


Università degli Studi di Napoli Federico II

Ph.D. Program in

Information Technology and Electrical Engineering

XXXVI Cycle

Thesis for the Degree of Doctor of Philosophy

Development of innovative techniques and methodologies for analysis and testing of Storage Systems interfaces based on System on Chip

by

Marco Vitone

Advisor: Prof. Nicola Petra

Co-advisor: Eng. Claudio Giaccio

Scuola Politecnica e delle Scienze di Base

Dipartimento di Ingegneria Elettrica e delle Tecnologie dell'Informazione

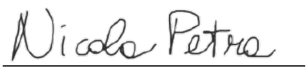
Development of innovative techniques and methodologies for analysis and testing of Storage Systems interfaces based on System on Chip

Ph.D. Thesis presented
for the fulfillment of the Degree of Doctor of Philosophy
in Information Technology and Electrical Engineering
by

Marco Vitone

October 2023

Approved as to style and content by



Prof. Nicola Petra, Advisor

Eng. Claudio Giaccio, Co-advisor

Università degli Studi di Napoli Federico II

Ph.D. Program in Information Technology and Electrical Engineering
XXXVI cycle - Chairman: Prof. Stefano Russo

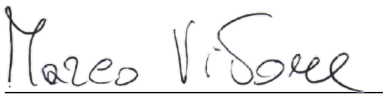
http://itee.dieti.unina.it

Candidate's declaration

I hereby declare that this thesis submitted to obtain the academic degree of Philosophiæ Doctor (Ph.D.) in Information Technology and Electrical Engineering is my own unaided work, that I have not used other than the sources indicated, and that all direct and indirect sources are acknowledged as references.

Parts of this dissertation have been published in international journals and/or conference articles (see list of author's publications at the end of the thesis).

Napoli, December 8, 2023



Marco Vitone

Abstract

System on Chip (SoC) is a complex Integrated Circuit (IC) that includes several components such as CPU, Programmable Logic (FPGA), on-chip memory, storage interfaces, and so on. Over the last decades, the SoC has been widely used in signal processing, communication, networking, and several industrial applications, from the automotive to storage systems management. Although the SoC platform represents an optimal solution for large-scale industrial product development, the SoC complexity determines a huge effort to validate the entire system. During my PhD program, I collaborated with Micron Technology and I worked on a state-of-the-art SoC device and storage system. One of the crucial steps of the design flow consists of the verification of the entire system and interfaces of the product. To address this issue, my research activity focuses on the development of innovative techniques for the validation of storage systems based on SoC. More specifically, a simulation environment has been developed for a custom Micron SoC according to the latest verification methodologies accepted in the Literature as well as in industrial companies. However, the simulations of complex devices, which include a huge number of peripherals, semiconductor intellectual proprieties, and bus interfaces, became a bottleneck in terms of time consumption. To this end, in this dissertation, an innovative *hardware emulation technique* has been presented which aims to overcome the simulation time bottleneck by integrating the hardware-in-the-loop in the verification flow.

Furthermore, another important issue related to the SoC design consists of the hardware accelerator implementation and system-level integration. My academic research also addresses this aspect of the SoC. In particular, a digital data path for the hardware acceleration of a Convolutional Neural Networks (CNNs) has been presented. First of all, a novel Fast FIR Algorithm (FFA) for mono- and bi-dimensional convolution has been explored, then the hardware implementation of the proposed algorithm has been presented.

Keywords: System on Chip, FPGA, digital verification, hardware accelerators, Fast Fir Algorithm, Convolutional Neural Network.

Sintesi in lingua italiana

Un System on Chip è un complesso circuito integrato che include numerevoli componenti come processori, logica programmabile basata su dispositivi FPGA, memoria integrata su chip. Nell'ultima decade, i dispositivi SoC sono stati ampiamente utilizzati nell'elaborazione dei segnali, nella telecomunicazioni e in diversi ambiti di interesse industriale, come nel comparto automobilistico o nello sviluppo di sistemi di memoria. Anche se dispositivi basati su tecnologia SoC rappresentano un'ottima soluzione per lo sviluppo di un'ampia varietà di prodotti industriali, la complessità di tali dispositivi determina un importante sforzo per validare sistemi che incorporano questa tipologia di circuiti integrati. Durante il percorso di dottorato, ho collaborato con Micron Technology ed ho lavorato su dispositivi SoC e sistemi di memoria allo stato dell'arte.Una delle fasi cruciali del processo di sviluppo dei sistemi digitali consiste nella verifica dell'intero sistema e delle relative interfacce integrate nel prodotto finale. Per affrontare tale tematica, la mia attività di ricerca si è focalizzata sullo sviluppo di tecniche innovative per la validazione di sistemi di memoria basati su dipositivi SoC. E' stato sviluppato un ambiente di simulazione per un SoC dell'azienda Micron in accordo con le più recenti metodologie di verifica accettate in Letteratura. Tuttavia, le simulazioni di dispositivi complessi, che includono un enorme quantitativo di periferiche, circuiti digitali e bus di interfacciamento, possono introdurre limitazioni a causa dei lunghi tempi di calcolo richiesti. A tal proposito, in questo lavoro è presentato una innovativa tecnica di emulazione hardware, la quale ha come obiettivo il superamento del limite temporale introdotto dall'approcio simulativo introducendo macchine di test hardware nel processo di verifica. Un'aspetto altrettanto importante in relazione allo sviluppo di un SoC cosiste nell'implementazione di acceleratori hardware ed il loro utilizzo a livello sistema. La mia attività accademica ha curato anche questo aspetto, in particolare è stato sviluppato un circuito digitale per l'accelerazione hardware di una rete neurale convoluzionale. In primo luogo, un originale algoritmo basato su FFA per l'ottimimizzazione della convoluzione mono- e bi- dimensionale è stato sviluppato, successivamente l'implementazione circuitale dell'algoritmo proposto è stata presentata in questo lavoro.

Parole chiave: verifica, sistemi digitali, memorie integrate, ambienti di simulazione, convoluzione.

Contents

Acknowledgements

The author's work has been supported by a PhD scholarship funded by Micron Semiconductor Italia S.R.L.

List of Acronyms and Symbols

The following acronyms and symbols are used throughout the thesis.

NDA	Non Disclosure Agreement
UFS	Universal Flash Storage
EDTL	Expected Data Transfer Length
LBA	Logical Block Address
DUT	Device Under Test
UVM	Universal Verification Methodology
VVM	Verification Methodology Manual
AVM	Advanced Verification Methodology
URM	Universal Reuse Methodology
OVM	Open Verification Methodology
TLM	Transaction Level Modeling
SV	SystemVerilog
API	Application Programming Interface

UVC	Universal Verification Component
EDA	Electronic Design Automation
FSM	Finite State Machine
RTL	Register Transfer Level
SoC	System on Chip
PL	Programmable Logic
PS	Processing System
CPU	Central Processing Unit
GPU	Graphic Processing Unit
VLSI	Very Large Scale Integration
FPGA	Field Programmable Gate Array
IC	Integrated Circuit
ASIC	Application-Specific Integrated Circuit
IP	Intellectual Property
VIP	Verification IP
AXI	Advanced eXtensible Interface
PCB	Printed Circuit Board
PC	Personal Computer
PCIe	Peripheral Component Interconnect express
LUT	Look Up Table

CDC	Clock Domain Crossing
SPI	Serial Peripheral Interface
LFSR	Linerar Feedback Shift Register
CNN	Convolutional Neural Network
FIR	Finite Impulse Response
FFA	Fast FIR Algorithm
FOM	Figure of Merit
AI	Artificial Intelligence
IoT	Internet of Things
BIST	Built In Self Test
CMU	Convolutional Multiplier Unit
MIT	Multiplier Idle Time

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivations

A SoC is a compact and highly integrated electronic device that combines various components of a computer system onto a single microchip. SoC devices are widely used in various electronic devices, ranging from smartphones and tablets to smart appliances, wearable devices, and even embedded systems. They enable the seamless integration of diverse functionalities and help in achieving high-performance computing in compact form factors. The key advantage of SoC devices lies in their ability to integrate multiple complex components onto a single chip, resulting in reduced size, power consumption, and cost compared to traditional systems that employ separate chips for each functionality. SoC devices typically consist of a Central Processing Unit (CPU) or multiple CPUs, storage systems (such as RAM and flash memory), input/output bus interfaces, Field Programmable Gate Array (FPGA), and various other specialized components like Graphic Processing Units (GPUs), audio codecs, and wireless communication modules. Figure 1.1 schematically depicts a typical architecture of a SoC.

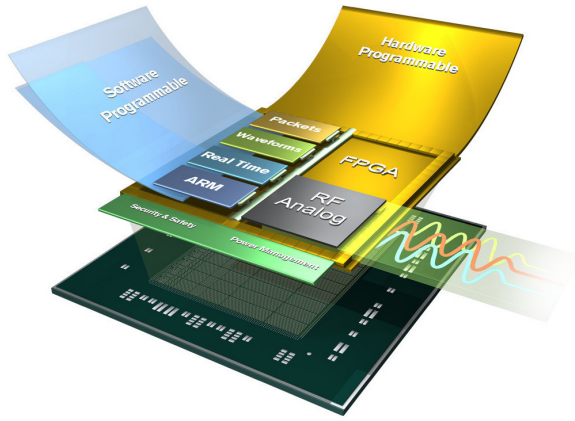


Figure 1.1. Schematic representation* of a system-on-chip device; several components are integrated onto a single chip.
*Source: AMD-Xilinx website.

The design and development of SoC devices require expertise in various engineering disciplines, including microelectronics, digital design, and digital verification. Increasing the complexity of the SoC devices, industrial companies require a robust, efficient methodology for the SoC digital verification process also aiming to reduce the overall time of the product validation. In this regard, an innovative approach in the digital verification field has been explored in this dissertation. Over the last three years, we collaborated with Micron Technology to improve the overall validation flow of crucial storage system products based on the SoC platform. Furthermore, the partnership with one of the largest corporations worldwide encouraged us to introduce innovation in the industrial process of SoC verification dealing with state-of-the-art issues related to the digital verification field. Hence, a robust, reusable, standardized simulation tool is carried out for the Micron SoC storage system devices according to Universal Verification Methodology (UVM), the latest verification methodologies accepted as IEEE standard [1],[2],[3],[4],[5]. The UVM approach enables proactive cooperation among team members as well as great knowledge sharing between companies and simulator tool vendors. Furthermore, the methodology promotes the creation of Verification IP (VIP) that can be reused across different projects and designs. VIPs encapsulate reusable

verification components, such as testbenches and random stimulus generators, allowing engineers to efficiently verify various design blocks and reduce the overall effort needed. For the sake of clarity, the Micron SoC devices must be considered the Device Under Test (DUT) within the simulation environment presented in this dissertation. In addition, this system-on-chip includes highly confidential parts about the commercial product, such as Intellectual Property (IP) and storage systems according to the Universal Flash Storage (UFS) 4.0 standard [6].To this end, an Non Disclosure Agreement (NDA) has been signed with Micron Technology thus only an overview of the entire system can be presented in the following sections. Although the UVM-based techniques for the SoC verification allow to drastically increase the predefined target coverage, the simulation time to deeply validate a complex SoC design could become a limitation for the verification flow. Therefore, within the collaboration with Micron Technology, an innovative *hardware emulation technique* has been carried out, which aims to combine the overall advantages of the UVM approach with an original hardware-in-loop methodology by means of the usage of both FPGA and SoC devices. Thus, the idea behind this technique is to accelerate the test using a hardware platform (SoC or FPGA) where dedicated components are synthesized on the SoC programmable logic and are used to stimulate the DUT by forcing input data or reading DUT output computations. Another significant issue related to the design of SoC devices regards the implementation of hardware accelerators, by means of FPGA or ASIC approach, as well as their integration in SoC and system-level usage. Research activity has been also carried out about this topic, where Micron Technology was not involved thus allowing us to focus on different aspects of the SoC design. In Literature, several applications in the field of electronics and software engineering require that part of the computations must be implemented through dedicated hardware accelerators speeding up the overall throughput of the applications. The mono-dimensional convolution is the prevalent operation used for digital signal processing [7]- [8], whereas the bi-dimensional convolution is widely used in CNN for image recognition and detection, speech recognition, text classification [9],[10],[11],[12].

The multiplication operation is the bottleneck for these kinds of circuits impacting the power dissipation and the speed of the hardware accelera-

tor. This is particularly true for neural networks where several thousands of multiplications are needed to compute a single detection or classification. For this reason, many results have been presented in Literature that address the efficient hardware implementation of convolution. Moreover, it is worth emphasizing that in Literature considerable attention is given to the minimization of the CNN time consumption during both the training and execution phase. Firstly, in this dissertation, a novel FFA algorithm is discussed which is in charge of reducing the number of multipliers required for mono- and bi-dimensional convolution operations. According to the novel FFA algorithm, a hardware data-path architecture, that aims to accelerate the Convolutional Neural Network throughout System on Chip platforms, is also discussed in this work.

1.2 Thesis outline

The remainder of this dissertation is organized as follows. Chapter 2 introduces the UVM, the widely accepted standard for the verification of digital designs. The section will describe the crucial role of this technique in addressing the industrial issues regarding the validation process of complex SoC. Furthermore, the Transaction Level Modeling (TLM) is discussed, this latter enables the verification engineer to use a high level of abstraction during the implementation of simulation systems thus achieving faster simulation and greater flexibility. In addition, the overall UVM architecture is presented thus describing the functionalities of several classes of a UVM-based simulation environment. Finally, a brief discussion about the *UVM phase mechanism* is done thus explaining the simulation steps within this methodology.

Chapter 3 introduces a Micron SoC design according to the NDA signed with the company. This latter must be considered the DUT for the simulation environment developed and described in the following section. Even if there is an NDA and the functionalities of the SoC are strictly reserved, a block-diagram overview is provided in this section aiming to underline the complexity of a state-of-the-art product based on SoC devices. Furthermore, by considering the DUT as a black box, the UVM simulation environment, developed during the collaboration with Micron Technology, is described thus giving emphasis on the several benefits achieved against

a non-standardized simulation environment.

In Chapter 4, the *hardware emulation technique* is presented. This innovative approach allows us to overcome the testing time bottleneck of the software simulations. Thereby, all rights of this technique are reserved, but the general idea behind the implementation is discussed as well as the overall results obtained.

Chapter 5 describes a general FFA algorithm that allows us to reduce the overall number of the multiplications for the convolution operations. We describe the algorithm for the mono-dimensional convolution and then a technique to extend the algorithm to the bi-dimensional one.

Chapter 6 shows a hardware data-path that is based on the proposed FFA algorithm. The architecture is shown as well as the test chip developed.

In Chapter 7, conclusions are finally drawn.

Chapter 2

Universal Verification Methodology

This Chapter aims to give a general overview of the Universal Verification Methodology (UVM) thus highlighting several vital properties and advantages within the digital verification fields. In section 2.1 a brief introduction of the UVM has been discussed underlining the importance of this methodology for companies. Then, the TLM has been treated in section 2.2 focusing on the *abstraction level* in the verification flow. Furthermore, many technical details about the components and their configuration within a UVM infrastructure have been presented in subparagraph 2.3 and 2.4. Finally, in section 2.5 the simulation time management, introduced by the standard [2], as well as its usage is also presented.

2.1 Introduction

Over the last decades, SoC devices have included more powerful components and peripherals enhancing the number of gates synthesized into a single chip by up to 200 million. To this end, industries require highly qualified verification teams that address the challenge of complex digital design verification. The Electronic Design Automation (EDA) providers tried to propose several customized verification libraries which aim at collecting all the best practices for digital verification issues also giving examples about the simulation tools usage.

As a consequence [5], it is worth mentioning that:

- In 2002 Synopsis Inc. announced the Verification Methodology Manual (VVM) based on SystemVerilog (SV) standard.
- In 2006 Mentor Graphics introduced the Advanced Verification Methodology (AVM), also based on SV standard, it was the first open-source verification solution.
- In 2007 Cadence Design Systems Inc. developed the Universal Reuse Methodology (URM) which included solutions for verification issues such as testbench configuration mechanism and class automation.

Although the mentioned verification methodologies enabled to spread of the best practices for digital verification within the industries, each methodology was supposed to be used only with one simulator solution. Thus, the digital verification team's interoperability was not completely allowed. To this end, in 2008 Cadence, in partnership with Mentor, made the first standardized verification library as Open Verification Methodology (OVM). The innovation in this proposal is that OVM was the first multi-vendor solution tested over more than a single-vendor simulator. As a consequence, the importance of a unified methodology for the validation process of the 2000 million gate-equivalent SoC designs was clarified; thus, with the cooperation of the semiconductor companies and the major EDA vendors (Cadence, Mentor, and Synopsis), in 2010 Accellera Initiative Systems announced the Universal Verification Methodology (UVM) standard [2]-[3] which was based on the OVM 2.1.1 version. The UVM libraries were tested and compiled by all the simulator tools thus improving the collaboration between companies and verification engineers who use different tool solutions. As an outcome, UVM provides a systematic and reusable approach to verify complex designs, ensuring their correctness and reliability. This technique has emerged as a powerful methodology due to its effectiveness in tackling the challenges posed by ever-increasing design complexity and shorter time-to-market demands. Finally, in 2017 the Universal Verification Methodology (UVM) was accepted as an IEEE standard [1]. In addition, it is worth highlighting that [1] defines the worldwide guidelines to build a flexible simulation tool addressing several abstraction levels of the SoC under test; however, only the base classes of the typical UVM

architecture [1] are provided within the standard SV packages. During the development of a SoC device, the verification engineers have the task of:

- importing UVM base classes and the related packages with the simulator tool.
- implementing UVM architecture and custom components as the DUT needs. Thus, the overall components developed must be extended by the base classes of the UVM packages according to the general guideline provided [2] and the DUT requirements.
- developing the scripts for the simulation environment management. The verification teams must deal with test repository creation and maintenance, directory organization of the overall simulation tool as well as the development of customized scripts that combine the hints provided by the standard [1] and the EDA vendor.

It must be clear that even though the Universal Verification Methodology gives a base infrastructure for simulation environments of digital designs through open source packages and several guidelines, the verification engineers play a vital part within the industrial process as well as the SoC device development. Starting from the functional specification [2] of the design to be implemented, both verification and designer teams lead with the implementation part of the project. From the verification point of view, the device under test could be considered a black box with several interfaces that meet different communication protocols. Consequently, the fulfillment of the simulation environment could start by developing all the components needed to manage the DUT ports as well as the protocol requirements included in the functional documentation. Hence, when the designers produce an Register Transfer Level (RTL) version of the SoC device, the DUT code must be integrated into the simulation tool and the validation of the product starts effectively. As a consequence, the overall time regarding the design development is drastically reduced by properly balancing the effort between verification and design engineers.

[1] Refer to section 2.3 for more details

[2] It is a document that collects all the requirements, technical details, and interface characteristics of a system that must be developed. On the basis of the functional specification, the implementation process effectively starts generating the RTL code of the design as well as the simulation environment for the validation step

2.2 Transaction Level Modeling

2.2.1 Overview

The Transaction Level Modeling (TLM)is a methodology used in the Universal Verification Methodology to model and verify the behavior of hardware designs at a higher level of abstraction. The TLM focuses on the communication and interaction between various components or modules in a system. While a digital device port could be managed within a simulation component through a cycle-accurate approach by means of the definition of the bus timing as well as the description of the behavior of the signals for each clock cycle, experience in the digital verification field has shown that engineers tend to prefer high-level models of both simulation components development and DUT signals management overcoming the need to look at the low-level activity of the DUT interfaces. This approach enables the improvement of the testbench productivity as well as the efforts needed to build a reusable simulation architecture. The objective of a transaction-level model is to represent the functionality of a design faithfully, in a way that is easy to write and simulate. The transaction-level model is designed to help the verification, and indirectly, the implementation of the SoC design. In UVM, a subset of transaction-level communication interfaces and channels have been included thus allowing engineers to interconnect UVM-based objects and DUT ports easily, making the simulation components easier to debug and understand. To use this approach, it is worth pointing out that a thin layer within the UVM classes [3] must be inserted, the latter has the task of translating the transaction-level communication into signal-level activity over the DUT ports. For the sake of clarity, the transaction [4] is an object defined by the user that contains all the information, methods, and constraints required to model an essential communication between two components within a simulation tool. A code example of a transaction is shown in Figure 2.1; therefore, in a high-level approach, this object may be used to manage a general bus protocol such as AXI Lite or AXI APB [5].

[3] These objects are called *drivers* and *monitor*, refer to section 2.3 for more details
[4] In the UVM, this class object extends the uvm_sequence_item
[5] Refer to [13] for additional information about the different protocol versions

```
class example_trans extends uvm_sequence_item;
rand int data;
rand int addr;
rand enum {WRITE,READ} kind;
constraint c1 {addr < 16h'1000}
...
endclass
```

Figure 2.1. Code example of a transaction.

