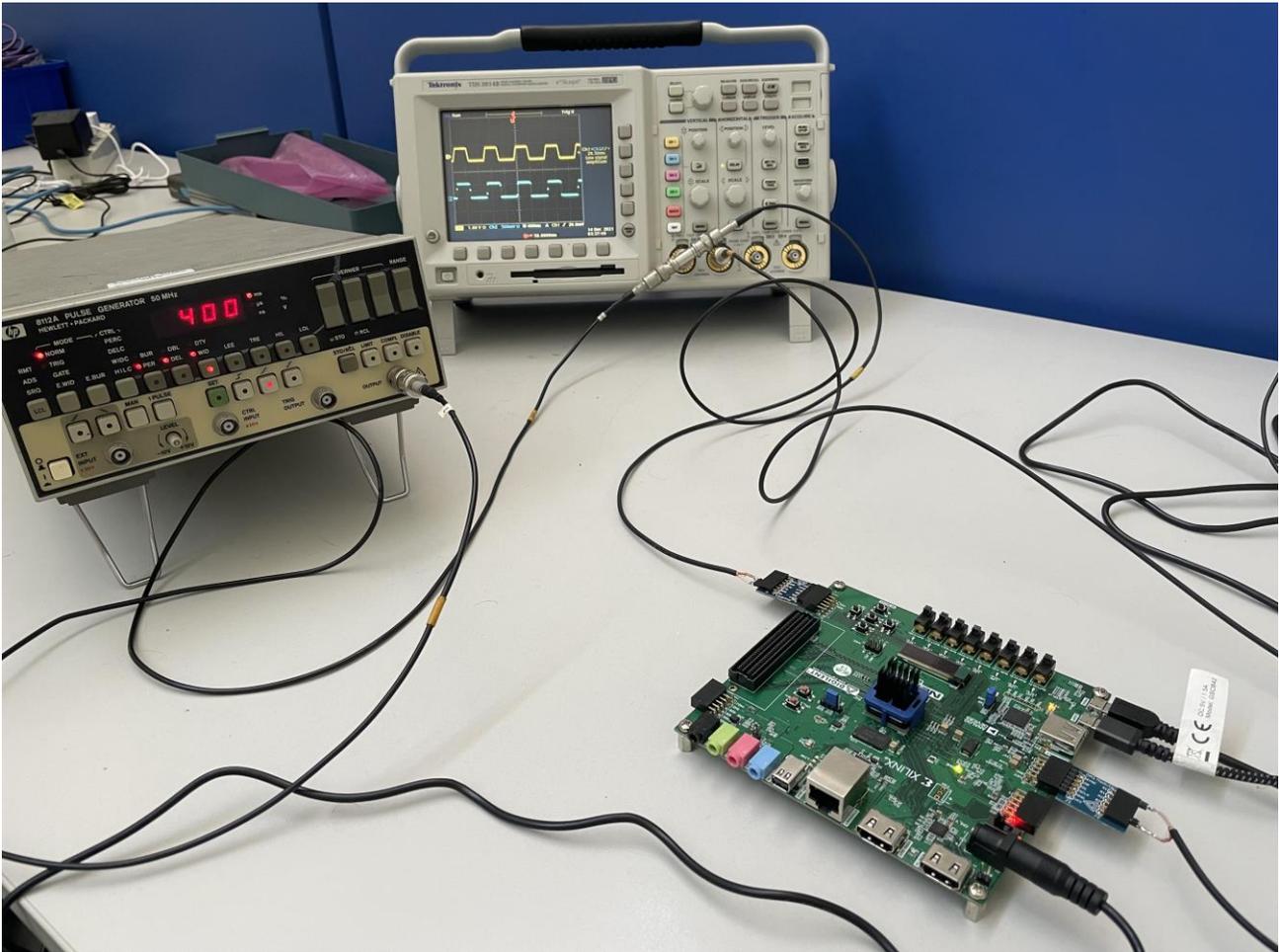
**Fig. 32** PmodDA1 [23]

The PmodDA1 converts an 8-bit digital input signal to a corresponding analog output voltage ranging from 0 to  $V_{dd}$ . Each of the two AD7303s on this Pmod have two 8-bit DACs allowing the user to select which DAC they want their stream of data to go through. Moreover, the AD7303 from Analog Devices operates at clock rates up to 30 MHz, and it is compatible with the SPI interface standard.

The PmodDA1 communicates with the host board via an SPI-like communication protocol. The difference between the standard SPI protocol and this protocol is manifested in the pin arrangement on this Pmod. A typical SPI interface would expect a Chip Select, a Master-Out-Slave-In, a Master-in-Slave-Out, and a Serial Clock signal. However, with the two DACs on this chip, both of the data lines (MOSI and MISO) are designed to operate only as inputs, making them both Master-Out-Slave-In data lines.

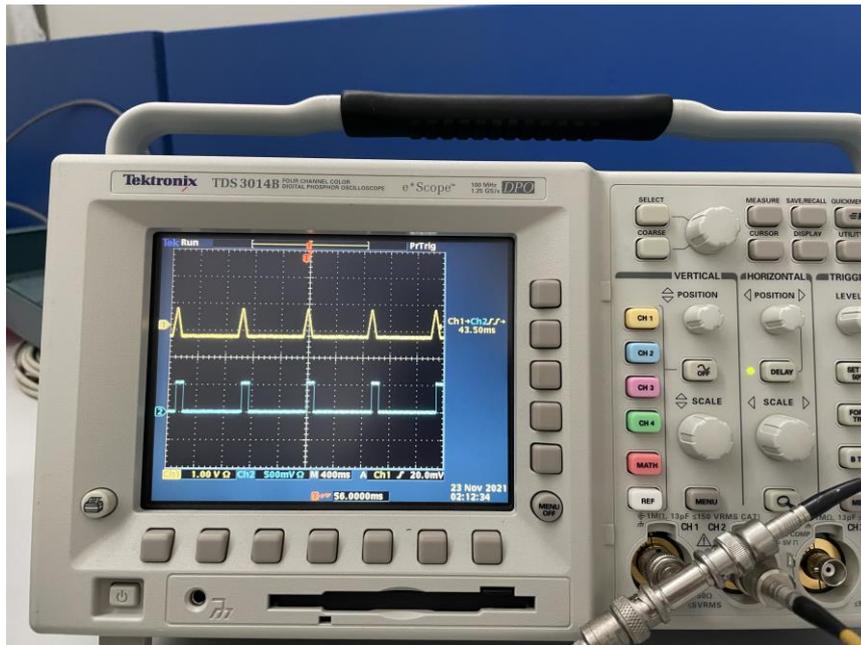
The PmodDA1 will receive its 8 bits of information from the system board through 16 clock cycles with first eight bits consisting of eight control bits and the remaining eight bits representing the 8 bits of the data with the MSB first. Each bit is received by the rising edge of the serial clock line. The function dictated by the first eight control bits is executed when the chip select line is brought high [23].

The output of the pulse generator is connected to the first channel of the oscilloscope and to the A0 pin of the PmodAD1 via a T connector. The PmodAD1 is connected to the JA connector of the Nexys Video board. The PmodDA1 is connected to the JB connector of the board, and the A1 output of the DAC is connected to the second channel of the oscilloscope. The UART port of the FPGA is connected to the COM7 port of the PC. In this way, on the first channel of the oscilloscope the output of the pulse generator is displayed, whereas on the second channel of the oscilloscope the output of the DAC is shown. The hardware setup is shown in figure 33.

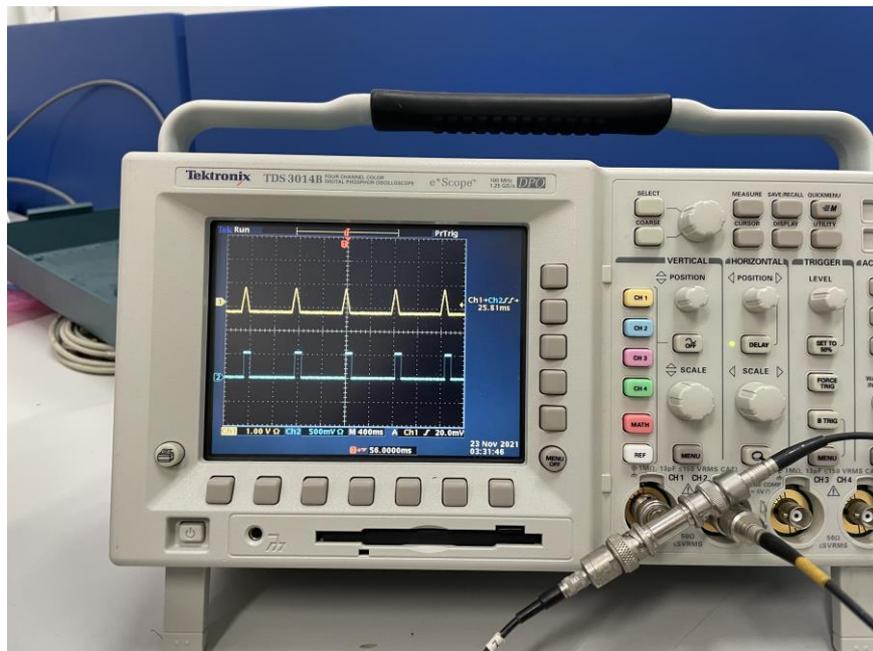


**Fig. 33** Hardware setup.

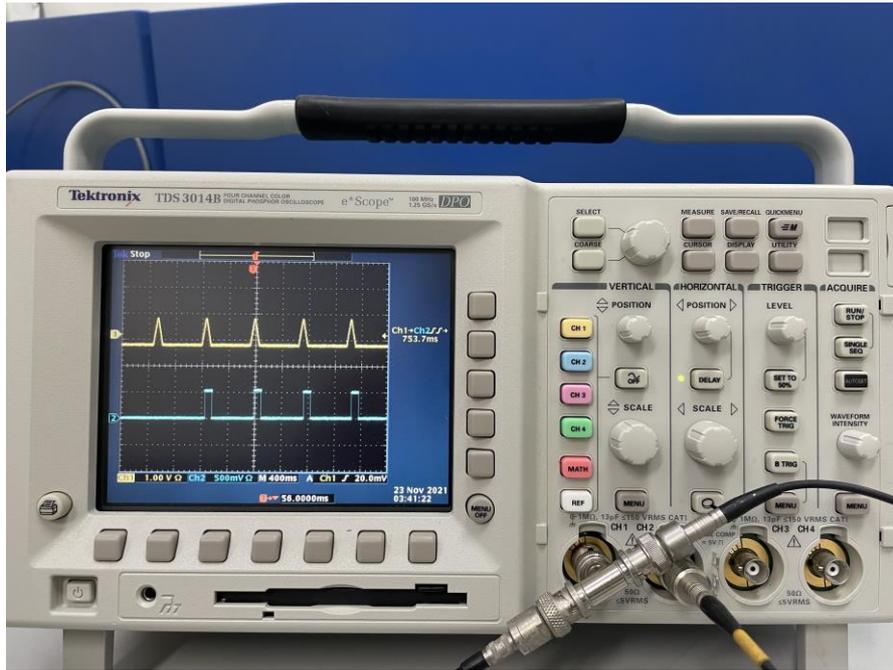
Next are some measurements carried out, which are obtained by changing the parameters on the pulse generator. As already shown in the SDK code, thanks to the infinite ‘while’ loop, the peaks are continuously printed on the COM7 port of the PC when the array that contains the peaks is full. The editable parameters of the pulse generator are the period (PER), the width (WID), the leading and trailing edge (LEE and TRE), the high level and the low level (HIL and LOL). Moreover, on the oscilloscope screen the time delay between channel 1 and channel 2 is displayed.



**Fig. 34** ADC input and DAC output with: PER = 900 ms, WID = 70 ms, LEE = TRE = 50 ms, HIL = 0.5 V, LOL = -0.5V.



**Fig. 35** ADC input and DAC output with: PER = 800 ms, WID = 70 ms, LEE = TRE = 50 ms, HIL = 0.5 V, LOL = -0.5V.

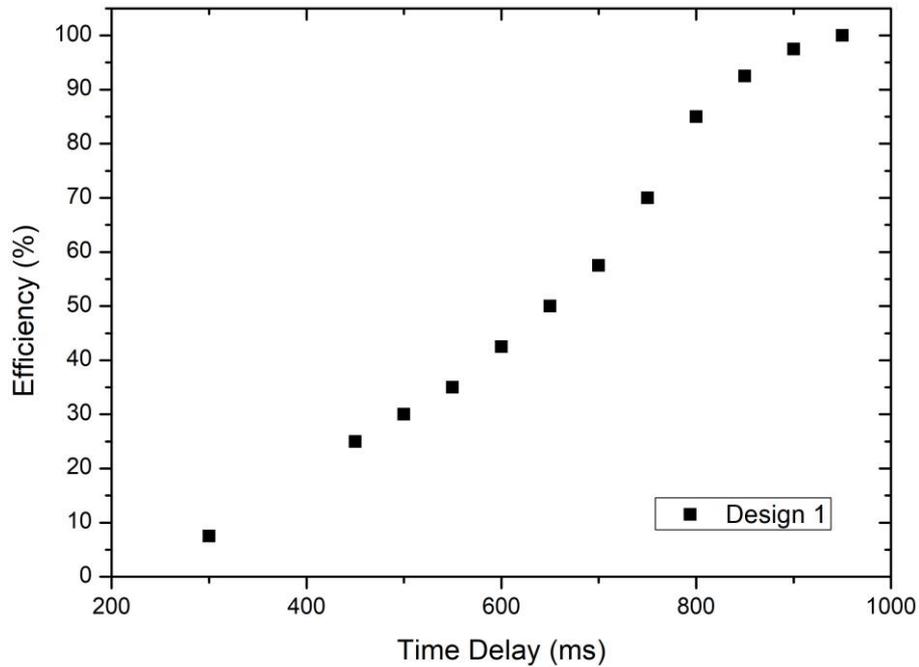


**Fig. 36** ADC input and DAC output with: PER = 700 ms, WID = 70 ms, LEE = TRE = 50 ms, HIL = 0.5 V, LOL = -0.5V.

As shown in figure 36, by reducing the period of the analog input, a peak is missing in the output signal. This means that the FPGA setup was not able to acquire and process all the input peaks. By increasing the frequency of the input signal, more peaks will arrive at the input of the FPGA in the same amount of time. Therefore, the higher the frequency of the input signal, the higher the probability for the FPGA setup to miss some peaks. For this reason, a graph which relates the percentage efficiency to the period of the input signal has been calculated. An observation window of 40 peaks has been considered. By progressively reducing the period of the input analog signal, the acquired peaks are counted with respect to the 40 input peaks. The efficiency has been calculated with the following formula:

$$Eff(\%) = \frac{Output\ peaks}{Input\ peaks} \times 100$$

Of course, when the period is reduced, also the efficiency of the setup decreases. The efficiency versus the time delay between pulses is shown in the following picture:



**Fig. 37** Efficiency versus time delay between pulses.

As the graph shows, efficiency of setup becomes 50% when period is about 650 ms. When the period is less than 300 ms the algorithm is not able to find any peaks.

