



UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA



Escola Tècnica
Superior d'Enginyeria
Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

HLS Kernel Execution on a Multi-FPGA Prototype

Master Degree Final Work

Master Degree in Computer Engineering

Author: Rosa Nuzzo

Tutor: José Flich Cardo

First External Tutor: Alessandro Cilardo

Director Experimental: Rafael Tornero Gavilá

Course 2020-2021

Contents

Contents	ii
List of Figures	iv
List of Tables	vi
Listings	vii
Glossary	ix
Abstract	1
1 Introduction	3
1.1 MANGO prototype	5
1.2 Goals and Motivations of this Work	6
1.3 Structure of the Document	7
2 Background and Tools	8
2.1 FPGA	8
2.1.1 FPGA Architecture	9
2.1.2 Applications	10
2.1.3 FPGA-based computing engines	11
2.2 High Level Synthesis	13
2.2.1 Flow	13
2.2.2 Coding	15
2.2.3 Optimization	16
2.3 Convolution	18
2.3.1 Convolution 2D	20
2.4 Tools	21
2.4.1 Xilinx Vivado Design Suite	21
2.4.2 proFPGA	26

2.4.3	HLS application	31
3	Single-FPGA	39
3.1	Design	39
3.1.1	MMI64 host interface	40
3.1.2	Convolution IP core	45
3.1.3	Memory	47
3.2	Test	49
3.2.1	System ILA	49
3.2.2	Programming the FPGA	50
3.2.3	Testbench	51
3.2.4	Run test	61
3.2.5	Hardware manager	65
4	Multi-FPGA	67
4.1	Design	67
4.2	Test	68
4.2.1	Inizialize FPGA	68
4.2.2	Test bench	69
4.2.3	Run test	75
5	Conclusions	77
	Bibliography	78

List of Figures

1.1	MANGO perspectives	4
1.2	MANGO prototype with its components	5
1.3	Photo of an HN in the MANGO prototype	6
2.1	FPGA Architecture	9
2.2	HLS steps	14
2.3	Vivado HLS Design Flow	23
2.4	Modular hardware	26
2.5	ProFPGA software and libraries	28
2.6	MMI64 communication channels	30
2.7	Synthesis Summary Report	36
2.8	Co-somilation report	37
2.9	Export RTL panel	38
3.1	Vivado design	40
3.2	Vivado mmi64 host interface module design	41
3.3	Profpga_ctrl module	42
3.4	MMI64_axi_master module	43
3.5	Profpga_clocksync module	44
3.6	Convolution IP core	45
3.7	DDR4	47
3.8	Address editor	48
3.9	Ila core	50
3.10	Scan MMI64 connected device	54
3.11	Application execution	63
3.12	Relation between the configuration and the efficiency	64

3.13	Ila settings	65
3.14	Trigget setup	65
3.15	Waveform	66
4.1	design multi-FPGA	67
4.2	Configuration mmi64_host_interface multi-FPGA	68
4.3	Scan MMI connected device	70
4.4	Pipelined execution	71
4.5	Multi-FPGA execution	76

List of Tables

2.1	XCKU115 proFPGA Characteristics	31
3.1	Input data test	62

Listings

2.1	pragma HLS loop_merge example	18
2.2	HLS example	32
2.3	Test bench	35
3.1	Convolution pragmas to create AXI interface	46
3.2	Configuration file	53
3.3	Loading configuration file	53
3.4	Components scanning	54
3.5	Modules identification	55
3.6	Open a connection to AXI Master module domain	55
3.7	Enable interrupts	56
3.8	Application flow	57
3.9	Function used to copy data from CPU memory to FPGA memory.	58
3.10	Function to input the necessary data to the convolution IP core	59
3.11	Start FPGA execution	60
3.12	Copy data from FPGA memory to CPU memory	60
3.13	Close connection to a MMI64 domain	61
3.14	Execution time	64
4.1	Configuration file multi_fpga	69
4.2	Identify AXI_master module	70
4.3	Buffer FPGA available	71
4.4	Assign configuration to FPGA	73
4.5	Read interrupt and CPU compiling	74
4.6	Push in the FPGA buffer	74
4.7	Push from the FPGA buffer	75
4.8	Checking that all executions have finished	75

Glossary

ACM	Advanced Clock Management.
API	Application Programming Interface.
ASIC	Application Specific Integrated Circuit.
AXI	Advanced eXtensible Interface.
CDFG	Control Data Flow Graph.
CLB	Configurable Logic Block.
CPU	Central Processing Unit.
DDR	Double Data Rate.
DFG	Data Flow Graph.
DMBI	Device Message Box Interface.
DSP	Digital Signal Processor .
ESL	Electronic system level.
FIFO	First In Frist Out.
FPGA	Field Programmable Gate Array.
GAP	Group of Parallel Architectures.
GN	General purpose Node.
GPU	Graphics Processing Units.

HDL	Hardware Description Language.
HLS	High Level Synthesis.
HN	Heterogeneous Node.
HPC	High Performance Computing.
I/O	Input/Output.
ILA	Integrated Logic Analyzer.
IP	Intellectual Property.
LUT	Look Up Table.
MMI	Module Message Interface.
MUX	Multiplexer.
PCIe	Peripheral Component Interconnect Express.
RTL	Register Transfer Level.
SDRAM	Synchronous Dynamic Random Access Memory.
SSI	Small Scale Integration.
SV	SystemVerilog.
TCL	Tool Command Language.
UPV	Universitat Politècnica de València.
VHDL	VHSIC Hardware Description Language.
VHSIC	Very High Speed Integrated Circuits.

WDB Waveform Database.

XDC Xilinx Design Constraints.

Resumen

El objetivo de este proyecto es permitir el uso de un prototipo multi-FPGA como plataforma de cálculo en la que se puedan ejecutar simultáneamente núcleos FPGA codificados en síntesis de alto nivel. El proyecto se divide en dos pasos:

En el primer paso, se desarrolla el soporte para compilar un kernel HLS desnudo para un prototipo FPGA, generando un núcleo IP. A continuación, se añade el núcleo IP generado con bloques lógicos de comunicación y memoria (DDR, AXI) en un diseño desarrollado para un dispositivo FPGA. Todos los dispositivos incluidos en la FPGA serán accesibles a través de las direcciones de memoria. En el lado del host, éste utiliza el protocolo MMI para lanzar el flujo de bits, escribir en la memoria, lanzar el kernel, leer la memoria y comprobar la corrección. El acceso a la memoria, la sincronización del kernel y el paso de argumentos se realizará a través de direcciones AXI mapeadas en memoria. Por último, se realiza la validación de los pasos anteriores mediante una ejecución satisfactoria.

En el segundo paso, el diseño a nivel de sistema se adapta a una arquitectura multi-FPGA interconectada. Cada FPGA será accesible individual y simultáneamente desde el mismo host, utilizando el protocolo MMI.

Este proyecto permitirá utilizar el prototipo como demostrador de sus capacidades computacionales.

Abstract

This project aims to enable the use of a multi-FPGA prototype as a to compute platform where FPGA kernels coded in High Level Synthesis can be run concurrently. The project is divided into two steps:

In the first step, develop support to compile a bare HLS kernel for one FPGA prototype, generating an IP core. Next, the generated IP core is added with communication and memory logic blocks (DDR, AXI) in a design developed for an FPGA device. All the devices included in the FPGA will be accessible through memory addresses. On the host side, the host uses the MMI protocol to launch the bitstream, write memory, launch the kernel, read memory and check correctness. Memory access, kernel synchronization and argument passing will be performed through memory-mapped AXI addresses. Finally, the validation of the previous steps is performed by mean of a successful run.

In the second step, the system-level design is adapted for an interconnected multi-FPGA architecture. Each FPGA will be individually and simultaneously accessible from the same host, using the MMI protocol.

This project will allow the prototype to be used as a demonstrator of its computational capabilities.

CHAPTER 1

Introduction

High Performance Computing (HPC) is always pushing technology for the quest of higher performance. However, the end of the era of transistor scaling and frequency increasing makes difficult to provide sufficient performance for emerging large-scale dataset applications, within a reasonable energy budget with current HPC architectures.

To overcome this situation, and to keep increasing performance in an energy efficient manner, heterogeneity has been adopted as key factor to design and implement performance and energy efficient HPC systems. Reconfigurable devices such as Graphics Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs) have found their niche in these systems. Both GPUs and FPGAs have been proved to provide significant performance gains for HPC applications. In particular, FPGAs have been shown to provide high energy efficiency as well.

In last recent years, new European Projects have been founded by the European Commission with the focus on HPC architectures. Among them, the Group of Parallel Architectures (GAP) of the Technical University of Valencia (UPV) has been granted with the FET-HPC MANGO project [1], which end up in May 2019. The goal of this project consisted of developing and exploring new heterogeneous architectures for future HPC systems. In order to achieve such a goal, a large prototype infrastructure made of FPGAs was deployed and different HPC architectures were proposed, built into a prototype and emulated via the FPGAs of the MANGO infrastructure.

Although MANGO encompasses mainly the exploration of unconventional architectures, the MANGO prototype described in Section 1.1, has been conceived to play a twofold role. This situation is captured by Figure 1.1, highlighting that the same hardware platform can support both a physical compute platform and an emulation platform.

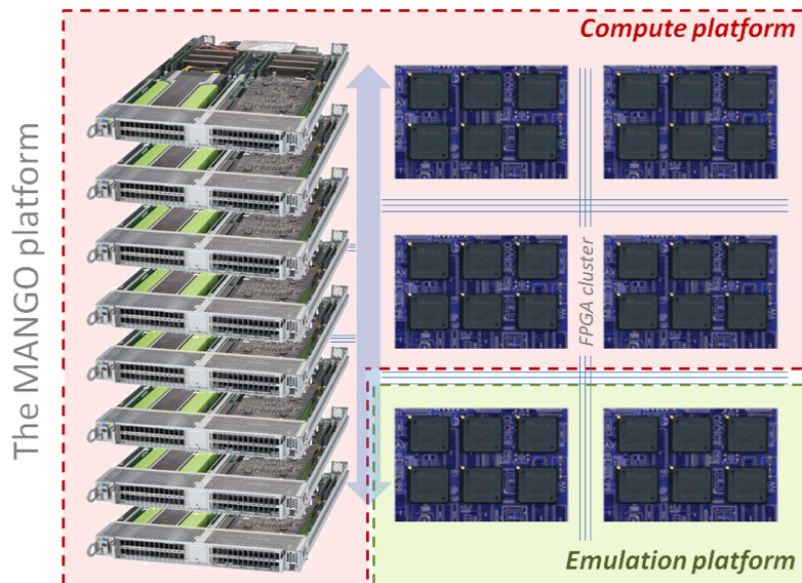


Figure 1.1: MANGO perspectives: I) computation and II) emulation. As a computation platform the focus is on bulk performance. As an emulation platform the focus lays on exploration and hardware solution prototyping.

As an emulation platform the focus is on architecture exploration, both at system and compute unit level, and validation of functional aspects. Here, performance numbers are inferred from suitable performance counters and evaluated in relative terms. Unlike, as a compute platform the focus is on bulk performance, thus the goal is to maximize the performance at system level in absolute terms, while keeping energy consumption in a manageable budget. For that reason, this latter type of platforms is being adopted and used massively in new HPC heterogeneous systems.

Enabling a reconfigurable compute platform made of multiple FPGAs involves two main challenges. The first one consists of reducing the programming complexity of these devices, which it can be done by means of programming models like OpenCL [2] or High Level Synthesis (HLS). The second one corresponds to the development of

software/hardware co-design support for managing the hardware, which includes support for efficient communication between the host and the different FPGA devices and tools to transform application kernels into hardware functions that can be uploaded to an FPGA.

1.1 MANGO prototype

Figure 1.2 shows a complete view of the MANGO prototype with its hardware components. Among them, the most relevant ones for this project are the General purpose Node (GN) and the Heterogeneous Node (HN).

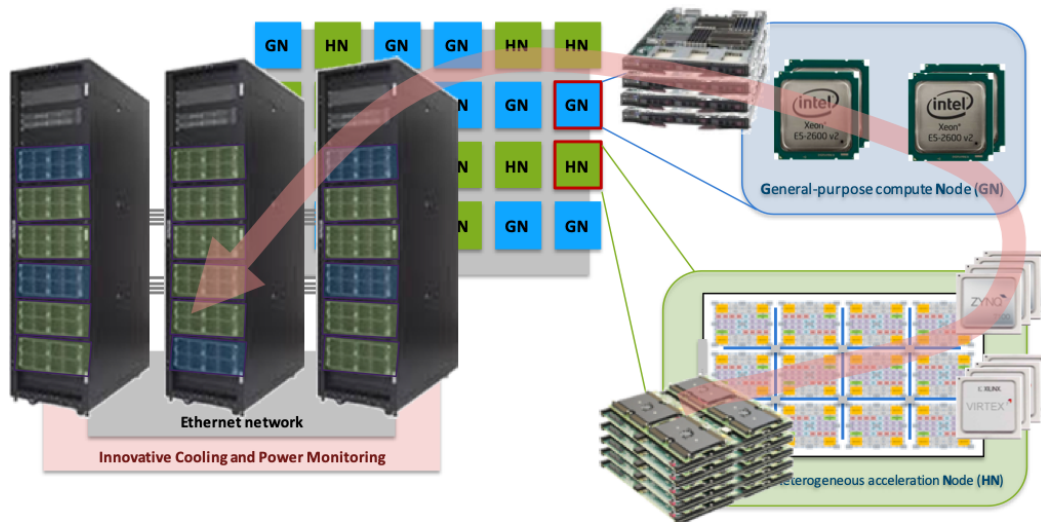


Figure 1.2: MANGO prototype with its components: I) General Purpose Node (GN), II) cluster of FPGAs, as known as Heterogeneous Node (HN), III) Ethernet interconnect and IV) Innovative cooling and monitoring subsystem.

The GN consists of a blade made of a high-end Intel Xeon E5 V3, 64 GB of DDR4 memory, 1 TB SSD hard disk and PCI Express (PCIe) connectivity. In this project the GN runs the low-level runtime system to support the managing of the FPGAs.

The HN, also as known as MANGO cluster, consists of 12 FPGAs and 22 GB of DDR3 and DDR4 memory. The cluster is also heterogeneous, since it is composed of different types of FPGAs: Xilinx Kintex Ultrascale (KU115) [3], Xilinx Virtex 7 Series (V2000T) [4], Xilinx Zynq 7000 SoC (Z100) [5] and Intel Stratix 10 (SG280) [6]. Every GN can access two different clusters through PCIe Gen3 x8 lanes, that is a total of 24

FPGAs and 44 GB DDR3/DDR4 memory. The Figure 1.3 shows a photo of how it looks currently. In that Figure, it can be seen the FPGAs and blue cables connecting FPGAs among them.



Figure 1.3: Photo of an HN in the MANGO prototype.

The whole MANGO prototype consisted of a total of 8 GNs and 16 HNs. However, the prototype has been split in different parts and each part has been delivered to a different partner of the MANGO consortium. UPV has received a half of the prototype, that is, 4 GNs and 8 HNs, which means a total of 96 FPGAs and 176 GB of device memory.

Initially, the MANGO prototype has been used completely as an emulation platform, however, there is an effort to adapt this huge system to a compute platform currently. Therefore, it motivates the development of this work as we show in Section 1.2.

1.2 Goals and Motivations of this Work

This master thesis project **aims to enable the use of a multiFPGA prototype as a to compute platform where FPGA kernels coded in High Level Synthesis can be run concurrently.** This goal fits clearly with the research interests of the GAP group, since it has been granted recently with two new projects where the MANGO prototype is targeted as a computation system. This is the case of the FET-HPC RECIPE [7]

and the ICT DeepHealth project [8]. Thus, as part of the activities of both projects, the MANGO prototype is being adapted from a strictly emulation platform to an actual compute platform.

As mentioned earlier, using a multi-FPGA system as a computational platform involves two challenges. This thesis work contributes to solving one of these challenges. A hardware design has been developed, which allows an optimized kernel to be loaded via the HLS. Moreover, software design to be able to make a host communicate with a set of FPGAs.

1.3 Structure of the Document

This thesis is structured in a logical and chronological order with respect to how it was approached in practice.

In Chapter 2 an overview of the technologies and concepts used are given in order to make the reading of the following Chapters clearer. Specifically, it will be explained what an FPGA is and the motivation for choosing this technology. Subsequently, what is High Level Synthesis and convolution (Section 2.2 and 2.3) is explained. This is because, in the design that was modelled in the following chapters, a convolutional IP core optimised through HLS was used. The IP core specifically was developed by the Group of Parallel Architectures (GAP) of UPV. Finally, there is a description of the tools used (Section 2.4).

The actual contribution of this thesis work is described in Chapters 3 and 4. Specifically, in Chapter 3, the various components of the hardware design developed (3.1) using a single FPGA are described in detail. In Section 3.2, the described design is tested using both a hardware approach and using screen printing. Chapter 4, is structured like the previous one but with the difference of using a multi-FPGA system.

Finally, the Chapter 5 presents the final considerations.

CHAPTER 2

Background and Tools

This chapter briefly discusses the fundamentals of the technologies and concepts that were applied in this Project. Section 2.1 describes what an FPGA is, illustrating the architecture and the essential components of which it is constituted (Subsection 2.1.1). It will describe the areas in which FPGAs can be used and the motivation for using them. Specifically, it will be shown the specific FPGA used in this Project (Subsection 2.1.2 and 2.4.2). Section 2.2 describes what HLS is and what the positive aspects of its use might be. In addition, the subsection 2.2.1 shows the HLS execution flow as well as some tricks to pay attention to in order to make the best use of the language and the optimisations that can be done with it (Subsection 2.2.2 and 2.2.3). Finally, a small example will be explained to understand better its working (Subsection 2.4.3). Section 2.3 describes what convolution is and how it works. Specifically, 1-dimensional and 2-dimensional convolution are illustrated. Finally, Section 2.4 discusses what tools were used in this Project.

2.1 FPGA

In industry, one of the key aspects is to reduce development and production time and introduce new products by reducing time-to-market. Also, there must be the lowest financial risk when a new product is introduced to propose and develop new ideas. FPGAs (Field-Programmable Gate Arrays) are a great solution to these time-to-market and risk issues because they provide instant production and prototyping at a meagre cost [9].

The use of FPGAs brings multiple advantages. The programmable nature of an FPGA allows manufacturers to correct errors and send patches or updates after the product has been purchased. Manufacturers also take advantage of this by creating their prototypes in an FPGA to be thoroughly tested and reviewed in the real world before sending the design to the IC foundry for ASIC production. This is because an ASIC can no longer be modified after it comes off the production line. They can also be used to implement causal logic and consequently as a replacement for SSI chips. Finally, a further advantage is the possibility to compile a program not via software but hardware.

2.1.1. FPGA Architecture

FPGAs are semiconductor devices based on an array of configurable logic blocks (CLBs) connected via programmable interconnects, which routes signals between the CLBs. Input/output (I/O) blocks interface between the FPGA and external devices.