2.2.2 TLM Ports and Exports

The UVM provides a TLM Application Programming Interface (API) based on the TLM 1.0 standard; furthermore, transaction-level interfaces encompass a collection of methods that employ transaction objects as parameters. A TLM *port* class establishes the array of methods designed for a specific connection. Conversely, a TLM *export* provides the implementation for these methods. *Ports* and *exports* objects are connected to each other by means of *connect()* method calls whenever a verification environment is constructed. Figure 2.2 depicts a schematic overview of the classes involved where the producer is a generic verification component that creates a transaction meanwhile the consumer object is in charge of elaborating the transaction coming from the producer.

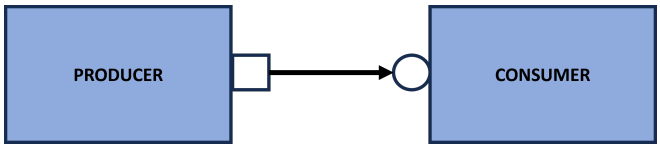


Figure 2.2. TLM Producer and Consumer block-diagram. The square box on the producer indicates a port and the circle on the consumer indicates the export.

Furthermore, the basic transaction level operation is based on the *put* method. This latter allows one component to insert a transaction into another. Figure 2.4 describes one possible implementation of a producer class, in this example, the object creates several transactions adding an optional randomization layer, and then sends the information within the

example trans to the consumer class by calling the **put** method. Moreover, a consumer class example is presented in Figure 2.5. Here, the implementation of the **put** method is explored; in addition, it is worth pointing out that this code shows a generic fulfillment of the level of abstraction conversion from transaction to pin-level stimulus[6]. Another transaction level operation is based on **get** method, a block diagram representation is shown in Figure 2.3. In this case, the consumer requests the transaction from the producer through the *get* method.

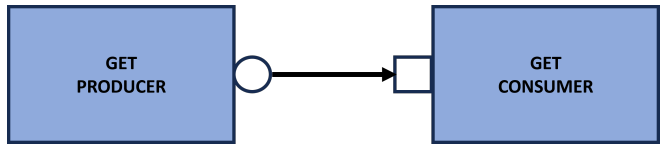


Figure 2.3. TLM Producer and Consumer block-diagram with get method. The square box on the consumer indicates a port and the circle on the producer indicates the export.

Conversely, the *get producer* creates the transaction and gives the implementation of the **get** method. The *get consumer* makes a transaction request through **get** method call, then the class object is in charge of elaborating the information inside the transaction object. Although the TLM methods described above allow the engineers to implement a TLM-based communication between two components, the TLM *port/export* interfaces are characterized by different drawbacks:

- Both put and get ports require that the corresponding export interfaces are connected before the simulation starts, otherwise an error occurs.
- The implementation of the put/get method might block the execution until the method completes. Thus, the caller must manage this condition.
- Multiple ports and export connections are not allowed.

However, within a simulation environment, some passive components such as *monitors* could be connected with more than one component as well

[6]This is a typical example of the *drivers class* behavior within a UVM Architecture

as left unconnected. As an example, passive components may be used to collect transactions transferring them to any objects involved in post-processing evaluation. In UVM, in order to address this kind of situation, the *analysis port and export* were introduced. In particular, the most valuable benefits of the analysis ports with respect to the standard TLM ports consist of:

- the ability to leave the analysis ports disconnected within the verification components, so no error occurs.
- The ability to connect one port with multiple analysis exports.

Moreover, an analysis port provides a single function void *write()*, which the implementation is left to the analysis export. When one component calls the write method of an analysis port with a transaction class, the analysis port calls the write method of every connected analysis port thus sending the transaction object to several components with a single line code. It is worth mentioning that because the *write* method is a function, It is ensured that the *write* method returns without blocking the overall code execution.

```
class producer extends uvm_component;
uvm_blocking_put_port #(simple_trans) put_port;
function new( string name, uvm_component parent);
put_port = new('put_port', this);
...
endfunction
virtual task run();
    example_trans t;
    for(int i = 0; i < N; i++) begin
        // Generate t.
        put_port.put(t);
        end
endtask
endclass

```

Figure 2.4. Code example of producer class with **put method call**

```
class consumer extends uvm_component;
uvm_blocking_put_imp #(example_trans, consumer) put_export;
virtual interface my_if vif;
...
task put(example_trans t);
    case(t.kind)
        READ: begin // Do read
            vif.araddr  = t.addr;
            vif.arvalid = 1;
            ...
            end
        WRITE: begin // Do write
            vif.wdata  = t.data;
            vif.wvalid = 1;
            ...
            end
    endcase
endtask
endclass

```

Figure 2.5. Code example of consumer class with **put method implementation.**

2.3 UVM Architecture

This section provides an overview of a typical architecture of a UVM testbench adding details for each class involved in Figure 2.6.

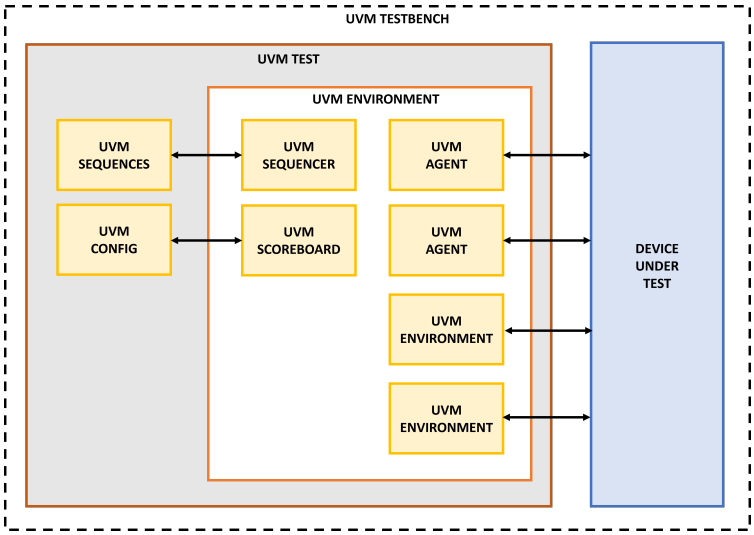


Figure 2.6. A typical testbench architecture based on UVM guidelines.

2.3.1 UVM Testbench

The *UVM testbench* is a component that collects the UVM test class and the Device Under Test (DUT) thus managing:

- The connection between the test class and the DUT via SystemVerilog (SV) interfaces.
- The configuration of the classes within the UVM packages.
- The generation of the clock signals of the DUT.

2.3.2 UVM Test

The *UVM Test* is the top-level Universal Verification Component (UVC) within a UVM-based testbench. Thus, it is in charge of :

- Instantiating top-level environment.
- Setting all the class variables by means of the configuration database, the UVM factory and/or the usage of a custom configuration class.
- Invoking a *default* sequence for each agent within the system thus generating a stimulus traffic for the DUT validation.

Conventionally, the best practices in the digital verification field recommend the usage of a base UVM test class which instantiates the top environment and configures the global variables. Then, all the UVM tests extend the base class thus:

- introducing additional configuration fields.
- highlighting the proper sequence class to customize the behavior of the stimulus generation.

2.3.3 UVM Environment

The *UVM Environment* generally includes one or more UVM Agents for the DUT interfaces management as well as additional objects such as *UVM scoreborards* to verify the behavior correctness of the DUT, and UVM subscriber to collect transactions and perform coverage analysis. More specifically, the *UVM scoreborards* are the UVCs within an environment that contains a *reference model* named *predictor*, this one produces the DUT expected transactions. Thus, the scoreboard classes receive transactions through UVM Agent analysis ports and compare the expected outputs with the actual outputs. Overall, the *UVM Environment* has the task of instantiating the agents, scoreboards, and subscribers and connecting the corresponding analysis ports and exports.

2.3.4 UVM Agent

The *UVM Agent* is a verification component that contains all the UVM classes useful for managing a DUT interface. Figure 2.7 depicts a block-diagram overview of a *UVM Agent*.

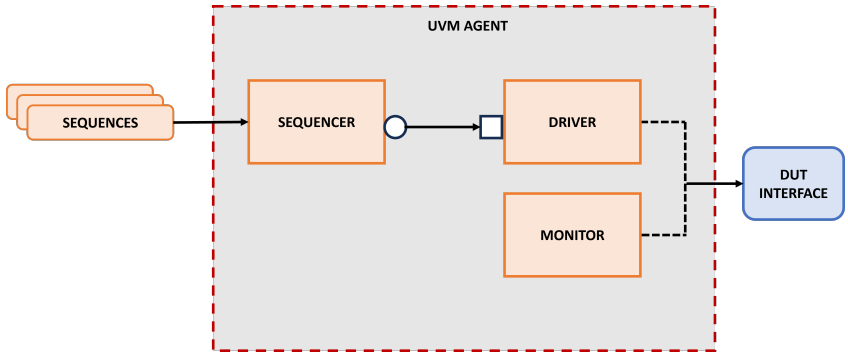


Figure 2.7. Overview of a UVM Agent class.

Usually, the agent class is characterized by:

- **Sequencer**
- **Driver**
- **Monitor**

More in detail, the *UVM Sequencer* can be compared to an *arbiter*, It is in charge of controlling the flow of UVM sequence items transactions generated by one or more sequence classes. Furthermore, the *UVM Sequence* is an object that creates one or more transactions and introduces the randomization according to the UVM thus managing the overall stimulus generation within a testbench. The *UVM Driver* receives individual sequence items from the sequencer and drives them on the DUT interface. Thus, It converts the transaction level stimulus into pin level stimulus. The communication with the sequencer is based on the TLM analysis port and export connection. Finally, the *UVM Monitor* is a passive entity that samples the DUT signals without driving them. It collects coverage information, performs checking operations, and converts the pin-level activity to the transactions. In addition, the monitor classes provide an analysis port that can be connected to several components[7] thus sending the collected transaction in the entire simulation environment.

[7]Generally, the monitor classes are connected to a scoreboard or a subscriber classes

2.4 UVM Configuration Mechanism

The Universal Verification Methodology provides a flexible and hierarchical configuration mechanism that allows engineers to customize the behavior of classes and verification components. The configuration mechanism in UVM enables run-time configuration, making it possible to adapt the testbench and its components dynamically based on specific requirements. In order to develop a reusable architecture, a good practice in the digital verification field consists of the usage of a configuration object that collects the overall configuration parameters within a simulation environment. For instance, an example configuration object may provide information about the number of slave and/or master agents for a specific environment, and, for each agent, the object could specify if the coverage computation must be performed or not. An example of a configuration class is presented in Figure 2.8.

```
1 class axi_config extends uvm_object;
2 rand int num_slaves;
3 rand int num_masters;
4 rand int coverage_enable;
5 . . .
6 rand bit[31:0] base_address;
7 endclass
```

Figure 2.8. Code example of configuration class for a specific bus.

It must be considered that each UVC in the UVM infrastructure can have its own configuration object, allowing for individual customization. Additionally, the UVM provides a hierarchical configuration database that stores configuration objects and their values; therefore, the objects could be stored and retrieved from the database using a hierarchical path-based naming scheme. The standard [1] introduces also configuration APIs to manage the database[8], allowing the user to set/get configuration objects as well as global variables inside an environment. Finally, the UVM libraries also provide an advanced implementation of the factory method pattern,

[8] uvm_config_db#(T)::set/get(uvm_component context, string inst_name, string field_name, T value) see IEEE standard for more details [1]

widely used in software designs. The UVM factory is a mechanism that allows the engineers to replace a pre-existing class object with any of its inherited child objects at run-time thus improving the overall reusability of the class infrastructure. Figure 2.9 provides a code example of both the UVM configuration and factory mechanism within a test class implementation.

```
1  class base_test extends uvm_test;
2     `uvm_component_utils(base_test)
3     my_top_env top_env;
4     axi_config cfg;
5
6  function new(string name = "base_test", uvm_component parent=
       null);
7      super.new(name,parent);
8  endfunction : new
9
10 virtual function void build_phase(uvm_phase phase);
11     super.build_phase(phase);
12     cfg = axi_config::type_id::create("cfg", this);
13     cfg.num_slaves  = 10;
14     cfg.num_masters = 5;
15     cfg.coverage_enable = 1;
16     cfg.base_address = 16'h40000;
17     // UVM configuration and factory management
18     uvm_config_db#(axi_config)::set(this,"top_env.*", "
       axi_cfg", cfg);
19     factory.set_type_override_by_type(base_agent::get_type(),
       child_agent::get_type());
20     // Create the tb
21     top_env = my_top_env::type_id::create("top_env", this);
22 endfunction : build_phase
23
24 endclass : base_test
```

Figure 2.9. Code example of a UVM test class with configuration ad factory management. In the build phase implementation, a configuration object is created and associated with the configuration class pointer of the top-level environment. Moreover, a factory override example is reported, where all the instances of a base agent are replaced with a child agent class in run-time.

2.5 UVM Phases Mechanism

The UVM class library provides a set of standard built-in simulation phase methods that allow the synchronization of the environments. The phased-based mechanism in UVM facilitates modularity, reusability, and scalability within the verification environments. It allows engineers to encapsulate specific functionality within each phase and provides a clear structure for managing the verification process. Additionally, the phased-based approach enables parallelism and concurrent execution of different tasks, improving overall efficiency and reducing verification time. A schematic representation of the simulation phase methods is depicted in Figure 2.10.

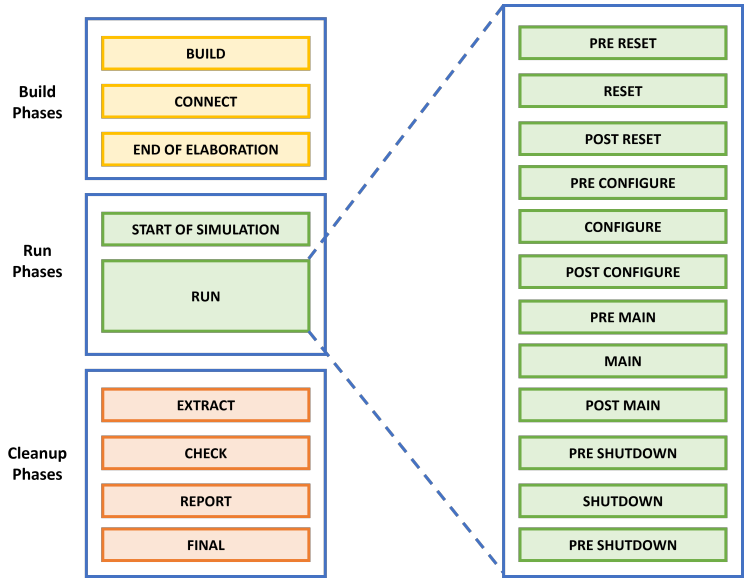


Figure 2.10. UVM phase methods block-diagram. There are three macro phases that include several low-level simulation steps.

Moreover, the initial steps of a simulation, named *build phases*, can be summarized as:

- **Build:**This is the first phase which is called for all UVM components. This method is used to create and configure the component's

child components. As an example, an agent class creates the driver, sequencer, and monitor objects during this phase of the simulation. To this end, this method is fundamental because is responsible for the generation of the entire UVM infrastructure.

- **Connect:**It is executed after the build phase, this method is used to make TLM connections, assign pointer references, and set/get virtual interfaces. Regarding the previous example of the agent class, within this phase, the analysis port and export of the sequencer and driver object must be performed in the connect phase allowing the proper link among the classes.

- **End of elaboration:**This phase ensures that all connections and references are properly set up from the connect phase. It is usually hopeful to execute the print() method of top-environment to display the complete testbench architecture.

As a consequence, the methods described above are used to properly configure the entire class infrastructure within a UVM-based simulation environment. It is worth underlining that a generic SystemVerilog (SV) solution to manage several simulation time steps within a simulation is hard to implement and also complicated to use. An important advantage of the UVM approach is related to the phase mechanism management, which is completely in charge of the UVM base class libraries. To develop a new verification component, the engineers need only to extend a *uvm_ component* and, automatically, all the methods for the phase management are imported from the base class. To this end, the user focuses on the development of the test and its functionalities and less on the class management issues of a simulation environment.

While the *build phases* are executed at the zero time of the simulation, the effective validation of the DUT is performed in the *run phases* that are divided as:

- **Start of simulation:** This phase plays a crucial role in a UVM simulation, It is responsible for controlling the initial state and configuration of the verification environment and implementing methods to avoid zero-time dependencies before the main simulation activity starts.

- **Run:**Implemented as a task, It defines the implementation of a component's primary run time functionality. As Figure 2.10 depicts, the run phase is also split into several sub-phases such as reset, configuration, and main phases. This allows the user to build reusable test classes, in which the maintenance is enhanced by using different UVM sequences for each phase. For instance, suppose that a test, where the post-configuration phase is not implemented, is written to replicate a particular scenario during the validation plan of an SoC design. Then, the same test may be extended more than one time thus invoking different sequences in the post-configuration phase. As a consequence, the same scenario can be verified with different post-configuration stimuli increasing the overall coverage of the test.

Finally, the UVM library provides additionally *cleanup phases* executed before the end of the simulation:

- **Extract:** This phase can be used to extract the simulation results at the end of simulations, but prior to checking in the next phase. It can be used to collect assertion error counts and obtain coverage information.
- **Check:** It is used to validate data coming from the extract phase and to determine the overall simulation outcome.
- **Report:** It is used to output results to files and/or the screen.
- **Final:** End of simulation.

Chapter 3

Device Under Test and Simulation Environment

This chapter aims to give a general overview of a complex digital system, developed by Micron Technology, which has been tested by adopting advanced verification techniques during the PhD program. First of all, the Micron state-of-the-art system is introduced, which includes Soc devices, Application-Specific Integrated Circuit (ASIC) components, and the latest storage systems according to the Universal Flash Storage (UFS) 4.0 standard. Consequently, the UVM-based simulation environment for the Micron solution is presented thus highlighting the performance as well as the overall benefits achieved by the proposed verification tool.

3.1 Micron System

Over the last three years, a successful collaboration with the Micron Technology hardware validation tool team has been carried out allowing us to deal with state-of-the-art issues related to the digital verification flow within the industries. While this section introduces a comprehensive description of the entire Micron system discussing all the products integrated, protocols involved, and system-level usage of the Micron product, it is worth pointing out that the system introduced in this work incorporates ASIC components as well as the latest commercial products within the Micron storage system solutions. To this end, documentation and

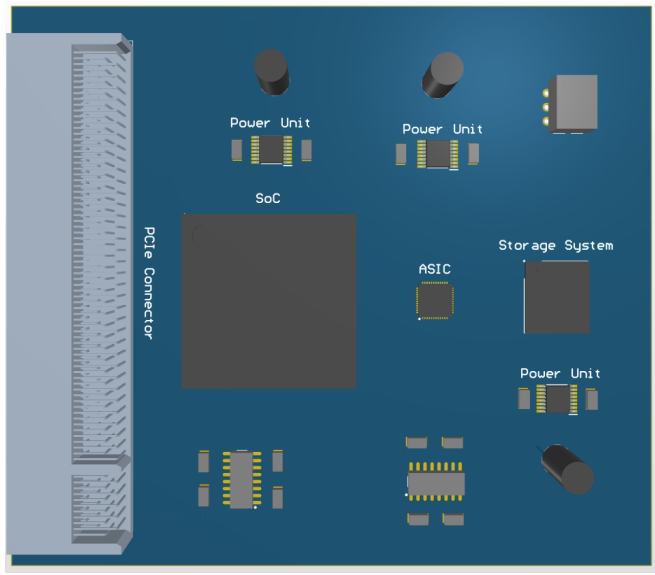


Figure 3.1. PCB of the Micron system.

rights of the system are highly confidential thus a Non Disclosure Agreement (NDA) has been signed between the University of Naples Federico II and Micron Technology. As a consequence, several details about the system under test as well as part of the verification techniques carried out thanks to the collaboration with Micron cannot be presented or discussed in this dissertation due to the NDA previously mentioned. The aim of this section consists of:

- introducing the complexity of the Micron solution.
- enumerating the components involved and their connections.
- explaining how this system is able to communicate with a general-purpose PC.

Figure 3.1 depicts a possible layout [1] of the Micron system which includes the main Integrated Circuit s (ICs) connected to each other. More in detail, the presented board is characterized by the following:

[1] The effective layout of the board cannot be shared due to the NDA

- top-level SoC device.
- ASIC chip.
- Micron Storage System based on Universal Flash Storage (UFS) 4.0 standard.
- Power Units to generate and dynamically manage the voltage of the storage system.
- Peripheral Component Interconnect express (PCIe) connector in order to enable communication between the board and an external Personal Computer (PC).

