Functional and timing in-hardware verification of FPGA-based designs using unit testing frameworks
UNIVERSITY OF CASTILLA-LA MANCHA
HIGHER SCHOOL OF COMPUTER ENGINEERING
DEPARTMENT OF TECHNOLOGY AND INFORMATION SYSTEMS

Functional and timing in-hardware verification of FPGA-based designs using unit testing frameworks

Ph.D. Dissertation

Author:
D. Julián Caba Jiménez

Supervisors:
Prof. Dr. Fernando Rincón Calle
Dr. Julio Daniel Dondo Gazzano

Ciudad Real, 2018
Functional and timing in-hardware verification of FPGA-based designs using unit testing frameworks

A dissertation submitted to the Department of Technology and Information Systems of Castilla-La Mancha University in fulfillment of the requirements for the degree of Philosophy Doctor

UNESCO Codes: 330406, 330703, 331117 and 330793
Functional and timing in-hardware verification of FPGA-based designs using unit testing frameworks

Ph.D. Dissertation

Author:

Julián Caba Jiménez
Computer Engineer
University of Castilla-La Mancha

Supervisors:

Prof. Dr. Fernando Rincón Calle
Ph.D. in Informatics Engineering
University of Castilla-La Mancha

Dr. Julio Daniel Dondo Gazzano
Ph.D. in Informatics Engineering
University of Castilla-La Mancha

Ciudad Real, 2018
Acknowledgments

Tal vez esta es la parte más complicada de escribir, cuando en unas cuantas líneas he de nombrar a tanta gente que directa o indirectamente ha participado en esta tesis. Perdón si me «olvido» de alguno.

En primer lugar, me gustaría agradecer el apoyo de mis directores. Estoy especialmente agradecido a Fernando porque desde el principio confió en mí, su constancia en el trabajo y su pasión por la docencia es sin duda alguna una gran referencia. Julio tiene gran culpa de todo esto, él despertó en mi el «gusanillo» de algo llamado investigación. También me gustaría agradecer a Juan Carlos la oportunidad que me ha ofrecido de formar parte de esta «pequeña familia» llamada ARCO.

Tengo que agradecer a todos los miembros del grupo ARCO (prefiero no nombrar a nadie para no dejarme ningún nombre en el tintero), tanto los que actualmente están como los que decidieron tomar otro camino. Todos ellos han aportado inquietudes en mi día a día y han contribuido a tener un ambiente de trabajo muy agradable. También gracias por esos «momentos lúdicos/sociales» que hemos podido disfrutar y conocernos algo más.

A mis padres por permitirme estudiar lo que me apasiona y haberme dejado caer y levantarme sólo, gracias por el esfuerzo que habéis realizado, ahora recogéis aquello que hace tiempo sembrasteis. A mi hermana, porque también tiene parte de «culpa» de todo esto. Intencionadamente dejé a María José, mi mujer, para la última, ha sido la primera que ha sufrido las consecuencias de esta tesis, gracias por levantarme el ánimo en esos días «difíciles», por estar a mi lado y por apoyarme en mis decisiones, espero seguir disfrutando de nuevas aventuras junto a ti.

Last but not least, I wish to thank Professor João Manuel Paiva Cardoso from FEUP (Porto, Portugal) for his support in the accomplishment of this dissertation, patience and immense knowledge.

This dissertation is partially supported by Ministry of Economy and Competitiveness of Spain under projects REBECCA (TEC2014-58036-C4-1R) and DREAMS (TEC2011-28666-C04-04), and European Regional Development Fund under project SAND (PEII-2014-046-P).
This thesis is dedicated to:

All people who ask me about it, particularly
My wife, my sister and my parents

«Learn from yesterday, live for today, hope for tomorrow.
The important thing is not to stop questioning»

Albert Einstein
Abstract

Nowadays, high-level modelling is becoming more and more popular to build new hardware designs, providing an early understanding of the design impact decisions, and allowing a more effective design space exploration, which results in a higher design productivity and improves the likelihood of finding the optimal implementation. However, the verification stage still entails an amount of non-trivial problems, such as the following: the trade-off between simulation effort and simulation accuracy completely depends on the design abstraction levels; each testing-level stage induces rewriting tests, which is time-consuming and prone to human errors; the time spent in verification accounts for roughly 60% of the development life-cycle. This task is therefore considered as the bottleneck of most projects; the synthesis consumption-time of a hardware design is too high and has third-party dependencies.

This dissertation proposes a hardware verification framework using the new generation tools provided by FPGA vendors, whose verification accuracy is close to a real scenario, considering functional and timing factors. In addition, a transparent and remote testing service is provided to automate the verification stage. This service is composed by a hardware platform where a Design Under Test (DUT) is deployed into a dynamically reconfigurable area. DUTs are generated using High-Level Synthesis (HLS) tools, and are verified through unit testing, checking its behavioural and timing correctness. These tests are the same at any abstraction level. The testing process is transparently automated; an engineer commits his design code and unit tests written in a high-level language, such as C, into a repository, and automatically the testing service is able to synthesise the design code, deploy the DUT remotely into a Field-Programmable Gate Array (FPGA) and exercise it with the original unit tests, reporting the testing result to the engineer. In addition, we provide some facilities to reduce third-party dependencies and to increase the intermediate results.

**Keywords:** verification, FPGA, unit testing, co-verification, in-hardware verification.
Resumen

Hoy en día, el modelado de alto nivel está cada vez más presente en la construcción de nuevos diseños hardware, proporcionando una rápida comprensión del impacto que tienen algunas decisiones de desarrollo y permitiendo una exploración del espacio de diseño más efectiva que permita encontrar una solución óptima. Sin embargo, la etapa de verificación sigue arrastrando una serie de problemas no triviales, entre los cuales se pueden encontrar: el balance entre el esfuerzo de simulación y la precisión de esta depende directamente del nivel de abstracción del diseño; cada nivel de abstracción supone una re-escritura de los tests, que conlleva a nuevos errores y a consumir tiempo del desarrollador; el tiempo empleado en la etapa de verificación está en torno al 60 por ciento del total del desarrollo del producto, considerándose esta etapa como el cuello de botella de la mayoría de proyectos; el tiempo de síntesis de un diseño es muy elevado; dependencia con terceros.

Esta tesis propone un framework de verificación haciendo uso de la nueva generación de herramientas que los proveedores de FPGAs proporcionan, cuya precisión es cercana a un escenario real, y considerando dos factores: comportamiento y tiempo. Además, un servicio transparente y remoto de pruebas es ofrecido para la automatización de la etapa de verificación. Este servicio está compuesto por una plataforma hardware donde el diseño a probar (DUT) es desplegado en una área dinámicamente reconfigurable. Estos diseños son descritos en lenguajes de alto nivel y sintetizados mediante herramientas de síntesis de alto nivel (HLS), y verificadas mediante el uso de test unitarios, que comprueban tanto su comportamiento funcional como temporal. Dichos tests son los mismos en cualquier nivel de abstracción. Además, el proceso de verificación es automatizado; el desarrollador sólo deberá subir los fuentes de su diseño junto con los tests descritos en algún lenguaje de alto nivel, como es C, en un repositorio, y automáticamente el servicio de verificación es capaz de sintetizar el diseño, desplegarlo de forma remota en una FPGA y estimular el DUT con los tests originales, reportando el resultado de la ejecución al desarrollador. Además, se proporcionan algunas herramientas o técnicas que permiten reducir las dependencias con terceros e incrementar la visibilidad de resultados intermedios.

Keywords: verificación, FPGA, unit testing, co-verificación, verificación en hardware.
## Contents

1 Introduction .............................................. 1
   1.1 Motivation and Problem Overview ................. 2
   1.2 Hypothesis ........................................ 10
   1.3 Thesis Objectives .................................. 10
      1.3.1 Mainstream Objective ....................... 11
      1.3.2 Other Objectives ............................ 11
   1.4 Thesis Contributions .............................. 12
   1.5 Thesis Outline ................................... 13

2 Background and Related Work ......................... 15
   2.1 Verification Challenges ........................... 16
   2.2 Functional Verification ............................ 17
      2.2.1 Functional Verification of FPGA-based designs .. 19
   2.3 Methodologies and tools .......................... 20
      2.3.1 Assertion-Based Verification ................. 20
      2.3.2 Universal Verification Methodology ........... 23
3 Hardware Unit Testing

3.1 Unit Testing

3.1.1 Testing frameworks: Unity

3.1.2 Test-Driven Development Methodology

3.2 Hardware Objects

3.2.1 Hardware Encapsulation

3.2.2 Communication Protocol

3.2.3 Data Serialisation

3.2.4 Bus Drivers

3.3 Hardware Verification Platform

3.3.1 Processing System Part

3.3.2 Programmable Logic Part

3.4 Integration with the typical design flow

3.4.1 High-level modelling verification

3.4.2 Co-simulation: RTL and Gate-level verification

3.4.3 Verification in a real device
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>7.1</td>
<td>Components of <em>Universal Verification Methodology</em> (UVM)</td>
<td>134</td>
</tr>
<tr>
<td></td>
<td>7.2</td>
<td>Black-box Testing Environment</td>
<td>135</td>
</tr>
<tr>
<td></td>
<td>7.2.1</td>
<td>Producer Part</td>
<td>136</td>
</tr>
<tr>
<td></td>
<td>7.2.2</td>
<td>Consumer Part</td>
<td>140</td>
</tr>
<tr>
<td></td>
<td>7.3</td>
<td>Writing Test Cases for Black-box Designs</td>
<td>143</td>
</tr>
<tr>
<td></td>
<td>7.4</td>
<td>Integration with our Verification Platform</td>
<td>145</td>
</tr>
<tr>
<td></td>
<td>7.5</td>
<td>Summary</td>
<td>147</td>
</tr>
<tr>
<td>8</td>
<td>8.1</td>
<td>Hardware Platform for Testing Service</td>
<td>150</td>
</tr>
<tr>
<td></td>
<td>8.1.1</td>
<td>Bitstream structure</td>
<td>151</td>
</tr>
<tr>
<td></td>
<td>8.1.2</td>
<td>zipFactory object: The deployment engine</td>
<td>154</td>
</tr>
<tr>
<td></td>
<td>8.1.3</td>
<td>Memory controller: AXI Read Memory</td>
<td>160</td>
</tr>
<tr>
<td></td>
<td>8.2</td>
<td>Remote Testing</td>
<td>162</td>
</tr>
<tr>
<td></td>
<td>8.2.1</td>
<td>Granting Remote Access</td>
<td>162</td>
</tr>
<tr>
<td></td>
<td>8.2.2</td>
<td>Remote Testing Frameworks</td>
<td>171</td>
</tr>
<tr>
<td></td>
<td>8.3</td>
<td>Testing Service</td>
<td>174</td>
</tr>
<tr>
<td></td>
<td>8.4</td>
<td>Summary</td>
<td>179</td>
</tr>
<tr>
<td>9</td>
<td>9.1</td>
<td>Main Contributions</td>
<td>183</td>
</tr>
<tr>
<td></td>
<td>9.2</td>
<td>Publications</td>
<td>185</td>
</tr>
<tr>
<td></td>
<td>9.3</td>
<td>Future Work</td>
<td>186</td>
</tr>
</tbody>
</table>
A Systematic Review

A.1 Planning of systematic review
   A.1.1 Framing the question
   A.1.2 Search protocol
   A.1.3 Criteria

A.2 Literature review execution

A.3 Grey Literature and systematic review expansion

A.4 Result analysis

B Facilitating DPR tasks through TCL scripts

B.1 Defining dynamic hardware project

B.2 Automatic generation of bitstreams
List of Figures

1.1 Moore law ............................................. 2
1.2 Verification gap. ........................................... 3
1.3 Average time FPGA design engineers spend in design vs. verification . 5
1.4 Trade-off between accuracy and effort. ................................. 7

2.1 Design intent, specification and implementation ............................... 17
2.2 Black-Box verification approaches ........................................... 18
2.3 White-Box verification approaches ........................................... 18
2.4 Grey-Box verification approaches ........................................... 19
2.5 Constrained-Random Stimuli ........................................... 24
2.6 UVM overview ........................................... 24
2.7 Example of Universal VHDL Verification Methodology (UVVM) . . . . 25
2.8 Verification environment of work [SBY11] using TLM ports and scoping rules ........................................... 28
2.9 Solution overview based on test reusing of work [Put14] . . . . . . . . . 29
2.10 Overview of block diagram of work [ICC10] ................................... 30
2.11 An example of reverse name matching of work [CCT07] ................. 31
2.12 Topological similarities by connectivities of work [CCT07] . . . . . . . 31
2.13 Correlation candidates by connectivities of work [CCT07] . . . . . . . 32
2.14 Overview of verification methodology proposed by [GPS14] . . . . . . 33
2.15 Smart FIFO interfaces proposed by [HCG13] . . . . . . . . . . . . . . 34
2.16 Methodology overview proposed by [BFPS15] . . . . . . . . . . . . . . 35
2.17 Hybrid verification framework proposed by work [BGJ11] . . . . . . . 36
2.18 Forced assertion based debug flow proposed by work [BGG13] . . . . 36
2.19 HPChecker Architecture of work [LWH08] . . . . . . . . . . . . . . . 37
2.20 Platform overview of work [CRL10] . . . . . . . . . . . . . . . . . . . . 38
2.21 Architecture of accelerated verification environment of work [PiCK15] . 39
2.22 In-Hardware verification process of work [LZ11] . . . . . . . . . . . . 40

3.1 Overview of unit testing frameworks . . . . . . . . . . . . . . . . . . 45
3.2 Test-Driven Development flow . . . . . . . . . . . . . . . . . . . . . . . 46
3.3 Main stages of feature extraction and object detection chain . . . . . . 51
3.4 $l^2$-norm algorithm and block diagram . . . . . . . . . . . . . . . . . 52
3.5 Overview of feature extraction and object detection chain . . . . . . . 54
3.6 Hardware object result from its C signatures . . . . . . . . . . . . . . . 60
3.7 Data flow inside c2hwobject . . . . . . . . . . . . . . . . . . . . . . . . 61
3.8 Overview of communication mechanism . . . . . . . . . . . . . . . . . . 62
3.9 Direct addressing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
3.10 Object broadcast address . . . . . . . . . . . . . . . . . . . . . . . . . . 64
3.11 Node broadcast address . . . . . . . . . . . . . . . . . . . . . . . . . . . 64
3.12 Node and Object broadcast address . . . . . . . . . . . . . . . . . . . . 64
3.13 Overview of flags field .................................................. 65
3.14 Overview of request message without payload ....................... 66
3.15 Overview of request message with payload .......................... 67
3.16 Overview of reply message without payload .......................... 67
3.17 Overview of reply error message ....................................... 68
3.18 Overview of reply message with payload .............................. 68
3.19 Example of argument alignment ....................................... 69
3.20 Example of pointer reinterpretation ................................... 70
3.21 Example of argument location ......................................... 70
3.22 Example of byte sequence conversion ................................ 72
3.23 Overview of hardware object in TLM ................................. 78
3.24 Overview of hardware verification environment ...................... 79
3.25 Overview of Remote Method Invocation (RMI) ....................... 80
3.26 Comparison between our scripts and Vivado tool (consumption-memory) 85
3.27 Comparison between our scripts and Vivado tool (consumption-power) 85
3.28 Overview of the typical FPGA design flow ............................ 86
3.29 Verification at different abstraction levels ............................ 88
3.30 Report of Unity framework in a pure software domain ................ 89
3.31 Report of Unity framework in a pure software domain (inducing a bug) 90
3.32 Unit tests and hardware object into co-simulation environment .... 90
3.33 Report of Vivado HLS tool using Unity framework after applying our proposal .......................................................... 92
3.34 Report of Vivado HLS tool using a co-simulation environment .... 93
3.35 Reports of sum_hist_pow, scale and mult_hist_scale modules ....... 94
3.36 Unit tests and *hardware object* into hybrid verification environment . . 94
3.37 Reports of *Unity* testing framework using a hybrid environment . . . 96

4.1 Overview of *Text Manager* object . . . . . . . . . . . . . . . . . . . . . 98
4.2 Overview of hardware verification environment with timing analysis
techniques . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 100
4.3 Reports of *Unity Timing* testing framework using *Vivado HLS* profiling 104

5.1 Example of work [KWS12] . . . . . . . . . . . . . . . . . . . . . . . . . . . 110
5.2 Overview of hardware assertions . . . . . . . . . . . . . . . . . . . . . . 112
5.3 Overview of *hardware object* with assertions . . . . . . . . . . . . . . 114
5.4 Report of test case with hardware assertions . . . . . . . . . . . . . . . 118

6.1 Overview of *Test Stub* . . . . . . . . . . . . . . . . . . . . . . . . . . . 122
6.2 Overview of *Test Fake* . . . . . . . . . . . . . . . . . . . . . . . . . . . 123
6.3 Overview of *Test Mock* . . . . . . . . . . . . . . . . . . . . . . . . . . . 123
6.4 Overview of *Test Spy* . . . . . . . . . . . . . . . . . . . . . . . . . . . . 124
6.5 Overview of *scale hardware mock object* . . . . . . . . . . . . . . . . 125
6.6 Overview of *scale mocked function* . . . . . . . . . . . . . . . . . . . . 128
6.7 Report of test case with hardware mocks . . . . . . . . . . . . . . . . . . 130
6.8 Report of test case with hardware mocks (Mock Failure) . . . . . . . . 131
6.9 Report of test case with a wrong hardware mock . . . . . . . . . . . . . 131

7.1 UVM Agent . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 134
7.2 Overview of $l^2$-norm streaming . . . . . . . . . . . . . . . . . . . . . 135
7.3 Overview of black-box testing proposal . . . . . . . . . . . . . . . . . . . 136
7.4 Overview of *sequencer* module . . . . . . . . . . . . . . . . . . . . . . 136
LIST OF FIGURES

B.1 Decision flow diagram to build configuration files ............... 209
Acronyms

ABV  Assertion-Based Verification.
API  Application Programming Interface.
ARP  Address Resolution Protocol.
AST  Abstract Syntax Tree.
AXI  Advanced eXtensible Interface.
DCP  Design Checkpoint.
DCT  Discrete Cosine Transform.
DES  Data Encryption Standard.
DMA  Direct Memory Access.
DOC  Depended-On-Component.
DPR  Dynamic Partial Reconfiguration.
DSE  Design Space Exploration.
DSP  Digital Signal Processor.
DUT  Design Under Test.
ESL  Electronic System-Level.
FIR  Finite Impulse Response.
FPGA  Field Programmable Gate Array.
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GCD</strong></td>
<td>Greatest Common Divisor.</td>
</tr>
<tr>
<td><strong>GPU</strong></td>
<td>Graphics Processor Unit.</td>
</tr>
<tr>
<td><strong>HDL</strong></td>
<td>Hardware Description Language.</td>
</tr>
<tr>
<td><strong>HLL</strong></td>
<td>High-Level Language.</td>
</tr>
<tr>
<td><strong>HLS</strong></td>
<td>High-Level Synthesis.</td>
</tr>
<tr>
<td><strong>HOG</strong></td>
<td>Histogram of Oriented Gradients.</td>
</tr>
<tr>
<td><strong>ICAP</strong></td>
<td>Internal Configuration Access Port.</td>
</tr>
<tr>
<td><strong>Ice</strong></td>
<td>Internet Communications Engine.</td>
</tr>
<tr>
<td><strong>IDL</strong></td>
<td>Interface Description Language.</td>
</tr>
<tr>
<td><strong>ILA</strong></td>
<td>Integrated Logic Analyzer.</td>
</tr>
<tr>
<td><strong>IP</strong></td>
<td>Intellectual Property.</td>
</tr>
<tr>
<td><strong>LLVM</strong></td>
<td>Low Level Virtual Machine.</td>
</tr>
<tr>
<td><strong>LUT</strong></td>
<td>Look-Up Table.</td>
</tr>
<tr>
<td><strong>OOCE</strong></td>
<td>Object-Oriented Communication Engine.</td>
</tr>
<tr>
<td><strong>OOP</strong></td>
<td>Object-Oriented Programming.</td>
</tr>
<tr>
<td><strong>OVL</strong></td>
<td>Open Verification Library.</td>
</tr>
<tr>
<td><strong>OVM</strong></td>
<td>Open Verification Methodology.</td>
</tr>
<tr>
<td><strong>PCAP</strong></td>
<td>Processor Configuration Access Port.</td>
</tr>
<tr>
<td><strong>PSL</strong></td>
<td>Property Specification Language.</td>
</tr>
<tr>
<td><strong>RMI</strong></td>
<td>Remote Method Invocation.</td>
</tr>
<tr>
<td><strong>RTL</strong></td>
<td>Register-Transfer Level.</td>
</tr>
<tr>
<td><strong>RVM</strong></td>
<td>Reference Verification Methodology.</td>
</tr>
<tr>
<td><strong>SDR</strong></td>
<td>Software Defined Radio.</td>
</tr>
<tr>
<td><strong>SoC</strong></td>
<td>System-on-Chip.</td>
</tr>
<tr>
<td><strong>SUT</strong></td>
<td>System Under Test.</td>
</tr>
</tbody>
</table>
SVA  *SystemVerilog Assertions.*

TCL  *Tool Command Language.*

TDD  *Test-Driven Development.*

TLM  *Transaction-Level Modeling.*

UVM  *Universal Verification Methodology.*

UVVM  *Universal VHDL Verification Methodology.*

XCI  *Xilinx Core Instance.*
Chapter

1

Introduction

«If we knew what we were doing, it wouldn’t be called research, would it?»
Albert Einstein

1.1 Motivation and Problem Overview
1.2 Hypothesis
1.3 Thesis Objectives
1.4 Thesis Contributions
1.5 Thesis Outline

Nowadays, embedded systems can be found inside any electronic device; televisions, vacuum cleaners, mobile phones, cars, medical tools, ... As is customary, these kind of devices are composed by two clearly separated domains; on the one hand, a hardware domain that is able to carry out the tasks for which it is developed. On the other hand, a software domain whose aim is the interaction between the hardware domain and users, allowing to manage the device.

Usually, embedded systems are used for specific tasks, optimising considerably the throughput and the reliability of a design. FPGA-based designs are dramatically increasing in embedded system designs. A Field Programmable Gate Array (FPGA) is a reconfigurable hardware device whose behaviour can be determined or configured using a programming file called bitstream. FPGAs provide some advantages in the embedded system field, highlighting their low cost and their high throughput allowing applications based on this technology to reach better performance. Some FPGA vendors offer their own embedded processor cores, such as MicroBlaze, as well as support for popular third party processors from suppliers like ARM. In recent years, FPGA vendors have focussed their efforts on reducing the development flow, exploiting as much as possible all resources provided by their FPGAs. At this sense, and in counterpart of the initial statement of this paragraph, cloud providers have introduced into their datacenters this kind of devices, allowing to speed up the algorithms deployed or developed by their clients considerably. This fact opens new research lines, at the same time that opens new problems and concerns.
1.1 Motivation and Problem Overview

The rapid advances in the System-on-Chip (SoC) field have provided better and more complex devices. These devices have a long resource area, better throughput and better reliability due to industrial market demands. The Moore law declares: «The complexity for minimum component costs has increased at a rate of roughly a factor of two per year. Certainly, over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years» [Moo65]. This hypothesis formulated by Gordon Moore has since been fulfilled. Evidence can be found in Figure 1.1. This plot illustrates the trade-off between the number of transistors of an integrated circuit and its introduction date, the fifth generation of intel processors, for example, satisfies this older law [Ant16]. Its truthfulness implies a higher complexity and brings benefits to developed devices, allowing compact, quick and safety designs, which include computational power saves.

This complexity is the result of including more functionality inside the same device. Consequently, attaining a solution based on an electronic system becomes a difficult task. Although device vendors make an effort to minimize this challenge, by including new features in their development tools, the complexity is an important challenge for developers. New development tools bring better resource utilisation and reduces the development effort.

Figure 1.1: Moore law [Wgs11]
Intel acquired FPGA chip vendor Altera in order to put CPU and FPGA onto one package, thus Intel plans hybrid FPGA-CPU chips [Pat17]. In the same research line FPGA vendors are betting heavily on heterogeneous systems, such as a hybrid FPGAs. The FPGA part is coded in a hardware description language such as Verilog or VHDL, and lots of software in C, assembly language or other popular languages. In embedded systems, FPGA implementations are widespread, because they are seen by developers as a solution to speed-up their algorithms, decreasing economic costs and risks. FPGAs are usually used for rapid prototyping, even to make a Design Space Exploration (DSE); embedded applications which require good consumption/throughput and high-performance computing amongst other capabilities. These kinds of devices have a faster time-to-market, which includes new version development and bug-fixing work. Therefore, the high manufacturer costs are counteracted by the sales, resulting in high economic benefits. FPGA vendors provide new development tools, such as High-Level Synthesis (HLS) tools, by investing their resources in them. These kind of vendor tools allow engineers to develop hardware designs and facilitate the development flow. These new marketing guidelines open new challenges and incite role intrusion, software engineers are able to build hardware designs without high knowledge about hardware design. For instance, designs can be described in a high-level programming language like software engineers do, however it really entails some challenges because HLS tools are not accuracy or engineer’s code is inaccuracy.

Therefore, HLS adds a new layer to the typical design flow for building hardware systems. A design can be captured by high-level programming languages such as C and mapped to an FPGA using HLS tools. HLS bridges both software and hardware domains; software developers are able to speed up the computationally intensive parts of their algorithms easily, while hardware developers are able to take advantage of the productivity benefits of working at a higher abstraction level [Xil12a]. But there is still a long way to go in terms of hardware verification and testing, which are the main bottlenecks in the development flow (Figure 1.2). The functional verification process is becoming increasingly difficult and time-consuming, regardless of the domain. It is a challenge for safety-critical applications in many sectors (such as the aerospace and medical sectors) [San12]. Indeed, the hardness of this task mainly

![Figure 1.2: Verification gap [MC07]](image-url)
resides in the low observability of the code generated by HLS tools [Fos16]. In the same context, advances in any domain of electronic systems bring novel and powerful solutions without a big effort. For instance, new programming languages, like LARA, allow the description of sophisticated code instrumentation schemes, advanced mapping strategies including conditional decisions, based on hardware/software resources, and sophisticated sequences of compiler transformations [BPN14].

Evolution of development based on FPGA-design

The increase of design complexity demands a revolution of the techniques used for digital electronic design. At the beginning, integrated circuits were built manually, transistor by transistor, denoting in the design the size, location and communication with the rest of the components (Bottom-Up methodology). Later, the appearance of Hardware Description Languages (HDLs), such as VHDL [IEE02] or Verilog [IEE01], propitiated a change into Top-Down methodologies. This methodology, along with the use of CAD-tools, has led to developers being able to focus on depicting a system description at the functional and behavioural level, allowing design simulation before its synthesis.

Years later, C programming language [KR88], which is the preferred language for embedded systems, was extended to build new High-Level Languages (HLLs) allowing an easy description of complex systems. Examples of this kind of language are Handel-C [Cel02] or SystemC [Ope03]. These languages depict the behaviour of a specific hardware element in the same way as in the software domain. At the same time, several companies were working on improving the quality of their products, and decreasing the development time. In order to reach a solution fulfilling these requirements, companies built their design by means of the technique design based on models, using specific tools such as Simulink from MATLAB. The main goal of these tools is decreasing the design gap and increasing the abstraction level. This leads to another way of conceiving electronic design, in which high-level programming languages, such as C/C++, play an important role in hardware development flow.

In this sense, FPGA vendors are making a big effort to fill the existing gap between development tools and the capabilities offered by the technology, including HLS or C synthesis technology as a novel solution. Various academic tools, such as HercuLeS [KM13] or LegUp [CCA11], and commercial tools, such as VivadoHLS from Xilinx [Xil16c] or Symphony from Synopsys [Syn16], have been developed to minimize this gap. HLS tools allow engineers to describe functionality or algorithms in a high-level programming language, such as C, C++ or SystemC. These high-level modelled algorithms are subsequently translated into low-level cycle-accurate Register-Transfer Level (RTL) descriptions, for an efficient implementation on an FPGA. These synthesis processes can be optimised, taking into account performance, power, resources, etc. [CLNN10]. Thus, HLS tools enable design space exploration and improve the likelihood of finding the most-optimal implementation [Xil12a].
On the other hand, verification process is the pending task of hardware development, because verification stage has not been improved like digital electronic design. In addition, verification stage is the bottleneck of most projects and entails some time-to-market problems. These two affirmations are aligned by the Wilson study. The Wilson Research Group does a functional verification study based on FPGA technology every two years, analysing working groups and their working at several companies. Authors of this study have drawn conclusions about three components: technology, design and verification [Fos16]. Although the study is based on FPGA-designs, some of their conclusions fit with a more generalist hardware-design point of view.

- Many companies have adopted hybrid solutions using hybrid FPGAs, such as Zynq, Cyclone or SmartFusion. Roughly 58% of all FPGA-designs contained one or more embedded processors. This kind of device includes a new verification complexity-layer due to its hybrid nature: interaction between hardware and software domain. This is another aspect of SoC design, regardless of the FPGA-type implementation. Most FPGA projects use an industry standard on-chip bus protocols, such as Advanced eXtensible Interface (AXI), which is increasingly used in FPGA-solutions.

- The average time spent on verification tasks is 48%. Many companies propose a solution to reduce verification time based on mixed working groups, which are composed by design and verification engineers. Although the number of engineers depends on the project, the ratio of design engineers versus verification engineers is approaching 1-to-1.

- In mixed working groups, it is important to note that FPGA verification engineers are not the only project members involved in the functional verification process. Design engineers spend a significant amount of their time on verification tasks as well. This process takes an average of around 51% of design engineer’s worktime (Figure 1.3).

![Figure 1.3: Average time FPGA design engineers spend in design vs. verification [Fos16]](image)

- The metric used by the study to check the verification effectiveness is the number of FPGA-iterations, of which the result was too high (7 or more iter-
1.1. Motivation and Problem Overview

The *time-to-market* challenge plays an important role in the company competitiveness, and its main hidden danger is the product development time and quality status, both independent of the offered product and marketplace. A good example of company competitiveness can be found in the video-game console market. The *Nintendo Wii console*\(^a\) was marketed before the console of its direct rival, *Sony*, put its *Playstation3 video-game console*\(^b\) on sale after the *Wii platform* was put on the market. This lead to *Nintendo* selling more units than *Sony* (see Figure) [Orl13].

\[\text{Time-to-Market}\]

\[\text{Time-to-Market}\]

---

\(^{a}\)Market announcement in Europe: December 8th, 2006.


- Simulation techniques based on FPGA-technology are focused on four models: code coverage, assertions, functional coverage and constrained-random simulation. In addition, the use of verification methodologies, such as *UVM*, Open Verification Methodology (*OVM*), Reference Verification Methodology (*RVM*), ..., is more popular in the industrial market. The study illustrates the trend of FPGA-verification and remarks that *UVM* will become the most used verification methodology in a few years.

Having a good design verification process is strategic because of the potential consequences of failures. If a product is marketed containing bugs, the company will encounter problems because it is responsible for changing the product, without charging the consumers for this. This could lead to bankruptcy, inasmuch as customers will distrust future products from this company. It could also be punished by a penalty fee. We can find several examples in the past that show the importance of the verification process is. The *Pentium FDIV bug* is one of the most representative examples of this; the processor could return incorrect binary floating-point results when dividing a number [Pri95]. Another current example is the problem with the batteries of the
1. Introduction

Samsung Galaxy Note 7. These batteries can suffer a short circuit, causing the mobile phone to burn off. This problem has resulted in important economic losses, about 4.863 million euro. The research process to find the bug involved 700 professionals, including technicians and engineers of the Samsung company and another three more companies: UL and Exponent consultants and TÜV Rheinland [Hip17].

The following paragraphs describe some problems of verification process of FPGA-based designs. These problems are the main motivation of this Ph.D. dissertation, and hence our start point.

- Most bugs that appear during the development flow are related to the low precision achievable during the verification process [Fos16]. On the other hand, in hardware, FPGA-based designs are typically verified through RTL-simulation, to validate the design intent and to ensure 100% code coverage. In simulation, results can be visualised, analysed, compared and the requirement traceability can be easily maintained, while in real hardware implementations, the FPGA is already configured with the design, designs cannot be easily traced back to simulation because of its poor visibility and accessibility. In real hardware implementations the achieved accuracy is high but the simulation effort and time demands are high as well, while in RTL-simulation the effort is very low but the accuracy is not good (Figure 1.4) [GD15]. Figure 1.4 illustrates the trade-off between simulation effort and simulation accuracy when simulating a design at different abstract levels. Here we can observe that the lowest simulation effort is obtained using High-Level Modelling, but it usually results in inaccuracies. A real hardware prototype is the perfect environment for verification, but this technique requires greater verification efforts. Summarising, the verification effort increases with the increment of the verification accuracy.

Using real hardware means an ideal verification environment, because it is very close to reality, and gets good cycle-accuracy results. It is also known as hardware-in-the-loop or on-board verification. Unfortunately, the effort involved is too high
and requires no trivial tasks. Firstly, to have full traceability in real hardware environments, designers should be able to compare the behaviour of the physical component with the designer intent. Designers must be able to exercise cores in bit-accurate level, requiring to have a testing environment per core and per design level. This implies that each one of these design levels must be tested and verified before going further in the design process. Thus, a testing environment must be developed at each level. As the reader can observe, each testing environment induces a test translation task or rewriting test, which is prone to human errors. Developers have to build new tests for each level; High Level Modelling, RTL level, Gate-level, implementation level or in-hardware level. Test translation processes lead to wrong decisions, developers try to modify their tests according to the testing level instead of the production code, with the only aim to pass their tests at any level.

<table>
<thead>
<tr>
<th>What is simulation and verification?</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Simulation</strong> is very widespread in order to check the correctness of a hardware design because it is physically independent. Thus simulation does not rely on any results from the implementation process or on the details of the target device to which the design is intended to be deployed. On the other hand, <strong>verification</strong> tries to establishing the truth, accuracy, or validating something (hardware designs in our case). Verification is quite complex because it includes different strategies. For instance, a design can be verified using formal methods in order to explore all possible scenarios or a design can run on the target device under real-world conditions in order to include physical parameters into verification process, such as timing. Debugging is a process that involves identifying, isolating and correcting a problem or determining a way to work around it.</td>
</tr>
<tr>
<td><strong>Formal verification</strong></td>
</tr>
<tr>
<td>It uses mathematical techniques to ensure that a given hardware design conforms to a set of precisely expressed notions of functional correctness. The basic goals are to verify that the design does everything it is supposed to do, and does not do anything that it is not supposed to do [Hu12].</td>
</tr>
<tr>
<td><strong>In-hardware verification</strong></td>
</tr>
<tr>
<td>It uses a real device to provide a real scenario and to keep same features of the final release or where the design will be run. It is also know as on-board verification or hardware-in-the-loop. On-board verification ensures that even if there is a bug in the FPGA modelling library it will be detected [LZ11].</td>
</tr>
</tbody>
</table>

Summarising, simulation verifies the design intent and it detects functional bugs, while verification validates the design completely.
• Another issue to consider in verification is the timing. One of the main differences with the simulation process in the software domain is that most projects do not consider the timing component, performing a pure functional verification, or an event-based verification where only the correct sequence of events is verified. However, in the hardware domain both is necessary, so functional and timing verification, including timing analysis in some verification levels. The timing component in the hardware domain makes the verification process more complex, because the timing factor results in-hardware level verification usually differ from verification models used in simulation, in which developers do not care about available resources and where these are located. This fact leads to differences about internal-signal propagation time between the simulated design and the real one.

• However, using a real hardware device introduces a new problem. Because the synthesis process time increases, the verification time increases as well, this leads to an aggravation of the associated bottlenecks and an increase of the power-consumption. Sometimes, the number of FPGA iterations is taken as a metric for determining the overall verification effectiveness [Fos16].

• Another problem in any verification domain is related to third-party dependencies. The hardest task might be to integrate a new component into a test environment fulfilling its dependencies, since several of them may not be implemented at that time, or other external devices are not available. Many companies provide the simulation models for their products, but these cannot be synthesised. Moreover, some of these models do not completely match with their real hardware implementation, which is prone to inaccuracy.

• In addition, a hardware component can be implemented in different ways, and according to the DSE done by the developer. Thus, the DSE takes place at implementation time in the early development flow stages. In this sense, from high-level modelling we are not able to ensure that the final implementation meets the time requirements, we only get a profiling or report about our implementation which many times does not match with reality.

Summarising, performing verification at the board-level is not only challenging and risky, it is sometimes just not feasible within project timescales. This is why engineers are increasingly adopting a so-called in-hardware verification methodology [LZ11]. Therefore, the know-how in hardware design/verification fields is very important to keep all verification requirements, and to build an in-hardware verification environment in project timescales [Fos16]. Although the motivation has been oriented to an FPGA domain, all challenges and advances are also applicable to other kinds of devices. For instance, the way to program a microprocessor is constantly being improved, for example by including new optimisation features into compilers, or by using novel programming languages oriented on solving or improving throughput problems.
1.2 Hypothesis

HLS tools bring new facilities for using high-level programming languages, such as C or C++, allowing software developers to speed up their algorithms easily. These tools enable hardware developers to take advantage of the productivity benefits of working at a higher abstraction level, and they are able to have an early definition of the design space for fast design exploration [Xil12a]. However, there is still a long way to go in terms of hardware verification and testing, opening a new research line, in which the High-Level Modelling plays an important role. At the same time, using a real device to carry out the verification step achieves a better verification accuracy, providing an ideal scenario, whereas using High-Level Modelling reduces the simulation effort that the real hardware introduces.

Melding both goodness, high-level modelling productivity and verification accuracy that provides a real device, entails new challenges and questions that this Ph.D. dissertation overcomes. In addition, this thesis attends to use reconfigurable hardware, even using dynamic partial reconfiguration feature which improves the time-consumption and power-consumption for generating the physical implementation of a design described from a high-level programming. A verification environment should be kept independently of the development stage, in order to check the design correctness with the same test and different simulation/verification accuracies. All verification processes can be automated without developer intervention, which allows building the design from its high-level modelling to a physical implementation when a new release is ready and enables an automated check, also known as continuous integration.

Analysing the above ideas and looking for a solution for verification problems in real hardware and providing some facilities to build hardware components and their verification, we propose the following hypothesis:

It is possible to build a hardware verification framework based on reconfigurable devices checking automatically the correctness of a design generated from a high-level language into a real device, using the same tests at any abstraction level and keeping high-level modelling productivity and verification accuracy that real devices provide

1.3 Thesis Objectives

This thesis aims to provide solutions to the significant challenges in hardware verification and to provide assistance to designers in testing and debugging hardware designs. Since design engineers also spend significant time testing and debugging their
designs, as we mentioned in previous sections, this thesis does not explicitly distinguish between design engineers and verification engineers but refers to both as «designers» or «engineers».

1.3.1 Mainstream Objective

The mainstream objective of this dissertation is to propose an integral solution that comprises several verification problems in digital hardware design. This solution is presented as a novel hardware development methodology, using the new generation of tools provided by FPGA vendors. At the same time, the dissertation should focus on a solution that includes two important verification intents: functional and timing factors; whose precision should be close to a real scenario. Thereby, hardware designs described by a high-level programming language must be verified through an automatic infrastructure allowing a transparent verification using a real hardware device.

1.3.2 Other Objectives

Other objectives of this thesis are as follows. These objectives are proposed to reach the mainstream objective.

- To provide a methodology, regardless of the final device, where hardware components will be deployed, allowing easy and quick hardware components modifications, providing a rapid design space exploration.
- Reducing designers effort in verification stage using real devices.
- To provide a transparent communication mechanism between components independently of their domain (software or hardware).
- To ease the verification stage without building the whole system or a specific verification environment.
- To assess functional and timing verification in real devices.
- To provide a testing framework that allows to build unit tests.
- To reuse tests independently of the abstraction level.
- To reduce or delete third-party dependencies.
- To provide a universal verification platform.
- To provide an automated, transparent and remote verification service allowing continuous integration.
1.4 Thesis Contributions

As described earlier, the trade-off between simulation effort and simulation accuracy depends on the simulation abstraction level. Thus, the highest accuracy is reached using real hardware, however the simulation effort is too high. On the other hand, high-level modelling reduces the simulation effort but achieves a bad simulation accuracy. The core idea of this work is to use the goodness of high-level modelling to achieve the same accuracy of the prototype level, reducing the disadvantages that it includes. The key contributions of this thesis are:

- We propose a hardware verification environment based on FPGA-technology using a Dynamic Partial Reconfiguration (DPR)-feature. This feature reduces the time of the synthesis process to get a partial bitstream that only implements the Design Under Test (DUT). In addition, this verification environment allows to exercise the DUT from the embedded processor, such as ARM, or from a remote node.

- From a communication point of view, we present a communication mechanism like a protocol, that can be adopted over a standard bus, such as an AXI bus. This communication mechanism is able to work on any standard bus due to our proposal being transmitted over the data channel.

- From a verifications perspective, we propose an extension of a testing framework that allows to build unit tests and bunch them into a test suite. This test suite can be executed at any development flow level (e.g. C level or Register-Transfer Level (RTL)). This extension includes timing analysis getting the elapsed time in algorithm execution or in a specific part of the algorithm. In order to achieve a solution that brings one test suite and multiple abstraction levels, our proposal is based on RMI-technology that allows communicating heterogeneous parts transparently.

- Third-party dependencies are solved through mock solutions. This technique is based on fake functions that simulate a specific kind of behaviour. These functions can be questioned by the extended testing framework, to know how much time was called upon, to know the correctness of call-upon parameters, ... In addition, we provide a library of hardware assertions that can be called upon, checking internal variables or intermediate results.