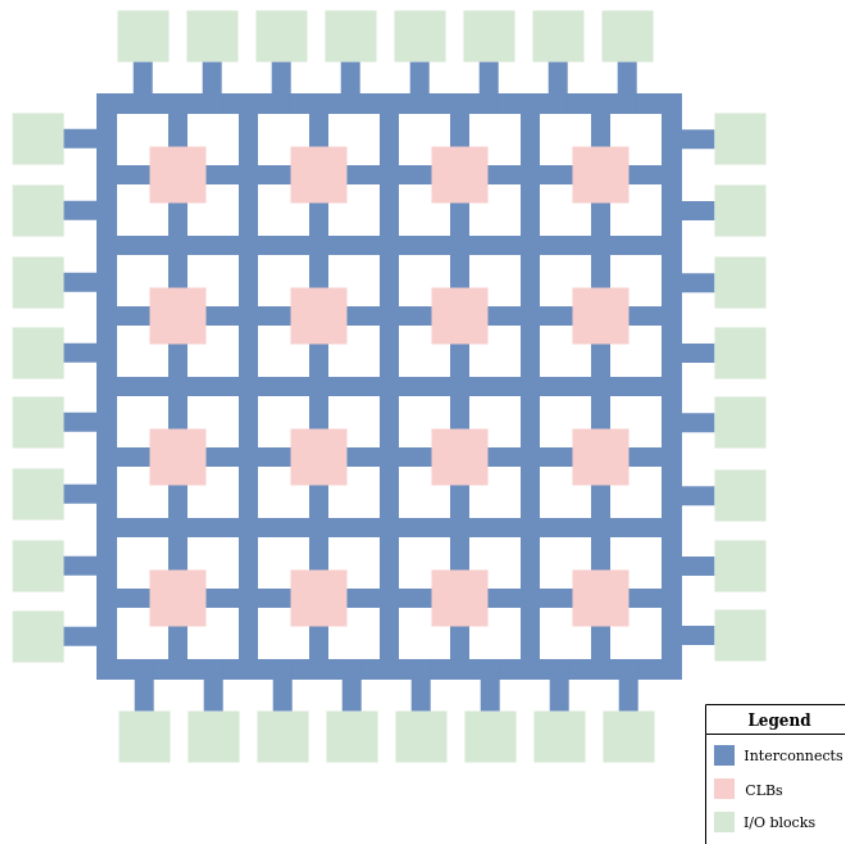


Figure 2.1: FPGA Architecture

Configurable Logic Block CLBs are the main components of an FPGA. They are the components that are programmed so that the FPGA can execute any program.

The CLB consists of a Look UP Table (LUT), a MUX (Multiplexer) and D flip flops.

- The LUT implements the combinational logic functions;
- the MUX is used for selection logic,
- moreover, the D flip flop stores the output of the LUT.

Interconnects The interconnection is a routing matrix comprising programmable switches and wires. The routing elements connect Input/Output blocks, logic blocks and between one CLB and another CLB.

Input/Output (I/O) Blocks Input/output blocks transfer data in and out of the FPGA. The I/O blocks can be configured according to the user's needs, depending on what they want to transmit and receive. They are similar to transceivers but operate at lower speeds and can maintain greater functional flexibility.

2.1.2. Applications

FPGAs can be used in many fields, including:

Prototyping The primary use of FPGAs is for prototyping applications. This is because FPGAs have a low cost, both in terms of implementation and time, and therefore provide significant advantages over other more traditional methods of prototyping hardware. Thus, the prototype version can be quickly implemented, and easy modifications can be made when needed.

Random logic implementation Random logic implementations are usually implemented with PALs, but if the speed of the circuit is not a problem with the speed of the circuit, it could be very advantageous to use FPGAs.

Replacing SSI chips for random logic FPGAs can also replace the SSI chips found in many commercial products, reducing the area used by these chips on circuit boards.

On-site hardware reconfiguration Advantageous use of FPGAs is when it is necessary to replace components in a machine already in operation. A system that contains a certain number of FPGAs connected via programmable interconnection provides a high degree of flexibility in increasing the functional behaviour of the circuits supplied by the board. The type of FPGA most suitable for this type of use contains re-programmable switches.

FPGA-based computing engines The exciting use of FPGAs is programming them so that a program is not compiled in software but hardware. This has two advantages:

The first one means that there is no need for instruction fetching as in traditional microprocessors; this is possible because the hardware incorporates the instructions.

The second advantage is due to the high levels of parallelism and consequent increase in speed.

In this Project, FPGAs are used according to the latter application type. Next, this type of application is discussed in more detail in Subsection 2.1.3.

2.1.3. FPGA-based computing engines

In this work, the FPGA has been used in the context of computing engines. An FPGA-based computing engine can be seen as a platform that includes a hardware element and can fully implement a software algorithm in part or whole. A programmable platform can consist of a single FPGA or several FPGAs.

On the one hand, FPGA-based computing engines act as a bridge between programmable software systems mounted on traditional microprocessors and customised hardware function-based application-specific platforms. On the other hand, progress in the design tools and technology for FPGA-based platforms permits the quick generation of hardware-accelerated algorithms.

The design for FPGAs can be seen as the design of embedded processors. Simulation tools can be used to debug and verify the functionality of an application before programming the physical device, and there are currently several tools that allow this. Although the tools may be more complex and may take longer to develop, the design can be seen as simple software development rather than hardware development with its respective advantages.

Indeed, in cases where optimisations are required to handle higher resolutions or higher bandwidth signals, these performance gains may be computational or may require entirely new approaches. Implementing a system directly into an FPGA removes or minimises the need for an instruction-based processor. This results in increased performance at a reduced cost compared to using DSP chips and ASICs.

Typically, hardware description languages such as Verilog or VHDL are used to program FPGAs. As will be shown in section 2.2, design tools have been developed that support high-level languages such as C/C++. These tools provide a process whereby an application can be transformed from its high-level description into an optimised low-level representation, which can be adapted according to the platform used. Using the C/C++ language, a software-based approach is a significant advantage because it quickly tries out different solutions. Also, making low-level changes can be very complex, and FPGAs help solves this problem because they can support such applications.

There are advantages and disadvantages to using various types of system processors. For example, although DSPs have low initial use in tools, they need a specific design and programming techniques at the assembly level. On the other hand, FPGAs need more design time and tool skills, mainly if hardware design languages are used as the primary design input method. However, compared to using ASICs, they are a lower-cost, lower-risk solution. Furthermore, if C-based design tools are used with FPGAs, the disadvantages listed above can be significantly reduced.

In conclusion, using FPGAs as a computing engine allows engineers to implement applications, or prototype applications, more quickly without the need to understand all the intricate details of the target, and to perform optimisations, while at the same time providing low-level features in order to extract the highest possible performance[10].

2.2 High Level Synthesis

As the complexity of applications has increased, the search has begun for new tools to work at a high level, improving performance over register transfer level (RTL). High-Level Synthesis (HLS) comes to the rescue in this regard. HLS makes it possible to synthesise high-level specifications and transform them into low-level RTL specifications; it also makes it possible to optimise the Synthesis, considering the performance, power, and cost requirements of a particular system [11].

HLS allows designers to specify design functionality in a high-level programming language and then generate a system-specific IP core. Unlike RTL IP, which has a static microarchitecture, behavioural IP can be adapted to different deployment technologies or system requirements. This reduces the time spent searching for the best compromise between area, power, and performance. Modern FPGAs incorporate many IP components, such as arithmetic function units, memories, processors, and system buses. These predefined blocks can be modelled in advance for each FPGA platform. In addition, FPGAs are often used for systems where time-to-market is critical. Thus, designers accept an increase in performance, power, or cost to reduce design time. With modern HLS tools, the designer makes this trade-off, allowing a significant reduction in design time. Finally, FPGAs are being used in reconfigurable computing platforms to accelerate High-Performance Computing (HPC) applications. However, most developers are not familiar with RTL programming in VHDL or Verilog, so providing a highly automated C/C++ to FPGA compile/synthesise flow is essential. [12].

2.2.1. Flow

The figure 2.2, shows the creation flow of an RTL. The development is divided into 5 phases, starting with specifications written in a high-level language and finally generating a module written in a low-level language and ready to be imported into the desired platform. The phases are explained below.

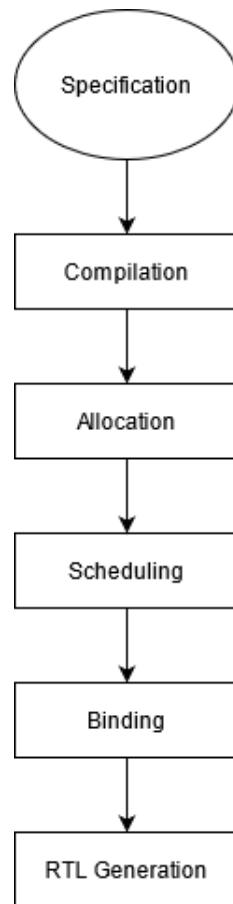


Figure 2.2: HLS steps

Compilation The first step is to transform the specification into a formal representation.

In this phase, optimisations can be made, such as eliminating dead-code or eliminating false dependencies. The output of the compilation shows the dependencies of the data. This formal representation consists of creating a DFG in which nodes represent operations and arcs represent input, output, and temporary variables. The DFG model can be extended with a CDFG, in which nodes represent basic blocks. A CDFG looks at the data dependencies within the basic blocks. The positive side of using CDFGs is that they can represent cycles with unlimited iterations. The problem is that only parallelism within basic blocks is considered. In order to consider the parallelism between the basic blocks, further analysis must be performed.

Allocation The allocation defines the type and number of hardware resources (e.g., functional units or connectivity components) required to meet design constraints. Some

components may be added during scheduling and binding activities. The components are selected from the component library RTL.

Scheduling The specifications' operations may terminate within one clock cycle or multiple clock cycles, depending on the functional units to which they have been mapped. Operations can be concatenated so that the output of one operation goes into the input of the next operation. Operations can also be executed in parallel, but only if they have no data dependencies.

Binding All operations in the specification model must be associated with one functional unit; the binding algorithm must optimise the choice when associated with more than one functional unit. Storage and functional unit binding depend on connectivity binding, which requires that the transfer from one component to another is done with a bus or multiplexer. Ideally, high-level Synthesis considers delays due to connections so that subsequent phases can best optimise the design. An alternative way to proceed is to specify the entire architecture during the allocation phase to use initial optimisations in the binding and scheduling phase.

Generation The last step is to generate a model RTL by considering the design choices made in the previous allocation, scheduling, and critical steps.

2.2.2. Coding

As already mentioned (section 2.2), the HLS allows the programmer to use C/C++ to generate components written in a low-level language. Various precautions need to be taken to ensure that there are no errors or malfunctions.

System Calls System-calls are tasks executed in the context of the operating system that hosts the C/C++ program, as a consequence they cannot be synthesized.

Dynamic Memory Usage Memory allocation dedicated system-calls, such as `malloc()`, use resources held by the operating system's memory and are created and released at runtime. Thus, to perform Synthesis, the Project must be completely autonomous, specifying all the resources required.

Pointer Limitations In the High Level Synthesis workflow general pointers (i.e. function pointers) casting is not supported, however native C/C++ types can always be cast between each other. Furthermore, arrays of pointers are also supported, given that each pointer points to a scalar or array of scalars. However, arrays of pointers cannot point to other pointers.

Recursive Functions Synthesis of functions that form unlimited recursion is not permitted. Moreover, queued recursion is not supported, in case of countless function calls. Templates adopted in C++ programming language tail recursion is often used, these templates can be synthesized in tail recursion designs.

2.2.3. Optimization

With HLS, optimizations can be made and used to produce a microarchitecture that can obtain desirable results in terms of area and performance goals. Optimisation directives can be added directly into the source code as compiler pragmas using various HLS pragmas or using the TCL `set_directive` commands to apply the optimisation directives in a TCL script by a solution during compilation.

Several optimisation directives can be applied to the Project:

Throughput Presents the main optimisations in the order in which they are typically used: pipelining tasks to improve performance, improving the flow of data between, and optimising structures to improve problems that may limit performance.

Latency Uses latency constraint techniques and removal of transitions to reduce the number of clock cycles required for completion. To reduce the number of clock cycles required for completion.

Area Focuses on how operations are implemented - controlling the number of operations and how these operations are implemented in the hardware is the primary technique for improving the area.

Logic Discusses optimisations concerning the implementation of RTL.

2.2.3.1. Optimization pragma example

A design can be optimised by reducing the clock numbers of an operation, thus optimising the latency. The HLS provides many pragmas for this purpose. An example of this type of optimisation is loop merging. The type of pragma that is useful for this application is the **pragma HLS loop_merge**.

Information about other pragmas can be found in the official Xilinx documentation [13], the pragmas are constantly evolving, so it is essential to use the proper documentation according to the tool used.

pragma HLS loop_merge

```
#pragma HLS loop_merge force
```

Merge consecutive loops into a single loop to reduce overall latency, increase sharing, and improve logic optimisation. Loop Merge:

- Reduces the number of clock cycles required in the RTL to transition between loop-body
- Allows loops to be implemented in parallel (if possible).

The LOOP_MERGE pragma will attempt to merge all loops within the scope in which it is placed. For example, if a pragma LOOP_MERGE is applied in the loop's body, the

Xilinx HLS tool will apply the pragma to any sub-loop within the loop but not to the loop itself. All loops inside, but not themselves, are merged by using the force option. The rules for joining loops are:

- If the loop bounds are variables, they must have the same value (number of iterations).
- If the loop bounds are constant, the maximum value of the constant is used as the boundary of the merged loop.
- Loops with variable limits and constant limits cannot be merged.
- Code between loops to be merged cannot have side effects. Multiple executions of this code should generate the same results (a=b is allowed, a=a+1 is not).
- Loops cannot be merged when they contain FIFO reads. Merging changes the order of the reads. Reads from a FIFO or FIFO interface must always be in sequence.

An example of the use of this pragma is illustrated in the following listing 2.1.

```
1 void foo (num_samples, ...) {  
2 #pragma HLS loop_merge  
3 int i;  
4 ...  
5 loop_1: for(i=0;i< num_samples;i++) {  
6 ...  
7  
8 loop_2: for(i=0;i< num_samples;i++) {  
9 #pragma HLS loop_merge force  
10 ...
```

Listing 2.1: pragma HLS loop_merge example

In the example, the two loops will be merged into a single loop. Moreover, all the engaged loops are merged in the second loop because the force option was used.

2.3 Convolution

Convolution is a technique widely used in signal processing, image processing and other fields of engineering/science. In image processing, convolution can be used to apply a

filter to an image. Convolution allows various types of filters to be used to extract different aspects and characteristics from an image. Similarly, in Convolutional Neural Network, different features are extracted using filters whose weights are learned automatically during training.

There are some advantages to doing convolution, such as sharing weights and invariant translation. Convolution also takes into account the spatial relationship of pixels. These could be very useful, especially in many computer vision tasks, as these tasks often involve identifying objects in which specific components have a specific spatial relation to other components.

The following Section shows an example to understand better how convolution works.

2.3.0.1. Convolution 1D

Convolution is executed by multiplying and accumulating the instantaneous values of superimposed samples corresponding to two input signals, inverted.

To better understand the convolution, below is an example. Given two signals, $X_1[n]$ and $X_2[n]$, where $X_1[n]$ is called the kernel and $X_2[n]$ are the inputs, what you want to do is convolve a kernel with information.

X_1 <table border="1" style="display: inline-table; border-collapse: collapse;"><tr><td style="padding: 2px 10px;">3</td><td style="padding: 2px 10px;">2</td><td style="padding: 2px 10px;">1</td></tr></table>	3	2	1	X_1 <table border="1" style="display: inline-table; border-collapse: collapse;"><tr><td style="padding: 2px 10px;">3</td><td style="padding: 2px 10px;">2</td><td style="padding: 2px 10px;">1</td></tr></table>	3	2	1		
3	2	1							
3	2	1							
X_2 <table border="1" style="display: inline-table; border-collapse: collapse;"><tr><td style="padding: 2px 10px;">1</td><td style="padding: 2px 10px;">2</td><td style="padding: 2px 10px;">3</td><td style="padding: 2px 10px;">4</td></tr></table>	1	2	3	4	X_2 <table border="1" style="display: inline-table; border-collapse: collapse;"><tr><td style="padding: 2px 10px; background-color: #e0e0ff;">4</td><td style="padding: 2px 10px; background-color: #e0e0ff;">3</td><td style="padding: 2px 10px; background-color: #e0e0ff;">2</td><td style="padding: 2px 10px; background-color: #e0e0ff;">1</td></tr></table>	4	3	2	1
1	2	3	4						
4	3	2	1						
X_1 <table border="1" style="display: inline-table; border-collapse: collapse;"><tr><td style="padding: 2px 10px;"></td><td style="padding: 2px 10px; background-color: #e0e0ff;">3</td><td style="padding: 2px 10px;">2</td><td style="padding: 2px 10px;">1</td></tr></table>		3	2	1	X_1 <table border="1" style="display: inline-table; border-collapse: collapse;"><tr><td style="padding: 2px 10px;"></td><td style="padding: 2px 10px; background-color: #e0e0ff;">3</td><td style="padding: 2px 10px; background-color: #e0e0ff;">2</td><td style="padding: 2px 10px;">1</td></tr></table>		3	2	1
	3	2	1						
	3	2	1						
X_2 <table border="1" style="display: inline-table; border-collapse: collapse;"><tr><td style="padding: 2px 10px;">4</td><td style="padding: 2px 10px;">3</td><td style="padding: 2px 10px;">2</td><td style="padding: 2px 10px; background-color: #e0e0ff;">1</td></tr></table> 3	4	3	2	1	X_2 <table border="1" style="display: inline-table; border-collapse: collapse;"><tr><td style="padding: 2px 10px;">4</td><td style="padding: 2px 10px;">3</td><td style="padding: 2px 10px; background-color: #e0e0ff;">2</td><td style="padding: 2px 10px; background-color: #e0e0ff;">1</td></tr></table> 6 2	4	3	2	1
4	3	2	1						
4	3	2	1						
= 3	= 8								
X_1 <table border="1" style="display: inline-table; border-collapse: collapse;"><tr><td style="padding: 2px 10px; background-color: #e0e0ff;">3</td><td style="padding: 2px 10px; background-color: #e0e0ff;">2</td><td style="padding: 2px 10px; background-color: #e0e0ff;">1</td></tr></table>	3	2	1	X_1 <table border="1" style="display: inline-table; border-collapse: collapse;"><tr><td style="padding: 2px 10px; background-color: #e0e0ff;">3</td><td style="padding: 2px 10px; background-color: #e0e0ff;">2</td><td style="padding: 2px 10px; background-color: #e0e0ff;">1</td></tr></table>	3	2	1		
3	2	1							
3	2	1							
X_2 <table border="1" style="display: inline-table; border-collapse: collapse;"><tr><td style="padding: 2px 10px;">4</td><td style="padding: 2px 10px; background-color: #e0e0ff;">3</td><td style="padding: 2px 10px; background-color: #e0e0ff;">2</td><td style="padding: 2px 10px; background-color: #e0e0ff;">1</td></tr></table> 9 4 1	4	3	2	1	X_2 <table border="1" style="display: inline-table; border-collapse: collapse;"><tr><td style="padding: 2px 10px; background-color: #e0e0ff;">4</td><td style="padding: 2px 10px; background-color: #e0e0ff;">3</td><td style="padding: 2px 10px; background-color: #e0e0ff;">2</td><td style="padding: 2px 10px; background-color: #e0e0ff;">1</td></tr></table> 12 6 3	4	3	2	1
4	3	2	1						
4	3	2	1						

$$[3 \ 2 \ 1] * [1 \ 2 \ 3 \ 4] = [3 \ 8 \ 14 \ 20]$$

1. The first step of the convolution is to flip and translate the vector $X_2[n]$ to the left;
2. The second step is to multiply the columns and sum the products;
3. The third step and translate the vector X_2 and repeat the second step;
4. Finally, the convolution will be given by the vector containing the previous sums-products.

