Testing framework for on-board verification of HLS modules using grey-box technique and FPGA overlays

Julián Caba[a,*], Fernando Rincón[a], Julio Dondo[b], Jesús Barba[a], Manuel Abaldea[a], Juan Carlos López[a]

[a] *University of Castilla-La Mancha, 13071 Ciudad Real, Spain*
[b] *National University of San Luis, D5700 BPB, San Luis, Argentina*

Abstract

High-Level Synthesis (HLS) provides a simple way to implement complex applications using Field Programmable Gate Array (FPGA) devices. Unfortunately, this technology introduces non-negligible problems related to verification: speed, accuracy and behavior mismatch between co-simulation and implementation.

This paper presents *RC-Unity*, a heterogeneous unit testing framework that integrates FPGA-in-the-loop devices in order to extend the scope and capabilities of current HLS tools. Verification engineers can focus on the design of the tests while the framework automates the generation of the underlying verification infrastructure, making the testbed reusable across different stages of the design flow as the experiments show.

Keywords: Design for testability, testing framework, FPGA-in-the-loop, high-level synthesis, verification overlay

1. Introduction

The evolution of Systems-On-Chip (SoC) requires new design tools to reduce the design gap and make an optimal use of the different processing technologies embodied in the newest devices. It is also necessary to handle the complexity of the hardware design process efficiently [1]. This is the aim of High-Level Synthesis (HLS) tools, which increase the designer's productivity when building specialized hardware accelerators, thanks to the use of High-Level Languages (HLLs), such as C, to describe the algorithms. The use of HLLs is becoming widespread in the hardware realm, since engineers can work at higher abstraction levels when the requirements can be fulfilled without the cumbersome, low-level, manual traditional hardware design process. The main benefits of this approach are higher adaptability to changes, faster design space exploration or shorter time-to-market [2].

Unfortunately, despite these recent advancements in digital hardware design, there is still a significant gap as regards the needs in the area of design verification. Latest studies conclude that hardware verification is the bottleneck in most projects, where up to 70% of design time is spent on those tasks [3], mainly due to the number of logic and functional bugs not detected in pre-silicon validation and the gap between pre-silicon and post-silicon validation targets, where pre-silicon target is a model of the design rather than an actual silicon artifact [4]. If we focus on the HLS development flow, these are some of the problems that need to be solved:

1. On-board verification is ignored by most HLS tools, which only offer a co-simulation environment in order to check the correctness of hardware designs described with an HLL, reusing the initial functional software tests.

*Corresponding author
E-mail address: julian.caba@uclm.es

2. The HLS report is not fully accurate, since it always reports the worst case, which may be an unrealistic scenario. On top of that, most HLS tools are not able to provide timing information for those algorithms that contain undefined bounds in loops.
3. The engineer's experience plays an important role in achieving the best solution. An expert knowledge of the concrete problem may avoid extra design space exploration iterations and error savings through the whole and costly development flow.

Therefore, current HLS tools fail to ensure the design correctness once it is deployed on a real device. Currently, tests must be ported to the deployment infrastructure, which is an error-prone task because of its manual nature, and an ad-hoc verification platform must be built for each development in order to validate the implementation. Moreover, the validation of hardware designs introduces new demands beyond simple checking of functional correctness via the comparison of the outputs against a golden model: for instance, to ensure that the performance parameters (e.g., latency, throughput) of the hardware are within a range.

In this paper, we propose a verification framework based on software testing techniques and reconfigurable computing to validate hardware modules generated using HLS tools. Our proposal considers the verification of a hardware module once it has been deployed on the actual execution platform as part of the process. Thus, we propose a response to the challenges previously identified concerning the validation of specialized hardware accelerators. To achieve the main objective our approach provides the following contributions:

- A single testing framework to validate the behavior of a Design Under Test (DUT) from the HLL description to the deployment, using assertion macros and enabling a grey-box verification strategy to check intermediate results in a hardware design.
- An extension of functional test to consider some physical parameters of the design in the verification flow. By the use of configuration macros, designers are able to configure the hardware environment in accordance with their requirements.
- A library of verification overlays, which are predefined hardware verification environments for several types of DUT interfaces. These overlays save designers from building a custom verification environment for each specialized hardware accelerator.
- DUTs are network accessible. Through a verification service layer based on a remote interface, engineers can exercise the DUTs from their own workstations.

This paper is organized as follows. Section 2 describes the current status in the field of system and component verification using reconfigurable technology. Section 3 introduces the foundations of the proposed hardware testing framework before analyzing in detail the architecture and implementation of *RC-Unity* in Section 4. In Section 5, a qualitative and quantitative study on the impact of the proposed solution is conducted. Finally, Section 6 summarizes the principal outcomes of our approach and proposes directions for future work.

2. Related Work

In-hardware verification of system components is an active area of research in reconfigurable technology [3]. Most of the works reviewed aimed to provide partial solutions to the verification problem, mainly focusing on the development of the testing infrastructure once the IP has been implemented. Therefore, works such as [5] [6] and [7] propose several FPGA-based testing platforms, with special emphasis on the communication infrastructure. However, unlike our approach, these solutions are quite dependent on the underlying ad-hoc architecture, making it difficult to reuse them in future projects.

Some works, such as [8] and [9], follow a more holistic approach to the verification challenge of FPGA-based systems, and offer solutions that go beyond

the mere implementation level, extending the functionality of the testing platform and spanning across more than one design abstraction level. For example, [8] leverages the use of high-level artifacts already present in System Verilog and SystemC, allowing the functional verification of the design through co-simulation, in a pure software domain, and then verified through co-emulation after the implementation of the hardware part in a specific hardware emulator.

The emulation strategy, as stated in [10], can also support the verification of FPGA designs at different concretion levels (e.g., bus functional models versus the actual system bus) for both the DUT and other system components. However, this strategy implies an important effort overhead when it comes to the development and maintainability of the testing framework.

The solution presented in this paper shares the UVM (Universal Verification Methodology) vision, which advocates for a neat separation between the generator of the input test vectors (stimuli) and the verification environment. This means that the interface between the DUT and the rest of the testing infrastructure is kept unmodified, regardless of the stage of the design, making the testbed reusable. Some works, such as [11] and [9], propose in-hardware verification environments inspired by the principles of UVM. The automated generation of the environment, an interesting feature which is also supported by our solution, is also proposed by Podivinski et al. in [12]. Nevertheless, UVM is a complex verification methodology that requires experienced verification engineers to build verification environments. Consequently, several works such as [13] and [14] pursue the simplification of the operational effort of UVM-inspired solutions by promoting the reuse of the tests through the entire verification flow, independently of the abstraction level. This test reusability feature is included in our solution.

