



UNIVERSITAT POLITÈCNICA DE CATALUNYA  
BARCELONATECH

Escola Tècnica Superior d'Enginyeria  
de Telecomunicació de Barcelona



# Hardware accelerators for real time processing systems

---

Master Thesis  
submitted to the Faculty of the  
Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona  
Universitat Politècnica de Catalunya  
by  
Mikel Alcubilla Ayestaran

In partial fulfillment  
of the requirements for the master in  
*(MEE)* **ENGINEERING**

Advisor: Josep Altet Sanahujes  
Barcelona, Date 02/07/2021



# Contents

<b>List of Figures</b>	<b>5</b>
<b>List of Tables</b>	<b>6</b>
<b>1 introduction</b>	<b>11</b>
1.1 Context . . . . .	11
1.2 Applications . . . . .	13
1.3 Problem statement . . . . .	14
1.4 Research question . . . . .	14
1.5 Outline . . . . .	14
<b>2 State of the Art</b>	<b>15</b>
2.1 Vivado HLS . . . . .	15
2.1.1 Scheduling and Binding example . . . . .	16
2.2 PYNQ . . . . .	17
2.2.1 Key technologies of PYNQ . . . . .	18
2.3 Vivado . . . . .	18
2.4 SDK . . . . .	19
2.5 Methodology / Project development . . . . .	19
2.5.1 Vivado HLS . . . . .	19
2.5.2 Vivado . . . . .	20
2.5.3 SDK . . . . .	20
2.5.4 PYNQ development . . . . .	20
2.6 Needed Xilinx IPs or blocks for the project . . . . .	21
2.6.1 AXI Timer . . . . .	22
2.6.2 DMA . . . . .	23
2.6.3 AXI Interconnect . . . . .	23
2.6.4 Processor System Reset . . . . .	23
2.6.5 Zynq-7000 SoC . . . . .	23
<b>3 Designs</b>	<b>25</b>
3.1 FIR filter . . . . .	26
3.1.1 Vivado HLS generated FIR filter . . . . .	26
3.1.2 Software FIR implementation . . . . .	27
3.1.3 PYNQ Hardware vs Software FIR filter comparison . . . . .	27
3.2 Half float . . . . .	28
3.3 Image Convolution . . . . .	30
3.3.1 Approximated Multiplier . . . . .	33
3.3.2 Approximated Adder . . . . .	34
<b>4 Results</b>	<b>35</b>
4.1 FIR . . . . .	35
4.2 Half float . . . . .	38
4.3 Convolution . . . . .	39

---

5 Conclusions	48
References	49

---

## Acronyms

**AI** Artificial Intelligence

**ASIC** Application Specific Integrated Circuit

**DMA** Direct Memory Access

**FIR** Finite Impulse Response

**FPGA** Field Programmable Gate Array

**GPU** Graphical Processing Unit

**IP** Intellectual Property core, a block of logic that is used in FPGAs or ASICs for a given product

**PL** Programmable Logic

**PS** Processing System

**SDK** Software Development Kit, IDE of Vivado, Xilinx

**SoC** System-on-Chip

**TPU** Tensor Processing Unit

**VFPU** Vector Floating Point Unit

## List of Figures

1	A simplified model of the Zynq architecture [3]. . . . .	11
2	Processor activity before (top) and after (bottom) hardware acceleration of FIR filtering [3]. . . . .	12
3	C code example . . . . .	16
4	PYNQ Framework . . . . .	17
5	PYNQ Key technologies . . . . .	18
6	Methodology diagram . . . . .	21
7	AXI Timer block diagram . . . . .	22
8	AXI DMA IP Core block diagram . . . . .	23
9	Zynq SoC block diagram . . . . .	24
10	PS PL communication for the different hardware accelerators . . . . .	25
11	FIR Filter of order $N$ . . . . .	26
12	FIR IP Core by Vivado HLS . . . . .	26
13	FIR IP Core Vivado design . . . . .	27
14	Overlay configuration via Jupyter Notebook . . . . .	28
15	Binary16 format . . . . .	28
16	Half float IP Core by Vivado HLS . . . . .	29
17	Half float IP Core Vivado design . . . . .	29
18	Image convolution example . . . . .	30
19	Line Buffer (input pixels of image) multiplied by a Memory Window (Kernel)	32
20	Modified Karnaugh Map for approximated multiplier . . . . .	33
21	Example of 4x4 Multiplier using 2x2 blocks . . . . .	33
22	Vivado Catalog FIR IP Core . . . . .	36
23	Area usage vs data type for 76,800 pixels . . . . .	40
24	Area usage vs data type for 230,400 pixels . . . . .	41
25	Processing times vs data types . . . . .	41
26	image convolved by an edge detection kernel . . . . .	42
27	Images of approx arithmetics applied into edge detection . . . . .	43
28	(a) input image. (b) Normal convolution. (c) Approximated adder convolution	44
29	(a) input image. (b) Normal convolution. (c) Approximated adder convolution	45
30	(a) input image. (b) Normal convolution. (c) Approximated adder convolution	46
31	(a) Normal convolution (b) Approx. multiplier convolution (c) Approx. adder convolution. The kernel in this case was the Identity . . . . .	47

---

## List of Tables

1	Kernel multiplication examples . . . . .	31
2	Hardware FIR filters Resource utilization . . . . .	36
3	FIR filter results with <i>Vivado General flow</i> . . . . .	36
4	FIR filter results with <i>PYNQ development flow</i> . . . . .	37
5	Utilization report comparing half and single precision floating point multi- plication . . . . .	38
6	Processing times of half and single precision floating point multiplication .	39
7	Resource utilization for different convolution implementations . . . . .	43
8	Processing times for different convolution implementations . . . . .	43

## Revision history and approval record

Revision	Date	Purpose
0	30/05/2021	Document creation
1	22/06/2021	Document revision
2	30/06/2021	Document revision

### DOCUMENT DISTRIBUTION LIST

Name	e-mail
Mikel Alcubilla Ayestaran	mikel.alcubilla@estudiantat.upc.edu
Dr. Josep Altet Sanahujes	josep.altet@upc.edu

Written by:		Reviewed and approved by:	
Date	02/07/2021	Date	02/07/2021
Name	Mikel Alcubilla Ayestaran	Name	Dr Josep Altet Sanahujes
Position	Project Author	Position	Project Supervisor



## Abstract

This Master Thesis presents different Hardware acceleration algorithms and its benefits compared to the software implementation. The proposed algorithms are implemented on Xilinx ZYNQ-7000 series XC7Z020 SoC using High-Level-Synthesis (HLS) tool. With todays System-on-Chips from Xilinx or Intel, a process can be chosen to be implemented in the Programmable Logic or in the Processing System.

In order to have a better acceleration factor, different approximate and accurate adders and multipliers were instantiated in *Verilog*, synthesized and simulated using *Vivado* and finally they were compared between each other to see if they really offer benefits or not. In the case of approximated adders, they showed very promising results for the application written in this Thesis. On the other hand, approximated multipliers exhibited worse results than the accurate ones.

---

## Acknowledgements

Foremost, I would like to express my sincere gratitude to my advisor Prof. Josep Altet for the continuous support of my master thesis study and research, for his patience, motivation, enthusiasm and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I would also like to extend my gratitude to the second advisor, Francesc Moll Echeto, also for his guidance in this research.

Secondly, I would like to express my gratitude to the UPC who offered me the opportunity and the necessary background and knowledge to develop this master thesis.

Last but not the least, I would like to thank my family: my parents Rafael and Pepi, my sisters Andrea, Carolina and Natalia, and to my girlfriend Maialen, for supporting me in the difficult moments in the development of this thesis.

# 1 introduction

## 1.1 Context

Hardware accelerators have a wide range of applications that include surveillance, automotive, robotics and medical diagnosis. Hardware acceleration is the use of specific hardware to perform more efficiently than a general-purpose Central Processing Unit (CPU) running a software designed to perform the same function. Hardware accelerators give the opportunity to decrease the latency and increase the throughput for a given computation. Any function or data transformation can be computed or calculated purely in software on a generic CPU. Each approach, developing software in a generic CPU or developing hardware for a specific function, has advantages and disadvantages.

The most known advantage of software is easier development (leading to faster times to market) and also there are more software developers than hardware ones. Other advantages like: heightened portability, and ease of updating features or patching bugs, at the cost of overhead to compute general operations.

Hardware acceleration is an old idea. For instance, *86 Family* processors [9] could have an arithmetic coprocessor for floating point operations. But right now with the appearance of *SoCs* (System-on-Chips), which offer the possibility to implement a whole system into a single chip, there is a need to have complex processes running in real time systems (Real Time Recognition, Autonomous driving). This have made hardware accelerators a very current need. In the figure 1 can be seen the general structure of this SoC devices, in this case a Xilinx Zynq Family FPGA.

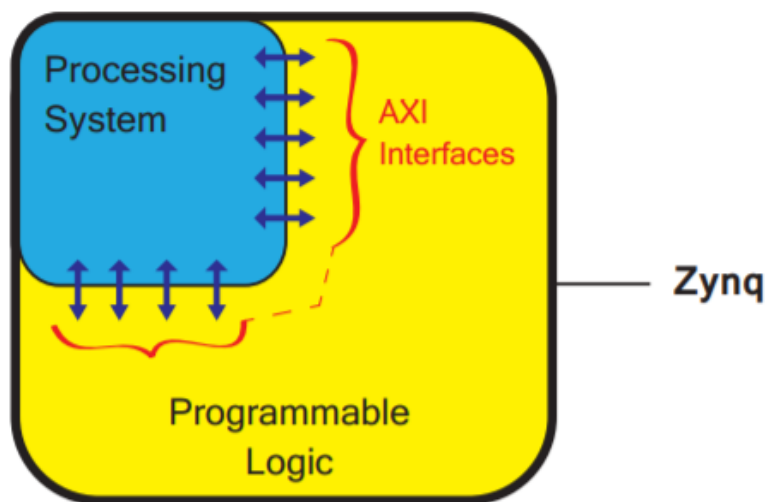


Figure 1: A simplified model of the Zynq architecture [3].

In this simplified model, the *SoC* is divided in two main parts:

- **Processing System (PS):** All Zynq devices have the same basic architecture, and all of them contain, as the basis of the processing system, a dual-core ARM Cortex-A9 processor. This is a ‘hard’ processor — it exists as a dedicated and optimised silicon element on the device.

- **Programmable Logic (PL):** this will be the FPGA. The PL section is ideal for implementing high-speed logic, arithmetic and data flow subsystems.

While the PS supports software routines and/or operating systems, meaning that the overall functionality of any designed system can be appropriately partitioned between hardware and software. Links between the PL and PS are made using industry standard Advanced eXtensible Interface (AXI) connections [3].

FPGA logic fabric permits a high-speed, fully parallel version of a design to be implemented. This ‘off-loading’ of operations from the processor into hardware has the potential to greatly reduce the overall execution time, as illustrated in Figure 2 for the case of FIR filtering. It also frees up the processor, such that it can undertake other operations using the processing cycles vacated by the accelerated function.