2.3.1. Convolution 2D

The definition of 1D convolution is also applicable for 2D convolution save that one of the inputs is inverted twice.

This type of operation is widely used in digital image processing, where the 2D matrix representing the image will be convolved with a relatively more minor matrix called the 2D kernel.

$$\begin{array}{|c|c|c|} \hline 0 & 1 & 2 \\ \hline 3 & 4 & 5 \\ \hline 6 & 7 & 8 \\ \hline \end{array} * \begin{array}{|c|c|} \hline 0 & 1 \\ \hline 2 & 3 \\ \hline \end{array} = \begin{array}{|c|c|} \hline 19 & 25 \\ \hline 37 & 43 \\ \hline \end{array}$$

2.3.1.1. Padding

One problem that can arise when performing convolution is losing pixels at the perimeter of the image. Considering that small kernels are used, this can lead to the loss of a few pixels, but when many convolutional layers are applied, the loss of pixels is more significant. One solution to this problem is padding, i.e. adding extra pixels around the perimeter of the image to enlarge the image. Usually, the extra pixels have a value of zero. In some cases, it wants to have the same height and width in the output as in the input, and padding makes this more accessible because it has more control over the construction of the mesh.

An example of padding is shown below:

$$\begin{array}{|c|c|c|c|c|} \hline 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 1 & 2 & 0 \\ \hline 0 & 3 & 4 & 5 & 0 \\ \hline 0 & 6 & 7 & 8 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 \\ \hline \end{array} * \begin{array}{|c|c|} \hline 0 & 1 \\ \hline 2 & 3 \\ \hline \end{array} = \begin{array}{|c|c|c|c|} \hline 0 & 3 & 8 & 4 \\ \hline 9 & 19 & 25 & 10 \\ \hline 21 & 37 & 43 & 16 \\ \hline 6 & 7 & 8 & 0 \\ \hline \end{array}$$

2.3.1.2. Stride

When calculating the convolution, the starting point is the first matrix in the top left-hand corner and is translated, to the right and down, one step at a time. For example, in some cases, to improve computational efficiency, the translation is not done one step at a time but n steps. This jump is called Stride. Stride can reduce the resolution of the output, for example, by reducing the height and width of the output to only $\frac{1}{n}$ (with $n > 1$) of the height and width of the input.

$$\begin{array}{|c|c|c|c|c|} \hline 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 1 & 2 & 0 \\ \hline 0 & 3 & 4 & 5 & 0 \\ \hline 0 & 6 & 7 & 8 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 \\ \hline \end{array} * \begin{array}{|c|c|} \hline 0 & 1 \\ \hline 2 & 3 \\ \hline \end{array} = \begin{array}{|c|c|} \hline 0 & 8 \\ \hline 6 & 8 \\ \hline \end{array}$$

Stride along with padding can be used to adjust the dimension of the data effectively.

2.4 Tools

2.4.1. Xilinx Vivado Design Suite

Vivado Design Suite is a software suite produced by Xilinx for HDL design synthesis and analysis, includes electronic system level (ESL) design tools for synthesising and verifying C-based algorithmic IP, standards-based packaging of both algorithmic IP and RTL for re-use, standards-based IP stitching and system integration of all types of system building blocks, and block and system verification.

Vivado IDE offer [14]:

-
- Register transfer level (RTL) design in VHDL, Verilog, and SystemVerilog
 - Intellectual property (IP) integration for cores
 - Behavioral, functional, and timing simulation with Vivado simulator
 - Vivado synthesis
 - Vivado implementation for place and route
 - Vivado serial I/O and logic analyzer for debugging
 - Vivado power analysis
 - SDC-based Xilinx design constraints (XDC) for timing constraints entry
 - Static timing analysis
 - High-level floorplanning
 - Detailed placement and routing modification
 - Bitstream generation

2.4.1.1. Vivado High-Level Synthesis

Xilinx Vivado HLS belongs to the tools available in the Xilinx Vivado Design Suite. The tool synthesizes functions written in C/C++ into an IP core that can be integrated into a hardware system [15].

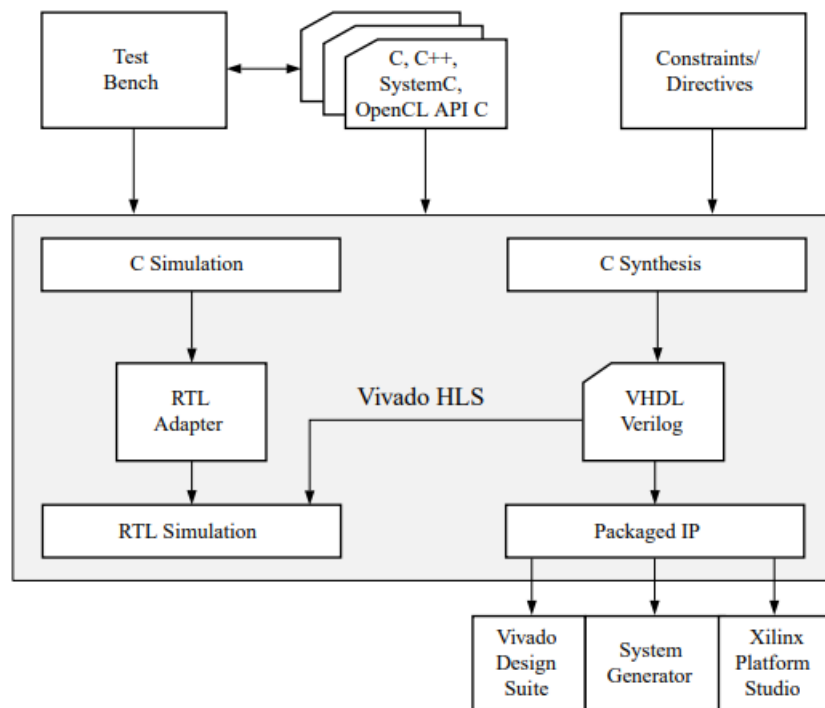


Figure 2.3: Vivado HLS Design Flow

The Vivado HLS Design flow is illustrated in the figure 2.3.

When creating a new project, the tool allows 4 types of files to be added to the project:

- C function written in **C**, **C++**, **SystemC**, or an **OpenCL API C** kernel.

These files contain the functions that are to be synthesised. These are the files that contain the code written at a high level that will later be converted.

- **Constraints:** Contain information about the clocks, such as period and uncertainty. They also have information about the target.

If not specified, the clock has an uncertainty of 12.5%.

- **Directives:** Directives are optional and direct the synthesis process to implement a specific behaviour or optimization.

- **C test bench:** To verify the output of RTL, Vivado uses the C/RTL Cosimulation. This step is performed before synthesis.

The **C Simulation** is made for the validation of the C algorithm that is an important part of the High-Level Synthesis (HLS) process. To ensure that the C algorithm performs the correct operation, a C test bench is used to confirm that the results are correct.

The Vivado simulator is a Hardware Description Language (HDL) event-driven simulator that supports functional and timing simulations for VHDL, Verilog, SystemVerilog (SV), and mixed VHDL/Verilog or VHDL/SV designs.

The Vivado simulator saves the simulation results of the objects (VHDL signals, or Verilog reg or wire) being traced to the Waveform Database (WDB) file.

Synthesis is the process of transforming an RTL-specified design into a gate-level representation. Vivado synthesis is timing-driven and optimised for memory usage and performance.

Interface synthesis is the process of adding RTL ports to the C design. In addition to adding the physical ports to the RTL design, interface synthesis includes an associated I/O protocol, allowing the data transfer through the port to be synchronised automatically and optimally with the internal logic.

When Synthesis completes, Vivado HLS generates a Synthesis Summary report for the top-level function. In this file, it is possible to find information on the resources used, times, latency.

A crucial part of creating high-quality RTL designs using High-Level Synthesis has the ability to apply optimisations to the C code. High-Level Synthesis always tries to minimise the latency of loops and functions. To achieve this, within the loops and functions, it tries to execute as many operations as possible in parallel. At the level of functions, High-Level Synthesis always tries to execute functions in parallel. However, after analysis, it is possible to change C/C++ code to optimise the performance of the function. Optimisations can be made by adding optimisation directives directly into the source code as compiler pragmas, using various HLS PRAGMAS.

This is often an iterative process, requiring multiple steps and multiple optimisations to achieve the desired results. Solutions offer a convenient way to configure the tool, add directives to the function to improve the results, and preserve those results to compare with other solutions.

The Vivado synthesis also generates a RTL implementation file in HDL (Hardware Description Language) format. The RTL is available in two industry-standard formats: VHDL and Verilog.

The High-Level Synthesis tool automates the process of **C/RTL CoSimulation**. To perform RTL verification, the tool uses both the RTL output from High-Level Synthesis (Verilog or VHDL) and the C test bench.

As described before, the test bench verifies output from the top-level function for Synthesis and returns zero to the primary () function of the test bench if the output is correct.

The RTL simulation in Vivado uses the same inputs as in the C simulation and synthesis to check the correctness of the results. If the simulation ends with a non-zero value, then the results are not correct.

The C/RTL verification process consists of three phases:

- The C simulation is executed, and the inputs to the top-level function, or the Design-Under-Test (DUT), are saved as “input vectors.”
- The “input vectors” are used in an RTL simulation using the RTL created by Vivado HLS in Vivado simulator. The outputs from the RTL, or simulation results, are saved as “output vectors.”
- The “output vectors” from the RTL simulation is returned to the C test bench’s primary () function to verify the results are correct. The C test bench verifies the results, in some cases, by comparing them to known good results.

After the end of the co-simulation, the report showing the latency values is displayed.

The final step in the Vivado HLS flow is to **export the RTLL** project in so that it can be imported on other platforms. The RTL component can be packaged in various formats:

- Vivado IP: The IP is exported as a ZIP file added to the Vivado IP catalogue.
- Vivado Kernel: The XO file output can be used for linking by the Vivado compiler in the application acceleration development flow.
- Synthesised Checkpoint: This option creates Vivado checkpoint files which can be added directly into a design in the Vivado Design Suite.
- Vivado IP for System Generator: This option creates IP for use with the Vivado edition of System Generator for DSP.

2.4.2. proFPGA

The proFPGA product family meets the highest requirements in the areas of FPGA boards and FPGA-based prototyping. With proFPGA products, it is possible to get maximum flexibility in building any hardware configuration, reconfigurable and adaptable to multiple applications. In this work was used the Kintex UltraScale FPGA (section 2.4.2.4).

PRO DESIGNs FPGA prototyping systems offer scalable, high-performance Single and Multi-FPGA solutions, which can be easily adapted and expanded by additional proFPGA modules with the latest FPGAs and extension boards equipped with interconnects, interfaces, or memories. The ProFPGA prototyping system is a set of modular blocks, which allows users of this system to customise the various solutions to best suit the project specifications [16].

The hardware and software layers of the proFPGA prototyping system are explained below.

2.4.2.1. Hardware Level

The hardware level of the proFPGA prototyping system is divided into several modular units, the main ones being (figure 2.4):

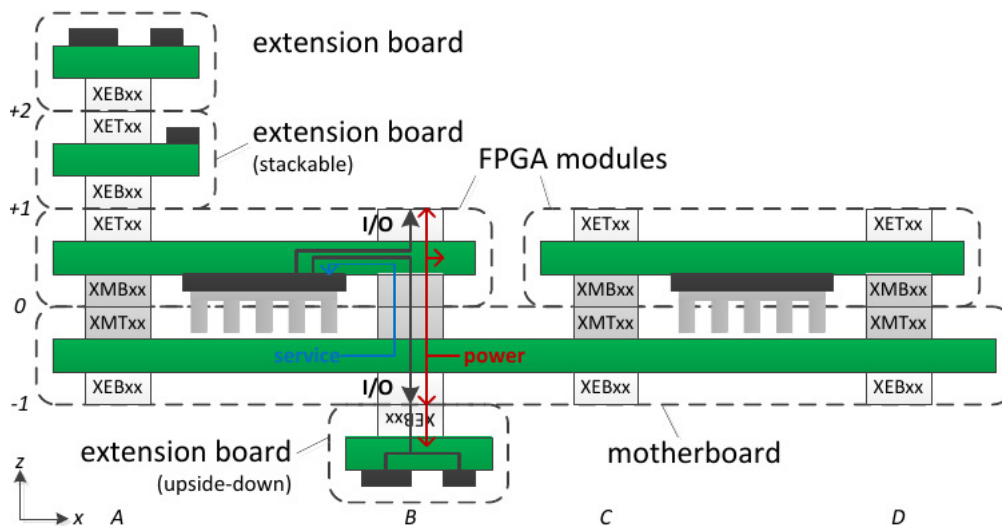


Figure 2.4: Modular hardware

Motherboards Motherboards are the heart of FPGA; they provide clock infrastructure, power supply, MMI-64 communication for multiple FPGA modules, I2C-based system management, and mechanical fixation. Motherboards have two types of connectors, one for the extension boards (carrying user I/O, power) on the bottom side and the other for the FPGA modules (carrying user I/O, power, service) on the top side. It also provides a transparent connection between the extension boards on the bottom side and the FPGA module on the top side; this is due to the direct link of the user I/O pins of the connectors on the top and bottom sides.

FPGA Modules The FPGA modules offer eight connectors connected to the extension sites and have 4 FPGA module connectors (user I/O, power, service) and 4 extension board connectors (user I/O, power). Each module can access the MMI-64 communication through the motherboard. Moreover, since the motherboard provides a transparent connection between the extension board and the FPGA module, each FPGA module can access up to 8 extension boards.

Extension Boards Extension boards provide hardware functions within FPGA modules, such as debug access, SDRAM memory, and user PCIe connection. In order to have exclusive access to the extension board, an extension board occupies one or more of the connectors of an FPGA module, and other extension boards can be added by mapping the non-used I/O pins of the FPGA module to a connector on the top side.

Interconnects Interconnects are used to connect the I/O pins of several FPGA modules and are unique extension boards. Interconnects are to be found in cables and boards and connect two or more extension sites. Finally, they can be point-to-point (e.g., all 2-way interconnection boards and cables) or broadcast (e.g., the 4-way interconnection board).

System Extension Boards System functionality can be expanded by special hardware, like the motherboard's PCIe adapter board or a motherboard-to-motherboard connector cable. This hardware uses designated connectors on the motherboard.

2.4.2.2. Software Level

The Software layer of proFPGA provides various tools:

- profpga_run: a command-line tool to turn on turn off and configure the system
- profpga_builder: GUI to create configuration setups of the board and to execute run-time accesses the system
- profpga_brdgen: a command-line tool to generate various VHDL/Verilog top-level files, constraint files, board description files, and a project-based self-test to test FPGA interconnects at speed
- profpga_selftest2: a command-line tool that performs the project-based self-test on the hardware
- profpga_freq: a command-line tool to determine proFPGA clock settings to generate a specific clock frequency or achieve the desired data rate with mux/demux proFPGA modules.

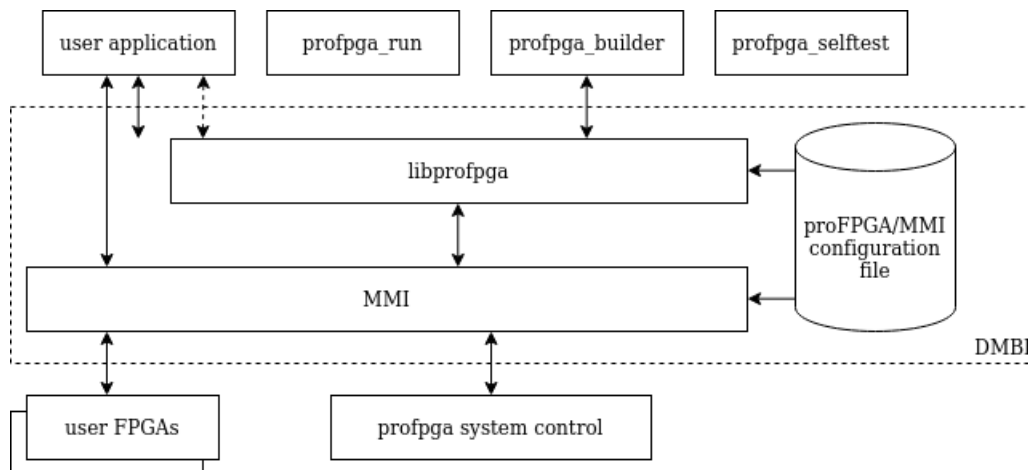


Figure 2.5: ProFPGA software and libraries

There are two parts to communicate with the system:

- MMI is a transport mechanism between the proFPGA system and a Host-PC. It is used for user project communication and system control and consists of extracting the different communication modes such as PCIe, Ethernet, and USB.
- libprofpga provides the tools to configure and control the system and to observe status information. libprofpga uses MMI to communicate with the hardware.

Both parts together form the proFPGA communication package, which is called DMBI.

2.4.2.3. MMI64

Communication between the proFPGA system and a Host-PC is achieved through a series of components [17].

Host Application The host application controls the communication. It will send MMI64 messages downstream and request MMI64 messages upstream.

Software Libraries Software libraries are pre-compiled libraries from Pro Design that implement API functions. They are used to communicate with the proFPGA system.

proFPGA Motherboard In order to communicate with the user application, the proFPGA motherboard provides four paths:

- PCIe
- Ethernet
- USB cable
- previous motherboard (for systems with multiple motherboards)

proFPGA User FPGA Control Unit Pro Design provides the proFPGA control unit, "profpga_ctrl", to have MMI64 communication.

The control unit has the following functions:

- decode the communication pins into MMI64 signals

- translate the clock/sync from the LVDS pins to the internal clock and reset
- configuration access for Advanced Clock Management (ACM) on clock/sync #1 to 7
- MMI64 communication interface for the user project inside the FPGA.

The control unit also provides a routing structure for MMI64 messages:

- MMI64 communication to user project
- Service

MMI64 PCIe PHY Pro Design provides another module, "mmi64_p_pcie", which allows, through an additional extension and by instantiating MMI64 PCIe PHY, not to use the motherboard. Usually, it is chosen to use this method when more data throughput is necessary than the motherboard can provide.