## 4.2 Peak finding algorithm using Microblaze and a custom IP core

In this section the second FPGA design is presented. In this design a simplified peak finding algorithm is implemented on the Microblaze Core and on a custom IP block created with the Vivado HLS tool. Given a specific number of ADC samples, the created custom IP core is capable of finding the peak among the received ADC samples. The ADC and the DAC used in the experiments carried out are, respectively, the Digilent PmodAD1 and the Digilent PmodDA1.

## 4.2.1 Vivado HLS implementation

The Xilinx Vivado High-Level Synthesis (HLS) tool transforms a C specification into a register transfer level (RTL) implementation that can be synthesized into a Xilinx FPGA. It is possible to write specifications in C or C++ language and the FPGA provides a massively parallel architecture with benefits in performance, cost, and power over traditional processors [24].

The custom IP core created on Vivado HLS calculates the peak among a given number of ADC samples. The implemented C++ code on Vivado HLS is presented:

```
float Max_finder_basic(float adc_sample, float current_max)
{
    #pragma HLS INTERFACE s_axilite port=return
    #pragma HLS INTERFACE s_axilite port=current_max
    #pragma HLS INTERFACE s_axilite port=adc_sample

    if(current_max < adc_sample)
    {
        current_max = adc_sample;
    }
    return current_max;
}
```

So, by comparing the incoming ADC sample with the current peak, this IP core calculates and updates the current peak at every iteration.

The HLS tool provides pragmas that can be used to optimize the design. The HLS INTERFACE pragma specifies how RTL ports are created from the function definition during interface synthesis. The AXI4 interfaces supported by Vivado HLS include the AXI4-Stream (axis), AXI4-Lite (s\_axilite), and AXI4 master (m\_axi) interfaces, which can be specified as follows [24]:

- AXI4-Stream interface: it acts on input arguments or output arguments only, not on input/output arguments;
- AXI4-Lite interface: it acts on any type of argument except arrays, it is possible to group multiple arguments into the same AXI4-Lite interface;

- AXI4 master interface: it acts on arrays and pointers (and references in C++) only; it is possible to group multiple arguments into the same AXI4 interface.

In this Vivado HLS project, only AXI4-Lite interfaces are implemented. An AXI4-Lite interface is implemented for the `adc_sample` port, the `current_max` port and the return port. In this way, the Microblaze Core is able to access the memory address of each port. The Microblaze Core can pass the ADC samples from the memory address of the `PmodAD1` to the AXI4-Lite interface corresponding to the `adc_sample` port and it can get the computed peak by accessing the AXI4-Lite interface corresponding to the `current_max` port. The return port is declared as an AXI4-Lite interface so the custom IP core can be connected to the Microblaze Core via an AXI Interconnect IP. After writing the source code, it is necessary to test its behavior. This step is performed by writing the test bench:

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

float Max_finder_basic(float adc_sample, float current_max);

int main()
{
    float adc_sample[20],current_max;
    srand (time(NULL));
    int num_samples = 20;

    for(int i=0;i<num_samples;i++)
    {
        adc_sample[i] = rand() % 100 * 0.01;

        printf("adc_sample[%d] = %.02f\n",i,adc_sample[i]);
    }

    current_max = Max_finder_basic(adc_sample[0], 0.0);

    for(int j=0;j<num_samples;j++)
    {
        current_max = Max_finder_basic(adc_sample[j], current_max);
    }

    printf("current_max = %.02f\n",current_max);
}
```

After the test bench code is written, the C simulation must be launched in order to test the functioning of the IP block. The results of the C simulation are presented:

```
INFO: [SIM 2] ***** CSIM start *****
INFO: [SIM 4] CSIM will launch GCC as the compiler.
  Compiling ../../../../../../AppData/Roaming/Xilinx/Vivado/test_core.cpp in debug
mode
  Generating csim.exe
adc_sample[0] = 0.48
adc_sample[1] = 0.10
adc_sample[2] = 0.09
adc_sample[3] = 0.93
adc_sample[4] = 0.51
adc_sample[5] = 0.77
adc_sample[6] = 0.23
adc_sample[7] = 0.18
adc_sample[8] = 0.07
adc_sample[9] = 0.60
adc_sample[10] = 0.22
adc_sample[11] = 0.40
adc_sample[12] = 0.77
adc_sample[13] = 0.11
adc_sample[14] = 0.02
adc_sample[15] = 0.39
adc_sample[16] = 0.39
adc_sample[17] = 0.10
adc_sample[18] = 0.70
adc_sample[19] = 0.22
current_max = 0.93
INFO: [SIM 1] CSim done with 0 errors.
INFO: [SIM 3] ***** CSIM finish *****
```

After completing the C simulation, the C synthesis must be launched. The C synthesis is an automated design process that takes an abstract behavioral specification of a digital system and find a register-transfer level structure that realizes the given behavior. After the C synthesis is completed without errors, the C/RTL Cosimulation must be launched in order to verify the RTL output. Once the C/RTL Cosimulation is completed, the Export RTL tool packages the RTL into the desired IP output format.

Now the created IP core can be added to the Vivado block design.

## 4.2.2 Vivado implementation

In this section the implemented design in Vivado is presented (Fig. 38). The memory addresses of each IP block are illustrated in figure 39. In order to create the block design illustrated in figure 38, it is necessary to include the “board files” of the FPGA board (Nexys Video) in the Vivado project. In order to instantiate the PmodAD1 and the PmodDA1 blocks, it is necessary to include the specific libraries. Moreover, it is necessary to include the libraries of the Vivado HLS project in order to add the Max\_finder\_basic IP core to the block design.

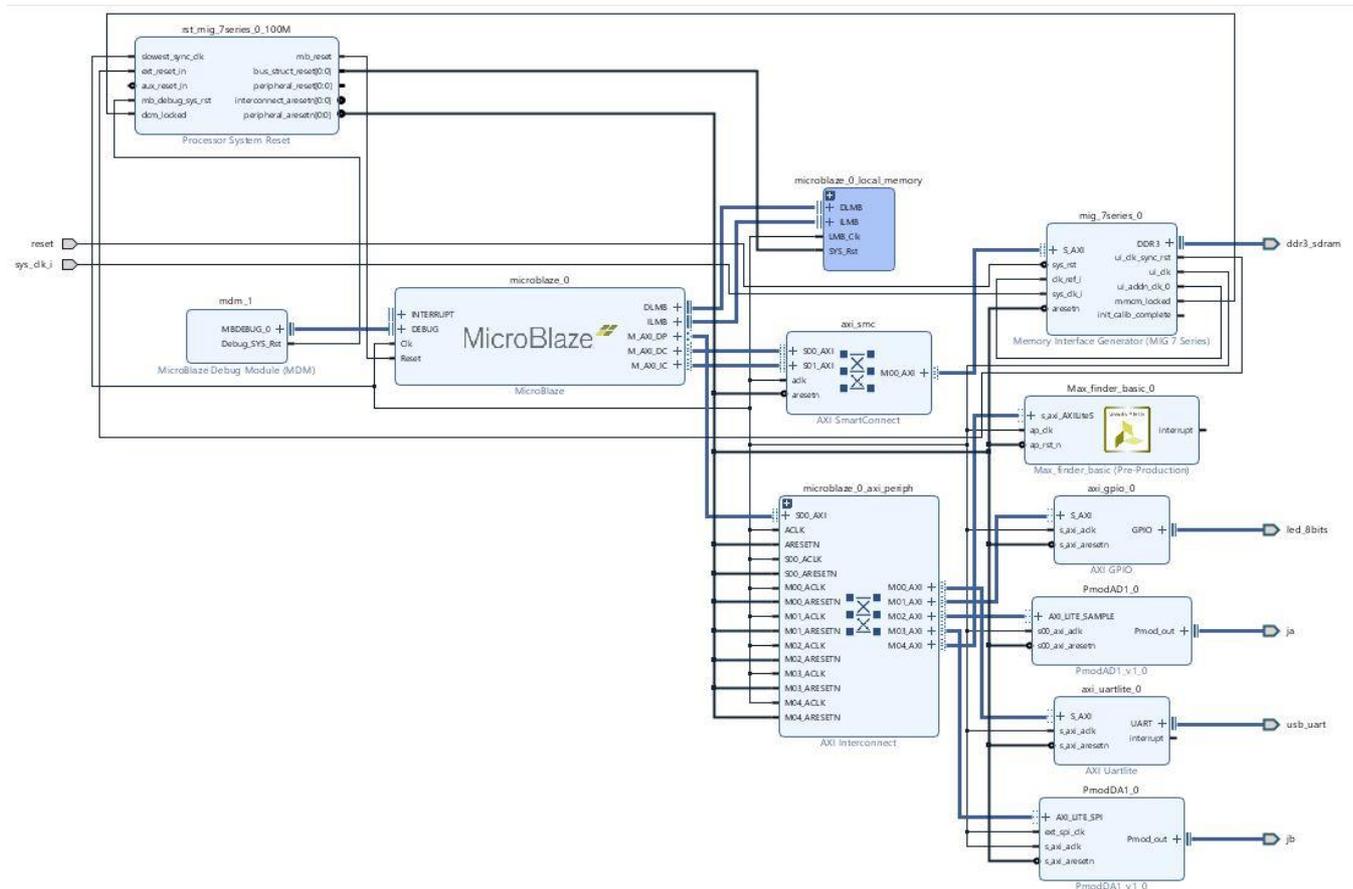


Fig. 38 Vivado implementation.

Cell	Slave Interface	Base Name	Offset Address	Range	High Address
microblaze_0					
Data (32 address bits : 4G)					
Max_finder_basic_0	s_axi_AXILiteS	Reg	0x44A1_0000	64K	0x44A1_FFFF
PmodAD1_0	AXI_LITE_SAMPLE	Reg0	0x44A0_0000	64K	0x44A0_FFFF
PmodDA1_0	AXI_LITE_SPI	Reg0	0x0001_0000	64K	0x0001_FFFF
axi_gpio_0	S_AXI	Reg	0x4000_0000	64K	0x4000_FFFF
axi_uartlite_0	S_AXI	Reg	0x4060_0000	64K	0x4060_FFFF
microblaze_0_local_memory/dlmb_bram_if_cntlr	SLMB	Mem	0x0000_0000	8K	0x0000_1FFF
mig_7series_0	S_AXI	memaddr	0x8000_0000	512M	0x9FFF_FFFF
Instruction (32 address bits : 4G)					
microblaze_0_local_memory/ilmb_bram_if_cntlr	SLMB	Mem	0x0000_0000	8K	0x0000_1FFF
mig_7series_0	S_AXI	memaddr	0x8000_0000	512M	0x9FFF_FFFF

**Fig. 39** Memory addresses.

As shown in figure 38, the Microblaze core is connected to the AXI Interconnect, AXI SmartConnect, local memory, MDM in the same way as the Vivado implementation of the 4.1.1 section.

The MIG 7 Series is connected to the AXI SmartConnect so it can communicate with the Microblaze IP, and to the ddr3\_sdram pin as in 4.1.1. Also the Processor System Reset behaves as in the 4.1.1 design.

The AXI GPIO, PmodAD1, PmodDA1 and AXI UARTlite peripherals are connected to the AXI Interconnect so they can communicate with the Microblaze IP, and to the led\_8bits, ja, jb, usb\_uart pins respectively, as in the 4.1.1 Vivado block design.

The reset and sys\_clk\_i pins are connected in the same way as in the 4.1.1 section.

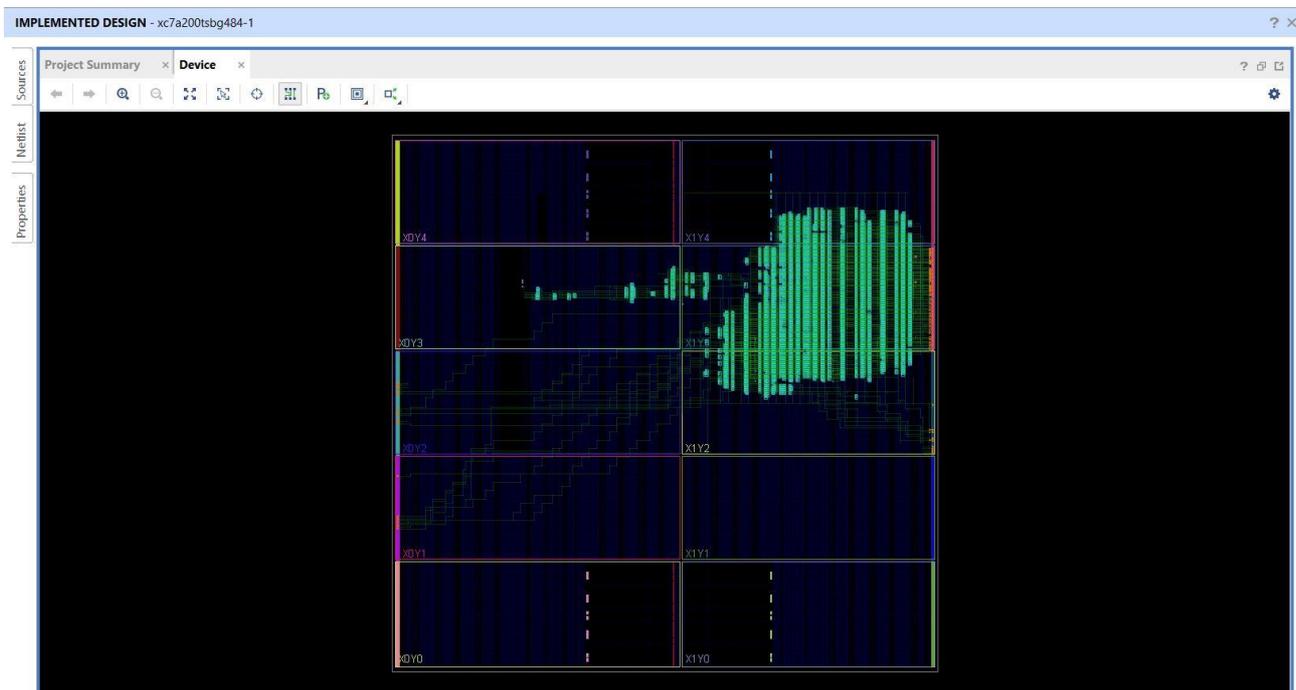
The new feature introduced in this project is the Max\_finder\_basic IP core created in Vivado HLS. It is connected to the AXI Interconnect IP with the AXI4-Lite interface that has been implemented in Vivado HLS with the #pragma commands. So, the AXI4-Lite interface allows the Microblaze IP core to communicate with the custom IP block. In this way, the Microblaze IP can send the ADC samples to the custom IP block and can receive the computed peak trough the AXI4-Lite interface.

For the same reasons as in the 4.1.1 section, a constraints.xdc file is needed with the following command:

```
set_property IOSTANDARD LVCMOS33 [get_ports sys_clock_i]
```

Once the block design is verified, the HDL wrapper is created and the Vivado synthesis is launched. Since the pins used are the same as in the 4.1.1 Vivado implementation, the pinout is the same as the one in the 4.1.1 section.

After the synthesis is completed, the implementation (Fig. 40) and the bitstream file are generated. Then the hardware is exported to the Xilinx SDK tool in order to program the Microblaze Core.



**Fig. 40** Implemented design.

### 4.2.3 Xilinx SDK implementation

In this section the SDK implementation is presented. The Microblaze Core is programmed with the following code:

```
#include <stdio.h>
#include "PmodAD1.h"
#include "sleep.h"
#include "xil_cache.h"
#include "xil_io.h"
#include "xil_types.h"
#include "xparameters.h"
```

```

#include "PmodDA1.h"
#include "xmax_finder_basic.h"

#define max_size 20 //size of the array which stores the maxima
#define samples 20 //number of samples on which the check is
performed

unsigned int float_to_u32(float val)
{
    unsigned int result;
    union float_bytes{
        float v;
        unsigned char bytes[4];
    }data;
    data.v = val;
    result = (data.bytes[3]<<24) + (data.bytes[2]<<16) + (data.bytes[1]<<8) +
(data.bytes[0]);
    return result;
}

float u32_to_float(unsigned int val)
{
    union{
        float val_float;
        unsigned char bytes[4];
    }data;

    data.bytes[3] = (val >> (8*3)) & 0xff;
    data.bytes[2] = (val >> (8*2)) & 0xff;
    data.bytes[1] = (val >> (8*1)) & 0xff;
    data.bytes[0] = (val >> (8*0)) & 0xff;
    return data.val_float;
}

PmodDA1 myDevice1;
PmodAD1 myDevice;

XMax_finder_basic max_finder_basic;
XMax_finder_basic_Config *max_finder_basic_cfg;

const float ReferenceVoltage = 3.3;
int i=0,k=0,count=0;
float max[max_size];
float input_data;
float current_max = 0.0;

void DemoInitialize();
void DemoRun();
void DemoCleanup();
void EnableCaches();
void DisableCaches();

int main() {

```

```

EnableCaches();

u32 current_max_32;
float current_max;

AD1_begin(&myDevice, XPAR_PMODAD1_0_AXI_LITE_SAMPLE_BASEADDR);
DA1_begin(&myDevice1, XPAR_PMODDA1_0_AXI_LITE_SPI_BASEADDR);

XMax_finder_basic_Initialize(&max_finder_basic,
XPAR_XMAX_FINDER_BASIC_0_DEVICE_ID);

XMax_finder_basic_Set_current_max(&max_finder_basic, float_to_u32(0.0));
current_max = u32_to_float(XMax_finder_basic_Get_return(&max_finder_basic));

// Wait for AD1 to finish powering on
usleep(1); // 1 us (minimum)

AD1_RawData RawData;
AD1_PhysicalData PhysicalData;

while (1) {

    if (count < samples){
        for(i = 0;i < samples; i++){
            AD1_GetSample(&myDevice, &RawData);
            AD1_RawToPhysical(ReferenceVoltage, RawData, &PhysicalData);
            DA1_WritePhysicalValue(&myDevice1, PhysicalData[0]);

            XMax_finder_basic_Set_adc_sample(&max_finder_basic,
float_to_u32(PhysicalData[0]));

XMax_finder_basic_Set_current_max(&max_finder_basic, float_to_u32(current_max));

            XMax_finder_basic_Start(&max_finder_basic);

            while(!XMax_finder_basic_IsDone(&max_finder_basic));

            current_max =
u32_to_float(XMax_finder_basic_Get_return(&max_finder_basic));
        }
        max[count] = current_max;
        XMax_finder_basic_Set_current_max(&max_finder_basic, float_to_u32(0.0));
        count++;
    } else {

        for (k = 0; k < samples; ++k) {
            printf("max[%d] = %f\r\n",k , max[k]);
            count = 0;
        }
    }
}

DemoCleanup();
DemoCleanupDA1();

return 0;
}

```

```

void DemoCleanup() {
    DisableCaches();
}

void DemoCleanupDA1() {
    DA1_end(&myDevice);
    DisableCaches();
}

void EnableCaches() {
#ifdef __MICROBLAZE__
#ifdef XPAR_MICROBLAZE_USE_ICACHE
    Xil_ICacheEnable();
#endif
#ifdef XPAR_MICROBLAZE_USE_DCACHE
    Xil_DCacheEnable();
#endif
#endif
}

void DisableCaches() {
#ifdef __MICROBLAZE__
#ifdef XPAR_MICROBLAZE_USE_DCACHE
    Xil_DCacheDisable();
#endif
#ifdef XPAR_MICROBLAZE_USE_ICACHE
    Xil_ICacheDisable();
#endif
#endif
}

```

The PmodAD1 and the PmodDA1 are declared in the same way as in 4.1.2 and the functions used in order to get the samples from the ADC and to send the samples to the DAC are the same as in the 4.1.2 SDK application.

The `"xmax_finder_basic.h"` header file is automatically created by the SDK tool, and it contains all the functions which can be used by the Microblaze to communicate with the Max\_finder\_basic IP core through the AXI4-Lite interface.

The first function needed is the `XMax_finder_basic_Initialize(&max_finder_basic, XPAR_XMAX_FINDER_BASIC_0_DEVICE_ID);` which initializes the Max\_finder\_basic IP core.

The Microblaze is able to send the value of the most recently acquired sample to the Max\_finder\_basic IP core through the AXI4-Lite interface with the function `XMax_finder_basic_Set_adc_sample(&max_finder_basic, float_to_u32(PhysicalData[0]));`

When the function `XMax_finder_basic_Get_return(&max_finder_basic)` is called, the `Max_finder_basic` IP core sends the computed peak to the Microblaze processor.

When the array `max[max_size]` (which contains all the peaks found) is full, the first 'if' statement in the `while(1)` loop is not verified, so the entire array is printed on the UART port. When the array is printed, the 'count' variable is set to 0, so a new array is computed.

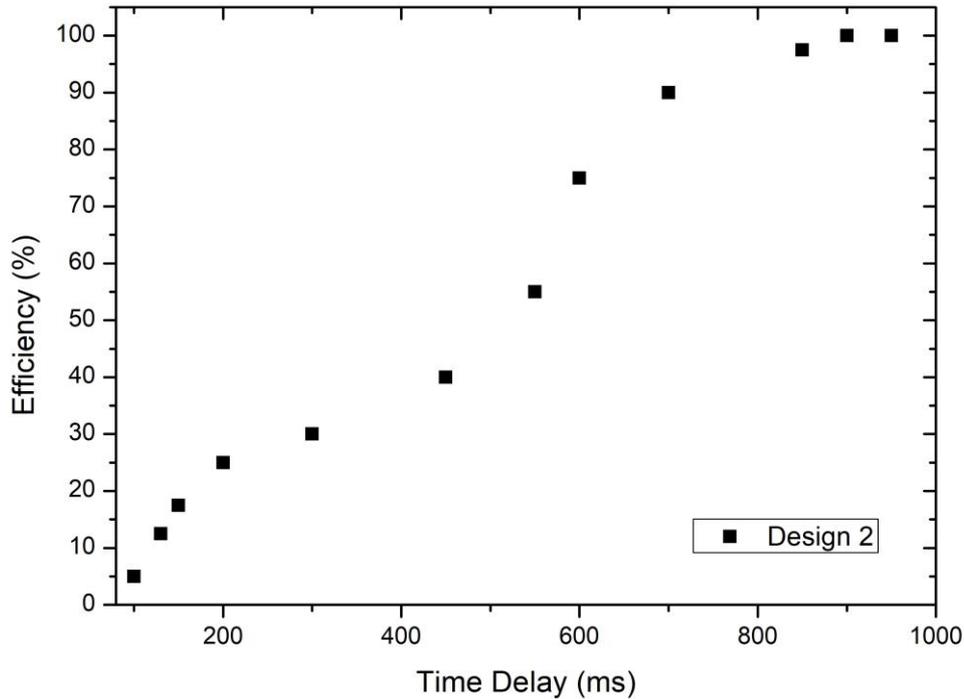
Once the SDK application is ready, it is launched on the hardware in order to test how the implemented design works.

#### **4.2.4 Hardware setup and results**

The hardware setup is the same as the one in section 4.1.3. So, on the first channel of the oscilloscope the output of the waveform generator is displayed, whereas on the second channel of the oscilloscope the analog output of the PmodDA1 is shown.

Next some measurements are carried out, which are obtained by changing the parameters on the pulse generator. As already shown in the SDK code, thanks to the infinite 'while' loop, the peaks are continuously printed on the COM7 port of the PC when the array that contains the peaks is full.

By reducing the period of the analog input, some peaks are missing in the output signal. This means that the FPGA setup was not able to acquire and process all the input peaks. The graph which relates the period of the input signal with the efficiency is shown in the next picture.



**Fig. 41** Efficiency versus time delay between pulses.

As shown, efficiency is 50% for a period of pulses of about 500ms. When the period is less than 100 ms, the setup does not acquire any peaks.

### 4.3 Peak finding algorithm using Microblaze and #pragma BRAMs

In this section the third FPGA design is presented. In this design the simplified peak finding algorithm is implemented on a custom IP block created with the Vivado HLS tool. The Microblaze Core sends an array of 100 ADC samples to the custom IP core and receives the array which contains the 20 peaks found among the ADC samples. Both the ADC samples array and the peaks array are implemented with a #pragma BRAM interface in the Vivado HLS block. The ADC and the DAC used in the experiments carried out are, respectively, the Digilent PmodAD1 and the Digilent PmodDA1.

### 4.3.1 Vivado HLS implementation

In this section the Vivado HLS implementation is presented.

```
void Max_finder_con_BRAM(float adc_data[100],float max_array[20])
{
#pragma HLS INTERFACE s_axilite port=return
#pragma HLS INTERFACE bram port=max_array
#pragma HLS INTERFACE bram port=adc_data

    int count = 0, N = 20;
    float current_max = 0.0;

    while(count < 20)
    {
        for(int i = N - 20; i < N; i++)
        {
            if(current_max < adc_data[i])
                current_max = adc_data[i];
        }
        max_array[count] = current_max;
        count++;
        N++;

        if(N == 100)
            break;
    }
}
```

The implemented IP core receives 100 ADC samples and stores the 20 peaks in the max\_array variable. Both the adc\_data and the max\_array are declared as a #pragma BRAM interface. This is a standard block RAM interface with data, address, chip-enable, and write-enable ports. A BRAM interface is displayed as a single grouped port which can be connected to a Xilinx block RAM using a single point-to-point connection. In this way, the values of adc\_data and max\_array are automatically stored in two different block RAMs.

The return port is declared as an AXI4-Lite interface so the Microblaze Core can communicate with the custom IP block via the AXI Interconnect.

In order to launch the C simulation a test bench code is written.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void Max_finder_con_BRAM(float adc_data[100],float max_array[20]);

int main()
```

```

{
    float adc_data[100];
    float max_array[20];

    for(int i = 0; i < 100; i++)
    {
        adc_data[i] = rand() % 100 * 0.01;

        printf("adc_data[%d] = %.02f\n",i,adc_data[i]);
    }

    Max_finder_con_BRAM(adc_data, max_array);

    for(int j = 0; j < 20; j++)
    {
        printf("max_array[%d] = %.02f\n",j,max_array[j]);
    }
}

```

After the C simulation is completed the C synthesis and the C/RTL Cosimulation are launched. Then the IP block is exported so it can be used in the Vivado block design.

### 4.3.2 Vivado implementation

In this section the implemented design in Vivado is presented (Fig. 42). The memory addresses of each IP block are illustrated in figure 43. In order to create the block design illustrated in figure 42, it is necessary to include the “board files” of the FPGA board (Nexys Video) in the Vivado project. In order to instantiate the PmodAD1 and the PmodDA1 blocks, it is necessary to include the specific libraries. Moreover, it is necessary to include the libraries of the Vivado HLS project in order to add the Max\_finder\_con\_BRAM IP core to the block design.

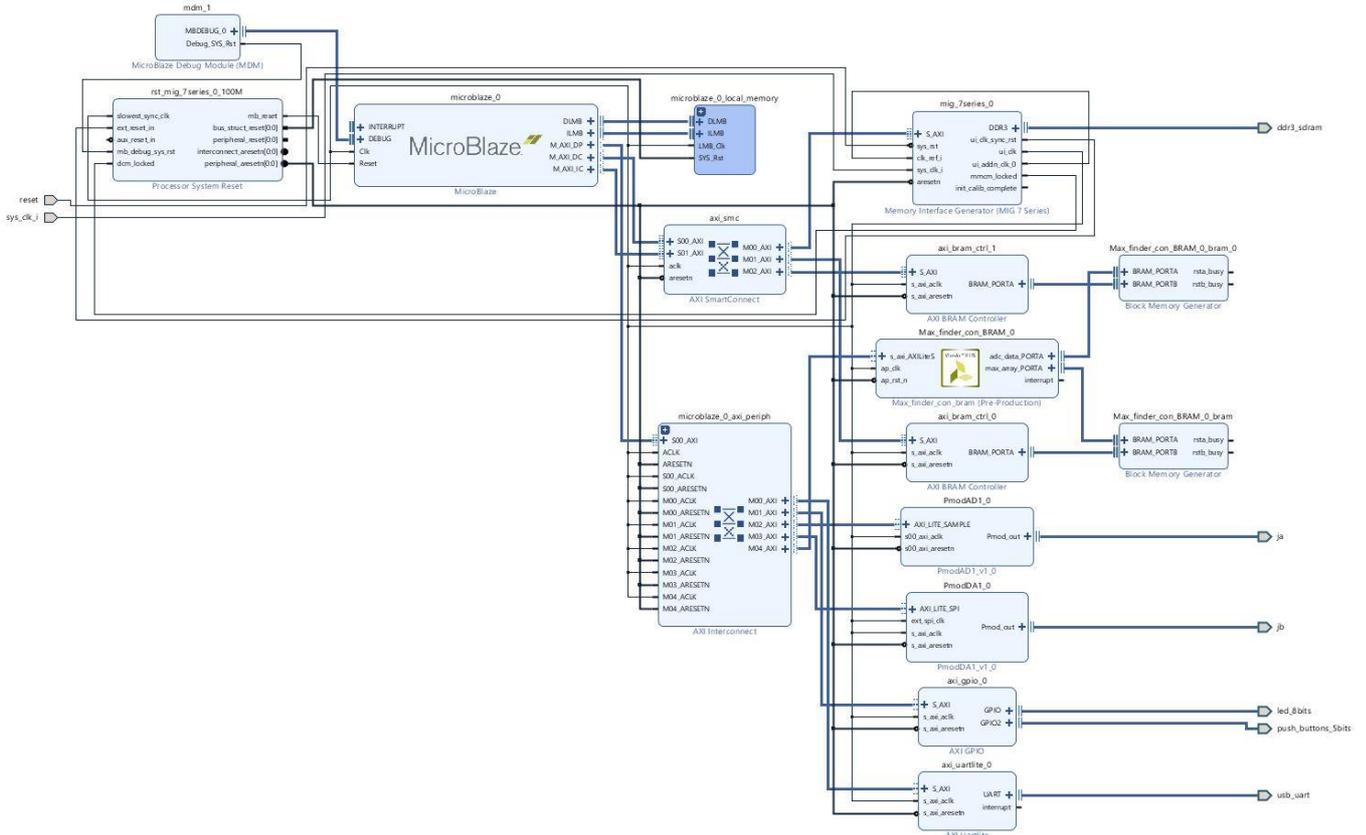


Fig. 42 Vivado implementation.