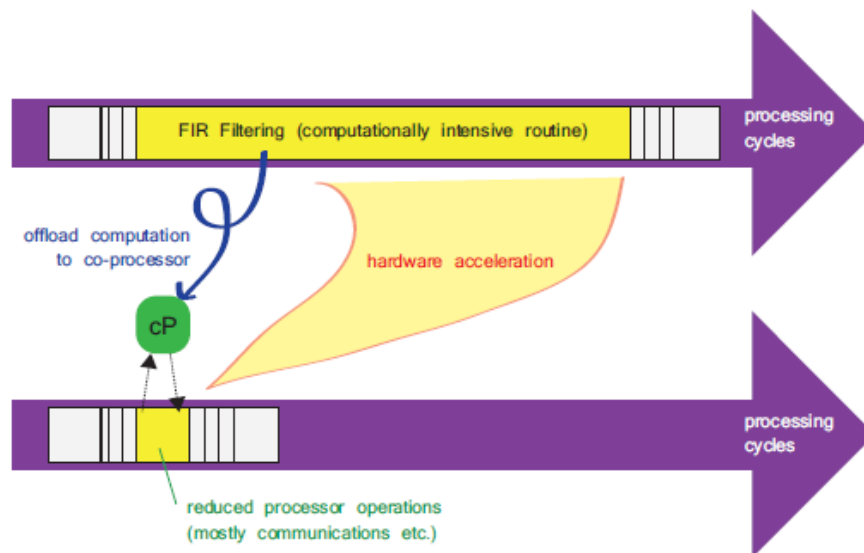


Figure 2: Processor activity before (top) and after (bottom) hardware acceleration of FIR filtering [3].

With the advent of reprogrammable logic devices such as FPGAs, the restriction of hardware acceleration to fully fixed algorithms has been removed these last years, allowing hardware acceleration to be applied to problem domains requiring modification to algorithms and specific solutions.

Software or hardware can compute any computable function. Hardware description languages (HDLs) such as Verilog and VHDL can model the same semantics as software and synthesize the design into a netlist that can be programmed to an FPGA or composed into logic gates of an application-specific integrated circuit. They normally offer higher performance per watt for the same functions implemented in software.

Custom hardware is limited in parallel processing capability only by the area and logic blocks available on the integrated circuit die. Therefore, hardware is much more free to offer massive parallelism than software on general-purpose processors, but this comes with

a cost. Right now, the space in an ASIC or in an FPGA is a very cost sensitive parameter. There are very large FPGAs where you can implement many hardware designs but those FPGAs are not intended to be used by a wide range of people due to its cost ( $\approx 30,000\$$ ).

Another hot topic right now that was also studied in this *Thesis* was implementing *Approximated Arithmetics*. Approximated multipliers or adders are intended to use in applications where the 100% accuracy is not needed. Approximate computing is extensively researched in its multiple forms and sources [5, 7].

## 1.2 Applications

Conventionally, processors were sequential (instructions are executed one at a time), and were designed to run general purpose algorithms controlled by instructions (for example moving temporary registers data from one register to another). Hardware accelerators improve the execution of a specific algorithm by allowing greater concurrency, having specific data paths for their temporary variables, and reducing the overhead of instruction control.

Modern processors are multi-core and often feature parallel "single-instruction; multiple data" (SIMD) units. Even so, hardware acceleration still yields benefits. Hardware acceleration is suitable for any computation-intensive algorithm which is executed frequently in a task or program. Depending upon the granularity, hardware acceleration can vary from a small functional unit, to a large functional block.

There are many applications and fields where hardware accelerators work pretty well. These are some examples:

1. **Computer Graphics:** the hardware accelerator here for excellence is the Graphical processing unit, also known as GPU.
2. **Digital signal processing:** in this application the specific chip is called DSP, Digital Signal Processor.
3. **Multilinear algebra:** This application is for Machine Learning, a really hot topic right now for many systems [1, 2]. Google developed the TPU, Tensor Processing Unit, an ASIC for Artificial Neural Networks and optimized to be used with Tensorflow, a very known library for machine learning.

On the other hand, here we will see how we implement hardware accelerators in FPGAs, which has a really big benefit from an ASIC, its reconfiguration capability. In an ever-changing world, this characteristic of an FPGA gives them a really good advantage versus ASICs.

FPGA technology is really good for emerging technologies which are constantly evolving, and the reconfiguration is a key aspect for this kind of situations.

As one can see, hardware accelerators can be used in many different sectors, but in this *Thesis*, different IP Cores are going to be developed with one application in mind, ***Autonomous driving***.

### 1.3 Problem statement

The aim of this *Thesis* is to show different design methodologies in order to implement different hardware accelerators in FPGAs and compare the results regarding to resource usage, execution time and precision.

A comparative on how hardware and software is processed in Xilinx platforms will be shown and how to manage which functions are executed on the Programmable Logic or in the Processing System of the FPGA.

Afterwards, a comparison in the PL side, comparing exact adders and multipliers with approximate adders/multipliers. This could lead to errors but a cost of speed and resource utilization. In applications where probability is involved (*Machine Learning, Image Recognition*), these kind of operands are of real interest if there is a better acceleration.

### 1.4 Research question

The research question can be formulated as: Which are the tradeoffs between implementing specific functions on the Programmable Logic of an FPGA versus programming them in the Processing system? Which is the design flow(s) on the different tools? Which are the benefits of implementing pure adders/multipliers against approximated versions of these?

There are more questions derived from the main one:

- Hardware utilization?
- Which *steps* have to be followed by the designer in order to implement a model on an FPGA?
- Which tools are available?
- Maximum frequency?
- Processing times for Hardware, Software and approximated Hardware?

### 1.5 Outline

In the 2nd chapter the different *tools* for the project development and the design flow will be explained. In the 3er chapter different design examples with different topologies will be shown. In the 4th chapter, the results of the experiments and in the 5th chapter conclusions and future work.

## 2 State of the Art

In this chapter, information about the project development and the different tools will be explained. First, an explanation on Vivado HLS, and how can be C code synthesized into HDL code. Once that the software code has been successfully exported to an IP Core Then, a Vivado project with its corresponding *Standalone Application*. Afterwards, a new open-source tool from Xilinx, *PYNQ*.

### 2.1 Vivado HLS

The Xilinx Vivado High-Level Synthesis (HLS) tool transforms a C specification into a register transfer level (RTL) implementation that you can synthesize into a Xilinx FPGA [6]. You can write C specifications in C, C++, or SystemC, and the FPGA provides a massively parallel architecture with benefits in performance, cost, and power over traditional processors. This chapter provides an overview of high-level synthesis.

High-level synthesis includes the following phases:

- **Scheduling**

Determines which operations occur during each clock cycle based on:

- Length of the clock cycle or clock frequency.
- Time it takes for the operation to complete, as defined by the target device.
- User-specified optimization directives.

If the clock period is longer or a faster FPGA is targeted, more operations are completed within a single clock cycle, and all operations might complete in one clock cycle. Conversely, if the clock period is shorter or a slower FPGA is targeted, high-level synthesis automatically schedules the operations over more clock cycles, and some operations might need to be implemented as multicycle resources.

- **Binding**

Determines which hardware resource implements each scheduled operation. Binding determines which library cell is used for each operation in the C code.

- **Control logic extraction**

Extracts the control logic to create a finite state machine (FSM) that sequences the operations in the RTL design.

HLS synthesizes the C code as follows:

- Top-level function arguments synthesize into RTL I/O ports.
- C functions synthesize into blocks in the RTL hierarchy.

If the C code includes a hierarchy of sub-functions, the final RTL design includes a hierarchy of modules or entities that have a one-to-one correspondence with the

original C function hierarchy. All instances of a function use the same RTL implementation or block.

- Loops in the C functions are kept *rolled* by default.

When loops are rolled, synthesis creates the logic for one iteration of the loop, and the RTL design executes this logic for each iteration of the loop in sequence. Using optimization directives, you can *unroll* loops, which allows all iterations to occur in parallel.

### 2.1.1 Scheduling and Binding example

In the following figure 3, shows an example of the scheduling and binding phases for this code.

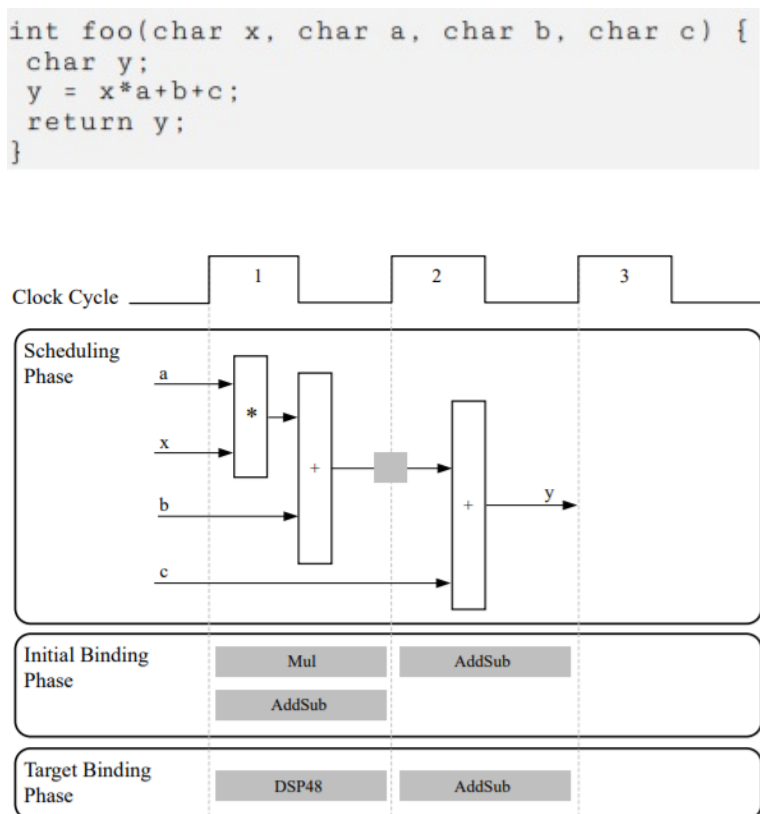


Figure 3: C code example

In the scheduling phase of this example, high-level synthesis schedules the following operations to occur during each clock cycle:

- First clock cycle: Multiplication and the first addition.
- Second clock cycle: Second addition and output generation.

In the preceding figure, the square between the first and second clock cycles indicates when an internal register stores a variable. In this example, high-level synthesis only requires



that the output of the addition is registered across a clock cycle. The first cycle reads  $x$ ,  $a$ , and  $b$  data ports. The second cycle reads data port  $c$  and generates output  $y$ .

