Testing framework for in-hardware verification of the hardware modules generated using HLS

Julián Caba, Fernando Rincón, Julio Dondo, Jesús Barba, Manuel Abaldea, Juan Carlos López
University of Castilla-La Mancha, 13071 Ciudad Real, Spain,
email: julian.caba@uclm.es

***Abstract*—High-Level Synthesis (HLS) allows Field Programmable Gate Array (FPGA) developers to easily implement complex applications in silicon, addressing the ever-growing size and complexity of modern embedded reconfigurable systems. Unfortunately, in spite of these advancements, new non-negligible verification problems arise. For instance, the co-simulation strategy may not provide trustworthy results due to the variable accuracy of simulation, or hardware synthesis issues (e.g. those related to signal routing) which are not detectable in the simulation. Hence, developers need new verification mechanisms to reduce the gap between the technology and the verification needs. In this paper, we propose a testing framework and a hardware verification platform based on FPGA technology in order to improve the verification accuracy and enable effortless and fully automatic in-hardware system validation. For instance, one of the mechanisms is the inclusion of physical configuration macros (e.g., clock rate configuration macro) and test assertions based on physical parameters in the verification environment (e.g., timing assertions). Experiment results demonstrate our approach in the context of a case study remaining the same testing technology independently of the module abstraction level.**

***Index Terms*—Design for testability, testing framework, in-hardware verification, debug, high-level synthesis, FPGA**

I. Introduction

Evolution of System-On-Chip (SoC) requires new design tools to make optimal use of new devices in order to reduce the design gap, as well as to handle efficiently the complexity of the hardware design process [1]. So, High-Level Synthesis (HLS) tools provide designers with the ability to speed up their applications or build their own platform components (i.e. Intellectual Property or IP) using algorithms written in a high-level programming language (HLL) such as C, C++ or SystemC. The use of these languages is becoming widespread in the hardware realm since engineers can work at higher abstraction levels when the requirements can be fulfilled without the low-level manual hardware design. Thus, engineers are able to adapt software legacy algorithms providing some goodness such as adaptability to changes, capability of quickening the design space exploration process or shorter time-to-market [2].

Unfortunately, in spite of hardware digital design advancements, there is still a significant gap between technology capabilities and verification needs. Latest studies affirm hardware verification is the bottleneck in the majority of hardware projects and up to 70% of design time is spent in tasks related to verification [3]. The situation is exacerbated by the use of HLS tools because of their limitations which have not been solved yet. 1) A co-simulation strategy is included by most HLS tools in order to check the correctness of hardware designs described in an HLL, reusing the prior software functional tests. However, sometimes the synthesized RTL code does not work in a real device because of the co-simulation environment does not take into account some physical aspects (e.g., routing paths, resource locations). Therefore, engineers may work at several levels (high-level description, RTL, etc.). Each level needs a test rewrite because test cases described for a particular abstraction level are not compatible with other abstraction levels. 2) The HLS report is not fully accurate since it always reports the worst case and, perhaps, it is not a realistic scenario. Thus, hardware designs could work with higher clock frequencies, for example. 3) Engineer's experience plays an important role to generate desired circuits or solutions using some techniques provided by HLS vendors, such as pragmas.

Therefore, current HLS tools do not meet the verification engineer needs and neither can they ensure the designs correctness once they are deployed on real device. Tests must be re-written which is an error-prone task because of its manual nature and a design specific hardware verification platform must be built in order to validate the implementation. Moreover, the validation of hardware designs introduces new demands in addition to the mere correctness of the functional behavior via the comparison of the outputs against a golden model. For instance, to ensure that the latency of each sub-block that compose a hardware design is within a range.

In this paper, we propose a hardware verification framework based on software testing techniques for hardware modules generated using HLS tools. Our proposal considers the verification of a design once it has been deployed on the actual execution platform as part of the process. Thus, we propose a response to those challenges concerning the validation of systems explained above. To achieve our main objective our approach provides the following aspects:

- A testing framework to check Design Under Test (DUT) behavior using macro assertions and enabling sub-block testing or grey-box verification.
- Bring some of the physical parameters into the verification flow using some configuration macros, which allow engineers to configure the hardware verification platform in accordance with the profiling of HLS tools or overclocking the solution generated by these tools.
- A dynamic hardware verification platform to handle the verification process, enabling self configuration from software by configuration macros. This platform is remotely

accessible through a network and decouples the testing framework from the hardware prototype.