Cell	Slave Interface	Base Name	Offset Address	Range	High Address
microblaze_0					
Data (32 address bits : 4G)					
Max_finder_con_BRAM_0	s_axi_AXILiteS	Reg	0x44A1_0000	64K	0x44A1_FFFF
PmodAD1_0	AXI_LITE_SAMPLE	Reg0	0x44A0_0000	64K	0x44A0_FFFF
PmodDA1_0	AXI_LITE_SPI	Reg0	0x0001_0000	64K	0x0001_FFFF
axi_bram_ctrl_0	S_AXI	Mem0	0xC000_0000	8K	0xC000_1FFF
axi_bram_ctrl_1	S_AXI	Mem0	0xC200_0000	8K	0xC200_1FFF
axi_gpio_0	S_AXI	Reg	0x4000_0000	64K	0x4000_FFFF
axi_uartlite_0	S_AXI	Reg	0x4060_0000	64K	0x4060_FFFF
microblaze_0_local_memory/dlmb_bram_if_cntrl	SLMB	Mem	0x0000_0000	8K	0x0000_1FFF
mig_7series_0	S_AXI	memaddr	0x8000_0000	512M	0x9FFF_FFFF
Instruction (32 address bits : 4G)					
axi_bram_ctrl_0	S_AXI	Mem0	0xC000_0000	8K	0xC000_1FFF
axi_bram_ctrl_1	S_AXI	Mem0	0xC200_0000	8K	0xC200_1FFF
microblaze_0_local_memory/ilmb_bram_if_cntrl	SLMB	Mem	0x0000_0000	8K	0x0000_1FFF
mig_7series_0	S_AXI	memaddr	0x8000_0000	512M	0x9FFF_FFFF

Fig. 43 Memory addresses.

As shown in figure 42 the Microblaze Core is connected to the AXI Interconnect, AXI SmartConnect, MDM and local memory as in the 4.1.1 project.

The MIG 7 Series IP core is connected to the AXI SmartConnect and to the ddr3\_sdram pin as in the 4.1.1 Vivado implementation.

The peripherals PmodAD1, PmodDA1, AXI UARTlite and AXI GPIO are connected to the AXI Interconnect as in 4.1.1.

The reset and the sys\_clock\_i pins are connected in the same way as in 4.1.1.

The Processor System Reset is connected as in the 4.1.1 block design.

The first new feature introduced in this project is the Max\_finder\_con\_BRAM IP core which is connected to the AXI Interconnect with an AXI4-Lite interface so it can communicate with the Microblaze Core.

As already mentioned, the adc\_data and the max\_array ports of the custom IP block created in Vivado HLS are declared as a BRAM interface, so they are both connected to a block RAM which will store the values contained in the arrays. The BRAMs in the design are instantiated as a two ports BRAMs; this is because one port is connected to the Max\_finder\_con\_BRAM IP core, but the other port is connected to an AXI BRAM Controller. In this way, also the Microblaze Core can read and write to the BRAMs memory addresses. This is possible because the AXI BRAM Controllers are connected to the AXI SmartConnect with an AXI4-Lite interface which allows the Microblaze to communicate with both the BRAMs.

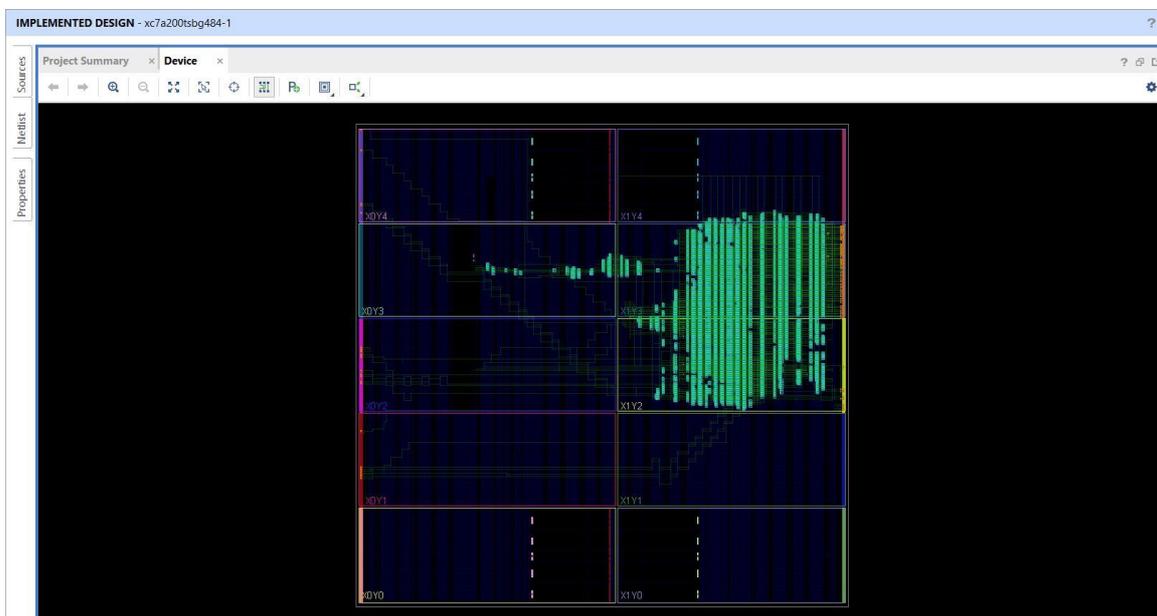
In this way, the Microblaze IP can send the ADC samples to the BRAM associated with the adc\_data port. Then, the Max\_finder\_con\_BRAM custom IP block automatically reads the ADC samples from the adc\_data BRAM interface and finds the peaks. The computed peaks are then stored in the second BRAM associated to the max\_array BRAM interface. Using the AXI BRAM Controller, the Microblaze Core is able to read the peaks from the second BRAM and can print the entire max\_array on the UART port.

For the same reasons as in the section 4.1.1, a constraints.xdc file is needed with the following command:

```
set_property IOSTANDARD LVCMOS33 [get_ports sys_clock_i]
```

Once the block design is verified, the HDL wrapper is created and the Vivado synthesis is launched. Since the used pins are the same as in the 4.1.1 project, except for the push\_buttons\_5bits pin, the pinout is the same as the one in the 4.1.1 with the addition of the D22, D14, C22, F15 and B22 pins which represent the on-board buttons of the Nexys Video board.

After the synthesis is completed, the implementation (Fig. 44) and the bitstream file are generated. Then the hardware is exported to the Xilinx SDK tool in order to program the Microblaze Core.



**Fig. 44** Implemented design.

### 4.3.3 Xilinx SDK implementation

In this section the Xilinx SDK implementation is presented.

```
#include <stdio.h>
#include "PmodAD1.h"
#include "sleep.h"
#include "xil_cache.h"
#include "xil_io.h"
#include "xil_types.h"
#include "xparameters.h"
#include "PmodDA1.h"
```

```

#include "xmax_finder_con_bram.h"
#include "xgpio.h"

PmodDA1 myDevice1;
PmodAD1 myDevice;
const float ReferenceVoltage = 3.3;

XMax_finder_con_bram max_finder;
XMax_finder_con_bram_Config *max_finder_cfg;

XGpio gpio;

float *max_array = (float *) XPAR_BRAM_0_BASEADDR;
float *adc_data = (float *) XPAR_BRAM_1_BASEADDR;

unsigned int float_to_u32(float val)
{
    unsigned int result;
    union float_bytes{
        float v;
        unsigned char bytes[4];
    }data;
    data.v = val;

    result = (data.bytes[3]<<24) + (data.bytes[2]<<16) + (data.bytes[1]<<8) +
(data.bytes[0]);
    return result;
}

void DemoInitialize();
void DemoRun();
void DemoCleanup();
void EnableCaches();
void DisableCaches();
void DemoCleanupDA1();

int main()
{
    u32 btn;

    EnableCaches();

    AD1_begin(&myDevice, XPAR_PMODAD1_0_AXI_LITE_SAMPLE_BASEADDR);
    DA1_begin(&myDevice1, XPAR_PMODDA1_0_AXI_LITE_SPI_BASEADDR);

    XMax_finder_con_bram_Initialize(&max_finder,
XPAR_MAX_FINDER_CON_BRAM_0_DEVICE_ID);

    XGpio_Initialize(&gpio, XPAR_AXI_GPIO_0_DEVICE_ID);

    XGpio_SetDataDirection(&gpio, 2, 0xFFFFFFFF);
}

```

```

// Wait for AD1 to finish powering on
usleep(1); // 1 us (minimum)

AD1_RawData RawData;
AD1_PhysicalData PhysicalData;

    while (1) {

        for (int i = 0; i < 100; i++){
            AD1_GetSample(&myDevice, &RawData); // Capture raw samples

            // Convert raw samples into floats scaled to 0 - VDD
            AD1_RawToPhysical(ReferenceVoltage, RawData,
&PhysicalData);

            //save the values coming from the ADC in the BRAM that
contains the adc data
            adc_data[i] = PhysicalData[0];

            DA1_WritePhysicalValue(&myDevice1, PhysicalData[0]);

        }

        XMax_finder_con_bram_Start(&max_finder);

        while(!XMax_finder_con_bram_IsDone(&max_finder));

        btn = XGpio_DiscreteRead(&gpio, 2);

        if (btn != 0){
            for(int j = 0; j < 20; j ++){
                {
                    printf("max_array[%d] = %.02f\n",j,max_array[j]);
                }
            }
        }

        DemoCleanup();
        DemoCleanupDA1();

        return 0;
    }

void DemoCleanup() {
    DisableCaches();
}

void DemoCleanupDA1() {
    DA1_end(&myDevice);
    DisableCaches();
}

void EnableCaches() {
#ifdef __MICROBLAZE__
#ifdef XPAR_MICROBLAZE_USE_ICACHE
    Xil_ICacheEnable();

```

```

#endif
#ifdef XPAR_MICROBLAZE_USE_DCACHE
    Xil_DCacheEnable();
#endif
#endif
}

void DisableCaches() {
#ifdef __MICROBLAZE__
#ifdef XPAR_MICROBLAZE_USE_DCACHE
    Xil_DCacheDisable();
#endif
#ifdef XPAR_MICROBLAZE_USE_ICACHE
    Xil_ICacheDisable();
#endif
#endif
}

```

In order to write and read from the two BRAMs in the design, the Microblaze Core needs two pointers to the memory base addresses of the two BRAMs. In the implemented SDK application, the pointers: `float *max_array = (float *) XPAR_BRAM_0_BASEADDR;` `float *adc_data = (float *) XPAR_BRAM_1_BASEADDR;` point respectively to the memory base address of the BRAM associated to the `max_array` port and to the BRAM associated to the `adc_data` port of the `Max_finder_con_BRAM` IP block.

The first step in the `main()` function is the initialization of the `PmodAD1`, the `PmodDA1`, the `AXI GPIO` and the `Max_finder_con_BRAM` IP blocks.

Then, the Microblaze Core gets the samples from the `PmodAD1` and converts them in float variables. At the same time the samples coming from the `PmodAD1` are stored in the `adc_data` pointer (in this way the BRAM connected to the `adc_data` port is filled with the ADC samples, since the `adc_data` variable points to the memory address of the BRAM).

When the `Max_finder_con_BRAM` IP block is started, it reads the values in the BRAM connected to the `adc_data` port, it calculates the peaks, and it stores them in the BRAM connected to the `max_array` port. So, since the `max_array` pointer points to the memory address of the BRAM connected to the `max_array` port, the Microblaze Core prints the values (which are the 20 peaks found) stored in the BRAM connected to the `max_array` port.

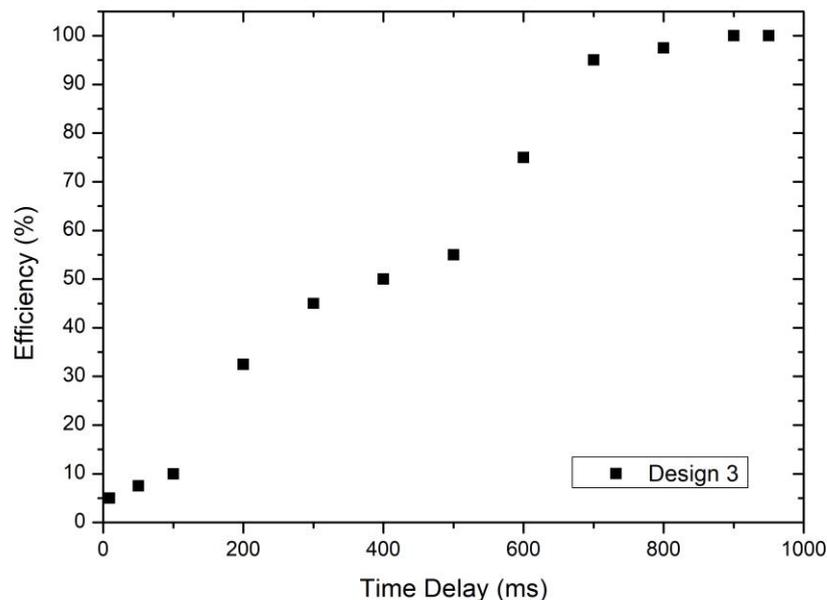
So, when the button is pressed, the peaks are printed on the UART port of the Nexys Video FPGA

#### 4.3.4 Hardware setup and results

In this section the hardware setup and the results are presented. The hardware setup is the same as the 4.1.3 section. So, on the first channel of the oscilloscope the output of the waveform generator is displayed, while on the second channel of the oscilloscope the analog output of the PmodDA1 is shown.

Next are some measurements carried out, which are obtained by changing the parameters on the pulse generator. As already shown in the SDK code, thanks to the infinite ‘while’ loop, the peaks are continuously printed on the COM7 port of the PC when any button of the FPGA board is pressed.

The efficiency versus the time delay for this design is shown in the next picture.



**Fig. 45** Efficiency versus time delay between pulses.

A 50% efficiency corresponds to a period of 400 ms. When the period is below 10 ms, no peaks are found by the setup.

## 4.4 Peak finding algorithm using Microblaze and an AXI DMA

In this section the fourth FPGA design is presented. In this design the peak finding algorithm is implemented on a custom IP block created with the Vivado HLS tool. The Microblaze Core shifts the ADC samples from the PmodAD1 to the DDR3 memory. Then the DMA transfers the ADC samples to the custom IP core through an AXI4-Stream interface. Once the ADC samples are sent to the custom IP block, the peaks are calculated and stored in a BRAM memory with a BRAM interface. After that, the ADC samples are sent back to the DDR3 memory by the AXI DMA using an AXI4-Stream interface. Finally, the Microblaze Core prints the peaks from the BRAM memory on the UART port. The ADC and the DAC used in the experiments carried out are, respectively, the Digilent PmodAD1 and the Digilent PmodDA1.