- We provide an automated transparent verification infrastructure, using several technologies; Git repository to store the source code of our DUT; Jenkins to ensure continuous integration feature; and Docker to ensure a clean verification container that is able to build partial bitstreams independently.
1.5 Thesis Outline

This dissertation is composed of nine chapters and two appendices. Bibliographic references and information about the author are also provided. The remainder of this dissertation is organised as follows:

- **Chapter 2 - Background and Related work.** This chapter discusses and includes an overview of the related work to our research, namely about the verification process in different simulation accuracies; hardware verification methodologies; and real hardware devices that carry out the verification.

- **Chapter 3 - Hardware Unit Testing.** This chapter explains the first part of this Ph.D. dissertation. This chapter describes how the unit testing technique is applied in logic programmable devices, building our solution on high-level modelling and **Object-Oriented Programming** (OOP). The chapter includes a hardware verification platform that carries out a verification with high accuracy and reuses the unit tests implemented in the first stage.

- **Chapter 4 - Timing Measurement in Hardware Unit Testing.** This chapter describes the new components added to our hardware verification platform to measure the function time execution. Moreover, in order to overcome timing measurement from unit tests, we propose an extension of a testing framework.

- **Chapter 5 - Hardware Asserts.** This chapter depicts how we can insert assertions in the middle of a hardware design, in order to retrieve information about internal signals. These assertions are synthesizable, thus the information retrieved comes from a real device and enables to check intermediate steps.

- **Chapter 6 - Hardware Mock Functions.** This chapter describes how third-party dependencies can be reduced using **Test Doubles**, such as mocks. Mocks sham to **DUT** that they are real, but in reality contain a pre-defined behaviour of the real component.

- **Chapter 7 - Black-box Designs.** This chapter illustrates how **black-box designs** can be verified using our hardware verification platform. The solution proposed is based on UVM and unit testing.

- **Chapter 8 - Hardware Testing Service.** This chapter explains our transparent, remote and efficient hardware testing service. It entails an amount of new challenges, overcoming them in this chapter.

- **Chapter 9 - Conclusion and Future Work.** This chapter summarises and discusses the most important aspects and contributions of this dissertation and also future research activities.
• **Appendix A - Systematic Review.** This appendix presents the systematic review to get the related work that are relevant to this Ph.D. dissertation. The result of this process is illustrated in Chapter 2.

• **Appendix B - Facilitating DPR tasks through TCL scripts.** This appendix explains how we can generate the configuration files or partial bitstreams using some *Tool Command Language* (TCL) scripts.
2. Background and Related Work

«The whole art of teaching is only the art of awakening the natural curiosity of young minds for the purpose of satisfying it afterwards»
Anatole France

2.1 Verification Challenges
2.2 Functional Verification
2.3 Methodologies and tools
2.4 Related Work

The purpose of this chapter is to provide a background of hardware verification across the abstraction levels in FPGA-designs. Previously, we illustrated the verification challenges that engineers face and which challenges must be overcome by which approach. In this chapter we present a general discussion on functional verification and its difficulty in an FPGA-device, being the main bottleneck of hardware design flow.

Finally, all studies that compose the related work are described here, ordered by the tuple simulation effort-accuracy. The studies are picking out their goodness and their badness, or alternatively, we summarise all studies of related work.

In order to obtain the related works, we applied some steps, the objectives that were close to this Ph.D. dissertation, are shown in Appendix A. This process is known as a systematic review. We have applied these steps since formulating our Ph.D. hypothesis and were able to get the most important studies for this thesis. In addition, the result of systematic review is extended with the Gray literature that contains hardware verification methodologies among other technical reports and articles from special issues, including specific congresses or journals.
2.1 Verification Challenges

The increasing complexity of designs and the shortened *time-to-market* have converted verification in an important challenge to address. This means that engineers must verify larger and more complex designs in a shorter time, making the verification stage the bottleneck of most hardware projects. The verification challenges that engineers must address, are as follows [Sas08] [WCH13] [GD15] [Lev13] [Fos16] [men09].

- The engineer must try to verify the whole design behaviour or at least maximise the behaviour checked. This is known as verification completeness. One of the main problems in verification completeness is capturing all scenarios that must be verified. This task introduces many errors because of its handmade nature: error-prone, omission-prone process, misunderstanding errors, ... An ideal verification environment should guarantee correct stimuli generation.

- The challenge in verification reusability is to increase portions of the verification environment infrastructure that can be used in the current project or in future projects. A high degree of reuse can be achieved for standardised interfaces or for standard bus protocols, such as AXI. It requires the verification environment to be highly reusable and maintainable, across different projects, perhaps with different teams, with a quick turn-around. This requires a certain methodology for verification work.

- The verification stage contains a certain amount of manual effort, which must be minimised to achieve a high degree of verification efficiency. Manual efforts lead to verification errors and are time consuming. In contrast, automated systems are able to complete many tasks in a short time. Therefore the verification reusability challenge and this challenge are similar, because reusing an automated well-checked infrastructure minimises the time and reduces errors.

- Another challenge similar to the verification efficiency, is the verification productivity. This challenge tries to maximise work that is produced manually in a certain period of time. Verification productivity gains can be obtained by moving to higher levels of abstraction, however, that can also cause a loss of precision.

- Maybe expert knowledge of tools and languages is the primary requirement of the verification stage. This is similar to maximising the efficiency of verification programs, known as verification code performance. This challenge is usually considered as a secondary task, but is important in reducing the verification time.

- Complex designs are being created in ever shrinking *time-to-market* windows, due to the use of third-party components. The use of these third-party components has created a new challenge in the verification stage. This challenge is known as verification correctness. For instance, a hardware component that uses a
third-party component is not addressed properly without a third-party simulation model. The verification stage must be able to integrate these third-party dependencies into the verification environment with minimal or no errors. The reality is that this type of integration is very time-consuming and error-prone. In addition, one factor that is not addressed in most projects is the time consumption of a hardware component. This requires a high effort by designers, because sometimes a correct component has to be re-designed because it does not meet timing requirements. This re-design stage is time-consuming and error-prone.

2.2 Functional Verification

Traditionally, product design flow is divided into two big stages; the first stage is to convert the idea into a functional specification, the second stage is to translate the specification defined at the first stage into design implementation. These two stages lead to functional errors and malfunctions in the final product, due to manual processing, vague specifications, misunderstanding by designers, implementation errors, etc. [Sas08].

The primary goal of functional verification is to check the correctness of a design from specific stimuli; checking the functional equivalence between the design implementation and the purpose that has to be achieved. Or alternatively, checking the convergence between the product intent, functional specification and design implementation, by identifying any differences between the three factors and giving the opportunity to eliminate these differences. When these three factors converge with a high percentage, this signifies a high degree of confidence (Figure 2.1).

![Figure 2.1: Design intent, specification and implementation [Sas08]](image)

**Black-Box, White-Box and Grey-Box Verification**

**Black-box** verification refers to verifying a block or functionality of a design through its signals. Figure 2.2 shows the general architecture of this verification type. In this approach, stimuli are applied to both the DUT and the golden reference model.
 Outputs produced by the DUT and the golden model are checked for equivalence [Sas08].

Figure 2.2: Black-Box verification approaches [Sas08]

Black-box verification approach has the following drawbacks and points to keep in mind when we apply this approach.

- Difficulty to verify features related to design decisions; one specification, multiple implementations.
- The complexity of DUT, which has a too long simulation to get a result. A wrong result means leads to a difficult task discovering the bug.
- Black-box verification requires an accurate reference model, which usually is implemented in a high-level programming language. The reference model must be verified. A wrong golden model leads to a final product with bugs.

White-Box and Grey-Box verification provide alternative approaches for addressing the limitations of Black-Box verification. In White-Box verification no reference model is needed, because the approach uses monitors and assertions on internal signals of the DUT (Figure 2.3) [Sas08].

Figure 2.3: White-Box verification approaches [Sas08]

Grey-Box verification is a mixture of both approaches described above. Monitors and assertions are used along a reference model. The use of monitors and assertions reduces the accuracy requirements and the debugging effort (Figure 2.4) [Sas08].
2.2.1 Functional Verification of FPGA-based designs

Functional verification is the bottleneck of most hardware design projects. In FPGA-design projects, the ratio of dedicated verification engineers to design engineers is one to one. In addition, verification engineers are not the only project members involved in functional verification tasks. Design engineers spend a significant amount of their time on verification as well [Fos16]. Although bugs in reconfigurable designs can quickly be recovered by downloading a correct version, the effort required to detect and fix functional bugs is big. Accelerating verification steps can thus reduce time-to-market and thereby reduce economic costs [GD15].

Generally, simulation is the most common method to verify hardware design functionality (DUT). Engineers check the results of the DUT generated by selected stimuli against the expected results, like black-box verification (see Figure 2.2). After engineers conclude that the functionality is correct, they need to check the code coverage obtained with the executed tests and, most times, they need to build new tests in order to cover the missing scenarios [GD15].

However, the above paragraph is a general outline of the simulation stage. Hardware design flow is more complex and is composed by certain quantity of stages. Starting with the design intent, someone describes a specification using the natural language, which then is translated into a programming language. At this point engineers can use a high-level programming language, such as C, and then use an HLS tool or an HDL to transcript the specification directly into a RTL description. Then the design is synthesised and implemented using FPGA-vendor tools (i.e. Vivado from Xilinx [Xil16d]). After this sequence of steps, the design is ready to be deployed on a device [CLNN10]. Each step needs to be verified to ensure the correctness of the hardware design.

Each step of hardware design flow provides different simulation accuracy, which depends on the level of abstraction. When the simulation contains more details, it is more accurate (see Figure 1.4). However, the verification productivity decreases and it is often more time-consuming for designers to trace a bug when the simulation
fails. For example, a hardware design can be modelled and simulated using high-level programming languages although the simulation is not cycle-accurate, its throughput is orders of magnitude higher than RTL designs. Thus, high-level modelling sacrifices simulation accuracy for the sake of verification productivity [GD15]. Moreover, a test of a hardware design flow stage cannot be reused in another stage in most hardware projects, and each stage contains its own tests.

Hardware design projects introduce new verification challenges, such as the timing factor. Timing simulation is very important in these kinds of projects, whose results change according to the abstraction level. In the high-level modelling or RTL-stage we do not have enough information about the design, however, since the design is modelled in more detail, we are able to identify incorrect timing requirements. Therefore, those stages of hardware design flow which are near to the final stage, contain accuracy information than the first stages because the information is extracted from the implemented design instead of implementation process. Therefore, timing simulation allows you to ensure that the implemented design meets functional and timing requirements and has the expected behaviour in the device [Xil16b].

2.3 Methodologies and tools

Nowadays, it is common to use verification methodologies for developing complex test environments or test bench. These methodologies usually use verification languages, such as SystemVerilog. SystemVerilog has become the dominant language standard for functional verification and most formal verification methodologies use it. In this section, we introduce some methodologies based on SystemVerilog and we explain other novel methodologies not based SystemVerilog, such as UVVM.

The scheduling semantics of SystemVerilog are extended in order to allow new language constructs, such as property evaluations while maintaining backward compatibility with Verilog. Cycle-based verification approaches are facilitated through the introduction of clocking blocks, that allow for automation of sampling and driving of design signals with respect to sampling clocks. SystemVerilog also provides new constructs for random stimuli generation, property and assertion specification and coverage collection. Therefore, Verilog has been extended to fulfil verification tasks, providing better control over time [Sas08].

2.3.1 Assertion-Based Verification

Assertion-Based Verification is a methodology for improving the effectiveness of a verification environment. Engineers define properties that specify the design intent,
2. Background and Related Work

and the correctness of this intent is checked by assertions. Thus, assertions are used to clarify specification requirements, capture the design intent and validate its correctness [RAA16]. While a property is built from Boolean expressions, that describe behaviour over one cycle; sequential expressions, that describe multi-cycle behaviour; and temporal operators, that describe relations over time between Boolean expressions and sequences [Ace04]. We can list the following goodness of this methodology [RAA16].

- **Verification efficiency.** Assertions can be verified by different tools.

- **Observability.** The **Assertion-Based Verification (ABV)** catches the wrong output where it occurs and reports the error, reducing the search area for finding the bug.

- **Reusability.** Different designs may share the same modules and interfaces, and the properties can be reused with the same assertions.

Assertions are primarily used to validate the behaviour of a design. They also provide information about the coverage. An assertion is basically a statement that checks the truthfulness of an expression, for example \( \text{assert}(a==b) \), checks that both \( a \) and \( b \) are equal. Assertions are very widespread. The most common standard assertion languages are **SystemVerilog Assertions (SVA)** and **Property Specification Language (PSL)**. The **Open Verification Library (OVL)** is not a language, it is a library of predefined checkers written in VHDL or Verilog.

**SystemVerilog Assertions**

SystemVerilog adds features to traditional Verilog to specify assertions of a system. A SystemVerilog assertion checks a property correctness. If the assertion is evaluated false, it will throw an error. In addition, an assertion can be followed by a report statement, thus we are able to give some information about the error. When an error happens, we can determine the severity of the error (fail, error, warning or info).

```
Listing 2.1: Example of **SystemVerilog** immediate assertion
1  assert (A == B) $display("OK. A equals B");
2  else $error("It's gone wrong");
```

However, the **SystemVerilog** contains another kind of assertion, called «concurrent assertions». This kind of assertion allows to define the behaviour of a design using statements like «read and write signals should never be active-high together». The following expression covers this restriction.
2.3. Methodologies and tools

Listing 2.2: Example of SystemVerilog concurrent assertion

```verilog
assert property (! (read && write));
```

In addition, SystemVerilog assertions enable to check a more complex sequence expression, like «A request is followed by an acknowledge, which it does not take more than two clocks after the request». This means that when a request happens, an acknowledgement is active-high on the next clock, on the one following, or both. This signal sequence expression is described by the implication operator (|->). This kind of assertion usually appears outside any initial or always blocks.

Listing 2.3: SystemVerilog assertion with implication

```verilog
assert property (@ (posedge Clock) Req |-> ##[1:2] Ack);
```

There are two forms of implication: overlapped using operator |->, and non-overlapped using operator |=>. For overlapped implication, the antecedent and the consequent sequence expression are evaluated on the same clock tick, while for non-overlapped implication, the first element of the consequent sequence expression is evaluated on the next clock tick.

The sequence expressions can be more complex using special operators, such as delays or consecutive repetitions. Combining sequences is possible on both sides (antecedent and consequent). In addition, SystemVerilog provides coverage statements (cover property) in order to monitor sequences and other aspects for functional coverage. The simulator keeps a count of the number of times that the sequence is fulfilled or has failed [Dou17] [Meh14].

Property Specification Language

PSL is a language for the formal specification of hardware. It is used for depicting properties that are required to check the correctness of a DUT. PSL provides a means to write specifications that are easy to read and mathematically precise. Thus, PSL provides some tools to write assertions independently of the abstraction level and the design language [Ace04].

Listing 2.4: Example of PLS assertion

```verilog
assert always {Req} (next[1:2] Ack);
```

A PSL-specification consists of assertions regarding properties of a design under a set of assumptions. For instance, translating the code depicted in Listing 2.3
into PSL, implies using the PSL-specification. This process results in the code listed in Listing 2.4.

In addition, the property can be a sequential expression that checks signals and their consequence. Modifying the above example, where the acknowledge signal takes place in the next cycle of a request event, and then two cycles after one of the ena or enb (or both) signal is active-high, the PSL-assertion is as follows.

Listing 2.5: Example of PLS assertion

```
1 assert always {Req;Ack} (next[2] (ena || enb));
```

Open Verification Library

The OVL is composed by a set of assertion checkers, thus it is not a language, it provides some functions in order to verify specific properties of a design. OVL assertion checkers are instances of modules whose purpose in the design is to guarantee that some conditions hold true. The library contains 50 checkers, of which the syntax consists of one or more properties, a message, a severity and coverage. The supported languages are PSL, SystemVerilog, Verilog and VHDL [Ace14]. An example of OVL is shown in Listing 2.6.

Listing 2.6: Example of OVL assertion

```
1 ovl_next handshake (clk, rst, (req & ~ack), (~req & ack));
```

2.3.2 Universal Verification Methodology

One of the most popular methodologies is UVM, it is very widespread in companies, and is projected to continue growing the coming years [Fos16]. This methodology is an Acellera standard, developed by the main EDA-vendors. They combined their approaches to converge a universal solution. The main goal of this methodology is to run tests of hardware designs in order to verify their behaviour [boo11] [Gla09].

UVM based on constrained-random stimuli. This technique increases the productivity of engineers; when one uses this methodology, he must define how the stimuli are generated. Thus, the aleatory nature of the technique result in a number of scenarios of which some are unusual, this implies new testing scenarios that were previously not taken into account, achieving a high coverage degree. However, some scenarios will never take place.
Figure 2.5 illustrates a process that these kinds of methodologies must follow to carry out the verification process. We can observe that three important elements are necessary; a stimuli generator driven by pre-defined constraints, the DUT and a checker to check the correctness of the output.

![Constrained-Random Stimuli](Gla09)

Figure 2.5: Constrained-Random Stimuli [Gla09]

Figure 2.6 shows an overview of UVM. In this case, the testing environment has two agents, although it is possible to instantiate one or several of them. Each agent emulates the behaviour of another component that is not yet implemented, or is not interested in the simulation process.

![UVM overview](Gla09)

Figure 2.6: UVM overview [Gla09]

**Scoreboard** The main function of this component is to check the behaviour of certain DUT. The UVM Scoreboard compares the outputs of the DUT, propagated by UVM Monitors. It usually uses reference models to check the correctness of designs.

**Agent** A UVM Agent is a hierarchical component that groups other verification components whose main goal is to provide transactions and check them at pin-level.
The *UVM Agent* needs to operate both in an active mode (in order to generate stimuli) and a passive mode (in order to monitor signals) [Ace15].

- **Sequence item** A *sequence item* is a stimulus.
- **Sequencer** A *sequencer* exercises the DUT through random stimuli. Therefore, it routes the *sequence items* to the *driver*.
- **Driver** A *driver* translates *sequence items* into a pin-level.
- **Monitor** A *monitor* captures the transactions between the *UVM Agent* and the DUT.

**Environment** The *environment* is the container of UVM components. It groups together other verification components that are interrelated. Thus, this part defines the communication between UVM components.

**Test** It is the main responsible to configure the *environment* in order to run it correctly.

### 2.3.3 Universal VHDL Verification Methodology

UVVM, also called «UVM for VHDL», provides a methodology and library to simplify the entire verification effort. UVVM supports the same capabilities that other verification languages support, from transaction level modelling, to functional coverage and randomised test generation, to data structures and basic utilities. Figure 2.7 shows an overview of this methodology.

![Figure 2.7: Example of UVVM](image)

**Test harness** The *test harness* contains some verification components connected to the DUT. In addition, other interface signals are connected to the test bench.
top-level. In the example shown in Figure 2.7, the verification components are highlighted orange. The two processes that control the DUT from outside the test harness, are a *clock generator* and a *sequencer* [Bit16].

**Sequencer** The *sequencer* runs some commands that match with the test specification. These commands look like pseudo code and are implemented into the library provided by the methodology.

**Verification Components** A *verification component* consists of three main parts.

- **Interpreter.** The *Interpreter* checks the commands from the central test sequencer to see if a given command is targeted at this particular component. If so, it checks whether the command is to be put on the *Queue* for further handling by the *Executor*, or whether it is to be handled immediately by the *Interpreter*. The command could be a flush operation.
- **Queue.** The *Queue* is, like the name indicates, just a standard queue for commands to be forwarded to the *Executor*.
- **Executor.** The *Executor* executes commands contiguously, as long as there are commands available in the *Queue*. The *Executor* checks the command type and performs the requested operation. This operation is typically to write or read a register in a component or to transmit or receive data.

### 2.4 Related Work

As we mentioned at the beginning of this chapter, the purpose of this chapter is to provide the background of hardware verification. We have made a systematic review which helped us to obtain a list of related work. The systematic review process can be seen in Appendix A. The list from systematic review has been organised according to the different abstraction levels in the trade-off between simulation accuracy and verification effort. Figure 1.4 illustrates this trade-off across five groups: *HL-Modelling, RTL Simulation, Timing Simulation, Emulation* and *Prototype*. The following sections match with these five groups that contain the related work of this dissertation.

#### 2.4.1 High-Level Modelling Simulation

In the hardware domain, High-Level Modelling is very widespread to model and simulate hardware designs using high-level languages, which describe the design at a level of abstraction higher than RTL. However, this technique does not provide a cycle-accurate simulation. High-level modelling is accurate enough to verify the behavioural correctness of a design. Therefore, this group contains those works that try to simplify other methodologies or describe a solution at a high-level abstraction.
Facilitating UVM

Nowadays, verification languages, such as SystemVerilog [man09], are common for developing complex test benches, and for several verification methodologies that use these languages. The UVM provides a SystemVerilog base class library and guidelines, improving verification efficiency. However, the engineer’s experience is too important to carry out a hardware verification based on UVM.

To address and overcome a verification based on UVM, authors of work [YKKM11] propose a standardised and well-organised test bench architecture that includes directory structure of test bench files, and mechanisms such as interface and handles across the components. The proposed UVM application method helps test benches developers maintain the consistency of test benches and reduce the verification gap. It ensures that Intellectual Property (IP)-verification engineers do their job independently and that the test benches can be reused in a top-level verification environment. In addition, this approach provides a good infrastructure for those engineers that have little knowledge about verification languages and methodologies, only writing their test cases.

Summarising, authors address the verification challenges when UVM is applied to the SoC verification, in which many engineers with difference knowledge and experiences are involved. This approach provides the tools to generate a well-structured environment.

Transaction level assertions

In order to satisfy the complexity of verification needs, powerful verification languages and verification methodologies are employed. Levels of abstraction play a crucial role in the verification world, as the engineer needs to encapsulate a complex system behaviour in a few lines of code.

ABV plays an eminent role in tackling complex verification challenges. Languages like SVA add powerful constructs to encapsulate the temporal behaviour of systems. Authors of work [SBY11] propose a new method for doing transaction level assertions, by exploiting concepts of method ports and SystemVerilog scoping rules. Transaction level assertions provide a high level of abstraction. A complex temporal relationship between transactions sequences can be easily modelled and checked by SVA.

Figure 2.8 shows block diagram representation of the verification environment. Monitors in the class-based environment pass transaction through analysis ports that are connected to the export port of the assertion module. In this way, the authors are able to pass transaction from a class-based environment to a module without changing the level of abstraction.
2.4. Related Work

Summarising, this proposal is based on translating the elements of a UVM environment. This new verification approach is an efficient way of implementing transaction level assertions in a class-based verification environment. This environment can be reused in other projects.

**Reusing verification code**

Generally, hardware verification at different abstraction levels implies rewriting the test suite. This process is error-prone due to its handmade nature. By rewriting tests, these can be modified which causes them to be unequal to the original ones.

Authors of work [EA14] propose how the translation step can be rendered obsolete, by mapping the C-tests to the UVM environment. UVM sequences are normally used for testing, however the authors’ experience shows sequences are hard to understand for new users and users find them difficult in use. Furthermore, sequences encourage randomisation at every step, but the users’ experience has shown that randomisation is undesirable for many tests. Authors propose a solution based on the use of a mapping function from C to UVM and a helper class that tracks SystemVerilog threads and classes. In this way, the user can move easily between SystemVerilog sequences and C-tests. This approach provides a transparent way to reuse device driver C-code with a UVM-based agent verification environment.

Other work based on reusing tests is presented in [Put14]. The proposed solution simplifies the task of UVM-code reuse and provides an ability of complete reuse of C++ code across stages of the IP-core verification process as shown in Figure 2.9.
Summarising, reusing tests at different abstraction levels helps to exercise the DUT at the same manner. This kind of solution avoids time-consuming and/or repetitive tasks.

**Accelerating FPGA design validation**

Design validation is the most time-consuming task in the FPGA-design cycle. Although manufacturers and third-party vendors offer a range of tools that provide different perspectives of a design, many require that the design to be fully re-implemented for parameter modifications, or do not allow the design to be run at full-speed. Designs are first modelled using a high-level programming language and are later rewritten in an HDL. Authors of work [ICC10] provide a way of directly validating synthesised hardware designs with the original high-level model, taking away the traditional bit-level view of designs.

The Dynamic Modular Development framework provides a means to integrate a high-level functional model and a hardware implementation. Figure 2.10 shows the block diagram of this approach. The approach provides an online comparison of the reference model and the hardware implementation running on the FPGA. Both domains are exercised by the same input vectors.

Summarising, this approach provides a higher level of abstraction to hardware and enables complex testing scenarios practical only in software.

### 2.4.2 Register-Transfer Level Simulation

RTL simulation is the most popular level of abstraction for functional verification. An RTL-design is described using HDL, such as VHDL [IEE02] or Verilog
2.4. Related Work

Figure 2.10: Overview of block diagram of work [ICC10]

[IEE01], and focuses on signal interactions. Generally, the RTL code of a design can be synthesised to logic circuits. RTL simulation meets two important requirements. Firstly, RTL simulation is \textit{cycle-accurate}, thus the design behaviour is captured at each clock cycle. Secondly, the simulation is \textit{physically independent}, it does not rely on any information from the implementation process or the target device. However, it does not take resource location or critical paths into account. Therefore, this group contains those works that carry out functional verification processes (whose aim it is to check the design intent). This means that proposals do not take into account the implemented design features.

\textbf{Generation of input test vectors lead by constraints}

The verification of floating-point units is difficult to achieve, and the costs of post-production bugs are severe, see for example the division bug in the Intel processor [Pri95]. The difficulty of verification is due to the huge input values or scenarios. This leads to a number of test vectors and golden or reference vectors. Therefore, the authors of study [NF15] try to choose a subset of the test space and add coverage goals for how to smartly generate the desired test vectors. They propose a verification methodology for binary float-point arithmetic operations by writing SystemVerilog constraints to constrict the data path, starting from the operands through intermediate results and rounding techniques until the result evaluation. Then, they pass the constraints to the simulator tool to randomly generate test vectors based on the above constraint model defined by the user.

Summarising, this approach is compatible with any simulator that supports SystemVerilog constraints, and it has been included at this group because it is only focused on randomly stimuli generation. Authors make a functional verification of float-point operations.
**Bridging RTL and Gate levels**

Although HLS tools are very widespread and decrease the complexity of hardware designs, some functions, architectures and communication tasks are already built on RTL, using an HDL. In both cases, some design issues can only be discovered and resolved at the gate-level, such as timing intent. In this sense, authors of work [CCT07] present a comprehensive approach to establish correspondence design objects or elements between a Gate-level implementation and its golden reference model specified at RTL. The approach integrates a set of correlation techniques to compare both domains.

- **Naming Similarity.** In order to make the task of post-synthesis verification feasible, synthesis tools preserve certain name patterns, especially for hierarchical names and sequential objects (see Figure 2.11).

![Figure 2.11: An example of reverse name matching of work [CCT07]](image)

- **Structural similarity with topological analysis.** The transitive fan-in and fan-out of the object-interest can determine a list of correlation candidate objects in the other abstraction level so that they have similar transitive fan-in and fan-out cones (see Figure 2.12).

![Figure 2.12: Topological similarities by connectivities of work [CCT07]](image)
• Functional similarity with comparison simulation. It checks the behaviours of each structural similarity. It helps to eliminate candidates (see Figure 2.11).

![Figure 2.13: Correlation candidates by connectivities of work [CCT07]](image)

Summarising, this proposal establishes the correlation between signals and objects in a gate-level implementation and an RTL design through three different similarity measurements. By combining the three techniques, they are able to show the effectiveness of the correlation, increasing the visibility of gate-level.

### 2.4.3 Timing Simulation

At the middle of the accuracy-effort spectrum is timing simulation, which annotates timing information in simulation. Timing simulation assists in identifying incorrect timing information and checking timing intent. Timing simulation depends on the implementation process because the timing information is extracted from the implemented design.

Therefore, this group includes those works whose goals assess the timing correctness or propose a solution close to timing simulation. In this sense, TLM-based approaches are included in this group and even some hardware verification methodologies or techniques are included, although the simulation accuracy is not high. Figure 1.4 illustrates this group encompasses a big part of the accuracy-effort spectrum.

**Timing monitors**

Verification of physical properties related to specific design constraints of a given digital IP at RTL has several limitations. Firstly, the simulation performance at RTL suggests high timing-consuming. The verification of the RTL code, once integrated into a high-level system description of a smart system, requires co-simulation instead
of simulation. The concept of co-simulation emerged due to the need of efficiency and higher simulation accuracy.

Since many physical properties affect the digital IP-timing, the use of timing monitors allows their effect to be captured concurrently. In this sense, authors of paper [GPS14] propose a methodology for system-level verification of digital IPs augmented with embedded timing monitors. This methodology relies on three steps (see Figure 2.14).

![Figure 2.14: Overview of verification methodology proposed by [GPS14]](image)

- Given the RTL-model of the digital IP and sensor, which are implemented in HDL at RTL, an abstraction tool is applied to abstract them into SystemC TLM. This tool translates the RTL to Transaction-Level Modeling (TLM).
- A set of C++ functions are implemented to simulate timing delays in the digital IP. These functions, hereafter called mutants, are automatically injected in the abstracted digital IP to verify, during simulation, the sensor correctness.
- The abstracted and injected digital IP and sensor are connected to a stimuli generator, that aims to generate a set of inputs for the digital IP.

Temporal Decoupling for FIFO-based Communications

The transactional abstraction level can be subdivided into many coding styles, each according to their timing accuracy. A better timing accuracy allows the use of the TLM-model for early performance evaluations, but unfortunately induces longer development time. Thereby, timing annotations integrated into the TLM-model improve the timing accuracy of loosely-timed TLM-models. The temporal decoupling,
which lets processes advance their local time in the future until a synchronisation is required, is described in the TLM reference manual. However, this description focuses only on memory-mapped bus and stream-based subsystems based on FIFO-protocols. The work [HCG13] describes a novel technique that allows to add timing annotations in an untimed TLM-model, without increasing the number of context switches. To apply this technique, the authors have developed a special element, known as Smart FIFO (see Figure 2.15).

![Smart FIFO interfaces proposed by [HCG13]](image)

Figure 2.15: Smart FIFO interfaces proposed by [HCG13]

Figure 2.15 illustrates the three interfaces that implements the Smart FIFO. The read-side and write-side interfaces block read and write accesses, with additional methods and events for simulating non-blocking accesses. The Smart FIFO assumes that each side is always accessed by the same process; if this is not the case in the design, then an arbiter must be added to ensure that two successive accesses on the same side cannot have decreasing local dates. The last interface is related to a monitor interface, which can be used to debug and for dynamic performance tuning.

Summarising, using the Smart FIFO approach can add timing annotations in an untimed TLM-model, speeding up simulations without any loss of timing or functional accuracy.

**Dynamic ABV environment**

A number of techniques and frameworks has been developed to apply ABV to Electronic System-Level (ESL) design, particularly at TLM. Authors of work [BFPS15] try to fill in the gap by presenting a technique to automatically abstract properties defined for RTL IP, with the aim to create dynamic ABV environments for the corresponding TLM-models. These ABV environments are created automatically from a set of properties initially defined for a RTL implementation. Figure 2.16 shows the methodology overview.

To achieve their goal, the authors propose a methodology with two directions. Firstly, they automatically rewrite the cycle-accurate RTL-model into a set of properties suited to be checked on an event-based TLM-model. This is done by applying a set of transformation rules. Secondly, they define an approach to synthesise TLM properties into checkers to be adopted for the dynamism ABV of the TLM-model. This approach is independent from the methods applied to generate checkers due to the wrapper.
2.4.4 Emulation

Hardware emulation is a technique that many engineers use in verification stage. This technique tries to imitate the behaviour of a hardware element. The main goal of this approach is normally debugging and functional verification of the system that is being designed. In some cases, building an emulator is more complex that an in-hardware verification. Therefore, this group contains those approaches that use or build an emulator platform, also known as virtualized platform.

Hybrid verification framework

Logic emulators can easily accommodate large designs and execute them at speeds two to three orders of magnitude higher than of software-based simulators. Emulation achieves speeds that closely matches the speed of a fabricated chip. However, emulation systems provide limited visibility of internal nodes, which makes debugging difficult. Hybrid systems harness the advantages of simulation and emulation. In this sense, authors of work [BGJ11] propose a technique that reduces the overhead associated with periodically checkpointing the design during emulation. To achieve a solution, authors have built a hybrid verification framework that comprises an FPGA-based emulator, software simulator and a verification controller program (see Figure 2.17). A design compiler and slicer program prepares the design for emulation and simulation. The verification controller is a program which serves as the kernel of the system. It controls design execution and error detection in emulation, manages saved checkpoints and traces, transfers the design to simulator on error detection and initialise a piecewise simulation run for debugging.

Forced assertions

It is important to minimise the iterations through design recompilation or FPGA reconfiguration process for validating repeated debugging change, to improve the debugging turnaround time of complex SoC designs on FPGA-based logic emula-
2.4. Related Work

Figure 2.17: Hybrid verification framework proposed by work [BGJ11]

Figure 2.18: Forced assertion based debug flow proposed by work [BGG13]

2.4.5 Prototype or In-Real Hardware

This group contains works and projects whose verification accuracy is too high. In this case the testing process runs the implemented design on the target device under real-word conditions, namely the DUT is implemented according to the real resources and their connections. However, this group presents a big challenge, because a bug on the implemented design requires extra effort to trace the cause of this bug. For instance, engineers must insert probing components of vendor tools into their designs, or must build special verification components. This means a re-implementation every time a different configuration would be probed, making it very time-consuming. Moreover, since probing logic can only visualise a limited number of signals for a limited period of time, it involves more iterations to identify the source of a bug than other simulation groups.
**HPChecker**

The work [LWH08] faces an integration challenge. Designer integrates his own IPs with a third-party IPs into the system and must make sure his IP works correctly after the integration. Authors denote many errors may occur in real-time because monitor-based approaches often cannot find errors in a simulated environment. Authors try to overcome this challenge and provide more efficient ways to debug the system. In this sense, authors propose an AMBA AHB bus protocol checker based on a monitor method (**HPChecker**).

**HPChecker** is able to find errors efficiently and rapidly in a real-time environment. This monitor-based approach has 73 rules that include master, slave, reset, bus components and performance issues. The architecture of the **HPChecker** is shown in Figure 2.19 and it contains four main function blocks.

![Figure 2.19: HPChecker Architecture of work [LWH08]]

- **Protocol Checker.** This part contains a set of well-defined rules. It is the main core of **HPChecker**. Every rule has its own corresponded bit, because in every cycle more than one error can occur. If an error occurs, the **HPChecker** will output the corresponded master ID-number or slave ID-number to indicate which of them violated the AHB protocol.

- **Configuration Registers.** It lets designers set some parameters, including mask, protocol checker enable and max waiting cycle.

- **Error Reference Table.** This table indicates which rules are not met by a concrete slave or master. Thus, authors provide an error reference table that summarises which errors have occurred.

- **Windowed Trace Buffer.** This buffer stores the trace data until the first error occurred. Therefore, it traces the history signals after an error occurred.
2.4. Related Work

Summarising, this proposal only checks the handshake of an IP-core connected to an AMBA AHB bus at run-time. Thereby, the signal timing of an IP is checked, however its functionality is not verified, and it is possible that an IP can fulfil the time rules while its behaviour is incorrect.

Co-simulation platform

Authors of paper [CRL10] propose an approach based on FPGA-assisted co-simulation because it created an accurate and efficient debugging method. This approach provides full visibility to test bench in an HDL simulator and also preserves a high simulation speed because DUT is executed on the hardware side. However, the approach suffers from debugging problems, and an example of this are the hidden DUT internal signals.

Therefore, authors propose a RTL debugging method for an event-driven FPGA-assisted co-simulation system. It achieves 100\% observability for DUT in FPGA. Figure 2.20 shows an overview of the developed platform. An HDL simulator was used to run test bench and other remaining blocks on a host. On the hardware side, the PCI bus is extended to an FPGA internal bus, and each DUT is connected to the bus with its ports mapped to different bus addresses. On the software side, a DUT wrapper module is used to replace the original DUT-module, thus the DUT-wrapper translates the stimuli into PCI-messages that contain these stimuli. In addition, they propose a debugging design flow to use their debugging platform. Firstly, the RTL code (DUT and test bench) is parsed to get the verification version; a modified DUT and a modified test bench. Both include a communication mechanism that allows to connect the DUT running on an FPGA to the simulator tool that contains the test bench.

![Figure 2.20: Platform overview of work [CRL10]](image)

Summarising, this work proposes a novel co-simulation method which involves two domains; a software domain to run the test bench in a simulator tool and a hardware domain to run a hardware design. However, this work does not explain how it builds the hardware side and it does not take in account about how much time the synthesis process takes.
FPGA Verification environment based on UVM

Hardware verification methodologies are becoming increasingly popular in the hardware domain. One of these methodologies is UVM, which uses SystemVerilog as a programming language. One drawback of this methodology is its verification accuracy, and although it is not very poor, it does not achieve the highest verification accuracy. In this sense, authors of work [PiCK15] try to achieve better accuracy. They propose a verification environment based on UVM, because they believe the UVM works well for unit level verification but does not take software embedded into processors into account.

The authors present an automated FPGA-prototyping and accelerated verification of these systems, while the accelerated verification environment corresponds to the principles of UVM. Figure 2.21 shows the architecture of the accelerated verification environment approach. It should be noted that almost all UVM components are moved into the FPGA, except for the reference model and scoreboard. Therefore, UVM agents are replaced by Hardware Agents. UVM test bench, Reference Model and Scoreboard are running in software simulation and the remaining parts are running in FPGA. Communication between both parts of the verification environment is accomplished using authors’ framework. It encapsulates the hardware part, allowing external communication.

![Figure 2.21: Architecture of accelerated verification environment of work [PiCK15]](image)

Summarising, this work provides better accuracy of pure software environment based on UVM, reducing the time of verification process. In addition, the proposal is understandable for verification engineers. However, it does not provide all facilities to build a verification environment and they do not include the synthesis time in their results, thus their solution is more time-consuming.
2.4. Related Work

**FPGA verification environment for RTCA DO-254**

One of the challenges that a verification engineer faces, is the standardisation of his products complying with a specific standard, such as RTCA DO-254 [men09]. This standard is a means of compliance and guidance for the design assurance of complex electronic hardware, such as FPGAs, in airborne systems. Section 6 of this standard relates to verification processes and defines a set of verification objectives and methods that present several new challenges to design and verification engineers.

The work [LZ11] addresses these several verification challenges according to the objectives imposed by RTCA DO-254, such as traceability of testing results, creation of test vectors, automation of the verification process among other. Authors propose a methodology based on a bit-accurate in-hardware verification platform that is able to verify and trace the same FPGA-level requirements from RTL to the target device at full speed.

Figure 2.22 shows the in-hardware verification process proposed by the authors. The input vectors are uploaded from the workstation to FPGA via PCIe interface. Once all of the vectors are stored in FPGA DDR, the testing process is ready to start. The testing process is automated by a specialised application capable of reading and applying the input vectors file during in-hardware verification. The results obtained are sampled at full speed and recorded in a waveform file called «output vectors». Another specialised application automatically compares the in-hardware verification results with the RTL simulation results (golden vectors).

![Figure 2.22: In-Hardware verification process of work [LZ11]](image-url)
Summarising, the proposal of this work is very close to our proposal, because it tries to address the same verification challenges. The proposal is too similar to a black-box verification technique, where the model reference and the DUT use the same input vectors and their results are compared afterwards. However, this approach does not give any information about timing intent, in addition, previously to in-hardware verification someone must manually verify the design at RTL-level and make sure that the result is correct.
Chapter 3

Hardware Unit Testing

«The value of an idea lies in the using of it»
Thomas A. Edison

3.1 Unit Testing
3.2 Hardware Objects
3.3 Hardware Verification Platform
3.4 Integration with the typical design flow
3.5 Summary

This chapter describes how unit testing is applied in logic programmable devices. Our solution is based on high-level modelling, which is becoming increasingly popular for building new hardware designs. It allows to describe hardware functionality in a high-level programming language, such as C or C++ [CLNN10]. These descriptions match with the DUT and are then translated into low-level cycle-accurate RTL [CLN11], then it is synthesised to exercise the DUT in real hardware, addressing the poor verification accuracy that high-level modelling provides [GD15].

On the other hand, author’s experience work [EA14] has shown that for many tests, randomisation is undesirable. Therefore, the solution needs something to cover all scenarios or at least needs to verify the developed code. Unit testing drives functional verification process and only gets the interesting scenarios. Testing frameworks help to build unit tests and bring some facilities to check the designer intent and the correctness of his design through an amount of pre-defined functions.

The solution presented is explained by a case study, to understand how our approach can be introduced in the hardware design domain. Our case study is based on Histogram of Oriented Gradients (HOG). The HOG is a feature descriptor used in computer vision and image processing for object detection, particularly suited for human detection in images [DT05]. This case study is evolved according to the different scenarios addressed in this dissertation.
3.1 Unit Testing

At a high-level description, unit testing relates to the practise of testing certain functionality (units) of our code. This allows us to validate that our functions work as we expected (engineer intent). In other words, for a particular function and a set of stimuli, we are able to determine the correctness of a function by its return values. This enables us to identify failures in our designs, testing each chunk of our code through small tests called unit tests, and these unit tests must cover all possible paths to obtain a good code quality. When we have written a number of unit tests that cover 100% or is very close to that, we have created a test suite. In this sense, unit testing is a way to easily verify our code, if our code supports that particular type of evaluation.

Another issue that covers the unit testing approach, is the breaking functionality after a code change. Writing solid unit tests and well-tested code allows to address an error when a failure happens, because the test suite can be run at any time to verify the code. The following list enumerates the main advantages of developing from a unit testing perspective.