One of the major challenges of in-hardware verification of HLS-generated designs is to provide a means with which to increase signal visibility beyond the barrier that represents the top-level function or entry point to the design. Therefore, the design must be instrumented so that the designer can actually see what is happening behind the scenes. Goeders et al. describe in [15] a comprehensive hardware monitoring system that traces a set of signals once the HLS circuit has been synthesized. Even though the functionality of the monitoring infrastructure exposes interesting features, and it is highly optimized, instrumentation at such a low level handicaps the understandability and traceability of the root cause of the problem back to the high-level model. For this reason, [16] and [17] propose a monitoring strategy at HLS level, allowing an easier method to identify when and where a mismatch between the expected and reference values occurs. However, the price to be paid comes in the form of resource and latency overheads as a consequence of the modification of the original HLS code, compared to HDL-based monitoring methods. [16] performs worse in this aspect, since it forces the inclusion of memories and logic to store and retrieve signal values. In contrast, [17] follows a less intrusive approach to identify performance bottlenecks without the need for bitstream generation, although a cycle accurate solution is possible. [17] uses source-to-source transformation techniques to add tiny monitors inside the DUT that measure and store the time of nested modules to obtain cycle-accurate timing results. Our approach differs from both [16] and [17] in the avoidance of the use of internal memories to monitor the behavior of the DUT. Thus, the probes are driven outside the component through an adapter, so an external component interprets these probes and determines the action to perform: continue to run the execution or stop it.

Finally, it is worth mentioning a family of works devoted to supporting Assertion-based Verification (ABV) for reconfigurable designs through the implementations of HLS techniques to efficiently support in-circuit assertions [18]. These works extend the functionality of the hardware monitors in order to perform on-line value checking of the overseen variables. Memory requirements are reduced, since only the registration of the mismatch events is needed.

3. *RC-Unity* Testing Framework

In order to lead the whole verification process by means of a single testing framework, *Unity* is pro-

posed, due to its simplicity and expandability [19]. *Unity* is fairly portable across unalike platforms, such as 8-bit microcontrollers or 64-bit processors, since it is written in ANSI C. *Unity*, like most testing frameworks, provides a variety of assertions that can be placed along the test to verify the production code.

For instance, the `TEST_ASSERT_EQUAL(expected, actual)` macro checks the equality between two values: `expected` and `actual`. Therefore, designers can use a set of assertions provided by testing frameworks in order to check the correctness of all functions that contain their projects by comparing output values with reference ones [19].

Following the testing frameworks typically used for the validation of software programs, and the fact that HLS is very widespread in the industry [3], it is interesting to apply testing framework facilities to HLS design of hardware accelerators, hence a testing framework adaptation process is necessary to meet the new requirements resulting from the nature of hardware developments. Unlike current testing approaches in HLS tools, software testing frameworks allow the checking of individual functions, providing a better understanding of the insight of the design. Note that functions might be called by other functions, building a calling hierarchy. Our solution, *RC-Unity*, includes some facilities that allow designers to verify hardware designs through the top-level function, establishing different checkpoints and retrieving the intermediate values generated by the system.

3.1. Verification flow

Figure 1 represents an overview of the three stages or domains described below. The main key is to separate the tests from the final implementation of the DUT in order to achieve a double goal: to reuse part of the original tests, adding a configuration for on-board testing, and to abstract the complexity of the digital hardware design. The test runner is responsible for executing the collection of the tests, but with the particularity that it does not directly interact with the DUT when it is running on an FPGA. Instead, the test runner uses a virtual agent of the DUT to bridge it with the tests (see Figure 1).

Software verification stage. The verification flow starts from the high-level description of a hardware design, exercising the top-function and the nested ones, or in other words exercising the Design Under Test (DUT) from the top-function. Although nested functions can be tested individually due to the software flexibility, it is not recommended, because the final component generated has an interface in accordance with the top-function. Sections 3.3.1 and 3.3.2 propose a solution to overcome this restriction with small components that minimally alter the original component, as the case studies section shows. In this stage, the *RC-Unity* uses the assertions of *Unity* because designers actually check the C code.

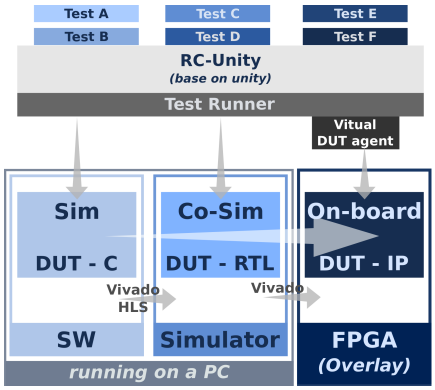

Figure 1: Big picture of our proposed approach

Co-simulation stage. Once the high-level description of the hardware design is tested in a purely software domain, an HLS tool translates the high-level code into an RTL description. In this work we use Vivado HLS from Xilinx. At this stage, designers can run the tests previously defined to check the top-function (software tests) over the RTL model without any change, but now the DUT is running on a simulator (*co-simulation stage*).

On-board verification stage. The last stage should be to verify the hardware design on the FPGA but, unfortunately, designers must manually build a custom architecture around the DUT in order to check the correctness of the hardware design. However, in this paper, we propose an automated solution for these tasks. It is based on a library that is composed of different overlays (see Section 4), which can be configured and managed during the testing process

Table 1: Macros of the *RC-Unity* testing framework

Macro	Description
`RCUNITY_RESET`	Sets the DUT to a well-known state
`RCUNITY_CONF_CLK_EN(time)`	Configures the time available to perform a operation. By default 0 (it means no limit of time)
`RCUNITY_START`	Enables the DUT during the time depicted in `RCUNITY_CONF_CLK_EN` macro
`RCUNITY_CONF_CLK_RATE(rate)`	Configures the clock rate to perform a operation. By default 100 MHz
`RCUNITY_CONF_START_FLAG(num)`	Modifies the moment to start the internal counter that measures the time elapsed by a operation. By default 1
`RCUNITY_CONF_STOP_FLAG(num)`	Modifies the moment to stop the internal counter that measures the time elapsed by a operation. By default 1
`RCUNITY_CONF_FU(num)`	Modifies the moment to start and stop the internal counter in accordance with the FU under test
`RCUNITY_CONF_BREAKPOINT(selID)`	Selects the explicit breakpoint to stop the DUT task. By default 0 (none selected)
`RCUNITY_CONF_FREEZE(bp)`	Freezes the DUT when the `bp` breakpoint (ID of breakpoint) happens. By default is deactivated
`TEST_ASSERT_TIME_XX(expected)`	Compares the time obtained and expected value in accordance with the comparison operator

from the testbed. That means that test cases configure the hardware environment before exercising the DUT (see Table 1).