This paper is organized as follows. Section 2 describes the current progress in hardware verification. Section 3 describes the proposed hardware testing framework, which is implemented on the architecture presented in Section 4. In Section 5, we present a case study with some experimental results using our approach. Finally, Section 6 summarizes this paper and proposes directions for future work.

II. Related Work

In-hardware verification of system components has recently attracted a lot of attention from the research community on reconfigurable technology. 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 [4] [5] [6] propose several FPGA-based testing platforms with special emphasis to the communication infrastructure. However, unlike our approach, these solutions are quite dependent on the FPGA technology and the tool chain, making it difficult their reusability in other reconfigurable embedded platforms.

Some works try to 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 levels. For example, [7] leverages the use high-level artifacts, already present in System Verilog and SystemC, allowing the functional verification of the design through co-simulation in pure software domain, and then verified through co-emulation after implementation of hardware part onto a specific hardware emulator.

The use of emulation platforms, as it is stated in [8], 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, the use of emulators implies an important effort overhead when it comes to the development and maintainability of the testing framework.

The solution presented in this article 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 the stage of the design, enabling the reusability of the testbed. Some works such as [9] [10] propose in-hardware verification environments inspired on the principles of UVM. The automated generation of the verification environment, an interesting feature which is also supported by our solution, is also proposed by Podivinski et al. in [11].

Nevertheless, it is claimed that one of the entry barriers for UVM to be adopted is its complexity and dependability on an experienced verification engineer. So, several works such as [12] and [13] pursuit to simplify the operation effort of UVM-inspired solutions by promoting the reutilization of the tests through the entire verification flow, independently of the abstraction level.

One of the major challenges of in-hardware verification of HLS-generated designs is to provide a mean to increase signal visibility beyond the barrier which 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. describes in [14] a comprehensive hardware monitoring system that traces a set of signals once the HLS circuit has been synthesized. Though the functionality of the monitoring infrastructure exposes interesting features and it is highly optimized, instrumentation at such low level handicaps understandability and traceability of the root cause of the problem back to the high level model. For this reason, [15] and [16] proposes a monitoring strategy at HLS level, allowing an easier method to identify when are where a mismatch between the expected and reference values occurs. However, the price to be paid comes in the form of an resource and latency overhead 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 memories and logic to store and retrieved the signal values. Contrarily, [15] follows an approach less intrusive, similar to the one implemented in the solution described in this work, where the probes are implemented as output stream channels and drove outside the component. Later, an external component will be in charge to store, analyze or send out the data to the test manager. The use of source-to-source transformation techniques by [15] to free the designer from any extra effort related to the code refactoring process.

Finally, It is worth mentioning a family of works devoted to support Assertion-based Verification (ABV) for reconfigurable designs by the implementations of HLS techniques to efficiently support in-circuit assertions [17]. These works extend the functionality of the hardware monitors so as to perform on-line value checking of the overseen variables. Memory requirements are reduced since it is only needed the registration of the mismatch events.

III. RC-Unity testing framework

In order to lead the whole verification process by means of the use of a single testing framework, Unity is proposed as the reference verification environment. The different artifacts and tools, devoted to validate software programs, will be applied to HLS designs and adapted to fulfil the new requirements resulting from the nature of hardware developments [18]. Notice that our proposal verifies hardware designs described in an HLL, although testing frameworks allow to check the correctness of individual functions which compose a hardware design, we only consider those tests that exercise the top-function.

Firstly, engineers must verify the high-level code which describes their hardware design, exercising the Design Under Test (DUT) from the top-function. We propose *Unity* testing framework, as the reference for RC-Unity, to check

the behavior of hardware modules using HLS. *Unity* is fairly portable across unalike platforms, such as 8-bit microcontrollers or 64-bit processors, because of it is written in ANSI C. *Unity*, and most frameworks, has a variety of assertions which can be placed in the test to verify the production code. For instance, the `TEST_ASSERT_EQUAL(expected, current)` macro checks the equality between two values: `expected` and `current`.

Once the high-level description of the hardware design is tested in a purely software domain, the HLS tool translates the high-level code into RTL code. In this work we use Vivado HLS from Xilinx. At this stage, we can run the tests (software) over the RTL model without any change (*co-simulation stage*).