### 4.4.1 Vivado HLS implementation

In this section the Vivado HLS implementation is presented. This time also a header file is created in order to define a new variable called 'my\_data' which is a struct with two arguments: a float value which represents the ADC sample and a boolean value that represents the TLAST signal which is crucial for the AXI DMA IP core.

```
struct my_data{
    float data;
    bool last;
};
```

Below the source code is presented.

```
#include <hls_stream.h>
#include <ap_axi_sdata.h>
#include "core_demo.h"

void max_find_DMA(hls::stream<my_data> &inStream, hls::stream<my_data>
&outStream, float max[20])
```

```

{
#pragma HLS INTERFACE bram port=max
#pragma HLS INTERFACE s_axilite port=return
#pragma HLS INTERFACE axis port=inStream
#pragma HLS INTERFACE axis port=outStream

    float current_max = 0.0;

    for(int j = 0; j < 20; j++){
        for(int i = 0; i < 20; i++){
#pragma HLS PIPELINE

            my_data valIn = inStream.read();
            my_data valOut;

            if(current_max < valIn.data)
                current_max = valIn.data;

            valOut.data = valIn.data;
            valOut.last = valIn.last;

            outputStream.write(valOut);
        }

        max[j] = current_max;
        current_max = 0.0;
    }
}

```

The inStream and outputStream ports are declared as an AXI4-Stream interface. The inStream port receives data from the AXI DMA, whereas the outputStream port sends the data back to the AXI DMA. The AXI4-Stream interfaces use the my\_data struct. The max array is declared as a BRAM interface, in this way the IP core fills the BRAM memory with the peaks found.

The `#pragma HLS PIPELINE` command reduces the initiation interval for a function or loop by allowing the concurrent execution of operations. The peaks are obtained by comparing the incoming ADC sample with the current max.

The return port is declared as an AXI4-Lite interface so the Microblaze Core can communicate with the custom IP block via the AXI Interconnect.

In order to launch the C simulation a test bench code is written.

```

#include <hls_stream.h>
#include <ap_axi_sdata.h>
#include "core_demo.h"

```

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void max_find_DMA(hls::stream<my_data> &inStream, hls::stream<my_data>
&outStream, float max[20]);

int main()
{
    float max[20];

    hls::stream<my_data> inputStream("input_stream");
    hls::stream<my_data> outputStream("output_stream");

    for(int i = 0; i < 400; i++)
    {
        my_data valIn;
        valIn.data = (rand() % 100) * 0.01;
        printf("In[%d] is %f\n",i,valIn.data);
        inputStream << valIn;
    }

    max_find_DMA(inputStream, outputStream, max);

    for(int j = 0; j < 400; j++)
    {
        my_data valOut;
        outputStream.read(valOut);
        printf("Out[%d] is %f\n",j,valOut.data);
    }

    for(int idx = 0; idx < 20; idx++)
    {
        printf("max[%d] = %f\n",idx,max[idx]);
    }

    return 0;
}

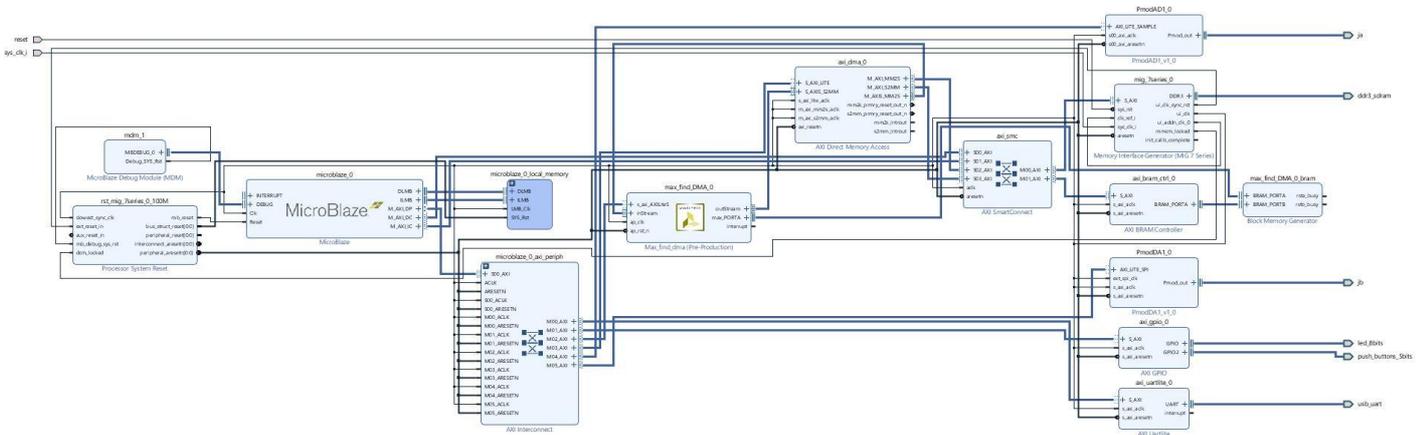
```

After the C simulation is completed the C synthesis and the C/RTL Cosimulation are launched. Then the IP block is exported so it can be used in the Vivado block design.

## 4.4.2 Vivado implementation

In this section the implemented design in Vivado is presented (Fig. 46). The memory addresses of each IP block are illustrated in figure 47. In order to create the block design illustrated in figure 46, it is necessary to include the “board files” of the FPGA board (Nexys Video) in the Vivado project. In order to instantiate the PmodAD1 and

the PmodDA1 blocks, it is necessary to include the specific libraries. Moreover, it is necessary to include the libraries of the Vivado HLS project in order to add the Max\_find\_dma IP core to the block design.



**Fig. 46** Vivado implementation.

Cell	Slave Interface	Base Name	Offset Address	Range	High Address
<b>microblaze_0</b> (Data: 32 address bits : 4G)					
PmodAD1_0	AXI_LITE_SAMPLE	Reg0	0x44A1_0000	64K	0x44A1_FFFF
PmodDA1_0	AXI_LITE_SPI	Reg0	0x0002_0000	64K	0x0002_FFFF
axi_bram_ctrl_0	S_AXI	Mem0	0xC000_0000	8K	0xC000_1FFF
axi_dma_0	S_AXI_LITE	Reg	0x41E0_0000	64K	0x41E0_FFFF
axi_gpio_0	S_AXI	Reg	0x4000_0000	64K	0x4000_FFFF
axi_uartlite_0	S_AXI	Reg	0x4060_0000	64K	0x4060_FFFF
microblaze_0_local_memory/dlmb_bram_if_cntr	SLMB	Mem	0x0000_0000	128K	0x0001_FFFF
max_find_DMA_0	s_axi_AXILiteS	Reg	0x44A0_0000	64K	0x44A0_FFFF
mig_7series_0	S_AXI	memaddr	0x8000_0000	512M	0x9FFF_FFFF
<b>Instruction</b> (32 address bits : 4G)					
axi_bram_ctrl_0	S_AXI	Mem0	0xC000_0000	8K	0xC000_1FFF
microblaze_0_local_memory/ilmb_bram_if_cntr	SLMB	Mem	0x0000_0000	128K	0x0001_FFFF
mig_7series_0	S_AXI	memaddr	0x8000_0000	512M	0x9FFF_FFFF
<b>axi_dma_0</b>					
<b>Data_MM2S</b> (32 address bits : 4G)					
axi_bram_ctrl_0	S_AXI	Mem0	0xC000_0000	8K	0xC000_1FFF
mig_7series_0	S_AXI	memaddr	0x8000_0000	512M	0x9FFF_FFFF
<b>Data_S2MM</b> (32 address bits : 4G)					
axi_bram_ctrl_0	S_AXI	Mem0	0xC000_0000	8K	0xC000_1FFF
mig_7series_0	S_AXI	memaddr	0x8000_0000	512M	0x9FFF_FFFF

**Fig. 47** Memory addresses.

As shown in figure 46 the Microblaze Core is connected to the AXI Interconnect, AXI SmartConnect, MDM and local memory as in the 4.1.1 project.

The MIG 7 Series IP core is connected to the AXI SmartConnect and to the ddr3\_sdram pin as in the 4.1.1 Vivado implementation.

The peripherals PmodAD1, PmodDA1, AXI UARTlite and AXI GPIO are connected to the AXI Interconnect as in 4.1.1.

The reset and the sys\_clock\_i pins are connected in the same way as in 4.1.1. Also, the Processor System Reset is connected as in the 4.1.1 block design.

The first new feature introduced in this project is the AXI Direct Memory Access (DMA). The AXI Direct Memory Access (AXI DMA) IP core provides high-bandwidth direct memory access between the AXI4 memory mapped and AXI4-Stream IP interfaces [25]. The AXI DMA is connected to the AXI Interconnect via an AXI4-Lite interface, so the Microblaze Core is able to communicate with the DMA IP core. Moreover, the AXI DMA is connected to the AXI SmartConnect with two different interfaces:

- M\_AXI\_MM2S: master memory mapped to stream, which takes the data from the DDR3 memory and shifts them to the Max\_find\_dma IP core;
- M\_AXI\_S2MM: stream to memory mapped, which takes the data from the custom IP core and transfers them directly to the DDR3 memory.

The AXI DMA core is connected to the Max\_find\_dma IP core with two AXI4-Stream interfaces:

- S\_AXIS\_S2MM, which takes the data from the IP core to the DMA;
- M\_AXIS\_MM2S, which transfers the data from the DMA to the IP core.

As already mentioned, the IP block created on Vivado HLS implements a BRAM interface for the max port. So, a block RAM is connected to the max\_PORTA interface. An AXI BRAM Controller is also used so the Microblaze Core can read and write on the BRAM memory. In this way, the Microblaze processor can write the samples coming from the PmodAD1 to the DDR3 memory, the DMA can read the data from the DDR3 and can send them to the Max\_find\_dma IP block (which calculates and stores the peaks in the BRAM memory). Finally, the Microblaze can

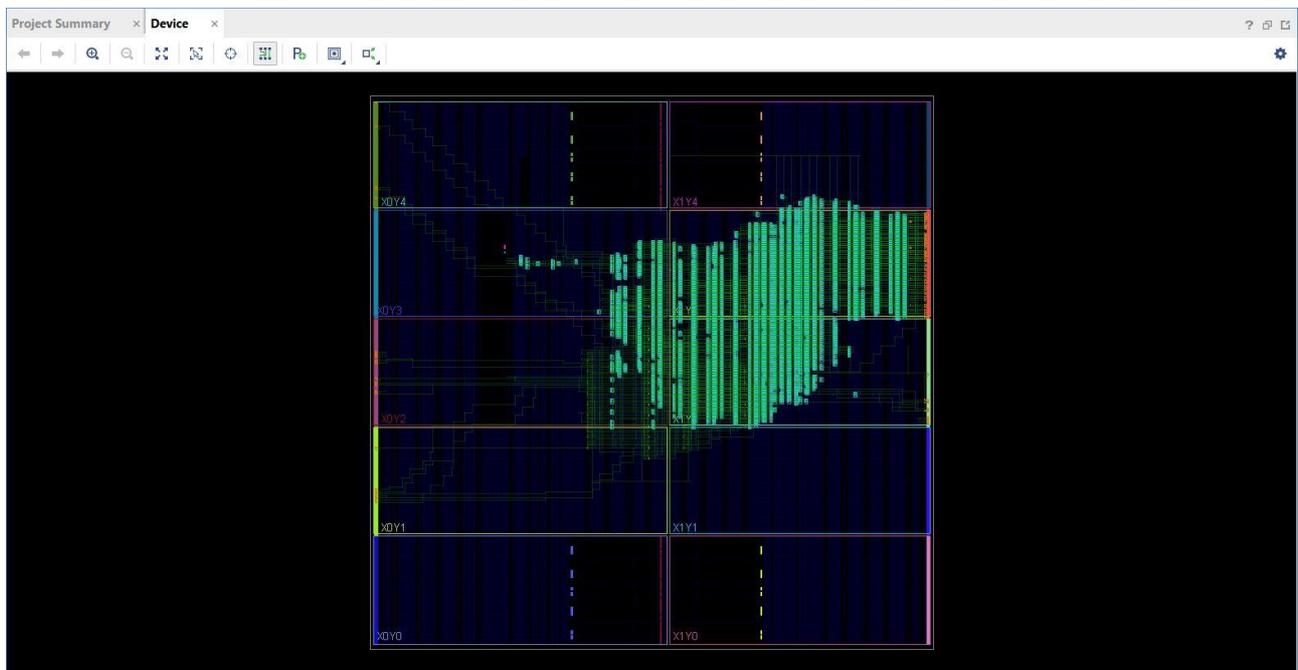
read the peaks stored in the BRAM through the AXI BRAM Controller and print them on the UART port.

For the same reasons as in section 4.1.1, a constraints.xdc file is needed with the following command:

```
set_property IOSTANDARD LVCMOS33 [get_ports sys_clock_i]
```

Once the block design is verified, the HDL wrapper is created and the Vivado synthesis is launched. Since the pins used are the same as in the 4.3.1 project, the pinout is the same as the one in the 4.3.1.

After the synthesis is completed, the implementation (Fig. 48) and the bitstream file are generated. Then the hardware is exported to the Xilinx SDK tool in order to program the Microblaze Core.



**Fig. 48** Implemented design.

### 4.4.3 Xilinx SDK implementation

In this section the SDK application is presented.

```
#include <stdio.h>
#include "xil_printf.h"
#include "PmodAD1.h"
#include "sleep.h"
```

```

#include "xil_cache.h"
#include "xil_io.h"
#include "xil_types.h"
#include "xparameters.h"
#include "PmodDA1.h"
#include "xgpio.h"
#include "xaxidma.h"
#include "xmax_find_dma.h"

#define SIZE_ARR 400
float inStreamData[SIZE_ARR];

//DMA addresses
#define MEM_BASE_ADDRESS 0x81000000
#define TX_BUFFER_BASE (MEM_BASE_ADDRESS + 0x00100000)
#define RX_BUFFER_BASE (MEM_BASE_ADDRESS + 0x00300000)

#define MAX_ARR 20
//Pointer to the BRAM controller
float *max_vec = (float *) XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR;

PmodAD1 myDevice;
PmodDA1 myDevice1;
const float ReferenceVoltage = 3.3;

XAXiDma axiDMA;
XAXiDma_Config *axiDMA_cfg;

//declaration of the custom IP block
XMax_find_dma max_find;
XMax_find_dma_Config *max_find_cfg;

void EnableCaches();
void DisableCaches();
void initPeripherals();

int main()
{
    EnableCaches();

    AD1_begin(&myDevice, XPAR_PMODAD1_0_AXI_LITE_SAMPLE_BASEADDR);
    DA1_begin(&myDevice1, XPAR_PMODDA1_0_AXI_LITE_SPI_BASEADDR);

    AD1_RawData RawData;
    AD1_PhysicalData PhysicalData;

    initPeripherals();

    //pointers to the DMA TX and RX addresses
    float *m_dma_buffer_TX = (float*) TX_BUFFER_BASE;
    float *m_dma_buffer_RX = (float*) RX_BUFFER_BASE;

    XGpio gpio;
    u32 btn;

    XGpio_Initialize(&gpio, XPAR_AXI_GPIO_0_DEVICE_ID);

```

```

        XGpio_SetDataDirection(&gpio, 2, 0xFFFFFFFF); // set BTN GPIO channel tristates
to All Input

        XMax_find_dma_Start(&max_find);

        while(1){

            btn = XGpio_DiscreteRead(&gpio, 2);

            if (btn != 0){
                for( int i = 0; i < SIZE_ARR; i++)
                {
                    // Capture raw samples
                    AD1_GetSample(&myDevice, &RawData);

                    // Convert raw samples into floats scaled to 0 - VDD
                    AD1_RawToPhysical(ReferenceVoltage, RawData,
&PhysicalData[0]);

                    inStreamData[i] = PhysicalData[0];