Thus, a testing tool (`dut_testing`) is provided to test the design in a particular domain, so that designers can use it to check their designs automatically by executing the `dut_testing` tool with the `sim cosim fil` options, which means a complete testing passing through the three verification stages: software, co-simulation and on-board.

3.2. On-board Verification

At this stage we introduce some novel contributions that facilitate the configuration tasks, extending the *Unity* testing framework with the inclusion of a number of new configuration macros (see Table 1). For instance, the tests can annotate the physical parameters, such as the speed rate to be used or the number of cycles that the clock enable must be set in order to configure the proposed hardware verification platform. Moreover, the *RC-Unity* framework allows engineers to retrieve information about the latency of the execution of a test. These physical annotations are the main difference between the in-hardware domain and the other two domains (pure software domain and co-simulation domain).

Listing 1 shows an example of a test in the *RC-Unity* framework. Firstly, designers configure the verification overlay through the configuration macros: they set the clock rate for the DUT, so here one is able to observe the DUT behavior under overclocking/underclocking conditions, and the number of cycles that the clock enable will be active (lines 4 and 5 of Listing 1). Configuration macros are lazy operations, in the sense that they take effect after the `RCUNITY_START` macro is executed (line 6 of Listing 1). Once the configuration annotations are executed, the DUT might be set to a well-known state with the `RCUNITY_RESET` macro (line 7 of Listing 1). At this moment, verification engineers can exercise the DUT as a black box by just invoking the top-function (line 9 of Listing 1). This invocation is in fact a fake local invocation, because the DUT is running on one of the three possibilities explained in Figure 1, hence the invocation bridges both domains. Finally, designers can assert the time elapsed during the test matches with the expected one (line 11 of Listing 1) and they can assert the equality between the output and the reference values (lines 13 and 14 of Listing 1). The values of both operations are retrieved from the on-board verification platform.

Listing 1: Example of a test within *RC-Unity*

```
1  void
2  test_module(){
3  #ifdef __SYNTHESIS__
4    RCUNITY_CONF_CLK_RATE(100); // 100MHz
5    RCUNITY_CONF_CLK_EN(200); // 200 cycles
         active-high
6    RCUNITY_START(); // Configure HW platform
7    RCUNITY_RESET(); // Reset Module
8  #endif
9    result = moduleDUT(stimuli)
10 #ifdef __SYNTHESIS__
11   TEST_ASSERT_TIME_LT(750); // Checking time
12 #endif
13   for(int i=0; i!=16; i++)
14     TEST_ASSERT_EQ(reference[i], result[i]);
15 }
```

At this point, designers are able to verify their hardware components using the *RC-Unity* framework. Therefore, the tests check the complete design as a black box in any of the target domains: software, co-simulation or in-hardware. However, it is usually interesting to also check the behavior in different internal points of the design, although that should not imply changes in the top interface nor final architecture. The following sections describe how our solution overcomes this challenge.

3.3. On-board verification using grey-box strategy

HLS engineers typically code their hardware designs into a top-function, which usually contains some nested ones. The functionality of this kind of hardware design is usually verified as a unique module (black-box verification), which makes it difficult in case of failure to identify the source in the code. Nevertheless, each function that contains the top-function must be checked individually to delimit possible bugs. Thus, these functions are seen as black boxes, whilst the overall architecture at block level is now known by engineers (grey-box verification).

3.3.1. Using implicit breakpoints

Following the translation process done by HLS tools, nested functions are mapped into functional units (FUs) and are linked between them to obtain the desired behavior. In the software domain, engineers are able to check the behavior of each nested function individually. In a co-simulation, domain grey-box verification strategy implies manually changing the top-function to choose the correct nested-function under test and select the appropriate test case(s). This is even worse in the hardware domain, since engineers do not have an easy way to check the behavior of each FU that composes a hardware design, nor a way to increase their visibility, since that requires the use of debug artifacts that may not appear in the final release of their hardware modules, which leads to building custom hardware projects.

To overcome this challenge, we propose a transparent and automatic grey-box strategy, which is based on the modification of the control data flow graph (CDFG) of the original hardware design, transparently instrumenting the developer's code in order to include breakpoints between the FUs that comprise the hardware design. Thus, the `dut_testing` tool (executed with the option `--grey`) reads the top-function from a C code and instruments it.

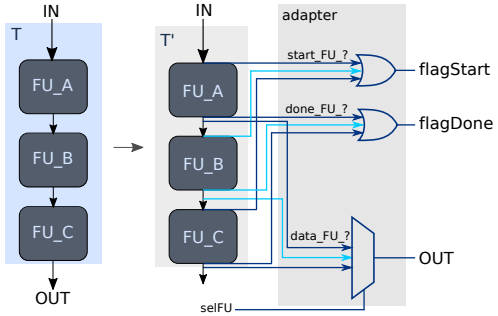

Figure 2: Original DUT vs Modified DUT

This instrumentation consists of duplicating the output path of each sub-block (or FU) to be tested, and routing that output to the module interface and to the next FU in order to not alter the original path (see Figure 2). Therefore, our proposal exercises the DUT from the original entry point, but the output point changes in accordance with the FU under test. In addition, `start` and `done` control flags for each FU are added to observe the execution of each one, which means the point being executed at a given time. The `start` signal is set one cycle before executing the FU's tasks, whereas the `done` signal is enabled for one clock cycle after the FU has finished its tasks. Figure 2 shows an example of our approach applied in a top-function that contains three nested-functions (`FU_A`, `FU_B` and `FU_C`); note that each FU depends on the result of the previous one. Thus, to verify the intermediate `FU_B` block, some extra code to bridge the output of `FU_B` block to the output point (right side of Figure 2) may be included.

To manage the extra signals in the hardware design, an adapter is generated by the `dut_testing` tool during the instrumentation phase. This adapter groups the control signals that will be routed to a hardware component, whose aim is to manage the verification process (*Test Manager* component, see Section 4). The output values are serialized and sent

to the original output interface. Figure 2 (right side) illustrates an overview of the communication between the modified DUT (`T'`) and the generated adapter. The `start` and `done` flags are grouped by `OR` gates, while the data signals (the output data of each FU) are selected by a multiplexer in accordance with the `selFU` signal, which is controlled by the *Test Manager* component. Note that the adapter is more complex, and the figure just shows an abstraction of its behavior.