In the final hardware implementation, HLS implements the arguments to the toplevel function as input and output (I/O) ports. In this example, the arguments are simple data ports. Because each input variable is a *char* type, the input data ports are all 8-bits wide. The function *return* is a 32-bit *int* data type, and the output data port is 32-bits wide.

The **main advantage** of implementing the C code in the hardware is that all operations finish in a shorter number of clock cycles. In this example, the operations complete in only two clock cycles. In a CPU, even this simple code example takes more clock cycles to complete.

## 2.2 PYNQ

As mentioned before, *PYNQ* is an open-source project from Xilinx, which tries to make easier to program their platforms with high-level programming languages like Python [8].

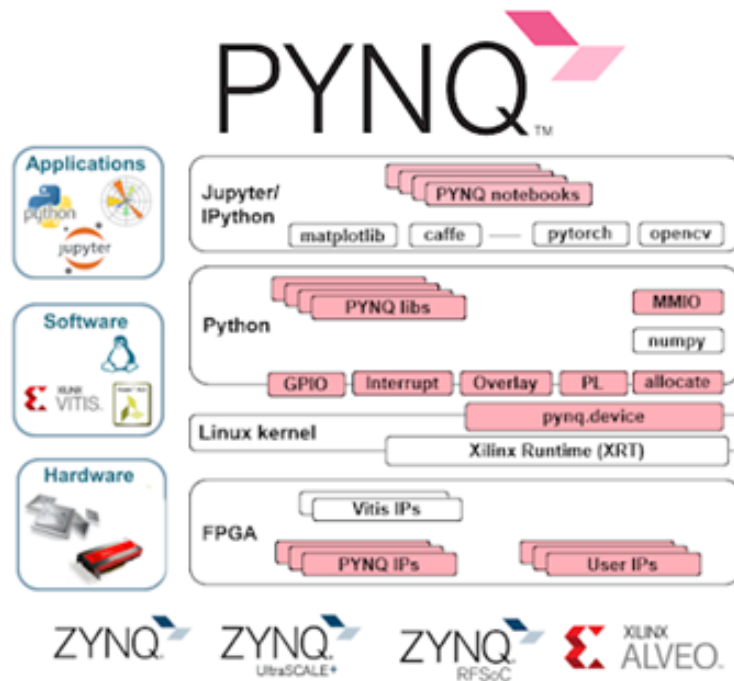


Figure 4: PYNQ Framework

Using high-level programming languages like Python and his libraries, designers can exploit the benefits of Programmable logic and microprocessors to build more easily exiting electronics systems. *PYNQ* can be used with these FPGA families from Xilinx: *Zynq*, *ZynqUltraScale+*, *Zynq RFSOC*, Alveo accelerator boards and *AWS-F1*.

With this project, Xilinx tries to include more developers and designers to program their platforms. People like:

- Software developers who want to take advantage of hardware accelerators and Xilinx board's capabilities without having or knowing to use ASIC-style design tools to design hardware.
- Hardware designers who want their designs to be used by the widest possible audience.
- System architects who want an easy software interface and framework for rapid development of their designs.

### 2.2.1 Key technologies of PYNQ

The three main technologies in order to run this in a Xilinx board are shown in the Figure 5.



Figure 5: PYNQ Key technologies

Jupyter notebooks is a browser based interactive computing environment. Jupyter notebook documents can be created that include live code, interactive widgets, plots, explanatory text, images and video.

A board running *PYNQ*, can be easily programmed using Python. Using Python, developers can easily preload what is called an overlay or hardware library which is reconfiguring the FPGA with a new bitstream. The main advantage of this is the ease programmability of the SoCs, with just two clicks in the web browser one can change FPGA's configuration.

In order to run the software *PYNQ* into a Xilinx board, one needs to create a bootable SD card preloaded with a *PYNQ* image. Everything is explained in the github repository: <https://github.com/Xilinx/PYNQ.git>.

## 2.3 Vivado

Vivado Design Suite is a software suite produced by Xilinx for synthesis and analysis of HDL designs, superseding Xilinx ISE with additional features for system on a chip development and high-level synthesis.

## 2.4 SDK

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. SDK is based on the Eclipse open source standard. SDK features include:

- Feature-rich C/C++ code editor and compilation environment.
- Project management.
- Application build configuration and automatic Makefile generation.
- Error navigation.
- Well-integrated environment for seamless debugging and profiling of embedded targets.
- Source code version control.
- System level performance analysis.
- Focussed special tools to configure FPGA.
- Bootable Image creation.
- Flash Programming.
- Scriptable command-line tool.

## 2.5 Methodology / Project development

With the used tools explained, the project development involves the following steps.

### 2.5.1 Vivado HLS

The first step was thinking in a hardware accelerator application for a real time processing system and implement it using the Vivado HLS tool. We will see later on that Vivado HLS sometimes adds *more* logic to the design so it will be better always to implement the HDL code. As it is been previously mentioned, the Vivado HLS tool has been chosen because of the fast prototyping possibilities of having an IP Core from a well known programming language like C or C++.

As this tool synthesises hardware from a software description, this tool works very well for this development flow.

In order to check the correct functionality of the IP, a simulation of the desired function was done also in C. This is the same as a hardware testbench. Once created a C simulation, one can check the correct behaviour of the signals with the Vivado HLS as well. The simulation is also done with the same tool, having a testbench (also written in C or C++) where the wanted software function is used.

Once the function is synthesisable and the behaviour is as expected, the next step was to export the C code to HDL code, both *Verilog* or *VHDL* and export the new IP Core to the Vivado IP Catalog.

### 2.5.2 Vivado

In order to make the design, some of the Vivado Catalog IP Cores were used. In the [subsection 2.6](#), these IP Cores will be explained with more detail.

As it will be shown later on, almost every IP Core that have been done, have the same structure/interfaces. So in this case the communication always starts at the processor, forwarding the information to the IP Cores and finally going back to the processor with the results.

Once the design has been done, the bitstream must be exported and create a *SDK Standalone Application* project.

### 2.5.3 SDK

Regarding the Zynq platform specifically, Xilinx provides a standalone OS platform that provides functions such as configuring caches, setting up interrupts and exceptions and other hardware related functions.

With the aim of seeing the results more graphically, IP Cores output was captured with a well known application like *TeraTerm* and then print the results more graphically with *Matlab*.

### 2.5.4 PYNQ development

Once that everything works fine, the bitstream and the block design were loaded into the FPGA's SD Card. With this, a Jupyter notebook was also created for the purpose of "playing" a bit with the design and the different *Python* libraries.

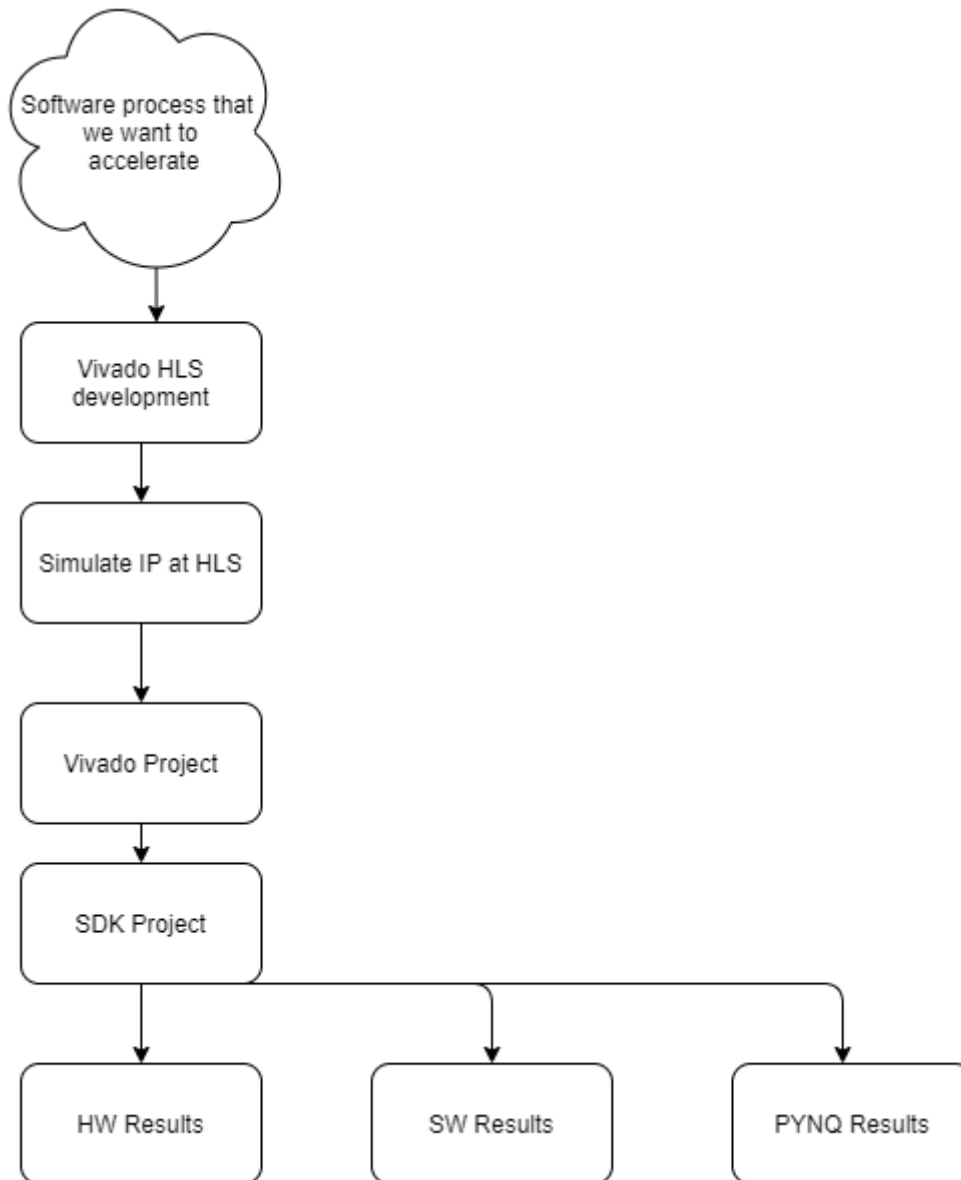


Figure 6: Methodology diagram

## 2.6 Needed Xilinx IPs or blocks for the project

In this subsection the main needed IPs from the Xilinx IP Catalog will be described.

These cores will be explained:

- **AXITimer IP Core.** This IP was instantiated in order to measure the latencies for a given design.
- **AXI DMA IP Core.** This IP Core was used in order to connect the PL side of the SoC to the PS. The connection with the SoC will be explained later on.
- **AXI interconnect.**

- **Processor System Reset.**
- **Zynq-7000 SoC.** The SoC of every implementation is explained here in more detail.

### 2.6.1 AXI Timer

In order to measure different processing times, the Xilinx IP *AXI TIMER* was instantiated in the block design. This *AXI TIMER* IP Core is a well known Xilinx IP which gives you the number of clock cycles count for a given clock. In this design, the clock was generated from the PS side, Xilinx ZYNQ FPGA Family, and had a *100MHz* clock frequency.