- The code is written to be easily testable. Thus, the code must support this particular evaluation. The code has to be kept clean and well structured, showing professional pride in workmanship and an investment in future ease of modification. The purpose is to make less work by creating code that is easy to understand, to evolve, and to maintain by others and ourselves.

- No production code exists without corresponding tests. The production code is verified with a high coverage degree.

- Differences between requirements and actual functionality are always evident.

- Small and large logic errors after changes in previous code are detected quickly.

- Unit tests are always written against interfaces. This favours modulation and separation of responsibilities.

- The tests keep track of exactly what is working and how much work is done. This creates an extra parameter for estimating and gives a definition of «done».

In embedded domain, developers face the same verification challenges and can expect the same benefits as described above. However, in the pure hardware domain engineers do not have any similar techniques that bring this goodness. Thus, hardware engineers need a solution to get the same results, plus new benefits specific to this domain, such as reducing risks or reducing debugging the hardware target device (where problems are more difficult to find and fix).
Unit tests have several valuable characteristics, and some authors consider these critical attributes. These attributes are known as FIRST rule [MM06]:

**F** Fast: Running tests fast enough to not be a practical problem for developers.

**I** Independent or Isolated: To avoid any dependence on other tests. One test does not set up the next test.

**R** Repeatable: The test must be repeatable. It must return the same result when it is run in loop.

**S** Self-validation: Test must return a Boolean result (pass or fail), without subjective considerations detecting if passed or not.

**T** Timely: Unit tests should be written just before the production code, preventing bugs.

### 3.1.1 Testing frameworks: Unity

Unit testing frameworks are related to unit tests, holding test code separate from code production. A unit testing framework consists of a library and a test runner. On the one hand, the library provides pre-defined functions for comparing expected and return values from a functional code. It collects and reports the test results to inform developers. These functions are more commonly called asserts, and developers must choose the right one to check the correctness of their designs. On the other hand, the test runner runs each test case following a sequence of steps: setup, run test case code, tear-down and, finally, report the test result. Thus the test runner is the core of unit testing frameworks. The common structure of testing frameworks is shown in Figure 3.1.

![Figure 3.1: Overview of unit testing frameworks](image)

There is a long list of testing frameworks for a number of programming languages, for instance, CppUnit for C++, igloo for C or JUnit for JAVA. Choosing the
best testing framework for engineers’ aim depends on certain criteria, such as portability. For instance, considering different environments in applications for embedded systems, portability and adaptability are a main requisite.

In this dissertation, *Unity* is our reference unit testing framework and it is especially designed for embedded software. *Unity* is written in C and a couple header files, which provide functions and macros to make testing easier. Most of the framework is a variety of assertions, which are meant to be placed in test to verify the code production. *Unity* is portable and fairly cross-platform, it is equally happy running tests for an 8-bit micro-controller as in a 64-bit processor, for instance into an embedded ARM. *Unity* is easily expanded, building new macros to fulfil new verification requirements [KMG]. For all these reasons this unit testing framework is a good option for our proposal.

*Unity* framework provides a number of assert functions to check the correctness of a production code. For instance, `TEST_ASSERT_EQUAL` assert checks the equality of two values. Through this, engineers can check the result of his functions with the golden results.

### 3.1.2 Test-Driven Development Methodology

Novel methodologies, such as *Test-Driven Development* (TDD), have appeared from unit testing. TDD inverts the traditional development/test cycle, driving the development by testing instead of writing the functional code first and later testing it. TDD flow is divided into three important states: red, green and refactor (see Figure 3.2). Red and green refer to the colours sometimes used in a unit test framework, indicating respectively test failure and success.

![Figure 3.2: Test-Driven Development flow](image-url)
The main objective of agile development is to reduce the officialdom stages of traditional development methodologies. The agile development contains some ideas from *Scrum* and *eXtreme Programming* which are listed in the agile manifest [MKCF].

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

This manifesto for agile software development can be extended to hardware development.

**Create test**  Firstly, a developer writes a test case, and the feature specifications are translated into an executable test. For instance, with a factorial algorithm, a first test looks like Listing 3.1. Note that the test is written in C for the *Unity* testing framework.

```c
void test_Natural()
{
    TEST_ASSERT_EQUAL(720, factorial(6));
}
```

The test flow consists of four ordered steps:

1. *Setup of the test*. At this step the necessary objects are created, values are set, connections are made, ... in order to start at initial state, which is common to all test cases.
2. *Executing the function under test*. The function is called upon and its results are collected for the next step.
3. *Asserting the results*. The results obtained in the previous step are compared with the golden values by assertion functions.
4. *Tear down of the test*. At this step the object created, the set values, ... in the first step are reset to get the state before executing the test.
Although, Listing 3.1 does not show the setup and tear down steps, the testing framework provides two functions to carry out these tasks (see Listing 3.2).

### Listing 3.2: `setUp` and `tearDown` functions

```c
#include "unity.h" include "factorial.h"

void setUp(void){}
void tearDown(void){}

//Test Cases
```

**Red** After an executable without failures, which contains the test case and the enough code, it becomes feasible to run the test and see the failures. Note that the new feature is not implemented yet. This means that engineers must develop the code for this new feature.

**Green** The engineer must provide a minimal implementation in order to pass the test. Passing the test means returning the result, that was expected (see Listing 3.3).

### Listing 3.3: `Factorial` code production

```c
int factorial(int n)
{
 return 720;
}
```

The first steps might be trivial tasks, but TDD is a cyclic process, so the next step is to write a new test or new feature, which means the current implementation is not sufficient. For example, we can add a new assertion function to the test case, defined in: Listing 3.1, like this one: `TEST_ASSERT_EQUAL(1, factorial(1))`. The subsequent tests might provide a more realistic solution.

At this point, after the code passes all test cases, one can start over, picking a new feature to implement, or refactoring the current implementation if it is necessary, including code and test cases.

**Refactor** Refactoring means rewriting code for its readability or removing duplication without changing or adding the behaviour. This step is necessary, because some implementations are not good in optimal terms. After this task, the engineer has to ensure that the behaviour has not been altered. It means if a failure occurs, it must be solved in quick order or the changes must be reverted.

This process is repeated until the developer covers the design intent [SW08]. The amount of test cases written can be added to an automated test suite and these, or a subset, can be run later.
3. Hardware Unit Testing

Unit Testing

Unit Testing is the level where individual units/components of a design are tested. The purpose is to validate that each unit of code performs as designed. The third-party dependencies are managed by special components called Test Doubles. Functions composed by nested functions (top-functions) are considered as black-boxes, and hence engineers perform black-box testing that is based on the specifications of the design that is to be tested (Functional Testing).

Integration Testing

Integration Testing is defined as the testing of combined parts of a system to determine if the design works correctly. Thus, all modules are combined and tested as a group.

System Testing

The purpose of this test is to evaluate complete designs with their specified requirements. This group includes non-functional tests, such as stress tests.

- **Load Testing** It is a process of testing the behavior of a design by applying maximum load in terms of accessing and manipulating large input data.
- **Stress Testing** The aim of stress testing is to test designs by applying the load to the system and taking over the resources used by designs to identify their breaking points.

Acceptance Testing

Acceptance Testing is a level of the testing process where a system is tested for acceptability. The purpose of this test is to evaluate the system’s compliance with the business requirements and to assess whether it is acceptable for delivery.
TDD principles provide a number of advantages, and they can be applied in the hardware domain using HLS tools. However, these TDD principles can only be applied at high-level modelling and cannot be applied at the whole development hardware flow, due to amount of time of the synthesis process, making this methodology unfeasible. This feeds some authors, that think high-level modelling is more often used as a fast simulation method than a synthesis approach [GD15], whose main drawback is the need to create a fast simulation model which is not used in the next steps of the development cycle [IPC14]. Hence, to carry out unit testing technique and its principles in a pure hardware domain requires a high effort because engineers cannot apply them directly. This dissertation tries to give a solution addressing the hardware verification through unit testing, reusing the same test suite at different abstraction level.

3.2 Hardware Objects

Unit testing cannot be applied directly to high-level modelling in a hardware domain, due to the restrictions imposed by HLS tools. HLS tools impose the following constraints [Xil12a].

- A top-level function must be defined. Its arguments establish the RTL input and output ports.
- Synthesises hierarchy C functions results in an RTL-design that has a hierarchy of modules or entities which have a one-to-one correspondence with the original C function hierarchy.
- Loops in the C function are rolled by default. This means synthesis creates the logic for one iteration of the loop, and the RTL-design executes this logic for each iteration of the loop in sequence.
- C arrays are synthesised arrays as block-RAM. If the array is on the top-level function interface, it is implemented as a port to access a block-RAM outside the design.

The first constraint limits any hardware design in high-level modelling. It makes most designers speed up their algorithms in a single function, however this practice is not recommended. The code must be organised into small functions in order to provide readability and give a specific interface [Xil12a] [Men10] [Mar08]. In hardware, a code organised in functions allows for application of other techniques, such as function pipeline or reusing resources. Therefore, the top function that contains another functions, usually builds a dataflow.
To apply unit testing in the hardware domain, we need to ensure individual accessibility of each function depicted in the code related to the hardware design. This fact fulfills unit testing principles. Therefore, the top function must route the stimuli to the correct function for exercising it. Our proposal is based on OOP that meets all requirements imposed by the unit tests principles (F.I.R.S.T. rule) and HLS tools.

From a logic point of view, an object encapsulates a set of methods that offers a specific functionality, which can be called upon from outside, and a stored state of itself [CKU04]. A method is defined as an operation or service offered to a third-party, which requires some parameters and returns a result. All methods of an object define its Application Programming Interface (API). On the other hand, the status object is defined by the values of its attributes. These attributes denote the current status of an object in a determined time, allowing for it to be stored.

Our solution is based on Object-Oriented Communication Engine (OOCE) work [Bar08], which has been developed by the ARCO research group. OOCE considers all elements of a SoC as a distributed system, and these components are modelled as distributed objects. In the hardware domain, a hardware component can be modelled as an object, denoting as hardware object. Although we do not use the pure OOCE approach, we can affirm that our solution is closer to OOCE approach. Before we explain our approach, we are going to describe the case study that will be used to explain our approach.

Our case study is based on HOG. The HOG is a feature descriptor used in computer vision and image processing for object detection, particularly suited for human detection in images. The algorithm implementation is divided into different steps: gamma and colour normalisation, gradient computation, orientation binning, illumination and contrast normalisation, block normalisation and a support vector machine (SVM) classifier (see Figure 3.3).

![Figure 3.3: Main stages of feature extraction and object detection chain](image)

In our case study, the step chosen is the vector normalisation with the normalisation factor $L^2$-norm (Equation 3.1). Let $v$ be the unnormalised descriptor vector and $\epsilon$ be the small constant [DT05].
We propose a solution for $l^2$-norm algorithm that is shown in Figure 3.4 along with its block diagram. The $l^2$-norm algorithm takes a single-precision float-point vector from the previous step as input, whose depth is 16 data, and returns a normalised single-precision float-point vector with the same depth. The solution has been divided into three steps: `sum_hist_pow`, `scale` and `mult_hist_scale`. The `sum_hist_pow` step reads 16 float-point data from input FIFO, these data correspond with a small window of 16 pixels. Each pixel is powered and the result is added to the preceding addition, and the original pixels are stored in an internal FIFO to ensure sequential accessing. The next step, called `scale`, calculates the scale factor with the summation obtained in the `sum_hist_pow` step. Finally, in `mult_hist_scale` step, the scale factor, result of the previous step, is multiplied by each original pixel, which are now stored into the internal FIFO, in order to get the normalised vector which will be written in the output FIFO. The algorithm does not support a pipeline, because each step depends on the previous one.

```
void l2_norm(float histIN[HIST_SIZE],
             float histOUT[HIST_SIZE])
{
    float scale = 0.;
    float sum = 0.;
    float histAUX[HIST_SIZE];

    loop1: for(int i=0; i < HIST_SIZE; i++){
        sum = histIN[i]*histIN[i];
        histAUX[i] = histIN[i];
    }
    scale = 1/(sqrt(sum) + HIST_SIZE * 0.1)

    loop2: for(int i=0; i < HIST_SIZE; i++)
    {
        histOUT[i] = histAUX[i]*scale
    }
}
```

Figure 3.4: $l^2$-norm algorithm and block diagram

In addition, a top function is defined to orchestrate all functions defined above. This top function will be the entry point of our hardware component and allows the data flow. Actually, this function calls upon each function tidily as Listing 3.4 illustrates. Each function must be verified in a pure software domain in order to check its functional behaviour.
3. Hardware Unit Testing

Listing 3.4: Top function of HOG case study

```c
void l2_norm(float histIN[HIST_SIZE],
             float histOUT[HIST_SIZE])
{
    #pragma HLS INTERFACE ap_fifo port=histIN
    #pragma HLS INTERFACE ap_fifo port=histOUT

    float histAUX[HIST_SIZE];
    float sum=sum_hist_pow(histIN, histAUX);
    float scale_ret=scale(sum);
    mult_hist_scale(scale_ret, histAUX, histOUT);
}
```

At this point we have defined the interface of the top function. According to the solution proposed in Figure 3.4, the best option is an interface based on FIFO for input and output ports. However, if we ignore lines 5 and 6 of Listing 3.4, the HLS tool will translate these parameters as a port to access a block-RAM. To get an interface based on FIFO, we use a `pragma` that allows us to define the used resource. We specifically use the following pragma: `#pragma HLS INTERFACE ap_fifo port=<port>`. After the whole design is defined, we must overcome the individual accessing of each function. This task is more complex and is addressed using a new approach that we denote as `hardware objects`, encapsulating all functions in a single component. In addition, we need a communication mechanism to recognise which method must be exercised and to separate the functionality from the communication tasks. This communication must be independent from the bus that the component is attached to. The following sections describe how we overcome these challenges.

3.2.1 Hardware Encapsulation

Since a `hardware object`, which matches with the DUT, is composed by some functions that are accessible by the same entry point, the data must route from this entry point to the correct function. Firstly, we must define the entry point of our hardware object, this entry point has some features, highlighting its universality. The entry point must be independent of any standard bus, such as AXI, to ensure full compatibility. In this sense, we propose a symmetrical entry point formed by a streaming input and a streaming output, as shown in Figure 3.5.

The port’s interface is based on FIFO signals. For the input channel we use three signals; the input data signal whose width is 32-bit words, the hardware object knows that there are data ready when the empty signal is active-high and it reads the data through the rd signal when it is able to attend the request. On the other side,
3.2. Hardware Objects

Figure 3.5: Overview of feature extraction and object detection chain

the output channel, we find another data signal which returns the hardware object results, denoting its validity by a wr signal. The actor which will receive the result data must be ready, the actor denotes that it is ready by communicating the full signal. Although Figure 3.5 shows two FIFOs, they are not required. Our hardware object is able to synchronise with other entities through these signals. We do not need the FIFOs because a hardware object is able to perform its tasks without copying data from one memory area to another. From a behavioural point of view, our hardware object works as follow, we suppose that it instantiates the two FIFOs.

1. From outside, someone writes a message into the input FIFO (FIFO IN), including a small header (see section 3.2.2) and stimuli for exercising a function. A header identifies a function implemented inside a DUT, while stimuli are the function’s parameters which must be serialised.

2. Then, the facade module, which is the main responsible to route the stimuli to the correct function, reads from input FIFO two 32-bit words that correspond to a header message. The module checks this header to know which function must be called upon to exercise it and the stimuli size that it has to read for performing this action.

3. At this point the function is running and the other modules are waiting to end the function execution.

4. Finally, the function results are stored into an output FIFO. In this sense, the actor who sends the request message, in our case a test case, can retrieve the result reading the output FIFO.

Therefore, Listing 3.5 shows the C code related to the new top function, which is a dataflow. The data from the input channel is sent to the facade function and when the result is ready, it sends the return data through its output channel.
After the entry point definition, we have to route the data received from the input port to the correct function. To address this routing, we must know where we have to send the data, thus the streaming data contains a small header to route the data. This header is explained in the next section, but now it is necessary to know that a header precedes the data. The request header is a 64-bit word which contains some identifiers among other attributes. These identifiers help us route the data. The data that receives and returns the hardware object is composed by a header of a 64-bit word, followed by a payload that matches with the stimuli. We denote this composition as a message (see section 3.2.2). Following the header is the data that complies with the function’s parameters or stimuli. These parameters must be serialised, obtaining a sequence of bytes which compose the payload (see section 3.2.3).

From a functional point of view, messages are read by the facade function placed inside the hardware object. This facade function works as the structural facade pattern hidden in the complexity of communication tasks and hardware encapsulation [GHJV94]. The facade function provides a simple interface which receives data and checks its header. For these headers, the facade function is able to determine which function must be called upon or if to forward the message. If all identifiers match with one functionality, the hardware object is able to handle the data. When an identifier does not match with the object configuration, the data is rejected. Therefore, the facade function is the main responsible to route the data from the input port to the correct function. Note that it can be see as a wrapper communication. Listing 3.6 depicts the facade function in C programming language for $l^2$-norm algorithm.

```c
void topTesting(hls::stream<unsigned int> FIFO_IN,
                hls::stream<unsigned int> FIFO_OUT)
{
    read_from_input(FIFO_IN);
    facade();
    write_to_output(FIFO_OUT);
}
```
3.2. Hardware Objects

Listing 3.6: Template of facade function for $l^2$-norm algorithm

```c
void facade()
{
    getRequestHead();

    if (match_nodeID_and_objID()){
        if (ID_OBJ_scale == header_OBJ.methodID)
            // Call upon scale function
        else if (ID_OBJ_sum_hist_pow == header.methodID)
            // Call upon sum_hist_pow function
        else if (ID_OBJ_mult_hist_scale == header.methodID)
            // Call upon mult_hist_scale function
        else if (ID_OBJ_l2norm == header.methodID)
            // Call upon l2norm function
        else
            // Forward
    } else
        // Forward
}
```

After the entry point and the routing of the data are defined, we must define how the data is translated into function arguments. In Figure 3.5, when the identifiers match with one function implemented, a function is called upon directly, however this is not done exactly this way. In reality, the facade function calls upon a new top function that envelops a user function. For instance, the scale method is not called upon by the facade function, the facade function calls upon the top_scale method, which is a function-wrapper instead of the original method. This wrapper-function sequentially carries out three stages: it reads parameters, calls upon the user function and writes return values. The following lines describe the elements necessary to build this new wrapper.

Function parameters  Firstly, we must define two structures for each function that keep input arguments and output values. In addition, we must describe the direction of each argument when it is necessary. For instance, the direction of an array type of argument is not defined, so we must define its direction. Listing 3.7 shows the two structures for the scale function, whose signature is float scale(float sum). For the input structure (PARAM_scale), we define the input arguments of user functionality, in the example we only define a single-precision floating-point according to the scale argument. On the other hand, the output structure, called RETURN_scale, has all return parameters defined by the developer, in our example a single-precision floating-point. Furthermore, these structures contain the padding when it is necessary, for instance the short foo(int i) function,
whose return parameter is a short type and needs padding space. This means the return structure contains two attributes: the return type and the padding. The return part is a short, while the padding width is 16 bits. Thus, the return message will be aligned to 32-bit, fulfilling the width of the entry point defined above.

Listing 3.7: Input and Output structs for scale function

```c
struct PARAM_scale{
    float sum;
}__attribute__((packed));
static struct PARAM_scale args_scale;

struct RETURN_scale{
    float _return;
}__attribute__((packed));
static struct RETURN_scale ret_scale;
```

Function wrapper The function wrapper matches with a top function, which is the main responsible to call upon the real user function, before this wrapper reads the function arguments from the input (reading state). Then, the wrapper is ready to call upon the real user function (running state) and when the results are ready, it writes them to output (writing state). Thus, this flow goes through the three states. Listing 3.8 shows the function wrapper of the scale function, the input channel corresponds to a stream of data whose size is 32 bits, according to the entry point’s width. The same applies to the output channel. The input data is read by the readParameters_scale function to get the function arguments, these arguments are used to call upon the real user function, in our case scale function. Finally, the writeReturn_scale function writes the function results in the output channel.

Listing 3.8: Function wrapper for scale function

```c
void testing_scale(hls::stream<unsigned int> &src,
                   hls::stream<unsigned int> &dst)
{
    readParameters_scale(src);
    ret_scale._return = scale(args_scale.sum);
    writeReturn_scale(dst);
}
```
Reading function  At the reading stage, we read from the input channel the function arguments and store them into a structure. Firstly, all data is read from the channel input in 32-bit words and is stored into a local variable array with the same width (\texttt{words32} variable of Listing 3.9). Then, a pointer is defined to translate the byte sequence stored in \texttt{words32} into the function parameters, in the order that it was serialised. We denote this process as hardware casting (lines 10 and 11) and it is explained in section 3.2.3. Finally, the result from the special function is stored into the input structure (line 11).

Listing 3.9: Reading function for \textit{scale} function

```c
void readParameters_scale(hls::stream<unsigned int> &src) {
    int words32[sizeof(args_scale)/BUS_WIDTH_BYTES];
    for(short it=0; it != sizeof(args_scale)/BUS_WIDTH_BYTES; it++)
        words32[it] = src.read();
    // Casting
    unsigned int *ptr = (unsigned int*)words32;
    args_scale.sum = toFloat(*ptr++);
}
```

Writing function  The last stage is the writing stage, in which the real user function results are written in the output channel. These results are stored into an intermediate variable and then serialised into a byte sequence, including the padding required. Like the reading stage, this stage translates the output values by special functions, which convert from a standard C type to a sequence of bytes. Finally, the sequence of bytes is written in the output channel. Listing 3.10 shows the writing function for the \textit{scale} function, whose return type is a single-precision floating-point. To translate a C type to a sequence, we use a special function called \texttt{toSequence} (line 8), this function depends on the pointer defined (line 6). This process is explained in section 3.2.3 as well. Subsequently, the result is stored into a function union that is able to transform the reading size. Finally, we read 32-bit words from the union and send them to the output channel (lines from 12 to 15).
3. Hardware Unit Testing

Listing 3.10: Writing function for `scale` function

```
void writeReturn_scale(hls::stream<unsigned int> &dst) {
  short index=0;

  unsigned int ptr_ret[sizeof(ret_scale._return)/SCALE_WRITE_SIZE_BYTES];
  toSequence(ret_scale._return, ptr_ret);
  for(short it=0; it!=sizeof(ptr_ret)/SCALE_WRITE_SIZE_BYTES; it++)
    byteRet_scale.words32[index++] = ptr_ret[it];

  for(short itReturn=0;
      itReturn != sizeof(byteRet_scale.words32)/BUS_WIDTH_BYTES;
      itReturn++)
    dst.write(byteRet_scale.words32[itReturn]);
}
```

Summarising, in order to reach the hardware encapsulation feature, each function is wrapped by a double wrapper; the first one refers to the communication mechanism, routing the messages to the correct function or rejecting them, and hardware encapsulation, which carries out the serialisation of payload. Thus, the `facade` function, really runs a top-function instead of a user function. In this sense, our approach does not modify the original code implemented by a designer, we only add two wrappers (double-wrapper) to ensure individual function accessibility and allowing multiple tasks for the same `hardware object`.

Figure 3.6 shows the translation process from the code described from C programming language to the final hardware component after using an HLS tool, before synthesizing the code. We introduce the double-wrapper described above using our own tool: `c2hwobject`. Indeed, we do not need all user code to build the double-wrapper, we only need the signatures that compose the object. Logically, to build the full `hardware object` we do need all the user code.

c2hwobject: An automated tool

At this point, our approach has a lot of repetitive tasks that can be automated such as hardware encapsulation, facade module and other elements. These modules are automatically generated from function signatures and some `pragmas` associated to each signature. The `c2hwobject` tool generates them, adding the two layers: communication and hardware encapsulation.
3.2. Hardware Objects

The `c2hwobject` tool is written in C++ programming language and its output is a synthesizable C code and some virtualisation files of the hardware object. All code has to be generated before the synthesis process, because hardware components do not offer code manipulation at run-time. The modules are generated in three steps (see Figure 3.7):

- The parser parses all header files and extracts information into an Abstract Syntax Tree (AST). This process is based on the `clang` library. Clang is a compiler frontend that uses Low Level Virtual Machine (LLVM) as its back-end [App].

- The AST is passed to the generator, which builds a hardware object code from this AST. The generator uses the `Ctemplate` library, which implements a simple but powerful template language for C++. It emphasises separating logic from presentation [Goo].

- Finally, the hardware object code is composed by the functionality depicted by engineers and an adapter that allows engineers to access each function. These files contain a synthesizable code in C programming language, which will be used
by an HLS tool to translate the generated code into an HDL, and then it will be synthesized and deployed into a hardware device. In addition, some virtualisation files are generated as well. These files are virtual representations of the hardware component will be used in different abstraction levels.

3.2.2 Communication Protocol

After the entry point is completely defined and our hardware object approach is presented, we must depict how these hardware objects communicate. In hardware, engineers usually use a standard bus as a communication channel, using its protocol specifications and addressing each hardware component into the memory address range. For example, the AXI4 protocol defines five different channels that are divided into two groups: two channels for readings and three for writings. In the reading group, one channel is the main responsible for addressing and controlling the communication while the other one is the data channel, thus the AXI4 protocol provides separate data and address connections [Xil11]. Our approach uses a double addressing: hardware and logical addressing. The hardware addressing relates to the memory address range where the hardware object is mapped (see Section 3.2.4).

The logical addressing is over the data channel of a standard protocol. Thus, our logical addressing approach can be taken in any communication bus, even into a streaming bus channel, such as AXI-Stream bus. This bus does not contain a hardware addressing but our approach provides a logical one in order to route the messages to their correct destination. To carry out the logical addressing and hardware object communication, we propose a communication protocol. As we mentioned in the previous section the message is usually composed by a header of a 64-bit word, but it can be composed by a 32-bit word in special cases. It is divided into fields and, sometimes, followed by a payload which contains the serialised data. Figure 3.8 shows an overview structure of our proposed header among an example of function float scale(float sum).

![Figure 3.8: Overview of communication mechanism](image)

**nodeID** This field identifies a node. A node contains one or more computational modules. For instance a **Graphics Processor Unit (GPU)** is a node. However, a hybrid
FPGA has two types of nodes that match with an embedded processor that is contained by the hybrid FPGA (processing system domain) and its logic area (programmable logic domain). Even a programmable logic domain can be divided into several domains, if the designer wants. For instance, each clock region could be treated as a different domain or we can group each hardware object according to the functionality that it offers. For instance, objects that work with video frames are grouped into one group, while the objects which work with the video sound are grouped into another group.

**objID** This field identifies a hardware object into a node. The facade module is the main responsible to decide if a message must be attended to or not. To perform this task, the facade compares the nodeID and objID field with the hardware object configuration. Indeed, there are four addressing options in our communication protocol. These can be reached according to the values of nodeID and objID identifiers. These addressing options depend on the scope, namely which entities should attend to the sent message.

- **Direct addressing:** this option denotes an addressing for a unique node and a unique object. Therefore, a message is attended by only one object, and both fields contain a specific identifier. Figure 3.9 shows an example of this type of addressing. The message is the same as Figure 3.8 illustrates. We focus on the three identifiers that contain a message: nodeID, objID and methodID. In this example we have two objects deployed in different domains, so each object has a different nodeID value. For instance, the left object is deployed in the logic part into a clock region of an FPGA, while the right one is running on the same part but into a different clock region. Their identifiers are 1 and 2 respectively. The message’s identifiers refer the left hardware object, because the nodeID value is 1 and matches with the left domain’s object. In addition, this domain contains an object whose identifier is 1. Moreover, this object contains a method whose identifier matches with the message’s methodID.

**Figure 3.9:** Direct addressing
• **Object broadcast address:** this option enables to send a message to all objects of a specific node. Figure 3.10 illustrates an example of this configuration. In this case, we include a new hardware object inside the left domain, replicating the same functionality. Its objID is 2 while the nodeID is 1, the same as the other object in the same domain. The message used in this configuration, shown in Figure 3.10, whose nodeID value corresponds with the left side domain, whereas the objID value is the broadcast address (0xFF). From a hardware design point of view, when one sends a message he needs to denote a hardware address that identifies a hardware component. In this case, the hardware objects are connected to an AXI bus on shared access mode. This mode allows to make N-to-M interconnections [Xil11]. This kind of configuration allows to send the same message to all objects of a group.

![Figure 3.10: Object broadcast address](image)

• **Node broadcast address:** this option allows to send a message to some nodes, but only those nodes that contain an object whose identifier matches with the message’s objID field that attends to the message. Figure 3.11 shows a similar example of the direct addressing configuration, the only change is the field nodeID, whose value now is 0xFF. This value indicates a broadcast message. Each hardware object whose objID is 1, must attend to the message, independently of the nodeID value. Therefore, we are able to send the same message to different groups of objects.
3.2. Hardware Objects

- **Node and Object broadcast address**: this option allows to send a message to all hardware objects of several nodes. Figure 3.12 shows an example in which the nodeID field and objID field of a message match with the broadcast address (0xFF). This kind of message is a full broadcast message, to which all hardware objects must attend.

Summarising, hardware objects can be grouped according to their functionality, thus we can send a message to a group of hardware objects without forwarding the same message all the time. This approach reduces the message traffic and improves the performance of a design. For instance, a design with some hardware replicated accelerators can be configured by only one message. Our proposal only uses the first addressing option, due to the unit testing principles, and each function must be accessed individually.
On the other hand, if the real devices are connected via Ethernet, they can be organised in subnet addressing. Thus, we have another way to group the real devices, and hence the hardware objects.

**methodID** This field identifies each method implemented inside a hardware object, namely object functionality. This field does not contain any broadcast address because it would suppose duplicate functionality inside the same object. Each method must perform a specific and individual task.

**flags** This field gives information about the message type and its content. Figure 3.13 illustrates the meaning of each bit. The information extraction is extracted through applying some masks.

**callback** This field denotes the logical response address that a hardware object must send its results to. Nowadays, this field is not used, because the asynchronous mode is not implemented yet. In addition, it is not necessary to carry out the hardware verification process.
3.2. Hardware Objects

**size** This field denotes the payload size, measuring it in 32-bit words. For instance, the *size* field of function `float scale(float sum)` is 0x1, because the payload is a `float` whose size is 32 bits, namely one 32-bit word. While the *size* field of the `float loop1(float histIN[16], float histAUX[16])` function is 0x10, because the payload is a `float` array, whose depth is 16. The `loop1` function takes the first argument as input, whereas the second one is an output channel to store the original pixels (see Listing 3.4).

This field is very useful when we need to discard a message. For instance, if the *facade* module does not understand a message, it must be rejected. This task is done after the header analysis process, when the *facade* module knows how many 32-bit words contain the payload. Thus the *facade* module manages the rd signal, it is active-high until reads all 32-bit words related to the message.

**Types of messages**

Our approach uses the request/reply model. When an initiator sends a request message to a target, the target always replies with a message that informs about the request message. Thus, the initiator is able to know the communication status. We define five different types of messages. These messages are according to this request/reply model. The following lines explain the five types of messages. These types can be divided in two groups: requests and replies.

**Request without payload:** this type of message is created by an initiator and is used for getting a reply from the target that receives the message. For example, the `int foo()` function does not contain payload and it expects an integer as return value. Figure 3.14 illustrates an example of this group. The *flags* field is set to 0x00, the *size* field is set to 0x0000 as well, because the message has not got payload.

![Figure 3.14: Overview of request message without payload](image-url)
Request with payload: this type of message is created by an initiator and is for sending some information to a target. According to the function implemented you could wait for a reply, if the function returns something. Otherwise, you are not waiting for anything. For instance, the `void foo(short row, short cols)` void function can occur in the configuration stage of a hardware object. Figure 3.15 illustrates an example of this group using the above function. The `flags` field is set to 0x04, denoting that the message contains payload. The `size` field is set to 0x0001, because the size of two `short` types is a 32-bit word (see data serialisation in Section 3.2.3). Those functions which contain return values, belong to this group as well.

![Figure 3.15: Overview of request message with payload](image)

Reply without payload: this type of message is created by a target and is used for notifying that the target has received a correct message. Therefore, the target sends an acknowledgement to the initiator, thus it is the reply part that complements the messages of the request without payload group. It is usually used in void functions because the initiator does not have any feedback about the communication process. Figure 3.16 illustrates an example of this group. The `flags` field is set to 0x01, denoting that the message is a reply message and it does not have payload. The header of the reply message can be smaller. In this case, the header is a 32-bit word, because it has not got `callback` nor `payload`, therefore the `callback` and `size` fields are not needed. This reduces the header time processing.

![Figure 3.16: Overview of reply message without payload](image)
**Reply error:** this type of message is too similar to the *reply message without payload*. The main difference is the error bit, which is active-high. This message is used when the target is not able to understand a request sent by an initiator, thus the target rejects the message and communicates this. Figure 3.17 illustrates a reply error message.

![Figure 3.17: Overview of reply error message](image)

**Reply with payload:** this type of message is created by a target and returns the values of a function. The function `float scale(float sum)` from our case study is a good example. This function returns a single-precision floating-point value. Therefore the reply message must set the flags field, denoting that it is a reply message and it has payload (0x03). This kind of messages does not need a callback field, however it needs the size field to indicate the payload size. Thereby, the header’s size is a 64-bit word. Figure 3.18 illustrates a reply message with payload.

![Figure 3.18: Overview of reply message with payload](image)

### 3.2.3 Data Serialisation

When we were describing the process to build the double wrapper and get the hardware encapsulation, we introduced a new term: *hardware castings*. A cast is a special operation that forces one data type to be converted into another. In our
proposa there are two types of explicit castings: from byte sequence to C user-defined type and from C user-defined type to byte sequence, used for the reading and writing stages respectively. This process can be denoted as data serialisation and it has only been applied to the message’s payload.

As we described above our entry point is based on FIFO signals, whose data width is a 32-bit word for both channels: input and output. Therefore, we must be able to convert from these 32-bit words to a C type and vice versa, besides, HLS only supports C types castings, so the minimum size is a byte or in other words a char type [Xil12a]. The 32-bit words can be denoted as a sequence of bytes, although we read a 32-bit word, we are really reading 4 bytes, so we treat the data as a sequence of bytes. This fact entails the problem of the misaligned data. The data that receive a hardware object must be aligned to 32-bit words. When the data does not comply with this restriction, we must add a padding at the end.

For example, the function `short foo(int a, short b)` has two arguments whose size is 6 bytes, 4 bytes of int argument and 2 bytes of short argument. The argument size is not aligned to a 32-bit word, it misses 2 bytes to fit the restriction. These 2 bytes are handled as padding, thus the payload is composed by the arguments and the padding (see Figure 3.19).

![Figure 3.19: Example of argument alignment](image)

In addition, the data serialisation entails the endianness problem that must be defined. Our approach uses a big-endian format. This implies a new challenge for our proposal due to the reinterpretation done by HLS tools in a x86 processor. For instance, when we assign a char pointer to an array of integers, HLS tools really use a union for the casting process. Unions allow one portion of memory to be accessed as different data types, thus its members concern the same physical space in the memory and its management is according to the processor endianness. The output RTL generated by the HLS tools depends on the endianness processors which are run. We run the HLS tool on a little-endian processor, which implies that unions will be reinterpreted as little-endian. Therefore, the payload’s message must be serialised following big-endian format, the HLS tool reverts the endianness due to the way that it generates the RTL code in a little-endian processor. Figure 3.20 shows an example of this reinterpretation.
3.2. Hardware Objects

This challenge implies a new restriction: arguments whose type size are bigger, are placed first. For example, for function signature `short foo(short a, int b)`, the first argument in the payload will be \(b\) and then \(a\). If \(a\) is an array of three components (`short foo(short a[3], int b)`), the serialisation position will be the same, because \(b\) is an integer whose type size is greater than \(a\) (see Figure 3.21).

In addition, the `c2hwobject` tool is not able to determine the direction (input or output) of special arguments such as arrays. It must be denoted by a `pragma`. For example, in the signature `int foo(int a[N], int b)`, we cannot determine the direction of the \(a\) argument without the function body. Thus, all arguments passed by reference must denote their direction explicitly, otherwise the tool will interpret them as inputs. Listing 3.11 shows a pragma example for this function, where the \(a\) argument is defined as input. In the second function, we use another pragma to denote that \(b\) parameter is an output, however the \(a\) parameter is not defined explicitly thus it takes as input.

```
Listing 3.11: Pragma example
1  #pragma DIRECTION func=foo param=a dir=INPUT
2  int foo(int a[N], int b);
3  #pragma DIRECTION func=foo2 param=b dir=OUTPUT
4  void foo2(int a[N], int b[N]);
```

Taking in account the rules explained above, we illustrate the two ways to serialise the data and how we can translate it from a sequence of bytes to C types and vice versa.
From byte sequence to C user-defined type

This type of serialisation takes place in the reading stage of our wrapping proposal. After storing the payload into an internal variable (\texttt{words32} in our case), we must define a pointer that points to this internal variable (see Listing 3.12).

**Listing 3.12: Template of hardware casting process**

```c
int words32[...];
...
unsigned <ptrType> *ptr = (unsigned <ptrType>*)words32;
args_<funcName>..<attrib> = to<Type>(*ptr++);
```

Indeed, the pointer type must be calculated to improve the hardware casting process. To calculate the pointer’s type, we use the \textit{Greatest Common Divisor} (GCD) between size of input function arguments and input channel size (32-bit word or integer size). For instance, the GCD of the function \texttt{short foo(int a, short b)} is 2 bytes, namely a \texttt{short} pointer, because \texttt{foo} arguments sizes are 4 bytes and 2 bytes for \texttt{a} and \texttt{b} arguments respectively, and the width of the input channel is always 4 bytes (see Listing 3.13).

**Listing 3.13: Hardware casting process of \texttt{foo} function**

```c
int words32[2];
unsigned short *ptr = (unsigned short*)words32;
args_foo.a = toInt(*ptr++, *ptr++);
ptr+=BUS_WIDTH_BYTES/foo_RD_GCD_BYTES-1;
args_foo.b = *ptr--;
```

Listing 3.13 shows the hardware casting process of the \texttt{foo} function example. The first line denotes the internal variable that stores the arguments, in this case we need two 32-bit words to store the \texttt{a} and \texttt{b} arguments. In the third line we declare a short pointer to read the \texttt{word32} variable in chunks of 2 bytes. Thus, to convert the input byte sequence with the short pointer into an integer for the first function parameter, we need to retrieve the value of the short pointer, increase it and retrieve its value again (line 4). For the second parameter, the cast is direct (line 6), but we must take the reinterpretation done by the HLS tool into account. Therefore, before increasing the pointer, we have to set the position of this pointer. Figure 3.22 illustrates how we convert from a sequence of bytes into C types using the above example. The \texttt{word32} array content is shown after the HLS tool reinterpretation.
3.2. Hardware Objects

Note that HLS tools do not allow a direct casting, they throw a reinterpretation error. Thus, we have built a number of synthesizable functions that translate from a C type to another C type to carry out the hardware casting. Table 3.1 describes these functions, denoting their signature and a brief description. Obviously, there is not any function that converts from a C user type to the same C user type, the assignment is directly as shown in the last line of Listing 3.13. These functions use shifter operations to perform the conversion process.

![Diagram](image)

**Figure 3.22:** Example of byte sequence conversion

<table>
<thead>
<tr>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>short toShort(unsigned char c1, unsigned char c2)</td>
<td>Converts from char type to short type</td>
</tr>
<tr>
<td>int toInt(unsigned char c1, unsigned char c2, unsigned char c3, unsigned char c4)</td>
<td>Converts from char type to int type</td>
</tr>
<tr>
<td>int toInt(unsigned short s1, unsigned short s2)</td>
<td>Converts from short type to int type</td>
</tr>
<tr>
<td>long long toLongLong(unsigned char c1, unsigned char c2, unsigned char c3, unsigned char c4, unsigned char c5, unsigned char c6, unsigned char c7, unsigned char c8)</td>
<td>Converts from char type to long long type</td>
</tr>
<tr>
<td>long long toLongLong(unsigned short s1, unsigned short s2, unsigned short s3, unsigned short s4)</td>
<td>Converts from short type to long long type</td>
</tr>
<tr>
<td>long long toLongLong(unsigned int i1, unsigned int i2)</td>
<td>Converts from int type to long long type</td>
</tr>
<tr>
<td>long long toLongLong(unsigned int i1, unsigned int i2)</td>
<td>Converts from int type to long long type</td>
</tr>
</tbody>
</table>

**Table 3.1:** C synthesizable casting functions (from byte sequence to integers C types)
The hardware casting process for floating-point types is more complex. Firstly, the floating-point must be converted into an IEEE-754 format or a hexadecimal format in order to be compatible with the byte sequence [IEE08]. Then, the byte sequence is assigned to a union getting the floating-point value.

Table 3.2 describes the functions for floating-point types, denoting its signature and a short description. In this case, we need a cast function to translate from integer pointer to a float type.