Listing 2 shows a new test, which delimits the verification, checking the `FU_B` block instead of the whole DUT. In accordance with Figure 2, light blue arrows are analyzed during the execution of the test shown in Listing 2, while the rest of the arrows are ignored and not taken into account for this test. The configuration macros are similar to the previous test (Listing 2), but it includes the `RCUNITY_CONF_FU` macro. This macro denotes the FU to be verified, which corresponds to the second FU (line 5 of Listing 2), so the test environment is configured to select the appropriate FU (`selFU` is set with the number two). Then, the hardware design is exercised by the test, using the same stimuli that are used to check the whole design (line 8 of Listing 2). After that, the test asserts that the time elapsed by the second FU matches with the expected one. Further, we can use `RCUNITY_CONF_START_FLAG` and `RCUNITY_CONF_STOP_FLAG` macros with values 1 and 2 to measure the time consumed by `FU_A` and `FU_B` blocks respectively, thus retrieving in this way the latency from the beginning of the stimulation to the end of the `FU_B` block execution. Finally, the test compares results related to the `FU_B` block, with some golden intermediate values (line 10 of Listing 2).

Listing 2: Example of a test for `FU_B` (using *RC-Unity*)

```
1  void
2  test_FU_B(){
3    RCUNITY_CONF_CLK_RATE(100); // 100MHz
4    RCUNITY_CONF_CLK_EN(200); // 200 cycles
         active-high
5    RCUNITY_CONF_FU(2); // Set 2nd FU under test
6    RCUNITY_START(); // Configure HW platform
7    RCUNITY_RESET(); // Reset Module
8    result_FU_B = moduleDUT(stimuli)
9    TEST_ASSERT_TIME_LT(50); // Checking time
10   TEST_ASSERT_EQ(reference_FU_B, result_FU_B);
11 }
```

In summary, to perform this new hardware verification strategy we consider the same stimuli that we use in a complete testing, but by using intermediate results that can be obtained from the non-top-function software tests. For instance, the example shown in Figure 2 can be checked as a whole design, exercising it from the input of `FU_A` function and checking the output of `FU_C` function, but it also can be checked in two intermediate points: at the end of `FU_A` and at the end of `FU_B`. Thus, the design is checked using three different tests that retrieve information from three different internal points. In addition, the `RCUNITY_CONF_XX_FLAG` macros allow measurement of the time elapsed by an individual FU or a chain of them.

Unfortunately, our proposal contains an important restriction. It is limited to those HLS hardware accelerators whose chain of FUs is a Directed Acyclic Graph (DAG). A DAG contains some vertices directed by edges, but there is not any path to start from a particular vertex and return to that vertex. However, DAGs are popular models in practice, with applications in machine learning and casual inference, which are being taken as a reference for hardware acceleration [20].

3.3.2. Using explicit breakpoints

A number of hardware designs are hard to be divided into nested functions using HLS, so designers build a single function that implements a large percentage of the behavior of the overall design. This function plays the role of the top-function required by the HLS tools. In order to provide a solution similar to the previous scenario, we propose the use of explicit breakpoints instead of the implicit ones associated with the FUs of a hardware design. Now, breakpoint sentences can be added in the middle of the top-function, using a simple signature: `rcunity_breakpoint(selID, ID)`, where *selID* is set from the test case using the `RCUNITY_CONF_BREAKPOINT(ID)` macro. When the `selID` and the `ID` values match, the execution is halted if the Test Manager is configured to freeze the execution (the `RCUNITY_CONF_FREEZE` macro configures the freeze feature). Thus, a single explicit breakpoint can be activated for each test execution

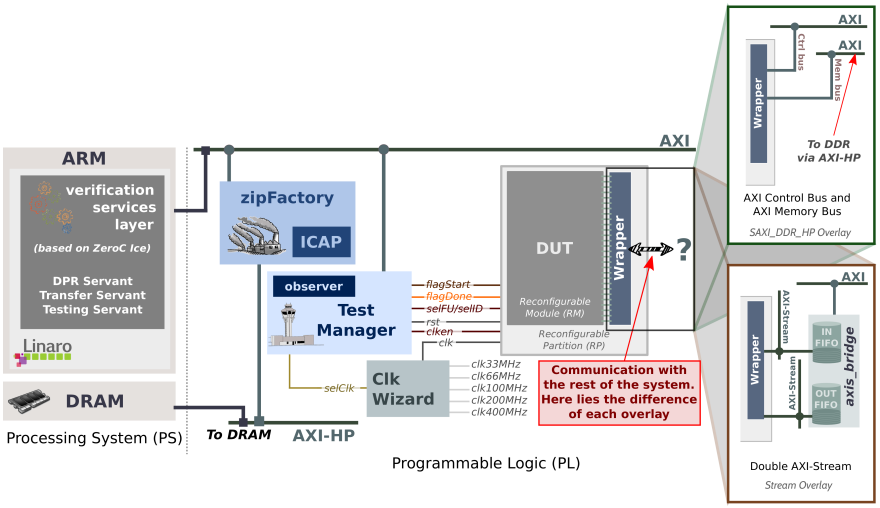

Figure 3: Overview of the PS and the PL part of verification overlays

(by default if there is no selected one). Explicit breakpoints include a heavy penalty, because the CDFG is severely modified when data is routed from the breakpoint location to the output channel, so this task is left to designers.

However, this solution allows measurement of the time elapsed between breakpoints. At the beginning of the developer's code a `start` signal is set for a clock cycle in order to denote the beginning of the DUT's execution. Then, each explicit breakpoint sets a `done` signal to flag the finishing of a block (a group of operations) and another `start` signal to denote the beginning of the next group of operations, regardless of the selected breakpoint. Thus, the *Test Manager* component is able to measure the time elapsed after executing a group of operations.

Explicit breakpoints also provide another benefit to those designs that work with a third-party component, such as a DRAM. Designers may be able to stop the DUT with explicit breakpoints and retrieve information from that third-party component by inferring the correctness of a hardware design at a particular point, as the second case study shows (see Section 5.2).

4. Verification Platform with Overlays

Xilinx defines an overlay as a pre-compiled FPGA design that can be downloaded to the Programmable Logic (PL) part of a hybrid FPGA, along with some software that controls the cores deployed in the PL to accelerate a software application. Thus, an overlay can be loaded to an FPGA like a software library. The overlay approach was initially introduced by Xilinx in their PYNQ boards, such as PYNQ-Z1. PYNQ provides a Python layer to facilitate the use of FPGA-based solutions that require hardware engineering knowledge and expertise [21]. This concept can also be adopted to other Zynq architectures and HLLs, such as the ZedBoard platform and the C programming language, as shown in the experimental evidence of this work.