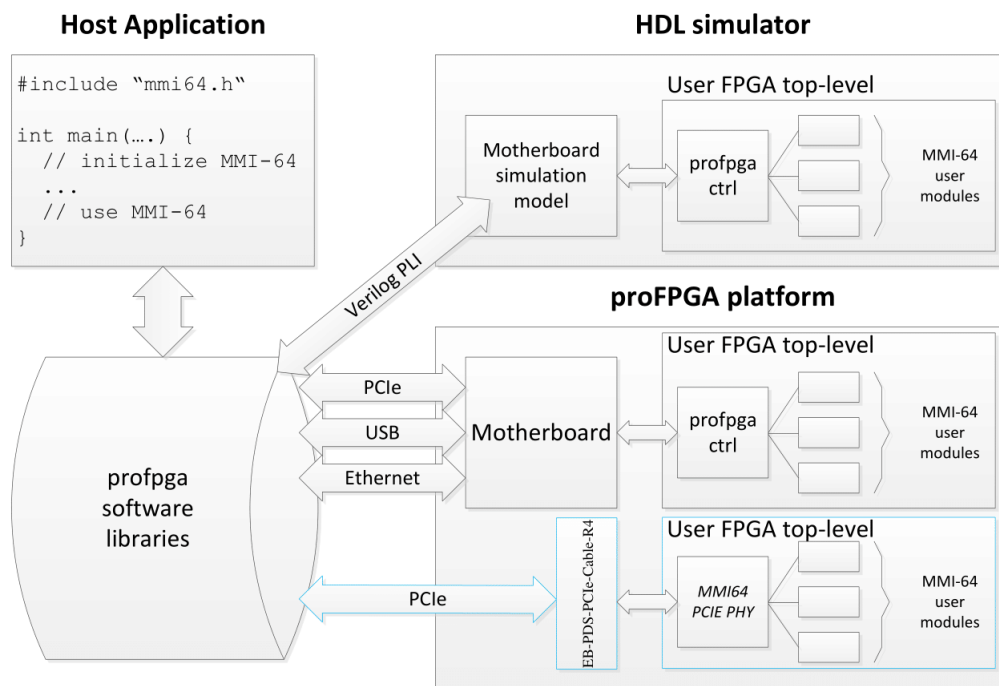


Figure 2.6: MMI64 communication channels

2.4.2.4. FPGA Kintex UltraScale

The XCKU115 proFPGA FPGA module is the logical heart of the scalable and modular multi FPGA proFPGA solution. This module is used by those who need a scalable

and more flexible high-performance ASIC Prototyping solution for initial software development and real-time system testing. The XCKU115 proFPGA FPGA module, which works only in combination with a proFPGA one, duo, or quad motherboard. Offers its latest Kintex UltraScale FPGA technology a maximum capacity of up to 7.9 M ASIC ports and 5520 DSP slices in a single FPGA. It is designed to achieve maximum performance in combination with its high-speed connectors. The module offers six extension sites up to 585 user I/Os for daughter cards (e.g., memory cards, interface cards), interconnect cables, or customer-specific application cards. In addition to the standard I/Os, the module also provides 56 high-speed serial transceivers (56 x GTH) operating at up to 16.375 Gbps for high-speed interfaces such as PCIe Gen4. All six extension sites offer individually adjustable and stepped voltage regions from 1.2V up to 1.8V.

Capacity	Up to 7.9 M ASIC gates
FPGA-internal memory	75 Mbit
DSP Slices	5520
Extension sites	Up to 6 Extension sites with high speed connectors
I/O resources	Overall up to 585 signals for I/O and inter FPGA connection 585 free I/Os per FPGA Module (Virtex® XCVU7P FPGA) 3x153 I/Os and 1x76 I/Os to top side connectors 2x25 I/Os to bottom side connectors Single-ended or differential
High Speed I/O transceivers	56 high speed transceivers (GTH) running up to 12.5/16.375 (speedgrade 1/2/3) Gbps
FPGAs interconnections	Flexible via high-speed interconnection boards or cables
Voltage regions	6 individually adjustable voltage regions per FPGA Module Stepless from 1.0V up to 1.8V on 6 extension sites Automated detection of daughter card and adjustment of right voltage
Configuration	With proFPGA uno, duo or quad Motherboard via Ethernet, USB, PCIe

Table 2.1: XCKU115 proFPGA Characteristics

2.4.3. HLS application

This section explains and demonstrates all steps in transforming C/C++ code to an RTL implementation using High-Level Synthesis. The Section shows how to create an initial RTL implementation and then transform it using optimisation directives without changing the C code.

In the Xilinx application acceleration flow, the Vivado HLS tool automates most of the code changes required to implement and optimise C/C++ code in programmable logic and

achieve low latency and high throughput. Inference of the pragmas needed to produce the correct interface to function arguments and pipeline loops and functions within the code is the foundation of Vivado HLS in the application acceleration flow. Vivado HLS also supports code customisation to implement different interface standards or specific optimisations to achieve design goals.

The following is the vivado HLS design flow:

1. Compile, simulate and debug the C/C++ algorithm.
2. Synthesise the C algorithm and view reports to analyse and optimise the design.
3. Verify the RTL implementation using RTL co-simulation.
4. Export the RTL implementation to an RTL IP.

2.4.3.1. Compile, simulate, and debug

The following listing 2.2 is represented as a simple example written in C/C++ and optimised using the HLS INTERFACE pragma.

```
1 void sum_value(volatile int *a){
2 #pragma HLS INTERFACE s_axilite port=return bundle=control
3 #pragma HLS INTERFACE m_axi depth=50 port=a offset=slave
4 #pragma HLS INTERFACE s_axilite port=a bundle=control
5
6 int i;
7 int buff[50];
8
9 memcpy(buff,(const int *)a,50*sizeof(int));
10
11 for(i=0; i < 50; i++){
12     buff[i] = buff[i] + 100;
13 }
14
15 memcpy((int *)a ,buff,50*sizeof(int));
16 }
```

Listing 2.2: HLS example

This example takes a pointer (**a**) as input; the pointer points to a vector with a maximum size of 50 integers. The vector is copied via the memcpy function into a local vector (**buff[i]**). The memcpy requires a local buffer to store the results of the memory transaction

and creates burst access to memory. Sequentially is added a constant to all the values in the vector, and finally, the modified buffer is reassigned to the pointer used as input. In this case, multiple calls of memcopy cannot be pipelined and will be scheduled.

In order to specify the I/O ports as master/slave axi buses, the HLS INTERFACE pragma was used.

HLS INTERFACE pragma specifies how RTL ports are created from the function definition during interface synthesis. The ports in the RTL implementation are derived from the following:

- Block-level I/O protocols: Provide signals to control when the function starts operation and indicate when function operation ends, is idle and ready for new inputs.
- Function arguments: Each function argument can be specified to have its port-level (I/O) interface protocol, such as a valid handshake, or acknowledge handshake. Port-level interface protocols are created for each argument in the top-level function, and the function return if the function returns a value. The default I/O protocol created depends on the type of C argument. After the block-level protocol has been used to initiate block operation, port-level I/O protocols will be used to route data input and output from the block.
- Global variables accessed by the top-level function, and defined outside its scope: If a global variable is accessed, all read and write operations are local to the function, the resource is created in the RTL design. The RTL requires no I/O port. If the global variable is supposed to be an external origin or destination, simply define the interface in a way analogous to the arguments of standard functions.

Syntax

```
#pragma HLS interface <mode> port=<name> bundle=<string> depth=<int>  
                        offset=<string>
```

Where:

- `<mode>`: Specifies the interface protocol mode for function arguments, global variables used by the function, or the block-level control protocols. The mode that can be specified is:
 - `s_axilite`: Implements all ports as an AXI-Lite interface. the HLS facility generates a set of C driver files as part of the RTL export process.
 - `m_axi`: Ports are implemented as AXI interfaces. To specify 32-bit (default) or 64-bit address ports and control any address offsets, use the `config_interface` command.
- `port=<name>`: Specifies the name of the function argument, function return, or global variable to which the `INTERFACE` pragma applies.
- `bundle=<string>`: Groups function arguments into AXI interface ports. The HLS tool groups all function arguments specified as an AXI-Lite (`s_axilite`) interface into a single AXI-Lite port. Similarly, all function arguments specified as an AXI (`m_axi`) interface are grouped into a single AXI port. This option explicitly groups all interface ports with the same `bundle=<string>` into the same AXI interface port and names the RTL port's value specified by `<string>`.
- `depth=<int>`: This setting specifies the highest number of samples that the test bench must elaborate on. This set-up specifies the maximum dimension of the FIFO required in the verification adapter the HLS tool creates for RTL co-simulation.
- `offset=<string>`: Controls the address offset in AXI-Lite (`s_axilite`) and AXI (`m_axi`) interfaces.
 - For the `s_axilite` interface, `<string>` provides the registration map address.
 - For the `m_axi` interface, `<string>` one of the following values is specified:
 - * `direct`: Generate a scalar input offset port.
 - * `slave`: Generate an offset port and automatically map it to an AXI-Lite slave interface.
 - * `off`: Do not generate an offset port.

C Simulation The first step in an HLS project is to confirm that the C code is correct. This process is called C Validation or C Simulation. This can be done with: **Project > Run C Simulation**, to compile and execute the C design.

The test bench 2.3 includes a main() top-level function. The main instantiates two vectors, one of which is used as input to the function that needs to be synthesised and another to execute the software version of the hardware function. At the end of the program, the two vectors are compared to validate the function.

```
1 void sum_value(volatile int *a);
2
3 int main()
4 {
5     int i;
6     int A[50];
7     int B[50];
8
9     printf("HLS AXI-Stream no side-channel data example\n");
10    //Put data into A
11    for(i=0; i < 50; i++){
12        A[i] = i;
13    }
14
15    //Call the hardware function
16    sum_value(A);
17
18    //Run a software version of the hardware function to validate results
19    for(i=0; i < 50; i++){
20        B[i] = i + 100;
21    }
22
23    //Compare results
24    for(i=0; i < 50; i++){
25        if(B[i] != A[i]){
26            printf("i = %d A = %d B= %d\n",i,A[i],B[i]);
27            printf("ERROR HW and SW results mismatch\n");
28            return 1;
29        }
30    }
31    printf("Success HW and SW results match\n");
32    return 0;
33 }
```

Listing 2.3: Test bench

2.4.3.2. Synthesize

The second step consists of synthesising the C design into an RTL design and reviewing the synthesis report to see the program's performance. **Solution > Run C Synthesis > Active Solution.** When Synthesis completes, the Xilinx tool generates a Synthesis Summary report (figure 2.7) for the top-level function.

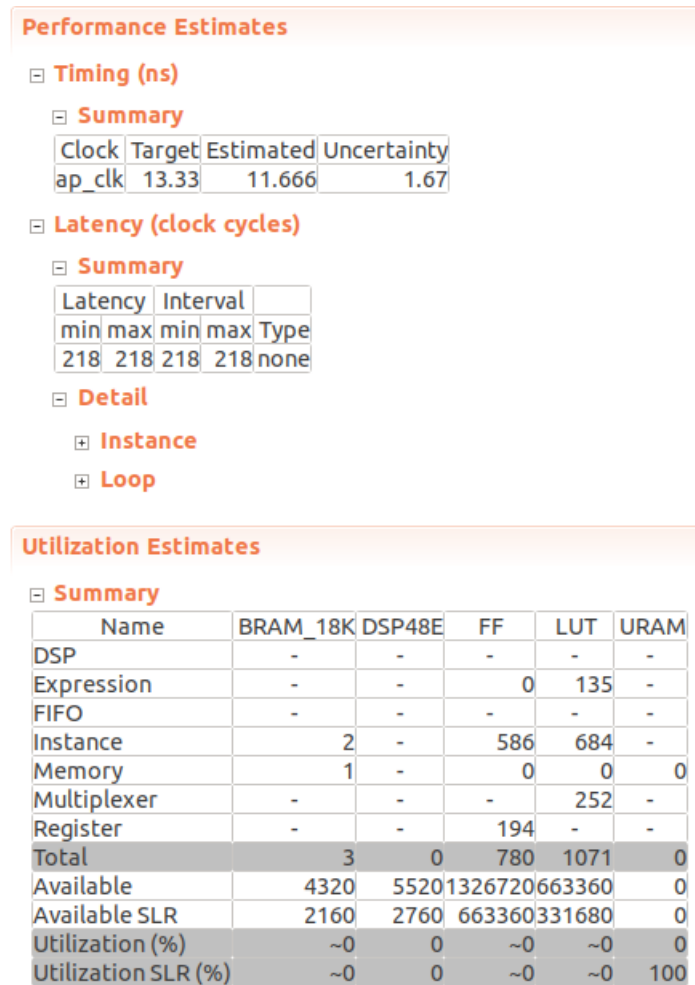


Figure 2.7: Synthesis Summary Report

In the Performance Estimates pane, shown in Figure 2.7, it is possible to see the clock period is set to 13.33 ns. Xilinx tool targets a clock period of Clock Target minus Clock Uncertainty ($13.33 - 1.67 = 11.66$ ns in this case). The clock uncertainty ensures there is some timing margin available for them (at this stage) unknown net delays due to place and routing. The estimated clock period (worst-case delay) is 11.666 ns, meeting the 11.66 ns timing requirement.

Another parameter seen in the summary is that the design has a latency of 128 cycles; it takes 128 clocks to output the results, and the next set of inputs is read after 128 clocks. The design is not pipelined. The subsequent execution of this function (or next transaction) can only start when the current transaction completes.

The Utilisation Estimates shows that the design uses 3 BRAM_18K, 780 flip-flops and 1071 LUTs. At this stage, the device resource numbers are estimates because RTL synthesis might perform additional optimisations, and these figures might change after RTL synthesis.

2.4.3.3. Verify the RTL implementation

High-Level Synthesis can re-use the C test bench to verify the RTL using simulation.

Solution > Run C/RTL CoSimulation.

If the C test bench returns a non-zero value, the Xilinx tool reports that the simulation failed. After simulation, the Cosimulation Report shows the pass or fail status and the measured statistics on latency (figure 2.8).

Cosimulation Report for 'sum_value'

Result							
		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	260	260	260	NA	NA	NA

Figure 2.8: Co-simulation report

2.4.3.4. Export the RTL implementation

The final step in the High-Level Synthesis flow is to package the design as an IP block with other tools like the Vivado Design Suite. **Solution > Export RTL.**

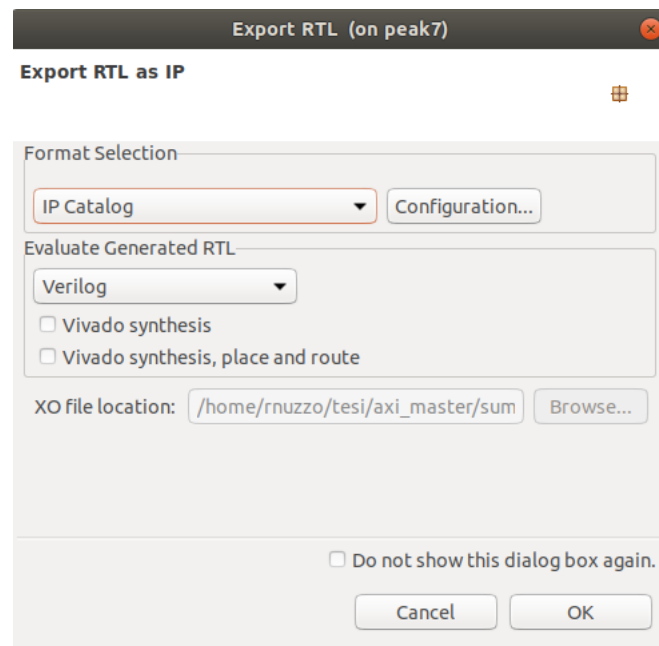


Figure 2.9: Export RTL panel

The RTL design can be packaged into different output formats. In this case, as shown in Figure 2.9, The IP is exported as a ZIP file that can be added to the Vivado IP catalogue. In addition to selecting the IP format on the Export RTL dialogue box, it is also possible to change the IP configuration. One important thing in the configuration options is to choose the IP name shown when adding the IP to the Vivado catalogue.

CHAPTER 3

Single-FPGA

The goal of this thesis work is to verify the possibility of using multiFPGA prototypes as a computing platform, discussed in the Section 2.1.3, where the FPGA kernels are coded in HLS (Section 2.2). In order to simplify the work, the project has been divided into two parts, the first part where only one FPGA is used and a second part, which will be addressed in the next chapter, where the project will be expanded to more FPGAs.

In this chapter, the design (Section 3.1) of the project using a single FPGA is explained, going into detail about the components of which it is composed. Once an overview of the design is done, Section 3.2 explains how the design was tested and what techniques were used.

3.1 Design

Figure 3.1 shows the design developed for the single FPGA case. The most important components of the design are as follows:

- mmi64_host_interface module;
- convolution IP core;
- SDRAM;
- System ILA core.

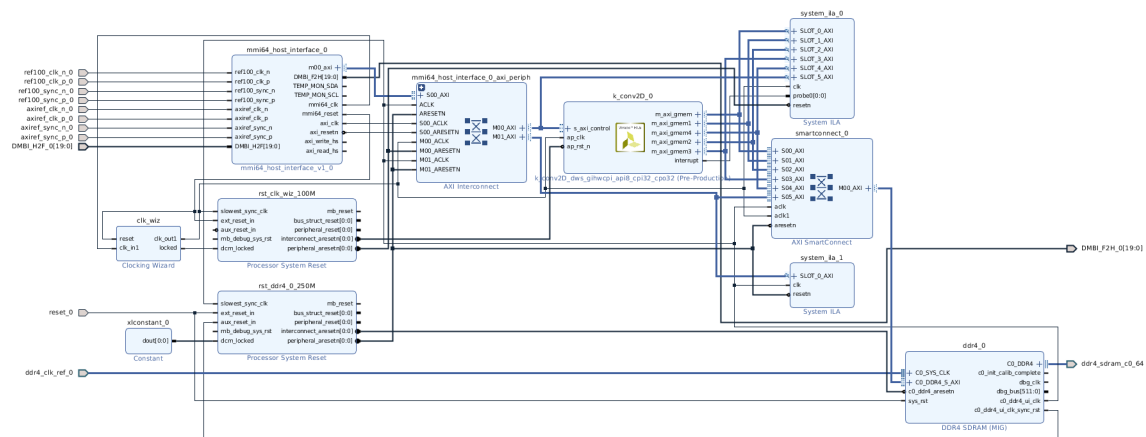


Figure 3.1: Vivado design

Using the different interfaces illustrated in 2.6, the host can communicate with the System through DMBI (Device Message Box Interface). In this case, the interface used is the proFPGA PCIe. So the user, thanks to this interface, communicates the desired inputs to the MMI64 host interface module, designed precisely with the function of acting as a bridge between the host and the FPGA. In essence, the host, through the module MMI64 host interface, can pass input values to the IP core of the convolution to perform the convolution and can communicate directly with the memory (DDR4) to make the appropriate settings, such as enable/disable interrupts or read the data saved by the IP core. In the design, there is also another module, system ILA, used to debug the System. In the following sections, the design will be explained in more detail.