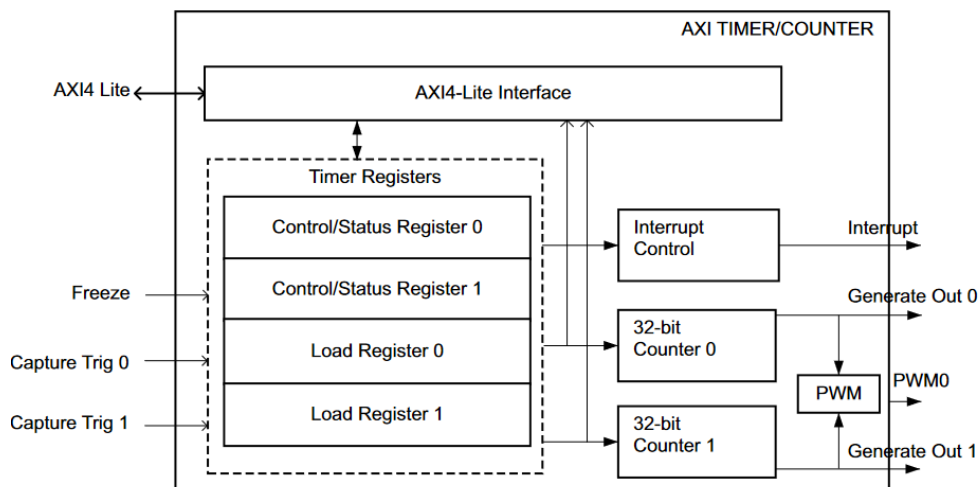


Figure 7: AXI Timer block diagram

## 2.6.2 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. Initialization, status, and management registers are accessed through an AXI4-Lite slave interface. Figure 8 illustrates the functional composition of the core.

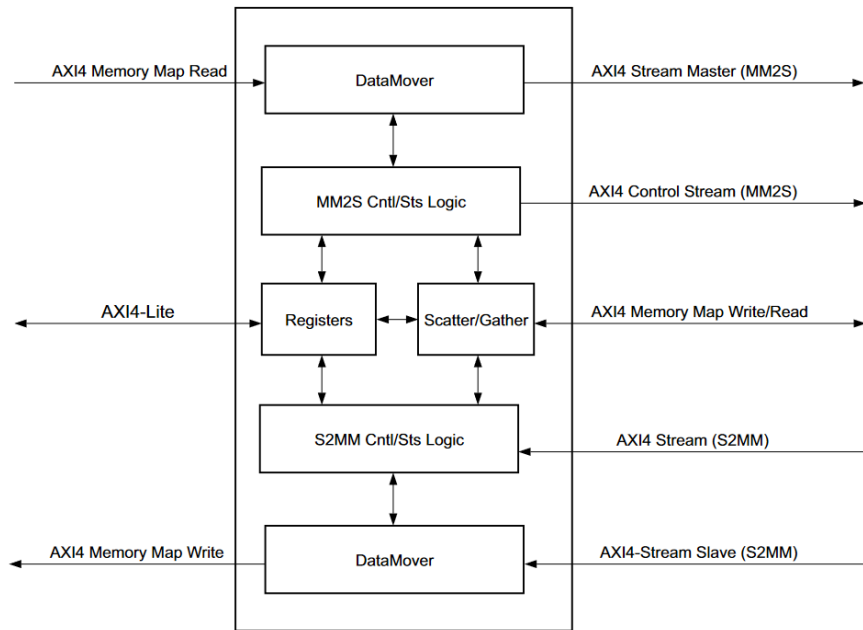


Figure 8: AXI DMA IP Core block diagram

## 2.6.3 AXI Interconnect

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, including the AXI4-Lite control register interface subset. The Interconnect IP is intended for memory-mapped transfers only; AXI4-Stream transfers are not applicable.

## 2.6.4 Processor System Reset

This module is used for customized resets for an entire processor system, including the processor, the interconnect and peripherals.

## 2.6.5 Zynq-7000 SoC

Well, this is not exactly an IP Core, but it was very necessary in this project in order to compare hardware and software results.

The Zynq-7000 SoC family integrates the software programmability of an ARM-based processor with the hardware programmability of an FPGA, enabling key analytics and

hardware acceleration while integrating CPU, DSP and ASSP.

The communication between the PL and the PS was using the AXI DMA IP Core.

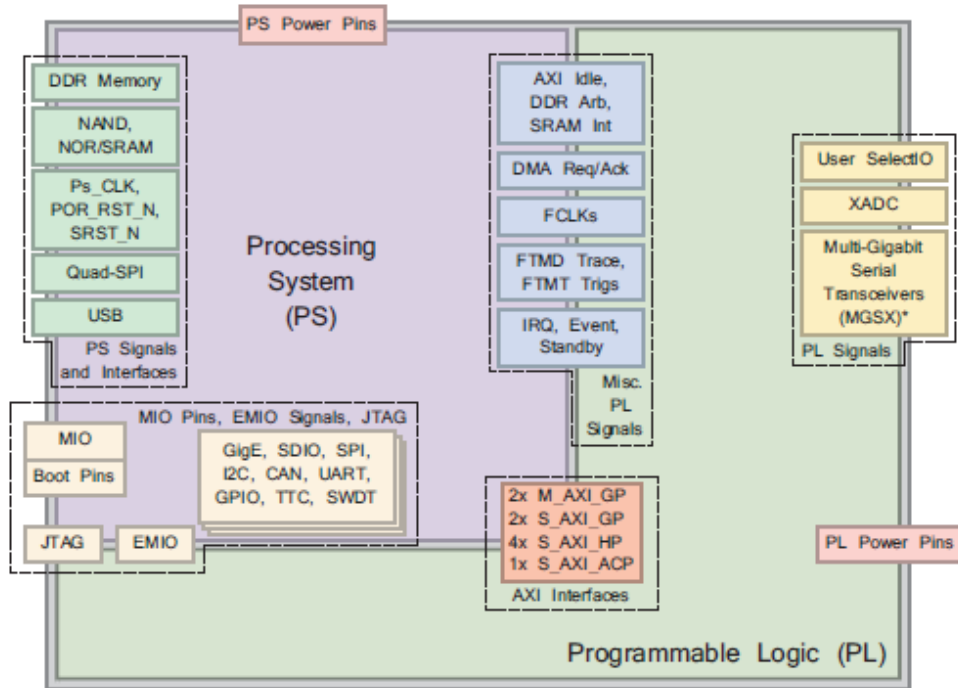


Figure 9: Zynq SoC block diagram



### 3 Designs

In this section different designs and development flow will be shown.

First, a **Finite Impulse Response**, FIR, filter will be explained. This application was chosen because its well known application and it also can be easily compared/implemented with *Python* libraries. Also, it was a kind of warm up with the Vivado HLS tool and how to manage the different AXI interfaces (Xilinx bus communication protocol).

After this, we will focus more on IP Cores related to *Autonomous driving*, and in this case and IP Core for **half floating point precision format** multiplication was developed. This kind of data types are used by Machine Learning in the learning process. The weights and the biases are normally float data type.

To end up, a **Convolution** IP Core was developed. This kind of process is very common in *Autonomous driving* and for *edge detection*. As matrix convolution has many multiplications and sumations, we thought implementing the Convolution process with *approximated multipliers* and *approximated adders* [5].

The figure 10 shows the communication between PS and PL of the Zynq-7000 SoC. As every design had an Standalone Application Project, the communication starts at the PS side (n<sup>o</sup>1 of the figure). Then, the processor tells to the AXI DMA IP Core to start forwarding the data to the custom IP Core (n<sup>o</sup>2). Once the IP has processed the data, it forwards it to the DMA again (n<sup>o</sup>3) and finally the DMA writes the results into registers of the processor (n<sup>o</sup>4). In order to use the DMA and have a continuous communication between the PS and PL, AXI-Stream protocol was needed. This interface will be explained with more detail in [subsubsection 3.1.1](#)

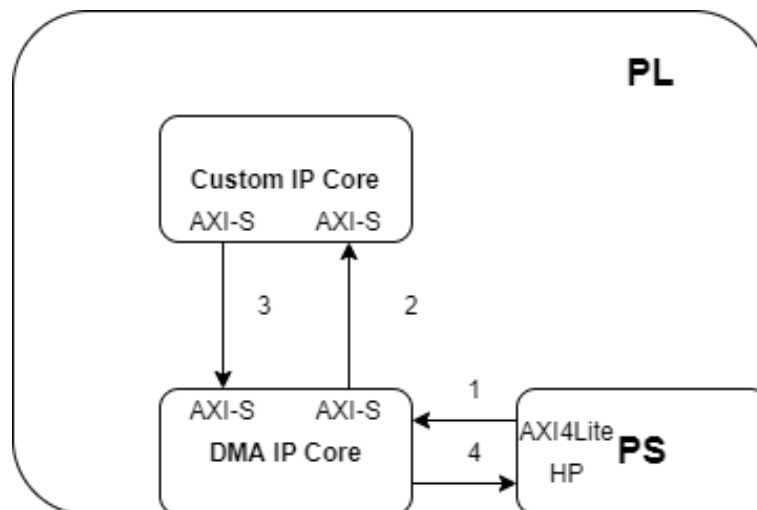


Figure 10: PS PL communication for the different hardware accelerators

### 3.1 FIR filter

In digital signal processing, an FIR is a filter whose impulse response is of finite period, as a result of it settles to zero in finite time. This is often in distinction to IIR filters, which can have internal feedback and will still respond indefinitely. The impulse response of an Nth order discrete time FIR filter takes precisely  $N+1$  samples before it then settles to zero. FIR filters are most popular kind of filters executed in software and these filters can be continuous time, analog or digital and discrete time

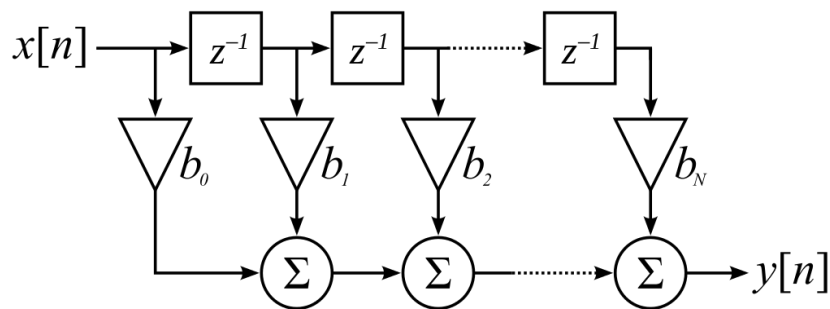


Figure 11: FIR Filter of order  $N$

#### 3.1.1 Vivado HLS generated FIR filter

As can be seen in the figure 12, the FIR IP designed by the Vivado HLS has the following interfaces:

- **AXI4-Stream:** used for continuous communication, inStream and outStream ports were used for the input and output data respectively.
- **AXI4-Lite:** used for reading and writing in memory registers of the IP. CTRL BUS for synchronization purposes and ORDER BUS for providing the FIR filter coefficients.