<table>
<thead>
<tr>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>float toFloat(unsigned char c1, unsigned char c2, unsigned char c3, unsigned char c4)</code></td>
<td>Converts from char type to float type</td>
</tr>
<tr>
<td><code>float toFloat(unsigned short s1, unsigned short s2)</code></td>
<td>Converts from short type to float type</td>
</tr>
<tr>
<td><code>float toFloat(unsigned int i1)</code></td>
<td>Converts from int type to float type</td>
</tr>
<tr>
<td><code>double toDouble(unsigned char c1, unsigned char c2, unsigned char c3, unsigned char c4, unsigned char c5, unsigned char c6, unsigned char c7, unsigned char c8)</code></td>
<td>Converts from char type to double type</td>
</tr>
<tr>
<td><code>double toDouble(unsigned short s1, unsigned short s2, unsigned short s3, unsigned short s4)</code></td>
<td>Converts from short type to double type</td>
</tr>
<tr>
<td><code>double toDouble(unsigned int i1, unsigned int i2)</code></td>
<td>Converts from int type to double type</td>
</tr>
</tbody>
</table>

Table 3.2: C synthesizable casting functions (from byte sequence to floating-point C types)

From C user-defined type to byte sequence

On the other hand, after the user function is completed, the function’s results are ready. We must serialise these results, translating the C user-defined types into a byte sequence. This process takes place in the writing stage of our wrapping proposal. Firstly, we declare an internal union for each object function. This union contains three attributes corresponding to the three possible sizes (8-bit, 16-bit or 32-bit words) that can be written in our output port (whose width is 32-bit). Remember the writing size must be the GCD between the output port and output function parameters. The Listing 3.14 illustrates a template of this union.
3.2. Hardware Objects

Listing 3.14: Template of union return

```c
union UNION_RET_<funcName> {
    unsigned int words32[sizeof(ret_<funcName>)/4];
    unsigned short words16[sizeof(ret_<funcName>)/2];
    unsigned char words8[sizeof(ret_<funcName>)];
} byteRet_<funcName>;
```

Then, we create an array for each output parameter of a particular function that is used to store the byte sequence of an output parameter. In the same way as in the reading stage, we use a special function to convert from C user type to byte sequence. These functions are described from C and are synthesizable. After the output parameter conversion process, we write the result into the union described above, concatenating all byte sequences. Finally, we add the padding when it is necessary and write the union value into the output port (see Listing 3.15).

Listing 3.15: Hardware casting serialisation template

```c
index+=...
unsigned <typeParameter> ptr_<retParameterName>;
ptr_<retParameterName> = ret_<funcName>._<retParameterName>;
byteRet_<funcName>._words<GCDSize>[...] = ptr_<retParameterName>;

// Padding
for(short it_padding=0;
    it_padding != sizeof(ret_<funcName>._padding)/<funcName>_WRITE_SIZE_BYTES; it_padding++)
    byteRet_<funcName>._words<GCDSize>[...] = 0;

for(short itReturn=0;
    itReturn != sizeof(byteRet_<funcName>._words32)/BUS_WIDTH_BYTES;
    itReturn++)
dst.write(byteRet_<funcName>._words32[itReturn]);
```

The Listing 3.16 illustrates an example of serializing the output parameters of the function short foo(int a, short b). Note that we must take in account the endianness as well, thus the first line of Listing 3.16 sets the index where the output value must be located. Then we assign the output parameter value to the return union in the position that indicates the index variable (lines 2 to 4). We use the word16 member of return union, because the GCD between a short size and the output width (32-bit word) is 16. In this case, the return message is not aligned to 32-bit words, therefore we must add a padding to fulfill this restriction (lines from 7 to 10). Finally, we use the words32 member of return union to write all values in the output port.
In the above example we do not use any special function to convert from C user-defined type to a sequence of bytes, we provide a list of these functions, which are listed in Table 3.3. For instance, if the above example returns a `short` value and an `int` value the GCD remains 16, but we need the special function `toSequence(int n, unsigned short words[2])` to convert the integer value.

<table>
<thead>
<tr>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void toSequence(short n, unsigned char bytes[2])</code></td>
<td>Converts from <code>short</code> type to sequence of bytes</td>
</tr>
<tr>
<td><code>void toSequence(int n, unsigned char bytes[4])</code></td>
<td>Converts from <code>int</code> type to sequence of bytes</td>
</tr>
<tr>
<td><code>void toSequence(int n, unsigned short words[2])</code></td>
<td>Converts from <code>int</code> type to <code>short</code> sequence</td>
</tr>
<tr>
<td><code>void toSequence(long long n, unsigned char bytes[8])</code></td>
<td>Converts from <code>long long</code> type to sequence of bytes</td>
</tr>
<tr>
<td><code>void toSequence(long long n, unsigned short words[4])</code></td>
<td>Converts from <code>long long</code> type to <code>short</code> sequence</td>
</tr>
<tr>
<td><code>void toSequence(long long n, unsigned int words[2])</code></td>
<td>Converts from <code>long long</code> type to <code>int</code> sequence</td>
</tr>
</tbody>
</table>

Table 3.3: C synthesizable casting functions (from integers C types to byte sequence)
In the same context that the *hardware casting* process from byte sequence to floating-point types. The reverse process is complex as well, and it is done by the same approach, using *unions* to get the IEEE-754 representation. Table 3.4 lists the functions that perform the translating process from floating-point types to byte sequence.

<table>
<thead>
<tr>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void toSequence(float n, unsigned char bytes[4])</code></td>
<td>Converts from <em>float</em> type to byte sequence</td>
</tr>
<tr>
<td><code>void toSequence(float n, unsigned short words[2])</code></td>
<td>Converts from <em>float</em> type to <em>short</em> sequence</td>
</tr>
<tr>
<td><code>void toSequence(float n, unsigned int words[1])</code></td>
<td>Converts from <em>float</em> type to IEEE-754 format (single precision)</td>
</tr>
<tr>
<td><code>void toSequence(double n, unsigned char bytes[8])</code></td>
<td>Converts from <em>double</em> type to byte sequence</td>
</tr>
<tr>
<td><code>void toSequence(double n, unsigned short words[4])</code></td>
<td>Converts from <em>double</em> type to <em>short</em> sequence</td>
</tr>
<tr>
<td><code>void toSequence(double n, unsigned int words[2])</code></td>
<td>Converts from <em>double</em> type to IEEE-754 format (double precision)</td>
</tr>
</tbody>
</table>

*Table 3.4: C synthesizable casting functions (from floating-point C types to byte sequence)*

**Serialisation Overhead**

Unfortunately, the proposed approach adds a small overhead according to the GCD, as we explained before. Adjusting the reading and writing sizes, we considerably reduce the *hardware casting* time. Table 3.5 shows the overhead voiced in cycles of our proposal for `<type> add(<type> a, <type> b)` signature, where `<type>` matches with the first column of the table, while reading and writing sizes are the rest columns, or (in other words) the GCD that complies with the casting size. The best option for the *int* type is 32-bit size, getting a small overhead (3 cycles for reading and 3 cycles for writing). Floating-point types add an extra overhead due to the conversion from the hexadecimal format to the IEEE-754 format and vice versa.
3. Hardware Unit Testing

<table>
<thead>
<tr>
<th>type</th>
<th>8-bit words</th>
<th>16-bit words</th>
<th>32-bit words</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>reading</td>
<td>writing</td>
<td>reading</td>
</tr>
<tr>
<td>char</td>
<td>3</td>
<td>10</td>
<td>-</td>
</tr>
<tr>
<td>short</td>
<td>3</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>int</td>
<td>3</td>
<td>17</td>
<td>3</td>
</tr>
<tr>
<td>long</td>
<td>5</td>
<td>34</td>
<td>5</td>
</tr>
<tr>
<td>float</td>
<td>5</td>
<td>18</td>
<td>5</td>
</tr>
<tr>
<td>double</td>
<td>6</td>
<td>35</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 3.5: Overhead (<type> add(<type> a, <type> b))

From a resources point of view, the hardware library provided to carry out the hardware castings process entails hardware resource utilisation. This resource utilisation (post-synthesis) is shown in Table 3.6. Note that the library’s functions are unique in the hardware design, thus the resources are reused for all functions that use it.

<table>
<thead>
<tr>
<th>C type</th>
<th>LUT</th>
<th>FF</th>
<th>BRAM</th>
<th>DSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>324</td>
<td>208</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>short</td>
<td>398</td>
<td>198</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>int</td>
<td>487</td>
<td>235</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>long</td>
<td>528</td>
<td>328</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>float</td>
<td>701</td>
<td>544</td>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>double</td>
<td>1207</td>
<td>1000</td>
<td>-</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 3.6: Hardware resource utilisation in post-synthesis for hardware casting library

3.2.4 Bus Drivers

Now that the core of the hardware object is explained, our proposal entails one problem due to its entry point. It cannot be connected directly in a hardware design. In most hardware projects, the hardware components are connected through a standard bus, such as AXI, using a bus interconnect component [Xil11]. Therefore, we must define a bridge between the standard bus and our hardware object’s entry point. The standard bus chosen is AXI4 which is the most used bus for hardware design projects [Fos16].

We name this bridge the axi2fifo driver and it is used for attending to third-party messages that are sent via AXI bus. This driver only works with slave components. Therefore, our hardware objects are slave hardware components, but this does not pose a challenge. To treat every hardware object as a slave mode implies that the initia-
tor must ask the target for everything. For instance, the function \texttt{float scale(float sum)} from our case study is translated into two parts. Firstly, the initiator sends the information to exercise the target, and then the initiator sends a message to retrieve the result. Thereby, the initiator always begins the communication. Figure 3.23 illustrates an overview of a \textit{hardware object} in TLM, including its driver and its core.

![Figure 3.23: Overview of hardware object in TLM](image)

From a behavioural point of view, this driver translates messages from an AXI bus to two FIFO interfaces. Firstly, the driver checks the message from the two address channels, if the read or write addresses match with the address interval of the driver, the message will be attended to. In the same cycle, the driver knows which type of message is: reading or writing. If the message is a writing the data channel is forwarded to the FIFO\_IN interface. On the other hand, reading messages implies that the data must be available in the FIFO\_OUT interface.

To check the throughput of our driver’s approach, we have built an experiment using a hybrid FPGA, specifically a \textit{Zedboard} from \textit{Xilinx} and its tools. The initiator part, an executable program running on the ARM processor, sends 1024 words, whose width is 32 bits, to a \textit{hardware object} using different techniques. This object increments the data received and forwards it. Finally, the software part retrieves the result returned by the \textit{hardware object} using the same technique that as in the writing stage and checks its correctness (it is a loopback example). Table 3.7 shows the results depicted in microseconds which are obtained in this experiment using four solutions with different drivers. The three drivers are: our proposal, \textit{Xilinx} drivers generated by its HLS tool and the GPIO drivers from \textit{Xilinx}. The four solutions are Xil\_IO library, memcpy, pointers and Direct Memory Access (DMA).

<table>
<thead>
<tr>
<th>Technique</th>
<th>Xil_io</th>
<th>pointers</th>
<th>memcpy</th>
<th>DMA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>read</td>
<td>write</td>
<td>read</td>
<td>write</td>
</tr>
<tr>
<td>Proposal</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Our</td>
<td>147.77</td>
<td>158.76</td>
<td>147.8</td>
<td>154.34</td>
</tr>
<tr>
<td></td>
<td>72.07</td>
<td>77.51</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>47.85</td>
<td>63.72</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Xilinx (GPIO)</td>
<td>230.91</td>
<td>256.56</td>
<td>230.0</td>
<td>256.0</td>
</tr>
<tr>
<td></td>
<td>139.05</td>
<td>153.97</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>56.87</td>
<td>63.72</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Xilinx (HLS)</td>
<td>230.94</td>
<td>192.40</td>
<td>184.68</td>
<td>225.0</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\textbf{Table 3.7:} Throughput time depicted in microseconds to write and read 1024 words
We can observe in Table 3.7 that our proposal is the best option to send data between components, independently of the operation to write or read data, and the method chosen to perform this operation.

### 3.3 Hardware Verification Platform

After hardware object definition, we describe the hardware testing environment in which the DUT is verified. This hardware platform brings a high accuracy, because the DUT is running on a real device after the implementation step.

Most projects based on FPGA contain one or more embedded processors. In addition, FPGA vendors have focused on building new FPGAs with embedded processors (such as the Zynq family from Xilinx or Cyclone from Altera). These kinds of FPGAs are known as hybrid FPGAs [Fos16]. The final verification environment is based on hybrid FPGAs, specifically a ZedBoard from Xilinx. This kind of board is divided into two main parts, as shown in Figure 3.24: a Processing System (embedded processors) and a Programmable Logic (FPGA logic). The first one contains an ARM processor which runs the software, whereas the Programmable Logic part contains several cores that we explain below, facilitating verification tasks. Both parts are connected by an AXI bus, thus information can be exchanged from one domain to the other.

![Figure 3.24: Overview of hardware verification environment](image)

In order to communicate the elements of both parts we propose a solution based on RMI, like Ph.D. [Bar08] proposes. We start from the premise that a SoC is a distributed system, in which all components are modelled as objects. We introduce a new concept that is fully compatible with our hardware objects, whose virtual
representation matches with the RMI philosophy.

Nowadays, as our hardware verification platform shows, communication tasks in a SoC are driven by a dedicated bus such as AXI, where the distributed object communication model can take place. One objective of a remote object is enabling the mechanisms to invoke remotely and transparently the object’s methods. Remote means an invocation between two entities connected to a transport network, such as an AXI bus, while transparent means that this communication can take place between objects of different domains, for instance, an object implemented in software and another implemented in hardware, and both are not able to distinguish which are on the other side.

![Figure 3.25: Overview of RMI](image)

The proposed remote object model uses the same artefacts as RMI, which is very popular in software object oriented distributed systems, such as Java RMI [Mic03] or ZeroC [Zer17], to disaggregate the behaviour from communication tasks. Any remote method invocation takes place between the communication part of an object, in our case the admin part of our hardware objects, and its virtual representation in order to make a delusion that both objects are directly connected. This delusion is achieved as Figure 3.25 shows.

1. The object \textit{objA} calls upon a method of \textit{virtual representation} of object \textit{objB}, which contains the same methods that object \textit{objB} has implemented.

2. The \textit{virtual representation} forwards the request through a communication channel, following a protocol and a message format previously established. In our case, the protocol matches with AXI specifications while the message format matches with our communication mechanism described in Section 3.2.2.

3. On the other side, \textit{objB} receives the message and it will reinterpret the message according to the format and communication protocol.

4. The \textit{objB} calls upon the function that carries out the operation. If the invocation
contains any return value, this should be replied to the object objA. Otherwise the process finishes here.

5 When the return values are ready, the object objB sends them to the object objA using the same communication mechanism.

6 Finally, the reply message will arrive to the virtual representation of object objB located in object objA. It manages the reply message and returns the reply values to the initial invocation.

In our approach, we do not fulfil all these steps, because our proposal uses initiator-target invocations, in which the object objA always starts the communication. In order to get return values, we translate the functions with return values into writings and readings over an AXI bus. For instance, the function int foo(int i) implies writing and reading AXI messages that Listing 3.17 illustrates. This example uses the xio library from Xilinx that provides some functions to read and write in a specific hardware address. The first three lines correspond to the request message following our communication approach, where the last one is the foo’s argument. The difference is in the reply message and the active object that must ask the passive object about it. This is done by three readings (lines from 8 to 10). The first two readings match with the header of our communication mechanism, while the last one is the return value. Listing 3.17 is a virtual function related to the foo function. These kinds of functions make up a virtual representation of an object.

### Listing 3.17: Writing and reading AXI messages

```c
int foo(int i)
{
    Xil_Out32(HW_ADDR, 0x00010204);
    Xil_Out32(HW_ADDR, 0x00000001);
    Xil_Out32(HW_ADDR, i);

    header_1 = Xil_In32(HW_ADDR);
    header_2 = Xil_In32(HW_ADDR);
    return_value = Xil_In32(HW_ADDR);

    return return_value;
}
```
3.3.1 Processing System Part

The Processing System part is related to the software domain, which is used for exercising the DUT and managing the verification process using unit tests. Although we do not explain how the original unit tests exercise the DUT, we will consider the unit test as part of an executable file, like testing framework libraries. Even though Figure 3.24 shows some software modules in its processing system part, we now consider them as an executable file. This will be explained further in section 3.4.3.

Generally, our hardware verification platform, in its software part, runs a Linaro Ubuntu distribution in its ARM from a SD card. The source files are compiled by a cross-compiler, obtaining a correct executable that can be run on the ARM architecture.

Summarising, now we consider the software that runs in the processing system as a black entity which manages the verification process. It contains a virtual representation of the DUT which allows to stimulate it. The executable of this black entity is obtained by a cross-compiler.

3.3.2 Programmable Logic Part

The Programmable Logic part contains several hardware components to carry out the verification process in a real device. These components are connected to an AXI bus, which is the same as to what the processing system part is connected. However, when a new version of our hardware design (DUT) is ready, we must synthesise it and all hardware components of our hardware testing environment as well.

The synthesis process is a hard task that requires a high computing power and costs consumption-time. In order to improve and minimise the synthesis process, we include an interesting feature available in some FPGAs, called DPR. This feature brings up design benefits. One of these benefits motivated by the DPR, is the fact that the designer can dynamically insert new functionality without redesigning the whole design or moving to a bigger device. Therefore, the DPR feature makes FPGA devices more flexible since it permits isolating those regions of the FPGA in which the modifications have to be introduced from those that remain unaltered. Considering the hardware verification platform proposed above, in which the DUT might be adapted to new versions, this dissertation takes advantage of the dynamic reconfigurability with the goal of introducing those hardware modifications at run-time without interrupting the execution of the rest of the system. Before going into detail in the physical implementation of the proposed system, the following paragraphs introduce several designing rules and considerations that have to be respected in order to exploit the dynamic reconfigurability successfully. In addition, these ideas will provide the
reader with a better understanding of the complexity of the dynamic reconfigurability in FPGAs [CCL15].

From a design point of view, a dynamically reconfigurable system has to split its design into two regions in the FPGA. On the one hand, those parts of the system that never change have to be allocated in a static region, whereas the elements that are sensible to be modified during the execution have to be placed in a dynamic region. It is possible to create one or more dynamic regions into the same FPGA, depending on the system or the architectural needs. At the same time, the dynamic region is subdivided into partitions, known as dynamic areas. Within each of these dynamic areas a hardware component, with a specific functionality or a fake functionality, is deployed. The DPR extends the inherent flexibility of the FPGA, since it only modifies specific elements in the dynamic region at run-time, without stopping the rest of the modules of the device. Therefore, this feature fits perfectly with the demands of adaptable and scalable solutions [CCL15].

DPR brings some benefits, and one of the benefits that fits perfectly in this Ph.D. dissertation, is the possibility to adapt the FPGA to different scenarios. This is done, by modifying the functionality or the performance of some tasks running on it. In addition, the DPR reduces costs and area, because it enables smaller designs by time-multiplexing portions of the available hardware resources. An example of this is presented in [MMT08], in which a single hardware platform can support different waveforms by using one or another at run-time, according to the user/application requirements. The DPR feature provides additional advantages, such as reducing the bitstream storage needs, the synthesis time or accelerating the computation [D.12][C.05] [CCL15].

In summary, the DPR process increases the power of the whole system by reusing elements that have already been implemented without redesigning the solution from scratch, after the system has been deployed. This characteristic is very convenient for all those applications that require real-time adaptability [VAD12]. The DPR is a useful characteristic in contexts in which the system cannot be stopped, but it needs to be adapted. Thus, the addition of new hardware components into the system requires that this one is instantiated in a partial reconfigurable region of the FPGA. The fact that a dynamically reconfigurable FPGA might keep running, even when a component failure has been detected, strengthens the robustness and reliability of pre-emptive systems. Therefore, a system breakdown can be avoided by replacing the damaged component by a new one at run-time, improving in this way the system fault-tolerance [DRV13] [CCL15].

Observing our hardware verification platform in Figure 3.24, we show an only one dynamic area which contains a DUT to verify its correctness. The interface of this dynamic area matches with our entry point defined in the previous section, a FIFO signals interface. In addition, we extract the bus driver module from the DUT and two
3.3. Hardware Verification Platform

small buffers, thus the DUT only contains its functionality, which really is the part that will change. The dynamic area contains 6120 LUTs, 12240 FlipFlops, 32 DSPs and 16 BRAMs, whose bitstream size is about 464 KB and is placed between the slices X14Y3 and X47Y47.

The DUT bus driver has been extracted and included into a hardware component called bridge. This bridge translates writing AXI messages into the input FIFO (FIFO_IN) and replies reading AXI messages with the data that FIFO_OUT contains. Thus, we reduce the elements that might be synthesised for a new DUT release. The driver is the same, independently of the DUT that it is running, even every hardware object has an axi2fifo driver with the same functionality. The difference lies in the interval memory address where is mapped.

Another hardware object instantiated into the hardware verification platform is the Test Manager object. The main objective of this object is to manage the verification process in the hardware domain. Although, it is explained in section 4.1, we need to know that this object resets the dynamic area before a test case is executed. Thus we ensure the Independently principle that unit tests may fulfil.

At this point, our hardware verification platform is defined, and it can be used for verifying hardware objects. Although it is not the final verification version, it is enough to carry out the verification process in a real hardware. This platform will be replaced by a more complex one, but it will keep the core of the current platform. In addition, the new platform provides a transparent testing service (see Chapter 8).

Automatic generation of configuration files

Building a reconfigurable project is a long and complex task, mainly because the Xilinx vendor tool works at very low level of abstraction. This Ph.D. dissertation provides an alternative flow, based on a scripting language, TCL, as a front-end of the vendor tools, but with the idea of rising the task abstraction level that engineers must perform. In addition we automate the configuration file generation. Our starting point is a Xilinx block design description (block design file) of the project hardware design.

To carry out this approach, we must define a project description file which depicts the dynamic hardware project. This file contains the information about the dynamic area and its location, including the hardware resources inside it. In our hardware verification environment, we only provide a dynamic area. This file contains the files which compose the dynamic area. To minimise the consumption-time in synthesis process tasks, we should first define a reference configuration, since all bitstream will be generated. In order to get the partial bitstream associated to a DUT, we need to add
its source files and a new configuration, thus the scripts use the reference to get the partial bitstream, reducing consumption-power and consumption-time (see Appendix B). These scripts are able to generate the (partial) bitstream files from command line without human interaction.

Figure 3.26 shows a comparison of consumption-memory during the DUT synthesis process between our scripts and the Xilinx tools. It shows the memory spectrum that takes place in the system which synthesize the DUT from when the tool starts. Observing both memory spectra, it is clear that our scripts reduces the consumption-memory and releases the memory before the Xilinx tool does. The experiment had been synthesized in a GNU/Linux environment onto a CPU i7-3770 @ 3.4GHz and 16GB of RAM. Our scripts takes about 5 minutes and 46 seconds, while Vivado tool takes 12 minutes and 10 seconds.

![Figure 3.26: Comparison between our scripts and Vivado tool (consumption-memory)](image)

The CPU consumption-power during the DUT synthesis process is shown in Figure 3.27. Again, our approach shows better results, speeding up this process and thereby releasing the CPU cores.

### 3.4 Integration with the typical design flow

Typical hardware design flow involves some stages which grow with the HLS incorporation. Figure 3.28 shows the three steps of the typical FPGA design flow including HLS. This flow involves human interaction to verify FPGA-based designs [PB04] [Jan03] [Tha09].
3.4. Integration with the typical design flow

![Figure 3.27: Comparison between our scripts and Vivado tool (consumption-power)](image)

![Figure 3.28: Overview of the typical FPGA design flow](image)

**Design Intent** The flow starts with the design intent, the designer creates, using natural language, project specifications. The result is a document which describes a hardware design.

**Captured Design** Project specifications are transcribed following one of the next options:

- Using a formal HDL, such as VHDL or Verilog.
- Using a high-level programming language, such as C. Thereafter, the code is translated into RTL description by an FPGA-vendor tool.

Independently of the method performed, we must verify the generated code. In addition, the second method involves a new testing step, we must check the behaviour of the described high-level code.

**Implemented Design** The RTL description is translated to a so-called netlist (synthesis) and mapped onto a particular device structure (place & route). The result
is a programming file called bitstream which is used to configure an FPGA. This process is performed by FPGA-vendor tools.

**Bitstream generation**

After the design intent is transcribed into a HDL or a high-level programming language in order to generate the captured design, engineers must generate the configuration file (bitstream) of their designs using FPGA-vendor tools. This process is divided into two steps: synthesis and place & route.

**Synthesis** this stage involves the conversion of an HDL description to a so-called netlist. This process is performed by FPGA vendor tools. At this stage, we must perform another verification using the synthesis result. The netlist allows a better verification accuracy with timing annotations.

**Place and Route** also known as implementation. The netlist is mapped onto a particular device’s internal structure. The result is a programming file called bitstream. Finally, the functionality is deployed onto an FPGA and we must test it to ensure that our hardware functionality works correctly.

This typical FPGA design flow requires manual efforts by trained engineers. TDD methodology can be applied in a hardware design flow reducing the complexity of it, but TDD involves a number of synthesis tasks; one per each unit test described. Although, we speed up the synthesis process, TDD requires time due to its flow which makes it unfeasible.

The main problem of this typical flow is based on verifying the generated code of each stage. In other words, each step needs to be verified. It induces to rewrite tests in accordance with each flow stage. Our approach introduces a new feature in this typical flow: reusing tests from the first stage. From the technical point of view, we keep the same test suite at different verification stages, replacing the functions that the unit test calls upon as Figure 3.29 shows. In a pure software domain, the unit test uses the original source, while the other two levels use bridge functions to write stimuli in the correct channel, using the proposed communication mechanism. This solution is based on the Remote Method Invocation (RMI) philosophy, which allows components on different nodes to interact in a distributed system. The main advantage of RMI is that it provides a neat separation between functionality and communication. Any method invocation in RMI must take place between certain adaptors, from the unit test’s point of view all bridge functions make a virtual component that is able to communicate to the real component.

Taking in account that our functionality is modelled as a hardware object, we must be able to verify this object following the typical verification flow. Remember that our hardware object is described from a high-level programming language, thus we
can use unit tests to verify its behaviour. At this point, we define the tests that verify the correctness of our functionality, and they should not change along the verification flow.

Summarising, we impose ourselves a new challenge: **reusing the test suite**. This challenge is the core of typical flow integration, thus the bitstream generation is preserved, but its verification is modified. The following sections explain how we can reuse the verification process using unit tests.

### 3.4.1 High-level modelling verification

Firstly, and before the use of any HLS tool, we must verify that our code described from a high-level programming language is correct, especially that the product intent, implementation and functional specification converge with a high percentage. To do this, we can use unit tests as we described at the beginning of this chapter (see section 3.1). Therefore, we check the correctness of our functionality using unit tests in a pure software domain.

For example, in our case study we can find three methods that make up the $\ell^2$-norm algorithm. These methods must be verified independently in a pure software domain. This process is led by the Unity testing framework, which is written in C programming language like our three methods. Listing 3.18 depicts the three unit tests in order to verify the correctness of our $\ell^2$-norm algorithm implementation. We verify the correctness of each method through `asserts` functions, comparing the method’s results obtained from particular input parameters with golden vectors associated to these input parameters. For instance, the `scale` function returns the floating-point value **0.217391** for the floating-point input **9.0**.
3. Hardware Unit Testing

Listing 3.18: Unit test cases for $\ell$-norm algorithm

```c
void
test_scale()
{
    TEST_ASSERT_EQUAL_FLOAT(0.217391, scale(9.0));
}

void
test_sum_hist_pow()
{
    float input[HIST_SIZE];
    for(int i=0; i != HIST_SIZE; i++)
        input[i] = i;
    TEST_ASSERT_EQUAL_FLOAT(1240.0, sum_hist_pow(input));
}

void
test_mult_hist_scale()
{
    float ref[HIST_SIZE], input[HIST_SIZE], out[HIST_SIZE];
    for(int i=0; i!= HIST_SIZE; i++)
        input[i] = i;
    ref[i] = input[i]*0.1;
    mult_hist_scale(input, 0.1, out);
    for(int i = 0; i != HIST_SIZE; i++)
        TEST_ASSERT_EQUAL_FLOAT(ref[i], out[i]);
}
```

Figure 3.30 shows the return report of Unity testing framework after running the test cases defined above in a pure software domain. We can observe that all test cases are successfully passed.

Figure 3.30: Report of Unity framework in a pure software domain
Modifying the test for `scale` method to induce a bug, we can observe in Figure 3.31 that the `scale test case` fails, because the testing framework expects a different floating-point value and it is not within delta value.

![Figure 3.31: Report of Unity framework in a pure software domain (inducing a bug)](image)

### 3.4.2 Co-simulation: RTL and Gate-level verification

After the pure software verification stage, we must apply our proposal to ensure the individual method accessing and fulfilling HLS restrictions and unit test principles (see section 3.1). Thus, we must wrapper the functionality that makes up our `hardware object`. However, we cannot apply the above unit test cases directly, now we have to send a message that the `hardware object` understands, following the communication mechanism explained in section 3.2.2 and serialisation process explained in section 3.2.3.

![Figure 3.32: Unit tests and hardware object into co-simulation environment](image)

To carry out the verification process we do not directly call upon the method that will be verified, we must call upon a virtual representation of this method (see Figure 3.32). This virtual representation is automatically obtained from functionality signatures by the `c2hwobject` tool, and it transforms the original software function call into an object message format that is forwarded to the `hardware object` and served by it. In addition, `c2hwobject` encapsulates the functionality according to the proposal explained in previous sections. Finally, we use the verification environment provided by FPGA vendors, they denote this verification environment as a co-simulation environment. This co-simulation environment is enabled automatically, FPGA vendor tools...
create the test bench wrappers and transactors that the designers can leverage the original test framework with to verify the correctness of the RTL output [CLNN10].

When we run the c2hwobject tool over a C functionality, we get a hardware object and its virtual representation. Thus, the C code has been modified; now it includes our encapsulation approach. Therefore, test cases must call upon the functions of this virtual representation instead of the original one. The only change that we have to make is to include the file at the beginning of unit tests file. The change is shown in Listing 3.19.

Listing 3.19: New header file of \( p \)-norm algorithm

```c
#include <unity.h>
#ifdef HW_COSIM
    #include <vector_norm_cosim.h>
#else
    #include <vector_norm.h>
#endif
```

On the other hand, the virtual representation tasks are more complex; we must fulfil the communication mechanism that our proposal imposes. Listing 3.20 depicts the virtual representation of the \texttt{float scale(float sum)} function in a co-simulation environment. Firstly, we set forth all internal variables that we are going to use, the two most important variables are \texttt{din} and \texttt{dout}, which send and receive the information respectively from our entry point. Then, we write the input stimuli into the \texttt{din} variable, according to the communication protocol depicted in section 3.2.2; we include the header followed by the payload (lines 9 to 11 of Listing 3.20). The values are in accordance with our case study (see Figure 3.8). After the stimuli writing, we must call to the \texttt{topTesting} function, which is our hardware object entry point. When the result is ready, we read it from the \texttt{dout} variable, which is the output channel of our entry point. Note that our communication approach contains a reply header that must read: «below are the return values of \texttt{scale}». Although, it is not commented, the floating-point values must be converted to IEEE-754 format and vice versa, lines 11 and 17 make this translation process through two special functions. For double precision types, another function is used: \texttt{toIEEE754_double_precision} and \texttt{toDouble}.
3.4. Integration with the typical design flow

Listing 3.20: Virtual function of `scale` (co-simulation)

```c
float scale(float sum) {
    int head1, head2;
    float _ret;

    hls::stream< unsigned int > din;
    hls::stream< unsigned int > dout;

    din.write(0x00010204);
    din.write(0x00000001);
    din.write(toIEEE754_single_precision(sum));

    topTesting(din, dout);

    head1 = dout.read();
    head2 = dout.read();
    _ret = toFloat(dout.read());
    return _ret;
}
```

Figure 3.33 shows the report obtained from the Vivado HLS tool after applying our hardware object approach, and using the Unity testing framework. Remember that this verification is done in a pure software domain as well, but using the FPGA vendor tool. In order to include the correct header, we must define a symbol (-D HW_COSIM) before running the verification.

![Report of Vivado HLS tool using Unity framework after applying our proposal](image.png)

Figure 3.33: Report of Vivado HLS tool using Unity framework after applying our proposal

After our approach is verified in a pure software domain, we must check it into a co-simulation environment. This environment is provided by the FPGA vendor, whose HLS tools enables this kind of verification without a big effort. Figure 3.34 shows the report obtained from Vivado HLS again, but in this case we use a co-simulation environment.
3. Hardware Unit Testing

Figure 3.34: Report of Vivado HLS tool using a co-simulation environment

Vivado HLS reports provide interesting information about how many hardware resources will be used and a summary related to the latency and timing of each hardware module. It is very useful for knowing the quality of our implementation. Figure 3.35 depicts our three module reports, `sum_hist_pow`, `scale` and `mult_hist_scale` respectively.

3.4.3 Verification in a real device

After the implement stage, we obtain the bitstream that allows us to configure the FPGA. This bitstream contains the real implementation according to the target device which our hardware object runs. Therefore, we must verify it in this environment. Remember that we proposed a hardware verification platform based on hybrid FPGA devices. This kind of FPGA is divided into two parts: the programmable logic part which is explained in section 3.3.2 and the processing system part which matches with the device’s software domain.

In this case, we run all test cases on the device’s software domain, in our study case into an ARM processor. A cross-compiler is used to get the correct executable. As in the previous case, we need a virtual representation of the hardware object, but
3.4. Integration with the typical design flow

Figure 3.35: Reports of `sum_hist_pow`, `scale` and `mult_hist_scale` modules

In this case this module translates original functions to AXI messages, remember that both FPGA domains are connected via an AXI bus (see Figure 3.24). The virtual representation is obtained by the `c2hwobject` tool as well as shown in Figure 3.36.

Figure 3.36: Unit tests and hardware object into hybrid verification environment

Listing 3.21 depicts the virtual representation of the `float scale(float sum)` function in a hybrid environment. Firstly, we get a file descriptor of the `/dev/mem` device that enables the accessing to a hardware address memory (lines 9-13). Then we create a new virtual mapping in the virtual address space, starting with the hardware address in which the hardware object is mapped (0x42000000) using the file descriptor obtained from the previous step. The `mmap` function returns a pointer to the mapped area, which is used for hardware accessing.
Listing 3.21: Virtual representation of \textit{scale} function (hybrid)

```c
float
scale(float sum)
{
  void *ptr;
  float _ret;
  unsigned int head1, head2;
  unsigned page_size=sysconf(_SC_PAGESIZE);

  int fd = open("/dev/mem", O_RDWR);
  if(fd < 1) return;

  ptr = mmap(NULL, page_size, PROT_READ|PROT_WRITE,
             MAP_SHARED, fd, 0x42000000);

  *((unsigned *)ptr) = 0x00010204;
  *((unsigned *)ptr) = 0x00000001;
  *((unsigned *)ptr) = toIEEE754_single_precision(sum);

  head1 = *((unsigned *)ptr);
  head2 = *((unsigned *)ptr);
  _ret = toFloat (*((unsigned *)ptr));

  munmap(ptr, page_size);

  return _ret;
}
```

Lines 19-21 depict the input stimuli according to the communication protocol; the first two lines from the header and the last one is the parameter of \textit{scale} function. The \textit{scale} parameter is a floating-point type that is converted to IEEE-754 format with the same function that is used in the co-simulation environment. These writings are translated into AXI messages whose hardware address is \texttt{0x42000000}. In this case, we do not call upon any function, because we directly send messages through the AXI bus. Therefore, the next step is reading the result generated by the \textit{hardware object}. This task is done by the lines 23-25, the two first lines are the header reply while the other one is the result of the \textit{scale} function that must be translated to a \texttt{float} type through the same function used in the co-simulation environment. Finally, we free the pointer.

Figure 3.37 shows the report obtained after running the test cases onto FPGA’s ARM processor. It looks like pure software domain results (see Figure 3.30).
3.5 Summary

In this chapter, we presented the main concepts of unit tests and testing frameworks. We described how we can apply unit testing in hardware designs and introduced a new concept: **hardware object**. By this approach the restrictions imposed by HLS tools can be overcome and the unit test principles are fulfilled, ensuring an individual accessing per each function. It entails a communication protocol to identify which function must be exercised, accessing each function by the same entry point. In the same context, the stimuli must be serialised as we explained. In addition, our **hardware objects** bring bus compatibility with any standard bus, thus the approach is over the bus data channel, and we provide a bus driver for AXI, but it can adopt another one, even a point-to-point bus, such as AXI-Stream.

Moreover, we described a **hardware verification platform**, which brings a quick and hybrid real environment due to DPR feature, reducing the consumption-time and consumption-power. We isolated the part that is continuously changing: the DUT.

Finally, we described a new development flow based on the traditional one, but **reusing test cases**. This approach enables to use the same test suite independently of the DUT’s abstraction level: high-level modelling, HDL, RTL or running on a hardware device. The test cases are described by means of a testing framework and they are unalterable because a virtual representation of the **hardware object** is used for exercising the real one.
Timing Measurement in Hardware Unit Testing

«We build too many walls and not enough bridges»
Isaac Newton

4.1 Verification Platform: Test Manager
4.2 Unity extension: Unity Time Framework
4.3 Summary

At this point, we have described how unit testing is integrated in the hardware domain. Thus, we only are able to check the behaviour of our hardware design at different abstraction levels reusing the same test cases, by performing a pure functional verification. However, in most hardware projects we must consider another issue: the timing factor. This new issue makes the verification process more complex for hardware engineers. Timing results in-hardware level verification usually differ from verification models used in simulation, in which developers do not care about available resources and their location. This leads to differences in the internal propagation time of signals between the simulated design and the real one.

The approach presented in this chapter is focused on the last verification stage, measuring the elapsed time by a function during the execution. It is not necessary to measure the time in all verification steps, even in high-level modelling and RTL we only check the DUT’s behaviour. On the other hand, reports of HLS tools show a profiling of timing and resource utilisation of each module after its synthesis. However, in the hardware verification stage, we find a blank. Engineers must include special third-party components to measure the time of their hardware designs or use expensive additional hardware. This is high-risk, because the engineer’s experience becomes too important.

When we explained our hardware verification platform in the previous chapter (see Figure 3.24), we did not explain the Test Manager component. In addition, we extend the Unity testing framework to fulfil hardware timing issues.
4.1 Verification Platform: Test Manager

To measure the execution time of a hardware object when it runs a function, we need to include a new hardware object into our hardware verification platform: Test Manager object. It is the main responsible to observe the transactions between the dpr_bridge core and the DUT area. Therefore, it plays the role of a spy, observing the transactions where it is connected. Figure 4.1 shows the block diagram of the Test Manager object. The object is divided into two big blocks; on the one hand its admin part, which has been described from high-level modelling using C programming language and following our hardware object approach, and on the other hand a handler part, which does low-level operations.

The handler part manages bit-level tasks, which include capturing the reading and writing operations through two input flags, flagRD and flagWR signals respectively. In addition, it provides two reset signals and a clock enable signal, which are used for managing the testing process. All these signals are driven by the admin part and depend on the configuration provided by the test case. Moreover, the handler module contains a timer that allows measuring the execution time of a function.

On the other hand, the admin module has been built with the C programming language, and it is composed by three functions: reset, configure and getTime. The following lines describe them.

void reset() This function sends an internal reset to the components that we want to reset in a concrete time. The rstTM signal is active-high during a cycle when a reset function is called upon. This signal is connected to the dynamic area, in order to reset this functionality before running another test case. Therefore, we must call upon this function before exercising the DUT.
void configure(int enable, short readings, short writings)  This function configures the Test Manager object. The function signature has three arguments. The first one is related to how many cycles the clkenTM signal is active-high. This signal plays the role of a clock enable that is connected to the dynamic area. To get a continuous clock enable, we must assign 0 value to this argument. By default, the enable argument is 255.

The readings argument depicts how many 32-bit words are expected before the increment of the internal counter (timer), which plays the role of a chronometer, while the writings argument depicts the number of 32-bit words expected of a reply message to stop the internal counter. These arguments manage the internal counter, allowing to shrug off part of a message, such as headers. For instance, in our communication mechanism the header’s width is usually two 32-bit words, so to discriminate this header, the value of the readings argument must be 3, while the writings must be 1, when the function has just finished. By default, these arguments are 1.

The rstTMInit signal is active-high during a cycle when the number of reading transactions matches with the readings configuration. This signal announces the moment that the internal timer is going to start.

int getTime()  After execution of a test case, we are ready to get the function’s execution time, which is depicted in cycles. In order to get this value, we must call upon the getTime function which returns the timing information about the last function executed.

Summarising, the Test Manager object carries out an amount of hardware verification tasks that allows the following operations: resetting the dynamic area or DUT before running a test case, configuring the measurement parameters, configuring the enable time to run a test case and getting the elapsed time. All these operations are done by the ARM processor, thus the Test Manager object is connected to the same AXI bus. Therefore, we use the axi2fifo driver in order to bridge AXI messages. In addition, the Test Manager’s signals are connected point to point, with the dynamic area as Figure 4.2 shows.

The reader can observe some differences between this new hardware platform and the platform shown in Figure 3.24. The new hardware verification platform includes a new hardware component: the chrono component. It manages the reset signal of our dynamic area, according to three signal rests: global reset and the two reset signals provided by the Test Manager object. Therefore, the Test Manager object does not reset directly over the dynamic area as shown in the older verification platform, it is done by a chrono component. Moreover, the chrono component provides a relative time through an internal timer which is restarted when the rstTMInit signal is active-high. This time is served by an HLS-Stream interface.
4.2 Unity extension: Unity Time Framework

Now the DUT’s interface does not match with the entry point defined in our proposal. The interface needs a new wrapper that connects these signals properly. The flagRD and flagWR signals are directly connected to the rd and wr signals respectively, while the clkenTM signal is not used now, it will be used in other testing approaches that are explained in the next chapters. The relative timer signals are not used either as they are reserved for a future use in this dissertation.

4.2 Unity extension: Unity Time Framework

After we depict the Test Manager object, we need a way to access it from the ARM processor. We propose the same solution that we apply for reusing test cases: a virtual representation of the Test Manager object. But with one important difference, the testing framework is the only entity that is able to execute the Test Manager functions. Therefore we must extend the Unity testing framework in order to include these functions and extend the macros in order to check the timing factor.