According to the previous sections of this dissertation, generally, an SoC device incorporates one or more Central Processing Units (CPUs) and one FPGA into a single chip, this is particularly true for the top-level SoC device integrated within the Micron system. Indeed, the SoC is composed of a Processing System (PS) and a Programmable Logic (PL). The PS part includes several ARM-based CPUs meanwhile the PL is characterized by a FPGA device that meets the overall requirements of the Micron system. Therefore, the application requires a high-performance PL with a huge density of logic cells [2] due to the complexity of the Micron IPs, which will be discussed in the following section. More specifically, the Micron IPs developed on the PL side of the SoC are in charge of configuring and communicating with the storage system based on the UFS protocol meanwhile the connection between the IPs on the FPGA and the memory package is performed by means of an ASIC chip. This latter has the task of managing the storage interface of the UFS card through complex Finite State Machines (FSMs) architectures, according to the protocol requirements. It is worth highlighting that the UFS 4.0 storage system in Figure 3.1 was announced on June 2023 by Micron Technology, the device is the *best-in-class* product on the market ensuring the fastest write/read operation with a considerable power consumption reduction[3]. Hence, the

[2]Within an FPGA device, the logic cell is the smallest hardware component that is possible to configure and connect to each other. Typically, a logic cell is composed of a Look Up Table (LUT) and one or more flip-flops

[3]The device achieves up to 25 % of power reduction with respect to the previous version based on the UFS 3.1 protocol

verification methodologies, described in detail in the following sections, are strictly related to the development and validation of one key product of the Micron companies. According to Figure 3.1, there are three power units controlled by Serial Peripheral Interface (SPI) protocol; these circuits are routed to the PL of the SoC via SPI interfaces. Thus, dedicated IPs have been developed to properly control the overall power units and, consequently, the supply voltage of the memory card. Overall, the SoC is also routed to a PCIe connector which allows the board to be inserted into a PCIe bus within a general-purpose PC desktop. More in detail, the Micron team involved in the collaboration within the PhD program, is in charge of developing the Micron IPs for the PL as well as implementing the firmware for the PS of the SoC. Moreover, the Processing System configures the entire system and the Micron IPs making the PL side an **end-point device** for the PCIe protocol communication. On the other hand, the external PC, linked to the system above described, presents a linux-based operating system that includes all the drivers needed for the PCIe bus management. Hence, the Micron board is directly controlled by using an application layer of the operating system; this latter is characterized by several C/C++ routines useful to properly manage the PCIe low-level driver and, consequently, the entire sub-system over the PCIe bus. Furthermore, the Micron software design tool team is in charge of developing the application layer of the Linux-based operating system and implementing a subset of APIs that allow the users to easily configure and control the SoC device and the storage system. The overall system above described is the result of a key project in the Micron business and requires multidisciplinary team cooperation where software architects, digital designers, verification engineers, and process engineers are continuously encouraged to collaborate to reduce the time-to-market of the product as well as to improve the overall quality of the solution. As an outcome, the work carried out regarding this ambitious project allowed us to collaborate with the software design tool team as well as the ASIC team ensuring international visibility of the PhD work inside the Micron company. More in detail, we worked on the verification and validation of the overall IPs synthesized in the PL-side of the SoC. In this regard, we proposed a robust and flexible simulation tool for the software verification, according to the UVM standard, as well as an innovative solution for the on-board valida-

tion of the entire system.

3.2 Simulation Environment

While in the previous section, several hardware details of the Micron solution have been introduced giving a global view of the devices involved and their connections, this part of the dissertation aims to highlight the overall IPs and digital circuits synthesized in the PL of the SoC device as well as the IPs connection with the external components with respect to the SoC device. Before describing many aspects of the Micron architecture, It is worth mentioning how the SoC solutions allow the connection and the data exchange between the ARM processor and the IPs within the Programmable Logic. Over the last decades, the AMBA Advanced eXtensible Interface (AXI) protocol [13] has been incorporated within several SoC designs; the standard, developed by ARM, serves as a high-performance and efficient interface for connecting different IPs thus enabling seamless communication and data transfer between IP cores, processors, and memory units. To this end, the Zynq processor is able to manage and configure the entire system within the programmable logic via AXI bus; conventionally, an AXI Interconnect IP is used for this purpose. Moreover, the interconnect is a configurable digital circuit provided by the SoC device vendor, this circuit connects the PS with the programmable logic sub-system. In addition, it includes a cross-bar circuit in order to support the communication between several master and slave IPs with AXI interfaces over the bus as well as a dedicated logic for the Clock Domain Crossing (CDC). According to this general discussion about the SoC devices, Figure 3.2 depicts a block diagram of the overall Micron solution giving additional information about the IPs architecture and connections that will represent the Device Under Test (DUT) for the simulation tool described in this dissertation. For the sake of clarity, only the key IP cores, essential to properly describe the following simulation architecture, have been reported in this diagram. The Programmable Logic system is characterized by:

- Main IP: a digital circuit that is in charge of elaborating all the requests of the operating system toward the storage device. The IP provides two AXI Full interfaces, the first one implements the slave version of the protocol meanwhile the second one acts as the master

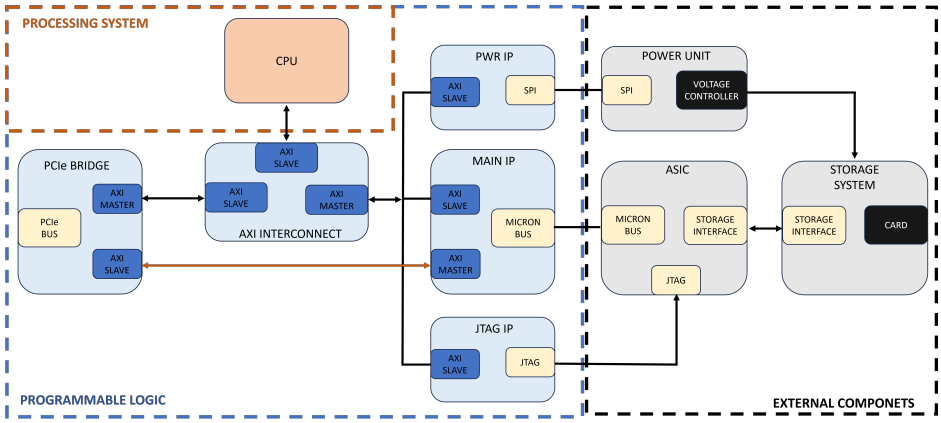


Figure 3.2. Block diagram of the Micron system.

within the AXI transaction generation. The latter interfaces are directly connected to the PCIe Bridge IP, also provided by the SoC vendor company, which has the task of translating the AXI transactions into PCIe packets and sending them over the PCIe bus. The PCIe packets are processed by the external PC by using dedicated PCIe drivers. Of course, the bridge allows the opposite flow, from PCIe to the SoC devices, converting the PCIe transactions into AXI payload, that is appropriately processed by the Micron IP cores. Finally, the Main IP analyzes the payload from the operating system, such as a list of UFS commands to be processed or data to be transferred to the storage system; then, the circuit inserts additional parameters and forwards the payload over a Micron Bus to the ASIC component according to the UFS timing requirements. The architecture synthesized into the ASIC chip is composed of several FSMs which process the command payload as well as the data traffic according to the UFS protocol and, consequently, convey all the required information to the storage system.

- JTAG IP: a circuit that allows the PS and the external PC to configure the ASIC digital system through the JTAG bus. To this end, the IP cores, the Processing System, and the external PC are able to dynamically change the signal processing performed by the ASIC

circuits during the start-up phase or the activity phase of the storage system.

- Power Management IP: this circuit controls the Power Unit components via SPI protocol. More specifically, the IP is able to configure the devices to generate different levels of voltage required by the UFS memory card. In addition, the voltage ramp could be dynamically rearranged depending on the UFS traffic thus achieving the best performance of the memory card while the UFS commands execution.

- Vendor IP: the AXI Interconnect allows the interoperability between the Micron IPs, the ARM CPU as well as the PCIe Bridge. This latter provides an optimal solution to connect the IP cores, which conventionally provide AXI interfaces, to the PCIe bus implementing:

  - the protocol conversion *AXI-to-PCIe* and vice-versa through dedicated hardware.
  - the physical layer to manage the PCIe bus.

As for the simulation environment, it is worth enumerating the IP cores and digital circuits which must be validated defining the architecture of the Device Under Test (DUT) within the simulation tool described in this dissertation. Considering the block diagram reported in Figure 3.2, although the vendor IPs play a crucial part within the Micron system, the AXI Interconnect and the PCIe Bridge IPs are not included in the DUT components for this simulation tool. The focus of the verification environment is on the exhaustive testing of the overall IPs developed by the Micron hardware validation tool team and less on evaluating IPs provided by other companies and included in the SoC design; in addition, both AXI interconnect and PCIe Bridge are deeply tested by dedicated verification teams, which provide accurate documentation, user's guide and support about the Vendor IPs architectures and their usage within a complex design thus the simulation of the mentioned IPs are out of the scope of the PhD work. Therefore, the architecture of the digital designs validated through the simulation tool described in this dissertation is shown in Figure 3.3. Within the IP cores and the ASIC circuit block diagram, the

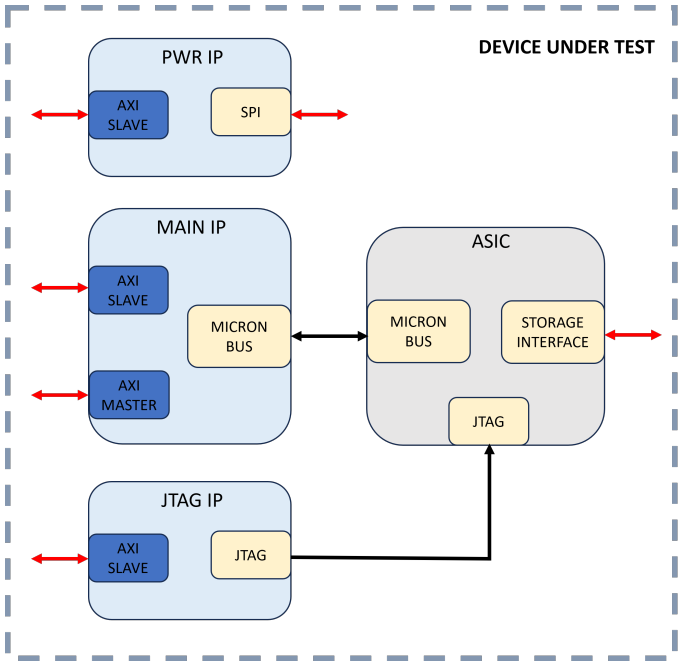


Figure 3.3. Block diagram of the Device Under Test for the proposed simulation environment.

interfaces that must be controlled to stimulate the system have been highlighted with red arrows. The first step of the verification tool development consists of identifying all the DUT interfaces and the related protocol for each of them; thus, the verification engineer is able to establish the architecture of the simulation environment including all the UVM agents needed to appropriately manage the input/output interfaces of the system under test. As for the Micron IPs, the circuits are characterized by AXI interfaces which implement both full and lite versions of the protocol; furthermore, the Power Management IP provides an additional SPI interface that must be controlled by a dedicated UVM agent. The ASIC chip is in charge of managing the storage interface according to the UFS protocol, this bus represents a fundamental part of the DUT thus the validation requires a robust environment to achieve a high level of verification coverage exploring an extensive number of scenarios allowed by the UFS standard.

Furthermore, within the proposed verification environment, the Micron bus is directly tested by simulating the data transfer between the Main IP and the ASIC component, as well as for the JTAG bus, where there is a direct connection between the Micron JTAG IP and the interface of the ASIC chip. It is worth mentioning that the validation of the JTAG IP has been carried out by means of a dedicated UVM simulation environment developed during the PhD work and, consequently, the Micron IP has been integrated within the verification tool for the DUT depicted in 3.3, where the system level functionalities of the JTAG IP have been evaluated[4]. The architecture of the proposed simulation environment fully compliant with the UVM standard is reported in Figure 3.4. From the simulation point of view, we need to emulate the behavior of the firmware execution of the PS side as well as the application layer usage of the external PC. In particular, the UFS protocol is based on several data structures that contain all the information needed by a specific command to be processed. Within the Micron system, the UFS commands toward the storage system are created by the software application layer and, consequently, transmitted to the SoC subsystem and storage card via PCIe bus. To this end, the operating system reserves a protected area within its system memory in order to store the UFS commands and manage the payload to/from the UFS card. Conversely, the UFS memory has the task of executing the command issued throughout the IP cores by the operating system; then, the memory creates a response data structure in order to notify the overall status of the commands executed. The latter information is collected by dedicated circuits in the ASIC component, processed by the Main IP and, finally, sent to the external PC over PCIe bus. It becomes clear that the data exchange mechanism above described between the PC and the Micron system must be introduced in the simulation environment. More specifically, the verification tool needs:

- a model of the system memory where the user is able to create his own UFS data structure, check the integrity of the response fields from the UFS device, and control the payload within the data transfers.
- a software implementation of the AXI Interconnect and PCIe bridge behavior to properly manage the Main IP interfaces, which is in

[4]The simulation environment for the JTAG IP is not discussed in this dissertation

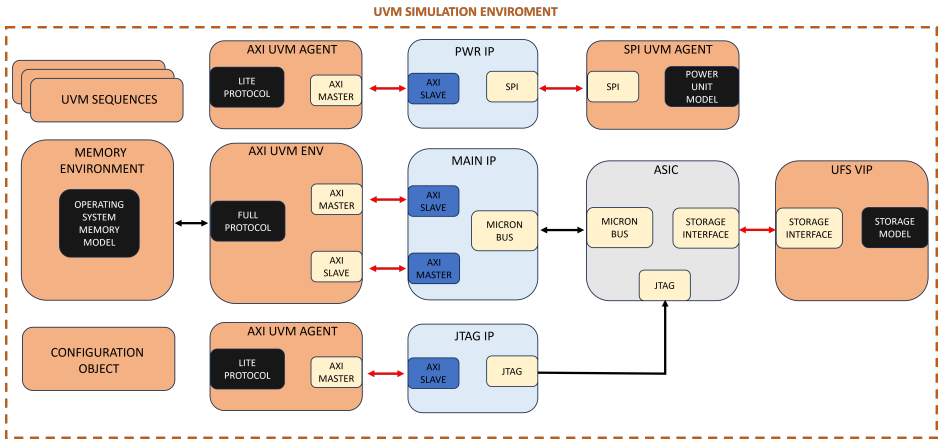


Figure 3.4. Block diagram of the proposed UVM simulation environment for the Micron system.

charge of accessing the system memory of the operating system thus processing the overall UFS commands and controlling both the ASIC component and the storage card.

As an outcome, a UVM memory environment class has been developed for this purpose, the object emulates a 64-bit system memory of a generic operating system that is managed by means of dedicated UVM sequences to configure memory locations or perform all the check operations properly; the latter ensure the correctness of the DUT computations with respect to the UFS standard during a UVM test execution. In addition, the system memory model must be accessible via AXI protocol by the Main IP ports; thus, an AXI environment class has been implemented which includes two different UVM agent classes to control and evaluate the UFS protocol over the Main IP interfaces. This class deploys the full version of the AXI protocol, the first agent acts as the master for communication over the AXI bus and it is useful to configure the IP core and the inner FSMs and circuits, the second one acts as the slave for the communication allowing the Main IP to perform read or write operation toward the system memory model. It is a reactive agent where a monitor class recognizes the AXI transactions issued by the DUT and, subsequently, generates a trigger event that is used by a dedicated UVM sequence. Here, the sequence randomizes a

transaction with a variable delay attribute which is processed by the driver class to introduce a random latency for system memory access. This AXI delay model has been introduced to properly emulate the hardware latency of both AXI Interconnect and PCIe bus of the Micron solution in Figure 3.2. Furthermore, the aim of the proposed simulation tool is to accurately evaluate the UFS protocol over the DUT interfaces, while the creation of the overall data structures is in charge of both memory and AXI full environment class, the UVM architecture requires an additional verification component that acts as UFS device within the system. Indeed, the Main IP receives the protocol commands through the AXI interfaces and starts with their elaboration; then, the UFS data structure, as well as the command's payload, is transferred over a Micron proprietary bus to the ASIC digital circuits. The latter thanks to several FSMs is able to control the storage interface according to the UFS protocol requirements and timing specifications. Hence, to sustain the communication of the implemented model of the operating system's memory and the Micron IPs, a state-of-the-art UFS Verification IP (VIP) has been introduced in the simulation architecture. The UFS model, adopted for this purpose, is provided by Cadence Design System, It allows us to establish the correctness of the overall elaborations performed by the DUT according to the UFS standard. The Cadence VIP also provides an accurate and configurable storage model as well as a dedicated reactive agent in charge of managing the storage interface connected to the ASIC circuit as reported in Figure 3.4. The UFS model is fully compliant with the UVM standard; therefore, the integration of the Cadence VIP has been carried out by analyzing the comprehensive Cadence documentation as well as a limited number of examples of the VIP integration and usage. Then, all the classes needed for our application within the UVM class library offered by Cadence company have been selected and properly integrated into the top-level class of the proposed verification tool. It is worth emphasizing that even though the inclusion of the UFS VIP requires a huge effort, the adoption of the UVM standard allows us to complete the import procedure without the need for the direct support of a specialized Cadence team thus reducing the overall time and the cost for the Micron company. This is a practical example of the advantages offered by a worldwide standard for the verification process, which speeds up the collaboration between verification engineers

of different companies by adopting the UVM techniques. As reported in Figure 3.3, the device under test, evaluated in this dissertation, is characterized by several IPs cores, and each of them provides an AXI interface for the connection with the processing system of the SoC device and the PCIe bridge IP. Furthermore, the Power Management IP and the JTAG IP implement the lite version of the AXI protocol; thus, a UVM agent to manage the AXI lite bus has been developed allowing the user to properly configure and control all the IPs within the DUT architecture. While the JTAG IP is directly routed to the ASIC circuit, the Power Management IP controls the Power Units within the Micron board by managing the SPI bus according to the configuration values given via AXI interfaces. To properly validate the FSMs of the Power Management IP which are in charge of elaborating the configuration data and, consequently, generating SPI transactions over the bus, a dedicated SPI agent class is also included in the top-level environment class. The SPI agent allows the data transfer over the bus and includes a model of the configuration registers of the Power Unit component, useful to verify the exactness of DUT behavior and the SPI transactions generation. Finally, the entire architecture of the proposed verification tool includes different configuration objects that allow us to make the tool reusable and flexible. As for the verification IPs developed within the PhD work, a dedicated configuration class has been also provided thus introducing several functionalities for the users. In particular, thanks to the proposed configuration class, the user is able to:

- modify the overall topology of the top-level environment class indicating the number of interfaces that must be managed.

- define a default UVM sequence to be executed in the configuration phase of the simulation. It is worth pointing out that the device under test requires the execution of dedicated procedures to set up all the FSMs of the Main IP and the ASIC component making the overall system ready to exchange data and information with the storage device.

- configure each agent within the architecture thus enumerating the active and the passive ones.

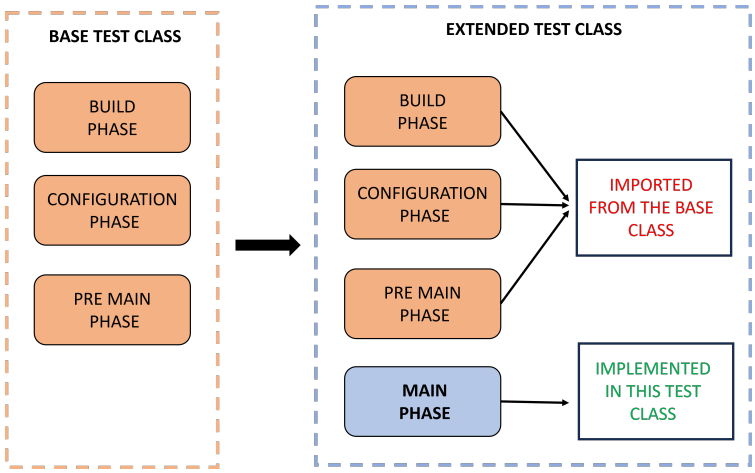


Figure 3.5. Overview of the base test class and the extended test class developed for the proposed verification tool.

In addition, the UFS VIP provides three different configuration classes to properly change the behavior of the inner storage model and, consequently, explore all the features of the UFS standard. The combination of the custom configuration classes and the Cadence VIP object allows us to dynamically define the architecture of the verification tool and the stimuli generation rules to stress the DUT according to the UFS standard. After a comprehensive description of the block diagram in Figure 3.4, the second part of this section focuses on the explanation of the numerous advantages and features of the proposed verification tool as well as several issues addressed within the verification flow. It must be considered that the effort required to develop a simulation tool is not restricted to the definition of the overall architectures of the UVC needed, the verification engineer has the task of managing the test repository as well as implementing smart scripts in order to make the tool user-friendly. In particular, the maintenance of the test repository requires a huge effort in the industries especially regarding projects that have a long target time. Here, the DUT continuously evolves with new versions implementing additional features or aspects of a predefined standard; as a consequence, over the development of a complex digital circuit, the tests must be modified or adapted

to new requirements. An optimal solution for the industries consists of the development of flexible and reconfigurable test classes for the simulation environment allowing the verification teams to reproduce a huge number of scenarios with a reduced number of tests within the repository. In order to improve the quality of the proposed verification tool, an advanced usage of the UVM phase mechanism [5] has been introduced in the UVM test class definition for the Micron system as depicted in Figure 3.5.