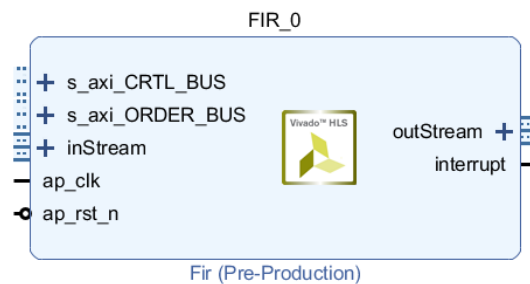


Figure 12: FIR IP Core by Vivado HLS

In order to create those Xilinx-related interfaces like AXI4-Lite or AXI4-Stream, there

is need to use Vivado HLS *directives* and declare those variables with AXI4-Stream or AXI4-Lite data structures.

AXI4-Stream is a protocol designed to transport arbitrary unidirectional data streams. In AXI4-Stream **TDATA** width of bits is transferred per clock cycle. The transfer is started once sender signals **TVALID** and received responds with **TREADY**. **TLAST** signals the last byte of the stream.

To sum up, AXI4-Stream interface is usually used for high-speed streaming data, and on the other hand, AXI-Lite is used for simple, low-throughput memory-mapped communication, for example, to and from control and status registers.

In this case, as we need high-speed streaming data for the FIR application, the data is forwarded through the PS side to the Direct Memory Access (DMA) and finally to the FIR IP Core. The coefficients are written into different register through AXI4-Lite interface because in this case there is no need to be constantly streaming data. When the FIR IP Core finalizes computing the input data with the coefficients, it writes directly to the PS memory thanks to the DMA.

In the figure 13 the block design can be appreciated.

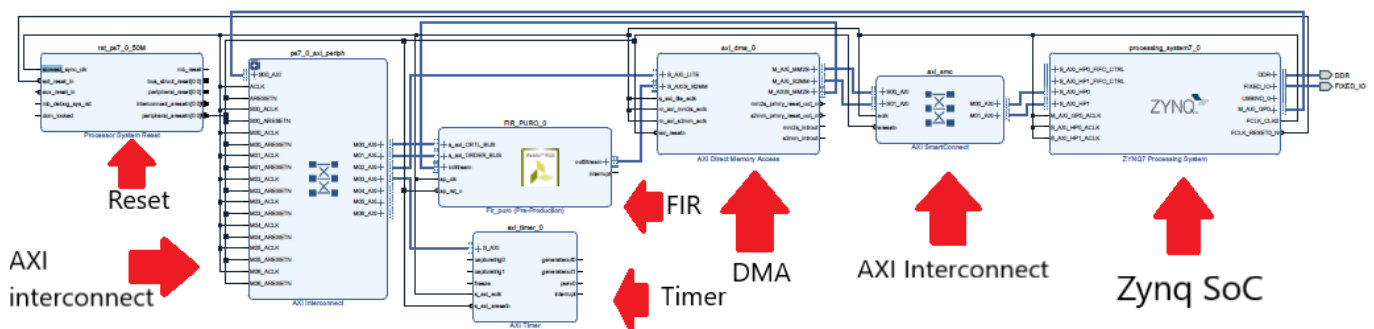


Figure 13: FIR IP Core Vivado design

### 3.1.2 Software FIR implementation

Once generated the bitstream of the previous FPGA configuration, an *SDK Standalone Application* was created in order to compare the performance of the Hardware FIR filter IP Core created by the Vivado HLS tool vs Software implementation of the FIR filter in the *Application project*.

### 3.1.3 PYNQ Hardware vs Software FIR filter comparison

Calling different *Overlays* will configure the FPGA with different bitstreams. In the next figure 14 can be appreciated how a bitstream is loaded from a **Jupyter notebook** and how we can check the *address map* from it too calling the *Overlay* function. Of course, in order to create an *Overlay*, a bitstream and a Block design need to be loaded into the SD card of the FPGA, so the great advantage here is to have different bitstreams and block

designs loaded into the SD card and be able to change FPGAs configuration in a very short period of time.

```
In [4]: from pynq import Overlay
import pynq.lib.dma
|
# Load the overlay
overlay = Overlay('/home/xilinx/pynq/overlays/fir_accel/fir_accel.bit')
|
# Load the FIR DMA
dma = overlay.filter.fir_dma
```

Figure 14: Overlay configuration via Jupyter Notebook

In order to implement the FIR filter design with *Python* libraries, the user can use the different Python packages for this purpose. In this case, the library *SciPy* package which contains functions like *lfilter* can be of great help.

Also, with the *time* library of Python, it can be easily computed the processing time of different processes.

### 3.2 Half float

The work developed in this study was partially made by the *DRAC* (Designing RISC-V-based Accelerators for next generation Computers) project, conformed by the institutions of the BSC and the UPC.

In computing, half precision (sometimes called FP16) is a binary floating-point computer number format that occupies 16 bits (two bytes in modern computers) in computer memory.

They can express values in the range  $\pm 65,504$ , with the minimum value above 1 being  $1 + \frac{1}{1024}$ .

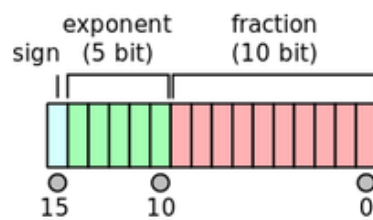


Figure 15: Binary16 format

Half precision data compared to higher precision FP32 vs FP64 reduces memory usage of the neural network, allowing training and deployment of larger networks, and FP16 data transfers take less time than FP32 or FP64 transfers. In this case, the DRAC project has a half precision float multiplier for their neural network biases and weights, but it is implemented in software.

So in this case different IP Cores for FP16 and FP32 were created and compared to the software solution of the *DRAC* project.

This example is given in bit representation of the floating-point value. This includes the sign bit, (biased) exponent, and significand:

$$0\ 0000\ 0000000001_2 = 0001_{16} = 2^{-14} \cdot (0 + \frac{1}{1024}) \approx 0.000000059604645$$

For the case of this IP Core, a constant multiplication was also required. So for this, an AXI4-Stream interface was needed also.

In the figure 16 can be seen the different interfaces of the IP Core. The development of this IP Core was relatively fast. With the software implementation of the *DRAC*, there was just a need to adapt the port interfaces of the IP in order to communicate with the PS of the SoC.

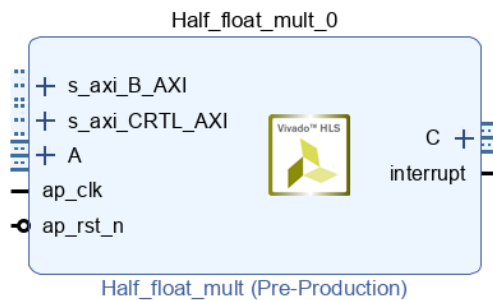


Figure 16: Half float IP Core by Vivado HLS

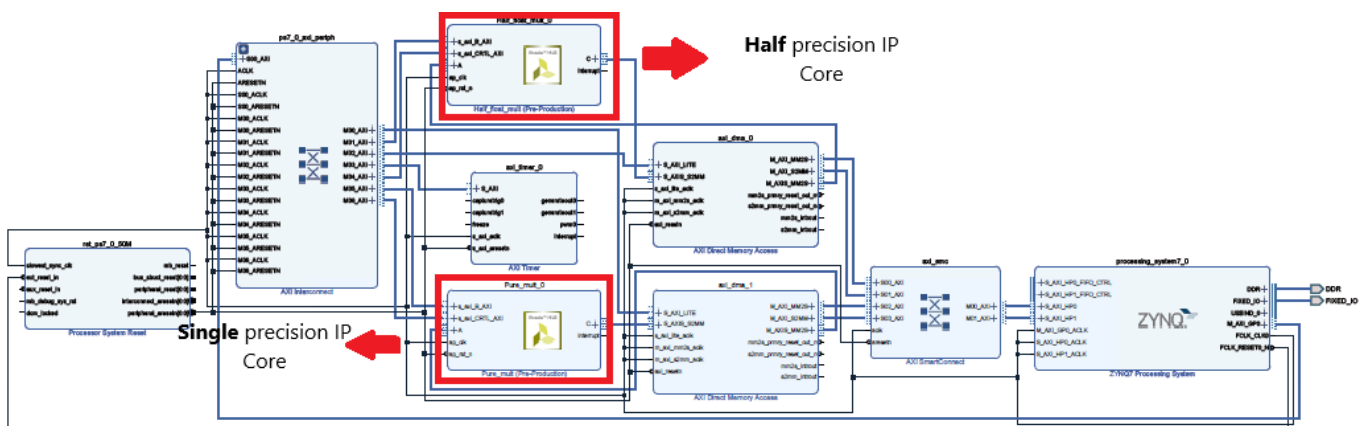


Figure 17: Half float IP Core Vivado design

In this case, the interfaces of the IP are the same as the FIR IP Core design. The Port description of the IP Core is almost the same as explained in the previous subsection,

except the `s_axi_B_AXI` port. This AXI4-Lite input port is constantly multiplied by the input data that comes from the *AXI4-Stream* input port *A*.

In the figure 17 can be appreciated two Vivado HLS IP Cores. One is the *Half float* IP Core and the other one is a *single precision float* IP Core which multiplies *single precision* float data types. This IP was created in order to rapidly compare the *half precision floating* point multiplication. In this case, each IP has a *DMA* connected to the PS of the SoC.

### 3.3 Image Convolution

The last IP Core developed for this study was an Image Convolution IP Core. As we will see in the section 4.3 in this case the acceleration factor is the greater one.

Convolution operation is the main operation in *Machine Learning* and *Edge detection* processes. Every dataset which can be sorted in a matrix form will have a convolution operation. Not just for *Image Recognition*, but for *Voice recognition*

The formula for matrix convolution is the following:

$$g(x, y) = w * f(x, y) = \sum_{dx=-a}^a \sum_{dy=-b}^b w(dx, dy) \cdot f(x + dx, y + dy)$$

Where  $g(x, y)$  is the filtered image,  $f(x, y)$  is the original image and  $w$  is the filter kernel. Every element of the filter kernel is considered by  $a \leq dx \leq a$  and  $b \leq dy \leq b$ .

Depending on the kernel values, a kernel can cause a wide range of effects. In the table 1 can be seen some examples with different kernel values for edge detection [4].