Listing 4.1 depicts the virtual representation of the reset function. The first lines assign the default values to Test Manager’s global variables. Then we send a message according to the communication mechanism format described in Section 3.2.2, using the mmap function. In this case, the request message has not got payload, because of the reset function has not got any argument, thus we only need to send. The header contains all information. On the other hand, the Test Manager object replies to another message that informs that it receives the reset request (line 30). This reply message only contains a 32-bit word that matches with the shorter reply header. The rest of the Test Manager functions follow the same philosophy as the reset function.
Listing 4.1: Virtual of \textit{reset} function

```c
void _UnityTimeReset()
{
    _inputWords = 0x0001;
    _outputWords = 0x0001;
    _enableCycles = 0x00000100;
    _hw_addr = 0x41000000;

    #ifndef RCUNITY_TEST
    int fd;
    void *ptr;
    unsigned page_size=sysconf(_SC_PAGESIZE);
    int _head;
    fd = open("/dev/mem", O_RDWR);
    if(fd < 1)
    {
        printf("Cannot open /dev/mem for writing\n");
        return;
    }
    ptr = mmap(NULL, page_size, PROT_READ|PROT_WRITE,
                MAP_SHARED, fd, _hw_addr);
    *((unsigned *)ptr) = 0x00010100;
    *((unsigned *)ptr) = 0x00110000;
    _head=*((unsigned *)ptr); // head
    munmap(ptr, page_size);
    if(_head == 0)
        printf("DummyCheck\n");
    #endif
}
```

\textit{Unity} testing framework is extensible. This means we can build new macros to ensure new verification requirements. In order to cover a real-time analysis we expand this framework, adding timing functions that allow the testing framework and the \textit{Test Manager} object to communicate. This extension is denoted as \textit{Unity time} and it is based on new functions that allow the configuration and rescue function execution time of the \textit{Test Manager} object. Therefore, \textit{Unity Time} extension must be able to manage this component.
As we mentioned before, we can reset the dynamic area from a test case with the new macro `TEST_RESET`. In addition, we can configure the `Test Manager` object depicting the hardware address where it is mapped, the number of cycles that the clock enable must be active-high and the number of expected readings and writings. All these configurations are done with the configuration macros shown in Table 4.1. However, this configuration does not take effect until the `TEST_CONFIGURE` macro runs, except the `CONFIGURE_HW_ADDR` macro which takes effect immediately.

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEST_RESET</td>
<td>Sends a reset signal to the dynamic area. It calls upon the <code>reset</code> function.</td>
</tr>
<tr>
<td>CONFIGURE_HW_ADDR(addr)</td>
<td>Configures the hardware address where is mapped the <code>Test Manager</code> object. By default <code>0x41000000</code>.</td>
</tr>
<tr>
<td>CONFIGURE_SKIP_INPUT(words)</td>
<td>Configures the number of 32-bit words ignored by <code>Test Manager</code> object before start its internal counter. By default 1.</td>
</tr>
<tr>
<td>CONFIGURE_SKIP_OUTPUT(words)</td>
<td>Configures the number of 32-bit words ignored by the <code>Test Manager</code> object before stop its internal counter. By default 1.</td>
</tr>
<tr>
<td>CONFIGURE_ENABLE_CYCLES(cycles)</td>
<td>Configures the number of cycles that <code>clk_en</code> must be active-high. By default 255.</td>
</tr>
<tr>
<td>TEST_CONFIGURE</td>
<td>Sends the configuration values to the <code>Test Manager</code> object. It calls upon the <code>configure</code> function.</td>
</tr>
</tbody>
</table>

| Table 4.1: Configuration macros of Unity Time |

In order to verify the time execution, we have defined an amount of timing macros, whose signature matches with the template `TEST_ASSERT_TIME_XX(expected)`. The XX indicates a comparison operator as shown in Table 4.2. When a function fails because the timing requirements are not met, the `Unity Time` testing framework shows one of the following error messages. This message shows the expected time value depicted in cycles and the real time obtained.

```
Time Expected XX Was YY
  XX not less than YY : TIME FAIL!!!
  XX not greater than YY : TIME FAIL!!!
```
4. Timing Measurement in Hardware Unit Testing

<table>
<thead>
<tr>
<th>Macro</th>
<th>XX</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEST_ASSERT_TIME_EQ(expected) EQ</td>
<td>EQ</td>
<td>Compares the time obtained from <code>get-Time()</code> function and expected value for their equality.</td>
</tr>
<tr>
<td>TEST_ASSERT_TIME_LT(expected) LT</td>
<td>LT</td>
<td>Compares the time obtained from <code>get-Time()</code> function is smaller than the expected value.</td>
</tr>
<tr>
<td>TEST_ASSERT_TIME_LE(expected) LE</td>
<td>LE</td>
<td>Compares the time obtained from <code>get-Time()</code> function is smaller or equal than the expected value.</td>
</tr>
<tr>
<td>TEST_ASSERT_TIME_GT(expected) GT</td>
<td>GT</td>
<td>Compares the time obtained from <code>get-Time()</code> function is greater than the expected value.</td>
</tr>
<tr>
<td>TEST_ASSERT_TIME_GE(expected) GE</td>
<td>GE</td>
<td>Compares the time obtained from <code>get-Time()</code> function is greater or equal than the expected value.</td>
</tr>
</tbody>
</table>

Table 4.2: Comparison operators

Listing 4.2 illustrates an example of a test case using our approach. This example verifies that the `scale` function takes between 30 and 40 cycles for the input floating-point value 9.0, the result value is verified as well. Firstly, we must configure the hardware address where the `Test Manager` object is mapped. The example assigns the same hardware address as the default value, this step is not necessary because our `Test Manager` object is mapped at the default address. The rest configuration assigns the number of input and output 32-bit words in order to manage the internal counter, in our example we denote that we expect 3 reading words before increasing the internal counter and only one 32-bit word to stop it, thus we eliminate the header processing. In addition, we disable the clock enable, setting its value to 0. For this configuration to take effect, we use the `TEST_CONFIGURE` macro. Before executing this macro, we ensure that the DUT is in its initial state, for this we run `TEST_RESET` macro. Following this configuration we exercise the DUT with stimuli as we have done in a pure software version, waiting a DUT response to verify it (line 11). Finally, we check the time execution of the implemented `scale` function through two new macros, verifying that the elapsed time is between 30 and 40 cycles. In order to keep the same test case, we have defined a symbol: `TIMING`. This symbol and the preprocessor blocks (`#ifdef`) contain those function invocations related to timing measurement. Thus, a cross-compiler generates an executable since developer files, which depends on the `TIMING` symbol. If the symbol is defined at compilation time, the compiler includes the functions related to timing measurement.
The report of the Unity Time framework after running the test suite with timing annotations is the same that Figure 3.37 shows. In order to illustrate that our extension measures the execution time of a function, we replace all timing macros for \texttt{TEST\_ASSERT\_TIME\_EQ(time)}, where \texttt{time} is done according to the profiling by the Vivado HLS tool (see Figure 3.35). The result is shown in Figure 4.3
As we can observe, the *Vivado HLS* timing profiling is not accurate, according to the results obtained in our hardware verification platform, namely in a real device. This makes it think that our design meets the timing requirements of the first steps of the development flow, when in really it does not meet those.

FPGA vendors provide special cores to measure elapsed time of tasks. For instance, the *AXI Timer* from Xilinx allows engineers to measure the time elapsed by their code from software domain using different configurations [Xil16a]. However, core’s results are not accurate when one wants to measure the time of his accelerated designs (his own hardware cores) because engineers must send AXI messages in order to write the internal *control* register of the component, and hence to stop and to start the internal counter of the timer component. Therefore, bus arbitration times are inside the return value. Altera provides a similar component called *Timer Core* [Alt07].

Our solution measures the elapsed time of a core in accordance with the input and output channels of the dynamic area or, in other words, channels of DUTs. Thus, we remove the bus arbitration time from the returned time value. In addition, we are able to discard 32-bit words in the beginning and in the end of transactions configuring the *Test Manager* object from software like FPGA vendors do.

### 4.3 Summary

In this chapter, we presented a new version of a hardware verification platform, which is an evolution from the previous one. It entails a new *hardware object*, *Test Manager* object, that is able to carry out hardware testing tasks, including timing measurement.

Moreover, we have extended the *Unity* testing framework used in all experiments of the previous chapter. We denote this extension as a *Unity Time* testing framework. It brings new testing macros that simplify the DUT timing verification process. We must only be concerned about how much time the function that is being verified takes.

The main contribution of this chapter is based on precisely measures and reusability. Precisely measures are reached due to the *Unity* testing framework extension, which allows to measure the time elapsed by DUT’s functionality in a real scenario. In addition, the framework extension provides new macros to configure the hardware verification environment without building a new hardware platform, enabling
the hardware verification reusability. Thus, engineers can reuse our hardware platform environment as well as their test suites are kept. Besides, the proposed platform does not need special components provided by FPGA vendors, such as hardware timers. Our verification platform is able to perform hardware verification tasks itself.
Debugging an FPGA-based system can be frustrating due to the lack of adequate internal visibility, even this visibility becomes a big issue. It entails a new challenge: *maximise or rising internal visibility*. There are two methods of taking internal trace measurements; routing nodes to pins and using a traditional external logic analyser, but usually, engineers are limited by the number of FPGA pins; or inserting a logic analyser core into the FPGA logic design part and routing out via JTAG, trace capture stored using internal FPGA memory, but this method is limited by the device’s logic size [Woo03].

Our upfront planning in order to reduce the time-consuming process of debugging hardware designs based on FPGAs and rise the internal signal visibility is:

- Implement a mechanism into the FPGA using our approach (*hardware objects*) to enhance the ability to check in-situ crucial signal data.
- Retrieve the wrong data indicating the time when it happened, allowing traceability.

Assertions fulfil these tasks perfectly. Therefore, our new challenge is to enable assert functions in the middle of our high-level description design and retrieve information from the test case.
5.1 Signal Visibility

A challenge to ensure that a hardware design is free of design errors, is caused by the internal signal visibility. Usually, signals can only be observed if they can be driven to pins of a chip, adding special debug components inside their hardware designs to record a small subset of signals at-speed for later off-chip analysis. However, the selection of which signals should be recorded is another problem. Some works propose to add hardware assertions translating this non-synthesizable technique into an FPGA.

5.1.1 Hardware Debugging Techniques

**Logic analysers** are used for debugging hardware designs. This technique involves bringing out the internal signals as a debug bus on the external pins of an FPGA and observing them on a logic analyser. However, this technique is only helpful for small logic designs and it entails some drawbacks, such as the limitation of the number of probes by the number of free pins available on an FPGA, or routing the internal signals from the hardware design to these available pins are done by handmade. Usually hardware logic analysers are too expensive.

Some complex logic implementations demand real time debugging or **on board debug logic**, in order to get better insight into the hardware design. This technique requires to insert debug logic, along with the hardware design itself, on an FPGA. These special debug components are drivers and monitors that allow to exercise the DUT and check its output, respectively. These monitors and drivers are not reusable, they are created ad-hoc for testing a hardware design. Besides, adding extra logic can lead to difficulties in meeting the desired timing requirements for the design.

Finally, and maybe the debugging technique most used, we find the **real-time debug tools**. These tools eliminate the need for propagating the internal signals on top of an FPGA. They utilise the FPGA resources to store data without requiring any external hardware. The FPGA vendor tool enables to read the data stored inside the FPGA. The debug window of this technique depends directly on the memory available for storage of dumped data. One example of this kind of debugging is the **Integrated Logic Analyzer** (ILA) from Xilinx [Xil14]. ILA is a customised logic analyser core that can be used for monitoring the internal signals of a hardware design. Signals of a hardware design are connected to an ILA component. These signals are sampled at design speed and stored using an on-chip block RAM when the Boolean trigger equations are evaluated true. Communication with the ILA component is driven by the JTAG interface of the FPGA. After the trigger occurs, the sample buffer is filled and upload into the Vivado logic analyser, which is the waveform tool of the FPGA vendor [Xil14].
5.1.2 In-Hardware Assertions

Although section 2.3.1 introduced this method, we try to give an overview of this technique in the hardware domain. Remember that an assertion is an expression that returns a Boolean value, indicating an error when its return value is false. Assertions within hardware are used for debugging hardware designs, checking their behaviour and displaying a message if a bug occurs. Assertions are generally used as monitors looking for bad behaviour, but may be used to create an alert for desired behaviour as well [SA01] [Sto02].

Summarising, assertions provide the following goodness for engineers. This goodness has encouraged its use to give a solution that entails rising the signal visibility and signal traceability.

- Providing internal test points inside the hardware design, thus increasing observability of the design.
- Simplifying the detection of bugs by localising the occurrence of a suspected bug to an assertion monitor.
- Increasing both controllability and the observability when formal verification is used.

However, assertions entail a technical problem: assertions are not synthesizable. Assertions are generally used as monitors looking for bad behaviour in simulation, but in-hardware verification cannot be directly applied. In the literature review, we find some related works about how assertions can be synthesizable and how to include them into hardware designs.

For instance, work [KWS12] proposes to parse the assertion into an internal representation that can be described with a graph. Hardware assertion configures an FPGA part structure into a circuit which is called Hardware Checker and is responsible for testing a given property. Hardware Checker generates an error signal which may be propagated out of the FPGA structure in which prototyping or emulation takes place and then can be associated with the proper assertion in the source code of the given module. This enables the designer to track down the actual reason for erroneous behaviour captured in hardware to be found in the simulator environment which provides much higher observability of the design.

Figure 5.1 shows an example of this approach. The assertion example recognises the occurrence of sequence a and then identifies sequence b. It illustrates the graph representing assertion and the generated code described from verilog. In this code signal, i[0] represents sequence a; i[1] represents sequence b and the signal o represents the value of the assertion.
In the same research line, we find a number of works that firstly analyse the assertion to assemble it automatically from synthesizable blocks [BZ08]. [US13] assembles from synthesizable basic building blocks by some Haskell functions and the described assertion checkers are used for both simulation and synthesis purposes without any change.

On the other hand, and closer to our approach, the work [HCL14] proposes an automated methodology that synthesises ANSI-C assertions as On-Chip Monitors during a High-Level Synthesis of hardware accelerators. The generated On-Chip Monitors checks at run-time both specification and implementation assertions in order to enhance embedded system monitoring.

Most proposals are oriented to hardware designs described from an HDL, however if we use a high-level programming language to depict our design, we should insert these assertions after HLS tool translates the high-level description. It means the HLS output is manually modified by the designer, which is prone to human errors. Therefore, we propose to provide a hardware assertion library described from C language, to rise the signal visibility. This way the engineer can debug his design without a big effort.

5.2 Hardware Asserts Library

HLS tools entail visibility problems. The code generated by this kind of tools is highly complex and even requires a big effort for signal traceability. Engineers are faced with these problems when examining the simulation scheduler to find the bug. Therefore, it would be great to add a function to inspect the internal variables of our
design. Assertions fulfil this task perfectly. They present a new challenge and we must
enable the following functions in the middle of our release code using our hardware ver-
ification platform and retrieve its information from the test case. In the assert example,
the type refers to user type, such as int, while XX refers to a comparison operator, such
as LT (Less Than). Both arguments must be two values according to the user type of
function signature. For instance, if the assertion signature is ASSERT_INT_LT, both
expected and actual should be integers.

\[
\text{ASSERT}_\text{<TYPE>}_\text{XX}(\text{expected}, \text{actual})
\]

To explain our proposal, we recover the \(l^2\)-norm case study algorithm and our
solution depicted in section 3.2. But in this section we consider the \(l^2\)-norm function
instead of scale function. This means the original top that Listing 3.4 shows. Following
the above premise, we can add assertions to the code that will be synthesised. For
example, we include two assertions between the sum_hist_pow function and the scale
function, both check the result of the sum_hist_pow function; the first one is correct
while the second will fail when the test case is run. In addition, we include another
hardware assertion between scale and mult_hist_scale that checks the return value of
the scale function, it must be less than a fixed value. Listing 5.1 depicts the \(l^2\)-norm
function body with these asserts. Note that all asserts compare single-precision floating-
point values.

```
Listing 5.1: Body of \(l^2\)-norm function with asserts

float scale = 0.f;
float sum = 0.f;

sum = sum_hist_pow(histIN);
ASSERT_FLOAT_EQ(1240.0, sum);
ASSERT_FLOAT_EQ(1.0, sum);
return_scale = scale(sum);
ASSERT_FLOAT_LT(1.0, return_scale);
mult_hist_scale(histAUX_1, return_scale, histOUT);
```

5.2.1 Hardware Domain of an Assertion

When an assertion is called upon, it compares at run-time the input values
according to the correct comparison operator. In our example, the first assert matches
with the operator ==. The first step is getting a relative time provided by the verification
platform, thus we capture the exact time when the assertion was called upon. In the
same cycle, the values are compared according to the comparison operator. If the
comparison is fulfilled, the \texttt{callCount} variable is incremented. The \texttt{callCount} variable stores the number of times that this assertion has been executed. If the comparison fails, the assertion stores a trace of the failure into an internal FIFO, which is composed by the value of \texttt{callCount}, a \texttt{timestamp}, the \texttt{current value} and the \texttt{expected value}. The depth of internal FIFO, which stores the trace failures, is 32. Then, both the \texttt{failureCount} and the \texttt{callCount} are incremented.

Figure 5.2 shows an overview of a hardware assertion. This structure is the same for each assertion type, except for the comparison operator. It means that the assertions of the same type and the comparison operator share the same variables and FIFO. For instance, in our example the two first assertions share the same variables: \texttt{callCount}, \texttt{failureCount} and \texttt{failures}. Thus, the \texttt{callCount} variable is 2, while the \texttt{failureCount} is 1. Using a FIFO to store the failure creates makes a sequential retrieval, thus the first failure will be the first returned.

![Figure 5.2: Overview of hardware assertions](image)

Translating Figure 5.2 into C code results in Listing 5.2. It depicts the body of the \texttt{ASSERT\_FLOAT\_EQ} assertion.

```
1 tASSERT\_FLOAT\_FAILURE _auxFailure;
2 unsigned int _time = timeClock.read();
3
4 if(actual!=expected){
5   _auxFailure._timestamp = _time;
6   _auxFailure._callCount = assertFLOAT\_EQ\_callCount;
7   _auxFailure._actual = actual;
8   _auxFailure._expected = expected;
9   assertFLOAT\_EQ\_failure.write(_auxFailure);
10  assertFLOAT\_EQ\_failureCount += 1;
11 }
12 assertFLOAT\_EQ\_callCount += 1;
```
The codes listed in Listing 5.1 and Listing 5.2 do not show the signature of both functions. The variables of assertions are not inside the assertion’s code, the solution passes them by reference to the top level of the hardware object that contains it. Therefore, assertion signatures are listed in Listing 5.3, the signature $l^2$-norm algorithm follows the same idea, it adds the internal variables. In addition, the reader can observe that the assertion signature does not match with the assertion call upon, we use macros to reduce the number of arguments. The variables are implicitly propagated and the code engineer is reduced, thus he can use an assertion like in the software domain. The macro associated to this assertion is listed at the beginning of Listing 5.3.

Figure 5.3 shows an overview of our hardware object with assertions that depict the $l^2$-norm case study. The object really contains two objects, the $l^2$-norm implementation itself and the assertion object, both objects contain different identifiers, but the same nodeID identifier. It works the same as illustrated in chapter 3, but in this case the facade module routes messages to the correct object according to the objID value. Both objects work together in order to manage the internal assertion variables, which are placed at the top-level of the hardware object. These variables are passed by reference, thus all functions access the same resource. Assert functions write over these resources while functions of the assertion object read these values in order to send the values when a test case retrieves them.

The assertion object contains three methods by each assertion type. These methods return the values stored into the internal variables; failureCount and callCount which return an integer each one; and failures which returns a user type with four attributes: two integers, related to the callCount identifier and the relative time when the failure happens, the other two attributes are two single-precision floating-point values, which match with the expected and current value. This entails a serialisation step like the proposed in section 3.2.3.
Unfortunately, Figure 5.3 only shows the hardware assertion of \texttt{FLOAT_LT} type. However, it misses other assertions to fulfil the code listed in Listing 5.1. The hardware assertion is related to the \texttt{FLOAT_EQ} type. Therefore, the image should contain three other methods included inside the same object, but with different method identifiers, thus the object identifier 11 is reserved to asserts functions. In addition, the internal variables associated to the \texttt{FLOAT_EQ} assertion are missing too, so they should appear in the figure. Moreover, if engineers want to include an integer hardware assertion in his code, we include a new \texttt{assertion object}, thus the 11 object manages float hardware assertions and, for instance, the 12 object manages the integer hardware assertions. Table 5.1 lists the identifiers of each assert function according to its comparison operator. The assertion type is determined by the object identifier.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
\textbf{Operator} & \textbf{Function} & \textbf{callCount} & \textbf{failureCount} & \textbf{failures} \\
\hline
TRUE & FLOAT_EQ_callCount & 23 & 24 & 26 \\
EQ & FLOAT_EQ_failureCount & 33 & 34 & 36 \\
GT & FLOAT_EQ_failures & 43 & 44 & 46 \\
LT & & 53 & 54 & 56 \\
GE & & 63 & 64 & 66 \\
LE & & & & \\
\hline
\end{tabular}
\caption{Identifiers of assert functions}
\end{table}
Table 5.2 lists all assertions implemented in the hardware asserts library. It only shows the comparison operator, so it fits in the table. A short description is attached to each hardware assertion.

<table>
<thead>
<tr>
<th>Assert (Macro)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASSERT_TRUE(condition)</td>
<td>Evaluates whatever code is in condition and fails if it evaluates to false</td>
</tr>
<tr>
<td>ASSERT_&lt;TYPE&gt;_EQ(expected, actual)</td>
<td>Compare two &lt;type&gt; for equality and store a failure trace.</td>
</tr>
<tr>
<td>ASSERT_&lt;TYPE&gt;_LT(expected, actual)</td>
<td>Compare two &lt;type&gt; for expected value is less than actual. Failures are stored for its traceability.</td>
</tr>
<tr>
<td>ASSERT_&lt;TYPE&gt;_LE(expected, actual)</td>
<td>Compare two &lt;type&gt; for expected value is less or equal than actual. Failures are stored for its traceability.</td>
</tr>
<tr>
<td>ASSERT_&lt;TYPE&gt;_GT(expected, actual)</td>
<td>Compare two &lt;type&gt; for expected value is greater than actual. Failures are stored for its traceability.</td>
</tr>
<tr>
<td>ASSERT_&lt;TYPE&gt;_GE(expected, actual)</td>
<td>Compare two &lt;type&gt; for expected value is greater or equal than actual. Failures are stored for its traceability.</td>
</tr>
</tbody>
</table>

Table 5.2: Hardware assertions

5.2.2 Software Domain of an Assertion

Since the test case, a developer must be able to retrieve the internal variables to trace his hardware design. Listing 5.4 depicts a test case for the $l_2$norm function using hardware assertions and their corresponding software function that allows to retrieve values of internal variables. Lines 4-29 match with a test case using the Unity Time testing framework, including timing analysis. The difference can be found in the rest lines, which call upon virtual functions that retrieve the internal values of assertions.

**ASSERT_<TYPE>_<XX>_CALLCOUNT** Retrieves the number of functions calls done by the hardware design during its execution. The retrieved values are related to the assertion’s type and comparison operator.

**ASSERT_<TYPE>_<XX>_FAILURECOUNT** Retrieves the number of failures of an assertion, which have taken place in the hardware design during its execution. The retrieved values are related to the assertion’s type and comparison operator.
PRINT_FAILURES_ASSERT_<TYPE>_<XX> Prints all failure traces of an assertion when the failureCount variable is greater than zero, otherwise it does nothing.

Listing 5.4: Test case for l2norm function

```c
void test_l2norm()
{
    const float ref[] = {0.0, 0.027164, 0.054328, 0.081492, 0.108655, 0.135819, 0.162983, 0.190147, 0.217311, 0.244475, 0.271639, 0.298802, 0.325966, 0.353130, 0.380294, 0.407458};
    float input[HIST_SIZE];
    int i;
    for (i = 0; i != HIST_SIZE; i++)
        input[i] = i;
    #ifdef TIMING
    CONFIGURE_SKIP_INPUT(18);
    CONFIGURE_SKIP_OUTPUT(1);
    CONFIGURE_ENABLE_CYCLES(0);
    TEST_CONFIGURE();
    #endif
    float out[HIST_SIZE];
    l2norm(input, out);
    #ifdef TIMING
    TEST_ASSERT_TIME_GT(220);
    TEST_ASSERT_TIME_LT(750);
    #endif
    for (i = 0; i != HIST_SIZE; i++)
        TEST_ASSERT_EQUAL_FLOAT(ref[i], out[i]);
    // Assert functions
    printf("CallCount (FLOAT_EQ) \%d\n",ASSERT_FLOAT_EQ_CALLCOUNT());
    printf("FailureCount (FLOAT_EQ) \%d\n",ASSERT_FLOAT_EQ_FAILURECOUNT());
    PRINT_FAILURES_ASSERT_FLOAT_EQ();
    printf("CallCount (FLOAT_LT) \%d\n",ASSERT_FLOAT_LT_CALLCOUNT());
    printf("FailureCount (FLOAT_LT) \%d\n",ASSERT_FLOAT_LT_FAILURECOUNT());
    PRINT_FAILURES_ASSERT_FLOAT_LT();
}
```
5.2.3 Generation of a **Hardware Object** with Assertions

After the definition of hardware assertions in both domains, we ask ourselves the next question: *How do I build a hardware object which contains hardware assertions?* The answer is too easy: `c2hwobject` tool enables this kind of objects, but it requires annotations.

Remember, the functions which call upon hardware assertions must change their signature and add an associated macro according to the hardware assertions used in order to propagate the internal variables of each hardware assertion. To overcome this task, a developer annotates a new *pragma* inside the header file to depict the hardware assertions needed by a function. For example, in our case study, the *pragmas* needed for the `l2norm` function are listed in Listing 5.5.

```
Listing 5.5: *Pragmas* for `l2norm` function

1. ...
2. #pragma ASSERT type=FLOAT op=EQ func=l2norm
3. #pragma ASSERT type=FLOAT op=LT func=l2norm
4. void l2norm(float histIN[HIST_SIZE], float histOUT[HIST_SIZE]):
```

The `c2hwobject` tool provides the *virtual representation* of each assertion, thus the developer only has to verify his hardware design.

5.3 Integration with our Verification Platform

Hardware asserts are fully compatible with our hardware verification platform, we only have to fit the dynamic area interface with the *hardware object* interface. Now, the *hardware object* interface contains a new input: the relative clock timer provided by the *chrono* component. To overcome this interface compatibility, we introduce a wrapper to bridge between both interfaces.

Figure 5.4 illustrates the elements that take place in the software domain of an FPGA. This part contains all *virtual representations* used for stimulating the DUT. In addition, Figure 5.4 shows the report obtained after test case execution. The hardware design passes the test case explained in this section successfully, but it finds a failure during its execution. The test case prints information about hardware assertions, which is shown in the report. We observe that hardware assertion `ASSERT_FLOAT_EQ` was called upon twice and a failure happens in one of these calls. Note that the index starts at 0. The failure trace gives more information about the failure, it depicts that the failure happens during the second call at the 183 cycle. This time is calculated according
5.4 Summary

In this chapter, we explained hardware assertions. These assertions rise the internal signal visibility without a big effort. They do not require special components such as logic analyser cores, which entail a whole synthesis and are time and power consuming. In addition, it requires human intervention to check the signals in a simulator. Therefore, the hardware object approach provides a value-added: it does not need a special component or a complex flow.
Furthermore, the hardware assertions described in this section are compatible with our current hardware verification platform. Even test cases are able to retrieve information about hardware assertions. Failures can be traced, retrieving a relative time when it happened, the number of assertions that fail, the actual and expected value.
Chapter 6

Hardware Mock Functions

«Tell me and I forget, teach me and I may remember, involve me and I learn»
Benjamin Franklin

6.1 Test Doubles
6.2 Hardware Mocks
6.3 Integration with our Verification Platform
6.4 Summary

Usually, a hardware component has several dependencies with third-party components, external devices, ..., however the component should be verified without these dependencies. Even if, a part of the hardware design is not clearly defined or its implementation is a difficult task. These dependencies entail some problems and make the verification stage more complex. Some third-party providers do not give a full access to their product, we have to pay to use them, or sometimes the simulation models do not match with the implemented design. This fact opens a new challenge: reduce third-party dependencies and their problems.

In order to overcome the problem of third-party dependencies, we propose special functions inside our hardware objects. The behaviour of these functions looks like their release-intended counterparts, reducing the complexity and facilitating testing process. It means they replace a third-party dependency, similar to how a stunt double stands in for an actor in a movie. These kinds of functions are based on the Test Double technique. This enables to set the behaviour of the Test Double and run a test case. A Test Double object is asked in order to get information about what happened during the test execution.
6.1 Test Doubles

Someone can ask himself, *how can I verify my design when it depends on third-party components?* Unfortunately, most designs depend on third-party entities and this makes it difficult to test the DUT, also known as *System Under Test* (SUT), because it depends on other components that cannot be used in the test environment because they are not available. In other cases, we need to control the third-party component easily to achieve the desired behaviour. When we are writing a design in which we cannot use a real *Depended-On-Component* (DOC), we can replace it with a *Test Double*. The *Test Double* does not have to behave exactly like the real DOC, it has to provide the same API as the real one, so the SUT thinks it is the real. This is similar to a situation in which a film actor wants to film something that is potentially risky or dangerous for the leading actor to carry out, so he hires a *stunt double* to take the place of the actor in the scene [Mes08].

*Test Doubles* come in several main flavours and are classified based on how and why we use them. The following lines describe these categories [Mes08] [MGV08].

**Dummy** Passes an object that has no implementation as an argument of a method called on the SUT. Usually they are just used to fill parameter lists.

**Stub** Replaces a real object with a test-specific object that feeds the desired indirect inputs into the SUT. Usually not responding at all to anything outside what is programmed in for the test.

Firstly, we define an interface on which the SUT depends in order to respond those calls done by the SUT with the values that will be exercised by the SUT, thus the SUT is exercised by two entities: the *test case* and the *Test Stub*. Before exercising the SUT, we create the *Test Stub* with the return values. The test case can then verify the expected outcome in the normal way (see Figure 6.1).

![Figure 6.1: Overview of Test Stub](Mes08)
**Fake** Replaces a component that the SUT depends on with a much lighter-weight implementation. It usually takes some shortcut which makes them not suitable for production.

A *Test Fake* is similar to a *Test Stub*. The main difference is that *Test Stub* injects indirect inputs into the SUT, while the *Test Fake* does not. It merely provides a way for the interactions, which are typically many, and the values passed as arguments of earlier method calls will often be returned as results of later method calls. This is in contrast with *Test Stubs* and *Test Mocks*, where the responses are either hard-coded or configured by the test case (see Figure 6.2).

![Figure 6.2: Overview of Test Fake](Mes08)

**Mock** Replaces an object the SUT depends on with a test-specific object that verifies if it is being used correctly by the SUT. They are pre-programmed with expectations which form a specification of the calls they are expected to receive.

A *Mock Object* is defined with the interface on which the SUT depends. Then, during the test, we configure the *Mock Object* with the values with which it should respond to the SUT and the method calls, including the expected arguments. When the SUT calls upon one method of *Mock Object*, it compares the actual arguments received with the expected arguments, using equality assertions and fails the test if they do not match (see Figure 6.3).

![Figure 6.3: Overview of Test Mock](Mes08)
6.2. Hardware Mocks

Spy

Captures the indirect output calls made to another component by the SUT for later verification by the test. It really is a stub that also records some information based on how it was called.

The Test Spy is designed to act as an observation point by recording the method calls made to it by the SUT as it is exercised. During the result verification phase, the test case compares the current values passed to the Test Spy by the SUT with the values expected by the test case.

![Diagram of Test Spy](image)

Figure 6.4: Overview of Test Spy [Mes08]

6.2 Hardware Mocks

In this section, we concern in a new problem, we try to answer the question raised in the previous section, but now in relation to a pure hardware domain (how can I verify my design when it depends on third-party components?). Almost all hardware engineers face this challenge at least once in their lives. Although HLS tools open new horizons, they do not provide a way to solve third-party dependencies, they must be overcome by a trained engineer. Therefore, the question entails new challenges inside this dissertation, we must enable Test Doubles inside our hardware verification platform in order to remove these third-party dependencies of a hardware design. In addition, we must follow our hardware object approach. To overcome this new challenge, we focus on Mocks type.

In order to illustrate how we overcome the hardware mocks, we resume our case study about the $l^2$-norm algorithm. In this new scenario we suppose the scale function is done by a third-party component, or even we could suppose this function has not been defined yet, because there is a number of possibilities to depict it. For instance, Figure 3.34 lists the resource utilisation of each function where the scale function is the stage that uses more resources. Therefore, engineers can code this function as a table which contains return values according to a series of intervals, although the function loses precision it reduces the resources too.
6.2.1 Translating a user function into mock function

Our hardware mocks are defined by a function signature, thus the API is defined by it. From a function signature we generate a hardware mock object that allows indirectly exercising the DUT. Figure 6.5 illustrates this process for the scale function. The hardware mock object contains several functions that enable configuration it and retrieve information about the testing process. Someone could think the generated object is a spy instead of a mock, however the hardware mock object checks that it is being used correctly by the DUT at run-time. Moreover, it can be asked about the checking process, for example, how many calls have been done, how many failures have occurred, ...

![Figure 6.5: Overview of scale hardware mock object](image)

The generation of hardware mock object from a function is realised by our c2hwobject tool, allowing the integration with unit testing frameworks. We need to depict that the scale function must be mocked using a new pragma as Listing 6.1 illustrates. The mocked function, after the mocking process, seems like the scale mock object of Figure 6.5.

Listing 6.1: Pragma example to mock the scale function

```
1 #pragma MOCK func=scale
2 float scale(float sum);
```
In this case, both the mocked function and the scale mock object read from and write to the internal variables. These variables are placed on the top level of the hardware object. The user functions and the mocked function are involved in a hardware object, while the mock functions are involved in another hardware object, the scale mock object. Therefore, the mock object variables store the following information:

- **callCount** Actual number of calls.
- **failureCount** Actual number of failures.
- **return values** FIFO containing all return values for a mocked method.
- **expected values** FIFO containing all expected values for arguments of a mocked method.
- **timestamp** FIFO containing all relative time flags of a mocked method. It stores the time when a mocked method is called upon.

Note that all FIFOs store their data into a structure related to their information, for example if a function contains two arguments, both arguments will form the structure. The following lines describe the mock functions of the scale mocked function. They retrieve the information from the scale mock object internal variables.

```c
void scale_return(float value) This function fills the return FIFO in order to add the return values of the mock object. When the last value is sent, the FIFO is empty and the DUT calls upon the scale mocked function, the mock object blocks the test.

void scale_expect(float value) This function fills the expect FIFO, in order to add the expected values of the mock object. It is able to check the similarity between the argument value and the expected one.

int scale_callCount() This function returns the actual number of calls that the DUT has done. It means the value returned is the callCount variable.

int scale_failureCount() This function returns the actual number of failures. It means that if the call’s argument does not match with the expected value stored in the top of the expect FIFO, it will be considered as failure. Thus, the function returns the value of the failureCount variable.

int scale_timestamp() This function returns a relative time corresponding with each call done. These times are stored in the times FIFO, thus the first time retrieved matches with the first call.
scale_failures() This function returns a failure trace sequentially. All failure traces are stored in the failures FIFO ordered by its detection.

The scale function is a good example because it includes all mock functions that the test case can call upon, however it does not always happen like this. For instance, the signature void foo(int i) does not contain the foo_return function because it is a void function. Table 6.1 lists the possibilities of mock properties.

<table>
<thead>
<tr>
<th>Mocked func.</th>
<th>return</th>
<th>expect</th>
<th>callCount</th>
<th>failureCount</th>
<th>timestamp</th>
<th>failures</th>
</tr>
</thead>
<tbody>
<tr>
<td>void foo(void)</td>
<td>-</td>
<td>-</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>void foo(args)</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ret foo(args)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 6.1: Trade-off between mocked functions and mock properties

Figure 6.6 shows an overview of the scale mocked function. This structure is not the same for each mocked function, however it is the biggest one. It means other mocked functions do not contain all tasks that Figure 6.6 shows. For instance, the void foo(int i) function does not contain the read return task.

From a behavioural point of view, a mocked function starts reading the expected value together, in the same cycle, and the relative time to know the exact time that the function was called. Then, the expected values are compared with the actual values (argument values). The result of this comparison divides the path and if it does not match, the mocked function will write a trace of the failure into the failure FIFO. It allows tracing the failure, retrieving the number of calls that provoked the failure, the relative time of the running call, the actual and expected values. Then the failure counter is increased. At this point both paths are joined, or when the comparison returns a true value, it starts with this step: the mocked function stores the relative time read in the first step, reads the return values and increases the callCount variable. Finally, the return value is returned. Remember that a mocked function contains a reference of those variables that will be modified. These variables are placed on top of the hardware object and they are exclusive for a hardware mock object.

Translating the scale mocked function illustrated in Figure 6.6 into C programming language results in Listing 6.2. The difference between the expected values and the actual values is discovered by an absolute subtraction, whose result is compared with a pre-defined delta value. If the result is greater than this value, it means that the comparison fails and the mocked function stores a trace of the failure.
Figure 6.6: Overview of `scale` mocked function

Listing 6.2: Body of `scale` mocked function

```c
    tSCALE_FAIL auxFail;
    float _return;
    float _expect_sum;
    float _diff;
    unsigned int _time;

    _expect_sum = scale_expect.read().sum;
    _time = timeClock.read();
    _diff = fabsf(sum - _expect_sum);

    if(_diff > DELTA){
        auxFail._callCount = scale_callCount;
        auxFail._param.sum = sum;
        auxFail._expect.sum = _expect_sum;
        auxFail._time = _time;
        scale_fails.write(auxFail);
        scale_failCount += 1;
    }
    scale_timestamp.write(_time);
    _return = scale_return.read()._return;
    scale_callCount += 1;

    return _return;
```
6.2.2 Working with mocked functions

After defining the *hardware mocked functions*, we can ask ourselves, *how we can use mock functions from a test case?*. The answer is to use a *virtual representation* of these *mock objects*. Listing 6.3 lists the $l^2$-norm test case, setting the *scale* function as *mocked function*. Firstly, before using *TEST_CONFIGURE* macro and exercising the DUT, we must fill the *return* and *expect* FIFOs (lines 8 and 9) that allow to work the *scale mocked object* correctly for this test. Then, the DUT is exercised to check its behaviour without a real implementation of the *scale* function, we know that it will receive 1240.0 as an input parameter and return 0.027164, note these values are only valid for this test case. When the DUT finishes its execution, we can ask about the number of calls that the DUT makes to the *scale mocked function*, the number of failures that happened and prints all failures (lines 22-24, respectively).

Listing 6.3: Body of $l^2$-norm test case with mocked functions

```c
...  
#endif TIMING
  CONFIGURE_SKIP_INPUT(18);
  CONFIGURE_SKIP_OUTPUT(1);
#endif

scale_return(0.027164);
scale_expect(1240.0);

#ifdef TIMING
  TEST_CONFIGURE();
#endif

float out[HIST_SIZE];
l2norm(input, out);

#ifdef TIMING
  TEST_ASSERT_TIME_GT(220);
  TEST_ASSERT_TIME_LT(500);
#endif

printf("scale-Callcount %d\n", scale_callCount());
printf("scale-FailureCount %d\n", scale_failureCount());
scale_print_failures();

for(int i = 0; i != HIST_SIZE; i++)
  TEST_ASSERT_EQUAL_FLOAT(ref[i], out[i]);
```
6.3 Integration with our Verification Platform

Like hardware assertions, hardware mocks are fully compatible with our verification platform. We have to fit the dynamic area interface with the same wrapper used for the hardware assertions. The hardware object with hardware mocked functions contains the relative clock timer provided by the chrono component. It enables our hardware mocks to know the relative time when a mocked function is called upon by a DUT.

Figure 6.7 shows the elements that run on the ARM embedded processor of an FPGA and a test report. On the right side, the figure illustrates the elements that take place in our case study, using the scale function as mocked function. This part contains all virtual representations used, including the scale mocked object. On the left side, we show the report obtained after test case execution. The hardware design passes the test case depicted in Listing 6.3 successfully. In addition, the test case prints a report related to the mock object, illustrating the number of calls done by the DUT, the number of failures found and a list of relative times that depicts when the scale function was called.

![Figure 6.7: Report of test case with hardware mocks](image)

To illustrate an error detected by a mocked function, we replace the expected value 1240.0 by 124.0. Thus, the scale mocked function detects a failure and the test case retrieves the failure trace. Figure 6.8 illustrates this new scenario. We observe that the DUT calls upon the scale function at the same time as in the above scenario, but now a failure is detected by the scale mocked function. The report shows that one error happened during the test case, whose trace indicates that it was the first call done by the DUT, the scale function expects an argument whose value is 124.0, but the current value was 1240.0.
In order to propagate the mock failures to the test case, we must add software assertions at the end of the test case, validating that the function mocked does not throw any failure. To carry out this process we use the equal assertion as the following line illustrates.