This approach aims to fragment the execution of the simulation into several steps by properly managing the different UVM phase methods. Firstly, a base test class has been developed that is in charge of implementing:

- Build Phase: here the top-level environment described in Figure 3.4 is created, and the configuration objects are properly associated with the different classes in the system.
- Configuration Phase: this is the first time-consuming step of the simulation. According to the configuration class, a dedicated sequence is executed which enables all the finite state machines and circuits inside the DUT according to Micron proprietary procedures.
- Pre-Main Phase: this step of the simulation, according to an input of the launch script of the simulator, is able to select dynamically a UVM sequence within a predefined list creating more than one scenario before the execution of the user's test within the main phase. As an example, the user could insert an additional configuration procedure before the test effectively starts thus changing the behavior of the DUT or the UFS VIP.

Then, an extended test class has been implemented, the latter includes the sequences that must be executed during the main phase of the simulation. As an outcome, a single test is able to simulate many aspects of the DUT as well as the UFS protocol by properly combining the inputs of the configuration and pre-main phase reducing the number of tests needed to achieve a predefined target coverage. It becomes clear that the test maintenance is extremely simplified; indeed, if an improvement of the simulation tool is required, the verification team has the task of changing a small subset of

[5] Refer to section 2.5 for details

tests, which are repeated in the regression test list with different input parameters and/or configuration objects. A general description of the scripts used to configure the simulation tool properly has been reported in this section as in Figure 3.6.

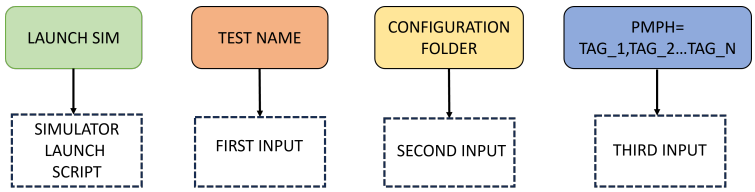


Figure 3.6. Simulator script description.

The launch script of the simulator requires three inputs:

- TEST NAME: the user must indicate the name of the desired test to be executed. The script, automatically, is able to find the correspondent file within the test repository and, subsequently, creates a link to the test within the launch directory.
- CONFIGURATION FOLDER: the user must indicate the name of the folder that contains all the files needed. As for the test name, the script is in charge of searching the files within the configuration object repository and creating the required links in the launch directory.
- PMPH: here the users must indicate a series of tags, accurately separated by a token, the latter inputs are processed by the test during the pre-main phase thus executing different UVM sequences.

For the sake of simplicity, a practical example of the pre-main phase parameter (PMPH in Figure 3.6) usage is now discussed aiming to highlight the considerable advantages regarding the repository maintenance. According to the UFS standard, the WRITE and READ commands require the definition of complex data structures that collect all the information needed to execute the operation such as the Logical Block Address (LBA) or the Expected Data Transfer Length (EDTL) fields. In addition, the standard indicates the operational speed of the entire system thus setting the frequencies of the data transfer operations between the storage

system and the digital circuits connected to its interfaces; typically, the UFS nomenclature highlights the allowed speeding configuration as *gears* with a range from one to five(highest operational frequency). Let us consider a simple test that properly simulates the execution of two WRITE commands and two READ commands comparing the data written with respect to the payload read from the storage system. In this example, the UVM pre-main sequence is able to process the third input parameter of the developed launch script as in Figure 3.6; accordingly, one tag is used to configure the expected data transfer length within the command data structures[6] meanwhile the second tag is used to change the gear. Please note that in order to modify the UFS gear a dedicated UVM sequence must be executed, then the FSMs in the IP cores as well as in the ASIC circuit are in charge of adapting their frequency and managing the storage interface according to the predefined gear. It can be demonstrated that by implementing one single UVM test up to ten different scenarios could be simulated by adequately combining the pre-main phase tags for a given configuration folder. Finally, let us analyze the following example script usage:

```
1 launch_sim  test_example  CONF_DIR  PMPH=BLK_2,GEAR_1
2 launch_sim  test_example  CONF_DIR  PMPH=BLK_4,GEAR_1
3 launch_sim  test_example  CONF_DIR  PMPH=BLK_2,GEAR_2
4
5 launch_sim  test_example  CONF_DIR  PMPH=BLK_4,GEAR_5
```

Figure 3.7. Code example of a launch_sim script whit different pre-main phase input parameters.

Here, the BLK parameter is processed by the pre-main phase sequence to assign a specified value of the EDTL, in this example, two or four UFS blocks are involved in the data transfer. It becomes clear that ten scenarios could be evaluated by properly choosing one of the lines shown in Figure 3.7 while only one UVM test has been written and added to the test repository. Furthermore, the advanced usage of the phase mechanism introduced in the verification tool improves the flexibility of a single test simulation,

[6]The UFS protocol states that the EDTL field for both WRITE and READ command must be a multiple of the UFS block that is 4 Kb in size

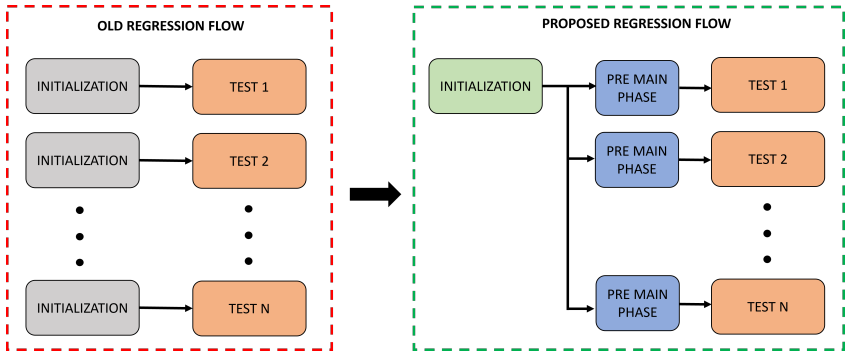


Figure 3.8. Regression flow block diagram.

useful to debug a new feature or develop a new test case. However, within a complex digital design, the verification flow requires the execution of one or more regression test lists which aim to validate the overall functionalities of the DUT before the implementation step of the system. In this regard, regression management plays a crucial part in the verification plan; conventionally, these procedures require a significant simulation time while the simulator or imported VIP licenses are locked [7]. As a consequence, the verification teams in the industries pay great attention to the regression techniques and innovative approaches to achieve the best solution and reduce the overall time to complete a single regression. Over the collaboration with Micron, considerable improvements regarding the regression flow have been treated increasing the overall quality of the validation plan as well as the hardware and software resources usage. To properly evaluate the benefits coming from the proposed methodology, we carried out a comparison between the performance of the proposed regression management and the previous one adopted in Micron, which has been used to validate the older version of the DUT based on the UFS 3.1 standard. It is worth pointing out that the previous Micron hardware validation tool team solution for the simulation environment is not compliant with the UVM standard thus restricting the interoperability of the team inside and outside the company. In addition, the implementation of the techniques,

[7] As an example, the Micron hardware validation tool team acquired a limited number of the UFS VIP licenses by Cadence Design System. Indeed, the cost of each license is 30k $

described in the following, is not allowed while the verification environment is not UVM based. Moreover, the idea behind the improvements achieved in addressing the regression issue is to import the advantages of the UVM standard thus increasing the versatility of the flow and reducing the regression time needed.

According to Figure 3.8, while the previous regression flow implemented by the Micron team simulates the initialization phase for each test of a regression list, the proposed methodology overcomes the need to replicate the simulation of the DUT configuration phase for a test list. More specifically, given a subset of tests that share the same configuration folder, an innovative *save and restart mechanism* has been implemented to reduce the regression time execution. The latter technique consists of creating a simulation snapshot at the end of the **configuration phase** thus storing the initialization stage for all the scenarios within the list. Then, each test starts from the **post configuration phase** on the basis of the stored checkpoint thus executing all the required sequences of both UVM pre-main and main phase. To this end, a dedicated script has been developed for the regression list management and it is in charge of elaborating a regression list as follows:

```
../regression_folder/test_list_1    ../config_folder/CONF_1
../regression_folder/test_list_2    ../config_folder/CONF_2

../regression_folder/test_list_N    ../config_folder/CONF_N
```

Figure 3.9. Code example of a regression list for the proposed verification tool.

It must be considered that most of the configuration attributes within the configuration classes are used in the build phase to properly adapt the architecture of the simulation tool or change the behavior of the UFS VIP; thus, the modifications are applied at the zero-time of the simulation and additional changes in the next phases are not allowed. To this end, more than one configuration folder is required in the proposed simulation tool, and, subsequently, the overall number of initialization checkpoints increases. For this reason, a massive effort has been dedicated to reducing the configuration files needed thus achieving the best decreasing time of

the regression execution. According to the example code in Figure 3.9, a generic test list can be summarized as:

```
../test_repository/test_A     PMPH=BLK_2,GEAR_1
../test_repository/test_B     PMPH=BLK_4,GEAR_4
../test_repository/test_B     PMPH=BLK_2,GEAR_5

../test_repository/test_M     PMPH=GEAR_3
```

Figure 3.10. Code example of a test list for the proposed verification tool, for each test the pre-main parameter is required.

It is worth emphasizing that the regression script is in charge of elaborating the pre-main phase parameter as well as the launch sim script. In addition, the replication of one test within a test list is allowed with a different PMPH parameter thus evaluating a different scenario by means of the same test. In this example, the *test_ B* is executed with both GEAR4 and GEAR5 speed mode configuration. Hence, the proposed regression management merges the overall advantages of the *pre-main phase management* and the *save and restart mechanism* giving a practical solution for a challenging issue of the digital verification plan. To quantify the benefits of both the techniques described above, experimental data are reported in the following Table 3.1.

	N° Scenario	N° Test	Regression Time	% Simulation Time Saved	% Test Reduced
Micron	82	82	14 h 39 m	-	-
Proposed	82	58	11 h 48 m	19.5	29.3

Table 3.1. Comparison between the previous regression flow in Micron company and the proposed one.

Here, the RTL simulations are performed by means of the latest generation of computer farms[8]. Therefore, the state-of-the-art servers allow us to execute the regression scripts with the highest performance reducing the simulation time required. Within this practical example, to simulate up

[8]The server model is HPE ProLiant DL365 Gen10 Plus characterized by 2 CPUs and 16 CORES. The maximum CPU frequency is equal to 3.693 GHz

to 82 scenarios the required number of tests is 58, the latter has been adequately replicated with different PMPH input parameters. In addition, instead of integrating the initialization phase for each scenario, only 25 different initialization sequences, have been executed and, consequently, stored as the checkpoint for the regression management thus decreasing the overall simulation time. Hence, the verification tool introduces a new methodology to speed up the regression test list execution as well as to reduce the number of tests within a test repository by a significant percentage. In addition, the regression script is able to run in parallel more than one test list thus optimizing the usage of the hardware resources and the software licenses and further reducing the simulation time for the regression evaluation.

Chapter 4

Hardware emulation technique

This Chapter describes an innovative technique for the verification of complex digital designs which aims to improve the efficiency and quality of the verification plan combining a considerable reduction of the validation time required. The software simulation of the RTL code of the digital system represents the first step of the verification process within the industries and the university. Over the last years, efforts of academic research and industry in the field of digital verification engineering have contributed to the development of methodologies and techniques to improve the robustness and effectiveness of the software verification approach thus collecting all the verification best practices within the definition of the Universal Verification Methodology standard, described in detail in the previous chapters of this dissertation. Nevertheless, it must be taken into account that there are profound differences between the simulated system and the physical system characterized by the Micron custom PCB board connected to a PC desktop via PCIe bus.

Figure 4.1 depicts a block diagram comparison of both systems.

A verification plan exclusively based on software simulations presents different disadvantages representing several open points within the digital verification field. The overall drawbacks consist of:

- the software simulation time required to achieve a high verification coverage for complex designs.
- the impurities or implementation defects introduced by the synthesizer tools during the conversion of the RTL code into digital design.

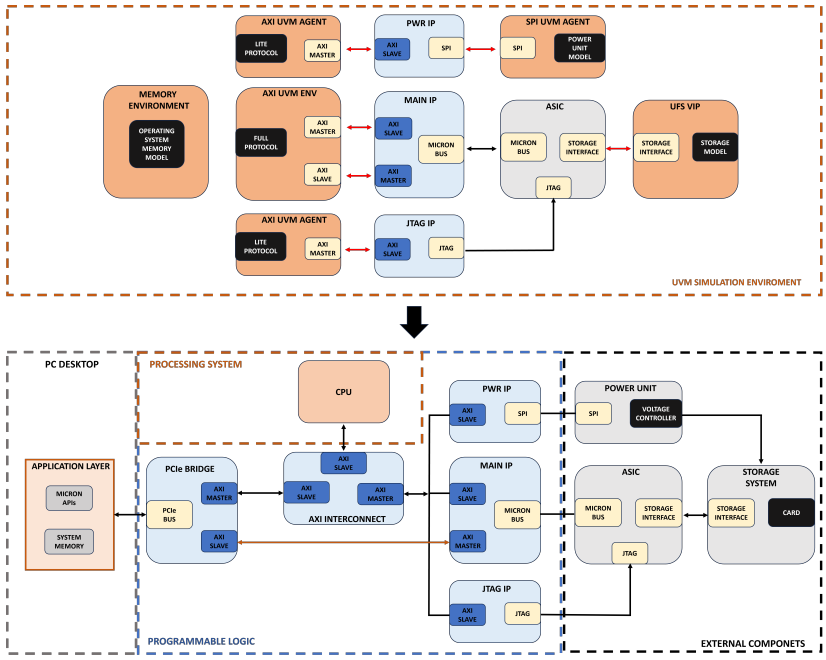


Figure 4.1. Block diagram comparison of the simulated system and the overall hardware components in the real system.

- the latency introduced by the operating system to generate the UFS command payload.
- the latency introduced by the PCIe controller to send/receive the UFS packets and the data payload from the application layer to the SoC components and vice versa.

During the PhD work, these aspects of the verification were deeply analyzed; therefore, we proposed two different approaches in charge of addressing the mentioned issues by introducing the hardware emulation in the verification plan. More in detail, we highlighted two possible flows for the hardware emulation introduction:

- System Level Technique: here the idea is to modify the UVM class infrastructure of the simulation environment by integrating the application layer's rule inside the test class definition. The approach

aims to generate a Python script using the simulation tool, the latter script will be processed by the operating system of the external PC, and, consequently, will apply the amount of stimulus directly on the digital circuits of the Micron system.

- Low-Level Technique: this is a more general approach for the hardware emulation, it is based on the introduction of hardware components inside the Programmable Logic of the SoC. The proposed circuits will translate into a hardware version of the functionalities of the UVM agent class allowing the input stimuli forcing of the DUT and the monitoring of the crucial interface activities. Since the entire validation flow can be implemented by exploring the SoC capabilities, the external PC is not required to test the Micron system.

Both techniques have been successfully developed allowing us to overcome the simulation time bottleneck as well as to test the real digital circuits synthesized. The choice to develop two versions of the hardware acceleration depends on several issues. The system-level technique does not allow direct control of previously described latencies whereas the low-level one introduces great flexibility in terms of latencies management, provided that there are sufficient resources in the SoC device to integrate the hardware components needed. It becomes clear that smart usage of these approaches ensures that the hardware emulation approach can be applied to the majority of test cases of interest. The translation process of a UVM test class into the emulation test is **automated** for both approaches implemented; furthermore, the overall hardware emulation solutions presented in this chapter could be continuously improved by inserting new components in charge of managing additional protocol interfaces.

The Chapter is organized as follows: the section 4.1 describes the hardware emulation performed by means of an accurate usage of the application layer of the external PC, connected to the Micron System over the PCIe bus; furthermore, the section 4.2 introduces an innovative approach to improve the quality of the verification by using the potentiality of the SoC platform and exploring the functionalities of both Processing System (PS) and Programmable Logic (PL) part within the SoC device. Finally, the section 4.3 summarizes the overall benefit of the methodologies analyzing the experimental results in Micron Technology. It must be considered that

only a general description of the implemented approaches for the hardware emulation can be discussed in this dissertation due to the Non Disclosure Agreement (NDA) already mentioned in the previous section. This part of the thesis focuses on highlighting the relevant innovation introduced within the Micron team collaboration and enumerating the advantages of the proposed methodologies.

4.1 System-level emulation technique

The proposed technique has been carried out in collaboration with Micron Technology and aims to accelerate the validation process of the system depicted in Figure 3.1. More in detail, the Micron system is connected to an external PC over the PCIe bus, a general overview of the components involved in this architecture is shown in Figure 4.2. Moreover, in the previous section of this dissertation, the DUT has been defined as the combination of the IP cores within the PL of the SoC device and the ASIC component; thus, the PCIe management and the connection with the PC have been excluded from the software validation process. The UVM simulation environment enables us to deeply verify the behavior of the IP cores within the SoC Programmable Logic and their connection, the communication of the Main IP with the ASIC component through a Micron proprietary bus, and, finally, the data traffic toward the storage device according to the UFS protocol. The focus of the verification approach described in the previous section is more on the exhaustive analysis of the overall features implemented in both SoC and ASIC devices to sustain the communication toward the UFS device and less on the PCIe bus management for the external PC connection.

Although the proposed simulation environment introduces several advantages for both verification and design flow, there are several drawbacks concerning the software validation approach that require an innovative solution for the industries. The first disadvantage of this approach is the software time consumption which conventionally increases with the complexity of the design under test. More specifically, there is a crucial tradeoff between the regression time execution and the desired target coverage for the device validation; thus, for the industries, an optimal choice of the maximum regression time could be quantified into 10/12 hours thus al-

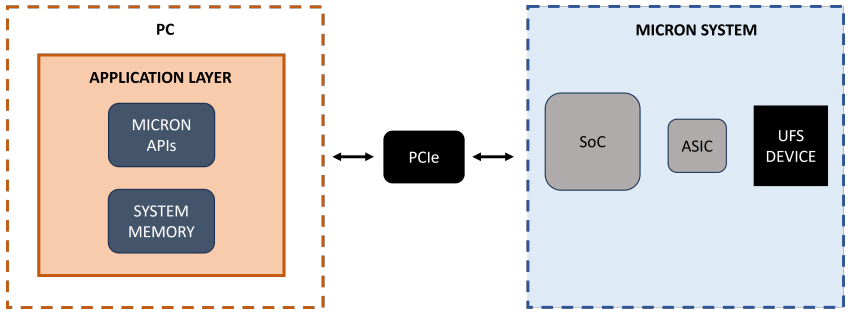


Figure 4.2. Block diagram of the Micron System and the PC connection.

lowing the competition of the overall simulations during the out-of-office time of the verification teams, typically during the night. For the sake of simplicity, considering the proposed validation tool for the Micron system, the time required for a single simulation could take at least 3 minutes to perform the DUT's configuration procedure meanwhile the duration of a test for a specific feature could require from 5 minutes to 1 hour. It becomes clear that a comprehensive debugging of the overall features within the system under test implies a considerable imbalance of the verification plan against the maximum simulation time allowed for the regression step. Even though we can reduce the regression time consumption by adopting the regression flow management described in Figure 3.8 and running more than one regression list in parallel, a huge number of tests are needed to achieve a high-level coverage of the design. In addition, it must be considered that the device under test validated through the software approach does not include the digital components for the PCIe bus management, this latter could introduce variable latency during the data exchange that may cause errors during the on-board execution of the UFS commands toward the storage device. Hence, in order to address these vital issues related to verification flow, an innovative *system-level hardware emulation technique* has been proposed in this dissertation. The idea behind this verification methodology is the introduction of hardware acceleration into the Micron validation process reducing the time required by the verification flow and analyzing also the interaction between the Micron board and the external PC. More in detail, this technique aims to combine the enormous advantages of the UVM standard with the hardware acceleration through-

out a novel **automated methodology**; thus, the latter is in charge of translating a UVM test into a Python test that can be directly executed by the application layer of the Micron system summarized in Figure 4.2. As described in the previous sections of this thesis, the Micron software tool team implemented a series of APIs that enables the users to configure a dedicated system memory properly, manage the PCIe bus, and, finally, control the system implemented in Figure 3.1. The proposed verification methodology is based on the smart usage of the operating system's application layer in order to replicate the same scenarios of a UVM test. To better clarify how this technique has been carried out, it is worth analyzing in more detail the structure of a generic UVM test implemented for the software simulation tool discussed in this work. Each test in the repository has the same structure as the *extended test class* highlighted in Figure 3.5. During the main phase of the simulation, this object is in charge of defining the input stimuli sequence for the DUT analyzing a particular feature of the system. In order to generate a complex sequence of stimuli, the test class randomizes and combines several more simple sub-sequence classes creating an elaborate scenario. The latter sub-sequence classes can be created by calling different tasks and functions that have been developed to make the test definition easier for the user. We collected the aforementioned base sequence classes and the implementation of the tasks into a file, named *UVM sequences collector*, that is included in each test of the proposed simulation tool. This emulation approach consists of importing the amount API rules of the Micron application layer inside the test class structure allowing the entire class infrastructure of the proposed simulation tool to automatically generate a Python script. The latter file will be in charge of executing on the hardware system all the scenarios highlighted by the software version of the test. Hence, the translation process of a UVM test for the Micron system starts from the development of a Python file that implements the same functionalities of the *UVM sequences collector* by using the APIs for the application layer management. Figure 4.3 depicts a possible structure of the *UVM sequences collector* and the *Python sequences collector*. It is worth pointing out that this step of the proposed methodology is not automatized while the conversion of a UVM sequence into a Python code is performed by means of an unconventional usage of the Micron APIs of the application layer. Then, the described Python

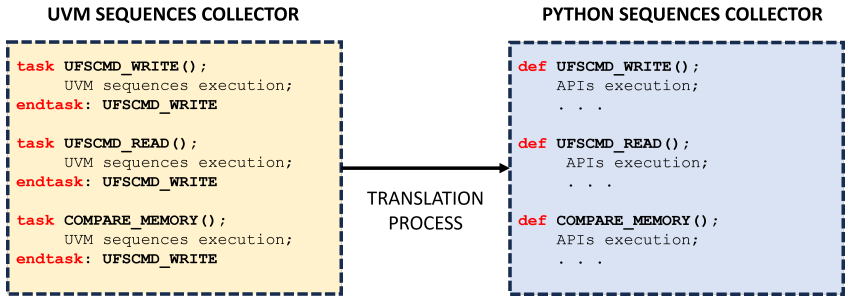


Figure 4.3. General overview of the UVM sequences collector and Python sequences collector.

file is included in all tests that can be directly executed over the hardware system as well as the UVM sequences collector file is inserted in each UVM test class. The proposed technique for the UVM test conversion exploits the similarity between the *collector files* to speed up the automated generation of the Python test including the benefit of the UVM standard. As an example, the translated test introduces a randomization process of sequences for each scenario. Indeed, during the generation of the Python collector file, a perfect match between task names and input parameters has been guaranteed between the two versions. Hence, an additional parameter to the *launch sim* script in Figure 3.6 has been introduced that allows configuring the verification tool to:

- Execute the software simulation by randomizing the UVM sequences, applying the stimulus to the DUT interfaces, and, finally, performing all the checks required.
- Execute only a subset of operation in software as the randomization process of the sequences. Then, instead of applying the stimuli by running the class previously randomized, the tool is able to create and print on a file the Python version of the test by choosing the function definitions within the *collector files*. Finally, the generated file provides, at the end of each scenario, the check procedure that must be performed in order to establish if the system works correctly.