Following the Xilinx approach, we provide a collection of verification overlays (see subsection 4.1) to allow FPGA-based designs to be verified from the *RC-Unity* framework running in the Processing System (PS), avoiding the manual building of a custom hardware verification platform. Verification overlays share the same architecture but differ in the communication channels between the DUT and the rest of the system, as shown in Figure 3. Furthermore, a communication layer based on ZeroC Ice middle-

Table 2: Available overlays in *RC-Unity* testing framework

Name	AXI-S		AXI-4		FIFO	
	Init	Target	Slave	Master	Rd	Wr
SAXI	0	0	1	0	0	0
Stream	1	1	0	0	0	0
FIFO	0	0	0	0	1	1
SAXI_MAXI	0	0	1	1	0	0
SAXI_Stream	1	1	1	0	0	0
SAXI_DRAM_HP	0	0	1	1	0	0

ware [22] has been included alongside special services in the PS part in order to turn our solution into a remote verification platform

4.1. Overlay layout

Figure 3 (right side) illustrates the block design of the architecture layout of our verification overlays. This block design view abstracts the interface of the DUT in order to show the common PL part, which shares the overlays that include our *RC-Unity* framework. This PL part is composed of a dynamic area where hardware accelerators are instantiated through the *zipFactory* component, whilst the *Test Manager* component configures the verification platform and manages the verification process using high-level functions.

4.1.1. Hardware accelerator interfacing

Unfortunately, the communication channels are too wide and varied, from a custom interface to a standard one, making it difficult to obtain a single solution for all interfaces. Due to the range of core interfaces, this work is focused on those interfaces that match with a standard bus, such as *AXI-Stream* or *AXI-HP*. Thus, the *RC-Unity* framework includes a number of available overlays (see Table 2) to allow designers to verify their hardware designs.

Table 2 shows the overlays that contain the *RC-Unity* framework, denoting the number of each kind of standard bus included. For instance, the *SAXI_Stream* overlay can be used in those hardware accelerators that contain an interface composed of an *AXI-Stream* input and an *AXI-Stream* output and an *AXI-4* slave. Most hardware accelerators use this interface, where the *AXI-4* slave interface configures the core itself, whilst both streaming channels are used to read and write the data that will be transformed. The streaming channels, or those overlays that contain it (*SAXI_Stream*, *Stream* and *FIFO overlay*), involve a previous task before injecting the data to the DUT: loading the data into temporary buffers. Thus, both *SAXI_Stream* and *Stream* overlays translate the data sent through the *AXI* bus to the *AXI-Stream* standard protocol as Figure 3 (top-right side) shows, while *FIFO* overlay does not follow a standard to forward the data to the DUT. In addition, Figure 3 (bottom-left side) illustrates the *SAXI_DRAM_HP* overlay, which contains two *AXI* channels: one for the control commands and the other to connect the DUT and the DRAM via *AXI-HP*.

The signal names of the DUT interface assigned by designers could mismatch with the signal names given by the overlay, it may be that even the reset signal must be negated according to the DUT reset's polarity. Therefore, a signal adaptation task is performed automatically, comparing the interface signals generated by the HLS tool to the DUT's overlay signals. The result of this task is a wrapper that is included as a source code to generate the configuration file properly (see Figure 3).

4.1.2. Loading hardware accelerators

Hardware accelerators or DUTs are loaded into the reconfigurable partition (RP) of the corresponding overlay, playing the role of a reconfigurable module (RM). Therefore, the PL layout of overlays is divided into two parts: a static one which contains those components that remain unchanged regardless of the DUT and the DUT itself. This feature is known as Dynamic Partial Reconfiguration (DPR) [23] and brings a major benefit to our solution: *the saving of synthesis time*. Since overlays are provided as reference projects with a number of checkpoints, the configuration file is generated from these pre-synthesized points, using several TCL scripts. Thus, the scripts open the synthesized checkpoint of the corresponding reference overlay and include the RTL logic of the DUT.

First, and before run tests, the configuration file related to a DUT must be loaded into the available RP of an overlay, so overlays require a reconfiguration engine to perform the DUT loading process. This task is done by a custom component called *zipFactory*

(see Figure 3). The loading process is carried out in two parallel steps: 1) data from the configuration file must be stored in a memory address of the DRAM; 2) the *zipFactory* component retrieves the data with an internal DMA, which is immediately forwarded in 32-bit words that match with the data bus width of the ICAP (32-bits). In addition, the component knows the type of the 32-bit word sent to the ICAP at each transaction; this fact allows the *zipFactory* to recognize the desynchronization command and finalize the reconfiguration process by attaching some NOPs to flush the command pipeline properly.

4.1.3. Management of the hardware verification process

The overlay's PL layout includes another component called the *Test Manager* (see Figure 3), whose aim is to perform a variety of hardware tasks that are not feasible from software or may result in poor accuracy when executed from software programs. Each hardware task is managed from the software testbench, using the macros listed in Table 1.

- It resets the RP's components in order to assure that the DUT starts from a well-known state.
- It manages the `clock enable` signal during the cycles indicated in the macro related to this task. The `clock enable` signal is active-low until `RCUNITY_START` is invoked. To active-high this signal all the time, a value of 0 must be set.
- It sets the clock rate of the RM and those modules that interact with it. The available clock speed rates are: 33 MHz, 66 MHz, 100 MHz, 200 MHz and 400 MHz.
- It measures the time elapsed at each FU or between consecutive FUs. The *Test Manager* looks at the `start` and `done` signals of those FU(s) that are under test and determines the number of cycles to complete an operation between two internal DUT points.
- It configures the DUT's data path in order to route the return values of an FU to the output point, and enables the explicit breakpoints to halt the execution of the hardware component.

4.2. Verification Services Layer

RC-Unit overlays provide a remote verification service running on the PS part, as shown in Figure 3 (left side). This service adds a new layer to allow overlays in the PL to be controlled remotely. Thus, a Linaro OS, which is a GNU/Linux distribution based on Ubuntu and tailored to embedded systems, is running on the ARM processor with our verification service layer, which has been incorporated into the OS as a Linux daemon.

The remote verification service is divided into three servants, based on ZeroC Ice [22]. ZeroC Ice is an object-oriented Remote Procedure Call (RPC) framework that provides a number of facilities for building network-based distributed applications with an abstraction layer that hides network communication issues in order to allow designers to focus on application logic. Thus, our overlays are compatible with the programming languages supported by ZeroC Ice, or in other words, another software testing framework, such as *unittest* from Python, which could be extended to verify hardware designs, thanks to the fact that our solution takes advantage of the properties of ZeroC Ice to build a hardware-software communication solution that is easy to use due to the high-level abstraction of the API [24]. Since our customized distributed solution preserves the location and access independence properties, the whole process can be performed either locally or remotely (i.e., via Ethernet).

Transfer : stores a file in the DRAM memory space.

DPR : runs the reconfiguration process, using the configuration file stored in the DRAM memory space, hence, before executing it, a properly configurated file must be available in the DRAM. This servant increases the availability and accessibility of the verification overlays, avoiding, for example, the need to configure the PL using a complete overlay (usual use case).

Testing : provides the necessary functionality to exercise the DUT and to configure the *Test Manager* component, both located in the PL part.