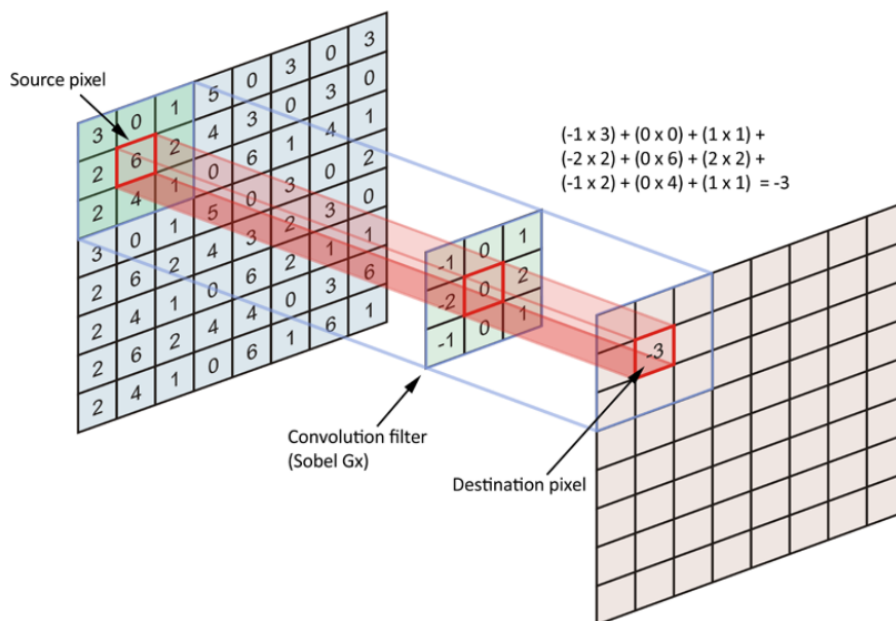


Figure 18: Image convolution example





Operation	Kernel $\omega$	Image result $g(x,y)$
<b>Identity</b>	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
<b>Edge detection</b>	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	

Table 1: Kernel multiplication examples

In order to implement this kind of operations which are most of the times computation intensive, Vivado HLS tool offers different tips in terms of synchronization signal handling and memory architectures. Synchronization was previously explained with the *AXI4-Stream* interface, when it comes to memory architecture, two different architectures were designed: Memory Windows for the kernel and Line Buffers for the input image [6].

- **Memory Windows**

This kind of memory structure is perfect for kernels in image convolution. A memory window is defined as a neighborhood of  $N$  pixels centered on pixel  $P$ . It can be viewed as a collection of shift register which forms a 2-dimensional data storage element.

One of the main characteristics of memory windows for image convolution is that all data elements are available simultaneously.

- **Line Buffer**

A line buffer is a multi-dimensional shift register capable of storing several lines of pixel data. Typically, line buffers are implemented as block RAMs to avoid the communication latency to off-chip DRAM memories.

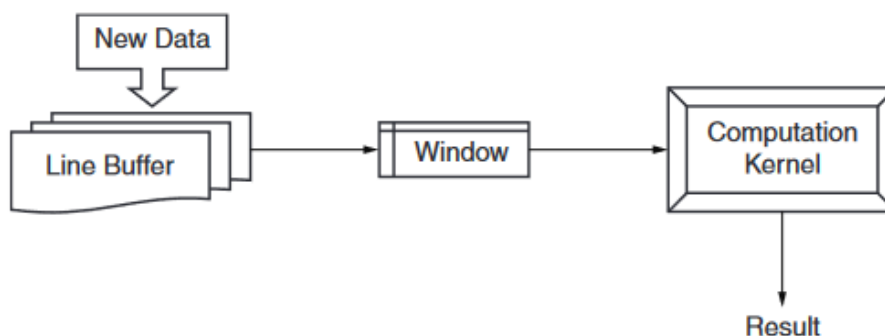


Figure 19: Line Buffer (input pixels of image) multiplied by a Memory Window (Kernel)

Next generation accelerators are intended for applications where the 100% of accuracy is not needed. That's why we will see different implementations of the Image Convolution IP Core with *approximated Multipliers* and *approximated Adders*.



### 3.3.1 Approximated Multiplier

Approximated arithmetics is extensively researched in multiple sectors. One of the main goals of approximated arithmetics is the reduction of the computation time (*Slack*) and computed *Area*, as well as power consumption.

There are many Approximated Multipliers, but in this case the **Under Designed Multiplier (UDM)** of Kulkarni was used [7]. This approximation resides in the modification of the Karnaugh Map of a 2x2 multiplier. The trick here is that the output of the 2x2 bit multiplication will be represented using only three bits instead of the normal four for this case. As the fourth bit is used just in one case ( $3 \cdot 3$ ). So for this case the multiplication will be correct for fifteen out of sixteen possibilities. The figure 20 shows the modified Karnaugh Map for a 2x2 multiplication.

		$B_1B_0$			
		00	01	11	10
$A_1A_0$	00	000	000	000	000
	01	000	001	011	010
	11	000	011	111	110
	10	000	010	110	100

Figure 20: Modified Karnaugh Map for approximated multiplier

Furthermore, larger multipliers can be built using the 2x2 block to produce the partial products and then adding the shifter partial products. This can also be done to obtain even larger multipliers.

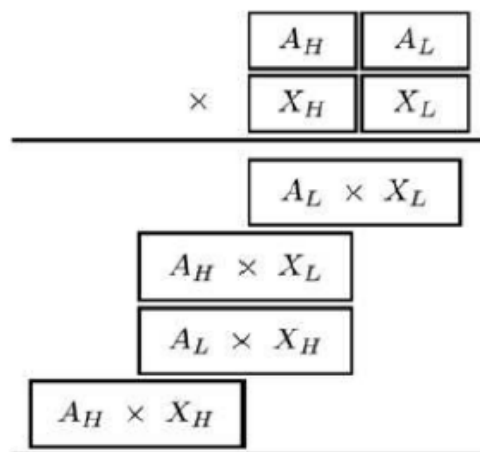


Figure 21: Example of 4x4 Multiplier using 2x2 blocks

### 3.3.2 Approximated Adder

There are different approximation sources such as eliminating transistors at transistor level or layout level of the basic Full Adder cell or at architectural level by truncating the carry propagation (which also happens to be the critical path in adders) by splitting the addition into smaller sub-additions and not propagating the carry.

So there are different approximated adders which offer different benefits and counterparts.

These are the different approximate adders:

1. Almost Correct Adder (ACA-I)
2. Accuracy Configurable Adder (ACA-II)
3. Generic Accuracy Configurable Adder (GeAr)

In this case, the **Generic Accuracy Configurable Adder (GeAr)** was chosen in the convolution application due to the different benefits presented among the others [5]. This adder has different parameters:  $N$ , length of operands to be added;  $L$ , length of the sub-adders in parallel to perform the approximate addition ( $L \leq N$ );  $R$ , represents the number of resultant bits contributing to the final sum; and  $P$ , represents the number of previous bits used for carry prediction. Each sub-adder produces  $R$ -bit results except the first one that produces  $L$ -bit result being  $L=R+P$ .

GeAr was implemented with a  $N=16$ ,  $R=4$  and  $P=4$  configuration, the configuration which suited most the image convolution for the given data type. The best results were obtained for this kind of configuration in terms of Area and Slack, but not in precision. For an application like image recognition or Edge detection, this is the perfect match [?].

## 4 Results

This section shows the results obtained with the aforementioned designs. Each section is divided in two subsections; the first one comparing the Vivado General working flow: Vivado HLS, importing the IP into Vivado Catalog and after that export the bitstream or the *BSP* into a Standalone Software application. And the second subsection, comparing the results in the new framework *PYNQ*, with the developed IP Cores and *Python* implementation of these.

Every IP was developed with a *100MHz* clock signal in mind coming from the PS side.

These were the indicators for the different IP Cores:

- Timing/critical path.
- Resources.
- Acceleration factor, Software vs Hardware. Which was calculated with the given formula:

$$Acceleration\ factor = \frac{Software_{time}}{Hardware_{time}}$$

- Precision for the case of *approximated arithmetics*.

### 4.1 FIR

#### Three different implementations

There were three FIR implementations:

1. HW FIR IP Core generated by Vivado **HLS**.
2. HW FIR IP Core from Vivado **Catalog**.
3. SW FIR implementation in the **Standalone Application** project. A C function running in the PS side of the Zynq SoC.

#### Timing/Critical path

First, the Timing or critical path. Here the Vivado HLS IP showed a critical path of  $8.165ns$  meaning a maximum frequency of  $122.47MHz$ . The FIR IP from the Vivado Catalog showed better numbers, in this case a delay path of  $7.47ns$ , which means a maximum clock speed of  $133.8MHz$ .

#### Resource utilization

The timing values are a consequence of the resource utilization. Looking at the table 2, can be appreciated that the Xilinx IP uses **23** DSPs. This was one drawback of the Vivado HLS tool, that even if there were two *for* loops and the user uses directives in order to use DSP for continuously processing data, one cannot control the used hardware in the final IP Core.

	BRAM	DSP	LUT	FF
FIR filter from Vivado HLS	2	3	2473	3064
FIR filter from Vivado Catalog	0	23	380	373

Table 2: Hardware FIR filters Resource utilization

### Acceleration factor

In the following table 3 can be appreciated the different results in terms of processing time for the Vivado HLS generated FIR IP Core compared to the *Vivado Catalog* FIR IP Core and a Software developed FIR implementation.

Vivado General flow		
HW Vivado FIR(s)	HW Vivado HLS(s)	SW SDK application (s)
0,00354	0,00423	0,092

Table 3: FIR filter results with *Vivado General flow*

It is pretty obvious that having dedicated hardware will lead to time acceleration. The acceleration factors versus the Software FIR filter implementation are:

- *Vivado HLS IP Core*: **25,98**
- *Vivado Catalog FIR IP Core*: **21,74**

*Vivado Catalog* IP Core shows the best numbers here. This may be due to its low number of interfaces (figure 22) and a better HDL code performance. It has the same structure than the *Vivado HLS* generated IP Core. Also, was seen that *Vivado HLS* sometimes adds *extra* logic that could be avoided by creating the IP from scratch by HDL languages. Obviously, Xilinx IP is highly configurable. Thanks to this, it does not modify the FIR filter coefficients through an AXI4-Lite interface, they can be modified by *generics* in the Vivado *GUI*.

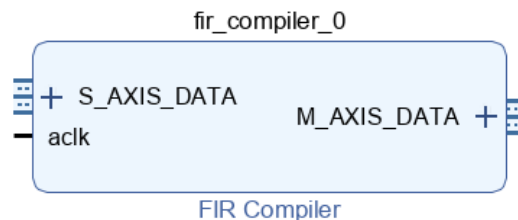


Figure 22: Vivado Catalog FIR IP Core

## PYNQ design flow

Using the *PYNQ* framework, the processing times for the different implementations were checked.

The results for the *PYNQ* development flow are summarized in the table 4. Here there were also three implementations:

1. FIR IP Core generated by Vivado HLS called by Python, using *Overlays*. (see figure 14).
2. FIR IP Core taken from Vivado Catalog, called by Python, using also *Overlays*.
3. Python implementation of the FIR filter.