Moreover, the proposed emulation flow can be summarized in three macro steps that each verification team member must apply to enable this tech-

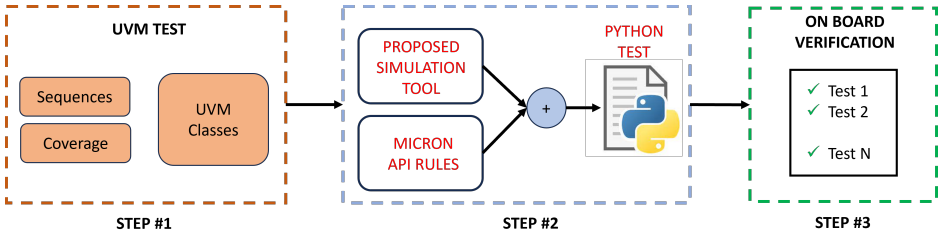


Figure 4.4. Block diagram of the proposed system-level hardware emulation technique.

nique and better verify the correctness of the Micron system. Figure 4.4 gives a block diagram visualization about the following steps within the hardware emulation flow:

- **Step #1:**The technique is based on developing a UVM test which is responsible for producing a huge number of scenarios and integrating a functional coverage definition for a specific feature under test. Therefore, the test structure must provide a code part that the user can insert in a for/while loop thus generating different scenarios within a single test definition. Hence, to exploit as much as possible the potentiality of this methodology, the user must define a dedicated randomization process of the overall inputs and/or sequences needed for each scenario. It is worth pointing out that this verification step aims to generate a UVM test which is impossible to execute with a software approach due to the simulation time required to complete all the scenarios. Optionally, we propose a *smart incremental coverage technique* that consists of simulating or emulating a specific scenario only if the latter increases the coverage level thus reducing the replication of the input parameters among the randomization process. Then, the UVM test ends when the computed functional coverage reaches a target value.

- **Step #2:** The user is able to simulate a reduced version of the simulation or generate the Python version of the implemented test, the latter collects all the scenarios and can be directly executed by the application layer. Indeed, the verification tool loads the API rules and, automatically, translates each scenario exploiting the similarity

between the *collectors files* above mentioned.

- **Step #3:** Finally, the output Python file must be transferred to a hardware platform composed of a Micron PC plus the DUT board connected via PCIe bus. Here, the application layer can execute the translated Python test thus evaluating directly the digital circuits implemented within the SoC design, the ASIC chip, and, subsequently, the UFS device and its physical interface. Then, the onboard verification flow produces detailed log files to capture additional information in case of test failure.

Furthermore, the proposed hardware emulation technique presents several advantages strictly connected with the simulation environment discussed in the previous chapter. The presented approach enables the Micron team to replicate on a hardware platform an enormous variety of tests overcoming the software limitation and integrating the test randomization process from the UVM standard throughout an automated verification flow. In addition, in case a test failure occurs, we add several details about the randomized scenario that reproduces the failure in a log file. Hence, this scenario could also be simulated by using the UVM verification tool thus enabling the user to analyze the RTL code and fix the detected bug within the design.

4.2 Low-level level emulation technique

The system-level emulation technique enables us to drastically reduce the Micron subsystem's testing time, achieving an unimaginative level of verification coverage. Thanks to this validation process, the quality of the design is constantly improved allowing all the Micron teams involved in this project to establish the behavioral correctness of the DUT before a product release. In addition, even if a customer detects a problem within the device, the analysis of the UFS command failure could be easily performed by means of both the simulation tool and the hardware acceleration of the UVM test thus collapsing the time required to fix the problem of the product.

However, the emulation technique discussed in the previous subsection is highly dependent on the Micron system's latencies introduced by:

- the management of the operating system's application layer and APIs usage.
- the digital circuits in charge of controlling both AXI and PCIe bus within the device under test thus introducing a non-deterministic delay during the communication between the PC and the DUT.

Unfortunately, the system under validation in Figure 4.2 is adversely affected by the mentioned variable latency; therefore, the IPs within the SoC Programmable Logic could generate a hardware error depending on one o more critical delay times. Although the simulation tool of the entire system introduces the possibility to define and randomize a time parameter to emulate the operating system's delay within each UVM agent, the system-level hardware emulation technique based on Python APIs usage does not provide direct control of the system latencies during the communication between the PC and the IP cores of the Micron system. Hence, while a system-level hardware acceleration is performed, there is a considerable random delay within the data transfer. It becomes clear that a comprehensive validation of the DUT introduced in this dissertation requires an additional improvement in the field of the emulation approach allowing the user to properly configure the delay time for the stimuli application over the DUT interfaces. In this regard, a *low-level hardware emulation technique*, based on smart usage of the SoC resources, has been carried

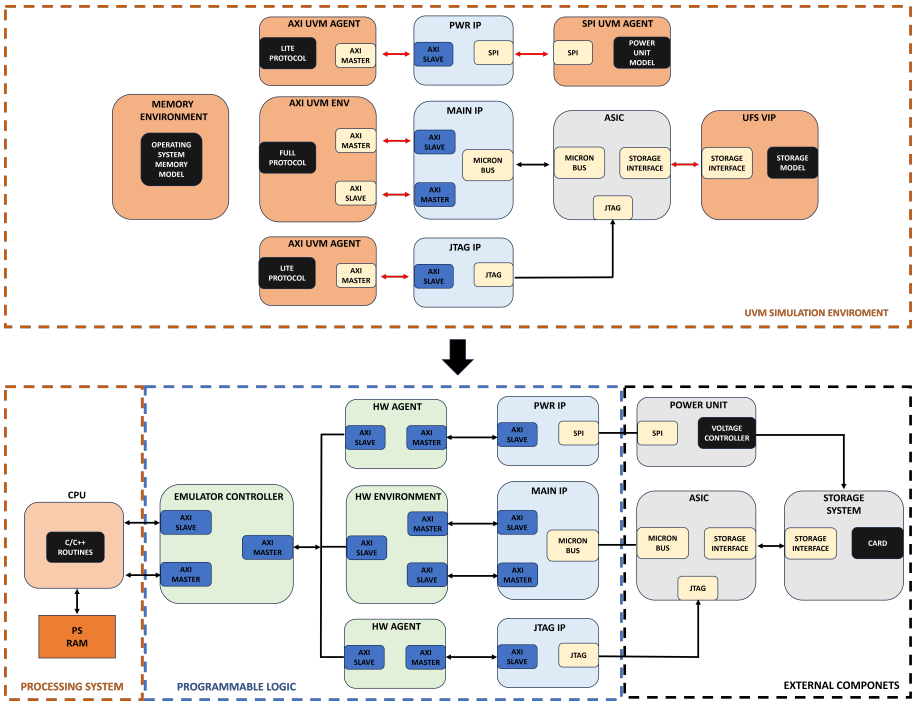


Figure 4.5. Low-level hardware emulation block diagram and comparison with the simulation tool architecture.

out during the collaboration with Micron. More specifically, the proposed methodology aims to combine the overall achievements of the hardware acceleration approach, described in the previous section, with an innovative methodology to properly control the DUT interfaces and the system latencies during the onboard execution of a test. The proposed approach for hardware acceleration introduces considerable flexibility within the verification plan by exploring the resources of the SoC device and adding several hardware components within the Programmable Logic of the SoC.

Even if the goal of both system-level and low-level emulation techniques consists of the development of an efficient methodology for the digital circuit's validation with a low testing time, it is worth emphasizing that the implementation steps, as well as the usage of these techniques within the Micron verification flow, are completely different. By adopting the low-

level technique, the idea is to avoid the usage of the external PC and, subsequently, the APIs of the operating system to control the IPs within the SoC, the ASIC chip, and the UFS 4.0 card. Figure 4.5 highlights the low-level hardware emulation architecture making a comparison with respect to UVM simulation tool presented in this work. Indeed, the low-level emulation methodology tries to take many advantages of the Universal Verification Methodology. By implementing this novel approach, we improved the architecture of the overall IPs within the Micron system as well as we introduced several C/C++ routines in order to control the Processing System of the SoC and, of course, the entire digital system. It is worth pointing out that the PCIe controller has been removed while the hardware acceleration process is performed through the usage of only PCB board shown in Figure 3.1. Here, the potentiality of the SoC integrated into this system has been analyzed to produce a hardware emulation version of both the CPU's firmware code and digital circuits synthesized within the FPGA device. In this regard, a verification release of the Micron product, which combines the improvements needed for our scope, has been implemented to validate the entire system and, consequently, the device under test for this work. According to Figure 4.5, this section gives a general overview of the translation process of a UVM simulation environment into an emulation infrastructure based on efficient usage of the SoC resources. As far as the Processing System management is concerned, different SystemVerilog tasks and functions from the simulation tool have been transposed into a C/C++ version and stored in a header file, which is included by all emulation tests. In particular, the obtained file collects all the procedures useful to process the UFS response data structures. According to the new hardware emulation approach, the CPU is in charge of:

- specifying the number of scenarios that the hardware subsystem must execute giving details about the UFS commands of interest. More specifically, the CPU indicates a subset of UFS command that will executed after a randomization process.
- defining the specific bus activities of the DUT's interfaces that must be monitored and collected.
- controlling the status registers of each IP in order to establish if an error occurs.

- defining a range of the desired delay time that the hardware components have to introduce during the hardware acceleration operation.
- performing a post-processing analysis of the overall data coming from the Programmable Logic and stored into the PS RAM during the emulation test execution. Thanks to this evaluation, the Processing System determines whether a test has been successful.

Therefore, the communication between the Processing System and Programmable Logic has been radically changed to implement the new emulation approach by inserting a hardware layer of custom IPs; here, the latter circuits have been highlighted by means of green boxes in Figure 4.5. Firstly, a *emulator controller IP* has been developed that is in charge of managing the communication between the PS and the additional components introduced within the PL side. Furthermore, the controller IP receives the configuration data from the CPU processing them to execute the desired test. More specifically, it has the task of:

- executing dedicated procedures to enable the entire digital system configuring the overall finite state machines within the IPs.
- translating the data coming from the CPU into specific operational code that could be directly processed by the FSMs within the hardware agents. Then, the controller sends the opcodes to the IPs to properly manage the behavior of the hardware agents.
- managing the start and stop signals to synchronize the activities of the hardware agents thus indicating the delay time parameter for each command execution.
- transferring the transactions collected by the monitoring activities to the PS RAM allowing the processor to process them.
- generating an interrupt when an error occurs within the data transfer according to the UFS protocol.

Therefore, the additional digital circuits implement directly in hardware several functionalities of the UVM classes. More in detail, in this PhD work a hardware version of the UVM agent classes has been carried out as depicted in Figure 4.6.

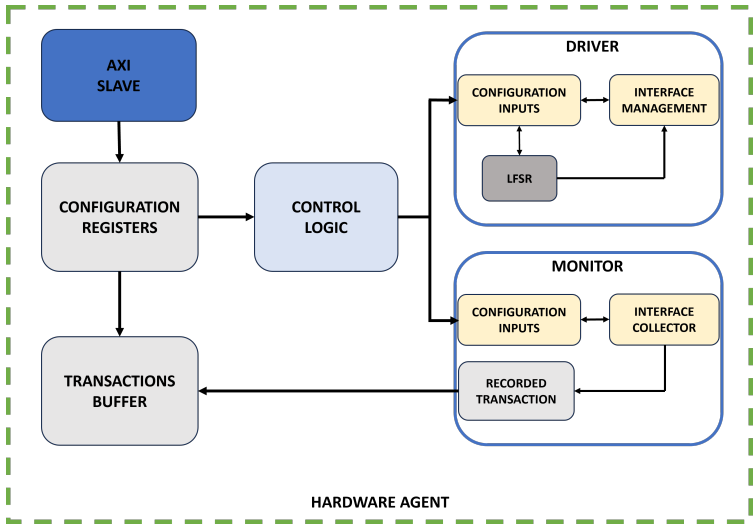


Figure 4.6. Generic hardware agent block diagram within the low-level emulation technique.

The latter digital system is characterized by an AXI interface, which implements the protocol's lite version and allows the *emulator controller IP* to access the configuration registers. The information stored in these registers is analyzed by an inner control logic circuit which is in charge of processing the operational codes and, subsequently, managing the control signals for both *diver* and *monitor* hardware version. Furthermore, the *driver* digital circuit instantiates different Linerar Feedback Shift Registers (LFSRs) to generate and send a *pseudo random payload* over the DUT interfaces. Depending on the configuration given to the hardware agent, the driver component is able to sample the output data from the LFSR circuits forwarding the information to the device under test. In addition, the interface management circuit introduces the desired latencies for testing purposes allowing direct control of this crucial aspect within the verification methodology. Conversely, the hardware version of the monitor class recognizes the UFS transaction flow by analyzing the DUT interface activities; then, when a valid transaction is detected, the information is sent to a *transactions buffer*. According to Figure 4.5, It must be considered that the Main IP interfaces are managed by a *hardware environment* component, the latter translates into a digital circuit the functionalities

of several UVM classes such as the memory environment and the AXI environment. Furthermore, the circuit includes the instance of two different hardware agents that have the task of generating pseudo-random UFS data structures according to a pre-configured command subset and, consequently, transmitting them thanks to the driver components. More specific implementation details about the emulation infrastructure cannot be discussed due to the NDA signed with Micron Technology. Therefore, this verification technique for digital designs allows full control of the UFS packets flowing through the DUT interface toward the storage device. Hence, the proposed methodology overcomes the limitation of the system-level hardware emulation approach concerning the latencies of several components. In addition, the low-level emulation needs a simplified hardware platform that consists of the digital system connected within the PCB board depicted in Figure 3.1. It is worth mentioning that both hardware acceleration methodologies presented in this work are essential to significantly improve the Micron verification flow. While the first solution enables the hardware acceleration of the overall UFS command giving important flexibility regarding the digital system configuration, the second one cannot execute the entire UFS command list within the specification but only a subset thus introducing a bottleneck for the validation purposes. Nevertheless, by properly combining the system-level and low-level emulation techniques, the overall efficiency of the verification plan within Micron is considerably increased allowing the development of a high-quality UFS 4.0 storage device and IPs system within the SoC Programmable Logic as well.

4.3 Benefits evaluation

Several features of the DUT have been validated by using the methodologies presented in this part of the dissertation during the collaboration with Micron Technology.

For the sake of clarity, an experimental example is reported thus quantifying the overall achievements of these techniques. Both system-level and low-level hardware acceleration techniques have been applied for the validation process of a crucial feature of that Main IP within the SoC device; therefore, the UVM test implemented and reported in this example

produces up to **twenty thousand** different scenarios achieving a prefixed target coverage. Each randomized scenario takes at least 5 minutes to complete the simulation; furthermore, we assume to reserve 10 licenses of the UFS VIP integrated into the simulation architecture for the regression execution within this example. For this reason, no more than 10 regression lists could be processed concurrently.

Hence, even if we can divide the 20K scenarios into ten different simulations by launching several scripts in parallel, the simulation time required consists of **7 days** where all the verification activities are impacted due to the lack of licenses.

	Numer of Scenarios	Execution Time	Target Coverage
Software approach	20 K	$\sim$ **7 days**	**UNREACHABLE**
Emulation approach	20 K	$\sim$ **5 minutes**	**ACHIEVED**

Table 4.1. Comparison between the software simulation approach and the proposed hardware emulation techniques.

As reported in Table 4.1, the proposed emulation techniques take about **5 minutes** to execute 20K different scenarios over the hardware platform. To this end, the techniques allow us to deeply verify a specific feature achieving a functional coverage level which is **unreachable** by using only the software approach within the validation plan. In addition, it must be considered that these techniques introduce a considerable reduction in the overall cost of the DUT verification step. Indeed, the total amount of 10 UFS VIP licenses consists of 300K $ whereas the price of the system depicted in Figure 3.1 is estimated at about 20K $ thus significantly decreasing the budget required for the validation process. As an outcome, the methodologies described in detail in this chapter represent an innovative improvement of the Micron verification flow allowing a comprehensive analysis of a complex digital system characterized by the interaction between many devices connected within one hardware platform.

Chapter 5

Novel FFA algorithm for Convolutional Neural Networks

In this Chapter, an innovative algorithm based on FFA approach is discussed. In particular, the methodology aims at reducing the overall computation of the Convolutional Neural Networks, widely used within deep learning applications concerning object detection, image recognition, and speech classification. The Chapter is organized as follows: section 5.1 introduces the motivations of the proposed solution, whereas section 5.2 gives an overview of the FFA algorithms and their application regarding the Artificial Intelligence (AI) solutions. Thus, section 5.3 introduces the novel algorithm implemented during the PhD work; finally, section 5.3.1 and 5.3.2 describe the mono-dimensional and bi-dimensional versions of the proposed approach, respectively.

5.1 Motivations

As far as the deep learning models are concerned, the Convolutional Neural Networks (CNNs) have emerged as a cornerstone of the overall AI solutions achieving unthinkable results in the field of computer vision, in particular with respect to image processing and object detection. Conventionally, most of the calculations of these networks are left to the cloud

platforms, where image or video data can be processed by means of dedicated GPUs utilizing high-performance digital devices to execute the entire computations needed. However, this approach presents several drawbacks when the CNN elaborations are required to be performed within an embedded digital system, an Internet of Things (IoT) application, or a real-time processing solution. As a consequence, for these kinds of applications a system based on local accelerators is required whereas a GPU-based platform to deploy a neural network becomes unfeasible due to the power consumption border, the maximum available area, or the cost of the components needed. Thus, an efficient hardware implementation of these networks becomes an open point within state-of-the-art deep learning development. In Literature, several architectures have been presented that aim at reducing the power consumption of the AI solution as well as the size of the digital devices in the system. It is worth pointing out that the vast majority of the CNN computations are restricted to the convolution layers where, primarily, multiplication and sum operations must be performed by means of dedicated architectures. Since the hardware implementation of multipliers increases significantly the power consumption and the overall area of a digital circuit, an efficient architecture for the hardware acceleration of the neural networks has to limit the usage of the multipliers. Hence, efforts in the field of academic research have been focused on the development of dedicated FFA algorithms to reduce the number of multiplication required by the convolutional layers. In this work, we propose a novel algorithm based on FFA approach and partial result reuse. This methodology aims to reduce the multiplication needed to accelerate the computations of the Convolutional Neural Network (CNN). On the basis of the proposed algorithm, a novel hardware data path suitable for both mono-dimensional and bi-dimensional convolution is also presented in Chapter 6.