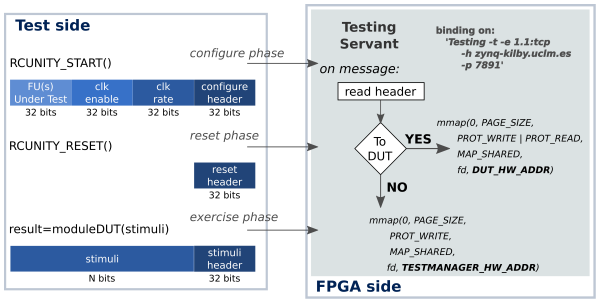

Figure 4: Communication between Tests and FPGA (*Testing servant*)

The services offered by the Verification Services Layer (VSL) are based on a set of routines that translate high-level invocations into ZeroC Ice messages, which in turn are embodied into TCP/IP packets. Thus, serializing and deserializing data for network transmission is performed transparently by Ice; tests run remotely on the developer's workstation, whilst the DUT runs on an FPGA platform connected to the network. The macros of the *RC-Unity* testing framework are implemented over the VSL, as well as the top-function when on-board verification is performed. Figure 4 shows a communication example: the arguments of macros and functions are serialized as a data stream and sent using the VSL; when the FPGA receives the streaming of data, the *Testing servant* deserializes the data and exercises the DUT or the *Test Manager* via `mmap`.

5. Case Studies

In this section, two case studies are presented. The first is based on the design of an IP implementing the Histogram of Oriented Gradients (HOG) feature descriptor algorithm and the second is based on the development of an accelerator for the Dijkstra shortest path algorithm for graphs. Each case study aims to highlight how the limitations of the current tools and verification flows can be overcome by the adoption of our solution. The HOG use case shows how *RC-Unity* can be used to get accurate latency measures for the different sub-modules that conform the architecture of the IP. This can be done by the using of

```
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);
}
```

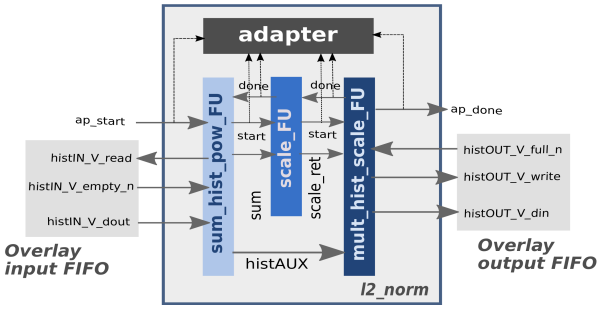

Figure 5: l^2-*norm* signature and block diagram overview after using *RC-Unity*

implicit breakpoints and the automation of the generation of the necessary infrastructure, which avoids the need for the high-level model of the HOG component. The second use case, *Dijkstra*, demonstrates how *RC-Unity* can be used to measure the elapsed time between two points in the component functionality, providing a finer grained view of the internals of the implementation. This use case also exemplifies the use of explicit breakpoints and the impact on resources overheads and clock frequency variation.

For the implementation of the testing environment, version 2017.4 of Xilinx's development toolchain (Vivado HLS & Vivado) has been used in these experiments, running on top of a GNU/Linux operating system. The target platform in both cases is a Xilinx ZedBoard working at 100 MHz. However, the clock speed of the RP and the components related to the DUT can be modified at runtime.

5.1. Case Study A: Histogram of Oriented Gradients

The HOG is a feature descriptor used in computer vision and image processing for object detection; it is particularly suited for human detection in images.

Table 3: Estimated (C), simulated (RTL) and actual (*RC-Unity*) timing comparison of l^2-*norm*

FU	Vivado HLS	Co-Simulation	Overlay
sum_hist_pow	146 cycles	146 cycles	145 cycles
scale	26 cycles	26 cycles	24 cycles
mult_hist_scale	81 cycles	83 cycles	80 cycles

The algorithm implementation is divided into different steps (i.e., gamma and color normalization, gradient computation or block normalization). In our case study, the step chosen is the vector normalization block with l^2-*norm* normalization factor [25]. The size of the window is 4x4 pixels.

Figure 5 shows the HLS top-function of the l^2-*norm algorithm* in C language. The *pragma* directives drive the HLS synthesis process in such a way that the input and output arrays are mapped to FIFO modules in order to match the required overlay interface. In addition, Figure 5 depicts the block diagram of the generated IP, after applying *RC-Unity's implicit breakpoint* approach (see subsubsection 3.3.1). Each FU corresponds to a processing stage in the original dataflow, plus the adapter. The synchronization signals between the stages (start/end) are split and driven to the input of the adapter, which represents the minimum level of interference in the original design. The adapter uses these signals to register the latency of each submodule in its internal registers, which are accessed through the interface with the overlay (not represented in Figure 5).

Table 3 shows a comparison of the latency measurements (in cycles) for each of the stages of the l^2-*norm algorithm*. A variation between the estimation made by the HLS tool can be observed, the result of the co-simulation of the synthesized RTL model and the on-board values provided by *RC-Unity*. Our *overlay* is the most accurate because it does not use an estimation to infer the time that takes an operation. *RC-Unity* works with synthesized designs instead of simulation models.

5.2. Case Study B: Dijkstra's algorithm

Dijkstra's algorithm is a search algorithm that finds the shortest path between two nodes of a graph. This algorithm is used in different domains, such as robotics or geographical maps, to find the optimal route or networking routing protocols, such as linkstate routing.

The reference implementation of the *Dijkstra's algorithms* makes use of a variety of data structures whose implementation directly influences the performance of the hardware component generated by the HLS tool. For example, a priority queue is used to maintain the collection of nodes to be visited. A priority queue can be instantiated either by using an ordered set or as a binary heap. Although both approaches are valid, they have an impact that it would be helpful to measure.

Our proposal allows the reuse of the verification environment, so that only the DUT needs to be resynthesized, thus saving time and effort. This is possible when the interface of the component remains invariant. In the case at hand, the component must access an external memory (DRAM), since the memory requirements for the graph and auxiliary variables exceed by far the in-chip capacity of the FPGA. In the HLS model, this is represented by the mapping of the memory pointers to an *AXI* bus.