                    DA1_WritePhysicalValue(&myDevice1, inStreamData[i]);
                }

                //Flush the chache of the buffers
                Xil_DCacheFlushRange((u32)inStreamData,SIZE_ARR*sizeof(float));
                Xil_DCacheFlushRange((u32)m_dma_buffer_RX,SIZE_ARR*sizeof(float));

                //The DMA passes the values of inStreamData from the DDR memory to
the custom IP block

                XAxiDma_SimpleTransfer(&axiDMA, (u32)inStreamData,SIZE_ARR*sizeof(float),XAXIDMA_D
MA_TO_DEVICE);

                //The DMA passes the received data from the custom IP block to the
DDR memory back

                XAxiDma_SimpleTransfer(&axiDMA, (u32)m_dma_buffer_RX,SIZE_ARR*sizeof(float),
XAXIDMA_DEVICE_TO_DMA);

                Xil_DCacheInvalidateRange((u32)m_dma_buffer_RX,SIZE_ARR*sizeof(float));

                for(int k = 0; k < MAX_ARR; k++)
                {
                    //print the array which contains the peaks
                    printf("max[%d] = %.02f\n",k,max_vec[k]);
                }
            }

            DisableCaches();
            return 0;
        }

void EnableCaches() {
#ifdef __MICROBLAZE__
#ifdef XPAR_MICROBLAZE_USE_ICACHE

```

```

    Xil_ICacheEnable();
#endif
#ifdef XPAR_MICROBLAZE_USE_DCACHE
    Xil_DCacheEnable();
#endif
#endif
}

void DisableCaches() {
#ifdef __MICROBLAZE__
#ifdef XPAR_MICROBLAZE_USE_DCACHE
    Xil_DCacheDisable();
#endif
#ifdef XPAR_MICROBLAZE_USE_ICACHE
    Xil_ICacheDisable();
#endif
#endif
}

void initPeripherals()
{
    axiDMA_cfg = XAxiDma_LookupConfig(XPAR_AXIDMA_0_DEVICE_ID);

    if(axiDMA_cfg)
    {
        int status = XAxiDma_CfgInitialize(&axiDMA,axiDMA_cfg);
        if(status != XST_SUCCESS)
        {
            printf("failed\n");
        }
    }

    max_find_cfg = XMax_find_dma_LookupConfig(XPAR_MAX_FIND_DMA_0_DEVICE_ID);

    if(max_find_cfg)
    {
        int status1 =
XMax_find_dma_CfgInitialize(&max_find,max_find_cfg);
        if(status1 != XST_SUCCESS)
        {
            printf("failed\n");
        }
    }
}

```

At first the memory addresses of the DMA and a pointer to the BRAM base address are defined. Then the PmodAD1, the PmodDA1 and the AXI GPIO are initialized. With the function `initPeripherals()`; the DMA and the Max\_find\_dma IP core are initialized. In the `while(1)` loop, a trigger signal is given when any on-board button is pressed. After the button is pushed, the Microblaze Core gets the ADC samples, converts them in float variables and sends them to the DAC. At the same time, the

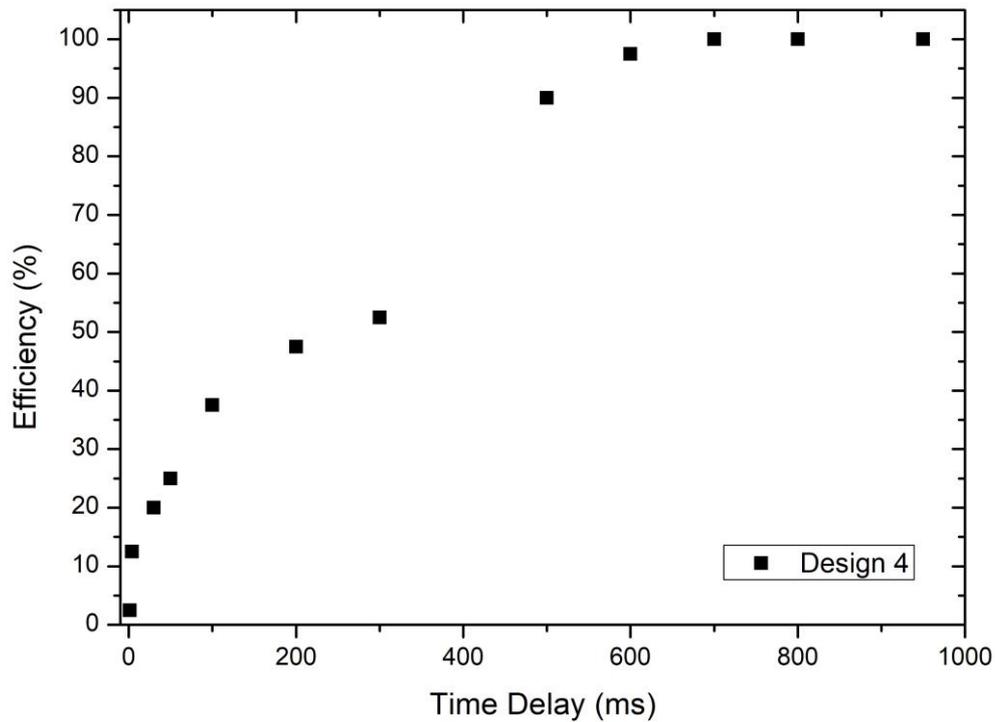
samples coming from the PmodAD1 are stored in the `inStreamData` array, which is passed from the DDR3 to the DMA with the function `XAxiDma_SimpleTransfer (&axiDMA, (u32)inStreamData, SIZE_ARR*sizeof(float), XAXIDMA_DMA_TO_DEVICE);`. The AXI DMA then transfers automatically the ADC samples to the `Max_find_dma` IP core which computes the peaks and stores them in the BRAM memory via the BRAM interface. Then, the ADC samples are transferred back from the custom IP core to the DDR3 by the AXI DMA with the function `XAxiDma_SimpleTransfer(&axiDMA, (u32)m_dma_buffer_RX, SIZE_ARR*sizeof(float), XAXIDMA_DEVICE_TO_DMA);`. Then the Microblaze Core prints the peaks found (which have been stored in the BRAM) on the UART port.

#### **4.4.4 Hardware setup and results**

In this section the hardware setup and the results are presented. The hardware setup is the same as the 4.1.3 section. So, on the first channel of the oscilloscope the output of the waveform generator is displayed, while on the second channel of the oscilloscope the analog output of the PmodDA1 is shown.

Next are some measurements carried out, which are obtained by changing the parameters on the pulse generator. As already shown in the SDK code, thanks to the infinite ‘while’ loop, the peaks are continuously printed on the COM7 port of the PC when any button of the FPGA board is pressed.

Efficiency versus time delay is shown in the next picture.



**Fig. 49** Efficiency versus time delay between pulses.

A 50% efficiency is provided with a 250 ms period of the input signal. Efficiency drops when the period is less than 500 ms, and when the period is less than 1 ms the setup misses all the input peaks.

#### **4.5 Peak finding algorithm implemented in hardware**

In this section the fifth FPGA design is presented. In this design the simplified peak finding algorithm is entirely implemented on a custom IP block created with the Vivado HLS tool. The custom IP core gets the ADC samples directly from the ADC memory address without using resources of the Microblaze Core. Then, the custom IP block calculates the peaks and sends them directly to the DDR3 memory without using CPU software resources. The Microblaze Core just converts the ADC samples and the peaks found in readable values and prints them. The ADC and the DAC used

in the experiments carried out are the Digilent PmodAD1 and the Digilent PmodDA1, respectively. In order to generate the external analog signal a trimmer is used. In order to check the output signals of the FPGA an Arduino UNO microcontroller is used.

## 4.5.1 Vivado HLS implementation

In this section the Vivado HLS implementation is presented.

```
#include <stdlib.h>
#include <inttypes.h>
#include <string.h>

#define MAX_ARRAY_SIZE 5    //size of the max array
#define NUM_SAMPLES 20     //window of sample that are checked

void max_find_ADC(uint16_t *adc_addr, uint16_t *out)
{
#pragma HLS INTERFACE m_axi depth=5 port=out offset=slave bundle=OUT_BUS
#pragma HLS INTERFACE m_axi depth=1 port=adc_addr offset=slave bundle=ADC_BUS
#pragma HLS INTERFACE s_axilite port=return

    static uint16_t input_data[1];
    static uint16_t current_max = 0;
    static uint8_t count = 0;
    static uint8_t max_count = 0;
    static uint16_t max[MAX_ARRAY_SIZE];

    memcpy(&input_data,adc_addr,sizeof(uint16_t));

    if(current_max < input_data[0])
        current_max = input_data[0];

    if(count == NUM_SAMPLES - 1)
    {
        max[max_count] = current_max;
        count = 0;
        current_max = 0;
        if(max_count == MAX_ARRAY_SIZE - 1)
        {
            max_count = 0;
            memcpy(out,max,MAX_ARRAY_SIZE*sizeof(uint16_t));
        } else {
            max_count++;
        }
    } else {
        count ++;
    }
}
```

The ports of the implemented IP core are the `adc_addr`, which will be connected to the memory address of the PmodAD1 through the AXI Interconnect IP core, and the out port which will be connected to the DDR3 memory through the AXI SmartConnect IP. The `adc_addr` port is declared as a 16 bits variable because the PmodAD1 converts the analog input in a digital output of 32 bits divided into two 16-bits segments. Since the PmodAD1 has two analog to digital converters (A0->D0 and A1->D1) the first 16 bits segment represents the digital value coming from the D0 channel and the following 16 bits represent the digital value of the D1 channel. So, since in this project the ADC samples are taken directly from the PmodAD1 IP core, its data format must be respected. The `adc_addr` and the out ports are declared as an AXI4-Master interface; in this way they can directly access the data contained in the memory addresses connected to the interface. With the BUNDLE option it is possible to rename the AXI4-Master ports in the visual representation of the IP block in Vivado. In this way, the AXI4-Master ports will be named `m_axi_ADC_BUS` and `m_axi_OUT_BUS` in the IP block.

The return port is declared as an AXI4-Lite interface so the Microblaze Core can set the PmodAD1 and the DDR3 memory addresses in order to be accessed by the AXI4-Master interfaces.

The `input_data` variable is declared as a two element array because the C++ code written in Vivado HLS is then transformed in hardware, and in hardware programming each accessed variable needs to have specified dimensions.

The variables are declared as static, so they maintain the previous value at every new iteration.

With the ‘`memcpy`’ command, the data from the memory address connected to the AXI4-Master interface is copied to the variable passed as the first argument of the function.

In order to launch the C simulation a test bench code is written.

```
#include <stdlib.h>
#include <inttypes.h>
#include <string.h>
#include <stdio.h>
```

```

#include <time.h>
#include <malloc.h>

void max_find_ADC(uint16_t *adc_addr, uint16_t *out);

#define MAX_ARRAY_SIZE 5 //size of the max array
#define NUM_SAMPLES 20 //window of sample that are checked

int main()
{
    uint16_t *adc_addr;
    uint16_t *out;

    adc_addr = (uint16_t*) malloc(sizeof(uint16_t)*64);
    out = (uint16_t*) malloc(sizeof(uint16_t)*64);

    for(int i = 0; i < (MAX_ARRAY_SIZE * NUM_SAMPLES); i++) //in this for
loop the variable "i" ranges from 0 to MAX_ARRAY_SIZE * NUM_SAMPLES
    {
        adc_addr[0] = rand() % 100 + 20;
        printf("adc_addr[%d] = %lu\n",i,adc_addr[0]);
        max_find_ADC(adc_addr,out);
    }

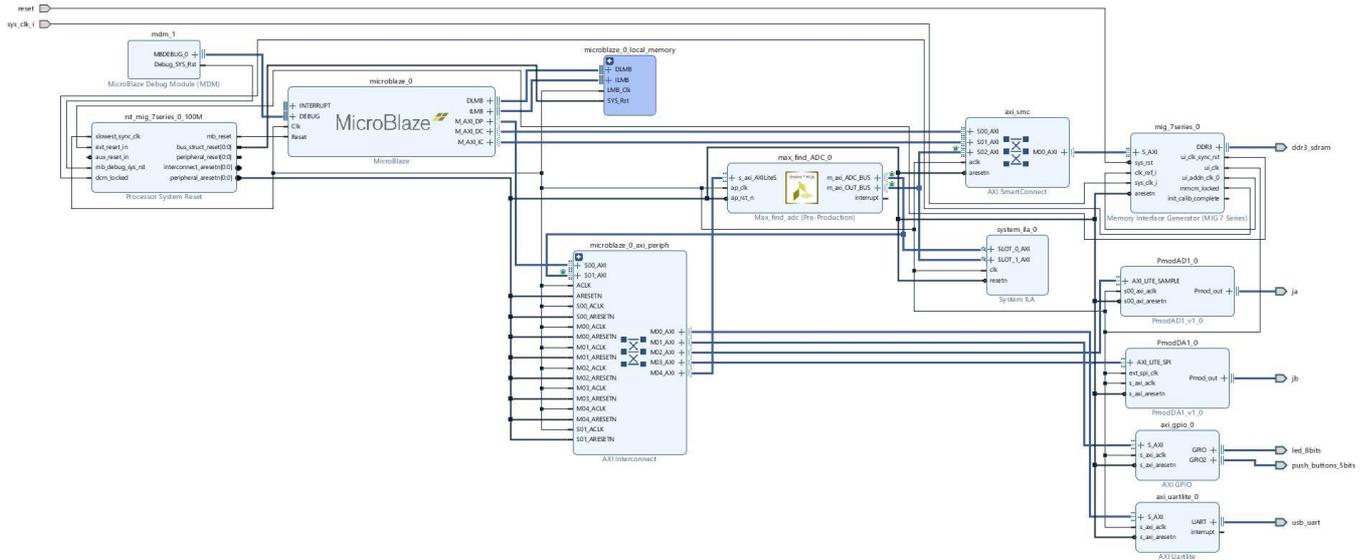
    for(int i = 0; i < MAX_ARRAY_SIZE; i++)
    {
        printf("out[%d] = %lu\n",i,out[i]);
    }
    return 0;
}

```

After the C simulation is completed the C synthesis and the C/RTL Cosimulation are launched. Then the IP block is exported so it can be used in the Vivado block design.

## 4.5.2 Vivado implementation

In this section the implemented design in Vivado is presented (Fig. 50). The memory addresses of each IP block are illustrated in figure 51. In order to create the block design illustrated in figure 50, it is necessary to include the “board files” of the FPGA board (Nexys Video) in the Vivado project. In order to instantiate the PmodAD1 and the PmodDA1 blocks, it is necessary to include the specific libraries. Moreover, it is necessary to include the libraries of the Vivado HLS project in order to add the Max\_finder\_adc IP core to the block design.



**Fig. 50** Vivado implementation.

Cell	Slave Interface	Base Name	Offset Address	Range	High Address
microblaze_0					
Data (32 address bits : 4G)					
PmodAD1_0	AXI_LITE_SAMPLE	Reg0	0x44A0_0000	64K	0x44A0_FFFF
PmodDA1_0	AXI_LITE_SPI	Reg0	0x0001_0000	64K	0x0001_FFFF
axi_gpio_0	S_AXI	Reg	0x4000_0000	64K	0x4000_FFFF
axi_uartlite_0	S_AXI	Reg	0x4060_0000	64K	0x4060_FFFF
microblaze_0_local_memory/dlmb_bram_if_cntlr	SLMB	Mem	0x0000_0000	8K	0x0000_1FFF
max_find_ADC_0	s_axi_AXILiteS	Reg	0x44A1_0000	64K	0x44A1_FFFF
mig_7series_0	S_AXI	memaddr	0x8000_0000	512M	0x9FFF_FFFF
Instruction (32 address bits : 4G)					
microblaze_0_local_memory/ilmb_bram_if_cntlr	SLMB	Mem	0x0000_0000	8K	0x0000_1FFF
mig_7series_0	S_AXI	memaddr	0x8000_0000	512M	0x9FFF_FFFF
max_find_ADC_0					
Data_m_axi_ADC_BUS (32 address bits : 4G)					
Excluded Address Segments (5)					
PmodAD1_0	AXI_LITE_SAMPLE	Reg0	0x44A0_0000	64K	0x44A0_FFFF
PmodDA1_0	AXI_LITE_SPI	Reg0	0x0001_0000	64K	0x0001_FFFF
axi_gpio_0	S_AXI	Reg	0x4000_0000	64K	0x4000_FFFF
axi_uartlite_0	S_AXI	Reg	0x4060_0000	64K	0x4060_FFFF
max_find_ADC_0	s_axi_AXILiteS	Reg	0x44A1_0000	64K	0x44A1_FFFF
Data_m_axi_OUT_BUS (32 address bits : 4G)					
mig_7series_0	S_AXI	memaddr	0x8000_0000	512M	0x9FFF_FFFF

**Fig. 51** Memory addresses.

As shown in figure 50 the MicroBlaze Core is connected to the AXI Interconnect, AXI SmartConnect, MDM and local memory as in the 4.1.1 project.

The MIG 7 Series IP core is connected to the AXI SmartConnect and to the ddr3\_sdram pin as in the 4.1.1 Vivado implementation.

The peripherals PmodAD1, PmodDA1, AXI UARTlite and AXI GPIO are connected to the AXI Interconnect as indicated in 4.1.1.

The reset and the sys\_clock\_i pins are also connected in the same way as in 4.1.1.

The Processor System Reset is connected as in the 4.1.1 block design.

The Max\_find\_adc IP core is connected to the AXI interconnect output with an AXI4-Lite interface, so it can communicate with the Microblaze IP core as the previous custom IP cores. But, as already mentioned in the HLS project, the Max\_find\_adc IP core has two AXI4-Master interfaces:

- m\_axi\_ADC\_BUS interface, which is connected to the input of the AXI Interconnect IP core; so, this interface can directly take the samples from the PmodAD1 IP core without using software resources;
- m\_axi\_OUT\_BUS interface, which is connected to the input of the AXI SmartConnect IP core; so, this interface can directly write the calculated peaks to the DDR3 memory without using software resources.

The System ILA IP core is connected to the AXI4-Master interfaces for debug purposes.

For the same reasons as in section 4.1.1, a constraints.xdc file is needed with the following command:

```
set_property IOSTANDARD LVCMOS33 [get_ports sys_clock_i]
```

Once the block design is verified, the HDL wrapper is created and the Vivado synthesis is launched. Since the pins used are the same as in section 4.3.1 project, the pinout is the same as the one in section 4.3.1.

After the synthesis is completed, the implementation (Fig. 52) and the bitstream file are generated. Then the hardware is exported to the Xilinx SDK tool in order to program the Microblaze Core.

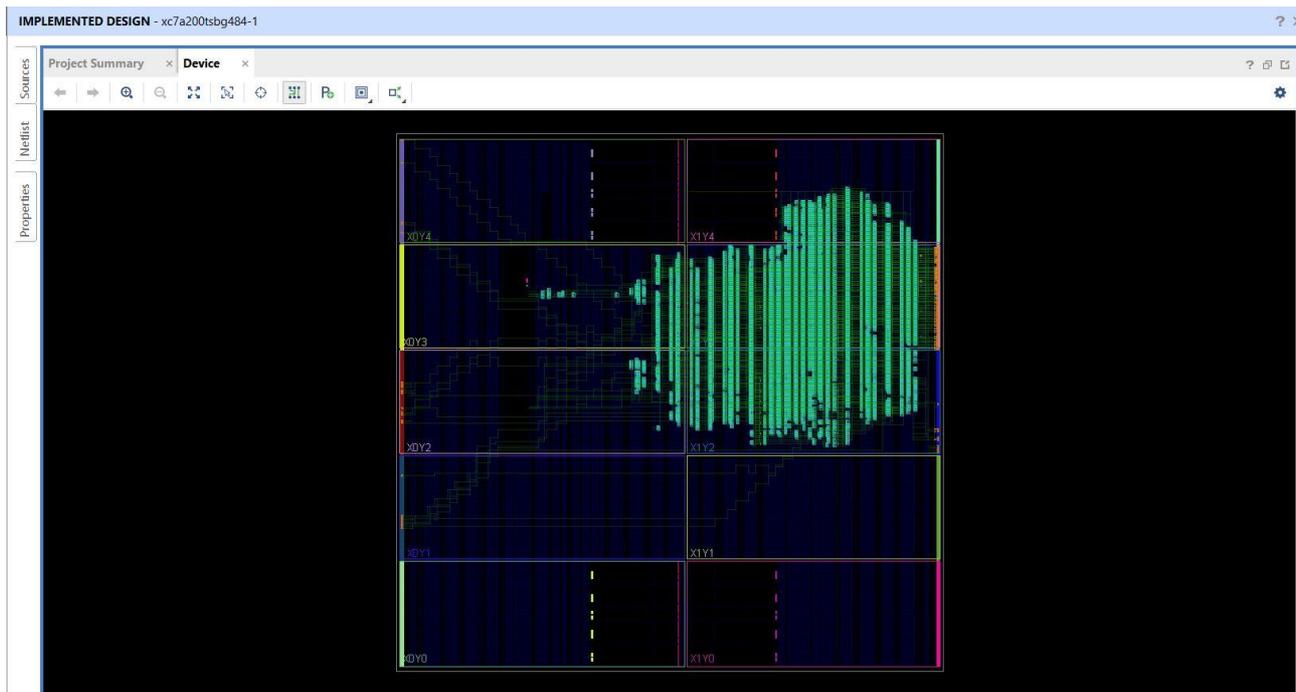


Fig. 52 Implemented design.

### 4.5.3 Xilinx SDK implementation

In this section the SDK implementation is presented.

```

#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "xparameters.h"
#include "PmodAD1.h"
#include "sleep.h"
#include "xil_cache.h"
#include "xil_io.h"
#include "xil_types.h"
#include "xmax_find_adc.h"
#include "PmodDA1.h"

#define MEM_BASE_ADDRESS 0x80000000
#define TX_BUFFER_BASE (MEM_BASE_ADDRESS + 0x00100000)
#define RX_BUFFER_BASE (MEM_BASE_ADDRESS + 0x00300000)

u32 *Mem = (u32*) TX_BUFFER_BASE;
u16 Mem16;
float memf;

u32 adc;
u16 adc16;
float adcf;

PmodAD1 AD1;
PmodDA1 DA1;

```

```

const float ReferenceVoltage = 3.3;

XMax_find_adc max_find;

float conversionFactor = ReferenceVoltage / ((1 << AD1_NUM_BITS) - 1);

int main()
{
    init_platform();

    AD1_begin(&AD1, XPAR_PMODAD1_0_AXI_LITE_SAMPLE_BASEADDR);
    DA1_begin(&DA1, XPAR_PMODDA1_0_AXI_LITE_SPI_BASEADDR);

    XMax_find_adc_Initialize(&max_find, XPAR_MAX_FIND_ADC_0_DEVICE_ID);

    XMax_find_adc_Set_adc_addr(&max_find, XPAR_PMODAD1_0_AXI_LITE_SAMPLE_BASEADDR);

    XMax_find_adc_Set_out_r(&max_find, TX_BUFFER_BASE);

    while(1){
    XMax_find_adc_Start(&max_find);
    while(!XMax_find_adc_IsDone(&max_find));

    for(int i = 0; i < 100; i++)
    {
        adc = Xil_In32(XMax_find_adc_Get_adc_addr(&max_find));
        adc16 = adc & AD1_DATA_MASK;
        adcf = adc16 * conversionFactor;
        printf("%d: adc = %f\n",i,adcf);
    }

    for(int i = 0; i < 5; i++)
    {
        Mem16 = Mem[i] & AD1_DATA_MASK;
        memf = Mem16 * conversionFactor;
        printf("mem[%d] = %f\n",i,memf);
        DA1_WritePhysicalValue(&DA1, memf);
    }
}

    cleanup_platform();
    return 0;
}

```

As we can see in the SDK code, the Microblaze just initializes the PmodAD1, the PmodDA1 and the Max\_find\_adc IP core. Then, the CPU sets the memory addresses for the AXI4-Master interfaces. The adc\_addr AXI4-Master interface is connected to the PmodAD1 memory base address; the out AXI4-Master interface is connected to the DDR3 memory base address plus a 0x00100000 offset. The memory offset is

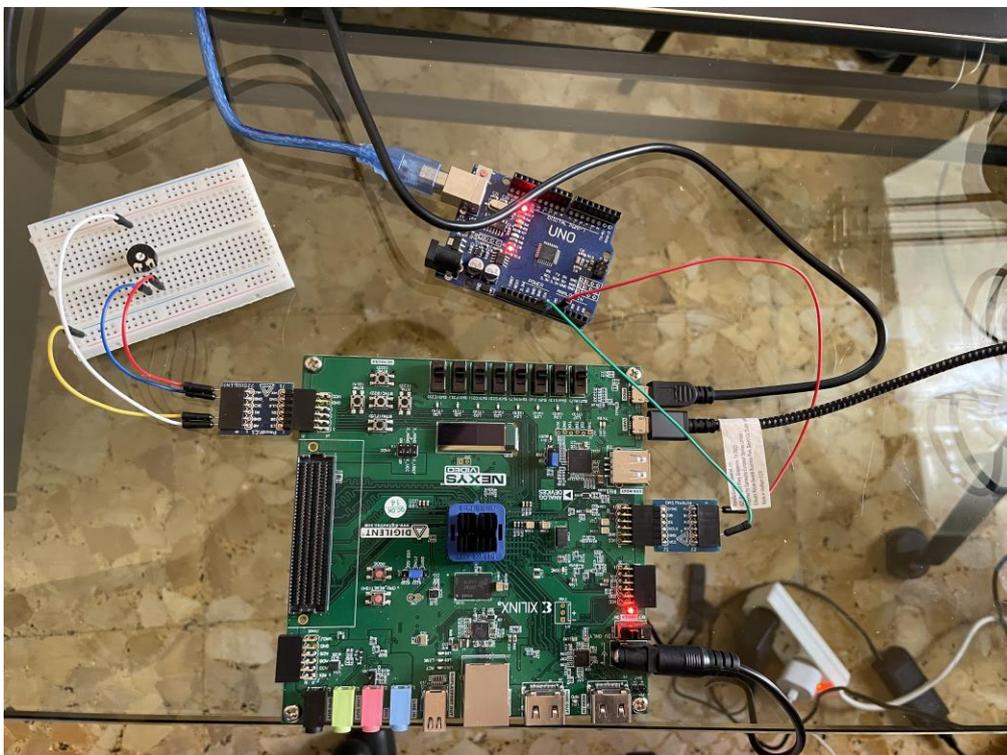
added in order not to affect the project code written in the first part of the DDR3 memory.

In the while(1) loop the ADC samples are converted and printed just to check the functioning of the PmodAD1. Finally, the peaks are printed from the DDR3 memory which is pointed by the mem pointer.

The crucial aspect of this project is that the software resources of the Microblaze CPU are used neither to transfer the ADC samples to the Max\_find\_adc IP core nor to calculate the peaks, because this process takes place in hardware by the custom IP core created in Vivado HLS. The Microblaze Core is used only to check the results.

#### 4.5.4 Hardware setup and results

In this section the hardware setup is presented (Fig. 53).



**Fig. 53** Hardware setup.

In this hardware setup the PmodAD1 is connected to the JA connector of the Nexys Video board while the PmodDA1 is connected to the JB connector. The analog input signal is passed to the A0 pin of the PmodAD1, and it is changed with a trimmer placed on the bread board. The A1 output pin of the DAC is connected to the A0 analog pin of the Arduino UNO microcontroller which is connected to the COM3 port of the PC. The UART port of the FPGA board is connected to the COM7 port. Next are some results which are the values printed on the COM7 port by the FPGA. As shown in the SDK code the printed values on the UART port are the ADC samples and the calculated peaks.

```
Connected to: Serial ( COM7, 9600, 0, 8 )
68: adc = 1.794652
69: adc = 1.791429
70: adc = 1.791429
71: adc = 1.791429
72: adc = 1.792234
73: adc = 1.792234
74: adc = 1.792234
75: adc = 1.789011
76: adc = 1.792234
77: adc = 1.795458
78: adc = 1.792234
79: adc = 1.794652
80: adc = 1.792234
81: adc = 1.792234
82: adc = 1.792234
83: adc = 1.794652
84: adc = 1.794652
85: adc = 1.792234
86: adc = 1.789011
87: adc = 1.794652
88: adc = 1.792234
89: adc = 1.791429
90: adc = 1.791429
91: adc = 1.791429
92: adc = 1.792234
93: adc = 1.791429
94: adc = 1.794652
95: adc = 1.791429
96: adc = 1.791429
97: adc = 1.794652
98: adc = 1.792234
99: adc = 1.794652
mem[0] = 1.794652
mem[1] = 1.791429
mem[2] = 1.794652
mem[3] = 1.794652
mem[4] = 1.794652
```

**Fig. 54** ADC samples acquired by the FPGA and found peaks printed on the COM7 port.



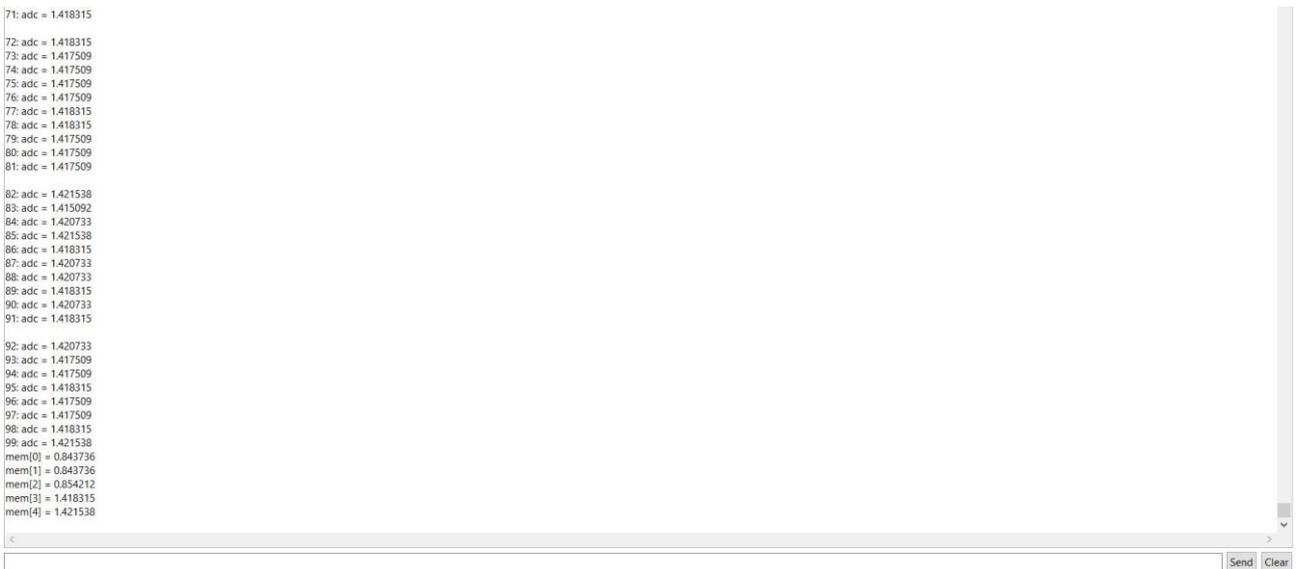
```
Connected to: Serial ( COM7, 9600, 0, 8 )
69: adc = 0.844542
70: adc = 0.837289
71: adc = 0.834872
72: adc = 0.843736