3.1.1. MMI64 host interface

The `mmi64_host_interface` module is a module developed to implement the communication between a host and an FPGA.

The module consists of three main blocks:

- `Profpga_ctrl`;
- `MMI64_axi_master`;
- `Profpga_clocksync`.

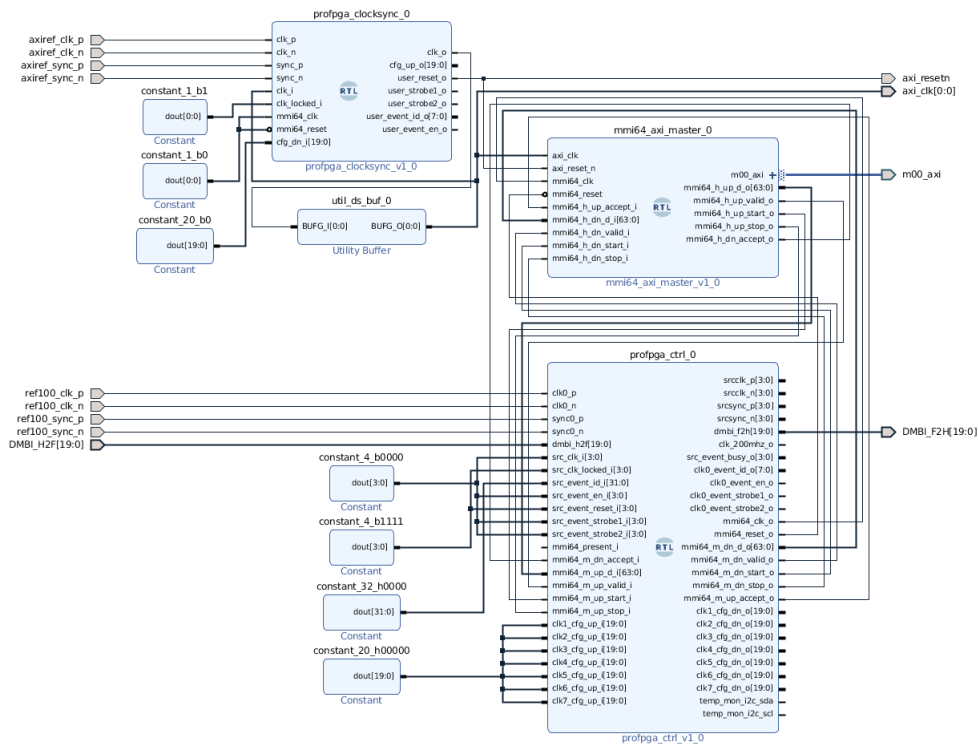


Figure 3.2: Vivado mmi64 host interface module design

The `profpga_ctrl` module is instantiated, which includes the MMI64 PHY. This module is directly driven by CLK/SYNC[0] and connects to the DMBI pins. It creates the MMI64 interface with the downstream and upstream data interface and the mmi64 clock and reset. This MMI64 interface is directly connected to the `mmi64_axi_master` module. A `profpga_clocksync` module creates the AXI clock and resets signals. The AXI clock is running on 125MHz, and the MMI64 clock is running on 100MHz.

3.1.1.1. Profpga_ctrl

The proFPGA control unit is the core interface of user FPGA designs to the infrastructure provided by the proFPGA motherboard and host software.

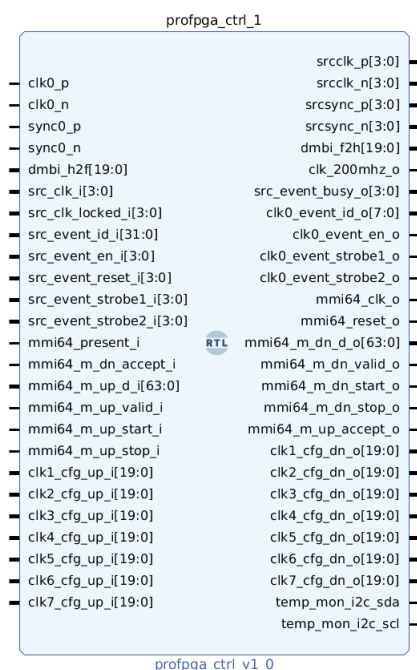


Figure 3.3: Profpga_ctrl module

This module contains the DMBI signals that allow the host to be able to communicate with the System. The proFPGA PCIe DMBI Interface offers a high rate of data exchange. The user can benefit from remote system configuration from using this high-speed interface more efficiently for debugging purposes such as streaming data or sending and receiving test patterns, and so on.

The MMI64 interface signals implement the MMI64 protocol. This interface allows communication with another module of the System, MMI64_axi_master, which will be discussed in more detail in the section 3.1.1.2.

When the MMI64 is used, the signals must be connected to the host-side interface of the connected module, and mmi64_present_i must be driven to '1'. The MMI64 clock is a free-running 100 MHz clock. The MMI64 reset is lowered during the final system initialization stage. The user may provide clocks to the motherboard using the source clock interface. The clock configuration interface must be connected to the corresponding profpga_clksync.

setup Various parameters of the component can be set, including:

- **Device:** in this module the type of FPGA used is chosen, in this case a Kintex UltraScale Section 2.4.2.4 - "XVUS".
- **ENABLE_DEBUG:** set to "FALSE" when not instructed by ProDesign
- **PIN_TRAINING_SPEED:** choose real pin training for synthesis or fast pin training for simulation - "auto" (synthesis tool must support "synthesis translate_off")
- **ENABLE_DISCO:** Enable MMI64 DisCo instance inside - "FALSE"
- **USE_CLK_INPUT_BUFG:** Use BUFG before PLL for UltraScale implementation - "0"

3.1.1.2. MMI64_axi_master

The MMI64 AXI Master module is intended to perform AXI4 bus transactions inside the user FPGA design. A host application controls it.

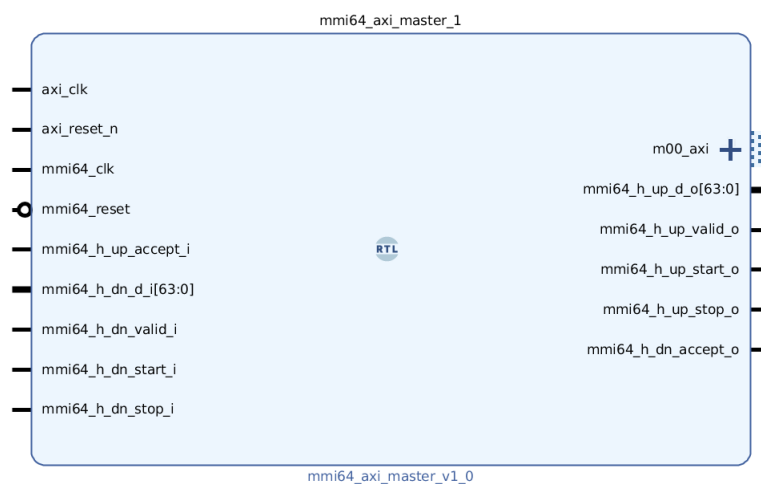


Figure 3.4: MMI64_axi_master module

The MMI64_axi_master module, on the one hand, is connected to the profpga_ctrl module via the MMI64 interface. On the other hand, the AXI master signals allow the communication with the memory and the convolution IP core. This module has two clock types, one at 100 MHz like the profpga_ctrl module and the other at 125 MHz, to synchronize the AXI communication.

setup An important parameter of this module is "MODULE_ID". This unique ID will be used to distinguish the various mmi64 AXI masters when multiple FPGAs are present.

- **MODULE_ID**: unique id of the module instance
- **AXI_ID_WIDTH**: ID width
- **AXI_ADDR_WIDTH**: AXI address width
- **AXI_USER_WIDTH**: AXI User signal width
- **AXI_DATA_WIDTH**: AXI Data signal width

3.1.1.3. Profpga_clocksync

A profpga_clocksync module creates the AXI clock and resets signals. The AXI clock is running on 125MHz, and the MMI64 clock is on 100MHz. The clock generated at 125 MHz is also output to the MMI64_host_interface module so that the slave is connected to the AXI master of the MMI64_axi_master (Section 3.1.1.2) module can run at the same speed.

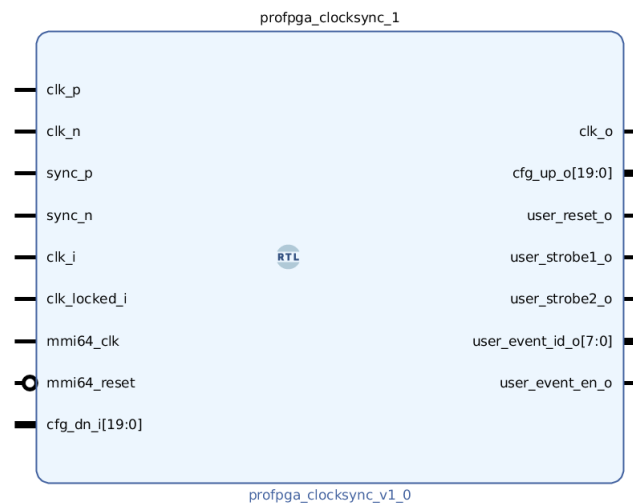


Figure 3.5: Profpga_clocksync module

To reduce synthesis timing issues, the clock is not used directly for the processing of the associated SYNC signal. Instead, the user design must provide clock feedback:

1. If the design uses a PLL or MMCM, it is recommended to provide a 1:1 output with no phase shift to clk_i and the PLL "locked" signal as clk_locked_i.

2. If the clock is used directly, the output of the clock buffer must be provided as feedback. In this case, the `clk_locked_i` signal should be tied to '1'.

setup

- `CLK_CORE_COMPENSATION`
 - "DELAYED": `clk_core` is delayed against clock from motherboard to be used for all XILINX 7-Series and ZYNQ-7000 FPGAs,
 - "ZHOLD": `clk_core` is compensated to negative phase offset,
 - "DELAYED_XVUS": `clk_core` is delayed against clock from motherboard to be used for all XILINX Ultrascale FPGAs.

3.1.2. Convolution IP core

In this work, HLS code was used to perform the convolution, developed by the Group of Parallel Architectures (GAP) of UPV.



Figure 3.6: Convolution IP core

The IP core is connected to the `MMI64_host_interface` via AXI slave signals and to the memory via AXI master signals. The changes made to the HLS code to fit the System involve just the addition of the HLS INTERFACE pragmas `s_axilite` and `m_axi`, so that upon generation of the IP core, the core would create the AXI interface.

```

1 #pragma HLS INTERFACE s_axilite bundle=control port=ptr_data
2 #pragma HLS INTERFACE s_axilite bundle=control port=ptr_out
3 #pragma HLS INTERFACE s_axilite bundle=control port=ptr_bias
4 #pragma HLS INTERFACE s_axilite bundle=control port=ptr_kernel
5 #pragma HLS INTERFACE s_axilite bundle=control port=ptr_dw_kernel

```

```

6 #pragma HLS INTERFACE s_axilite bundle=control port=ptr_pw_kernel
7 #pragma HLS INTERFACE s_axilite bundle=control port=pos_shift
8 #pragma HLS INTERFACE s_axilite bundle=control port=dir_shift
9 #pragma HLS INTERFACE s_axilite bundle=control port=max_clip
10 #pragma HLS INTERFACE s_axilite bundle=control port=min_clip
11 #pragma HLS INTERFACE s_axilite bundle=control port=enable_shift
12 #pragma HLS INTERFACE s_axilite bundle=control port=enable_clipping
13 #pragma HLS INTERFACE s_axilite bundle=control port=enable_avgpooling
14 #pragma HLS INTERFACE s_axilite bundle=control port=enable_maxpooling
15 #pragma HLS INTERFACE s_axilite bundle=control port=enable_lower_padding
16 #pragma HLS INTERFACE s_axilite bundle=control port=enable_upper_padding
17 #pragma HLS INTERFACE s_axilite bundle=control port=global_offset
18 #pragma HLS INTERFACE s_axilite bundle=control port=enable_relu
19 #pragma HLS INTERFACE s_axilite bundle=control port=o_iter_last
20 #pragma HLS INTERFACE s_axilite bundle=control port=o_iter_first
21 #pragma HLS INTERFACE s_axilite bundle=control port=I_ITER
22 #pragma HLS INTERFACE s_axilite bundle=control port=I
23 #pragma HLS INTERFACE s_axilite bundle=control port=O
24 #pragma HLS INTERFACE s_axilite bundle=control port=I
25 #pragma HLS INTERFACE s_axilite bundle=control port=W
26 #pragma HLS INTERFACE s_axilite bundle=control port=H
27 #pragma HLS INTERFACE s_axilite bundle=control port=rows
28 #pragma HLS INTERFACE s_axilite bundle=control port=return
29
30 #if defined(DIRECT_CONV) || defined(WINOGRAD_CONV)
31   DO_PRAGMA(HLS INTERFACE m_axi port=ptr_kernel      depth=KERNEL_PORT_DEPTH  offset=
32     slave bundle=gmem1)
33 #endif
34 #ifndef DWS_CONV
35   DO_PRAGMA(HLS INTERFACE m_axi port=ptr_dw_kernel depth=DW_KERNEL_PORT_DEPTH
36     num_read_outstanding=CPI offset=slave bundle=gmem1)
37   DO_PRAGMA(HLS INTERFACE m_axi port=ptr_pw_kernel depth=PW_KERNEL_PORT_DEPTH
38     num_read_outstanding=CPI offset=slave bundle=gmem4)
39 #endif
40 DO_PRAGMA(HLS INTERFACE m_axi port=ptr_data      depth=DATA_IN_PORT_DEPTH
41   num_read_outstanding=CPI offset=slave bundle=gmem)
42 DO_PRAGMA(HLS INTERFACE m_axi port=ptr_bias      depth=BIAS_PORT_DEPTH
43   offset=slave bundle=gmem2)
44 DO_PRAGMA(HLS INTERFACE m_axi port=ptr_out      depth=DATA_OUT_PORT_DEPTH
45   num_write_outstanding=CPO offset=slave bundle=gmem3)

```

Listing 3.1: Convolution pragmas to create AXI interface

In the listing 3.1 the reader can see the same type of pragma but with two different modes. The pragma is of type INTERFACE then to create the input and output ports of the IP core. As for the mode, all data that must be input to the IP core are mapped as AXI4-Lite

interface, then with a mode "s_axilite". All data have the same "bundle", so they will be grouped in the same port, connected to the MMI64_host_interface module so that the host through the AXI bus can pass the parameters to the IP core.

The other mode used concerns the ports that are output to the IP core, mapped as AXI4 interface, therefore with a "m_axi" mode. The output of the IP core is connected to memory.

After modifying the code, the IP core was generated following the flow described in Section 2.4.3.

The clock feeding the IP core is a clock generated through a clock wizard at a frequency of 100MHz.

3.1.3. Memory

Figure 3.7 shows the IP Core controller of the DDR4 memory attached to the single FPGA prototype. The memory has a capacity to store 2GB of data and it is accessible from both the host and the FPGA convolution kernel. For that purpose an AXI interconnect is used. Then, the AXI master signals of the convolution kernel and the AXI master signals coming from the host are connected to a X:1 AXI interconnect.

Next, the most important configuration options of the DDR4 IP core are described.

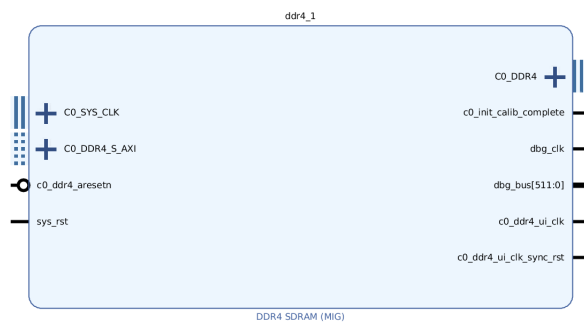


Figure 3.7: DDR4

setup

- Basic
 - Memory Device interface Speed = 1000
 - Reference input Clock Speed = 8000

- Memory Part = EDY4016AABG-DR-F
- AXI Options
 - Data = 512.
 - Address = 31

The Memory Device Interface Speed is used to set the interface speed. This is related to Reference Input Clock Speeds because it adjusts the speed. In addition, the Reference Input Clock Speed is greater than the Memory Device Interface Speed.

In this case, the Reference input Clock Speed is set to 8000 ps or 125MHz, which corresponds to the input clock frequency of the component.

In the configuration, the ID corresponding to the physical memory in the FPGA and the data and address size are also selected.

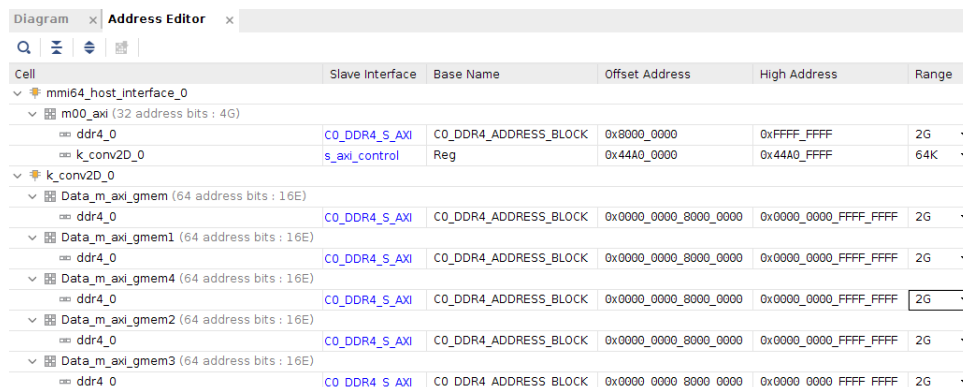
3.1.3.1. Address Memory

An important thing to do is to choose how to map the memory addresses. Memory can be accessed via AXI signals by two components: the host and the convolutional IP core.

In order to ensure that there is communication between host and kernel, host and memory, and kernel and memory, a mechanism based on address mapping is used.

In this case, the host can communicate with the kernel through addresses ranging from 0x44A0_0000 to 0x44A0_FFFF and can communicate with the memory through addresses ranging from 0x8000_0000 to 0xFFFF_FFFF.

The kernel also uses the same range of addresses the host uses to communicate with the memory (0x8000_0000 to 0xFFFF_FFFF) to exchange information with the memory.