Listing 3: Example of use of explicit breakpoints in *Dijkstra* HLS model

```
1  HLSTL_MW_T dijkstra(...,
2             RC_UNITY_AXIS_OVERLAY_IFACE()) {
3  ...
4   while (vit_first != vit_last) {
5    Hls_Vx_It_get(mem_graph,vit_first,&e);
6    Hls_Array_get(mem_visited,&visited,
7                  e._tgt_vx,&done);
8    //Example breakpoint #1
9    rcunity_breakpoint(1);
10   if (done == HLSTL_FALSE) {
11     ...
12   }
13   //Explicit breakpoint #2
14   rcunity_breakpoint(2);
15   Hls_Vx_It_inc(&vx,&vit_first);
16  }
17 ...
18 }
```

Therefore, the use of a *SAXI_DRAM_HP* overlay and explicit breakpoints (see subsubsection 3.3.2) are recommended in this case study. The code in Listing 3 exemplifies the use of this feature by the designer. First, the adaptation of the top-level function interface is done by means of an *RC-UNITY* macro (line 2 of Listing 3), which adds the necessary signals, de-

Table 4: SW and *RC-Unity* timing analysis of *Dijkstra's algorithm*

Timer source	1^{st} Breakpoint	2^{nd} Breakpoint	Total
PMU	N/A	N/A	67489 ns
Overlay	37 ns	560 ns	63670 ns

pending on the type of overlay used. Then, the designer inserts as many breakpoints as necessary using the library function *rcunity_breakpoint*) (lines 9 and 14 of Listing 3). The argument to this function is a number that unequivocally identifies the breakpoint and is used later to access the right timer counter from the software.

The use of explicit breakpoints is of special importance, given the nature of the algorithm. The latency of a path-finding operation is not known beforehand, since it depends on a series of attributes (i.e., graph size, average degree of the vertices, etc.). Also, the RTL simulation of the model is not feasible in this case, due to the limitations of the co-simulation environment; there is a maximum size of the test input vector that exceeds the size of the graphs by far (in order of millions of nodes and vertices).

In this context, only when the IP is deployed in the FPGA is it possible to get some timing feedback through the use of either software or hardware timers, if they are present in the platform. However, these observations are not error-free because of timer resolution and software overheads. The last column in Table 4 shows a comparison of the measured latency for one execution of the *Dijkstra's algorithm* obtained from: (a) the internal timer of the ARM processor (PMU); and (2) *RC-Unity*, using explicit breakpoints. As the reader can see, there is a significant variation between the time measured with the PMU and our approach (3819 cycles), which represents an error of 5.9%.

Not only does *RC-Unity* accurately measure the time elapsed between the start and the end of the component operation, but it also provides a means with which to perform (fine-grain) hardware profiling, which is impossible using only software timers. In this use case, two explicit breakpoints were placed: the first one was located after the initialization phase of the local variables and the second one once the initial node was visited. The second and third columns in Table 4 show the intermediate completion times obtained using explicit breakpoints.

The proposed framework is highly flexible and the level of granularity, as well as the minimum interval to be measured, are only restricted by the minimum set of operations between explicit breaks. Without *RC-Unity*, platform timers are the instrument to perform such types of profiling. Nevertheless, this mechanism is only applicable at the DUT level, which prevents getting segmented statistics of the operation of the component. Also, platform timers introduce uncertainty, since their operation is affected by external factors such as operating system overload, interruption handling overheads, etc.

5.3. Resource Overhead and Critical Path Penalty

RC-Unity instruments the original HLS model so that the developer can introspect and monitor the hardware component. In order to analyze the impact of the extra logic introduced by this proposal, the increase in resources and the maximum working frequency have been measured.

In the first use case, HOG algorithm and implicit breakpoints, the use of resources has been broken down by resource type and the comparison of results is made before and after the Place and Route stage; that is, the estimated values provided by Vivado HLS tools and the actual figures on board. In Table 5 it can be observed that, concerning the latency overhead, our approach introduces one cycle per FU present (three clock cycles in total) in the design due to the way the start and done signals are managed. On the contrary, *RC-Unity* does not significantly affect the clock frequency, preserving the initial timing restrictions. In this experiment, the verification platform includes the minimum elements to support the basic functionality of the proposed verification platform, namely: *FIFO* overlay. Our solution reduces roughly 100% of the BRAMs and 96% of the LUTs used, by employing the approach proposed by Y. K. Choi et al. in [17]. Unfortunately, the FFs resources are not provided in [17].

As to the second use case, the *Dijkstra's algorithm* and explicit breakpoints, Table 6 shows the extra de-

Table 5: Estimated and actual overheads for the (l^2-*norm algorithm*) introduced by *RC-Unity's implicit breakpoints*

	Vivado HLS (estimated)		P&R (on-board)	
	Initial	RC-Unity	Initial	RC-Unity
BRAM	0	4	0	1
DSP	13	13	13	13
FF	1316	2116	1107	1778
LUT	1928	2415	1095	1607
Latency[1]	255	258	249	252
Max. Freq.[2]	123	114	121	109

[1] In clock cycles. [2] In MHz.

Table 6: Extra demand of resources by explicit breakpoints

Number of breakpoints	LUTs	FFs	LUTs (%)	FFs (%)
1	9	100	0,02%	0,09%
2	17	136	0,03%	0,13%
3	21	137	0,04%	0,13%
4	28	138	0,05%	0,13%
5	72	141	0,14%	0,13%
6	133	102	0,25%	0,10%
7	170	143	0,32%	0,13%
8	174	145	0,33%	0,14%
9	188	155	0,35%	0,15%
10	191	252	0,36%	0,24%

mand of resources for an increasing number of breakpoints; up to ten checkpoints were embodied in the HLS model. The figures correspond to the actual usage of LUTs and FFs, after the Place and Route of the design for a Xilinx ZC702 prototyping board. In absolute (first and second columns) and relative (third and fourth columns) terms, a steady lineal growth with a variable gradient can be observed, due to the effect of the optimizations performed by the synthesis tool.

The analysis of the clock period after synthesis for the *Dijkstra IP* concludes a variation of ± 0.45 ns compared with the original design ($T = 9,01ns$) with no breakpoints.

6. Conclusion

This paper has introduced a hardware testing framework, *RC-Unity*, for in-hardware verification of the components developed using HLS. This framework embraces both functional and timing planes and is well-suited for software or hardware developers. The proposed verification platform is customized through the use of a variety of configuration/control macros that tweak physical parameters such as the operating clock frequency.

Focusing on the utilities provided by *RC-Unity*, verification engineers can find some macros to accurately measure time, at a variable level of granularity, and other macros to enable a level of introspection not available with commercial tools. Thus, the output of the different FUs, or sub-functions that the HLS is made of, can be accessed for further checking of the correctness of the component. This is based on the use of explicit or implicit hardware breakpoints. They may be useful for checking the time elapsed between operations, or to halt a hardware component in order to retrieve the DUT's state in a specific time.

In addition, our proposal provides a remote and transparent dynamic verification service through a collection of overlays. Engineers can exercise a DUT remotely, breaking down the test from the hardware prototype and using portable stimuli across the target domain. The verification overlays use DPR features, thus enabling reuse in future improvements of DUTs and increased engineers' productivity. Engineers need only to configure the verification environment in accordance with the DUT's requirements.