```
TEST_ASSERT_EQUAL(0, <mock>_failureCount());
```
Using mocked functions generates a problem. Engineers have to be very careful with their configuration. An error in this stage can provoke wrong decisions. For instance, the above examples pass the tests and both tests are valid, but if the engineer makes a mistake configuring the return values, the test will not pass. Figure 6.9 illustrates this scenario, previously we replaced the return value by 0.1. Now, the test fails because the return value provided by the scale mocked function is wrong, however the engineer can think that the failure is located in the next step (mult_hist_scale).

6.4 Summary

In this chapter, we described Test Doubles using mock type to integrate them into hardware. We explained hardware mocks. These mocks allow reducing third-party dependencies without a big effort. Therefore, engineers can check their hardware designs without a third-party component, it lightens up the synthesis process because it is not necessary to synthesise a number of components, we only need to know how it works to configure our hardware mock object. Summarising, a mock object simulates the behaviour of a third-party component.

Moreover, hardware mocks are compatible with our current hardware verification platform. Thus, they enable retrieving information about the mocked function through the test cases. Among the information that a test case can retrieve, is the number of calls done by the DUT. In addition, hardware mock objects allow to trace the failures done by the DUT, checking the input parameters.
Chapter 7

Black-box Designs

«If debugging is the process of removing bugs, then programming must be the process of putting them in»
Edsger W. Dijkstra

7.1 Components of UVM
7.2 Black-box Testing Environment
7.3 Writing Test Cases for Black-box Designs
7.4 Integration with our Verification Platform
7.5 Summary

At this point, we have defined several approaches that overcome some verification problems, such as individual accessing to apply unit testing or reducing third-party dependencies. Unfortunately, the most extended design style is not covered by any scenario shown until now. This style matches with black-box designs, which work as follows; it takes a streaming of data, then performs some operations with these data, and finally returns another streaming of data. An image filter is a good example of hardware black-box designs; the filter takes an image, applies a transformation such as sobel and returns the new image. This kind of design entails a new challenge for this dissertation: testing black-box designs keeping our hardware verification platform and unit testing frameworks.

To overcome this new challenge, we propose a solution based on UVM, specifically based on some elements used in this methodology, such as monitors. Unlike UVM, our approach can be synthesised, thus the design is verified in-hardware. In addition, the exercising task is done by test cases as we have done until now, using our hardware verification platform. It enables us to measure the timing elapsed that carries out the pipeline to perform the operation over the input data.

[ 133 ]
7.1 Components of UVM

Although we explained the UVM in section 2.3.2, this section tries to depict some elements of UVM that are our reference to build our streaming testing environment. In addition, the following sections compare UVM elements with our approach elements.

**Scoreboard** The main function of this component is to check the behaviour of a certain DUT. The *UVM Scoreboard* usually receives transactions carrying inputs and outputs of the DUT through the *UVM Agent* analysis ports, runs the input transactions through some kind of a reference model to produce expected transactions, and then compares the expected output to the actual output [Ace15]. These tasks follow the black-box verification approach, thus we can affirm the *UVM Scoreboard* makes a black-box verification.

**Agent** A *UVM Agent* is a hierarchical component that groups together other verification components that are dealing with a specific DUT interface. A typical *UVM Agent* includes the following elements [Ace15] (see Figure 7.1).

- **Sequencer** A *sequencer* serves as an arbiter for controlling the transaction flow from multiple sequence objects. While a *sequence* is an object that contains a behaviour for generating stimuli, it is not a part of a *UVM Agent*.

- **Driver** A *driver* receives a transaction from the *UVM Sequencer* and drives it on the DUT interface. It spans abstraction levels by converting stimuli into pin-level signals.

- **Monitor** A *monitor* captures the information and is sent out to other UVM components for further analysis. Thus, a *UVM monitor* is the counterpart of a *UVM driver*, it translates from pin-level activity to transactions.

Although more elements take place in a UVM environment, our approach has focused on these ones, exporting their core to get a new verification environment leading by unit testing.
7.2 Black-box Testing Environment

The new challenge can be summarised in the question *how can we verify in-hardware black-box designs?* The following restrictions are added: using our hardware verification platform and unit testing frameworks. To answer this question, we propose to reuse our $l^2$-norm case study. In the new scenario, both input and output channels are streaming channels. Thus, the $l^2$-norm algorithm takes a 4x4 pixel window and returns a normalised 4x4 pixel window continuously (see Figure 7.2). The size of the windows can be anything, for example 8x8. Now, we do not care about how the $l^2$-norm algorithm is described; it can be described by only one function, or for example as we proposed in section 3.2.

Our current hardware verification platform (see Figure 4.2) is able to verify black-box designs because the dynamic area interface contains two streaming channels. Both channels are driven by two FIFOs in the dpr_bridge component. Its behaviour is similar to a UVM driver and an UVM monitor together. It translates AXI write messages to input DUT signals and output DUT signals to AXI read messages. In addition, the dynamic area interface contains some control signals driven by the chrono component and the Test Manager object. However, it only enables the black-box verification approach without measuring timing. On the other hand, the verification is done offline, this implies that the test case retrieves the output data and checks them after the $l^2$-norm finished. Another problem is related to the number of DUT’s streaming channels; the $l^2$-norm interface proposed matches with the dynamic area of our hardware platform, however it is not always fulfilled. For instance, a function that adds two images or adds constant to image requires two streaming input channel and one streaming output channel.

First we show an overview of the proposed approach for verifying black-box designs. This overview is illustrated in Figure 7.3. It has been divided into two parts: producer and consumer. The producer part is the main responsible of providing stimuli to the DUT, checking how it consumes these stimuli through monitors. The consumer part checks the correctness of output channels at run-time through scoreboards and, besides, checks how the DUT produces these results using monitors. For the translation from our communication protocol to pin-level activity and the other way around, we use two drivers, one for each part.
7.2. Black-box Testing Environment

7.2.1 Producer Part

This part is composed by four elements: an input driver and n-elements of a signal buffer, sequencer and monitor. Its general purpose is to provide data for the DUT and to give information about how this data are read.

**Sequencer** The sequencer is responsible to exercise the DUT. The stimuli are retrieved from a small internal buffer that must be filled like in the test case. In addition, a clock enable signal allows controlling the stimulation process, it will enable the sequencer to send the stimuli to the DUT. The interface between the sequencer and the DUT is based on HLS-Stream signals, thus the empty signal driven by a sequencer is active-low until the clock enable signal is active-high. Each streaming channel of the DUT must be led by a sequencer, or they can be grouped into a user type. Moreover, when the internal buffer is empty, the sequencer is able to repeat the last value sent until the clock enable signal is active-low.

Figure 7.4 shows an overview of the sequencer module. The input sequencer interface is equal to the output counterpart. The sequencer module manages the DUT stimulation in accordance with the internal buffer and the clock enable signal (clkenTM).
Translating the above behaviour into a C programming language results in the following code (see Listing 7.1).

```
Listing 7.1: Body of sequencer module

    if (clk_en){
        a_empty_n = true;
        if(a_read){
            if(a_input.read_nb(a_aux)){
                a_aux_tmp = a_aux;
                a_input_last = a_aux;
            }
            else
                a_aux_tmp = a_input_last;
        }
        else{
            a_aux_tmp = 0;
        }
    }
    else{
        a_empty_n = false;
        a_aux_tmp = 0;
    }
    a_dout = a_aux_tmp;
```

**Monitor** A *monitor* on the *producer* side checks the number of times that an input is read and the relative time when it took place. Thus, we can know how many times an input was read by the DUT and the time of each reading. Figure 7.5 shows an overview of this component.

![Figure 7.5: Overview of monitor module (Producer)](image)

Summarising, the *monitor* spies the transactions between the *sequencer* and the DUT. Each captured transaction involves two operations; the first is increasing the `callCount` variable, while the second operation is related to storing the relative time when the DUT reads the input. Remember that the relative time is...
provided by the hardware verification platform through its *chrono* component. Between each *sequencer* module and the DUT is a *monitor* that enables to observe all reading operations done by the DUT.

Listing 7.2 depicts in C programming language the above behaviour. This is the *monitor* behaviour. When a reading takes place, the *monitor* stores the relative time inside a small buffer. This process is not blocking and if there are a lot of transactions, the DUT can continue its tasks, although not all call times will be stored. At the same cycle, the *monitor* increases the *callCount* variable.

```c
1    callcount_a = callcount_a_i;
2    if (trigger_a){
3        timestamp_a.write_nb(timeClock);
4        callcount_a_i +=1;
5    }
```

The *AXI Performance Monitor* core measures major performance metrics for the AMBA AXI system and measures bus latency of a specific master/slave in a system, the amount of memory traffic for specific durations, and other performance metrics [Xil17a].

On the other hand, our monitor measures the number of write requests or read responses of DUTs, like monitor of Xilinx does. In addition, monitor of Xilinx measures the latency which is calculated is between the write address issuance and the last write. Our solution annotates the relative time of each transaction in an internal memory instead of the latency, which can be calculated between the first read/write and last read/write. Thus, our solution provides more information about the transactions.

**Driver** The *driver* module translates the stimuli into pin-level signals in accordance with the DUT interface. This is only one *driver* module in the *producer* side. It is the main responsible to fill the internal buffers and retrieve information from the different monitors when someone requires this information. Thus, it informs about *what is happening on the producer side*, such as number of readings done by the DUT over a specific channel. Therefore, a *driver* module really is a *hardware object*, which is able to offer this information to third-party entities.

Figure 7.6 shows an overview of *driver* modules following our *hardware object* approach. The *driver* contains three methods, although it could be more. It depends on the DUT interface and the number of signal groups created by the
developer. For instance, a rgb to luma converter can have three input channels, one per component. This means that the driver module contains the above three methods repeated three times. These methods contain the following tasks.

- **return** This method fills an internal buffer related to a specific input.
- **callCount** This method returns the number of readings done by the DUT over a specific input.
- **timestamp** This method returns the relative times of each reading done by the DUT over a specific input.

All these methods can run when the clock enable signal is active-low, because it does not affect this module, nor all monitors.

The **VIO core** is a customizable core that can both monitor and drive internal FPGA signals in real time. The number and width of the input and output ports are customizable in size to interface with the FPGA design. Because the VIO core is synchronous to the design being monitored and/or driven, all design clock constraints that are applied to your design are also applied to the components inside the VIO core. Run time interaction with this core requires the use of the Vivado logic analyzer feature [Xil17c].

The VIO core must be configured through Vivado logic analyzer for each input with engineer intervention, this fact makes a hard task to simulate long input stimuli. In addition, one cannot retrieve information about the consumer (DUT).

In our case study we only have one input channel to spy, this means the environment has one sequencer, one monitor and one internal buffer. In addition, the driver is composed by three methods that fill the internal buffer and retrieve the information captured by the monitor. Figure 7.7 illustrates the block diagram of the $L^2$-norm producer side, including the relationship between the above modules.
7.2.2 Consumer Part

This part is composed by four elements: an output driver and n-elements of a signal buffer, scoreboard and monitor. Its purpose is to check at run-time the correctness of the DUT outputs, annotating the time when each event happens. The clock enable signal does not touch on this side.

Scoreboard The scoreboard module is the main responsible to check the output correctness at run-time (see Figure 7.8). The output values are stored into an internal output buffer (actual buffer) and the scoreboard reads them in order to compare these values with the golden values which are stored into another buffer (expect buffer). This last buffer must be filled from the test case. In addition, the scoreboard module is able to check the elapsed time between two consecutive outputs. Firstly, the delay variable must be set, which indicates the maximum time between two data. Therefore, the scoreboard module retrieves the time when the data was written into the output buffer (actual buffer) from the times buffer and compares the previous time with the actual time; it subtracts the times and compares the result with the delay variable, if the comparison is not fulfilled, this means the subtraction result is greater than delay value, the scoreboard will throw an error, saving the failure into the failures buffer and increasing the failureCount variable. When the comparison between actual and expected values fails, the scoreboard module also throws an error. This comparison is also done by a subtraction, whose result is compared with a pre-defined delta value. Remember that the variables are not inside the scoreboard module, it retrieves from and writes to these variables. They are placed at the top-level.
The failure trace contains some information about why the scoreboard throws a failure; it is composed by the following five attributes:

- The **output identifier**, which is a simple counter that is increased when the scoreboard makes a new checking.
- The **relative time** when the output was generated, whose value is obtained from the hardware verification platform; note that this time is not when the scoreboard module checks the correctness.
- The **actual** value, this is the output value produced by the DUT.
- The **expected** value is our golden reference vector.
- The **difference** between two consecutive data.

**Monitor** The monitor module creates the same tasks as its counterpart on the producer side, but here it spies the output DUT signals, annotating their relative time just when the DUT generates them. Another difference is based on how it
stores the relative time when an output is generated by the DUT. Now, it stores
two copies of this information; one used by the scoreboard module, while the
other one is used by the driver module. This double-copy is done because we use
FIFOs to store the information (see Figure 7.9).

**Driver** Like its counterpart, the driver module is a hardware object and it is a single
one in this side. This object provides some functions that retrieve information
about the verification process done by the scoreboard modules. Thus, it is the
main responsible to inform about the verification result.

![Figure 7.10: Overview of driver module (Consumer)](image-url)

Figure 7.10 shows an overview of driver modules following our hardware object
approach. The driver contains six methods, although it could be more. Like its
counterpart, it depends on the DUT interface and the number of signal groups
created by the developer.

- **expect** This method fills an internal buffer related to a specific output,
storing the golden reference values.
- **callCount** This method returns the number of writings done by the DUT
over a specific output. This is the number of outputs produced by the DUT.
- **timestamp** This method returns the relative times of each writing done by
the DUT over a specific output.
- **failureCount** This method returns the number of failures. These failures
have been found by a scoreboard, thus the method returns the failureCount
variable.
- **failure** This method returns the trace of a failure. Thus, the method pops
the traces stored into the failure buffer, which previously has been filled
by a scoreboard when it detected any incongruity.
- **intervalDelay** This method sets the maximum time allowed between two
consecutive output data. It is depicted in cycles.
In our case study we have only one output channel to spy. This means the environment has one scoreboard, one monitor and one internal output buffer. In addition, the driver is composed by the six methods explained above. Figure 7.11 illustrates the block diagram of the $l^2$-norm consumer side, including the relationship between the explained modules.

Figure 7.11: Block diagram of $l^2$-norm (Consumer side)

The VIP core is used for generating master AXI commands and write payload, generating slave AXI read payload and write responses and checking protocol compliance of AXI transactions. The AXI VIP is a SystemVerilog class library and uses similar naming and structures as the UVM for core design. The AXI VIP uses advanced verification techniques such as constrained randomization and transaction level modeling [Xil17b].

The VIP core facilitates the synthesis process of UVM environments, however building AXI transactions are complex (it is similar that UVM).

7.3 Writing Test Cases for Black-box Designs

Writing test cases for a black-box environment is different, because we do not stimulate the DUT directly. Firstly, we fill an input buffer connected to the DUT and set the testing parameters, such as delay. In addition, the verification tasks are done inside the hardware environment instead of in the test case.
Listing 7.3 depicts a black-box test case for our case study ($l^2$-norm). Like other test cases, the first step is to reset the dynamic area using the `TEST_RESET` macro and configuring the Test Manager through its macros. In streaming scenarios, we need to calculate the time from the first data read to the first data written, because streaming scenarios usually do not have a header. Besides, the clock enable is active-high during a specific time, in our test case it is 144 cycles (line 19).
Then, in order to prepare the testing environment, we must set the interval delay between two consecutive outputs, the golden reference values and the input stimuli. After we have set all these parameters, we must advise to the Test Manager that the testing environment is ready to exercise the DUT and check its results.

When the verification finishes, we are able to retrieve information about the testing process, like how much time takes the DUT to complete its task using the extended macro \textit{TEST\_ASSERT\_TIME\_XX} (line 27). We can also retrieve the number of readings and writings done by the DUT, using the \texttt{l2norm\_callCountInput} and \texttt{l2norm\_callCountOutput} functions respectively. Although in our test case we printed this information, it could be tested by an assertion such as \texttt{TEST\_ASSERT\_EQUAL}. Furthermore, the streaming testing environment allows to retrieve the number of failures that occurred during the testing process and their trace.

Finally, in order to ensure that the test returns the appropriate status, we must add an assertion checking the number of failures is zero, thus we add the last assertion in line 34.

### 7.4 Integration with our Verification Platform

Like other scenarios, hardware black-box designs are fully compatible with our verification platform, we only have to define which input plays the role of reading flag (\texttt{flagRD} signal) and which output plays the role of writing flag (\texttt{flagWR} signal), even if it is possible to define a group of signals to define these flags. In addition, the relative clock time and the clock enable that provide the \textit{chrono} component, must be connected in order to annotate the relative times when an input stimuli is read and when an output data is written. Besides, the clock enable is active-high during a period of time which matches with the number of cycles available that the DUT can run.

Figure 7.12 shows the elements that run on the ARM embedded processor of an FPGA and a test report. On the right side, the figure illustrates the elements that take place in our case study, using the \texttt{l2norm} algorithm as streaming unit processor. This part contains all \textit{virtual representations}, unlike other scenarios we do not have the DUT \textit{virtual representation}, we replaced it for a \textit{streaming testing} module which contains the remote functions that configure the black-box environment and retrieve information from it. On the left side, we show the report obtained after run the test case depicted in Listing 7.3. In addition, the test case prints a report that shows the testing process done, illustrating the number of readings and writings done by the DUT and the number of failures found and their traces. In this scenario, the test case did not find any errors.
In order to illustrate a bug detected by our black-box testing approach, we modify two values of our golden reference (ref array in Listing 7.3). Figure 7.13 shows the report after running the test case again with the new golden reference. The report prints the failure traces which provides information about the failures and how we can solve them.
In addition, the black-box testing environment shows the time of each reading and writing done by the DUT. This means we are able to build the scheduler illustrating the real throughput of our design and its latency. This information is shown in Figure 7.14.

```
<table>
<thead>
<tr>
<th>Time</th>
<th>RdTime</th>
<th>EmptyTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>14</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>17</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>20</td>
<td>23</td>
<td>23</td>
</tr>
<tr>
<td>23</td>
<td>26</td>
<td>26</td>
</tr>
<tr>
<td>26</td>
<td>29</td>
<td>29</td>
</tr>
<tr>
<td>29</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>32</td>
<td>35</td>
<td>35</td>
</tr>
<tr>
<td>35</td>
<td>38</td>
<td>38</td>
</tr>
<tr>
<td>38</td>
<td>41</td>
<td>41</td>
</tr>
<tr>
<td>41</td>
<td>44</td>
<td>44</td>
</tr>
<tr>
<td>44</td>
<td>47</td>
<td>47</td>
</tr>
<tr>
<td>47</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>50</td>
<td>53</td>
<td>53</td>
</tr>
<tr>
<td>53</td>
<td>56</td>
<td>56</td>
</tr>
<tr>
<td>56</td>
<td>59</td>
<td>59</td>
</tr>
<tr>
<td>59</td>
<td>62</td>
<td>62</td>
</tr>
<tr>
<td>62</td>
<td>65</td>
<td>65</td>
</tr>
<tr>
<td>65</td>
<td>68</td>
<td>68</td>
</tr>
<tr>
<td>68</td>
<td>71</td>
<td>71</td>
</tr>
<tr>
<td>71</td>
<td>74</td>
<td>74</td>
</tr>
<tr>
<td>74</td>
<td>77</td>
<td>77</td>
</tr>
<tr>
<td>77</td>
<td>80</td>
<td>80</td>
</tr>
<tr>
<td>80</td>
<td>83</td>
<td>83</td>
</tr>
<tr>
<td>83</td>
<td>86</td>
<td>86</td>
</tr>
<tr>
<td>86</td>
<td>89</td>
<td>89</td>
</tr>
<tr>
<td>89</td>
<td>92</td>
<td>92</td>
</tr>
<tr>
<td>92</td>
<td>95</td>
<td>95</td>
</tr>
<tr>
<td>95</td>
<td>98</td>
<td>98</td>
</tr>
<tr>
<td>98</td>
<td>101</td>
<td>101</td>
</tr>
<tr>
<td>101</td>
<td>104</td>
<td>104</td>
</tr>
<tr>
<td>104</td>
<td>107</td>
<td>107</td>
</tr>
<tr>
<td>107</td>
<td>110</td>
<td>110</td>
</tr>
<tr>
<td>110</td>
<td>113</td>
<td>113</td>
</tr>
<tr>
<td>113</td>
<td>116</td>
<td>116</td>
</tr>
<tr>
<td>116</td>
<td>119</td>
<td>119</td>
</tr>
<tr>
<td>119</td>
<td>122</td>
<td>122</td>
</tr>
<tr>
<td>122</td>
<td>125</td>
<td>125</td>
</tr>
<tr>
<td>125</td>
<td>128</td>
<td>128</td>
</tr>
<tr>
<td>128</td>
<td>131</td>
<td>131</td>
</tr>
<tr>
<td>131</td>
<td>134</td>
<td>134</td>
</tr>
<tr>
<td>134</td>
<td>137</td>
<td>137</td>
</tr>
<tr>
<td>137</td>
<td>140</td>
<td>140</td>
</tr>
<tr>
<td>140</td>
<td>143</td>
<td>143</td>
</tr>
<tr>
<td>143</td>
<td>146</td>
<td>146</td>
</tr>
<tr>
<td>146</td>
<td>149</td>
<td>149</td>
</tr>
<tr>
<td>149</td>
<td>152</td>
<td>152</td>
</tr>
<tr>
<td>152</td>
<td>155</td>
<td>155</td>
</tr>
</tbody>
</table>
```

Figure 7.14: Report of reading and writing times and the timing scheduler

### 7.5 Summary

In this chapter, we described how we can verify black-box designs using our hardware verification platform. In order to carry out a black-box verification, we introduced some components which look like UVM elements. In this way, we can ensure that the verification environment is based on UVM. The verification tasks are done in a pure hardware domain, checking the DUT correctness at run-time.

The stimulation process is performed following the virtual representation of a black-box environment. In addition, the testing environment can measure the time that the streaming process takes, using our extended testing framework.
Chapter 8

Hardware Testing Service

«I do not think there is any thrill that can go through the human heart like that felt by the inventor as he sees some creation of the brain unfolding to success... such emotions make a man forget food, sleep, friends, love, everything»

Nikola Tesla

8.1 Hardware Platform for Testing Service
8.2 Remote Testing
8.3 Testing Service
8.4 Summary

Although we have exhaustively explained our approach, it seems incomplete. The last piece to complete this dissertation, is a transparent testing service through our hardware testing platform where a DUT is deployed into a dynamically reconfigurable area. The testing process is transparently automated; an engineer commits his design code and unit tests written in a high-level language, such as C, into a repository, and automatically the testing service is able to synthesise the design code, deploy the DUT remotely into an FPGA and also exercise it with the original unit tests remotely, reporting the testing result to the engineer. This transparent, remote and efficient testing service entails new challenges.

• Building partial bitstreams from a high-level programming language.

• Deploying new DUTs or hardware objects remotely, being able to deploy partial bitstreams remotely.

• Exercising the DUT remotely by another actor/entity.

• Informing the engineer about the testing result.
At this point, our proposal requires human intervention to carry out an in-hardware verification. Tests must be compiled to generate the correct executable and run them in the ARM processor over the embedded operating system. It implies copy the executable into the FPGA and then run it, besides, the bitstream configuration must be deployed in the PL part. Figure 8.1 adds a new virtual representation of a hardware design for in-hardware domain. In this case, this representation allows engineers to exercise a DUT from their workstation, besides the c2hwobject only generates the virtual representation of a hardware design the adapter is the same. The following section explains how we can use an FPGA as a service in order to verify hardware designs depicted in a high-level programming language.

Figure 8.1: Same unit test at different verification stages (with remote version)

8.1 Hardware Platform for Testing Service

In order to serve a hardware verification service, we must modify the hardware platform to reduce the remote messages that a client of this service has to send. The biggest problem is when a process wants to send a partial bitstream remotely. This includes a number of remote messages that should be transferred by a network, which leads to network traffic. To carry out light network traffic, we compress partial bitstreams and extend our hardware verification platform to understand these kinds of partial bitstreams, moreover, we speed up the DPR deployment.

Figure 8.2 shows the block diagram of our hardware verification part related to the programmable logic part. There are two new hardware components inside our new hardware verification service that carry out the DPR process. The zipFactory object is the main responsible to physically deploy partial bitstreams, while the AXI Read
8.1.1 Bitstream structure

Firstly, we describe the Xilinx bitstream structure to understand the compression technique applied in this dissertation. The bitstream is divided into individual units called packets and there are two header-packet types: type1 and type2. Both packets are composed by a fixed number of bits (32 bits), which contains a small header at the beginning (the 3 first bits) to mark off the packet type and the operation code [Xil15] [Xil12b].

The type1 packet consists of a 32-bit word that is used for defining which register will be written to or read from. Only 20-bits are used, the rest is reserved for future use. These bits should be written as zeros. Figure 8.3 shows an example of this packet type, where the packet type is 001 and the operation code is a write (10 write, 01 read, 00 no operation and 11 not used). Then the register address comprises the next 13-bits, using the last 5-bits (00100) and writing the rest bits as zeros. Finally, the last 10-bits match with the number of data packets that must be processed.

![Figure 8.3: Header packet (type1)](image-url)
The type2 packet is used for writing long blocks and must follow a type1 packet. No address is presented here, because this field is defined in the type1 packet. Moreover, type2 packet uses 30-bits, while the rest bits (2-bits) are reserved for future use, thus the small header-packet is remained and the word count is increased from 11 to 27 bits. Figure 8.4 shows an example, the first 5-bits match with the small header-packet like type1 packet: the first 3-bits of this 5-bits denote the packet type (010), while the next 2-bits are the reserved bits on this packet type. The rest bits are the number of data packets that must be processed.

![Figure 8.4: Header packet (type2)](image.png)

Following a header packet we find data packets. Some of them are pre-defined in accordance with the register that will be written to or read from. For example, a 32-bit word whose content is 0...0101, means that a start-up sequence begins. When the bitstream data contains unused resources, data packets are represented as 32-bit words filled with zeros.

The Internal Configuration Access Port (ICAP) is the internal interface provided by Xilinx for its Virtex and Zynq devices, and it is hardwired in the FPGA logic which interprets the above packets. The ICAP provides read and write access to the FPGA configuration memory, with separated data ports. ICAP data ports can be used as 8, 16 or 32 bit wide, which is configurable through a bus width auto detection packet. The theoretical bandwidth of the ICAP is 400 MB/s with a maximum reconfiguration clock of 100MHz [Xil12b].

The smallest group of configuration bits in Zynq devices is composed by 101 32-bit words and is it known as a frame. The number of frames in a bitstream file depends on the size and type of resources inside a reconfigurable region. The Xilinx bitstream contains commands for FPGA configuration logic, as well as configuration data. It is divided into four sections: bus width auto detection, sync word, configuration data and desync word. The first section, bus width auto detection, is done through two 32-bit words: 0x00000BB and 0x11220044. The configuration logic only checks the low eight bits of the second word. Depending on the received byte sequence, the configuration logic can automatically switch to the appropriate bus width. For example, when the configuration logic finds 0xBB, followed by 0xBB, the bus width is configured as x8. When the configuration logic finds 0xBB, followed by 0x44, the bus width is configured as x32. If the configuration logic does not find 0xBB, it will disregard the rest of the bitstream.
A special sync word (0xAA995566) is used for allowing the configuration logic to align at a 32-bit word boundary. Any packet before the sync word will not be processed by the configuration logic. After the sync word, the configuration logic processes each 32-bit word as a packet, as was explained at the beginning of this section. Finally, a special packet is sent to the configuration logic to desynchronize it. The header packet is 0x30008001, while the data packet is 0x0000000D.

Bitstream compression

Generally, a dynamic design has several dynamic areas, whose sizes depend on the modules that will be deployed into them. At this point, there are two options; fitting the dynamic areas for an ad-hoc system or building a grid of dynamic areas with different sizes, independently of the modules that will be deployed. Unfortunately, the first option depends on the first design or reference design whereas the second option generates partial bitstream data composed by many zero data packets when the design is too small, due to unused resources. Therefore, the large number of zeros allows applying compression techniques, getting a decent compression ratio.

The FPGA vendor provides a constraint that is not enabled by default in its write_bitstream tool, called BITSTREAM.GENERAL.COMPRESS. This constraint minimises the size of partial bitstream data, clearing unneeded used logic, writing a large number of packets that contain only zeros. Besides, the manufacturer provides an option in the same tool called reference_bitfile, which builds a new bitstream file from a reference bitstream file. This new partial bitstream contains only differences from the reference file and can be used for incremental programming. Take for example, a system that has three dynamic modules: A, B and C. The A module is the reference bitstream to build the other ones, so the reconfiguration from B to C is not allowed, unless a new bitstream will be written with B module as reference. To cover all possible scenarios in this example, seven partial bitstream files are necessary (Equation 8.1, NDA is the number of dynamic areas). Moreover, a scheduler must be implemented to know which module is running on the device and to decide which partial bitstream must be deployed for new environment requirements.

\[ NDA^{NDA} - (NDA - 1) \]  

(8.1)

Although the BITSTREAM.GENERAL.COMPRESS and reference_bitfile options are very useful, the number of bitstream files and their size are very costly in terms of memory. In addition, the manufacturer tool includes some packets before the bus width auto detection packet and after the desync word, which are ignored by the configuration logic. Thus, the header and tail could be reduced, removing ignored packets, and the sequential packets filled with zeros can be replaced by a new header packet following its data packet.
Figure 8.5: a) Xilinx sequence b) Proposal sequence

Figure 8.5 shows how these improvements are done. The packets before the bus width auto detection packet are removed (red) and a small header is written with the bus width auto detection packet and the sync word (yellow and green), this reduction can also be generated by Xilinx tools, using its raw bitstream file. And then the sequential packets that contain more than two packets filled with zeros are identified and replaced by a new data packet and its header. In order to follow the Xilinx bitstream structure, the header packet is the 32-bit word 0x30036001, whose packet type matches with 001 and the operation code is a write (10). The register address does not exist, because we really do not write any register, whereas the word count is one, so we must only process one data packet. Following the header packet, the data packet designates the number of data packets that were replaced. For instance, if an original bitstream file contains a sequence of 25 zero packets, the data packet will be 0x00000019 (orange). If the number of zero sequential packets is less than three, they will not replace them because the effort is not worth it. Finally, the packets after the desync word are removed (blue) and a desync word is written (green). This process is performed by a bitstream compression tool called write_factory_bitstream.

8.1.2 zipFactory object: The deployment engine

Dynamic reconfiguration management is typically driven by software, using an embedded processor to transfer bitstream data to the device logic. We present an approach that relies this task on a specialised hardware object: the deployment engine (zipFactory). The zipFactory object is the configuration logic mentioned above, and it is based on a software design pattern: the abstract factory pattern [GHJV94]. This object offers a set of reconfiguration related services through a simplified interface connected to an AXI bus, it is for instance able to deploy a new bitstream that is stored in an external device.
8. Hardware Testing Service

From an architectural point of view, the zipFactory object is composed by different modules as shown in Figure 8.6. The hardware object works with three different interfaces which allow communicating the hardware object with third-party entities:

- **AXI interface**: This interface allows to dispatch deployment requests and informs about the process status. This interface uses an axi2fifo driver to translate AXI messages (see section 3.2.4).

- **HLS-Stream**: Usually, bitstream data is stored in the DDR memory device and several memory addresses locate each bitstream position in the DDR. The object can get the data from the memory through a third-party component (see Section 8.1.3), denoting a memory address and a size in words to read. A word-flag indicates the end of reading (endWord). The third-party component is a memory controller core connected via HLS-Stream, in accordance with the FIFO handshake from Vivado. The memory controller core translates the memory address, size and endWord parameters to a formal AXI request and sends it by an AXI High Performance interface, in this way it gets bitstream data through burst transfers. Bitstream data is sent to the zipFactory core via the AXI-Stream.

- **AXI-Stream**: This streaming interface receives bitstream data from a storage device through a memory controller core (DMA or similar). The AXI-Stream bus width is 32 bits.

According the architectural point of view, the object functionality is divided into two big modules. The administration part of the zipFactory object is described from C programming language and is subsequently translated into RTL by the Vivado.
HLS tool, in accordance with our hardware object approach. The other part has been developed in VHDL and carries out the physical deployment process of compressed partial bitstreams.

On the other hand, from the behavioural point of view, the admin module is composed by two functions: newObject and status. With both methods partial bitstreams can be deployed without microprocessor intervention. The description of both methods are explained in the following lines.

void newObject(unsigned int addr) This function starts a new physical deployment of a new functionality; a new DUT is deployed into our dynamic area. In order to carry out this task, the function needs to go through the following steps.

1 - Firstly, a third-party component sends a message request, whose format matches with our communication protocol. This request is received through the AXI interface and it is translated into the FIFO and then dispatched by the admin module, getting the argument functions, address argument in this case. In addition, the internal status variable is set to the reconfiguration state; zipFactory resets the variable. Then the zipFactory object must inform the third-party actor with a reply message, which initiates the communication, that it received a request and it is dispatched.

2 - The address argument matches with a bitstream location, which agrees with a memory address (32-bit word). In our hardware verification platform, partial bitstream files are stored in the DDR, thus the component is connected with the DDR to get specific bitstream data. Moreover, an intermediary (the memory controller core) is the main responsible for translating from an address parameter into an AXI request. Both components, the memory controller and the zipFactory object, are connected through an HLS-Stream interface, whose management is done by the adminPR module.

3 - When the memory controller core reads the partial bitstream data from DDR, it is sent by an AXI-Stream interface to the zipFactory object. The memory controller core stops reading when the desync word is found: 0x30008001 as header packet, following 0x0000000D as data packet. All data received through this interface is stored into a FIFO by the axis2fifo bridge module.

4 - After the data buffer contains an amount of 32-bit words, the deployment starts. The adminPR module sends a flag to start the deployment, whereas the unzip module decompresses the new packet at run-time. The buffer contains a small portion of a compressed partial bitstream, reducing the resources used due to its small depth.

5 - Then the pr module sends the data given by the unzip module to the internal FPGA configuration logic, the ICAP, which is the main responsible for physical deployment. The ICAP bandwidth is 32-bits, which matches with packet word sizes. The pr module knows which packet type will be sent to the ICAP at every
moment (a header or data packet), thus when the command matches with the desync word, the pr module sets a flag, announcing that the deployment process has just finished. After the desync word, the pr module sends a few NOP words to flush the command pipeline properly.

6 - Finally, the adminPR sets the internal status variable as done (1) and this value is sent to third-party components when a status request message takes place.

int status() This function returns the reconfiguration process status. It checks the internal variable status and returns its value. It will return 1 value when the zipFactory object finishes deploying a new DUT or when it is in its IDLE state, otherwise it returns 2, which means that it is deploying a new partial bitstream.

Following the same line as other hardware objects, we build a virtual representation of the zipFactory object to grant its utilisation from software artefacts, such as programs running on the ARM processor. This virtual representation looks like the Test Manager object, aside of the function name, the main difference is the communication message corresponding to each function. Listing 8.6 depicts its virtual representation.

Results of zipFactory object

Table 8.1 shows the resource utilisation of the zipFactory object, including a comparison with other approaches. It achieves a configuration rate of about 387.59MB/s, very closer to the highest possible (400MB/s) [Xil12b]. We do not get the highest rate because of the overhead in filling the buffer, we remain that the bitstream data is stored in the DDR device and we have to transfer from it to the AXI-zipFactory through the memory controller component. Table 8.1 shows that our proposal reduces the resource utilisation and gets a high configuration speed rate. The throughput results are in accordance with the maximum ICAP clock frequency (100MHz), thus we compare our proposal with other controllers containing the original ICAP, discarding the overclocked approaches.

Most compression algorithms exploit statistical characteristics such as patterns found in the input data, independently of their location. In dynamic partial projects, only a subset of logic is used to implement a specific task (user logic), which could be encoded inside a dynamic area. As work [KBT07] highlights: if we want to build a suitable benchmark for investigating bitstream compression algorithms, we have to generate bitstreams that represent a wide range of different FPGA designs. In addition, dynamic area definition is directly matched with the compression ratio, although unused resources produce significant overheads, we consider a big dynamic area to offer users enough resources to deploy their designs. Remember that our dynamic area contains 6120 LUTs, 12240 FlipFlops, 32 DSPs and 16 BRAMs, and it is located between the slices X14Y3 and X47Y47.
In order to check our proposal and to detect the compression ratio achieved, we selected some benchmarks from work [KBT07] and included new ones from different application domains, including examples generated by an HLS tool, specifically the Vivado HLS tool from Xilinx [Xil13c]:

- In *hello world* applications, we chose a simple counter. This kind of application uses few resources, allowing to observe the behaviour of compression algorithms when there are a lot of unused resources.

- In *cryptography* applications, we chose a *Data Encryption Standard* (DES) module from *OpenCores* [Ope]. This module mostly contains bit shuffling operations and shift registers.

- In *communication* applications, we chose an *Address Resolution Protocol* (ARP) response module [Ope], which consists of a huge amount of communicating state machines and Boolean functions.

- In *signal processing* applications, we chose a *Finite Impulse Response* (FIR) filter and *Discrete Cosine Transform* (DCT) module [Xil13c]. These modules consist of several multiply and accumulate operations. Both modules are generated from a C code reference by the Vivado HLS tool.

- In *normalisation* applications, we chose the $l^2$-norm and $l^2$-hys algorithms [DT05]. Both algorithms consist of multiply-accumulate operations, using `sqrt` and `recip` functions. In addition, the $l^2$-hys algorithm was implemented in two different ways: by VHDL language and C language. The C version is translated to RTL code by the Vivado HLS tool.

Table 8.2 shows resource utilisation of each example. All benchmarks are able to be placed into the dynamic area that we defined previously. The C version of $l^2$-hys
algorithm is the only benchmark that uses all available Digital Signal Processors (DSPs) and more than 50% of Look-Up Table (LUT).

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>FF</th>
<th>LUT</th>
<th>BRAM</th>
<th>DSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>counter</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>DES</td>
<td>574</td>
<td>431</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ARP</td>
<td>143</td>
<td>107</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>FIR</td>
<td>217</td>
<td>234</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>DCT</td>
<td>182</td>
<td>185</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>$l^2$-norm</td>
<td>1737</td>
<td>1516</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>$l^2$-hys</td>
<td>1524</td>
<td>1538</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>$l^2$-hys</td>
<td>3959</td>
<td>3726</td>
<td>0</td>
<td>16</td>
</tr>
</tbody>
</table>

*HLS implementation

Table 8.2: Benchmark resource utilisation

Figure 8.7 lists the compression ratios for all benchmarks and some compression algorithms. The first chart shows the compression ratios achieved with gzip (enabling -9 option), RLE, Huffman [Huf52], LZ77 [ZL77] algorithms and our proposal. The results for our technique reveal good compression ratios when there are unused resources, our technique is even better than other algorithms in some benchmarks. Unfortunately, when the resources utilisation is higher, our proposal does not get good compression ratios. The second chart shows the compression ratios blended by our proposal with one of the compression algorithms listed before. The results reveal important optimisation in some algorithms, reducing about 10% of the results obtained in the first chart. In the third chart this optimisation looks better, the best option is our proposal blended with the Huffman algorithm, achieving better reduction ratios than the LZ77 algorithm. Unfortunately, applying our technique with the RLE algorithm does not obtain good results.

<table>
<thead>
<tr>
<th>Case study</th>
<th>Bitstream Size (Kb)</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unit testing</td>
<td>154</td>
<td>66.8%</td>
</tr>
<tr>
<td>Asserts</td>
<td>156</td>
<td>66.4%</td>
</tr>
<tr>
<td>Mocks</td>
<td>120</td>
<td>74.1%</td>
</tr>
<tr>
<td>Streaming</td>
<td>205</td>
<td>55.8%</td>
</tr>
</tbody>
</table>

Table 8.3: Bitstream size and compression ratio of $l^2$-norm

In our hardware verification platform, the tool generates partial bitstreams whose sizes are about 464Kb., independently of the functionality. The $l^2$-norm algorithm generates a partial bitstream with that size. Applying our compression technique
to the four case studies explained in this dissertation, we get the following sizes and compression ratios Table 8.3.

![Figure 8.7: Achieved compression ratios for benchmarks in Table 8.2](image)

### 8.1.3 Memory controller: AXI Read Memory

The memory controller core is really a smart DMA, whose main task is transferring bitstream data from the DDR memory to the zipFactory object. Figure 8.8 shows an overview of our smart DMA called *AXI Read Memory*. The memory controller core works with three interfaces that allows us to read data from the DDR device, bridging between DDR and an *AXI-Stream* bus. Its management is driven by the pure hardware domain without processor intervention.

- **HLS-Stream.** This interface connects with the zipFactory HLS-Stream interface (*admin controller*), whose signals are *address*, *size* and *endWord* signals. The
Figure 8.8: Overview of AXI Read Memory component

address signal denotes the memory address where the partial bitstream is stored, indicating its starting memory address. To stop the reading, we have two options: using the size signal or endWord signal. The first one denotes the bitstream size in 32-bit words, while the second signal denotes a 64-bit word that the AXI Read Memory component must detect to stop the transfer. In our hardware verification platform, we use the second option whose 64-bit word is the desync word (0x300080010000000D).

- HLS-High Performance. The memory access is done through an AXI High Performance bus, which allows efficient access and data transferring. The signals received by the above interface are translated to a number of AXI requests, automatically increasing the memory address. Bitstream data is received by burst transfers whose data width is 64-bit words, storing them into an internal buffer that contains the AXI Read Memory. The core only builds another AXI request whenever it will be able to store all transaction data into its internal buffer without any overflow.

- AXI-Stream. Finally, an AXI-Stream interface offers third-party components the data retrieved from the DDR and now it is stored in the internal buffer. The width of this interface is 32 bits, thus the internal buffer must transform from 64-bit words to 32-bit words. This interface is ready when the buffer threshold is overtaken. The threshold can be configured by the designer at the design time and it is depicted in a number of 32-bit words that the component must wait for, before announcing that it has data that can be consumed. For instance, we can look at a scenario in which the threshold is 32 words and it only has 23 words stored into the internal buffer, the AXI-Stream interface does not active-high its valid signal, since the buffer contains at least 32 words.

Form the point of view of resource utilisation, the AXI Read Memory com-
ponent uses 318 FlipFlops, 550 LUT and 1 BRAM, while the smaller configuration of Xilinx DMA component uses 1722 LUT and 3824 FlipFlops [Xil13a]. Note that our proposal only reads while the Xilinx approach does both operations: readings and writings.

8.2 Remote Testing

In order to achieve a hardware testing service, we must be able to grant a remote access to our hardware verification platform. This creates new challenge: all our hardware services must be accessed by a network. We can find some hardware services in our proposal right now; the service to physically deploy a new DUT is done by the zipFactory object; the DUT timing measurement is driven by the Test Manager object, and the DUT itself, which enables to exercise it. All these services must be translated into remote services that third-party entities can use them, instead of executing them locally on an ARM processor.

In order to grant the remote access, we propose a new communication layer based on ZeroC Ice middleware [Zer17] using an extension made by the ARCO Research Group [Ace17], known as IceC. On the other hand, we need remote testing frameworks to use our hardware verification platform remotely.

8.2.1 Granting Remote Access

Figure 8.9 shows an example of our remote service between two entities. Both entities are connected via Ethernet and they build a heterogeneous infrastructure. On the left side, we find an entity that plays the client role in a distributed system domain. It could be a developer workstation whose architecture is an amd64 processor and it runs a Debian distribution system for this example, but it could be another entity as well. In this new scenario, the entity runs the test cases using a remote testing framework described from any programming language compatible with this architecture, such as Python [Pyt17] (see Section 8.2.2). In addition, this part carries out other tasks, for instance, it sends the partial bitstream to the FPGA and requests its deployment.

On the right side, we find the FPGA device that plays the role of a server in a distributed system domain, this part offers some type of service. The figure only shows its programmable logic part, whose ARM architecture runs a Linaro distribution. In this part, we deploy a new service, called the testing service, which enables remote access. This module offers three services related to deploying a new partial bitstream (DPR Service), sending a block of data (Transfer Service) and sending a message to
the *programmable logic* part (*GCommand Service*). All these services are executed by a client using a virtual representation of our testing service, following the RMI approach. The services can be accessed by three different endpoints, as Figure 8.9 illustrates.

### Server side

The server side has to offer some services to a third-party, however it must depict these services to distribute them easily. Then clients use it to invoke the offered services remotely. ZeroC Ice middleware proposes an *Interface Description Language* (IDL) file to describe the offered services. In our example, we depict the three services of our testing service module into an IDL file, as Listing 8.1 illustrates.

The *Internet Communications Engine* (Ice) is an object-oriented RPC framework that helps you build distributed applications with minimal effort. Ice allows you to focus your efforts on your application logic, and it takes care of all interactions with low-level network programming interfaces. With Ice, there is no need to worry about details such as opening network connections, serializing and deserializing data for network transmission, or retrying failed connection attempts [Zer17].
8.2. Remote Testing

Although Listing 8.1 depicts the interface of our testing service module, someone could ask himself how I offer these three services (transfer a file, deploy a bitstream and send a message to programmable logic part)? The following lines describes precisely each service from a server point of view.

**Transfer Service** This service covers the tasks related to receiving a file from a client. The file is divided into several chunks to send them through the network. The data that the FPGA receives is first stored into a file, concretely in the `/tmp/config.bit` file, and then is transferred to the DDR if the file transmission does not contains any bug. Our service only receives partial bitstream files, although this service also allows other kind of files. If we want to change the file location, we must send the path too, replacing the `startTransfer` function or adding a new one. The data will be copied into DDR, starting at the memory address that was depicted.

**void startTransfer(int fsize, int addr)** This function sets the size of the file that it will be sent and the hardware direction where it will be stored. This hardware direction must be inside the DDR address range. The function resets an internal counter which counts the current bytes that have been received. Therefore, the first task for sending a file is setting the file size and its location. If the file exists, it will be removed (see Listing 8.2).
void sendBlock(DataBlock block)  This function receives a block of bytes which matches with a chunk of a file that the client is sending. It is concatenated to the previous blocks inside the /tmp/config.bit file, or if it is the first block received, the function will create a new file. Moreover, this function updates the internal counter, adding the size in bytes of the block received (see Listing 8.3).

int endTransfer()  Finally, in order to end up with the file transmission, it is necessary to inform that the client has sent all file chunks, this is done after the last file block sent. It returns an integer value to inform about the file transmission process, thus the client is able to know if there was any error when he sent the file (see Listing 8.4). After all chunks are received and the transmission is done successfully, the /tmp/config.bit file is copied into DDR, stating from the hardware address that is set in targetAddress variable. copy_to_mem() function overcomes the transferring process.
8.2. Remote Testing

DPR Service This service enables a partial reconfiguration process for third-party artefacts. It provides two options for deploying new functionality, DUTs in our case: using the internal Processor Configuration Access Port (PCAP) or using our zipFactory object. Previously, the client had to send a partial bitstream using the above service.

void pcapReconfig() This function deploys a new functionality via PCAP using an internal script which configures an internal register, indicating that we are going to deploy a partial bitstream. The script sends the partial bitstream via PCAP (see Listing 8.5). Remember that the partial bitstream is stored in the /tmp/config.bit file, and we need it when we use this option. The body of the pcapReconfig function is not shown because it only calls upon this script.

Listing 8.5: Script to deploy bitstreams via PCAP