Cell	Slave Interface	Base Name	Offset Address	High Address	Range
mmi64_host_interface_0					
m00_axi (32 address bits : 4G)					
ddr4_0	C0_DDR4_S_AXI	C0_DDR4_ADDRESS_BLOCK	0x8000_0000	0xFFFF_FFFF	2G
k_conv2D_0	s_axi_control	Reg	0x44A0_0000	0x44A0_FFFF	64K
k_conv2D_0					
Data_m_axi_gmem (64 address bits : 16E)					
ddr4_0	C0_DDR4_S_AXI	C0_DDR4_ADDRESS_BLOCK	0x0000_0000_8000_0000	0x0000_0000_FFFF_FFFF	2G
Data_m_axi_gmem1 (64 address bits : 16E)					
ddr4_0	C0_DDR4_S_AXI	C0_DDR4_ADDRESS_BLOCK	0x0000_0000_8000_0000	0x0000_0000_FFFF_FFFF	2G
Data_m_axi_gmem4 (64 address bits : 16E)					
ddr4_0	C0_DDR4_S_AXI	C0_DDR4_ADDRESS_BLOCK	0x0000_0000_8000_0000	0x0000_0000_FFFF_FFFF	2G
Data_m_axi_gmem2 (64 address bits : 16E)					
ddr4_0	C0_DDR4_S_AXI	C0_DDR4_ADDRESS_BLOCK	0x0000_0000_8000_0000	0x0000_0000_FFFF_FFFF	2G
Data_m_axi_gmem3 (64 address bits : 16E)					
ddr4_0	C0_DDR4_S_AXI	C0_DDR4_ADDRESS_BLOCK	0x0000_0000_8000_0000	0x0000_0000_FFFF_FFFF	2G

Figure 3.8: Address editor

3.2 Test

In this section, the system testing process will be explained. First, in the Section 3.2.1, we will explain what the System ILA core is and how it was used in the design. then we will show how to program the FPGA (Section 3.2.2) and the test bench used to test the system (Section 3.2.3). Finally, through the hardware manager, we will perform validation of the results obtained from running the System (Section 3.2.5).

3.2.1. System ILA

The customizable IP core System Integrated Logic Analyzer (System ILA) is a logic analyzer that can be used to monitor the internal signals and interfaces of a hardware design. The core also offers to interface debug and monitoring capability along with AXI4-MM and AXI4-Stream protocol checking. Because the System ILA core is synchronous to the design being monitored, all design clock constraints applied to the design are also applied to the components of the System ILA core. As we can see in figure 3.9, two ILA cores have been used in this project because they are connected to two different modules with two different clock frequencies.

In this case, the ILA cores were connected to the AXI bus to monitor the correct reading and writing in the various blocks.

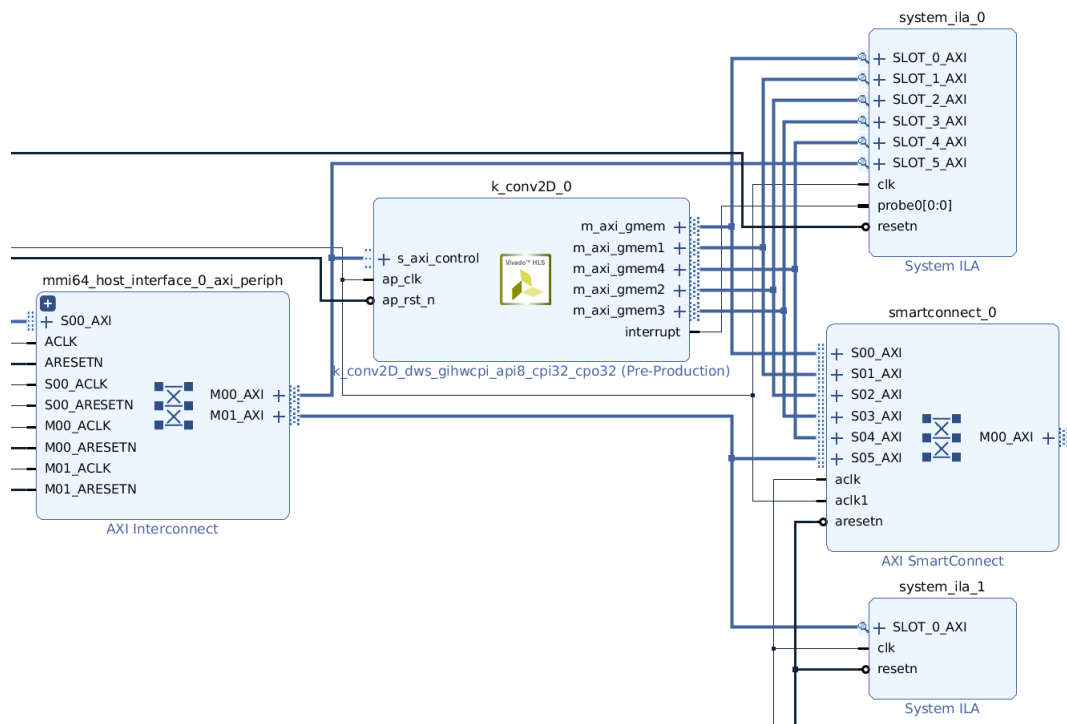


Figure 3.9: Ila core

3.2.2. Programming the FPGA

Prodesign tools were used to program the FPGA. In particular, the following commands were used:

- profpga_builder GUI to create board configuration setups and to perform runtime accesses to the system.

```
1 profpga_builder --config -only --scan = 169.254.0.2
```

This command scan a system, create the configuration file only.

- profpga_run

```
1 profpga_run <configuration_file> <mode> <[options]>
2 profpga_run -get-exe-version
```

profpga_run is the command-line tool to access the FPGA system. It will be called with a mandatory mode switch, optional parameters and switches, and a path to a configuration file that contains a section for MMI configuration and a section for proFPGA system configuration.

```
1 profpga_run IP_169.254.0.2.cfg --up
```

This command initializes and configures the system based on the contents of the configuration file.

- `profpga_brdgen`

```
1 profpga_brdgen <cfgfile> <mode> <[options]>
```

`profpga_brdgen` is a command line tool to generate various VHDL/Verilog toplevel files, constraint files, board description files and a design based self-test to test FPGA interconnections at speed.

```
1 profpga_brdgen IP_169.254.0.2.cfg --xdc
```

This command generates for each FPGA an XDC constraints file.

3.2.3. Testbench

In this section, we will show how to initialize the FPGA (Section 3.2.3.1). In addition, the complete operation of the system will be explained through the code description (Section 3.2.3.2). Finally, there will be a brief demonstration of the execution and how it was debugged through the hardware manager (Section 3.2.4).

3.2.3.1. Initialize FPGA

Before running the actual program, it is necessary to initialize the FPGA so that the various components in the project are recognized. In particular, the first step is to analyze the configuration file. As can be seen in the code is present, the bitstream is loaded in the FPGA. Clocks are described. In this case, there are two clocks, one with a local 100 MHz source and one with a 125 MHz source. Also, all the synchronization configuration is specified, and finally, what kind of memory is in the FPGA is specified.

```
1 motherboard_1 :  
2 {  
3     type = "MB-4M-R2";  
4     fpga_module_ta3 :  
5     {
```

```
6     type = "FM-XCKU115-R1";
7     bitstream = "design_1_wrapper.bit";
8     speed_grade = 2;
9     v_io_bal = "AUTO";
10 };
11 clock_configuration :
12 {
13     clk_0 :
14     {
15         source = "LOCAL";
16     };
17     clk_2 :
18     {
19         source           = "125MHz" ;
20         multiply         =      5 ;
21         divide           =      5 ;
22     };
23 };
24 sync_configuration :
25 {
26     sync_0 :
27     {
28         source = "GENERATOR";
29     };
30     sync_2 :
31     {
32         source = "GENERATOR";
33     };
34 };
35 };
36
37 sync_events = ( ( "motherboard_1.sync_2", "RESET_0" ), # low active reset signal
38               ( "sleep", 100 ), # wait for 1000 ms
39               ( "motherboard_1.sync_2", "RESET_1" ) # release reset
40             );
41
42 x_board_list = ( "ta1_eb1", "icc_1", "icc_2", "bb2_eb1", "tc1_eb1", "icc_3", "bd2_eb1"
43               , "ta3_eb1", "icc_4", "tc3_eb1" );
44 ta3_eb1 :
45 {
46     type = "BOARD";
47     vendor = "ProDesign";
48     name = "EB-PDS-DDR4-R3";
49     size = "A1A1";
50     positions = ( "motherboard_1.TA3" );
51     top_connectors = ( );
52     v_io_bal = "AUTO";
```



```

52  si5338_registermap_file = "RegisterMap_125MHz.txt";
53  si5338_validate_input_clocks_1_2_3 = "yes";
54  si5338_validate_input_clocks_4_5_6 = "no";
55  };
56  icc_4 :
57  {
58    type = "CABLE";
59    vendor = "ProDesign";
60    name = "IC-PDS-CABLE-R1";
61    size = "A1A1";
62    positions = ( "motherboard_1.TC4", "motherboard_1.TA4" );
63    v_io = "AUTO";
64  };
65  };

```

Listing 3.2: Configuration file

When the test bench is run, the first thing that is done is to read the configuration file, and a connection to the ProFPGA system is opened, using functions provided by PRO DESIGN. The function "profpga_open" (listing 3.3) opens a connection to a proFPGA system based on a given configuration file. This allows us to scan the System and check that all the components involved in the design are physically present.

```

1  //-----
2  // Open MMI64 connection
3  //-----
4  printf(NOW("INFO: Loading config file %s\n"), mmi64_config_file);
5  status = profpga_open (&profpga , mmi64_config_file);
6  if (status != E_PROFPGA_OK) {
7    printf("ERROR: Failed connect to PROFPGA system (%s)\n", profpga_strerror(status));
8    return status;

```

Listing 3.3: Loading configuration file

In the listing 3.4 the function "mmi64_identify_scan" get information about connected mmi64 domain: type, addr and data width of each available modules must be called by the user prior to call of "mmi64_identify_by_." functions (listing 3.5).

```

1  //-----
2  // Scan MMI64 connected device
3  //-----
4  mmi64_status = mmi64_identify_scan(profpga->mmi64_domain);

```

```

5  if (mmi64_not_ok(mmi64_status)) {
6      printf(NOW("ERROR (main): Failed to scan \gls{MMI}64 domain - mmi64_status: %d\n"),
              mmi64_status);
7      return mmi64_status;
8  }
9  // print scan results
10 mmi64_status = mmi64_info_print(profpga->mmi64_domain);

```

Listing 3.4: Components scanning

Scanning the System gives as output the figure 3.10, which shows information about identified MMI64 domain. The critical thing to check is the presence of the AXI_master, without which the host would not be able to communicate with the System.

```

File-based test...
Using configuration file IP_169.254.0.2.cfg
INFO: Loading config file IP_169.254.0.2.cfg
INFO : This system has 1 motherboard(s)
INFO : Short FW revision report:
INFO : MB1: HW=2 CTRL=1825 CLK01=368 CLK23=368 CLK45=368 CLK67=368 ARM=2018B
INFO : Plugin '/opt/prodesign/profpga/proFPGA-2018B/plugin/linux_x86_64/si5338.so' loaded
-----+-----
MMI64 Address : Module ID : Module Type :
-----+-----
<root> : 0x800000ff : router : PORT_COUNT=17 PRESENCE=0b11111111100100
02 : 0x800001ff : router : PORT_COUNT=3 PRESENCE=0b111
02-00 : 0x11111111 : axi_master : AXI_ADDR_WIDTH=32 AXI_DATA_WIDTH=32 Rev.2
02-01 : 0x80000101 : regif : REGISTER_COUNT=128 REGISTER_WIDTH=16
02-02 : 0x80000102 : devzero :
05 : 0x80000005 : selectmapif :
06 : 0x80000006 : selectmapif :
07 : 0x80000007 : selectmapif :
08 : 0x80000008 : selectmapif :
09 : 0x80000009 : regif : REGISTER_COUNT=128 REGISTER_WIDTH=16
0a : 0x8000000a : regif : REGISTER_COUNT=48 REGISTER_WIDTH=16
0b : 0x8000000b : regif : REGISTER_COUNT=48 REGISTER_WIDTH=16
0c : 0x8000000c : regif : REGISTER_COUNT=52 REGISTER_WIDTH=16
0d : 0x8000000d : regif : REGISTER_COUNT=34 REGISTER_WIDTH=16
0e : 0x8000000e : mbox : MAX_SUPPORTED_MSG_SIZE=1024
0f : 0x8000000f : devzero :
10 : 0x80000010 : regif : REGISTER_COUNT=500 REGISTER_WIDTH=32
-----+-----

```

Figure 3.10: Scan MMI64 connected device

After the scanning, the function "mmi64_identify_by_type" identify all modules of a given module type and returns a list of all matching modules. Moreover, it checks the parameter "identified_modules_count", which represents the number of matching modules detected during identity run; In order to identify the module AXI_master (listing 3.5).

```

1  //-----
2  // Get all modules with MMI64_TID_M_AXIM ID
3  //-----
4  mmi64_status = mmi64_identify_by_type(profpga->mmi64_domain, MMI64_TID_M_AXIM, &
              identified_modules, &identified_modules_count);
5  if (mmi64_not_ok(mmi64_status)) {

```

```

6   printf(NOW("ERROR (main): Failed to identify mmi64 regif module - mmi64_status: %d\n
      "), mmi64_status);
7   return E_MMI64_INTERNAL_ERROR;
8   }
9
10  //-----
11  // Select one MMI64 Master device
12  //-----
13  if (identified_modules_count == 0){
14      printf(NOW("ERROR (main): Failed to identify mmi64 AXI master module\n"));
15      return mmi64_status;
16  }
17
18  test_module = identified_modules[0];

```

Listing 3.5: Modules identification

The last thing to do to finish configuring the FPGA is to open a connection to the AXI Master module domain, with the function "axim_open" (listing 3.6). This must be the first function called before any Read or Write access on the AXI interface. The function reads out AXI access parameters which are used for AXI read and write access.

```

1   //-----
2   // Open communication
3   //-----
4   test_axi_master = axim_open(test_module);
5   if (test_axi_master == NULL){
6       printf(NOW("ERROR (main): Connect process to AXI master failed \n"));
7       return E_MMI64_INTERNAL_ERROR;
8   }
9
10  uint32_t nullo, Register, Mask, Data;
11  size_t size = sizeof(uint32_t);
12
13  //disable autorestart
14  Data = 0;
15  axi_status = axim_write_32(test_axi_master, 0x1, size, ADDR_AP_CTRL, &Data);
16  if (axi_status != E_AXI_OK){
17      printf("ERROR (main): Write autorestart %d\n", axi_status);
18      return E_MMI64_INTERNAL_ERROR;
19  }

```

Listing 3.6: Open a connection to AXI Master module domain

Furthermore, finally, the program uses interrupts, so they must be enabled before starting actual execution.

```
1 //-----
2 // Enable interrupt
3 //-----
4
5 //InterruptGlobalEnable
6 Data = 1;
7 axi_status = axim_write_32(test_axi_master, 0x1, size, ADDR_GIE, &Data);
8 if (axi_status != E_AXI_OK){
9     printf("ERROR (main): Write Disable global interrupt %d\n", axi_status);
10    return E_MMI64_INTERNAL_ERROR;
11 }
12
13 //InterruptEnable
14 Data = 1;
15 axi_status = axim_write_32(test_axi_master, 0x1, size, ADDR_IER, &Data);
16 if (axi_status != E_AXI_OK){
17     printf("ERROR (main): Write Disable interrupt %d\n", axi_status);
18     return E_MMI64_INTERNAL_ERROR;
19 }
20
21 //no idle
22 do{
23     axi_status = axim_read_32(test_axi_master, 0x1, size, ADDR_AP_CTRL, &Data);
24     if (axi_status != E_AXI_OK){
25         printf("ERROR (main): Read Register address: %d\n", axi_status);
26         return E_MMI64_INTERNAL_ERROR;
27     }
28 }while(!((Data >> 2) & 0x1));
29
30 return axi_status;
31 }
```

Listing 3.7: Enable interrupts

3.2.3.2. Test bench

The test bench is structured in two parts, the first part is for the execution of the convolution by the CPU, and the second part is for the execution of the convolution in the FPGA.

The execution of the convolution through the CPU is carried out to test the execution in the FPGA. The final part of the test bench compares the results of the two executions.

The listing 3.8 contains the most significant functions, helpful in describing the general flow of the System's operation.

The program's first thing is to allocate buffers in the CPU's memory (`allocate_buffers()`). These buffers will contain the inputs passed to the application via an "input.data" file, which are parameters used to configure the convolution, and the data to be convolved, generated randomly using the "init_data()" function.

```
1 void compute(int *enable, int *cpu, int *retval) {
2     ...
3
4     allocate_buffers();
5     init_data();
6
7     run_kernel();
8
9     cpu_conv2D();
10
11    check_result(&max_difference, &num_differences);
12
13    deallocate_buffers();
14 }
```

Listing 3.8: Application flow

In addition to random data generation, the "init_data()" function calls another function, "init_data_fpga()" (listing 3.9), which is used to copy the data from the CPU memory to the memory of the FPGA. This ensures data consistency. When writing data to memory, special attention must be paid to the memory area in which the data is being written. The listing 3.9 shows that the addresses used to write the host to memory are from address `0x8000_0000` to `0xFFFF_0000`.

```
1 int init_data_fpga() {
2
3     int temp = 0;
4
5     // write data in input data buffer - ptr_data
6     size_t size_data_in_bytes = I_input * W * H * sizeof(data_type);
7     addr_mem_in = 0x80000000;
8     if (size_data_in_bytes > 1023) {
9         temp = size_data_in_bytes / 1024;
```

```

10 } else {
11     temp = 1;
12 }
13 axi_status = axim_write_block(test_axi_master , 8,AXI4_BURST_INCR, size_data_in_bytes /
    temp, addr_mem_in, size_data_in_bytes/temp, temp, data_in , mmi64_true);
14 if (axi_status != E_AXI_OK){
15     printf("ERROR (main): Write AXI data %d\n", axi_status);
16     return E_MMI64_INTERNAL_ERROR;
17 }
18
19 ...
20 return axi_status;
21 }

```

Listing 3.9: Function used to copy data from CPU memory to FPGA memory.

After the data has been stored in memory, the host can tell the convolution IP core which pointers contain the data addresses to apply the convolution to and does so using the "copy_in_fpga" function (listing 3.10). This time the addresses used by the host to communicate with the IP core are 0x44A0_0000 to 0x44A0_FFFF.

```

1 int copy_in_fpga(int o_iter_first , int o_iter_last , int o_iter_per_kernel){
2
3     // ptr_data_in
4     addr_mem = 0x0000000080000000;
5     Data = (uint32_t) addr_mem;
6     Data2 = (uint32_t) (addr_mem >> 32);
7     axi_status = axim_write_32(test_axi_master , 0x1, sizeof(uint32_t), ADDR_PTR_DATA_DATA,
    &Data);
8     if (axi_status != E_AXI_OK){
9         printf("ERROR (main): Write \gls{AXI} multi , status: %d\n", axi_status);
10        return E_MMI64_INTERNAL_ERROR;
11    }
12    axi_status = axim_write_32(test_axi_master , 0x1, sizeof(uint32_t), ADDR_PTR_DATA_DATA
    + 4, &Data2);
13    if (axi_status != E_AXI_OK){
14        printf("ERROR (main): Write AXI multi , status: %d\n", axi_status);
15        return E_MMI64_INTERNAL_ERROR;
16    }
17
18    ...
19
20    // ADDR_H_DATA
21    Data = (uint32_t ) H;

```

```

22  axi_status = axim_write_32(test_axi_master , 0x1, sizeof(uint32_t), ADDR_H_DATA, &Data)
    ;
23  if (axi_status != E_AXI_OK) {...
24
25  ...
26
27  return axi_status;
28  }

```

Listing 3.10: Function to input the necessary data to the convolution IP core

The FPGA needs to be enabled in order to perform convolution. The host then sets the "start" bit in the control register to one. In addition, in order to calculate the execution time, it is necessary to wait for the FPGA to finish, and this is done by reading the "done" bit in the control register (listing 3.11). When it is high, the FPGA will have finished execution, and the program can continue. The execution time is to be calculated so the efficiency can be evaluated.