5.2 Fast FIR Algorithms

The output $y(n)$ of the convolution between the infinite length sequence $x(n)$ and the finite length sequence $h(n)$ can be written as:

$$y(n) = \sum_{i=0}^{k-1} x(n-i)h(i) \tag{5.1}$$

Where k is the length of the sequence *h(n)*. The computation of the single value of *y(n)* requires k multiplications and *k-1* additions. However, if we compute in parallel several values of the sequence *y(n)* we can reuse some of the partial values. In order to do that we can define:

$$\delta = [\delta_0, \delta_1, \cdots, \delta_{2k-2}] \tag{5.2}$$
$$\gamma = [\gamma_0, \gamma_1, \cdots, \gamma_{k-1}] \tag{5.3}$$
$$\bar{x}(l) = [x(l), x(l-1), \cdots, x(l-2k+2)] \tag{5.4}$$
$$\bar{h} = [h(0), h(1), \cdots, h(k-1)] \tag{5.5}$$

Where $\delta_i \in \{0,1\}$ and $\gamma_i \in \{0,1\}$ are binary values and $\delta, \gamma, \bar{x}$ and $\bar{h}$ are row vectors. We then define the partial values $P(\delta, \gamma)$ as:

$$P(\delta, \gamma) = \left(\bar{x} \cdot \delta^T\right) \cdot \left(\gamma \cdot \bar{h}^T\right) \tag{5.6}$$

Where $\cdot$ is the inner product of the vectors. It is worth pointing out that each partial value requires one single scalar multiplication. The values of the sequence $y(n)$ for $n \in \{l-k+1, l\}$ can be computed as a linear combination of the partial values $P(\delta, \gamma)$. Hence, let us introduce an example where $k = 3$ and $\bar{h} = [h(0), h(1), h(2)]$, the value $y(l)$ can be computed as:

$$\begin{aligned} y(l) &= h(0) \cdot x(l) + h(1) \cdot x(l-1) + h(2) \cdot x(l-2) \\ &= (x(l) + x(l-1) + x(l-2)) \cdot (h(0) + h(1) + h(2)) + \\ &\quad + 2\,(x(l-1) \cdot h(1)) + \\ &\quad - (x(l) + x(l-1)) \cdot (h(1) + h(2)) + \\ &\quad - (x(l-1) + x(l-2)) \cdot (h(0) + h(1)) \\ &= P\left(\delta^1, \gamma^1\right) + 2 \cdot P\left(\delta^2, \gamma^2\right) - P\left(\delta^3, \gamma^3\right) - P\left(\delta^4, \gamma^4\right) \end{aligned} \tag{5.7}$$

Where

$$\begin{cases} \delta^1 = [1,1,1,0,0]\,; & \gamma^1 = [1,1,1] \\ \delta^2 = [0,1,0,0,0]\,; & \gamma^2 = [0,1,0] \\ \delta^3 = [1,1,0,0,0]\,; & \gamma^3 = [0,1,1] \\ \delta^4 = [0,1,1,0,0]\,; & \gamma^4 = [1,1,0] \end{cases} \tag{5.8}$$

It is straightforward to verify that the value of $y(l-1)$ can be obtained as:

$$y(l-1) = P\left(\delta^4, \gamma^4\right) - P\left(\delta^2, \gamma^2\right) - P\left(\delta^5, \gamma^5\right) + P\left(\delta^6, \gamma^6\right) \tag{5.9}$$

Where

$$\begin{cases} \delta^5 = [0,0,1,0,0]\,; & \gamma^5 = [1,0,0] \\ \delta^6 = [0,0,0,1,0]\,; & \gamma^6 = [0,0,1] \end{cases} \tag{5.10}$$

As can be seen, the computation of $y(l-1)$ reuses some of the partial values used in the computation of $y(l)$. By properly choosing the set of partial values, it is possible to reduce the overall number of multiplications needed to compute the result of the whole convolution. It must be considered that each value $x(n-1) \cdot h(i)$ used in Eq. (5.1) is one of the possible partial values $P(\delta, \gamma)$, where the vectors δ and γ have only one non-zero value. Therefore, the FFA algorithm generalizes the straightforward method for the computation of the convolution.

5.3 Proposed Algorithm

A novel algorithm based on FFA approach and partial result reuse has been carried out. In the following sections, we will describe the mono-dimensional algorithm and, subsequently, we will analyze a methodology to extend the mono-dimensional approach to the bi-dimensional version, that is suitable for the acceleration of the neural network.

5.3.1 Mono-dimensional approach

We assume that the algorithm can be executed in steps. In each step, we assume to read a number of values of the sequence $x(n)$ equal to the length k of the sequence $h(n)$. We also assume that the algorithm provides, at each step, k different values of the sequence $y(n)$. Therefore, at the first step of the algorithm, the values $x(0), x(1), \cdots, x(k-1)$ are used to compute $y(0), y(1), \cdots, y(k-1)$. At the *i-th* step of the elaboration, the proposed methodology uses the values:

$$\bar{x}(i) = [x(k \cdot i - 1), \cdots, x(k \cdot (i-1))] \tag{5.11}$$

thus producing the following outputs:

$$\bar{y}(i) = [y(k \cdot i - 1), \cdots, y(k \cdot (i-1))] \tag{5.12}$$

It is worth pointing out that (5.11) is different from (5.4) since we consider only k values of the sequence $x(n)$. For each algorithm iteration, we can introduce $(2k-1)$ intermediate results described as:

$$d_{-z} = \sum_{j=0}^{k-z-1} x\left(k \cdot i - 1 - (j+z)\right) h(j) \tag{5.13}$$

$$d_{z} = \sum_{j=z}^{k-1} x\left(k \cdot i - 1 - (j-z)\right) h(j) \tag{5.14}$$

where the parameter $z \in [0, k-1]$. Hence, the value $\bar{y}(i)$ can be evaluated as:

$$\begin{aligned} y(k \cdot (i-1)) &= d_{-(k-1)}(i) + d_1(i-1) \\ y(k \cdot i - (k-1)) &= d_{-(k-2)}(i) + d_2(i-1) \\ &\vdots \\ &\vdots \\ y(k \cdot i - 3) &= d_{-2}(i) + d_{k-2}(i-1) \\ y(k \cdot i - 2) &= d_{-1}(i) + d_{k-1}(i-1) \\ y(k \cdot i - 1) &= d_0(i) \end{aligned} \tag{5.15}$$

Therefore, the aim of the proposed algorithm consists of the computation of the intermediate results in (5.13) and (5.14) with a reduced number of multiplications; in the proposed algorithm we rewrite equation (5.13) and (5.14) as the linear combination of two types of partial values.

The first one is expressed as:

$$P_{typeI}\left(\delta, \gamma, i\right) = \sum_{j_1=\delta}^{\delta+\gamma} x\left(k \cdot i - 1 - j_1\right) \cdot \sum_{j_2=k-(\delta+\gamma)-1}^{k-\delta-1} h\left(j_2\right) \tag{5.16}$$

Where

$$\delta \in [0, k-1] \tag{5.17}$$

$$\gamma \in [0, k-1] \tag{5.18}$$

As can be seen, P_{typeI} is computed using only contiguous values of the input sequence $\bar{x}(i)$ and the coefficient sequence of the filter $h(n)$. Each value of P_{typeI} can be computed with one single multiplication. The second one is computed as:

$$\begin{aligned} P_{typeII}(\delta, \gamma, i) = [x(k \cdot i - 1 - \delta) + x(k \cdot i - 1 - (\delta + \gamma))] \cdot \\ [h(k - (\delta + \gamma) - 1) + h(k - \delta - 1)] \end{aligned} \tag{5.19}$$

Here we considered only two samples of the $\bar{x}(i)$ and $h(n)$ to compute one multiplication.

For the sake of simplicity, an example of the partial values usage to

compute the mentioned intermediate results has been reported in this section. Consider a mono-dimensional Finite Impulse Response (FIR) filter with $k = 3$, the intermediate result d_0 at the $i-th$ iteration of the algorithm can be expressed as:

$$d_0(i) = x(3 \cdot i - 1) \cdot h(0) + x(3 \cdot i - 2) \cdot h(1) + x(3 \cdot i - 3) \cdot h(2) \quad (5.20)$$

It is straightforward to verify that Eq.(5.20) can be rewritten as:

$$\begin{aligned} d_0(i) = {} & [x(3 \cdot i - 1) + x(3 \cdot i - 2) + x(3 \cdot i - 3)] \cdot [h(0) + h(1) + h(2)] - \\ & + [x(3 \cdot i - 1) + x(3 \cdot i - 2)] \cdot [h(1) + h(2)] - \\ & + [x(3 \cdot i - 2) + x(3 \cdot i - 3)] \cdot [h(0) + h(1)] - \\ & + 2 \cdot x(3 \cdot i - 2) \cdot h(1) \end{aligned} \quad (5.21)$$

More in detail, we can rearrange the expression of the d_0 by means of a linear combination of the partial values in (5.16) and (5.19) as reported in the following equation:

$$\begin{aligned} d_0(i) = {} & P_{typeI}(\delta^1, \gamma^1, i) - P_{typeI}(\delta^2, \gamma^2, i) - P_{typeI}(\delta^3, \gamma^3, i) + \\ & + 2 \cdot P_{typeI}(\delta^4, \gamma^4, i) \end{aligned} \quad (5.22)$$

Where

$$\begin{cases} \delta^1 = 0; & \gamma^1 = 2 \\ \delta^2 = 0; & \gamma^2 = 1 \\ \delta^3 = 1; & \gamma^3 = 1 \\ \delta^4 = 1; & \gamma^4 = 0 \end{cases} \quad (5.23)$$

In addition, several partial values highlighted in Eq.(5.22) can be reused for the computation of other intermediate results in Eq.(5.13) and (5.14). Considering the same example, the d_1 can be written as:

$$\begin{aligned} d_1(i) = {} & x(3 \cdot i - 1) \cdot h(1) + x(3 \cdot i - 2) \cdot h(2) = \\ & = [x(3 \cdot i - 1) + x(3 \cdot i - 2)] \cdot [h(1) + h(2)] - \\ & + x(3 \cdot i - 1) \cdot h(2) - x(3 \cdot i - 2) \cdot h(1) = \\ & = P_{typeI}(\delta^2, \gamma^2, i) - P_{typeI}(\delta^4, \gamma^4, i) - P_{typeI}(\delta^5, \gamma^5, i) \end{aligned} \quad (5.24)$$

Where

$$\begin{cases} \delta^5 = 0; & \gamma^5 = 0 \end{cases} \tag{5.25}$$

Generally speaking, not all the possible partial values defined in Eq. (5.16) and (5.19) are required to compute the intermediate results d_{-z}, d_z. The purpose of this section is the introduction of an algorithm that allows selecting a reduced number of partial values for the computation of the intermediate results for every value of the FIR length k. The proposed algorithm is shown in Figure 5.1.

```
for a = 0 : k − 1
  for b = 0 : a
    if (a is even) then
      if (b is odd) then
        if [(b == 1) OR (b == a)] then
        P_typeI (δ = k − b − a, γ = a, i)
        end if
      else
      P_typeII (δ = k − b − a, γ = a, i)
      end if
    else
    P_typeI (δ = k − b − a, γ = a, i)
    end if
  end for
end for
```

Figure 5.1. Pseudo code for the partial values generation.

In order to clarify how this algorithm has been carried out, we introduce a practical example. Considering a filter length k equal to 3, the intermediate results d_{-2}, d_2 can be expressed as:

$$\begin{aligned} d_{-2}(i) &= x(3 \cdot i - 3) \cdot h(0) = P_{typeI}(\delta^A, \gamma^A, i) \\ d_2(i) &= x(3 \cdot i - 1) \cdot h(2) = P_{typeI}(\delta^B, \gamma^B, i) \end{aligned} \tag{5.26}$$

Where

$$\begin{cases} \delta^A = 2; & \gamma^A = 0 \\ \delta^B = 0; & \gamma^B = 0 \end{cases} \tag{5.27}$$

These kinds of partial values are required by the proposed algorithm since they identify two intermediate results throughout a single multiplication. Then, we can evaluate the d_{-1}, d_1 reusing $P_{typeI}(\delta^A, \gamma^A, i)$ and $P_{typeI}(\delta^B, \gamma^B, i)$ as :

$$\begin{aligned} d_{-1}(i) &= x(3\cdot i-2)\cdot h(0) + x(3\cdot i-3)\cdot h(1) = \\ &= [x(3\cdot i-2) + x(3\cdot i-3)]\cdot[h(0)+h(1)] + \\ &\quad - x(3\cdot i-2)\cdot h(1) - x(3\cdot i-3)\cdot h(2) = \\ &= P_{typeI}(\delta^C, \gamma^C, i) - P_{typeI}(\delta^D, \gamma^D, i) - P_{typeI}(\delta^A, \gamma^A, i) \\ d_1(i) &= x(3\cdot i-1)\cdot h(1) + x(3\cdot i-2)\cdot h(2) = \\ &= [x(3\cdot i-1) + x(3\cdot i-2)]\cdot[h(1)+h(2)] + \\ &\quad - x(3\cdot i-1)\cdot h(2) - x(3\cdot i-2)\cdot h(1) = \\ &= P_{typeI}(\delta^E, \gamma^E, i) - P_{typeI}(\delta^D, \gamma^D, i) - P_{typeI}(\delta^B, \gamma^B, i) \end{aligned} \tag{5.28}$$

Where

$$\begin{cases} \delta^C = 1; & \gamma^C = 1 \\ \delta^D = 1; & \gamma^D = 0 \\ \delta^E = 0; & \gamma^E = 1 \end{cases} \tag{5.29}$$

It is straightforward to verify that the linear combination of the partial values $P_{typeI}(\delta^A, \gamma^A, i)$ and $P_{typeI}(\delta^A, \gamma^B, i)$ with three additional expressions of P_{typeI} allows computing the d_{-1}, d_1 results. For this reason, the partial values $P_{typeI}(\delta^C, \gamma^C, i)$, $P_{typeI}(\delta^D, \gamma^D, i)$ and $P_{typeI}(\delta^E, \gamma^E, i)$ must be taken in account in the algorithm execution.

Finally, the d_0 can be computed as:

$$\begin{aligned} d_0(i) &= x(3\cdot i-1)\cdot h(0) + x(3\cdot i-2)\cdot h(1) + x(3\cdot i-3)\cdot h(2) = \\ &= P_{typeI}(\delta^F, \gamma^F, i) - P_{typeI}(\delta^E, \gamma^E, i) - P_{typeI}(\delta^C, \gamma^C, i) + \\ &+ 2\cdot P_{typeI}(\delta^D, \gamma^D, i) \end{aligned} \tag{5.30}$$

Where

$$\begin{cases} \delta^F = 0; & \gamma^F = 2 \end{cases} \tag{5.31}$$

By inserting the partial value $P_{typeI}(\delta^F, \gamma^F, i)$ the total amount of intermediate results can be obtained and, consequently, the algorithm is able to compute the $\bar{y}(i)$ sequence. Please note that, according to this simple example, the overall number of multiplication required is equal to 6 instead of the 9 multiplications in the standard convolution. The algorithm shown in Figure 5.1 generalizes for each FIR length k the procedure highlighted in the previous example indicating the overall partial values needed for the algorithm execution. Although the reported methodology has not been mathematically demonstrated in this dissertation, a comprehensive numerical analysis has been carried out to validate the entire algorithm for a given k value. The scripts implemented to verify the correctness of the proposed algorithm compare the outputs of the standard convolution with those obtained by the approach described in this work. Once the partial values have been enumerated with the algorithm in Figure 5.1, they can be combined to compute the intermediate results as described in the algorithm in Figures 5.2 - 5.6. Since one partial value identifies one multiplication, the required number of multiplications for the mono-dimensional convolution is:

$$RM\,(k) = \sum_{m=1}^{k} m - \sum_{j=3}^{\lceil k/2 \rceil} (j-2) \tag{5.32}$$

In this part of the dissertation, a comparison between the proposed algorithm with respect to other FFA architecture is presented. In [7], an *L-parallel* Fast FIR Algorithm is proposed, the authors introduced a novel approach to reduce the amount of multiplication required by the mono-dimensional convolution. The FFA architecture in [7] computes L different values in parallel for a given *N-tap* FIR filter. Conversely, our algorithm produces an N parallel output of the convolution, where N is equal to the depth of the filter. For these reasons, we introduce a novel Figure of Merit (FOM) defined as the required multiplication for a given algorithm divided by the number of outputs computed into a single iteration. Table 5.1 shows the comparison of the proposed algorithm with respect to [7] where L indicates the outputs computed, RM quantifies the required number of multiplication required for each algorithm.

Furthermore, the work [8] proposed a *L-parallel* Fast FIR Algorithm for the implementation of symmetric convolution based on odd length. By exploiting the filter coefficients symmetry, the authors proposed a novel FFA architecture suitable for the hardware implementation. Table 5.2 compares our work and [8]. It is worth pointing out that the algorithm in [8] is dedicated to FIR filters with a particular coefficient symmetry meanwhile we propose a general approach that is applicable to all types of filters. As can be seen, our approach achieves a better reduction of multiplications for a wide range of $N - tap$ filters by analyzing the mentioned FOM.

Outputs	N-taps	Work	RM	RM/Outputs
4	144	[7]	288	72
6	144	[7]	360	60
144	144	Proposed	7955	55.24
6	576	[7]	1440	240
9	576	[7]	1600	177.78
576	576	Proposed	125135	217.25

Table 5.1. Comparison of the proposed algorithm and the work presented in [7].

Outputs	N-taps	Work	RM	RM/Outputs
3	27	[8]	38	12.67
4	27	[8]	51	12.75
6	27	[8]	76	12.67
27	27	Proposed	300	11.11
3	81	[8]	110	36.67
4	81	[8]	149	37.25
6	81	[8]	203	33.83
81	81	Proposed	2541	31.37
3	147	[8]	198	66
4	147	[8]	261	65.25
6	147	[8]	366	61
147	147	Proposed	8250	56.12
3	591	[8]	790	263.33
4	591	[8]	1036	259
6	591	[8]	1439	239.83
591	591	Proposed	131571	222.62

Table 5.2. Comparison of the proposed algorithm and the work presented in [8].

$d_0(i) = P_{typeI}(0, k-1, i)$;
for $step = 0 : \lceil (k-3)/2 \rceil$
  $\delta_1 = k - (step+1) - \gamma_1;\ \delta_2 = k - (step+2) - \gamma_2$;
  $\gamma_1 = \gamma_2 = k - 1 - (2 \cdot step + 1)$;
  if($step$ *is even*)**then**
    $\delta_3 = k - (step+2) - \gamma_3;\ \gamma_3 = k - 1 - 2 \cdot (step+1)$;
    $d_0(i) += 2 \cdot P_{typeII}(\delta_3, \gamma_3, i) - P_{typeI}(\delta_1, \gamma_1, i) - P_{typeI}(\delta_2, \gamma_2, i)$;
  else
    $\delta_3 = k - (step+1) - \gamma_3;\ \delta_4 = k - (k - step) - \gamma_4;\ \gamma_3 = \gamma_4 = 0$;
  end if
$d_0(i) += P_{typeI}(\delta_1, \gamma_1, i) + P_{typeI}(\delta_2, \gamma_2, i) - 2 \cdot P_{typeI}(\delta_3, \gamma_3, i) + P_{typeI}(\delta_4, \gamma_4, i)$;
end for

Figure 5.2. Pseudo code for $d_0(i)$ intermediate result computation.