```bash
1  echo 1 > /sys/devices/soc0/amba/
2       f8007000.devcfg/is_partial_bitstream
3  cat /tmp/config.bit > /dev/xdevcfg
```

void zipFReconfig(int address) This function allows to deploy a partial bitstream using our zipFactory object. Remember that this hardware object deploys a partial bitstream stored in DDR, thus the function needs a memory address to carry out its task. This kind of reconfiguration is more complex because we must fulfil the communication mechanism of hardware objects. Listing 8.6 shows the C code for carrying out a physical deployment. The first lines obtain a pointer related to the zipFactory object, thus we can read and write it to the hardware address where it is mapped. Lines 13 to 16 build the request message to deploy a partial bitstream, while the 21 line checks that the zipFactory object receives the request. After the reconfiguration process begins, we ask the zipFactory object about its status. This task is done by lines 23 to 31, which make a polling until the status is done. Then we free the pointer.

GCommand Service This service contains a unique function (DataBlock remoteExec(int address, DataBlock dataInput)) that permits to write blocks of data to a specific hardware memory address and read blocks of data from a hardware memory address. It allows us to send and receive a sequence of bytes, which fit with our communication mechanism because all request messages have a reply message, even in void functions. Remember that our communication mechanism follows the request-response model and all messages are translated into a sequence of bytes.
Listing 8.6: Body of `zipFReconfig` function

```c
void *ptr;
unsigned page_size=sysconf(_SC_PAGESIZE);

int fd = open("/dev/mem", O_RDWR);
if(fd < 1) {
    printf("[FAIL] Cannot open /dev/mem for writing\n");
    return;
}

ptr = mmap(NULL, page_size, PROT_READ|PROT_WRITE,
    MAP_SHARED, fd, FACTORY_HW_ADDR);

*((unsigned *)ptr) = FACTORY_RECONFIGURE_METHOD;
*((unsigned *)ptr) = FACTORY_CALLBACK;
*((unsigned *)ptr) = address;
*((unsigned *)ptr) = FACTORY_AREA;

int head = *((unsigned *)ptr);

printf("[INIT] Partial Reconfiguration\n");

int status = 2;
while(status == 2){
    *((unsigned *)ptr) = FACTORY_STATUS_METHOD;
    *((unsigned *)ptr) = 0xCA110000;
    head = *((unsigned *)ptr);
    status = *((unsigned *)ptr);
    printf("[INFO] Status: %d\n", status);
    sleep(1);
}

printf("[DONE] Partial Reconfiguration\n");
munmap(ptr, page_size);
close(fd);
```

Listing 8.7 illustrates part of the `remoteExec` function. It does not show the beginning because it is equal to any virtual representation of a hardware object, so lines 1 to 12 of Listing 8.6 matches with the same lines of this function. To forward a request message from a client to a hardware object, or DUT, is easy, we only have to build a pointer to the input sequence of bytes and send it (lines 3 to 6). However, the reply message is more complex due to our dynamic header, some
messages contain a 32-bit header while others contain a 64-bit header. Thus, we read a 32-bit word (line 8), this word gives information about the rest of the message, if it has payload (flag active-high) the message is composed by more 32-bit words. When a message contains payload, we must read another 32-bit word (line 13) which informs about the payload size. Now, the function is able to read all payload generated by the DUT.

```
// Lines 1 to 12 of other object
....

unsigned int *ptrItems = (unsigned int*) input.items;

for (itWR=0; itWR != input.size/sizeof(int); itWR++)
    *((unsigned *)ptr) = *ptrItems++;

head1 = *((unsigned *)ptr);
dout.size=4;

rdSize = 0;
if (((head1 & 0xFF) & FLAG_HAS_PAYLOAD)){
    head2 = *((unsigned *)ptr);
dout.size+=4;
    rdSize = (head2 & 0xFFFF);
}

dout.size+=rdSize*sizeof(int);
unsigned char dout_i[dout.size];
unsigned int *ptrOut = (unsigned int*)dout_i;

*ptrOut++ = head1;
if (dout.size-rdSize*sizeof(int) == 8)
    *ptrOut++ = head2;

for (itRD=0; itRD != rdSize; itRD++)
    *ptrOut++ = *((unsigned *)ptr);
dout.items = dout_i;
munmap(ptr, page_size);
close(fd);
return dout;
```
Client side

At this point the right side of Figure 8.9 is completely described, however we have not explained how a client is able to use it yet; this is the left side. To use all services provided by the testing service we have to use a virtual representation of it. This representation can be described from any programming language compatible with the architecture that will be run. In our case, we use Python, due to bring some goodness, such as higher abstraction level or a number of libraries that increase designer productivity. Using Python with our ZeroC Ice middleware for building distributed systems reduces its complexity. For instance, the virtual representation is the interface description file that Listing 8.1 depicts along other easy operations.

Listing 8.8 illustrates an example that includes communication tasks. These tasks are common or very similar for building clients that use a remote service. Firstly, before using any service, the interface description file must be loaded and the service must be imported. Then, the communicator object is created. This is the communication core. In order to obtain a virtual representation of each service offered by our remote testing service, we use some functions of our middleware. Now, we are able to use the services. Note that in our example we only use the DPR and Transfer service.

```python
1 Ice.loadSlice("../ice/testingService.ice")
2 import TestingService
3
4 class Client(Ice.Application):
5   def run(self, args):
6     ic = self.communicator()
7
8     vrTransfer = ic.stringToProxy(
9         'Transfer -e 1.0 -t:tcp -h zynq-kilby.uclm.es -p 7891')
10    vrTransfer = TestingService.TransferPrx.uncheckedCast(vrTransfer)
11
12    vrDPR = ic.stringToProxy(
13        'DPR -e 1.0 -t:tcp -h zynq-kilby.uclm.es -p 7891')
14    vrDPR = TestingService.DPRPrx.uncheckedCast(vrDPR)
```

Now the question is: how do you use these services? We propose the next scenario to answer this question. Imagine you have a bitstream file called pb.bit, which really is a partial bitstream and you want to deploy it into an FPGA. You can use the above services to carry out this task. The Transfer service helps you to transfer the partial bitstream to the FPGA as shown in Listing 8.9. Firstly, the pb.bit file is opened to know the size and send it, using the startTransfer function. Then, the file is sent in chunks of 1024 bytes (block_size value), using the sendBlock function.
service must be informed when the client finishes the file transmission, this is done by
the \textit{endTransfer} function which checks that it receives all bytes that you denoted at
the beginning of the transfer, replying its status; 1 if the process is right, otherwise 2.
Finally, the last step is to deploy physically the partial bitstream that you have just
sent, you call upon only the \textit{zipReconfig} function and wait to finish the process.

\begin{lstlisting}[language=Python]
... doneTransfer = 2
while doneTransfer == 2:
  f = open('pb.bit', 'r')
  file_data = f.read()
  file_size = len(file_data)
  vrTransfer.startTransfer(file_size)

device = 'vp9.8.11'
block_size=1024
iterations = file_size / block_size
for i in range (0, iterations):
  vrTransfer.sendBlock(file_data[i*block_size: (i+1)*block_size])

vrTransfer.sendBlock(file_data[iterations*block_size:])
doneTransfer = vrTransfer.endTransfer()
if doneTransfer == 2:
  print ("[FAIL] File Transfer")
print ("[DONE] File Transfer")
vrDPR.zipFReconfig()
\end{lstlisting}

Figure 8.10 shows the output generated by the testing service when it receives
a bitstream file through its \textit{Transfer} service and a request to deploy it through its
\textit{DPR} service. Firstly, the report shows the three endpoints related to the three offered
services. Then it prints the file size that is going to be transferred and deletes an older
\textit{config.bit} file. After this process, the service starts receiving data blocks of 1024
bytes, except the last block. Finally, to complete the file transmission the \textit{Transfer}
service receives a request from the client and checks that the sum of all size data blocks
matches with the initial value and prints a message depicting its correctness. Then the
reconfiguration process starts, denoting the hardware memory address where the partial
bitstream is stored. It automatically asks the \textit{zipFactory} object about its status until it
finishes the reconfiguration process. After the partial bitstream deployment, the service
is ready to receive remote commands that are forwarded to the AXI bus.
8.2.2 Remote Testing Frameworks

Now the hardware verification platform can be accessed remotely, it means that any client can use it, moreover, the unit tests are decoupling the target device. Therefore, the test cases do not run in the target device, they can be executed in other architecture, such as `amd64`. This means that we need remote testing frameworks to exercise the remote DUT without missing the timing extension described in chapter 4. In order to get remote unit tests we use the virtual representation of testing service, specifically the `GCommand` service. This service works between the test cases and the DUT, forwarding the messages from a pure software domain to the hardware domain.

Therefore, we can use a programming language compatible with the architecture that runs the test cases. In our case, we use Python Unit Testing framework to exercise the DUT. This entails a new challenge: the testing framework must be extended using `GCommand` service to send remote commands.

The tests cases did not change, they are the same that were used in the Unity testing framework. Thus, the test case for the `scale` function is translated into Python programming language as Listing 8.10 shows.
8.2. Remote Testing

Our Python tests cases imports a GCommand client, called FPGA_hwtClient, that enables the communication between different domains. It allows us to send a sequence of bytes to the DUT and receive another sequence of bytes from it. The arguments function of FPGA_hwtClient sets the hardware memory address, the DUT memory address, and the input sequence of bytes, but it does not send the sequence. This task is driven by the run function, it gets the reply message, whose result is returned by the result function. Listing 8.11 illustrates the GCommand client.

Now the unit testing framework can be extended using the above client to include a timing analysis inside test cases. In addition, this client is used in the virtual representation of the DUT. Remember that the test case does not call upon the DUT's
function directly, it calls upon a virtual function that forwards the message to the real function. The structure of both cases is similar. Listing 8.12 depicts the scale virtual function.

### Listing 8.12: scale virtual function in Python

```python
def scale(sum):
    din = []
    din.extend(int_to_byte(0x00010204))
    din.extend(int_to_byte(0x00000001))
    din.extend(int_to_byte(float_to_ieee754(sum)))
    testCli = FPGA_hwtClient()
    testCli.arguments(0x42000000, din)
    testCli.main([None])
    dout = testCli.result()
    idout = charSeq_to_intSeq(dout)
    head1 = idout[0]
    head2 = idout[1]
    _ret = ieee754_to_float(idout[2])
    del testCli
    del idout
    return _ret
```

First the scale function translates all data to a sequence of bytes. This means our communication header is translated too. This process is done by the int_to_byte function which converts an integer to a byte sequence (lines 3-5). However, the argument of the scale function is a single-precision floating-point, it entails a new transformation; from floating-point to IEEE-754 standard before translating it into a sequence of bytes. All conversion results are concatenated into an array of bytes which will be sent to the DUT using the GCommand client (line 8). After the input stimuli are ready, the scale function initiates the communication with the testing service located in a remote FPGA. Then, it retrieves the result through the result function (line 10). Finally, it translates the sequence of bytes to the correct user-defined type again. In the scale function the return value is another single-precision floating-point, thus the single payload value is converted into this type using the ieee754_to_float function (line 15).
8.3 Testing Service

At this point, we have described how someone can carry out a remote testing using unit tests in a heterogeneous environment, where different actors interact with each other. Now, this section depicts the final twist of the screw of this Ph.D. dissertation. This section introduces new actors to facilitate the verification process, providing a transparent, remote and efficient hardware testing service. Figure 8.11 illustrates all actors that take part in the in-hardware verification process.

The previous section describes the remote verification process between a client and a remote FPGA. In that case the client, the developer, must manually manage all tasks, from building the design to deploying the partial bitstream and exercising it. The verification process tries to reduce the engineer’s tasks until he only writes his design in C programming language and updates it into a remote repository, such as GitHub [Inc17]. The rest of the tasks is done automatically by other entities.

*GitHub* is a service based on *Git*. *Git* is an open-source version control system that was started by Linus Trovalds. *Git* allows developers to store different versions of their code. This allows developers to easily collaborate, as they can download a new version of the software, make changes, and upload the newest revision. Every developer can see these new changes, download them, and contribute. The location where all files of a particular project are stored is a *repository*. 
Therefore, a developer depicts his design in a synthesizable programming language, such as C, or he can depict his design in any HDL fulfilling our dynamic area entry point. Automating all tasks entails a good file system organisation, which is described into a bash script file and contain all steps to get the partial bitstream, deploy it and exercise the design. Thus, we provide a template of this script in order to facilitate this task (see Listing 8.13). At the beginning of this template, we must describe where the hardware source files are located, the location of the tests and which type of verification it is; unit testing (UT), asserts (ASSERT), mocks (MOCKS) or streaming (STREAM). This template can be modified according to the developer needs, but the core needs to be saved to create these three tasks. For instance, if the developer build his design using an HDL such as VHDL, lines 14 to 16 will be removed, because he depicts that all files are located in the HW_SOURCES variable. The type of verification denotes which wrapper must be applied, thus if the developer’s design fulfils the dynamic area interface, the TEST_TYPE variable must set to NONE value. Finally, the developer must depict how test cases are executed and their location, we propose to use a Python library called nose [Nos17], this extends unittest framework, making testing easier.

Listing 8.13: Verification-tasks description file

```
#!/bin/bash

HW_SOURCES=src/hw_src
TEST_DIR=tests
TEST_TYPE=UT

echo "[INFO] Setup testing environment"
source /opt/Xilinx/Vivado/2015.4/settings64.sh
cp -r /opt/hw_testing/platform/ .
mkdir logs output

echo "[INFO] Getting HW sources"
cd src; make; cd -

echo "[INFO] Synthesis design - Partial bitstream"
./hwt_genTCLDesignConfiguration.py $TEST_TYPE $HW_SOURCES platform
cd platform; make buildPartial; cd -

echo "[INFO] Send partial bitstream to FPGA"
./remoteDPR.py output/partial.bit

echo "[INFO] Running tests"
cd $TEST_DIR
make tests
```
When the developer defines his file system structure and how the three tasks
are carried out, he must upload his design into the GitHub repository. The script
file must be called run.sh and its location must be the root project directory. This
repository is offered by an external service and it has been configured to throw a hook
when all entities that are subscribed are changed. An example of such a change is a
commit performed by a developer for a new release of his design. This hook wakes up
a new actor, a Jenkins server [Jen17]. Jenkins helps to automate the non-human part
of the development process. Therefore, when a commit takes place, the Jenkins server
is informed about this change, and it is able to automate the non-human tasks that
remain to achieve our objective: in-hardware verification. This means the repository
is cloned and then it starts the synthesis process. At this point we invite a new actor
to our party: a powerful server. This server is able to build a partial bitstream in a
few minutes. Thus, the Jenkins server orders the non-human tasks to this powerful
server. Summarising, the Jenkins server plays the role of intermediary, dispatching
costly processes to other actors that are able to perform these tasks more quickly.

Figure 8.12 shows the dashboard of the Jenkins server. It contains five jobs in
accordance with the different scenarios shown in this dissertation. The ball indicates
the current status of the project, a red ball means that the job fails while a blue ball means
that the current version passes all tests. In addition, Jenkins gives some information
about the previous buildings, indicating the status with weather icons. For instance,
sunny means a release is stable, the recent builds do not fail, whereas rainy means a
release that is not stable at all, the recent builds fail.

![Dashboard view of Jenkins](image)

**Figure 8.12:** Dashboard view of Jenkins

Now, we know all actors of our verification service and who does what. But our
verification service does not finish here. The powerful server performs the most difficult
task using a new technology called Docker [Doc17]. Docker allows us to build isolated
environments through a container, it only requires libraries and settings to make the
software work - remember that the vendor tools are software and they are used for
building bitstream files, besides the test cases are executed in a software environment.
This creates efficient, lightweight, self-contained systems and guarantees that software
will always run the same, regardless of where it is deployed.
Jenkins & Continuous Integration

Jenkins is an open source automation server written in JAVA for testing. Jenkins enables developers to find and solve bugs in a code rapidly and to automate testing process. Continuous Integration has developed since this conception.

Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily, leading to multiple integrations per day. Each integration must be verified by an automated build to detect integration errors as quickly as possible [Fow06].

A Dockerfile helps to build an isolated environment and it should be located in the root project directory. It is a script, composed of a number of commands and arguments listed successively to automatically perform actions on a base image, in order to create or form a new one. Listing 8.14 illustrates the Dockerfile used for building our isolated environment. The first line defines the base image to use, in our case, a Debian distribution (testing release). Then we can denote the maintainer of this file. The WORKDIR directive builds our workspace directory. The RUN directive takes the whole line and runs it to form the image; our Dockerfile installs a number of packages that we need to carry out the verification, such as the python3-nose package. The ENV directive sets up the environment variables of our container, thus the tool license must be exported to its environment variable. Finally, we copy all projects cloned from GitHub to our workspace directory.

Listing 8.14: Example of Dockerfile

```bash
FROM debian:stretch
MAINTAINER Julian Caba <julian.caba@uclm.es>
WORKDIR /test
RUN apt-get update && apt-get install -y make libncurses5 libx11-6 libc6-dev python3.4 python3-jinja2 python3-zeroc-ice python3-nose
ENV LM_LICENSE_FILE=1900@atclab.esi.uclm.es
COPY . /test/
CMD sh run.sh
```
Summarising, the verification-tasks description file related to code depicted in Listing 8.13 will run inside a container. On the other hand, about synthesis process, the powerful server uses the approach of Appendix B. It performs the partial bitstream through a reference model. This means adding a new module and configuration into the script that depicts the system. This task is automatically done by a tool denoted `hwt_genTCLDesignConfiguration`, which is called upon by verification-tasks description file. It needs the location of hardware source files and the verification type that it will perform, both parameters are defined at the beginning of verification-tasks description file. In addition, we can denote the output directory where we want to export the new design description file.

Figure 8.13 shows the report generated by Jenkins server after ordering a verification to the powerful server for Demo-UnitTests job.

![Jenkins report](image.png)

*Figure 8.13: Report generated by Jenkins for Demo-UnitTests*
8.4 Summary

In this chapter, we presented a hardware testing service, including an amount of technologies, such as *Jenkins* or *Docker*. This service **hides the complexity of the hardware design** flow, a developer only depicts his design in a high-level programming language, such as C. The rest of non-human tasks are automated by this service. Moreover, the service **returns a report** with information about the whole process.

Among tasks done by this service, we can find the following: it is able to **build partial bitstreams from high-level descriptions** in a remote server whose computational power is higher, saving consumption-time. The remote server uses *Docker* technology, which enables building partial bitstreams into isolated environments, thus we ensure that the environment is clean when the service starts this task.

Automatically, it **deploys partial bitstreams remotely** into an FPGA and is able to **exercise the DUT remotely** with the test cases defined by the developer. *Jenkins* helps to automate non-human tasks and enables **continuous integration**, which is a process in which all development work is integrated as early as possible. The high-level description is automatically created and tested. This process should identify errors early on in the process.
Chapter 9

Conclusion and Future Work

«A conclusion is the place where you got tired of thinking»
Arthur Bloch

9.1 Main Contributions
9.2 Publications
9.3 Future Work

Nowadays, high-level modelling is becoming more and more popular to build new hardware designs, providing an early understanding of the design impact decisions, and allowing a more effective design space exploration, which results in a higher design productivity and improves the likelihood of finding the optimal implementation. In order to fill the existing gap between development tools and the capabilities offered by the technology, FPGA vendors are making a big effort to include High-Level Synthesis (HLS) as a solution [CLN11]. However, the verification stage still entails a number of non-trivial problems.

- The trade-off between simulation effort and verification accuracy completely depends on the design abstraction levels. Using high-level modelling provides the lowest simulation effort but it usually results in inaccuracies, whereas a real hardware prototype would be the perfect environment for an accurate verification but it implies an important verification effort [GD15], and even more the use of real hardware devices introduces a new problem: the exponential synthesis time.

- Each testing-level stage induces rewriting tests, which is time consuming and prone to human errors. The test translation process may result in wrong decisions while developers try to modify them according to the new verification level, instead of just working in production code with the only aim to pass their tests at any level.
• The time spent in verification accounts for roughly 80 percent of the development life-cycle, making this task the bottleneck of most projects and verification engineers are not the only staff to check the design [Fos16]. Therefore, nowadays the biggest challenge in design and verification is identifying solutions to reduce the verification gap, optimising the time-to-market and the product reliability/quality.

• Timing is a major issue in hardware projects, which makes the verification process even more complex. Due to timing results in-hardware level verification usually differs from verification models used during simulation, where developers do not care about available resources and their location. This usually results in non-negligible differences in the internal propagation between the simulated design and the real one.

• Integrating a new component into a test environment fulfilling all its dependencies is a difficult task, since several ones will not be implemented yet or will simply be not available (this is the case for third-party components, for example). Many vendors provide simulation models for their products, but they can rarely be synthesized, and in many cases they do not exactly match their real implementation. In addition, a part of a hardware component can be implemented in different ways, and the decision to do so should be taken at implementation time, where it may prove not to be the correct one because the time requirements are not met.

In this thesis, we presented an approach for verifying hardware FPGA-designs using unit testing frameworks. In order to provide a full verification framework for hardware components, and to overcome the limitations related to hardware/software communication, high-level synthesis tools, and unit testing principles, we explored an alternative based on the RMI paradigm. Our proposal is able to check the correctness of a hardware design at different abstraction levels using the same test cases. In addition, we overcame some of the verification problems listed above, such as internal signal visibility and third-party dependencies, using hardware assertions and hardware mocks respectively. Moreover, our proposal is able to verify in-hardware streaming designs, checking the data production rate and the correctness of these data production at run-time.

On the other hand, all verification processes have been automated through a complex infrastructure. Thus, a developer only has to upload his code in a remote repository. This fact triggers the verification process through several entities to get the bitstream, since the C code depicts the hardware design. This bitstream is deployed in a remote FPGA and exercised remotely with the original test cases. Finally, the developer is able to observe the report provided by the verification infrastructure.

All solutions or scenarios proposed in this dissertation are implemented using an FPGA board, a zedBoard from Xilinx, and the results presented consist of measure-
ments using real hardware. In some cases they have been compared to other hardware results or by the profiling done by FPGA-vendor tools.

The main contributions of the thesis are summarised in section 9.1, while the future work is presented in section 9.3.

9.1 Main Contributions

The major contributions of this thesis can be summarised as follows.

- To communicate a hardware component with other hardware or software components we built a communication mechanism based on RMI that is able to route the required messages. This mechanism can be used for any communication bus, such as AXI, also on a point-to-point bus such as AXI-Stream. In addition, the hardware component was wrapped to ensure individual function access, fulfilling the unit testing principles and HLS restrictions to get a solution based on RMI technology, addressing these limitations. We denote this kind of components as **hardware objects**, which is really a higher abstraction of a hardware component. This new abstraction concept is related to **verification productivity** challenge, moving the hardware domain into high-level modelling.

  - DUTs do not contain code without tests. Each function is verified by one or more unit tests, resulting in fine-grained testing. Thus, engineers assure **verification completeness** capability using unit tests.
  - The **verification efficiency and productivity** are fulfilled because engineers can reuse their test suites in any abstraction level of the typical flow.
  - Our approach meets the FIRST rule, e.g. all tests are **repeatable and independent** using our universal verification platform combined with our hardware component vision (**hardware objects**).
  - Our **hardware objects** provides a neat separation between functionality and communication using a double-wrapper.
  - Our communication mechanism is able to reduce the number of repeated messages, joining **hardware objects** in accordance with their functionality. In addition, communication times are improved using our bus drivers.
  - Our approach removes some restrictions and problems of HLS tools, such as unique entry point, endianess interpretation and hardware castings without **unions**.

- The testing framework chosen was **Unity**, and has been extended to fit hardware verification features. Other testing frameworks can be used extended, as we did
with the `unittest` from Python. Precisely measures are reached due to the Unity testing framework extension, which allows to measure the time elapsed by DUT’s functionality in a real scenario. In addition, the framework extension provides new macros to configure the hardware verification environment without building a new hardware platform, enabling the **hardware verification environment reusability**. Thus, engineers can reuse our hardware platform environment as well as their test suites are kept, hence the combination of unit testing frameworks and RMI technology allows **reusing the same test at any abstraction level**. Besides, the proposed platform does not need special components provided by FPGA vendors, such as hardware timers. Our verification platform is able to perform hardware verification tasks itself.

- **We built a hardware verification platform** based on FPGA technology that is able to **exercise the DUT remotely or from the embedded processor**. In addition, the hardware platform allows **measuring the time elapsed by a DUT task** and checks it from the test case. This hardware platform can be modified using different version tools or increasing the dynamic area. This and the above point are related to the **verification reusability** challenge, our hardware verification platform can be used in future projects while test suites of the current project do not be modified independently of the abstraction level.

- **To facilitate the DPR process**, we built a **fast reconfiguration component** which is able to deploy new functionality without stopping the whole design. So we must synthesize the logic of DUT instead, reducing the computation time and power consumption. In other words, our testing service only generates a partial bitstream in accordance with the component code depicted in a high-level programming language. Our testing platform provides a hardware deployment component (zipFactory) that achieves the maximum theoretical speed up to deploy a partial bitstream though the ICAP component (about 400 MB/s).

- **We addressed the visibility problem** through the implementation of **hardware asserts**, providing a hardware library which contains a set of functions that are able to check intermediate results. The value added by **hardware asserts** is related to exploit the verification capabilities of our hardware verification platform. **Hardware assertions** increase the system observability and controllability and add verification features without adding special debugging components whose interpretation results are complex. In addition, these debugging components store useless information.

- **The verification correctness challenge** is addressed using **mock components** which reduce or **eliminate the third-party dependencies**. Our mock proposal provides extra functionality to express how often, with which arguments and the relative time when the method shall be called, this information is stored internally and can be accessed from unit tests. Thus, the verification stage integrates the third-party dependencies of hardware designs into the verification environment with minimal or no errors, achieving an accuracy verification.
• **Black-box designs** are taken into account in this dissertation. We propose a solution based on UVM’s monitors and unit testing to verify these kinds of designs, which is able to measure the time elapsed between two output data and check its correctness at run-time. The difference with UVM is based on the location of the verification components: our approach runs the verification components in the programmable logic of the FPGA. This fact enables a realistic verification without the need to change the context continuously.

• We have integrated our approach using *Jenkins* as a continuous integration platform connected to a repository, such as *GitHub* which stores the source code: the DUT and its tests. Thereby, when a change is committed to the repository, the *Jenkins* framework triggers the building of the new DUT version in a remote node through a *Docker* container. After the synthesis process, the container sends the partial bitstream to the real hardware and it deploys the new functionality through the DPR capabilities of the FPGA. Then the container stimulates the DUT using the tests defined by the developer. Finally, the developer is able to observe the correctness of the new version in a real hardware. Thus, we get a remote, transparent and automate hardware testing service. This contribution is related to **verification efficiency**. Our approach reduces manual efforts automating the verification process.

### 9.2 Publications

The different stages through which this Ph.D. dissertation has evolved have given rise to several publications. The complete list is presented below:

• The paper entitled «*Rapid prototyping and verification of hardware modules generated using HLS*,» accepted for publication in the *International Symposium on Applied Reconfigurable Computing (ARC)*, proposes a software system (extension of unit testing frameworks) and a hardware architecture (hardware verification environment) to run test cases of FPGA-based designs generated with HLS.

• This Ph.D. dissertation won an award in 2017 in the *Ph.D. category for Xilinx Open European Design Contest* with the work entitled «*FPGAUnit: Hardware verification based on unit tests*». This award indicates the originality and interest of the work done.

• The Ph.D. forum paper «*Functional and timing in-hardware verification of FPGA-based designs using unit testing frameworks*» presented in the *International Conference on Field-Programmable Logic and Applications (FPL)* demonstrates the research interest of the work done.
9.3 Future Work

The work presented in this dissertation can be extended for further improvement. We suggest the following directions for future work.

- Continue the research related to hardware objects for more complex objects. Now our objects are passive, they are not able to initiate a communication themselves. One possibility might be to use an asynchronous communication between two or more entities. For this proposed approach, we suggest two tasks.

  - Implement an asynchronous communication with passive hardware objects, where an entity starts the communication and then asks the hardware object about the result. This does not block the initiator entity.
  - Implement active hardware objects which are able to initiate a communication when the result is ready. The result will be send to the original initiator or to a third-party target, known as indirect communication. This option implies transforming our slave hardware object into a slave-master component.
• Provide a hardware testing platform that enables exercising DUTs with big data. This entails a problem, because the hardware platform must read from and write to DDR, where the stimuli and results are stored. These DDR access operations should be done as fast as possible.

• Nowadays we provide a double FIFO interface between static and dynamic part. Perhaps, it will be interesting to provide other interfaces, such as Xilinx HLS handshake, configuring the communication in accordance with the design requirements.

• Investigate how we can apply other verification techniques whose popularity has been accepted by the industry, such as mutation testing. Mutation testing is used to evaluate the quality of existing tests. This feature is important because we need to assure our design is correct and our tests are right.

• Develop hardware breakpoint-assertions that can stop the testing process, rescue the stored values of internal variables, analyse these values off-chip and restore the testing process or kill it, in accordance with the verification engineer’s decisions.

• Resilent feature is an important capability that our remote testing service must provide. Current testing service only contains a powerful server and an FPGA. Therefore, if these devices fail or are not available, our testing service will not work properly. In addition, users must use the same tool version that was generated the complete bitstream which are running on the FPGA. Therefore, we should provide a heterogeneous service, composed by a grid of FPGAs and powerful servers, which covers the shortcomings of the current service: multi-version tool and different dynamic area sizes. Thus, our remote testing service will be able to scale up or scale down.
Appendix

Systematic Review

«If there is any one secret of success, it lies in the ability to get the other person’s point of view and see things from that person’s angle as well as from your own»

Henry Ford

A.1 Planning of systematic review

A.2 Literature review execution

A.3 Grey Literature and systematic review expansion

A.4 Result analysis

The objective of this section is to review those studies whose research work is in accordance with this thesis. Thus, we are able to know many issues related to this thesis topic: the actual state, the progress and the approaches in this research line. The systematic review is the means to know these issues and helps us looking for related works about verification in SoC or hardware verification methodologies.

The systematic review process is carried out by the Kitchenham’s proposal [Kit04]. This proposal suggests some advice, which help to get a set of works related to a research topic. The digital library, where we search the studies, is the IEEE Xplore library. This library contains all works related to this dissertation topic. It houses electronic systems, distributed systems and reconfigurable systems areas among others.

The following sections explain the different steps that we carried out to perform the systematic review. After, we look at those works that have been marked as important for this dissertation. This selection is done by a set of criteria which helps to select works on the basis of certain criteria. The selected works will be the starting point of this Ph.D. dissertation.
A.1 Planning of systematic review

From the hypothesis of this work, we can shape the search string or strategy, that allows to find a number of related works with the same topic of this thesis. This query should be executed in a digital library to get the list of related works. However, some of these works are not close enough to our topic, thus we need some criteria to get the most related works.

A.1.1 Framing the question

The first step is setting the key questions that the systematic review must answer. These kinds of questions are defined by the populations, the interventions and the outcomes, even the context in which it is applied.

- The populations: digital circuit designs that can harbour the programmable logic devices.
- The interventions: processes, techniques or methodologies that allow carrying out a verification of a digital design.
- The outcomes: speeding up or facilitating the verification step in the defined context. Or, skipping out the verification challenges.
- The context: hardware designs, logic programmable devices and their different verification levels, focusing on the high verification accuracy: the prototype group.

The key questions obtained from the above points are shown in Table A.1.

<table>
<thead>
<tr>
<th>Id</th>
<th>Question</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>Which methodologies are applied to the verification in hardware design context?</td>
</tr>
<tr>
<td>Q2</td>
<td>Which mechanisms are used to carry out the verification in logic programmable devices or in a real hardware?</td>
</tr>
</tbody>
</table>

Table A.1: Key questions

A.1.2 Search protocol

After the key questions have been defined, we may accomplish a search protocol that allows to find the primary studies which will be analysed. This protocol defines
the search strategy, indicating what and where we have to search. The search string is obtained from the key questions defined above and it is composed by several keywords. Thereby, the what part is set by the search string, whereas the where part contains the digital libraries to execute it.

The first step is identifying the keywords from the key questions. These terms have been combined to build the search string, including the synonyms of each term. Table A.2 shows all keywords taken out from the key questions and their synonyms.

<table>
<thead>
<tr>
<th>Id</th>
<th>Term</th>
<th>Synonyms</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>SoC</td>
<td>FPGA</td>
</tr>
<tr>
<td>T2</td>
<td>Verification</td>
<td>Debugging, Testing</td>
</tr>
<tr>
<td>T3</td>
<td>Methodology</td>
<td></td>
</tr>
</tbody>
</table>

Table A.2: Keywords and synonyms

The blending of these terms and their synonyms are done by the logic operators AND and OR. All term and their synonyms are blended through the disjunction, whereas each term is blended through the conjunction with the rest of the terms. This rule is applied if, and only if, the key questions are answered.

FPGA and SoC terms have a similar meaning in this dissertation, thus they are contemplated as synonyms, because our topic is related to the verification and validation of hardware designs, whose verification accuracy is high. Therefore, it is interesting to build the search string with a disjunction of both terms. This fact helps not skewing the search result. Following the rule explained above, the search string obtained is shown as follows.

T1 AND T2 AND T3

Building the string search with the different synonyms, the result search string is shown below. The first part restricts the device to the logic programmable devices or those devices related to hardware components. The second part restricts the result to the verification and validation topics. The third part limits those works that apply a development methodology, however it does not answer the key questions because the third group of terms skews the results, not answering the second question.

(FPGA OR SoC) AND (verification OR testing OR debugging) AND methodology
Before accomplishing the final search string, we tested some search strings, whose results had been analysed. The following items show all search proofs, denoting its badness and its goodness.

- The first proof was the inclusion of the *simulation* term as *verification* synonym. Most results given by this term diverge from the established hypothesis.

- The next search proof was the addition of the *assertion* term to the search string, which was included until the last proofs. Firstly, the term was included as a new conjunction of the string search. This new search string skews the number of obtained works, thus it was discarded. Although with poor results, it was considered in other searching proofs due to its importance in the verification topic. Then the *assertion* term was included as disjunction of the T2 term group, incorporating it as a synonym of *verification, debugging* and *testing*. The obtained results in this pilot search were the same as the results without it. This is caused by the fact that the authors used keywords that did not include the *assertion* term as a keyword, they prefer to include other generalist keywords, such as *verification*. Therefore, we decided to remove this term of the string search after we analysed that it did not give extra information or good results.

- Finally, the last proof was the inclusion of several terms used by some authors in the verification topic, such as *formal verification, co-emulation* or *co-verification*. These terms have been included as synonyms of conjunction of terms T1 and T2. This allowed us to find very interesting works for this dissertation.

After the search proof, the search string was used to get the related work of this Ph.D. dissertation, and it answers the key questions. See below.

\[
(FPGA \text{ OR } \text{SoC}) \text{ AND } (((\text{verification OR testing OR debugging}) \text{ AND } \text{methodology}) \text{ OR } (\text{formal verification OR co-emulation OR co-verification}))
\]

On the other hand, using hardware verification methodologies is more popular in hardware. This fact implies a big effort from the scientific community. It is interesting to include in the related work those works whose main proposal is related to these kinds of techniques, such as Universal Verification Methodology (*UVM*), Open Verification Methodology (*OVM*) and Reference Verification Methodology (*RVM*). The new string search is as follows (results are added to the previous results).

\[
\text{SoC AND (UVM OR OVM OR RVM) AND verification}
\]
The library chosen to find the works related to this dissertation and to carry out the systematic review is *IEEE Xplore Digital Library*. This digital library contains those works related to electronic systems, distributed systems and reconfigurable systems areas among others.

### A.1.3 Criteria

After the definition of the search protocol, the works are selected according to defined criteria, which are divided into two groups: criteria of inclusion (Table A.3) and criteria of exclusion (Table A.4). This list of criteria allows to identify the primary studies, analysing title, abstract and keywords of each work obtained from the string search. This analysis helps to decide which work is inside or outside of the systematic review context of this dissertation.

**Table A.3: Criteria (inclusion)**

<table>
<thead>
<tr>
<th>Id</th>
<th>Criterion</th>
</tr>
</thead>
<tbody>
<tr>
<td>I1</td>
<td>It directly answers any key questions</td>
</tr>
<tr>
<td>I2</td>
<td>It was published between January 2008 and January 2016</td>
</tr>
<tr>
<td>I3</td>
<td>It is focused on using methodologies or formal techniques</td>
</tr>
<tr>
<td>I4</td>
<td>It is focused on hardware verification components</td>
</tr>
<tr>
<td>I5</td>
<td>It uses a real hardware device to carry out the verification process</td>
</tr>
</tbody>
</table>

**Table A.4: Criteria (exclusion)**

<table>
<thead>
<tr>
<th>Id</th>
<th>Criterion</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>It does not answer any question</td>
</tr>
<tr>
<td>E2</td>
<td>It is not related with engineering</td>
</tr>
<tr>
<td>E3</td>
<td>It is not published between January 2008 and January 2016</td>
</tr>
<tr>
<td>E4</td>
<td>It is focused on fault-tolerance</td>
</tr>
<tr>
<td>E5</td>
<td>It shows timing analysis, power consumption, throughput or efficiency</td>
</tr>
<tr>
<td>E6</td>
<td>It is focused on device building (FPGA, PCB, ...)</td>
</tr>
<tr>
<td>E7</td>
<td>It is duplicated</td>
</tr>
<tr>
<td>E8</td>
<td>It is a journal or congress proceedings</td>
</tr>
</tbody>
</table>

After the execution and the analysis process, we can add new articles or works, such as Ph.D. dissertations, technical reports or abstracts, whose topic is related to this dissertation, and they can even be obtained from the bibliography of the selected studies in the systematic review. These kinds of works are known as *Grey Literature*.
and we apply the same criteria to it as defined above, except for the E2 criterion which should be ignored.

A.2 Literature review execution

The next step, after the planning of the systematic review, is its execution, with the aim to obtain the primary studies. This collection is obtained using the string search in the library. The obtained works are filtered by the criteria for the primary works, are analysed to get the related works. Table A.5 shows the result of the literature review execution. This illustrates the percentage of accepted works and its source: congress or journal.

<table>
<thead>
<tr>
<th></th>
<th>Global</th>
<th>Journal</th>
<th>Congress</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>63</td>
<td>4</td>
<td>59</td>
</tr>
<tr>
<td>Accepted</td>
<td>26</td>
<td>2</td>
<td>24</td>
</tr>
<tr>
<td>Percentage</td>
<td>41.3%</td>
<td>50%</td>
<td>40.7%</td>
</tr>
</tbody>
</table>

Table A.5: Summarising of search process in IEEE Xplore

During the analysis process of the primary studies, we have found several perspectives of engineers facing the verification stage in a logic programmable device. These points of view are classified according to the tuple precision-effort, obtaining the actual-state topic and catching sight of the points that should be explored. Some works could be placed in more than one category, in this case, the work is framed in that category whose research work is more interesting or in accordance with its proposal. Therefore, looking at the plot shown in Figure 1.4, the works are categorised according to these five groups, appointing them with the same name: HL-Modelling, RTL Simulation, Timing Simulation, Emulation and Prototype.

- **HL-Modelling.** This group contains works that use a high-level modelling and are able to simulate, verify or debug a hardware design. This includes those works that simplify or automate low-level tasks, such as code translators.

- **RTL Simulation.** This group contains works that build a verification environment based on RTL, this environment is able to simulate or check a code described in any HDL. It includes works that make a functional verification without keeping in mind the timing component and it only contains works whose aim is to check the behaviour of hardware designs.
• **Timing Simulation.** This group contains works that are able to build a verification environment that checks the timing component or, in other words, works that check a hardware design which meets timing requirements. Generally, most methodologies and hardware verification techniques make this kind of simulation. Thus, the works related to hardware verification methodologies are included in this group.

• **Emulation.** This group contains works whose aim is to build an emulation platform. This platform is able to replace or virtualise some elements of a verification environment that are not available, such as real hardware or third-party components.

• **Prototype.** This group contains works whose proposal is based on using real hardware to check the hardware design, such as a hybrid platform. The software domain manages the verification process, whereas the hardware domain contains the hardware design to verify.

<table>
<thead>
<tr>
<th>Year</th>
<th>HL-Modelling</th>
<th>RTL Simulation</th>
<th>Timing Simulation</th>
<th>Emulation</th>
<th>Prototype</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2009</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2010</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2011</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>2012</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2013</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>2014</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>2015</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Total</td>
<td>5</td>
<td>2</td>
<td>9</td>
<td>3</td>
<td>7</td>
<td>26</td>
</tr>
</tbody>
</table>

| Percentage | 19.2% | 7.7% | 34.6% | 11.5% | 26.9% |

**Table A.6:** Summary by categories and years (primary studies)

Table A.6 shows the categorisation of primary studies ordered by the above categories and their publication year. Analysing this table, we can deduce that the topic of this dissertation has been taken into account the last years, because it is a hot topic in the scientific community. This can be seen in the selected works, of
which roughly 50% was published between 2013 and 2015. On the other hand, we can observe the Timing Simulation group has grown, due to timing component concern of hardware designs, in addition most work-groups of different companies have taken in a verification methodology for their development flow [Fos16].

In order to get the list of related works, we should analyse the primary studies, reading their abstract introduction and experimental results. To facilitate this task, we have created a template that meets all information of each primary study, denoting the importance of a work for this dissertation. This template helps us to quickly identify a work. The list of related works composes the chapter titled related work. The template used is as follows.

![Study Title Table]

<table>
<thead>
<tr>
<th>Authors</th>
<th>Author(s) of study</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity</td>
<td>Author’s organisation</td>
</tr>
<tr>
<td>Date</td>
<td>Publish date</td>
</tr>
<tr>
<td>Source</td>
<td>Name of Journal or Conference</td>
</tr>
<tr>
<td>Category</td>
<td>Group where it is categorised</td>
</tr>
<tr>
<td>Criteria</td>
<td>IDs of inclusion criteria</td>
</tr>
<tr>
<td>Problems</td>
<td>List of treated problems</td>
</tr>
<tr>
<td>Proposal</td>
<td>Summary of proposal</td>
</tr>
<tr>
<td>Relevant</td>
<td>Yes/No</td>
</tr>
<tr>
<td>Reference</td>
<td>Bibliography reference</td>
</tr>
</tbody>
</table>

Primary studies detected during the systematic review process are listed below. This list summarises the above form, denoting if the study is relevant or not, its bibliography reference and its title.

- [LWH08] “AMBA AHB bus protocol checker with efficient debugging mechanism”
- [Abr08] “In-System Silicon Validation and Debug”
- [ZPS12] “Practical and efficient SOC verification flow by reusing IP testcase and testbench”
A. Systematic Review

- [MSEKD14] “System Verilog Assertion Debugging Based on Visualization, Simulation Results, and Mutation”
- [BGJ11] “A scalable hybrid verification system based on HDL slicing”
- [LLR08] “Hierarchy Communication Channel in Transaction-Level Hardware/-Software Co-emulation System”
- [YKKM11] “Beyond UVM for practical SoC verification”
- [KWH09] “Design of SoC verification platform based on VMM methodology”
- [NW15] “Functional coverage-driven UVM-based UART IP verification”
- [MI11] “Developing an integrated verification and debug methodology”
- [SKK12] “Platform for automated HW/SW co-verification, testing and simulation of microprocessors”
- [Put14] “Method of free C++ code migration between SoC level tests and standalone IP-Core UVM environments”
- [BUEO15] “A layered UVM based testbench design for SpaceWire”
- [MSC13] “Formal equivalence checking between high-level and RTL hardware designs”
- [LBGS11] “A Low-Cost Emulation System for Fast Co-verification and Debug”
- [HYH11] “SoC HW/SW verification and validation”
- [LZHX14] “UVM-AMS based sub-system verification of wireless power receiver SoC”
- [SCDG14] “UVM based STBUS verification IP for verifying SoC architectures”
- [MYS14] “A framework for rapid prototyping of embedded vision applications”
- [NF15] “Binary floating point verification using random test vector generation based on SV constraints”
- [GPK12] “Formal-Analysis-Based Trace Computation for Post-Silicon Debug”
- [GLG13] “Design and Verification of a MAC Controller Based on AXI Bus”
- [PiCK15] “FPGA Prototyping and Accelerated Verification of ASIPs”
A systematic review result could be expanded. The aim of this expansion is to determine how extensive the literature related to this thesis is. Therefore, we add two not chained stages. In the first stage, we identify the journals and congress related to the topic of this dissertation, including other studies from the primary studies analysed previously. In this case we must remove the criterion that restricts the published year. While the second stage contains the related literature that is not formally published, this kind of literature is known as «Grey Literature». This group contains Ph.D. dissertations, technical reports, research works without a formal publishing, ...

Beginning with the second stage, the literature that was not formally published or «Grey Literature», we have included the following references.


In the first stage, the published literature between 2012 and 2015, we have selected the following special issues and specific congresses or journals.

- **DATE**. Design, Automation Test in Europe. (Congress)
- **DAC**. Design Automation Conference. (Congress)
- **DVCON**. Design and Verification Conference and Exhibition. (Congress)
- **i-CAV**. International Conference on Computer-Aided Verification. (Congress)
- **Design & Test**. (Journal)
- **VLSI**. Very Large Scale Integration Systems. (Journal)
A. Systematic Review

The list of this systematic review extension is shown below. This list summarizes the form used for the primary studies, denoting the work’s bibliography reference and its title.

- [CRL10] “A run-time RTL debugging methodology for FPGA-based co-simulation”
- [ICC10] “Using partial reconfiguration and high-level models to accelerate FPGA design validation”
- [LZ11] “FPGA level in-hardware verification for DO-254 compliance”
- [EA14] “UVM SchmooVM - I want my c tests!”
- [CCT07] “Bridging RTL and gate: correlating different levels of abstraction for design debugging”
- [BGG13] “Logic emulation with forced assertions: A methodology for rapid functional verification and debug”
- [GPS14] “A cross-level verification methodology for digital IPs augmented with embedded timing monitors”
- [CLYX14] “Coverage evaluation of post-silicon validation tests with virtual prototypes”
- [HCG13] “Fast and accurate TLM simulations using temporal decoupling for FIFO-based communications”
- [BFPS15] “RTL property abstraction for TLM assertion-based verification”
- [YOS09] “Implementation of a hardware functional verification system using SystemC infrastructure”

A.4 Result analysis

The main goal of this systematic review is to know the actual state of our topic and its trends. This information is obtained through an exhaustive analysis of related works and Grey Literature. Therefore, after the systematic review we are able to deduce how important the topic of this dissertation is within scientific community, even in the industrial field. This fact can be observed by the author’s entity. A number of related works are written by employees of development-tool vendors, such as Mentor, Acellera or Aldec.
During the analysis of selected studies, we have identified a set of problems, grouping them into different classes. Table A.7 and Table A.8 show the problems in columns and the works in rows. This organisation simplifies the way to find a study that tries to solve a specific problem. The following points describe the identified problems organised by problem classes.

- **Completeness:** This class identifies those problems related to completeness challenge. Including the features which are not related to the system behaviour.
  
  - *Control of handshaking.* There are some works that include a communication protocol between DUT and its outside, thus they bring a small driver that is able to manage the communication layer. Its main objective is to check the protocol *handshake*, including the signal timing.
  
  - *DUT exercising.* Stimuli generation is one of the problems more popular in verification processes, and it maybe one of the most important problems. Good input vectors create good verification, in which the most possible scenarios are covered. To achieve a high number of scenarios, some works propose a novel solution based on random stimuli generation which is adopted by some verification methodologies.
  
  - *Timing control.* Nowadays, timing verification is more popular, checking the timing correctness of a hardware design. For example, in a video algorithm the client should indicate the maximum time allowed to handle a frame.
  
  - *Code coverage.* One of the most important features in the verification process is the code coverage, in which a high percentage means that the stimuli are good. This group contains those works that contribute with a novel proposal to achieve a high degree coverage.

- **Reusability:** This group contains those works increasing portions of the verification environment infrastructure that can be used in the current project or in future projects.
  
  - *Reuse.* One solution is the verification element reuse (component, environment, design, ...). The main problem of this solution is its universality, which should be reused for the same kind of applications.
  
  - *Component verification.* This group denotes those works which face the hardware component verification directly, thus the verification step at least checks the behaviour of a hardware design.
  
  - *DUT communication.* This problem group contains works that are focused on direct communication with DUT. Thereby, it contains the works that propose a protocol, using a standard bus, or a universal interface.

- **Efficiency:** Some works try to reduce manual efforts, in counterpart automated systems are able to complete many tasks in a short time. Therefore this group contains works that automate or reduce manual efforts and errors.
- **Task automation.** Automation helps to reduce the effort and the bugs generated by engineers when they build something hand-made. This group contains the above problems because automation reduces the verification effort, reduces the *time-to-market*, reduces bugs and even promotes reuse.

- **Reduction of time-to-market.** One challenge in digital electronic design of companies is the *time-to-market*. This feature is also relevant in verification processes, because of its actual difficulty and the shorter window for *time-to-market*.

- **Reduction of the verification effort.** The verification step is the most difficult task in the development flow. Works that reduce this task are included in this group.

- **Correctness:** This group includes those works that check the correctness of designs, including third-party dependencies.

  - **Reduction of bugs in tests.** Tests can contain some errors that a verification engineer must fix. A bug in a test creates problems which are difficult to solve, because the engineer modifies the code instead of the test.

  - **Third-party dependencies.** A number of designs have a high third-party dependency. This fact makes for a difficult verification process, even in extreme cases it is not feasible to carry out the verification process, unless the third-party provides a simulation model. There are several proposals that reduce or remove these third-party dependencies.

  - **Using assertions.** Assertions terms are included in some works as approach or as a problem, depending on the author’s point of view. This kind of proposal allows verifying the results or middle results of a hardware design whenever we have a golden reference available. A special component related to this group are the *checker* components that allow to check the correctness of a design. Its main drawback is the complexity they bring to the verification stage, becoming a difficult task for verification engineers.

- **Productivity:** This class includes those works that try to maximise work that is produced manually in a certain period of time.

  - **Independence of verification level.** In hardware verification, we can find different abstraction levels, RTL, Gate-level, ...Each level depends on simulation precision. The main problem is building a verification environment or the tests to verify a design, because test description depends on the verification level which hinders its reutilisation.

  - **Methodologies.** Using hardware verification methodologies or a well-defined verification process, increases the development time also contributes to building a robust design. Some works drive their research work to improve the development flow of these methodologies, while others build their own verification methodology.
– **Visibility.** New vendor tools bring new problems. One of these problems is the signal visibility, due to the high gap between high-level programming languages and the code generated by these tools. In addition, these tools do not provide any mechanism to inspect the signals generated by them, thus a verification engineer is faced with a manual verification through a signal schedule in a simulator tool, looking for the signal and/or internal signals generated by the tool. The works simplify the signal visibility are included in this group.
## Table A.7: Summary of partial problems (Systematic review)

<table>
<thead>
<tr>
<th>Article</th>
<th>Control of handshaking</th>
<th>DUT exercising</th>
<th>DUT communication</th>
<th>Timing control</th>
<th>Independence of verification level</th>
<th>Third-party dependencies</th>
<th>Methodologies</th>
<th>Component verification</th>
<th>Using assertions</th>
<th>Code coverage</th>
<th>Visibility</th>
<th>Reduction of bugs in tests</th>
<th>Reduction of verification effort</th>
<th>Reduction of time-to-market</th>
<th>Reuse</th>
<th>Task automation</th>
</tr>
</thead>
<tbody>
<tr>
<td>[LWH08]</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[Abr08]</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[ZPS12]</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[Sal14]</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[SBY11]</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[MSEKD14]</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[BGJ11]</td>
<td>-</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[LLR08]</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[YKKM11]</td>
<td>-</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[KWH09]</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[ZLG15]</td>
<td>-</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[NW15]</td>
<td>-</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[MI11]</td>
<td>-</td>
<td>-</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[SKK12]</td>
<td>-</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[Put14]</td>
<td>-</td>
<td>-</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[BUEO15]</td>
<td>-</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[MSC13]</td>
<td>-</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[LBGS11]</td>
<td>-</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[HYH11]</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[LZH14]</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[SCDG14]</td>
<td>-</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[MYS14]</td>
<td>-</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[NF15]</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[GPK12]</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[GLG13]</td>
<td>-</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[PiCK15]</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Article</th>
<th>Control of handshaking</th>
<th>DUT exercising</th>
<th>DUT communication</th>
<th>Timing control</th>
<th>Independence of verification level</th>
<th>Third-party dependencies</th>
<th>Methodologies</th>
<th>Component verification</th>
<th>Using assertions</th>
<th>Code coverage</th>
<th>Visibility</th>
<th>Reduction of bugs in tests</th>
<th>Reduction of verification effort</th>
<th>Reduction of time-to-market</th>
<th>Reuse</th>
<th>Task automation</th>
</tr>
</thead>
</table>
A.4. Result analysis

<table>
<thead>
<tr>
<th></th>
<th>Communication</th>
<th>Techniques</th>
<th>Analysis</th>
<th>Facilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Article</td>
<td>DUT</td>
<td>DUT</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>handshaking</td>
<td>communication</td>
<td>Timing control</td>
<td>Independence of verification level</td>
</tr>
<tr>
<td>[CRL10]</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>[ICC10]</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>[LZ11]</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>[EA14]</td>
<td>-</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[CCT07]</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>[BGG13]</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>[GPS14]</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>[CLYX14]</td>
<td>-</td>
<td>✓</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>[HCG13]</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>[BFPS15]</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>[BEG15]</td>
<td>-</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>[YOS09]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table A.8: Summary of partial problems (Systematic review extension)
Facilitating DPR tasks through TCL scripts

«Never walk on the traveled path because it only leads where others have been»
Graham Bell

B.1 Defining dynamic hardware project
B.2 Automatic generation of bitstreams

Some FPGA vendors have made advances in reconfigurable devices, adding DPR characteristics. It makes FPGAs more attractive for embedded system designs. This technique allows to adapt the system at run-time, considering environmental changes [MMT08]. This feature increases the flexibility of FPGAs, because a single platform could be used to implement a larger variation of functionality without a full configuration [Bob07]. DPR reduces costs, for example reusing logic areas in Software Defined Radio (SDR) projects is not necessary to deploy all possible radio waveforms, only the one required in a concrete context may be configured, so context variations may imply new dynamic waveforms.

Building a reconfigurable project is a long and a complex task, mainly because the FPGA vendor tools works at a very low level of abstraction, we use the Vivado tool from Xilinx. This chapter presents an alternative based on TCL scripting compatible with the FPGA vendor toolset. This alternative works as a front-end of the Vivado tool, but with the idea of rising the task abstraction level that a developer must perform. Our starting point is a block design generated from the Vivado tool.

The following sections explain the flow proposed to get full and partial bitstreams easily. Firstly, we must depict the hardware project design into a TCL file, called project description file. Then we must run another script that automates all synthesis process, this second script gets the project information from the project description file described previously.
B.1 Defining dynamic hardware project

Firstly, we must define the whole hardware project using the Vivado tool from Xilinx. We really need the block diagram generated by this tool, so we must export it into a TCL file. Then, a project description file is written to translates this static hardware project into a dynamic one. This file is written in TCL, which is compatible with the Vivado toolset, and it is organised as follows.

**Board settings** This small section only contains the board settings; the part, package and speed-grade of target device and board device. For example, a project runs on a zedboard. The part section is xc7z020clg484-1 (see Listing B.1).