After the execution of the convolution in the FPGA is finished, the convolution in the CPU can be started.

```

1  int run_fpga() {
2
3  //-----
4  //  Enable accelerator
5  //-----
6
7  //start the accelerator
8  axi_status = axim_read_32(test_axi_master , 0x1, sizeof(uint32_t), ADDR_AP_CTRL, &
    tempData);
9  Data = tempData & 0x80;
10  if (axi_status != E_AXI_OK){
11  printf("ERROR (main): Write AXI multi, status: %d\n", axi_status);
12  return E_MMI64_INTERNAL_ERROR;
13  }
14  Data2 = Data | 0x01;
15  axi_status = axim_write_32(test_axi_master , 0x1, sizeof(uint32_t), ADDR_AP_CTRL, &
    Data2);
16  if (axi_status != E_AXI_OK){
17  printf("ERROR (main): Write AXI multi, status: %d\n", axi_status);
18  return E_MMI64_INTERNAL_ERROR;
19  }
20  }

```

```

21 //-----
22 // Wait writing
23 //-----
24 Data = 0;
25
26 do{
27     axim_read_32(test_axi_master , 0x1, sizeof(uint32_t), ADDR_AP_CTRL, &Data);
28 }while (!((Data >> 1) & 0x1));
29
30 }

```

Listing 3.11: Start FPGA execution

When both CPU and FPGA executions have finished, the results can be compared to verify that the FPGA execution is correct. In order to do this, the output data saved in memory by the IP core after memory execution must be copied to the CPU memory (listing 3.12).

```

1 int copy_out_fpga(){
2     size_t size_output_in_bytes;
3     if ((enable_maxpooling) || (enable_avgpooling)) {
4         size_output_in_bytes = O_output * (W/2) * (H/2) * sizeof(data_type);
5     } else {
6         size_output_in_bytes = O_output * W * H * sizeof(data_type);
7     }
8     int temp = 0;
9     if (size_output_in_bytes > 1023) {
10        temp = size_output_in_bytes/1024;
11    } else {
12        temp = 1;
13    }
14    addr_mem_out = 0x84000000;
15    axi_status = axim_read_block(test_axi_master , 8,AXI4_BURST_INCR, size_output_in_bytes /
16        temp, addr_mem_out, size_output_in_bytes/temp, temp, out);
17    if (axi_status != E_AXI_OK){
18        printf("ERROR (main): Write AXI data %d\n", axi_status);
19        return E_MMI64_INTERNAL_ERROR;
20    }
21    return axi_status;
22 }

```

Listing 3.12: Copy data from FPGA memory to CPU memory

Finally, the initially allocated buffers can be deallocated, and the connection to the MMI64 domain can be closed (listing 3.13).


```
1 int close_mmi64_backend() {
2     // close mmi64 backend
3     mmi64_status = mmi64_close(&profpga->mmi64_domain);
4     if (mmi64_not_ok(mmi64_status)) {
5         printf(NOW("ERROR (main): Failed to close MMI64 domain - mmi64_status %d\n"),
6             mmi64_status);
7         return mmi64_status;
8     }
9     return mmi64_status;
}
```

Listing 3.13: Close connection to a MMI64 domain

3.2.4. Run test

Before running the application, the bitstream needs to be loaded into the FPGA. The command to do this is as follows:

```
1 profpga_run IP_169.254.0.2.cfg --up
```

In the Section 2.4.2.2 this command and all the others related to the software part of ProDesign are explained in detail.

Program execution is started with the following command:

```
1 ./main input.data IP_169.254.0.2.cfg
```

Where "input.data" are the parameters present in the table 3.1 and are used to configure the convolution. Instead "IP_169.254.0.2.cfg" is the configuration file described in 3.2.3.1. As input parameters (table 3.1), there are matrices of different sizes that require different resources and affect execution times.

id	H x W x L x O	EUP	ELP	RELU	MAXPOOL	AVGPOOL	SHIFT	DIRECTION SHIFT	POS SHIFT	CLIP	MINCLIP	MAXCLIP
1	4x4x1x1	1	1	1	0	1	0	0	0	0	0	0
2	8x8x5x1	1	1	0	0	1	0	0	0	0	0	0
3	8x8x5x1	1	1	0	0	1	0	0	0	0	0	0
4	8x8x3x4	1	1	1	0	0	0	0	0	0	0	0
5	32x32x1x1	1	1	1	1	0	1	0	3	0	0	0
6	32x32x1x1	1	1	1	0	0	0	0	0	0	0	0
7	8x8x1x16	1	1	1	0	0	0	0	0	0	0	0
8	8x8x3x8	1	1	1	0	0	0	0	0	0	0	0
9	32x64x1x1	1	1	1	1	0	1	1	2	1	-2	2
10	32x64x1x1	1	1	1	1	0	0	0	0	0	0	0
11	32x64x1x1	1	1	1	0	1	0	0	0	0	0	0
12	8x8x3x16	1	1	1	0	0	0	0	0	0	0	0
13	8x8x3x16	1	1	0	0	0	0	0	0	0	0	0
14	8x8x3x16	1	1	0	0	0	0	0	0	0	0	0
15	64x64x1x1	1	1	1	0	0	0	0	0	0	0	0
16	8x16x3x16	1	1	1	0	0	0	0	0	0	0	0
17	8x32x3x16	1	1	1	0	0	0	0	0	0	0	0
18	16x32x3x16	1	1	1	0	0	0	0	0	0	0	0
19	64x64x4x4	1	1	0	0	0	0	0	0	0	-3	5
20	32x64x3x16	1	1	1	0	0	0	0	0	0	0	0
21	32x64x12x8	1	1	1	0	0	0	0	0	0	0	0
22	32x64x6x16	1	1	1	0	0	0	0	0	0	0	0
23	32x64x12x16	1	1	0	0	0	0	0	0	0	0	0
24	256x256x4x4	1	1	1	1	0	0	0	0	0	0	0
25	32x64x12x128	1	1	1	0	0	0	0	0	0	0	0
26	256x256x8x8	1	1	1	0	0	0	0	0	0	0	0
27	256x256x16x16	1	1	1	0	1	0	0	0	0	0	0
28	256x256x35x27	1	1	1	0	0	0	0	0	0	0	0

Table 3.1: Input data test

In the figure 3.11 the execution of the application is shown, which ends with a "SUCCESS"; in addition to this check, another check was made using the Xilinx tool, which will be discussed in the Section 3.2.5.

```

=====
| Input: 64 x 64 x 4 x 4 | Kernel: 3 x 3 | Padding: 1 x 1 | Stride: 1 x 1 | |
|---|---|---|---|---|
| ReLU: No | MaxPooling: No | AvgPooling: No | Clipping: No (-3: 5) | Shift: No (LEFT, 0) |
|-----|-----|-----|-----|
| launching kernel 0 (output iterations 0 to 0) |
|-----|-----|-----|-----|
| Time 198 usec | Time per iteration 198 usec | Expected time 40 usec | Efficiency 0.2020 |
|-----|-----|-----|-----|
| SUCCESS |
|-----|-----|-----|-----|

=====
| Input: 32 x 64 x 1 x 1 | Kernel: 3 x 3 | Padding: 1 x 1 | Stride: 1 x 1 | |
|---|---|---|---|---|
| ReLU: Yes | MaxPooling: Yes | AvgPooling: No | Clipping: Yes (-2: 2) | Shift: Yes (RIGHT, 2) |
|-----|-----|-----|-----|
| launching kernel 0 (output iterations 0 to 0) |
|-----|-----|-----|-----|
| Time 171 usec | Time per iteration 171 usec | Expected time 20 usec | Efficiency 0.1170 |
|-----|-----|-----|-----|
| SUCCESS |
|-----|-----|-----|-----|

=====
| Input: 32 x 32 x 1 x 1 | Kernel: 3 x 3 | Padding: 1 x 1 | Stride: 1 x 1 | |
|---|---|---|---|---|
| ReLU: Yes | MaxPooling: Yes | AvgPooling: No | Clipping: No (0: 0) | Shift: Yes (LEFT, 3) |
|-----|-----|-----|-----|
| launching kernel 0 (output iterations 0 to 0) |
|-----|-----|-----|-----|
| Time 176 usec | Time per iteration 176 usec | Expected time 10 usec | Efficiency 0.0568 |
|-----|-----|-----|-----|
| SUCCESS |
|-----|-----|-----|-----|

=====
| Input: 8 x 8 x 5 x 1 | Kernel: 3 x 3 | Padding: 1 x 1 | Stride: 1 x 1 | |
|---|---|---|---|---|
| ReLU: No | MaxPooling: No | AvgPooling: Yes | Clipping: No (0: 0) | Shift: No (LEFT, 0) |
|-----|-----|-----|-----|
| launching kernel 0 (output iterations 0 to 0) |
|-----|-----|-----|-----|
| Time 147 usec | Time per iteration 147 usec | Expected time 0 usec | Efficiency 0.0000 |
|-----|-----|-----|-----|
| SUCCESS |
|-----|-----|-----|-----|

```

Figure 3.11: Application execution

3.2.4.1. Timing

Another thing we notice in the figure 3.11, is the presence of some times:

- Time: is the execution time of the program in microseconds;
- Time per iteration: is the time for each iteration;
- Expected time: estimated time of execution;
- Efficiency: the ability to use as few resources as possible during its execution.

```

1 gettimeofday(&prof_t1, NULL);
2
3 run_kernel();
4
5 gettimeofday(&prof_t2, NULL);
6
7 prof_time = ((prof_t2.tv_sec - prof_t1.tv_sec) * 1000000) + (prof_t2.tv_usec - prof_t1.
   tv_usec);
8 unsigned long long time = prof_time;

```

```
9 unsigned long long time_per_iteration = prof_time / i_iter / o_iter;  
10 unsigned long long expected_time_one_iteration = W * H * 1000;  
11 unsigned long long expected_time = (expected_time_one_iteration * i_iter * o_iter) /  
    100000;  
12 float efficiency = ((float)expected_time / (float)time);
```

Listing 3.14: Execution time

In this work, a study was also carried out on execution times. Using the expected execution time and the actual time, the efficiency of execution was calculated.

The figure 3.12 shows the relationship between configuration and efficiency. This is by communication overhead. There is more time spent in communication than in execution for small input sizes.

So, when the input size increases, the communication is still constant, but the execution time is longer, minimizing the communication overhead and making efficiency approaches 1.

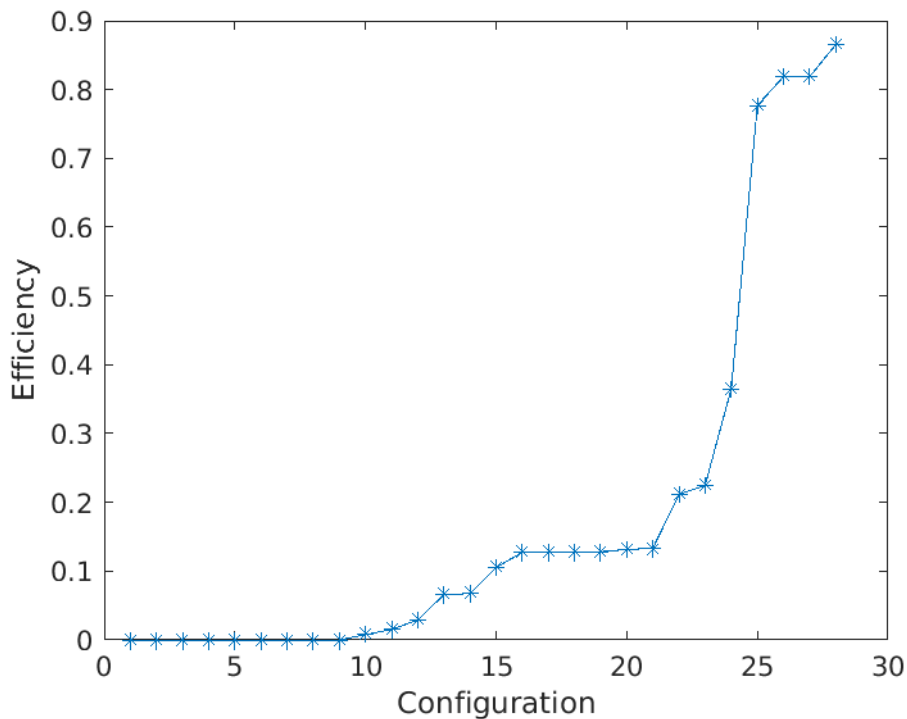


Figure 3.12: Relation between the configuration and the efficiency

3.2.5. Hardware manager

In addition to a check carried out using the on-screen printout of the successful execution termination, a graphical check can be made by seeing the correct reading and writing data in the registers. Vivado allows this type of check to be made via the hardware manager. The hardware manager uses the system ILA core (described in 3.2.1) to monitor the various signals.

The figure 3.13 shows the ILA system configuration to set the trigger mode to trigger on various events in hardware [18].

- BASIC_ONLY: The ILA Basic Trigger Mode can be used to trigger the ILA core when a basic AND/OR functionality of debug probe comparison result is satisfied.
- Use ALWAYS and BASIC capture modes to control filtering of data to be captured.
- The number of ILA capture windows.
- The data depth of the ILA capture window.
- The trigger position to any sample within the capture window.

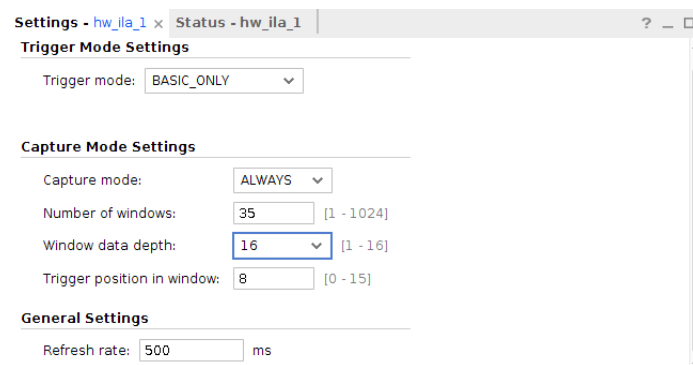


Figure 3.13: Ila settings

Another thing to set is when to trigger. In this case, figure 3.14, the System is triggered when there is a 0 → 1 change of state of the "WVALID" signal, meaning when the host performs a valid write to memory.

Name	Operator	Radix	Value	Port	Comparator Usage
slot_5 : mmi64_host_interface_0_axi_periph_M00_AXI : WVALID	==	[B]	R	probe163[0]	1 of 1

Figure 3.14: Trigget setup

The figure 3.15, shows at which points the System was triggered. As it is possible to see, a "1" has been written to address "0x04", this address corresponds to ADDR_GIE (Global Interrupt Enable) and concerns the global enablement of interrupts. The waveform perfectly mirrors the 3.7 code, which implies that the writer was correct. This step has been iterated for each reading and writing in the FPGA.

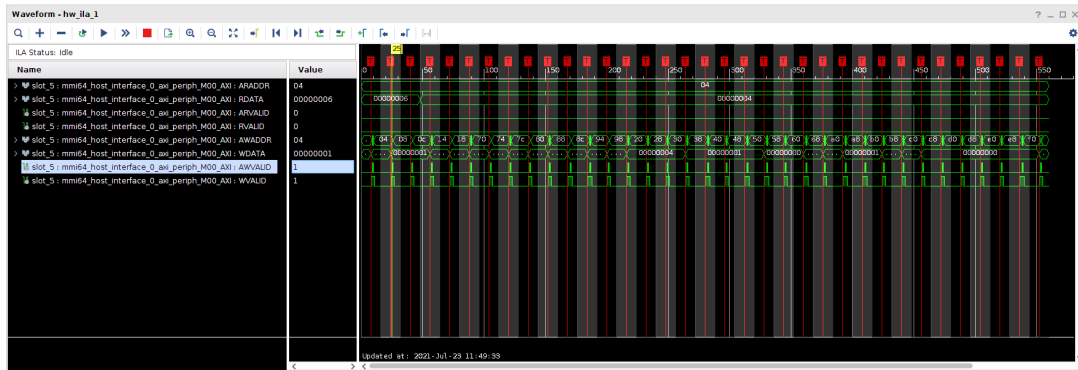


Figure 3.15: Waveform

CHAPTER 4

Multi-FPGA

As introduced in the introduction of Chapter 3, the goal of this project is to use a multi-FPGA prototype as a computational platform. In the previous Chapter, the system was described using a single FPGA; in this one, however, the discussion will be expanded using a multi-FPGA system. Specifically, the design of the system will be explained, going on to outline the differences with the single-FPGA design (Section 4.1). Moreover, finally, how to program the individual FPGAs to work concurrently will be described (Section 4.2).

4.1 Design

For the realization of this final part of the project, the configuration of the MANGO project described in Section 1.1 was used. Specifically, 4 FPGAs were used, independent of each other, connected to a host via PCIe (figure 4.1).

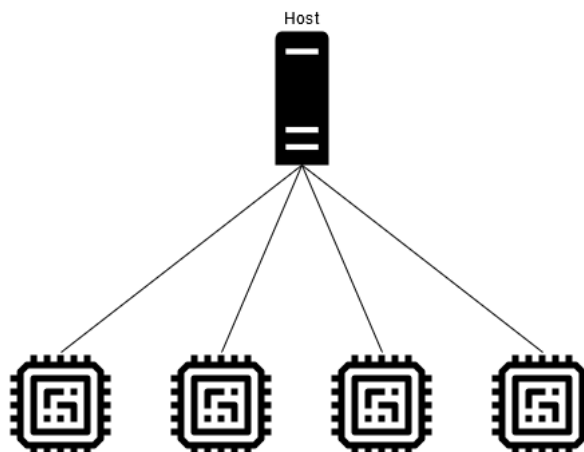


Figure 4.1: design multi-FPGA

The design for communication with individual FPGAs is the same as described in Section 3.1. The only thing that needs to be changed is the configuration of the MMI64_host_interface module (Section 3.1.1). As shown in the figure, the MMI64_host_interface module has a parameter named "Id Module", this parameter is mandatory to set and is unique for each FPGA. The identifier must be unique because it is used by the host to distinguish and recognize the different devices.