73: adc = 0.837289
74: adc = 0.838095
75: adc = 0.843736
76: adc = 0.837289
77: adc = 0.834872
78: adc = 0.834872
79: adc = 0.830842
80: adc = 0.834066
81: adc = 0.830842
82: adc = 0.827619

83: adc = 0.827619
84: adc = 0.834066
85: adc = 0.830842
86: adc = 0.818755
87: adc = 0.824396
88: adc = 0.834872
89: adc = 0.818755
90: adc = 0.838095
91: adc = 0.834872
92: adc = 0.834066
93: adc = 0.834872
94: adc = 0.838095
95: adc = 0.837289
96: adc = 0.838095
97: adc = 0.834872
98: adc = 0.834872
99: adc = 0.846960
mem[0] = 1.098388
mem[1] = 3.296777
mem[2] = 0.841319

mem[3] = 0.844542
mem[4] = 0.838095
```

**Fig. 55** ADC samples acquired by the FPGA and found peaks printed on the COM7 port.



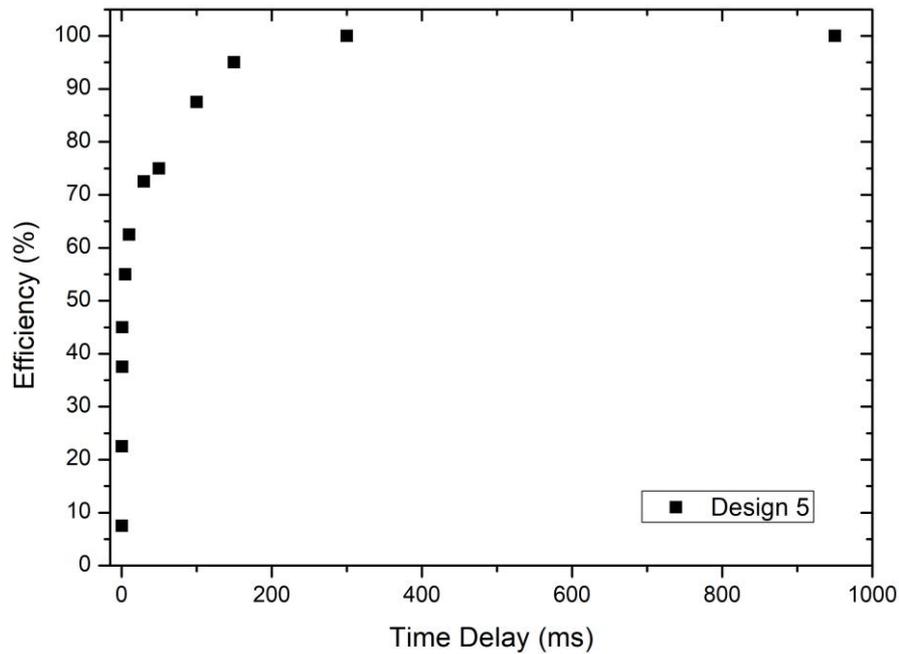
```
71: adc = 1.418315
72: adc = 1.418315
73: adc = 1.417509
74: adc = 1.417509
75: adc = 1.417509
76: adc = 1.417509
77: adc = 1.418315
78: adc = 1.418315
79: adc = 1.417509
80: adc = 1.417509
81: adc = 1.417509

82: adc = 1.421538
83: adc = 1.415092
84: adc = 1.420733
85: adc = 1.421538
86: adc = 1.418315
87: adc = 1.420733
88: adc = 1.420733
89: adc = 1.418315
90: adc = 1.420733
91: adc = 1.418315

92: adc = 1.420733
93: adc = 1.417509
94: adc = 1.417509
95: adc = 1.418315
96: adc = 1.417509
97: adc = 1.417509
98: adc = 1.418315
99: adc = 1.421538
mem[0] = 0.843736
mem[1] = 0.843736
mem[2] = 0.854212
mem[3] = 1.418315
mem[4] = 1.421538
```

**Fig. 56** ADC samples acquired by the FPGA and found peaks printed on the COM7 port.

Finally, some measurements are conducted using the same hardware setup as the previous projects. The Efficiency versus time delay between pulses is shown in figure 57.

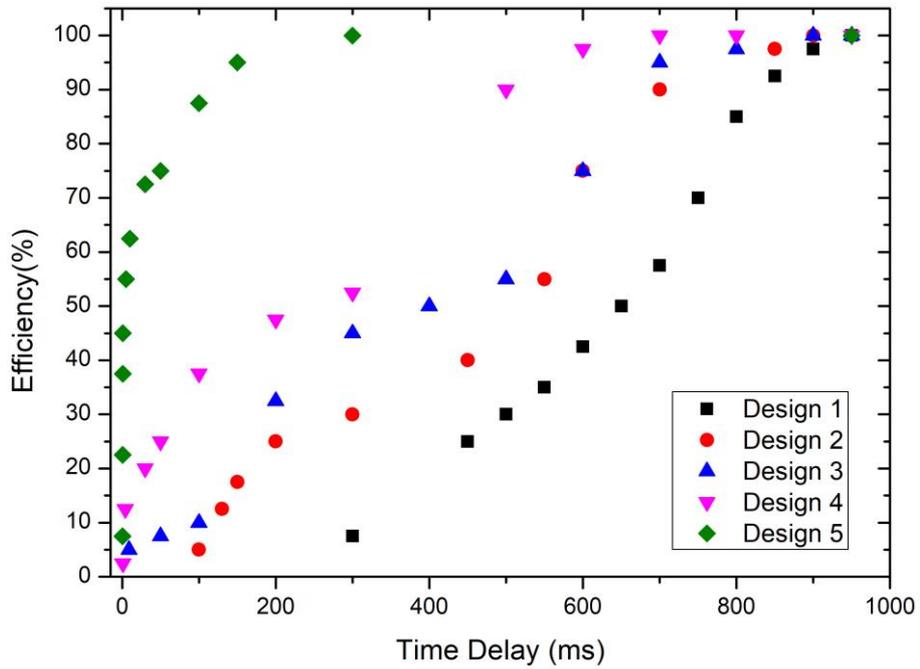


**Fig. 57** Efficiency versus time delay between pulses.

As the graph shows, an efficiency of 50% is obtained for about 1 ms period. When the period of the input signal is below 1 ms, the graph starts to drop and by the time the input period is of the order of 0.5 ms, the setup misses almost all the input peaks.

## 4.6 Efficiency comparison

Finally, in order to compare the efficiency of each design, all the efficiency graphs are superimposed (Fig. 58).



**Fig. 58** Superimposition of the efficiency graphs.

As shown in the graph, the project presented in section 4.5 appears to be the most efficient, while the project presented in section 4.1 is the less efficient. The projects presented in section 4.2, 4.3 and 4.4 show an intermediate performance with respect to the other designs.

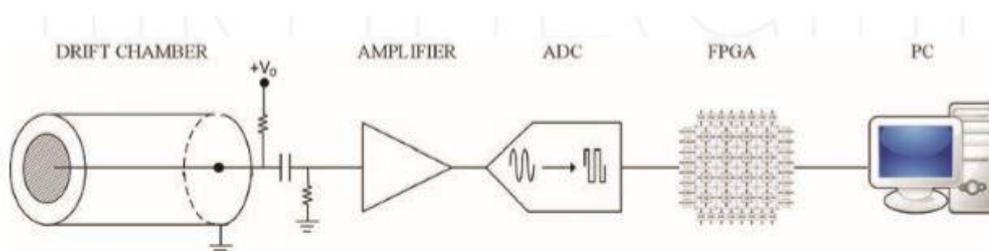
## CONCLUSIONS

In this work, a peak-finding algorithm implemented on different FPGA architectures has been presented. Each design allows the processing of the input data in real time, thus reducing the amount of data to be stored since only the useful data (such as the peaks and their amplitudes) are saved in the internal memory of the FPGA. Moreover, each design uses progressively less software resources, thus making the peak-finding algorithms more and more efficient.

By comparing the efficiency graph of each design, the FPGA implementation of section 4.5 appears to be the most efficient one with respect to how many peaks are acquired in relation to the increasing frequency of the input signal.

As shown in the efficiency graphs, the implemented FPGA designs are able to acquire the input peaks when the distance between two peaks is in the order of the milliseconds. As already mentioned in the first chapter, the time interval between two peaks in a signal coming from a drift chamber filled with a “slow” gas is in the order of 1 nanosecond. Therefore, the FPGA setups implemented in this work provide preliminary results since the implemented architectures start to fail to acquire the input peaks when the time interval between two peaks is in the order of hundreds of milliseconds. So, the provided FPGA setups are not suited to be used in a real application.

A big limitation for the efficiency of each FPGA design is the used hardware. In fact, an acquisition chain is represented in figure 59.



**Fig.59** Acquisition chain for a drift chamber. [1]

As figure 59 shows, the setup includes an amplifier and an ADC to get the signal from the drift chamber, then the signals are processed by the FPGA and the found peaks are sent to the PC. Instead, in the presented hardware setups, the output signal is sent to a DAC in order to be displayed on the oscilloscope, slowing down the peak acquisition. Moreover, the used ADC and DAC are low-cost devices.

Another big limitation of the implemented designs is how the ADC sends the data to the FPGA. The PmodAD1 gets the signal from the wave generator and serially sends the samples to the FPGA using the SPI protocol. Of course, serially sending the samples to the FPGA is not the optimal way to acquire the input signal. By using an ADC which parallelly sends the samples to the FPGA, the efficiency of the designs would drastically improve.

The topic addressed in this dissertation leaves a lot of room for future developments.

Ideas for future work include:

- the progressive elimination of the Microblaze IP core; in this way the implemented FPGA design could work without a microcontroller;
- the use of an ADC which exploits parallel communication to send the samples to the FPGA;
- implementing more elaborated/efficient peak-finding algorithms on the FPGA;
- storing not only the amplitude of the peaks found, but also the timing information by adding new IP cores to the block design in Vivado such as an AXI Timer IP core.

## References

- [1] Chiarello G., C. Chiri, G. Cocciolo, A. Corvaglia, F. Grancagnolo, M. Panareo, A. Pepino and G. Francesco Tassielli (2017), ‘The Use of FPGA in Drift Chambers for High Energy Physics Experiments’, in *Field – Programmable Gate Array* chapter 7.
- [2] García Ferreira I-B, J García Herrera and L Villaseñor (2005), ‘The Drift Chambers Handbook, introductory laboratory course (based on, and adapted from, A H Walenta’s course notes)’, *Journal of Physics: Conference Series*, Vol. 18, pp.346–361.
- [3] Ege1 Y., H. Çıtak2, M. Çoramık1, ‘A FPGA Project with MYRIO’, *8<sup>th</sup> International Advanced Technologies Symposium (IATS’17)*, 19-22 October 2017.
- [4] Digilent Inc., *Nexys Video™ FPGA Board Reference Manual*, September 1, 2020.
- [5] Xilinx Inc., *The MicroBlaze Soft Processor: Flexibility and Performance for Cost-Sensitive Embedded Designs*, April 13, 2017.
- [6] Xilinx Inc. *Microblaze processor reference guide*, May 22, 2019.
- [7] Analog Devices Inc., *AD9689 (Rev. A)*, 2017.
- [8] Serial Peripheral Interface, Internet Document, [https://it.wikipedia.org/wiki/Serial\\_Peripheral\\_Interface](https://it.wikipedia.org/wiki/Serial_Peripheral_Interface) , Accessed on October 30, 2021.
- [9] Xilinx Vivado, Internet Document, [https://en.wikipedia.org/wiki/Xilinx\\_Vivado](https://en.wikipedia.org/wiki/Xilinx_Vivado) , Accessed on 30 October 2021.
- [10] Mattoccia S. *Calcolatori Elettronici Modulo 2*, 2010.
- [11] IENITK National Institute of Technology Karnataka, Surathka , Internet Document, <https://ie.nitk.ac.in/virtual-expo/> , Accessed on 1 November, 2021.
- [12] Visconti P., *IL PROTOCOLLO SPI*, [THE SPI PROTOCOL], Lecture notes.
- [13] G. Chiarello, C. Chiri, G. Cocciolo, A. Corvaglia, F. Grancagnolo, A. Miccoli, M. Panareo, C. Pinto, F. Renga, G. F. Tassielli, C. Voena, ‘Improving spatial and PID performance of the high transparency Drift Chamber by using the Cluster

Counting and Timing techniques' *Nuclear Inst. And Methods in Physics Research*, A, (2018), <https://doi.org/10.1016/j.nima.2018.10.181>.

[14] AXI4-Lite Interface, Internet Document, <https://www.realdigital.org/doc/a9fee931f7a172423e1ba73f66ca4081> , Accessed on 03 January 2022.

[15] Advanced eXtensible Interface, Internet Document, [https://en.wikipedia.org/wiki/Advanced\\_eXtensible\\_Interface#AXI](https://en.wikipedia.org/wiki/Advanced_eXtensible_Interface#AXI), Accessed on 03 January 2022.

[16] Arbitrary data streams, Internet Document, <https://lauri.xn--vsandi-pxa.com/hdl/zynq/axi-stream.html> , Accessed on 03 January 2022.

[17] Xilinx Inc., *SmartConnect v1.0 LogiCORE IP Product Guide*, February 3, 2020.

[18] Xilinx Inc., *MicroBlaze Micro Controller System v3.0 LogiCORE IP Product Guide*, July 15, 2021.

[19] Xilinx Inc., *AXI Interconnect v2.1 LogiCORE IP Product Guide (PG059)*, December 20, 2017.

[20] Xilinx Inc., *AXI GPIO v2.0 LogiCORE IP Product Guide (PG144)*, October 5, 2016.

[21] Xilinx Inc., *Processor System Reset Module v5.0 LogiCORE IP Product Guide (PG164)*, November 18, 2015.

[22] Digilent Inc., *PmodADI™ Reference Manual*, April 15, 2016.

[23] Digilent Inc., *PmodDAI™ Reference Manual*, May 24, 2016.

[24] Xilinx Inc., *Vivado Design Suite User Guide: High-Level Synthesis*, May 4, 2021.

[25] Xilinx Inc., *AXI DMA v7.1 LogiCORE IP Product Guide*, June 14, 2019.