```tcl
set originDir [file dirname [info script]]
set part xc7z020clg484-1
set board "em.avnet.com:zed:part0:1.2"
```

**Project settings** After the board settings section, the project settings must be defined. Firstly, the project name and the IP cores directory are defined (lines 1 and 2 of Listing B.2, respectively). This path contains the location of hardware components developed by the designer or by a third-party, which were exported as a IP-XACT standard. Then, a block design file is added, depicting its path and design name (lines 3 and 4 of Listing B.2, respectively). The block design is built by engineers using the Vivado tool, including all components that integrate the whole design project, and it is exported by the same tool into a TCL file, which defines the design components, its connections and configurations. This design must include the dynamic components or a fake of them. For example, if we know the interfaces of each dynamic area, we can code a component that simulates the real component without any functionality, this is a dummy or fake component. Finally, the constraints are included in this section too (line 6 of Listing B.2). The constraints can be defined in one or more files, thus we only have to include their location in both cases. We will assume that the designer has already built all required components for the hardware design project, and has integrated and verified them, thus its behaviour is correct.

```tcl
set prjName tmp
set userIPDir ip_repo
set blockDesign src/design/design_1.tcl
set designName "ps_system"
set xdcFiles [fileGroup::create]  # constraints
fileGroup::add $xdcFiles src/xdc/ topZedboard.xdc
```
Dynamic areas After the system is defined with all its components and constraints, we have to specify which components or what parts of them are dynamic and their location in the logic device part. Therefore, we define a dynamic module with an identifier, which really is a virtual representation of a dynamic area (line 1 of Listing B.3). Then we include the dynamic part path from the source tree generated by Vivado tool. Its instance name is also added (lines 2 and 3 of Listing B.3, respectively). Finally, we must add the physical location of the dynamic area and its resources, such as DSP48, BRAMs, etc. into this virtual representation (line 4 of Listing B.3).

Listing B.3: Dynamic areas section

```tcl
set area1 [rmGroup::create "area1"]
rmGroup::setSrcLocation $area1 "ps_system_i/dpr_0/U0/"
rmGroup::setInstanceName $area1 "leds_i"
rmGroup::setHwResources $area1
[list SLICE_X34Y31:SLICE_X45Y44 DSP48_X2Y14:DSP48_X2Y17]
```

Dynamic modules After the dynamic areas section, we must include the source files of each dynamic module by a file group object created in TCL (lines 1 to 3 of Listing B.4). Each file group contains the behaviour of a dynamic component, which is coded by any HDL, such as VHDL or Verilog. Therefore, a file group contains the source files of a dynamic component. This group type can contain a component generated by the Vivado tool described from Xilinx Core Instance (XCI) or Design Checkpoint (DCP) format. For instance, when a dynamic module contains floating-point operations, such as adders, we must generate a module from the Vivado tool with specific settings to get the desired performance. Finally, we must associate a file group with a dynamic area defined in the previous step, this linking is identified by a unique name (impl1, line 4 of Listing B.4). Summarising, a dynamic area can contain several file groups and a file group can be part of some dynamic areas.

Listing B.4: Dynamic modules section

```tcl
set impl1Files [fileGroup::create]
fileGroup::add $impl1Files src/leds1/ add.dcp
fileGroup::add $impl1Files src/leds1/ leds.vhd
rmGroup::addNewModule $area1 "impl1" $impl1Files

set impl2Files [fileGroup::create]
fileGroup::add $impl2Files src/leds2/ mult.dcp
fileGroup::add $impl2Files src/leds2/ leds.vhd
rmGroup::addNewModule $area1 "impl2" $impl2Files
```

Environment configurations Finally, we must define the possible environment that could be reached (see Listing B.5). In our example, we have two dynamic modules
for a dynamic area, so we must define all possible scenarios. Listing B.5 shows the configuration of the second dynamic module defined in Listing B.4 for the dynamic area defined at Listing B.3, because the first module plays a role of reference design and it is used for reducing the synthesis process time, thus this scenario contains the first linking that the designer made at the dynamic modules stage, and therefore the first linking does not need an explicit configuration. We do not need any more configurations to define all possible scenarios.

This section includes two options related to the bitstream generation, which are enabled and disabled with the first two lines as shown in Listing B.5, by default both options are disabled. The first option is the bitstream compression applied by the Vivado tool, the second is blanking bitstream generation, which can be used in an initial deployment, so the first environment which is deployed will not contain a dynamic module in any of its dynamic areas.

Listing B.5: Environment configuration section

```
1 set compressReference off
2 set createBlanking off
3 set cfg0 [cfgGroup::create "cfg0"]
4 cfgGroup::addNewModule $cfg0 $area1 "impl2" "implement"
```

Although we are able to translate a static project into a dynamic one, that is not an objective of this dissertation. However, reducing the synthesis process time is one of our objectives. This project description file and other TCL scripts for bitstream generation process allows synthesizing an isolate part of the hardware design project. This fact brings an important goodness; we only synthesise the DUT part. To carry out this task we must define a reference configuration, which contains a dummy hardware object performing as DUT, but it does not have any functionality. Therefore, there is not any configuration defined explicitly in the project description file. Finally, we must run the synthesis through scripting files provided by our proposal and store all files generated to use them as reference design for future synthesis, saving consumption-time and consumption-power. When we have defined our hardware object, we can synthesise it to get the partial bitstream. To carry out this process we must define a new module inside the project description file and a new configuration that enables the generation of its configuration files or bitstreams. Then we run the synthesis scripts again to obtain these configuration files.

## B.2 Automatic generation of bitstreams

Figure B.1 shows a decision flow diagram for building configuration files, through the building script. All stages check which files have been generated previ-
ousely, if the file exists, the script jumps to the next stage, otherwise the script runs the TCL commands to build the stage’s files.

Figure B.1: Decision flow diagram to build configuration files
About the Author

Julián Caba received the B.S. and M.S. degrees in Computer Science from the University of Castilla-La Mancha (UCLM), Spain, in 2006 and 2009, respectively. In 2009, he started working as a researcher in the group of Prof. Juan Carlos López López, ARCO (Computer Architecture and Networks). In 2013 he started his Ph.D. in Informatics Engineering at UCLM. He visited the Faculty of Engineering of the University of Porto (FEUP), Portugal, and worked in the group of Prof. João M. Paiva Cardoso (Special-Purpose Computing Systems, languages and tools). He is currently an Assistant Professor at the University of Castilla-La Mancha. His current research interests include hardware verification methodologies, high-level synthesis, run-time reconfigurable systems and heterogeneous distributed systems.

Publications Related to the Ph.D. Dissertation

«RC-Mock: Mocking Framework para módulos hardware generados mediante HLS»

<table>
<thead>
<tr>
<th>Authors</th>
<th>J. Caba, F. Rincón, J.D. Dondo, J. Barba, M.J. Abaldea and J.C. López</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date</td>
<td>2018</td>
</tr>
<tr>
<td>Publisher</td>
<td>Jornadas de Computación Empotrada y Reconfigurable (JCER)</td>
</tr>
<tr>
<td>Funding</td>
<td>PLATINO (TEC2017-86722-C4-4-R)</td>
</tr>
<tr>
<td></td>
<td>SimbIoT (SBPLY-17-180501-000334)</td>
</tr>
</tbody>
</table>
**B.2. Automatic generation of bitstreams**

<table>
<thead>
<tr>
<th>«Testing framework for in-hardware verification of the hardware modules generated using HLS»</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Authors</strong></td>
</tr>
<tr>
<td><strong>Date</strong></td>
</tr>
<tr>
<td><strong>Publisher</strong></td>
</tr>
<tr>
<td><strong>Rating</strong></td>
</tr>
<tr>
<td><strong>Funding</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>«Rapid prototyping and verification of hardware modules generated using HLS»</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Authors</strong></td>
</tr>
<tr>
<td><strong>Date</strong></td>
</tr>
<tr>
<td><strong>Publisher</strong></td>
</tr>
<tr>
<td><strong>Rating</strong></td>
</tr>
<tr>
<td><strong>Funding</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>«Functional and timing in-hardware verification of FPGA-based designs using unit testing frameworks»</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Authors</strong></td>
</tr>
<tr>
<td><strong>Date</strong></td>
</tr>
<tr>
<td><strong>Publisher</strong></td>
</tr>
<tr>
<td><strong>Rating</strong></td>
</tr>
<tr>
<td><strong>Funding</strong></td>
</tr>
</tbody>
</table>
## «A Scalable and Dynamically Reconfigurable FPGA-Based Embedded System for Real-Time Hyperspectral Unmixing»

**Authors**  
T.G. Cervero, J. Caba, S. López, J.D. Dondo, R. Sarmiento, F. Rincón and J.C. López  
**Date**  
2015  
**Publisher**  
Journal of Selected Topics in Applied Earth Observations and Remote Sensing (JSTARS)  
**Impact Factor**  
2.145 (Q1)  
**Funding**  
DREAMS (TEC2011-28666-C04-04)

## «Run-Time Partial Reconfiguration Simulation Framework Based on Dynamically Loadable Components»

**Authors**  
X. Peña, F. Rincón, J.D. Dondo, J. Caba and J.C. López  
**Date**  
2015  
**Publisher**  
International Symposium on Applied Reconfigurable Computing (ARC)  
**Rating**  
B- (GSS Class 3)  
**Funding**  
DREAMS (TEC2011-28666-C04-03)

## «Facilitating Preemptive Hardware System Design using Partial Reconfiguration Techniques»

**Authors**  
J.D. Dondo, F. Rincón, C. Valderrama, F.J. Villanueva, J. Caba and J.C. López  
**Date**  
2014  
**Publisher**  
Scientific World Journal (SWJ)  
**Impact Factor**  
1.73 (Q2)  
**Funding**  
DREAMS (TEC2011-28666-C04-03)
«Plataforma de co-simulación basada en SystemC para el desarrollo de sistemas de gestión y planificación de sistemas reconfigurables en FPGAs»

**Authors** X. Peña, J.D. Dondo, F. Rincón, J. Caba and J.C. López  
**Date** 2013  
**Publisher** Jornadas de Computación Empotrada (JCE)  
**Funding** DREAMS (TEC2011-28666-C04-03)  
ENERGOS (CEN-20091048)

«Development flow for FPGA-based scalable reconfigurable systems»

**Authors** J. Caba, J.D. Dondo, F. Rincón, J. Barba and J.C. López  
**Date** 2013  
**Publisher** Euromicro Conference on Digital System Design (DSD)  
**Rating** B (GGS Class 3)  
**Funding** DREAMS (TEC2011-28666-C04-03)  
ENERGOS (CEN-20091048)

**Awards**

«FPGAUnit: Hardware verification based on unit tests»

**Contest** Xilinx FPGA and SOC University Design Contest, Europe (Open Hardware Design Contest)  
**Date** 2017  
**Category** Winner of Ph.D. Category

**Speeches**

«Verificación de diseños hardware basados en FPGAs mediante test unitarios y hardware reconfigurable»

**Organizer** Escuela de Ingeniería Minera e Industrial de Almadén  
**Topic** VI Jornadas de Investigación EIMIA  
**Date** 2017
Other Publications

«HLSTL: Biblioteca para programación genérica sobre FPGA con HLS»

Authors  M.J. Abaldea, J. Barba, J. Caba, F. Rincón, J.D. Dondo and J.C. López
Date  2017
Publisher  Jornadas Sarteco
Funding  REBECCA (TEC2014-58036-C4-1R)
          SAND (PEII-2014-046-P)

«FPGA Acceleration of Semantic Tree Reasoning Algorithms»

Authors  J. Barba, M.J. Santofimia, J.D. Dondo, F. Rincón, J. Caba and J.C. López
Date  2015
Publisher  Journal of Systems Architecture (JSA)
Impact Factor  0.683 (Q3)
Funding  DREAMS (TEC2011-28666-C04-03)
          ENERGOS (CEN-20091048)

«Integracion Hw/Sw para aplicaciones medicas basadas en el estándar OpenMAX»

Authors  D. Fuente de San Venancio, J.D. Dondo, J. Barba, J. Caba and J.C. López
Date  2014
Publisher  Conferencia Regional en Instrumentación Avanzada (CRIA)
Funding  DREAMS (TEC2011-28666-C04-03)
          ENERGOS (CEN-20091048)
### « Diseño de controladores de comunicaciones a la medida a partir de síntesis de alto nivel »

<table>
<thead>
<tr>
<th><strong>Authors</strong></th>
<th>F. Rincón, J.D. Dondo, J. Barba, J. Caba and F. Sánchez</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Date</strong></td>
<td>2013</td>
</tr>
<tr>
<td><strong>Publisher</strong></td>
<td>Jornadas de Computación Empotrada (JCE)</td>
</tr>
<tr>
<td><strong>Funding</strong></td>
<td>DREAMS (TEC2011-28666-C04-03)</td>
</tr>
<tr>
<td></td>
<td>ENERGOS (CEN-20091048)</td>
</tr>
</tbody>
</table>

### « Flujo de desarrollo de sistemas reconfigurables escalables sobre FPGAs »

<table>
<thead>
<tr>
<th><strong>Authors</strong></th>
<th>J. Caba, J.D. Dondo, F. Rincón and J.C. López</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Date</strong></td>
<td>2012</td>
</tr>
<tr>
<td><strong>Publisher</strong></td>
<td>Jornadas de Computación Reconfigurable y Aplicaciones (JCRA)</td>
</tr>
<tr>
<td><strong>Funding</strong></td>
<td>DREAMS (TEC2011-28666-C04-03)</td>
</tr>
<tr>
<td></td>
<td>ENERGOS (CEN-20091048)</td>
</tr>
</tbody>
</table>
Bibliography

[Abr08]  M. Abramovici.  
In-system silicon validation and debug.  

[Ace04]  Acellera.  


Icec.  

Timer core (d8254).  

[Ant16]  Sebastian Anthony.  
Transistors will stop shrinking in 2021, but moore's law will live on.  
Apple.
Clang project.

J. Barba.
*Infraestructura HW-SW orientada a objetos para la gestión uniforma de las comunicaciones en sistemas en chip.*

The next generation of virtual prototyping: Ultra-fast yet accurate simulation of hw/sw systems.

B. Bombieri, R. Filippozi, G. Pravadelli, and F. Stefanni.
Rtl property abstraction for tlm assertion-based verification.

Logic emulation with forced assertions: A methodology for rapid functional verification and debug.
*Fifth Asia Symposium on Quality Electronic Design (ASQED 2013)*, 2013.

A scalable hybrid verification system based on hdl slicing.
*2011 IEEE International High Level Design Validation and Test Workshop*, 2011.

Bitvis.

C. Bobda.
*Introduction to Reconfigurable Computing: Architectures, algorithms and applications.*
University of Kaiserslautern, 2007.
Doulos, 2011.

The lara compiler suite.
*Design Automated Test in Europe*, 2014.

A layered uvm based testbench design for spacewire.
*2015 9th International Conference on Electrical and Electronics Engineering (ELECO)*, 2015.

[BZ08] M. Boul and Z. Zilic.
*Generating Hardware Assertion Checkers.*

[C.05] Kao C.
Benefits of partial reconfiguration.
*XCell*, 2005.

Legup: An open-source high-level synthesis tool for fpga-based processor/accelerator systems.
2011.

A scalable and dynamically reconfigurable fpga-based embedded system for real-time hyperspectral unmixing.

[CCT07] Eric Cheung, Xi Chen, Furshing Tsai, Yu-Chin Hsu, and Harry Hsieh.
Bridging rtl and gate: correlating different levels of abstraction for design debugging.
[Cel02] Celoxica. 

[CKU04] G. Chen, M. Kandemir, and Sezer U. 
Configuration-sensitive process scheduling for fpga-based computing platforms. 

High-level synthesis for fpgas: From prototyping to development. 

An independent evaluation of: The autoesl autopilot high-level synthesis tool. 
2010.

Coverage evaluation of post-silicon validation tests with virtual prototypes. 

[CRL10] X. Cheng, A. W. Ruan, Y. B. Liao, P. Li, and H. C. Huang. 
A run-time rtl debugging methodology for fpga-based co-simulation. 

A multi-platform controller allowing for maximum dynamic partial reconfiguration throughput. 
*Field Programmable Logic and Applications*, 2008.

[D.12] Dye D. 
Partial reconfiguration of xilinx fpgas using ise design suite. 
2012.
Docker.

[Dou17] Doulos.

Facilitating preemptive hardware system design using partial reconfiguration techniques.

Histograms of oriented gradients for human detection.

Uvm schmoovm - i want my c tests!

[Fos16] H. Foster.
Functional verification study.

[Fow06] M. Fowler.
Continuous integration.


[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vissides.
Design Patterns: Elements of Reusable Object-Oriented Software.
Addison-Wesley, 1994.
Open Verification Methodology Cookbook.

Design and verification of a mac controller based on axi bus.

Ctemplate project.

Formal-analysis-based trace computation for post-silicon debug.

A cross-level verification methodology for digital ips augmented with embedded timing monitors.

Fast and accurate tlm simulations using temporal decoupling for fifo-based communications.

A design approach to automatically synthesize ansi-c assertions during high-level synthesis of hardware accelerators.

Fast dynamic and partial reconfiguration data path with low hardware overhead on xilinx fpgas.
IEEE International Parallel and Distributed Processing Symposium, 2010.
[Hip17] **Hipertextual.**
Samsung explica las causas de las explosiones en las baterías de los galaxy note 7.

[Hu12] **Y. Hu.**

[Huf52] **D.A. Huffman.**

[HYH11] **C. Y. Huang, Y. F. Yin, C. J. Hsu, T. B. Huang, and T. M. Chang.**

[ICC10] **Y. Iskander, S. Craven, A. Chandrasekharan, S. Rajagopalan, G. Subbarayan, T. Frangieh, and C. Patterson.**

[IEE01] **IEEE Computer Society.**

[IEE02] **IEEE Computer Society.**

[IEE08] **IEEE.**

[Inc17] **GitHub Inc.**
Github.


VanderVoord M. Karlesky M. and Williams G. A simple unit test framework for embedded C.


M. Kubica, W. Wrona, and W. Sakowski. Supporting hardware assisted verification with synthesizable assertions.
Design and Reuse, 2012.

A low-cost emulation system for fast co-verification and debug.

Addressing the challenges with soc integration and verification.

[LLR08] Y. B. Liao, P. Li, A. W. Ruan, Y. W. Wang, W. C. Li, and W. Li.
Hierarchy communication channel in transaction-level hardware/software co-emulation system.
2008 Ninth International Workshop on Microprocessor Test and Verifi-
cation, 2008.

Accellerate partial reconfiguration with a 100% hardware solution.
XCell, 2012.

[LWH08] Yi-Ting Lin, Chien-Chou Wang, and Ing-Jer Huang.
Amba ahb bus protocol checker with efficient debugging mechanism.

Fpga level in-hardware verification for do-254 compliance.

Uvm-ams based sub-system verification of wireless power receiver soc.
2014 12th IEEE International Conference on Solid-State and Integrated
Circuit Technology (ICSICT), 2014.

[man09] IEEE standard for SystemVerilog—unified hardware design,
specification, and verification language - redline.
[Mar08] **R.C. Martin.**
*Clean code: A handbook of agile software craftsmanship.*

[MC07] **A. Molina and O. Cadenas.**
Functional verification: Approaches and challenges.
2007.

[Meh14] **A.B. Mehta.**
*SystemVerilog Assertions and Functional Coverage.*

[men09] **FPGA Design Assurance for DO-254 and safety-critical applications.**

[Men10] **Mentor.**
High-level synthesis: Blue book.

[Mes08] **G. Meszaros.**
Test double patterns.

[MGV08] **F. Moya, C. Gonzalez, D. Villa, S. Perez, M.A. Redondo, C. Mora, F.J. Villanueva, and M. Garcia.**
*Desarrollo de Videojuegos: Técnicas Avanzadas.*
Bubok, 2008.

[MI11] **A. Matsuda and T. Ishihara.**
Developing an integrated verification and debug methodology.

[Mic03] **Sun Microsystems.**
Java remote method invocation specification.


Nose.
Nose.

W. Ni and X. Wang.
Functional coverage-driven uvm-based uart ip verification.
2015 IEEE 11th International Conference on ASIC (ASICON), 2015.

OpenCores.
Opencores projects.

Open SystemC.

K. Orland.
Analysis: Xbox 360 poised to pass wii in us sales by year’s end.

A. Patrizio.
Intel plans hybrid cpu-fpga chips.

F. Pardo and J.A. Boluda.
VHDL: Lenguaje para síntesis y modelado de circuitos.
Ra-Ma, 2004.

J. Podivinsky, M. imkova, O. Cekan, and Z. Kotasek.
Fpga prototyping and accelerated verification of asips.

D. Price.
Pentium fdiv flaw-lessons learned.


Uvm based stbus verification ip for verifying soc architectures.
18th International Symposium on VLSI Design and Test, 2014.

Platform for automated hw/sw co-verification, testing and simulation of microprocessors.
2012 13th Latin American Test Workshop (LATW), 2012.

[Sto02] R. Stolzman.
Understanding assertion-based verification.
EE Times, 2002.

The art of agile development.
O’Reilly, 2008.

[Syn16] Synopsys.
Symphony high-level synthesis.
2016.

Dynamic partial reconfiguration manager.

Fpga synthesis can be a leverage point in your design flow.
2009.

Synthesizable assertion checkers in high levels of abstraction.

Dynamic reconfiguration of modular i/o ip cores for avionic applications.
Conference on Reconfigurable Computing and FPGAs (ReConFig), 2012.
A high speed open source controller for fpga partial reconfiguration.

Tackling verification challenges with interconnect validation tool.

Ley de moore.

In-circuit debug of fpgas.

[Xil11] Xilinx.
Axi reference guide (ug761).

[Xil12a] Xilinx.

[Xil12b] Xilinx.
Partial reconfiguration user guide (ug702).

[Xil13a] Xilinx.
Logicore ip axi dma v7.0: Product guide for vivado design suite.

[Xil13b] Xilinx.
Logicore ip axi hwicap v3.0 (pg134).

[Xil13c] Xilinx.
Vivado design suite tutorial: High-level synthesis (ug871).


Implementation of a hardware functional verification system using systemc infrastructure.

[Zer17] ZeroC.

A universal algorithm for sequential data compression.

A novel low-cost interface design for systemc and systemverilog co-simulation.
2015 IEEE 11th International Conference on ASIC (ASICON), 2015.

Practical and efficient soc verification flow by reusing ip testcase and testbench.