Future work will be targeted to extract intermediate results from software simulations and automatically integrate them into the verification process as reference vectors, and then compare the solution with other verification techniques and FPGA families, such as the Kintex family from Xilinx, which does not contain an embedded processor, and the Arria family from Intel. In addition, we will raise the visibility of internal signals, using synthesizable hardware assertions. The purpose of these assertions is to monitor critical aspects of the system and trigger an event when the behavior is wrong. Another working line is oriented to the building of synthetic components in order to reduce third-party dependencies for verification purposes.

Acknowledgments

This work is supported by Spanish Government under project PLATINO (TEC2017-86722-C4-4-R). Also, it has been supported by Regional Government of Castilla-La Mancha under project SymbIoT (SBPLY/17/180501/000334).

References

References

[1] A. Canis, J. Choi, B. Fort, R. Lian, Q. Huang, N. Calagar, M. Gort, J. J. Qin, M. Aldham, T. Czajkowski, S. Brown, J. Anderson, From software to accelerators with LegUp high-level synthesis, in: 2013 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES), 2013, pp. 1–9. doi:10.1109/CASES.2013.6662524.

[2] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, Z. Zhang, High-Level Synthesis for FPGAs: From Prototyping to Deployment, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 30 (4) (2011) 473–491. doi:10.1109/TCAD.2011.2110592.

[3] W. Chen, S. Ray, J. Bhadra, M. Abadir, L. C. Wang, Challenges and Trends in Modern SoC Design Verification, IEEE Design Test 34 (5) (2017) 7–22. doi:10.1109/MDAT.2017.2735383.

[4] P. Mishra, R. Morad, A. Ziv, S. Ray, Post-silicon validation in the soc era: A tutorial introduction, IEEE Design Test 34 (3) (2017) 68–92. doi:10.1109/MDAT.2017.2691348.

[5] L. D. Luna, Z. Zalewski, FPGA level inhardware verification for DO-254 compliance, in: 2011 IEEE/AIAA 30th Digital Avionics Systems Conference, 2011, pp. 7D5–1–7D5–5. doi:10.1109/DASC.2011.6096130.

[6] X. Cheng, A. W. Ruan, Y. B. Liao, P. Li, H. C. Huang, A run-time RTL debugging methodology for FPGA-based co-simulation, in: 2010 International Conference on Communications, Circuits and Systems (ICCCAS), 2010, pp. 891–895. doi:10.1109/ICCCAS.2010.5581847.

[7] A. Wicaksana, A. Prost-Boucle, O. Muller, F. Rousseau, A. Sasongko, On-board nonregression test of HLS tools targeting FPGA, in: 2016 International Symposium on Rapid System Prototyping (RSP), 2016, pp. 1–7. doi:10.1145/2990299.2990307.

[8] M. K. You, G. Y. Song, Case study : Cosimulation and co-emulation environments based on System; SystemVerilog, in: TENCON 2009 - 2009 IEEE Region 10 Conference, 2009, pp. 1–4. doi:10.1109/TENCON.2009.5395829.

[9] Y. N. Yun, J. B. Kim, N. D. Kim, B. Min, Beyond UVM for practical SoC verification, in: 2011 International SoC Design Conference, 2011, pp. 158–162. doi:10.1109/ISOCC.2011.6138671.

[10] L. Feng, Z. Dai, W. Li, J. Cheng, Design and application of reusable SoC verification platform, in: 2011 9th IEEE International Conference on ASIC, 2011, pp. 957–960. doi:10.1109/ASICON.2011.6157365.

[11] A. Organization, Standard Universal Verification Methodology Class Reference Manual, Release 1.1, Tech. rep., Accellera (2011).

[12] J. Podivinsky, M. imková, O. Cekan, Z. Kotásek, FPGA Prototyping and Accelerated Verification of ASIPs, in: 2015 IEEE 18th International Symposium on Design and Diagnostics of Electronic Circuits Systems, 2015, pp. 145–148. doi:10.1109/DDECS.2015.33.

[13] H. Zhaohui, A. Pierres, H. Shiqing, C. Fang, P. Royannez, E. P. See, Y. L. Hoon, Practical and efficient SOC verification flow by reusing IP testcase and testbench, in: 2012 International SoC Design Conference (ISOCC), 2012, pp. 175–178. doi:10.1109/ISOCC.2012.6407068.

[14] R. Edelman, R. Ardeishar, UVM SchmooVM - I want my c tests!, in: 2014 Design and Verification Conference and Exhibition (DVCON), 2014, pp. 1–10.

[15] J. Goeders, S. J. E. Wilton, Signal-Tracing Techniques for In-System FPGA Debugging of High-Level Synthesis Circuits, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 36 (1) (2017) 83–96. doi:10.1109/TCAD.2016.2565204.

[16] J. P. Pinilla, S. J. E. Wilton, Enhanced source-level instrumentation for FPGA in-system debug of High-Level Synthesis designs, in: 2016 International Conference on Field-Programmable Technology (FPT), 2016, pp. 109–116. doi:10.1109/FPT.2016.7929514.

[17] Y. K. Choi, J. Cong, HLScope: High-Level Performance Debugging for FPGA Designs, in: 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2017, pp. 125–128. doi:10.1109/FCCM.2017.44.

[18] J. Curreri, G. Stitt, A. D. George, High-level synthesis techniques for in-circuit assertion-based verification, in: 2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010, pp. 1–8. doi:10.1109/IPDPSW.2010.5470747.

[19] M. Karlesky, M. VanderVoord, G. Williams, A simple Unit Test Framework for Embedded C, Tech. rep., Unity Project, https://fenix.tecnico.ulisboa.pt/downloadFile/845043405431090/Unity%20Summary.pdf; last accessed 18-Mar-22 (2012).

[20] D. Koller, N. Friedman, Probabilistic Graphical Models: Principles and Techniques - Adaptive Computation and Machine Learning, The MIT Press, 2009.

[21] Xilinx, Python productivity for Zynq (Pynq), Tech. rep., Xilinx Inc. (2018).

[22] ZeroC, ICE: A comprehensive RPC framework, Tech. rep., ZeroC (2018).

[23] W. Lie, W. Feng-yan, Dynamic Partial Reconfiguration in FPGAs, in: 2009 Third International Symposium on Intelligent Information Technology Application, Vol. 2, 2009, pp. 445–448. doi:10.1109/IITA.2009.334.

[24] J. Barba, F. Rincon, F. Moya, F. J. Villanueva, D. Villa, J. Dondo, J. C. Lopez, OOCE: Object-Oriented Communication Engine for SoC Design, in: 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007), 2007, pp. 296–302. doi:10.1109/DSD.2007.4341483.

[25] N. Dalal, B. Triggs, Histograms of oriented gradients for human detection, in: 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05), Vol. 1, 2005, pp. 886–893 vol. 1. doi:10.1109/CVPR.2005.177.