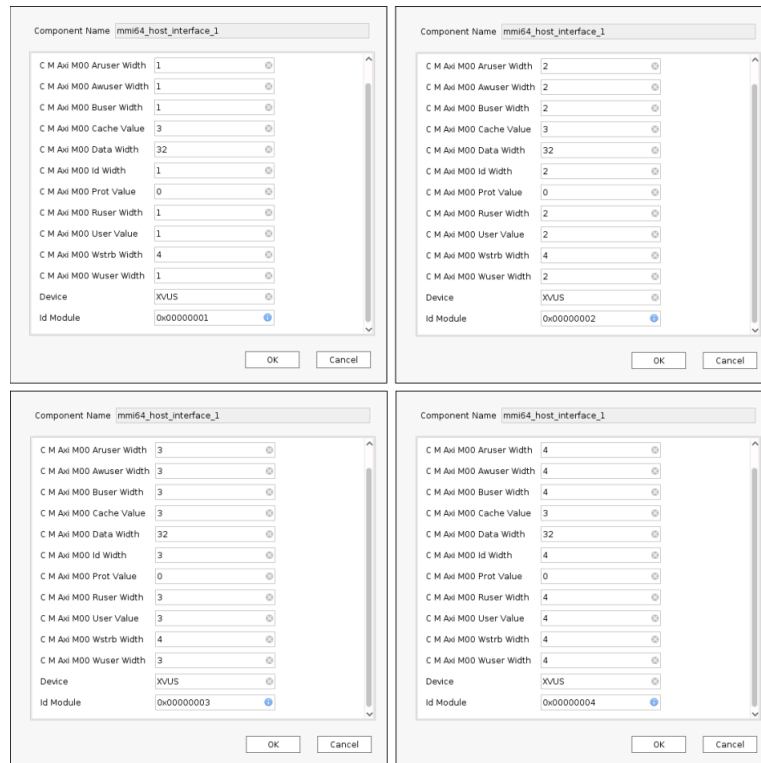


Figure 4.2: Configuration mmi64_host_interface multi-FPGA

4.2 Test

This Section will describe how the bitstream is loaded into the different FPGAs (Section 4.2.1), how the host can communicate with the different devices. And the algorithm used to make the host send inputs to the different FPGAs (Section 4.2.2).

4.2.1. Inizialize FPGA

As the initialization was performed in the single-FPGA case (Section 3.2.3.1), it must also be done for the multi-FPGA case.

Considering that the design of the single FPGAs of the multi-FPGA system is identical to the single-FPGA case, the configuration parameters are the same that were described in Section 3.2.3.1. So for each design, a bitstream was generated that will be loaded into the FPGA via the configuration file. In the configuration file, each FPGA module was associated with its respective bitstream (listing 4.1).

```
1 motherboard_1 :
2 {
3   type = "MB-4M-R2";
4   fpga_module_ta1 :
5   {
6     type = "FM-XCKU115-R1";
7     bitstream = "design_1_wrapper_1.bit";
8   };
9   fpga_module_tc1 :
10  {
11    type = "FM-XCKU115-R1";
12    bitstream = "design_1_wrapper_2.bit";
13  };
14  fpga_module_ta3 :
15  {
16    type = "FM-XCKU115-R1";
17    bitstream = "design_1_wrapper_3.bit";
18  };
19  fpga_module_tc3 :
20  {
21    type = "FM-XCKU115-R1";
22    bitstream = "design_1_wrapper_4.bit";
23  };
```

Listing 4.1: Configuration file multi_fpga

4.2.2. Test bench

As mentioned in Section 4.1, each MMI64_host_interface module, has its own identifier. This can also be seen in the test bench. The figure 4.3, shows that after scanning the system, 4 AXI_masters are recognized. With the identifiers of the AXI_masters the FPGAs are initialized and are associated with variables, which are used as reference to be able to read/write in the different devices (listing 4.2).

```

1 //-----
2 // Select one MMI64 Master device
3 //-----
4 if (identified_modules_count == 0){
5     printf(NOW("ERROR (main): Failed to identify mmi64 AXI master module\n"));
6     return mmi64_status;
7 }
8
9 init_fpga_1(identified_modules[0]);
10 init_fpga_2(identified_modules[1]);
11 init_fpga_3(identified_modules[2]);
12 init_fpga_4(identified_modules[3]);
13
14 ID_1 = test_axi_master_1;
15 ID_2 = test_axi_master_2;
16 ID_3 = test_axi_master_3;
17 ID_4 = test_axi_master_4;

```

Listing 4.2: Identify AXI_master module

As in Section 3.2.3.1, here too, it is necessary to check that the AXI_masters are present. In the figure 4.3, the four AXI_masters that are available and have been assigned a unique ID, discussed in Section 4.1, are highlighted.

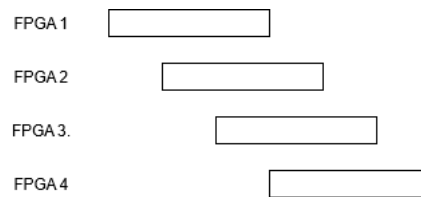
MMI64 Address	Module ID	Module Type	
<root>	0x800000ff	router	PORT_COUNT=17 PRESENCE=0b1111111111111110
01	0x800001ff	router	PORT_COUNT=3 PRESENCE=0b111
01-00	0x00000001	axi_master	AXI_ADDR_WIDTH=32 AXI_DATA_WIDTH=32 Rev.2
01-01	0x80000101	regif	REGISTER_COUNT=128 REGISTER_WIDTH=16
01-02	0x80000102	devzero	
02	0x800001ff	router	PORT_COUNT=3 PRESENCE=0b111
02-00	0x00000003	axi_master	AXI_ADDR_WIDTH=32 AXI_DATA_WIDTH=32 Rev.2
02-01	0x80000101	regif	REGISTER_COUNT=128 REGISTER_WIDTH=16
02-02	0x80000102	devzero	
03	0x800001ff	router	PORT_COUNT=3 PRESENCE=0b111
03-00	0x00000002	axi_master	AXI_ADDR_WIDTH=32 AXI_DATA_WIDTH=32 Rev.2
03-01	0x80000101	regif	REGISTER_COUNT=128 REGISTER_WIDTH=16
03-02	0x80000102	devzero	
04	0x800001ff	router	PORT_COUNT=3 PRESENCE=0b111
04-00	0x00000004	axi_master	AXI_ADDR_WIDTH=32 AXI_DATA_WIDTH=32 Rev.2
04-01	0x80000101	regif	REGISTER_COUNT=128 REGISTER_WIDTH=16
04-02	0x80000102	devzero	
05	0x80000005	selectmapif	
06	0x80000006	selectmapif	
07	0x80000007	selectmapif	
08	0x80000008	selectmapif	
09	0x80000009	regif	REGISTER_COUNT=128 REGISTER_WIDTH=16
0a	0x8000000a	regif	REGISTER_COUNT=48 REGISTER_WIDTH=16
0b	0x8000000b	regif	REGISTER_COUNT=48 REGISTER_WIDTH=16
0c	0x8000000c	regif	REGISTER_COUNT=52 REGISTER_WIDTH=16
0d	0x8000000d	regif	REGISTER_COUNT=34 REGISTER_WIDTH=16
0e	0x8000000e	mbox	MAX_SUPPORTED_MSG_SIZE=1024
0f	0x8000000f	devzero	
10	0x80000010	regif	REGISTER_COUNT=500 REGISTER_WIDTH=32

Figure 4.3: Scan MMI connected device**4.2.2.1. Test bench**

The critical part of this part of the project is to find a way to send the configuration inputs so that the FPGAs can work simultaneously. Taking into consideration that the host works sequentially.

The basic idea is based on the concept of a pipeline, i.e. inputs are sent sequentially one at a time without waiting for the execution of the previous configuration to finish.

So, as the figure 4.4 shows, the next FPGA can start executing before the previous one has finished. In this way, even comparing with the single-FPGA system, the execution time of the single configuration remains unchanged. However, if the total time of all configurations is considered, the multi-FPGA model is more efficient. Of course, the more configurations that need to be executed, the more efficient the multi-FPGA system will be.

**Figure 4.4:** Pipelined execution

The basic idea is to use a buffer to insert the available FPGA, which can be used to perform the convolution. Initially, the buffer has already been filled with the IDs of the four FPGA, as they are all ready to be used (listing 4.3).

```

1 int buffer[4];
2
3     buffer[0] = 1;
4     buffer[1] = 2;
5     buffer[2] = 3;
6     buffer[3] = 4;

```

Listing 4.3: Buffer FPGA available

Considering that we do not wait for an execution to end, a "count_read" counter is used to keep track of the running configurations. This counter is incremented when the configuration is input to a device and is decremented when the execution is finished.

After that, the "read_interrupt" function is used to retrieve the ID of the FPGA ready to be used.

Once the available FPGA has been extracted, all the parameters needed to execute the convolution are passed, and the accelerator is started (listing 4.4).

```

1
2  while (!read_test_file(&enable, &cpu)) {
3
4  count_read++;
5
6  int interr=0;
7
8  interr = read_interrupt();
9
10
11 if (interr == 1){
12     H_1 = H; W_1 = W; I_1 = I; O_1 = O; enable_upper_padding_1 = enable_upper_padding;
13     enable_lower_padding_1 = enable_lower_padding; enable_relu_1 = enable_relu;
14     enable_maxpooling_1 = enable_maxpooling; enable_avgpooling_1 = enable_avgpooling;
15     enable_shift_1 = enable_shift; dir_shift_1 = dir_shift; pos_shift_1 =
16     pos_shift;
17     enable_clipping_1 = enable_clipping; min_clip_1 = min_clip; max_clip_1 = max_clip;
18
19     // derived arguments
20     rows_1 = H;
21     I_kernel_1 = ((I + (CPI - 1)) / CPI) * CPI;
22     O_kernel_1 = ((O + (CPO - 1)) / CPO) * CPO;
23     i_iter_1 = (I + (CPI - 1)) / CPI;
24     o_iter_1 = (O + (CPO - 1)) / CPO;
25     global_offset_1 = 0;
26     GI_1 = I_kernel / CPI;
27     GO_1 = O_kernel / CPO;
28     if (enable_maxpooling || enable_avgpooling) {HO_1 = H / 2; WO_1 = W / 2;} else {
29         HO_1 = H; WO_1 = W;}
30
31 #ifdef IHW_DATA_FORMAT
32     I_input_1 = I;
33     O_output_1 = O;
34 #endif
35 #ifdef GIHWCPIData_FORMAT
36     I_input_1 = ((I + (CPI - 1)) / CPI) * CPI;
37     O_output_1 = ((O + (CPO - 1)) / CPO) * CPO;
38 #endif
39     compute_fpga_1(&enable, &cpu, &retval);
40 }
41 else if (interr== 2){

```

```
38     ...
39 }
40 else if (interr == 3){
41     ...
42 }
43 else if (interr == 4){
44     ...
45 }
46 }
```

Listing 4.4: Assign configuration to FPGA

The "read_interrupt" function (listing 4.5) first reads the control register to check if any execution in the FPGA has terminated. If so, it also starts CPU execution to perform a correctness check. After CPU execution, the FPGA ID can be put back into the buffer for the next execution.

Finally, it checks that the buffer is not empty and picks up the value at the head of the queue.

```
1 int read_interrupt(){
2     do{
3
4         D1 = 0, D2 = 0, D3 = 0, D4 = 0;
5
6         axim_read_32(ID_1, 0x1, sizeof(uint32_t), ADDR_AP_CTRL, &D1);
7         axim_read_32(ID_2, 0x1, sizeof(uint32_t), ADDR_AP_CTRL, &D2);
8         axim_read_32(ID_3, 0x1, sizeof(uint32_t), ADDR_AP_CTRL, &D3);
9         axim_read_32(ID_4, 0x1, sizeof(uint32_t), ADDR_AP_CTRL, &D4);
10
11         if ((D1 >> 1) & 0x1){
12             compile_CPU_1(&retval_check);
13             push(1);
14         }
15         if ((D2 >> 1) & 0x1){
16             compile_CPU_2(&retval_check);
17             push(2);
18         }
19         if ((D3 >> 1) & 0x1){
20             compile_CPU_3(&retval_check);
21             push(3);
22         }
23         if ((D4 >> 1) & 0x1){
24             compile_CPU_4(&retval_check);
```

```
25     push(4);
26     }
27
28 } while(buffer[0]!=0);
29
30 r = pull();
31
32 return r;
33 }
```

Listing 4.5: Read interrupt and CPU compiling

When a FPGA has completed its execution, its ID can be inserted back into the buffer using the 'push' function (listing 4.6). This function inserts the ID in the queue, increments the index to insert a new ID and decrements the execution counter.

```
1 void push(int id){
2     buffer[fpga_available] = id;
3     fpga_available++;
4     count_read--;
5 }
```

Listing 4.6: Push in the FPGA buffer

The queue used in this algorithm is a FIFO (First In First Out) queue, i.e. the first value entered will be the first to leave. For this reason, when a value is taken from the buffer, the value at the head is pulled. Subsequently, all other IDs will be shifted, and the available FPGA will be decreased (listing 4.7).

```
1 int pull(){
2
3     D = buffer[0];
4
5     for(int i=1; i<fpga_available; i++)
6         buffer[i-1] = buffer[i];
7     for(int i=fpga_available; i<50; i++)
8         buffer[i] = 0;
9
10    fpga_available--;
11
12    return D;
13 }
```

Listing 4.7: Push from the FPGA buffer

Before we can consider the execution of all convolutions complete, we must check that the FPGA's are all free and that there are no configurations still running (listing 4.8).

```
1 printf("-----finish configurations -----\\n");
2
3 while(count_read != 0 ){
4     read_interrupt();
5 }
6
7 close_test_file();
8 }
```

Listing 4.8: Checking that all executions have finished**4.2.3. Run test**

To execute the program, the same methodology as for the single-FPGA design was followed, using the same inputs (Section 3.2.4).

The figure 4.5, shows the output of the test bench execution, where it is shown that all FPGAs are running.

```

fpga used: 1
fpga used: 2
=====
| Input: 64 x 64 x 4 x 4 | Kernel: 3 x 3 | Padding: 1 x 1 | Stride: 1 x 1 | |
|---|---|---|---|---|
| ReLU: No | MaxPooling: No | AvgPooling: No | Clipping: No (-3; 5) | Shift: No (LEFT, 0) |
|-----|-----|-----|-----|
| SUCCESS |
=====
fpga used: 2
fpga used: 1
=====
| Input: 32 x 64 x 1 x 1 | Kernel: 3 x 3 | Padding: 1 x 1 | Stride: 1 x 1 | |
|---|---|---|---|---|
| ReLU: Yes | MaxPooling: Yes | AvgPooling: No | Clipping: Yes (-2; 2) | Shift: Yes (RIGHT, 2) |
|-----|-----|-----|-----|
| SUCCESS |
=====
fpga used: 3
fpga used: 2
=====
| Input: 32 x 32 x 1 x 1 | Kernel: 3 x 3 | Padding: 1 x 1 | Stride: 1 x 1 | |
|---|---|---|---|---|
| ReLU: Yes | MaxPooling: Yes | AvgPooling: Yes | Clipping: No (0; 0) | Shift: Yes (LEFT, 3) |
|-----|-----|-----|-----|
| SUCCESS |
=====
fpga used: 4
fpga used: 1
=====
| Input: 8 x 8 x 5 x 1 | Kernel: 3 x 3 | Padding: 1 x 1 | Stride: 1 x 1 | |
|---|---|---|---|---|
| ReLU: No | MaxPooling: No | AvgPooling: Yes | Clipping: No (0; 0) | Shift: No (LEFT, 0) |
|-----|-----|-----|-----|
| SUCCESS |
=====
fpga used: 1
fpga used: 2
=====
| Input: 4 x 4 x 1 x 1 | Kernel: 3 x 3 | Padding: 1 x 1 | Stride: 1 x 1 | |
|---|---|---|---|---|
| ReLU: Yes | MaxPooling: No | AvgPooling: No | Clipping: No (0; 0) | Shift: No (LEFT, 0) |
|-----|-----|-----|-----|
| SUCCESS |
=====
fpga used: 2
fpga used: 1
=====
| Input: 32 x 64 x 1 x 1 | Kernel: 3 x 3 | Padding: 1 x 1 | Stride: 1 x 1 | |
|---|---|---|---|---|
| ReLU: Yes | MaxPooling: Yes | AvgPooling: Yes | Clipping: No (0; 0) | Shift: No (LEFT, 0) |
|-----|-----|-----|-----|
| SUCCESS |
=====

```

Figure 4.5: Multi-FPGA execution

CHAPTER 5

Conclusions

In this thesis, a co-design hardware and software was developed by using a multi-FPGA prototype as a computational platform.

This project allowed me to understand the potential of High-Level Synthesis and generate an IP core starting from a high-level language. It also allowed me to work with new technologies made available by ProDesign. To understand how a host communicates with an FPGA and how to best test it in hardware using the ILA core. Finally, it was very instructive to design and develop a multi-FPGA system. In addition, I had the opportunity to see the system that performs the 2D convolution in operation, making considerations about the system's efficiency. I obtained an efficiency of 90%, considering a theoretical expected time compared to the real execution time.

As future developments of this project, there is the intention to extend the support of this system to the other types of FPGAs in the cluster.

Bibliography

- [1] “The mango project website,” 2017.
- [2] “Opencl overview,” 2021.
- [3] Xilinx, “Ultrascale fpga product tables and product selection guide.” Product documentation.
- [4] Xilinx, “Ds180(v2.6.1). 7 series fpgas data sheet: Overview.” Product documentation.
- [5] Xilinx, “Ds190 (v1.11.1). zynq-7000 soc data sheet: Overview.” Product Specification.
- [6] Intel, “Intel® stratix® 10 gx/sx device overview.” Product Specification.
- [7] “The recipe project website,” 2018.
- [8] “The deephealth project website,” 2019.
- [9] S. D. Brown, R. J. Francis, J. Rose, and Z. G. Vranesic, *Field-programmable gate arrays*, vol. 180. Springer Science & Business Media, 1992.
- [10] D. Pellerin and S. Thibault, *Practical FPGA programming in C*. Prentice Hall Press, 2005.
- [11] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach, “An introduction to high-level synthesis,” *IEEE Design & Test of Computers*, vol. 26, no. 4, pp. 8–17, 2009.
- [12] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, “High-level synthesis for fpgas: From prototyping to deployment,” *IEEE Transactions on*

Computer-Aided Design of Integrated Circuits and Systems, vol. 30, no. 4, pp. 473–491, 2011.

- [13] P. D. E. GmbH, “Ug1399 vitis high-level synthesis user guide,” 2021.
- [14] I. Xilinx, “Vivado design suite user guide, using the vivado ide,” 2021.
- [15] V. Xilinx, “Vivado design suite user guide-high-level synthesis,” 2018.
- [16] P. D. E. GmbH, “Ud001 profpga hardware user manual,” 2018.
- [17] P. D. E. GmbH, “An002 getting started with mmi64,” 2016.
- [18] P. D. E. GmbH, “Ug908 vivado design suite user guide,” 2021.