PYNQ		
HW Vivado FIR(s)	HW Vivado HLS(s)	SW Python (s)
0,004017	0,005923	0,107

Table 4: FIR filter results with *PYNQ development flow*

In this case, the numbers are a bit worse when it comes to processing time. This is due to the *Linux Operating System* that is running in the *PYNQ* development tool. It is obvious that having these kind of Software Application running on top of the hardware design, will lead to more complex designs, thus, having greater processing times.

Still, the processing times are much better with an RTL FIR design with a whole *Linux* environment than a Software implementation on a *Baremetal* solution.

The acceleration factors in this case compared to a Software Python implementation are:

- *Vivado HLS IP Core*: **26,63**
- *Vivado Catalog FIR IP Core*: **18,06**

## 4.2 Half float

In this case, two IP Cores were implemented both using the Vivado HLS.

1. **Half** precision floating point multiplication IP Core.
2. **Single** precision floating point multiplication IP Core.

When it comes to *Half precision floating point* multiplication, the same behaviour as in the FIR implementation is shown.

### Timing/Critical path

The data path delays for these implementations were:

- **Half** =  $8.910ns \rightarrow f_{max} \approx 112MHz$
- **Single** =  $8.510ns \rightarrow f_{max} \approx 117MHz$

Also the latencies were worse for the Half multiplication:

- **Half** =  $9000 \text{ clock cycles} \approx 90\mu s$
- **Single** =  $3000 \text{ clock cycles} \approx 30\mu s$

The timing results are also conditioned by the hardware utilization. In the table 5 the hardware utilization will be shown and discussed.

### Resource utilization

As the implementation of the Half precision float is relatively new, the Vivado HLS tool uses less DSPs for it, thus, giving worse numbers compared to a well known single precision float data type.

	BRAM_18K	DSP48E	FF	LUT
Half	0	2	623	2022
Single	0	4	568	935

Table 5: Utilization report comparing half and single precision floating point multiplication

Having this kind of difference in the hardware utilization, lead to big processing time differences as shown as in the table 6. One may think that having a half precision floating point multiplication will be faster, but in this case, as it does not have specific hardware like DSPs, lead to worse processing times.

In order to use DSPs, the *Synthesis* tool needs to know exactly that this block wants to be used. In other words, a specific RTL code, DSP instantiation must be done. If the *Synthesis* tool does not understand that multiplication process, it will use a combination of *LUTs* and *Flip-Flops* like in this case.

Clock signal (MHz)	Single ( $\mu\text{s}$ )	Half ( $\mu\text{s}$ )
100	75	178
200	58	128
250	55	121

Table 6: Processing times of half and single precision floating point multiplication

And when it comes to errors, both IPs were designed with a  $100\text{MHz}$  clock in mind. Even though a  $250\text{MHz}$  clock signal was selected in the *PS* side and the timing was not successful (there were *timing violations*), both IPs did not have errors or in other words, the critical path was not met.

There was an evident difference in the results of both IPs, which was caused by the loss of precision of the *Half precision* floating point multiplication.

In this case the difference in the results between both IPs was:

$$\frac{1}{N} \sum_{i=1}^N (x_{h,i} - x_{s,i})^2 \approx 0.104\%$$

Where:

- $N$ : is the number of samples. In this case  $N$  was 1000.
- $x_{h,i}$ : is the  $i_{th}$  result of the *half precision* IP Core multiplication.
- $x_{s,i}$ : is the  $i_{th}$  result of the *single precision* IP Core multiplication.

The acceleration factor in this case was low. Software implementation of multiplying 1000 single precision floating point numbers was done in  $0.000035$  seconds.

This is mainly because this operation was not really demanding in processing time and also this SoC has a Single and double precision Vector Floating Point Unit (VFPU).

So, even if Vivado HLS *directives* are used for rolling the *for* function, Vivado HLS still adds some *logic* to the process, leading into a not fully optimized/parallel function.

### 4.3 Convolution

In this subsection the results obtained with the Image convolution IP Core will be discussed. First, the results of the Normal Convolution will be shown and afterwards the different implementations with the approximated arithmetics.

For the case of Normal convolutions, we will discuss first different parameters results obtained for an operation like convolution (input matrix size, data type). Critical path and Resource utilization will be discussed later on comparing the normal arithmetics with the approximated ones.

## Normal convolution

One of the main parameters in image convolution is the input and output image size and the data type of each pixel.

In this case, the input image size was implemented with two different resolutions:

- $320 \times 240 = 76,800 \text{ pixels}$
- $360 \times 640 = 230,400 \text{ pixels}$

When it comes to the data type, the first implementation was with 8-bit pixels (values between 0 to 255) for input/output image and kernel matrix, which is the basic operand for image convolution. In order to have an *Edge detection* application, the kernel must have negative values, so in this case a kernel with short data types (16-bit) was implemented. In the case that the sumation of a window gives negative values it will be set to 0 and if it gives values *greater than* 255 the pixel will be set to 255. And the last data type which was implemented was the float data type.

The figures 23 and 24 shows the different Area usage for the different data types and for 76,800 pixels and 230,400 pixels respectively.

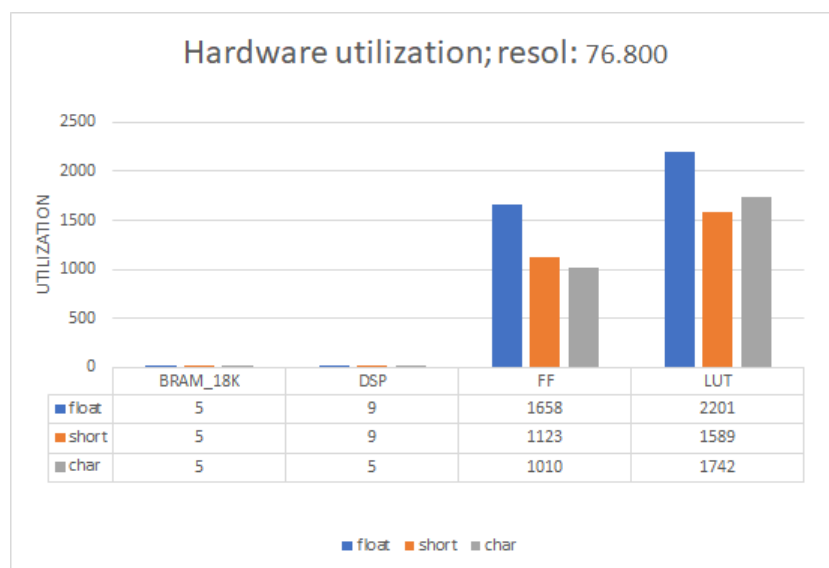


Figure 23: Area usage vs data type for 76,800 pixels

The results were as expected, the same number of *BRAMs* and *DSP*, because the Line Buffers and Memory Windows were defined with the same size in both implementations and the addition and multiplication process can be done with the same number of *DSPs*.

When it comes to processing times, the figure 25 summarizes the different implementations. Having an input image with higher resolution lead to higher processing times. This relation is completely **linear**, as can be seen in the case of *char* data type.



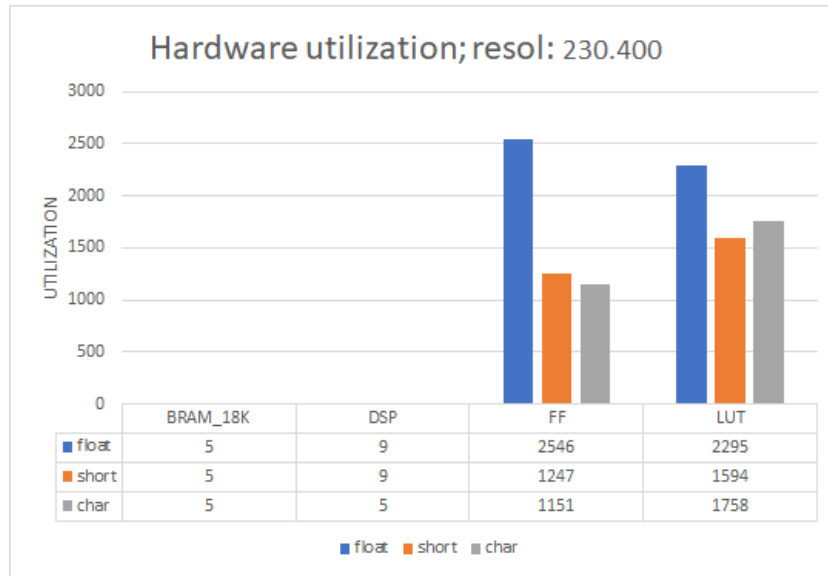


Figure 24: Area usage vs data type for 230,400 pixels

Both resolutions that were implemented have a  $\times 3$  relation, and the Hardware convolution for the 76.800 resolution and for the 230.400 resolutions are 0,007244 and 0,021491 respectively, which is also a  $\times 3$  relation.

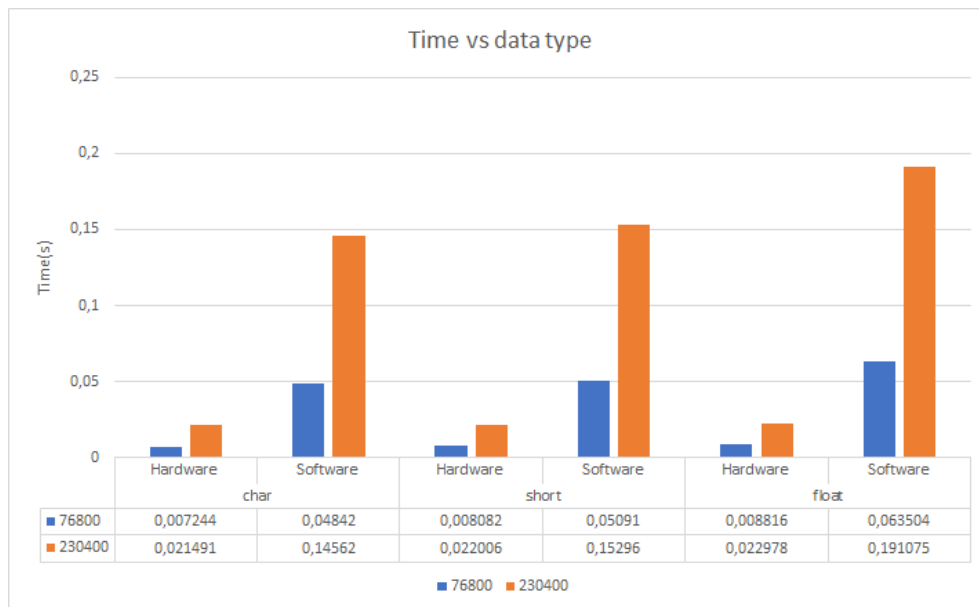


Figure 25: Processing times vs data types

Regarding the **acceleration factor**, with the results of the figure 25 a hardware acceleration is pretty obvious for the different cases. The arithmetic mean is:

$$\frac{1}{N} \sum_{i=1}^N \frac{Software_{time}}{Hardware_{time}} \approx 7.028$$



(a) Original image



(b) Convolved image

Figure 26: image convolved by an edge detection kernel

## Approximate arithmetics

As in the section 3.3 was explained, the new trend in *Machine Learning* or *AI* is the approximated arithmetics, and as the convolution process [5, 7] has many multiplications and additions, both approximated arithmetics have been implemented in the convolution process with interesting results.