$\gamma_0 = k - z - 1;\ \delta_0 = 0;\ \delta_1 = k - (z+1) - \gamma_1;\ \delta_2 = k - (z+2) - \gamma_2;$
$\delta_3 = k - (z+1) - \gamma_3;\ \delta_4 = k - (z+2) - \gamma_4;$
$\delta_5 = k - (z+1) - \gamma_5;\ \gamma_1 = \gamma_2 = k - z - 2;\ \gamma_3 = \gamma_4 = k - z - 3;\ \gamma_5 = 0;$
$d_z = P_{typeI}(\delta_0, \gamma_0, i) - P_{typeII}(\delta_1, \gamma_1, i) - P_{typeI}(\delta_2, \gamma_2, i);$
$d_z += P_{typeI}(\delta_5, \gamma_5, i) - P_{typeI}(\delta_3, \gamma_3, i) - P_{typeI}(\delta_4, \gamma_4, i);$
$pos = z + 3;\ init\_step = (z-1)/2 + 1;$
for $step = init\_step : (\lfloor k/2 \rfloor - 2)$
  if($step$ *is even*)**then**
    $\delta_1 = k - pos - \gamma_1;\ \gamma_1 = k - 1 - 2 \cdot (step + 1);$
    $d_z += P_{typeII}(\delta_1, \gamma_1, i);$
    if($\gamma_1 != 1$)**then**
      $\delta_2 = k - pos - \gamma_2;\ \delta_3 = k - (pos+1) - \gamma_3;$
      $\delta_4 = (step - init\_step) + 1;\ \gamma_2 = \gamma_3 = \gamma_1 - 1;\ \gamma_4 = 0;$
      $d_z += P_{typeI}(\delta_2, \gamma_2, i) - P_{typeI}(\delta_3, \gamma_3, i) + 2 \cdot P_{typeI}(\delta_4, \gamma_4, i);$
    else
      $\delta_2 = k - pos;\ \delta_3 = k - (pos+1);\ \gamma_2 = \gamma_3 = 0;$
      $d_z += 2 \cdot P_{typeI}(\delta_2, \gamma_2, i) + P_{typeI}(\delta_3, \gamma_3, i);$
    end if
  else
    $\delta_1 = k - pos - (\gamma_1 - 1);\ \gamma_1 = 2 \cdot (step + 1);$
    $d_z += -P_{typeII}(\delta_1, \gamma_1, i);$
    if($\gamma_1 != 1$)**then**
      $\delta_2 = k - pos - \gamma_2;$
      $\delta_3 = k - (pos+1) - \gamma_3;\ \delta_4 = k - pos;\ \gamma_2 = \gamma_3 = \gamma_1 - 1;\ \gamma_4 = 0;$
      $d_z += -P_{typeI}(\delta_2, \gamma_2, i) + P_{typeI}(\delta_3, \gamma_3, i) + 2 \cdot P_{typeI}(\delta_4, \gamma_4, i);$
    else
      $\delta_2 = k - (pos+1);\ \delta_3 = k - pos;\ \gamma_2 = \gamma_3 = 0;$
      $d_z += 2 \cdot P_{typeI}(\delta_2, \gamma_2, i) + P_{typeI}(\delta_3, \gamma_3, i);$
    end if
  $pos = pos + 2;$
  end if
end for

Figure 5.3. Pseudo code for $d_z(i)$ intermediate result computation for positive odd values of z.

$c = |z|;\ pos = 1;\ \gamma_0 = k - c - 1;\ \gamma_1 = \gamma_2 = k - c - 2;\ \gamma_3 = \gamma_4 = k - c - 3;\ \gamma_5 = 0;$
$\delta_0 = \delta_5 = c;\ \delta_1 = k - pos - \gamma_1;\ \delta_2 = k - (pos + 1) - \gamma_2;$
$\delta_3 = k - (pos + 1) - \gamma_3;\ \delta_4 = k - (pos + 2) - \gamma_4;$
$d_z = P_{typeI}(\delta_0, \gamma_0, i) - P_{typeI}(\delta_1, \gamma_1, i) - P_{typeI}(\delta_2, \gamma_2, i) + P_{typeI}(\delta_3, \gamma_3, i) +$
$- P_{typeI}(\delta_4, \gamma_4, i) + P_{typeI}(\delta_5, \gamma_5, i);$
$pos = 1;\ init\_step = (c - 1)/2 + 1;$
for $step = init\_step : (\lfloor k/2 \rfloor - 2)$
  if($step\ is\ odd$)**then**
    $\delta_1 = k - pos - \gamma_1;\ \gamma_1 = k - 1 - 2 \cdot (step + 1);$
    $d_z+ = -P_{typeII}(\delta_1, \gamma_1, i);$
    if($\gamma_1 != 1$)**then**
      $\delta_2 = k - pos - (\gamma_2 - 1);\ \delta_3 = k - (pos + 1) - \gamma_3;\ \delta_4 = (step + init\_step);$
      $\gamma_3 = \gamma_2 = \gamma_1 - 1;\ \gamma_4 = 0;$
      $d_z+ = P_{typeI}(\delta_2, \gamma_2, i) - P_{typeI}(\delta_3, \gamma_3, i) + 2 \cdot P_{typeI}(\delta_4, \gamma_4, i);$
    else
      $\delta_2 = k - pos;\ \delta_3 = k - (pos + 1);\ \gamma_2 = \gamma_3 = 0;$
      $d_z+ = 2 \cdot P_{typeI}(\delta_2, \gamma_2, i) + P_{typeI}(\delta_3, \gamma_3, i);$
    end if
  else
    $\delta_1 = k - pos - \gamma_1;\ \gamma_1 = k - 2 \cdot (step + 1);$
    $d_z+ = -P_{typeII}(\delta_1, \gamma_1, i);$
    if($\gamma_1 != 1$)**then**
      $\delta_2 = k - pos - \gamma_2;$
      $\delta_3 = k - (pos + 1) - \gamma_3;\ \delta_4 = k - pos;\ \gamma_2 = \gamma_3 = \gamma_1 - 1;\ \gamma_4 = 0;$
      $d_z+ = -P_{typeI}(\delta_2, \gamma_2, i) + P_{typeI}(\delta_3, \gamma_3, i) + 2 \cdot P_{typeI}(\delta_4, \gamma_4, i);$
    else
      $\delta_2 = k - (pos + 1);\ \delta_3 = k - pos;\ \gamma_2 = \gamma_3 = 0;$
      $d_z+ = 2 \cdot P_{typeI}(\delta_2, \gamma_2, i) + P_{typeI}(\delta_3, \gamma_3, i);$
    end if
  $pos = pos + 2;$
  end if
end for

Figure 5.4. Pseudo code for $d_z(i)$ intermediate result computation for negative odd values of z.

$\delta_0 = 0;\ \gamma_0 = k - z - 1;\ d_z = P_{typeI}(\delta_0, \gamma_0, i);$
$pos = z + 1;\ init\_step = (z-1)/2 + 1;$
for $step = init\_step : (\lfloor k/2 \rfloor - 1)$
  if($step\ is\ even$)**then**
    $\delta_1 = k - pos - \gamma_1;\ \delta_2 = k - (pos+1) - \gamma_2;$
    $\gamma_1 = \gamma_2 = k - 1 - (2 \cdot step + 1);$
    $d_z += -P_{typeI}(\delta_1, \gamma_1, i) - P_{typeI}(\delta_2, \gamma_2, i);$
    if($\gamma_1 == 1$)**then**
      $\delta_3 = k - (pos+1);\ \gamma_3 = 0;$
      $d_z += 2 \cdot P_{typeI}(\delta_3, \gamma_3, i);$
    end if
  else
    $\delta_1 = k - pos\gamma_1;\ \gamma_1 = k - 1 - 2 \cdot step;$
    $d_z += 2 \cdot P_{typeII}(\delta_1, \gamma_1, i);$
    if($\gamma_1 != 1$)**then**
      $\delta_2 = k - pos - \gamma_2;\ \delta_3 = k - (pos+1) - \gamma_3;$
      $\delta_4 = k - (step - init\_step);\ \delta_5 = k - pos;$
      $\gamma_2 = \gamma_3 = k - 1 - (2 \cdot step + 1);\ \gamma_4 = \gamma_5 = 0;$
      $d_z += P_{typeI}(\delta_2, \gamma_2, i) + P_{typeI}(\delta_3, \gamma_3, i) - 2 \cdot P_{typeI}(\delta_4, \gamma_4, i) - 2 \cdot P_{typeI}(\delta_5, \gamma_5, i);$
    else
      $\delta_2 = k - (step - int\_step);\ \delta_3 = k - pos;\ \gamma_2 = \gamma_3 = 0;$
      $d_z += -P_{typeI}(\delta_2, \gamma_2, i) - P_{typeI}(\delta_3, \gamma_3, i);$
    end if
  end if
$pos = pos + 1;$
end for

Figure 5.5. Pseudo code for $d_z(i)$ intermediate result computation for positive even values of z.

$c = |z|;\ pos = 1;\ init\_step = (c-1)/2 + 1;$
$\delta_0 = k - 1 - \gamma_0; \gamma_0 = k - 1 - c;\ d_z = P_{typeI}(\delta_0, \gamma_0, i);$
for $step = init\_step : (\lfloor k/2 \rfloor - 1)$
  if(*step is even*)**then**
    $\delta_1 = k - pos - \gamma_1;\ \delta_2 = k - (pos + 1) - \gamma_2;$
    $\gamma_1 = \gamma_2 = k - 1 - (2 \cdot step + 1);$
    $d_z += -P_{typeI}(\delta_1, \gamma_1, i) - P_{typeI}(\delta_2, \gamma_2, i);$
    if($\gamma_1 == 1$)**then**
      $\delta_3 = k - (pos + 1);\ \gamma_3 = 0;$
      $d_z += 2 \cdot P_{typeI}(\delta_3, \gamma_3, i);$
    end if
  else
    $\delta_1 = k - pos - \gamma_1;\ \gamma_1 = k - 1 - 2 \cdot step;$
    $d_z += 2 \cdot P_{typeII}(\delta_1, \gamma_1, i);$
    if($\gamma_1 != 1$)**then**
      $\delta_2 = k - pos - \gamma_2;\ \delta_3 = k - (pos + 1) - \gamma_3;$
      $\delta_4 = k - (step + init\_step);\ \delta_5 = k - pos;$
      $\gamma_2 = \gamma_3 = k - 1 - (2 \cdot step + 1);\ \gamma_4 = \gamma_5 = 0;$
      $d_z += P_{typeI}(\delta_2, \gamma_2, i) + P_{typeI}(\delta_3, \gamma_3, i) - 2 \cdot P_{typeI}(\delta_4, \gamma_4, i) - 2 \cdot P_{typeI}(\delta_5, \gamma_5, i);$
    else
      $\delta_2 = k - (step + int\_step);\ \delta_3 = k - pos;\ \gamma_2 = \gamma_3 = 0;$
      $d_z += -P_{typeI}(\delta_2, \gamma_2, i) - P_{typeI}(\delta_3, \gamma_3, i);$
    end if
  end if
$pos = pos + 1;$
end for

Figure 5.6. Pseudo code for $d_z(i)$ intermediate result computation for negative even values of z.

5.3.2 Bi-dimensional approach

The output $y(n_1, n_2)$ of the bi-dimensional convolution between an infinite two-dimensional input sequence $x(n_1, n_2)$ and a k by k discrete

filter $h(n_1, n_2)$ can be expressed as:

$$y(n_1, n_2) = \sum_{i_1=0}^{k-1} \sum_{i_2=0}^{k-1} x\left(n_1 - i_1, n_2 - i_2\right) \cdot h\left(i_1, i_2\right) \tag{5.33}$$

The proposed algorithm elaborates, at each iteration, an input sequence with k rows and k columns as:

$$\bar{\bar{x}}\left(i_1, i_2\right) = x\left[\left(k \cdot i_1 - 1\right) : \left(k \cdot \left(i_1 - 1\right)\right) \ ; \ \left(k \cdot i_2 - 1\right) : \left(k \cdot \left(i_2 - 1\right)\right)\right] \tag{5.34}$$

and computes k rows and k columns if the output sequence as :

$$\bar{\bar{y}}\left(i_1, i_2\right) = y\left[\left(k \cdot i_1 - 1\right) : \left(k \cdot \left(i_1 - 1\right)\right) \ ; \ \left(k \cdot i_2 - 1\right) : \left(k \cdot \left(i_2 - 1\right)\right)\right] \tag{5.35}$$

where i_1 and i_2 indicate the iteration of the algorithm. By reiterating the reasoning of the mono-dimensional algorithm, we introduce four intermediate results for the bi-dimensional version of the proposed algorithm as:

$$d_{-z_1,-z_2} = \sum_{j_1=0}^{k-z_1-1} \sum_{j_2=0}^{k-z_2-1} x\left(k \cdot i_1 - 1 - (j_1 + z_1), k \cdot i_2 - 1 - (j_2 + z_2)\right) h(j_1, j_2) \tag{5.36}$$

$$d_{-z_1,z_2} = \sum_{j_1=0}^{k-z_1-1} \sum_{j_2=z_2}^{k-1} x\left(k \cdot i_1 - 1 - (j_1 + z_1), k \cdot i_2 - 1 - (j_2 - z_2)\right) h(j_1, j_2) \tag{5.37}$$

$$d_{z_1,-z_2} = \sum_{j_1=z_1}^{k-1} \sum_{j_2=0}^{k-z_2-1} x\left(k \cdot i_1 - 1 - (j_1 - z_1), k \cdot i_2 - 1 - (j_2 + z_2)\right) h(j_1, j_2) \tag{5.38}$$

$$d_{z_1,z_2} = \sum_{j_1=z_1}^{k-1} \sum_{j_2=z_2}^{k-1} x\left(k \cdot i_1 - 1 - (j_1 - z_1), k \cdot i_2 - 1 - (j_2 - z_2)\right) h(j_1, j_2) \tag{5.39}$$

where $z_1, z_2 \in [0, 1, \cdots, k-1]$. As seen in the previous section, it can be proved that the output sequence of the convolution (5.35) can be computed

by means of the linear combination of intermediate results (5.36)-(5.39) as shown in the following equations. Considering the row $r_{k-1} = k \cdot i_1 - 1$ (the last row of $\bar{\bar{y}}(i_1, i_2)$) of the Eq.(5.35), it is straightforward to verify that:

$$\begin{aligned} y(r_{k-1}, k \cdot (i_2 - 1)) &= d_{0,-(k-1)}(i_1, i_2) + d_{0,1}(i_1, i_2 - 1) \\ &\vdots \\ y(r_{k-1}, k \cdot i_2 - 2) &= d_{0,-1}(i_1, i_2) + d_{0,k}(i_1, i_2 - 1) \\ y(r_{k-1}, k \cdot i_2 - 1) &= d_{0,0}(i_1, i_2) \end{aligned} \tag{5.40}$$

Meanwhile considering the row $r_{k-2} = r_{k-1} - 1$ we can establish that:

$$\begin{aligned} &y(r_{k-2}, k \cdot (i_2 - 1)) = d_{-1,-(k-1)}(i_1, i_2) + d_{k-1,-(k-1)}(i_1 - 1, i_2) + \\ &+ d_{-1,1}(i_1, i_2 - 1) + d_{k-1,1}(i_1 - 1, i_2 - 1) \\ &\vdots \\ &y(r_{k-2}, k \cdot (i_2 - 2)) = d_{-1,-1}(i_1, i_2) + d_{k-1,-1}(i_1 - 1, i_2) + \\ &+ d_{-1,k-1}(i_1, i_2 - 1) + d_{k-1,k-1}(i_1 - 1, i_2 - 1) \\ &y(r_{k-2}, k \cdot i_2 - 1) = d_{-1,0}(i_1, i_2) + d_{k-1,0}(i_1 - 1, i_2) \end{aligned} \tag{5.41}$$

Finally, considering the row $r_0 = r_{k-1} - k$ of the Eq.(5.35), we can demonstrate that:

$$\begin{aligned} &y(r_0, k \cdot (i_2 - 1)) = d_{-(k-1),-(k-1)}(i_1, i_2) + d_{1,-(k-1)}(i_1 - 1, i_2) + \\ &+ d_{-(k-1),1}(i_1, i_2 - 1) + d_{1,1}(i_1 - 1, i_2 - 1) \\ &\vdots \\ &y(r_0, k \cdot (i_2 - 2)) = d_{-(k-1),-1}(i_1, i_2) + d_{1,-1}(i_1 - 1, i_2) + \\ &+ d_{-(k-1),k-1}(i_1, i_2 - 1) + d_{1,k-1}(i_1 - 1, i_2 - 1) \\ &y(r_0, k \cdot i_2 - 1) = d_{-(k-1),0}(i_1, i_2) + d_{1,0}(i_1 - 1, i_2) \end{aligned} \tag{5.42}$$

The methodology to compute all the partial values for the k *by* k convolution is based on the mono-dimensional approach. The values P_{typeI} and P_{typeII} in Eq. (5.16) and (5.19) are computed starting from a linear combination of elements of the sequence $\bar{x}(i)$ and $h(n)$. The parameters δ

and γ define the indexes of the elements of $\bar{x}(i)$and $h(n)$ that are used in each partial value. In the bi-dimensional case, we need to find the indexes of the rows and the columns of each element of $\bar{\bar{x}}\,(i_1, i_2)$ and $h(n_1, n_2)$ that are used in the definition of the partial values. We have found that the same values of δ and γ of the mono-dimensional case can be reused in the bi-dimensional convolution as shown in the following example. In order to compute d_0 and d_1 of the mono-dimensional case, for $k = 3$, we need to compute:

$$d_0(i) = P_{typeI}(\delta^1, \gamma^1, i) - P_{typeI}(\delta^2, \gamma^2, i) - P_{typeI}(\delta^3, \gamma^3, i) + \\ + 2 \cdot P_{typeI}(\delta^4, \gamma^4, i) \tag{5.43}$$

$$d_1(i) = P_{typeI}(\delta^2, \gamma^2, i) - P_{typeI}(\delta^4, \gamma^4, i) - P_{typeI}(\delta^5, \gamma^5, i) \tag{5.44}$$

Where

$$\begin{cases} \delta^1 = 0; & \gamma^1 = 2 \\ \delta^2 = 0; & \gamma^2 = 1 \\ \delta^3 = 1; & \gamma^3 = 1 \\ \delta^4 = 1; & \gamma^4 = 0 \\ \delta^5 = 0; & \gamma^5 = 0 \end{cases} \tag{5.45}$$

The intermediate result $d_{0,1}$ of the bi-dimensional case, can be written as a linear combination of bi-dimensional partial values that use the same parameters of Eq.(5.45). We introduce four expressions of bi-dimensional partial values. The first one can be computed as:

$$P_{typeI,I}(\delta^{'}, \gamma^{'}, \delta, \gamma, i_1, i_2) = \sum_{j_1=\delta^{'}}^{\delta^{'}+\gamma^{'}} \sum_{j_2=\delta}^{\delta+\gamma} x(k \cdot i_1 - 1 - j_1, k \cdot i_2 - 1 - j_2) \cdot \\ \cdot \sum_{j_3=k-(\delta^{'}+\gamma^{'})-1}^{k-\delta^{'}-1} \sum_{j_4=k-(\delta+\gamma)-1}^{k-\delta-1} h(k \cdot i_1 - 1 - j_3, k \cdot i_2 - 1 - j_4) \tag{5.46}$$

Where

$$\delta^{'}, \gamma^{'}, \delta, \gamma \in [0, k-1] \tag{5.47}$$

As can be seen, $P_{typeI,I}$ is computed using only contiguous values of rows and columns of the input sequence $\bar{\bar{x}}(i_1, i_2)$ and the coefficient sequence of the filter $h(n_1, n_2)$.As for the mono-dimensional approach, each value of $P_{typeI,I}$ can be computed with one single multiplication.

The second partial value can be expressed as:

$$P_{typeII,I}(\delta^{'}, \gamma^{'}, \delta, \gamma, i_1, i_2) = \sum_{j_1=0}^{1} \sum_{j_2=\delta}^{\delta+\gamma} x(k \cdot i_1 - 1 - s(j_1), k \cdot i_2 - 1 - j_2) \cdot$$
$$\cdot \sum_{j_3=0}^{1} \sum_{j_4=k-(\delta+\gamma)-1}^{k-\delta-1} h(k \cdot i_1 - 1 - u(j_3), k \cdot i_2 - 1 - j_4) \tag{5.48}$$

Where

$$s(j) = \begin{cases} \delta^{'} & if \quad j = 0 \\ \delta^{'} + \gamma^{'} & if \quad j = 1 \end{cases}$$
$$u(j) = \begin{cases} k - \delta^{'} - 1 & if \quad j = 0 \\ k - (\delta^{'} + \gamma^{'}) - 1 & if \quad j = 1 \end{cases} \tag{5.49}$$

The latter takes into account only two rows of $\bar{\bar{x}}(i_1, i_2)$ and $h(n_1, n_2)$ whereas contiguous elements of the two rows are processed.

The third value required by this example is computed as:

$$P_{typeI,II}(\delta^{'}, \gamma^{'}, \delta, \gamma, i_1, i_2) = \sum_{j_1=\delta^{'}}^{\delta^{'}+\gamma^{'}} \sum_{j_2=0}^{1} x(k \cdot i_1 - 1 - j_1, k \cdot i_2 - 1 - t(j_2)) \cdot$$
$$\cdot \sum_{j_3=k-(\delta^{'}+\gamma^{'})-1}^{k-\delta^{'}-1} \sum_{j_4=0}^{1} h(k \cdot i_1 - 1 - j_3, k \cdot i_2 - 1 - v(j_4)) \tag{5.50}$$

Where

$$t(j) = \begin{cases} \delta & if \quad j = 0 \\ \delta + \gamma & if \quad j = 1 \end{cases}$$
$$v(j) = \begin{cases} k - \delta - 1 & if \quad j = 0 \\ k - (\delta + \gamma) - 1 & if \quad j = 1 \end{cases} \tag{5.51}$$

Conversely, this expression elaborates consecutive rows of $\bar{\bar{x}}(i_1, i_2)$ and $h(n_1, n_2)$. For each row, only two column elements are processed.

Finally, the fourth partial value can be obtained as:

$$P_{typeII,II}(\delta^{'}, \gamma^{'}, \delta, \gamma, i_1, i_2) = \sum_{j_1=0}^{1} \sum_{j_2=0}^{1} x(k \cdot i_1 - 1 - s(j_1), k \cdot i_2 - 1 - t(j_2)) \cdot$$
$$\cdot \sum_{j_3=0}^{1} \sum_{j_4=0}^{1} h(k \cdot i_1 - 1 - u(j_3), k \cdot i_2 - 1 - v(j_4)) \tag{5.52}$$

Here we consider only two rows of $\bar{\bar{x}}(i_1, i_2)$ and $h(n_1, n_2)$.For each row, only two column elements are processed. By using the bi-dimensional partial values above described, the intermediate result $d_{0,1}$ for the single iteration of the algorithm can be evaluated as:

$$\begin{aligned} & d_{0,1}(i_1, i_2) = \\ & \{P_{typeI,I}(\delta^1, \gamma^1, \delta^2, \gamma^2) - P_{typeI,I}(\delta^1, \gamma^1, \delta^4, \gamma^4) - P_{typeI,I}(\delta^1, \gamma^1, \delta^5, \gamma^5)\} + \\ & - \{P_{typeI,I}(\delta^2, \gamma^2, \delta^2, \gamma^2) - P_{typeI,I}(\delta^2, \gamma^2, \delta^4, \gamma^4) - P_{typeI,I}(\delta^2, \gamma^2, \delta^5, \gamma^5)\} + \\ & - \{P_{typeI,I}(\delta^2, \gamma^2, \delta^2, \gamma^2) - P_{typeI,I}(\delta^3, \gamma^3, \delta^4, \gamma^4) - P_{typeI,I}(\delta^3, \gamma^3, \delta^5, \gamma^5)\} + \\ & + 2 \cdot \{P_{typeI,I}(\delta^4, \gamma^4, \delta^2, \gamma^2) - P_{typeI,I}(\delta^4, \gamma^4, \delta^4, \gamma^4) - P_{typeI,I}(\delta^4, \gamma^4, \delta^5, \gamma^5)\} \end{aligned} \tag{5.53}$$

In conclusion, the overall intermediate values for the bi-dimensional version of the proposed algorithm can be obtained by means of the usage, in recursive mode, of the mono-dimensional algorithm. The number of multiplications required for the bi-dimensional convolution is

$$RM_{2D}(k) = (RM(k))^2 \tag{5.54}$$

Chapter 6

Hardware Architecture and Test Chip

This Chapter describes the architecture of a test chip, fabricated in a 28nm TSMC CMOS technology, that has been developed during the PhD. The test chip includes a hardware data path that aims to deploy the bi-dimensional FFA-based algorithm, described in section 5.3.2, by means of digital circuits. In addition, a Built In Self Test (BIST) logic has been integrated into the test chip architecture. The Chapter is organized as follows: the test chip block diagram is discussed in section 6.1 meanwhile the experimental results are listed in section 6.2.

6.1 Test Chip

A test chip has been fabricated in 28nm TSMC CMOS technology, the latter includes a hardware accelerator for neural networks based on the bi-dimensional version of our algorithm. We developed a digital circuit for the hardware implementation of a well-known neural network such as the Alex-Net. This network has been selected since it is one of the first deep convolutional neural networks including up to five hidden layers with an impressive depth of the filter. Therefore, the amount of multiplications required by this kind of CNN is considerable. In addition, it must be noted that each Alex-Net convolutional layer computes a huge quantity of multiplication operations making this neural network an optimal choice

Layer	Kernel	Multiplications required	RM_{2D} per iteration*
1	11x11	$105x10^6$	3136
2	5x5	$223x10^6$	196
3	3x3	$149x10^6$	36
4	3x3	$112x10^6$	36
5	3x3	$74x10^6$	36

Table 6.1. Alex-Net multiplications required by the convolutional layers. *Number of required multiplication by the proposed algorithm to complete a single iteration (i_1, i_2).

to underline the advantages introduced by the novel FFA-based algorithm proposed in this dissertation. Details about the Alex-Net convolutional layers are provided in Table 6.1 highlighting the overall number of multiplications required by the proposed FFA algorithm.

As can be seen, the Alex-Net neural network is characterized by five convolutional layers using kernels of different dimensions. For this reason, the proposed hardware accelerator must be *reconfigurable* in order to be used on each layer. A block diagram representation of the proposed data path is reported in Figure 6.1.

The architecture can be divided into three main blocks and a dedicated logic for reconfiguration purposes.

More specifically, the hardware data path is composed of:

- Pre-processing block: this part of the architecture elaborates the input sequences $\bar{\bar{x}}(i_1, i_2)$. It has the task of computing all the sum operations needed to obtain the partial values in Eq.(5.46)-(5.52). To this end, the input sequence is routed to the adders layer through a series of reconfigurable multiplexers directly driven by the FSM circuit within the control logic block.

- Multipliers block: this circuit processes the data coming from the pre-precessing block through 42 custom circuits, named Convolutional Multiplier Unit (CMU). The latter is in charge of producing the bi-dimensional partial values, the Alex-Net coefficients are stored in a dedicated ROM within each CMU instance.

- Post-processing block: here the data path performs the linear combi-

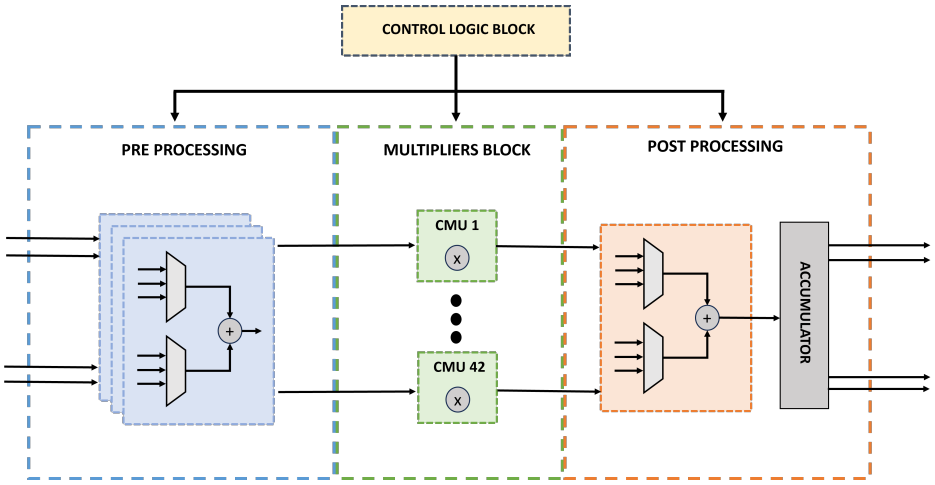


Figure 6.1. Block diagram of the data path for the Alex-Net hardware acceleration.

nation of the partial values thus generating the intermediate results as in Eq. (5.36)-(5.39). This block integrates also an accumulator circuit that is in charge of storing the d_{z_1,z_2} values for the future iteration of the novel algorithm proposed. Overall, the post-processing block computes the output sequences $\bar{\bar{y}}(i_1, i_2)$ of the convolutional layer.

- Control logic block: It is composed of one FSM able to generate the overall control signals needed for the circuits above described.

The architecture of the Convolutional Multiplier Unit is reported in Figure 6.2. This circuit contains one multiplier to elaborate the processed input sequence and the filter coefficients of the neural network thus computing the partial values of the algorithm according to the layer configuration. It is worth pointing out that the CMU instantiates a ROM memory, the latter stores the overall coefficients of the Alex-Net neural network also including the linear combination of the $\bar{\bar{h}}(i_1, i_2)$ required for the computation of the bi-dimensional version of the proposed algorithm. The data path integrates 42 instances of the CMUs, and consequently, an equivalent number of multipliers. The number of CMUs has been selected with

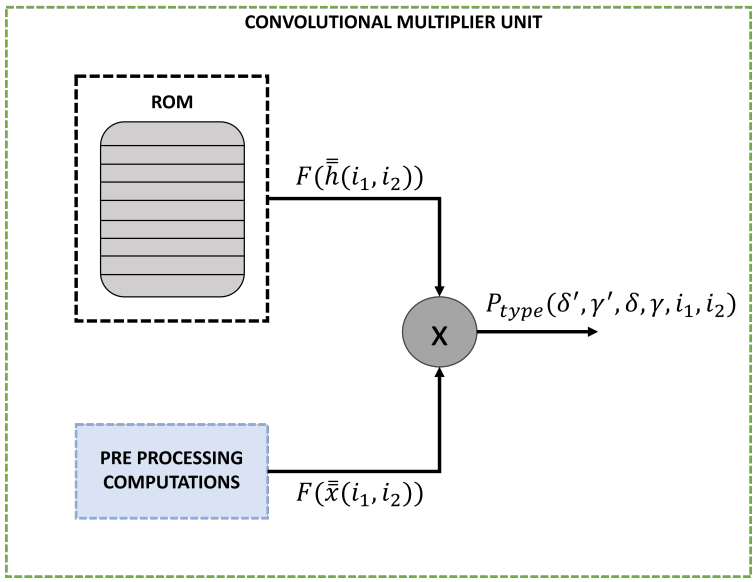


Figure 6.2. Overview of the Convolutional Multiplier Unit architecture.

the aim of maximizing the ratio between the number of operations computed by the accelerator and the amount of hardware resources used. For this purpose, we can define the Multiplier Idle Time (MIT) as the metric that measures the mean amount of time each multiplier is not used to compute a valid result. As an example, the datapath of Figure 6.1 can compute the single iteration on the proposed algorithm applied to layer 1 of the AlexNet in a time equal to 75 clock cycles. In this time the circuit computes 3136 multiplications. However, the datapath would be able to compute $42 \cdot 75 = 3150$ multiplications. The mean number of clock cycles that a multiplier is in idle state can hence be computed as:

$$MIT = \frac{3150 - 3136}{3136} \tag{6.1}$$

Table 6.2 shows the number of clock cycles per iteration and the MIT for the architecture in Figure 6.1.

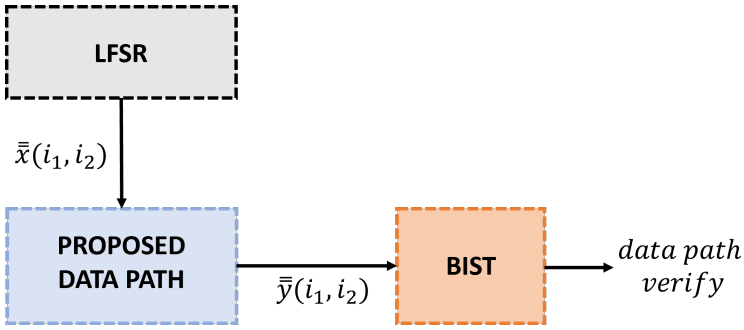


Figure 6.3. Block diagram of the test chip developed for the data path verification.

Layer	Kernel	Clock cycles per iteration	MIT
1	11x11	75	0.44 %
2	5x5	5	7.14 %
3	3x3	1	16.67 %
4	3x3	1	16.17%
5	3x3	1	16.17%

Table 6.2. Clock cycle computation per single iteration and MIT evaluation.

It is worth pointing out that for this dissertation the author could not find an algorithm for the minimization of the MIT. The number 42 has been selected as a good trade-off between hardware complexity and latency of the architecture. The proposed hardware accelerator has been tested by using dedicated digital components inside the test chip architecture as depicted in Figure 6.3. The input sequence $\bar{\bar{x}}(i_1, i_2)$ is generated by a Linerar Feedback Shift Register, subsequently, the proposed hardware accelerator elaborates the overall input data thus producing the output sequence of the convolution $\bar{\bar{y}}(i_1, i_2)$, according to the FFA-based algorithm discussed in this dissertation. Finally, the data path computations are processed by a Built In Self Test circuit in order to verify the correctness of the data path behavior.

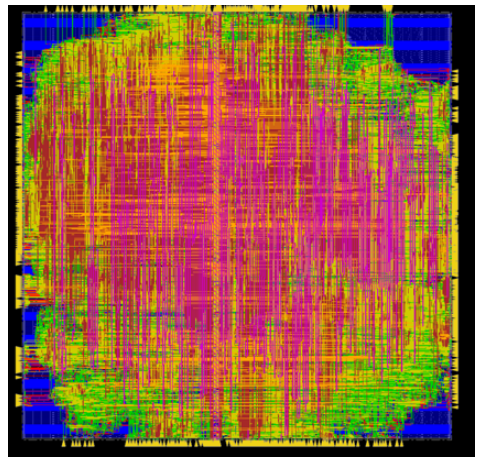


Figure 6.4. Layout of the implemented test chip.

6.2 Experimental Results

The test chip has been synthesized in 28nm TSMC CMOS, and the IC layout is reported in Figure 6.4

The IC performance is reported in the Table 6.3.

Technology	Area (mm^2)	Power dissipation (mW)	Frequency (MHz)
TSMC CMOS 28 nm	0.152	68.24	500

Table 6.3. IC performance.

A PCB board has been developed for testing purposes. The measurement setup includes the usage of Tektronix TLA pattern generator/logic analyzer for the setting of chip inputs, Tektronix TDS 7704B oscilloscope for signal visualization, Tektronix DG2040 Data Generator for the clock, and Keithley 2400 SourceMeter for the voltage supply and the power assessment. With regards to the mentioned test chip, the operational frequency of the circuit is equal to *500 MHz*. The verification operations carried out demonstrate that, for each layer of the Alex-Net neural net-

Layer	Kernel	Multiplications required from standard convolution	Multiplications required from proposed algorithm	Proposed Datapath Latency (clock cycles)
1	11x11	$105x10^6$	$76x10^6$	$0.8x10^6$
2	5x5	$223x10^6$	$143x10^6$	$1.75x10^6$
3	3x3	$149x10^6$	$115x10^6$	$1.35x10^6$
4	3x3	$112x10^6$	$86x10^6$	$1x10^6$
5	3x3	$74x10^6$	$57x10^6$	$0.7x10^6$

Table 6.4. IC performance related to Alex-Net computations.

work, the results of the data path are consistent with the expected output sequences of the convolution.

Finally, IC performance related to Alex-Net computation is reported in Table 6.4. As can be seen, the overall computations of the neural network are reduced by implementing the proposed bi-dimensional algorithm through the development of dedicated hardware circuits.

Chapter 7

Conclusions

The central theme of this dissertation is the analysis and development of innovative techniques for digital verification of hardware accelerator based on SoC platform.

In this regard, a fruitful collaboration with Micron Technology has been signed allowing us to work on the digital verification process of a complex SoC device and the state-of-the-art storage system based on the UFS 4.0 standard.

An ambitious RTL simulation environment for a Micron custom SoC system has been developed within the PhD work. The latter environment is fully compliant with the Universal Verification Methodology, the latest verification guideline, accepted by IEEE as a standard, that includes the best practices for the validation of digital systems.

There are countless advantages introduced by the proposed UVM-based environment in terms of flexibility, robustness, and maintenance of the overall architecture. During the implementation phase, we paid close attention to the regression flow improvements; indeed, a novel *save and restart* technique has been implemented which aims to reduce the overall time consumption of a regression list execution. More specifically, the proposed technique aims to store a simulation snapshot of the configuration and/or initialization sequences common for a given test list. Then, to simulate the entire list, each test starts from the common snapshot thus reducing significantly the overall time consumption with respect to the previous regression management flow in Micron company.

Nevertheless, the RTL software verification shows different drawbacks, in particular for complex designs under test the software simulation time increases drastically resulting in a low level of coverage reachable. In addition, by analyzing only the RTL code of the Device Under Test it is not possible to evaluate implementation defects introduced by synthesizer tools.

In this dissertation, two innovative emulation techniques for the hardware acceleration of the digital verification process have been carried out. Both of them have been successfully applied for the evaluation of a digital system developed by Micron. The latter system includes several hardware accelerators synthesized in Programmable Logic of a top-level SoC device, one external ASIC circuit, and, finally, a UFS 4.0 device. All of these components are integrated within a custom PCB board and inserted into a PC desktop by means of PCIe bus.

The first technique, named *system level emulation* in this dissertation, allows us to generate, through an automated process, a Python test starting from the proposed UVM simulation tool for the Micron solution. Then, the Python script is directly executed on the Micron hardware platform allowing the evaluation of a wide range of scenarios that can not be analyzed by the software approach due to the testing time needed.

The second technique, named *low-level emulation* in this work, implements a more general approach for the emulation by introducing a hardware version of the UVM agents. These circuits, as well as additional control logic, are synthesized inside the SoC Programmable Logic thus managing the DUT interfaces. Since the usage of an operating system is not required with this approach, the implemented hardware emulation solution provides direct control of the overall latencies of the Micron system during the onboard test execution.

Another relevant theme of this work is the implementation of efficient architectures for hardware acceleration of neural networks. As far as the neural network is concerned, the major operation computed is the convolution, where conventionally several thousands of multiplications are performed by the inner layers. Some applications, such as automotive platforms, multimedia processing, and IoT solutions require the convolution to be implemented through dedicated hardware accelerators. Conventionally, an efficient hardware architecture for neural network acceleration aims to

reduce the overall number of the multiplication required by the convolutional layers.

For this reason, a novel FFA algorithm for the mono- and bi-dimensional convolution has been presented in this dissertation. The proposed algorithm explores the partial results reuse aiming at reducing the overall multiplication needed. On the basis of the mentioned algorithm, a hardware data path has been developed for the acceleration of a well-known Convolutional Neural Network as the Alex-Net which was one of the first deep neural networks presented in Literature.

Hence, the proposed data path implements the bi-dimensional version of the novel FFA algorithm. Since the Alex-Net is characterized by five convolutional layers with different kernel sizes, the implemented architecture is reconfigurable in order to adapt the data path computations to the desired kernel configuration. By developing in hardware the proposed algorithm, the architecture presented in this work is able to reduce the amount of multiplication for each convolutional layer of the Alex-Net.

The IC has been synthesized in 28nm TSMC CMOS technology with a maximum frequency equal to *500 MHz*. In addition, a test chip has been fabricated that includes a BIST logic to verify the correctness of the developed data path. A PCB board has been developed to perform experimental analysis and the test chip has been validated by utilizing a pattern generator, logic analyzer, source meters, and oscilloscope.

Bibliography

[1] IEEE. *IEEE Standard for Universal Verification Methodology Language Reference Manual. IEEE Std 1800.2-2020 (Revision of IEEE Std 1800.2-2017)*, pages 1–458, 2020.

[2] Accellera Systems Initiative. *UVM User Guide.* https://www.accellera.org/downloads/standards/uvm, 2012.

[3] Accellera Systems Initiative. *UVM Reference Manual.* https://www.accellera.org/downloads/standards/uvm, 2012.

[4] Janick Bergeron. *Writing Testbenches using SystemVerilog.* Spinger, 2006.

[5] Sharon Rosenberg and Kathleen Meade. *A Practical Guide to Adopting the Universal Verification Methodology (UVM),Second Edition.* Cadence Design System, 2012.

[6] Jadec. *Universal Flash Storage (UFS).* https://www.jedec.org/standards-documents/focus/flash/universal-flash-storage-ufs, 2020.

[7] Chao Cheng and Keshab K. Parhi. *Hardware Efficient Fast Parallel FIR Filter Structures Based on Iterated Short Convolution. IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS–I: REGULAR PAPERS*, 51(8):1492–1500, AUGUST 2004.

[8] Yu chi Tsao and Ken Choi. *Area-Efficient Vlsi Implementation For Parallel Linear-Phase Fir Digital Filters of Odd Length Based On Fast Fir Algorithm. IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS–II: EXPRESS BRIEFS*, 59(6):371–375, JUNE 2012.

[9] Jichen Wang,Jun Lin,and Zhongfeng Wang. *Efficient Hardware Architectures for Deep Convolutional Neural Network. IEEE TRANSACTIONS*

ON CIRCUITS AND SYSTEMS–I: REGULAR PAPERS, 65(9):1941–1953, JUNE 2018.

[10] Chao Cheng and Keshab K. Parhi. *Fast 2D Convolution Algorithms for Convolutional Neural Networks. IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS–I: REGULAR PAPERS*, 67(5):1678–1691, MAY 2020.

[11] Yizhi Wang,Jun Lin,and Zhongfeng Wang. *FPAP: A Folded Architecture for Energy-Quality Scalable Convolutional Neural Networks. IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS–I: REGULAR PAPERS*, 66(1):288–301, JANUARY 2019.

[12] Huizheng Wang,Weihong Xu,Zaichen Zhang,Xiaohu You,Chuan Zhang. *An Efficient Stochastic Convolution Architecture Based on Fast FIR Algorithm. IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS–II: EXPRESS BRIEFS*, 69(3):984–988, MARCH 2022.

[13] ARM Developer. *AMBA AXI Protocol specification.* https://developer.arm.com/documentation/ihi0011/a?lang=en, 2023.

Author's Publications

M.Vitone,N.Petra,"Reconfigurable Datapath for Hardware Acceleration of Convolutional Neural Network",52nd *Annual Meeting of Associazione Società Italiana di Elettronica (SIE)*,2021