Three different Convolution IP were implemented:

- **Normal** convolution IP.
- **Approximated multiplication** convolution IP.
- **Approximated addition** convolution IP.

The implementations of the approximated adder and multiplier were in *Verilog* and were instantiated in the HDL code generated by the HLS tool.

## Timing/Critical Path

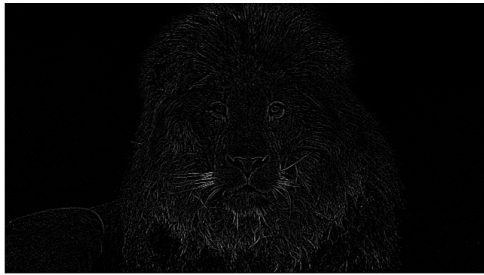
When it comes to the critical path, the approximated arithmetics didn't give a very promising results compared to the exact ones:

- **Normal** =  $9.870ns \rightarrow f_{max} \approx 101.3MHz$
- **Approx. Adder** =  $11.570ns \rightarrow f_{max} \approx 86.4MHz$
- **Approx. Multiplier** =  $16.89ns \rightarrow f_{max} \approx 59.2MHz$

The Master Thesis of *Imanol Etxezarreta* [5] showed a similar data path delays  $\approx 10ns$  for approximated multiplier, so adding the logic for multiply negative numbers and the interfaces of the IP, lead to have  $16ns$ .

With the same input image as shown in the figure 26, the approximated arithmetics implemented in the convolution give the results shown in the figure 27.

The approximate multiplier was not good enough for this application due to its critical path. That's why the image in 27 seems like "decentralized".



(a) Convolution with approximated adder

(b) Convolution with approximated multiplier

Figure 27: Images of approx arithmetics applied into edge detection

## Resources

The same thing that happened to the Half precision multiplication IP happens here for the convolution process. Once modified the RTL code generated by Vivado HLS, Vivado starts using more LUTs instead of DSP leading to greater delays in the logic.

	BRAM	DSP	LUT	FF
Normal	2.5	8	553	223
Approx Add	2.5	13	714	295
Approx mul	2.5	4	1155	389

Table 7: Resource utilization for different convolution implementations

## Acceleration factor

When it comes to the acceleration factor for the approximated arithmetics compared to the pure ones, the table 8 shows that the processing times were quite similar.

Normal	Add	Mult
0,021719s	0.021728s	0.021670s

Table 8: Processing times for different convolution implementations

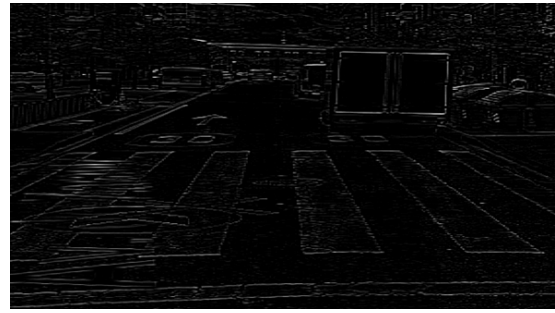
- *Approx. Multiplier* = 1.0004
- *Approx. Adder* = 0.9974

## Precision

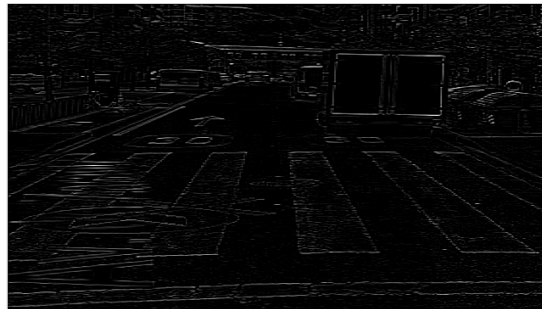
In this case, a comparison with multiple images was done just for the approximated adder. The following figures show some examples.



(a)



(b)



(c)

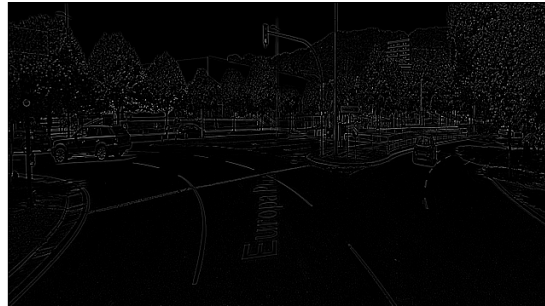
Figure 28: (a) input image. (b) Normal convolution. (c) Approximated adder convolution



(a)



(b)



(c)

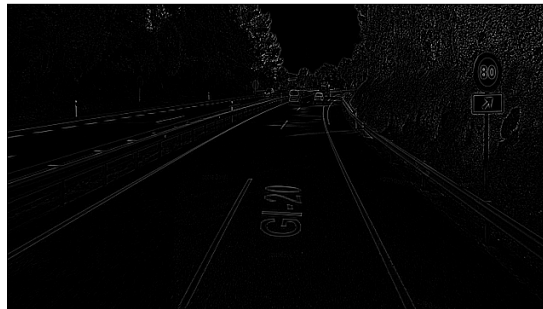
Figure 29: (a) input image. (b) Normal convolution. (c) Approximated adder convolution



(a)



(b)



(c)

Figure 30: (a) input image. (b) Normal convolution. (c) Approximated adder convolution

For the given kernel for edge detection the precision was of 100%. The thing is that for having edge detection, there are normally too many 0s in the output image and few pixels close to the maximum value. This leads to a very high precision for this type of kernels.

Looking at the truth table of the approximated multiplier, it gives wrong values for the high output numbers (in this case close to white pixel), that's why convolving the input image by positive kernels, for instance the Identity kernel, lead to lost in precision ( $\approx 95\%$ ). It can be appreciated in the figure 31 that in the *white* pixels (top corners), there is a slight difference in the output images. The approximated multiplication give well the values because the critical path was met when multiplying negative values.

$$1 - \frac{1}{N} \sum_{i=1}^N (x_{normal\ conv} - x_{approx\ adder})^2 \approx 0.954\%$$

$$1 - \frac{1}{N} \sum_{i=1}^N (x_{normal\ conv} - x_{approx\ multiplier})^2 \approx 0.942\%$$



(a)



(b)



(c)

Figure 31: (a) Normal convolution (b) Approx. multiplier convolution (c) Approx. adder convolution. The kernel in this case was the Identity

## 5 Conclusions

To end up with the study, conclusions about the results are presented, with the objective of assessing whether the usage of hardware accelerators and approximated arithmetics are beneficial for different applications like *Image Recognition* in Machine Learning. First, the conclusions about the different tools available for RTL development, that in this case were Vivado HLS and *PYNQ* framework. Then the conclusions about the hardware acceleration are exposed as well as the Vivado HLS tool for hardware development and then the conclusions about the approximated adders and multipliers.

When it comes to the HLS tool, it showed great advantages like fast developing, not just the process that wants to be implemented in the PL logic, but also implementing different interfaces was relatively fast. Apart from fast developing, Vivado HLS also creates C function drivers for the given IP in order to rapidly communicate with it from the Processing System. This was really helpful for rapidly write and read different registers of the given IP Core. On the other hand, not having the total control on the generated hardware can lead to *extra* logic in the IP, giving worse performance of the hardware.

As regard the *PYNQ* framework, as Python is a really hot topic right now for Machine Learning applications, it is nice to have this kind of working environments where one can program the FPGA using Python. But obviously, running a *Linux* application on top of the hardware will always lead to greater processing times. A Standalone Application will be always be faster in terms of performance. But when it comes to rapidly modify the bitstream of the FPGA, the *PYNQ* framework offer great advantages here as well as using Python libraries.

Regarding the hardware acceleration, it is obvious that they offer great advantages when it comes to processing times for real time applications. In today's real time processing systems, these kind of System-on-Chips like the Xilinx Zynq-7000, can offer a very interesting interaction between software and hardware, giving the possibility to decide which process is executed in hardware or in software. This kind of "*feature*" of this All-in hardware will play an important role in the near future for Real Time processing systems [1].

To sum up with the hardware accelerators, depending on the application and the specifications, they could be very beneficial. Again, thanks to the System-on-Chip Architecture, one can decide which functions are programmed in the PL side of the SoC, thus, having an improvement in most of the cases.

From the speed point of view, this approximate adder that was instantiated in the IP was always faster than the accurate one. Regarding the area, the normal adder and the approximated adder showed very similar numbers. As always, there will be a relationship, begin normally the fastest adder the one that require higher area.

To end up with the multiplier's conclusions, the studied approximate multiplier do not show to be promising in almost any aspect that has been able to study. Even though that the it showed the best numbers compared to others [5], still was worse than the accurate version.



---

## References

- [1] Alima Damak, Mohamed Krid and Dorra Sellami Masmoudi. Neural Network Based Edge Detection with Pulse Mode Operations and Floating Point Format Precision.
- [2] Jamie Schiel and Dr. Andrew Bainbridge-Smith. Efficient Edge Detection on Low-Cost FPGAs. Improving the efficiency of edge detection in embedded applications such as UAV control.
- [3] The Zynq Book. Embedded Processing with the ARM Cortex-A9 on the Xilinx Zynq-7000 All Programmable SoC.
- [4] Image Filtering and Edge Detection. Eng.Mohamed Hisham.  
[https://sbme-tutorials.github.io/2018/cv/notes/4\\_week4.html](https://sbme-tutorials.github.io/2018/cv/notes/4_week4.html)
- [5] Imanol Etxezarreta Martinez. Approximate arithmetic units under voltage under-scaling. Master Thesis, 2020-2021. Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona. Universitat Politècnica de Catalunya.
- [6] Vivado Design Suite Tutorial: High-Level Synthesis. A collection of smaller tutorials that explain and demonstrate all steps in the process of transforming C, C++ and SystemC code to an RTL implementation using High-Level Synthesis.
- [7] P. Kulkarni, P. Gupta, and M. Ercegovac. Trading accuracy for power with an underdesigned multiplier architecture. In *2011 24th International Conference on VLSI Design*, pages 346–351, 2011.
- [8] *PYNQ*. An open-source project from Xilinx that makes it easier to use Xilinx platforms.  
<http://www.pynq.io/>
- [9] **x87**, a floating point-related subset of the x86 architecture instruction set.  
<https://en.wikipedia.org/wiki/X87>