The last stage generates the configuration file and deploys it to the hardware verification platform. Therefore engineers have to build on handmade a custom hardware verification platform to exercise their hardware designs, however, in this paper, we propose a hardware verification platform which is configurable from software (see Section IV). To facilitate the configuration tasks we have extended the Unity testing framework with the inclusion of a number of configuration macros (see Table I). For instance, the tests can annotate the physical parameters, such as the clock rate to be used or the number of cycles the clock enable must be set active in order to configure the proposed hardware verification platform. Moreover, *RC-Unity* framework allows engineers to retrieve information about the latency of the execution test.

`RCUNITY_RESET` sets the DUT to a well-known initial state and `RCUNITY_CONF_CLK_EN` configures a timer that establishes the operation time of the DUT. `RCUNITY_CONF_CLK_RATE` configures the clock rate of the area where the DUT will be deployed. RCUNITY_START makes the DUT active, starting an internal counter and waiting for the DUT to complete its operation unless a limit of time had been established using `RCUNITY_CONF_CLK_EN`. By default, the test manager measures the time for an operation taking as the reference the start signal of the first FU block (start time) and the first done signal of the first FU block (stop time) issued by the DUT. This behavior can be modified by verification engineers, which are given the possibility to alter the standard behavior (`RCUNITY_CONF_FU_INPUT` and `RCUNITY_CONF_FU_OUTPUT` macros) as to the moment the *begin* and *end* counters are started/stopped. Last but not least, the macros `TEST_ASSERT_TIME` compares the elapsed time (retrieved from the platform) against the value of the macro parameter which expresses clock cycles. Several comparison operators are available in *RC-Unity* such as `EQ` (equal than), `LT` (less than), `GT` (greater than), `LE` (less or equal than) and `GE` (greater or equal than).

Listing 1 shows an example of a test with *RC-Unity* framework. Firstly, verification engineers configure the hardware verification platform through the configuration macros, hence engineers set the clock rate which the DUT works, here one is able to observe the DUT behavior under overclocking/underclocking conditions, and the number of cycles that the clock enable is in active high mode (lines 4 and 5 of Listing 1). Configuration macros are lazy operations, 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 in 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, thus they invoke the top-function (line 8 of Listing 1). This invocation really is a fake invocation because the DUT is running on hardware and the test runs on software domain, hence the invocation bridges both domains. Finally, verification engineers can assert the time elapsed by the test (line 9 of Listing 1).

Listing 1: Example of a test with *RC-Unity* testing framework

```
1  void
2  test_module(){
3  #ifdef HW_TEST
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    result = moduleDUT(stimuli)
9    TEST_ASSERT_TIME_LT(750); // Checking time
10 #else
11   result = moduleDUT(stimuli);
12 #endif
13   for(int i=0; i!=16; i++)
14     TEST_ASSERT_EQ(reference[i], result[i]);
15 }
```

A. Grey-Box Verification

In Listing 1, we do not use `RCUNITY_CONF_FU_INPUT` and `RCUNITY_CONF_FU_OUTPUT` macros because the hardware design or DUT has been described into a single function which plays the role of the top-function. Nevertheless, engineers usually encapsulate their hardware designs into a top-function which calls upon other functions, hence the top-function contains nested functions whose functionality is usually verified as a unique module (black-box verification). Nevertheless, verifying each function individually enables to delimit potential bugs, these nested functions are seen as black-boxes but the overall view of the hardware-module's architecture is known by engineers (grey-box verification).

Following the translation process done by HLS tools, nested functions are translated into functional units (FUs) and are linked between them to perform the desired behavior. In software domain, engineers are able to check the behavior of each nested function individually, also in a co-simulation domain, verification engineers only have to change the top-function to choose the correct nested-function under test and select the properly test case(s). Unfortunately, in hardware domain, engineers do not have good solutions to check the behavior of each FU that compose a hardware design increasing their visibility, because engineers include some debug artifacts that do not appear in the final release of their hardware modules. These debug artifacts can disturb the behavior of hardware designs.

To increase the verification visibility and to verify the behavior of FUs chain, we propose to modify the control data flow graph (CDFG) of the original hardware design. We

Table I: Macros of *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_FU_INPUT(num)`	Modifies the moment to start the internal counter that measures the time elapsed by a operation. By default 1
`RCUNITY_CONF_FU_OUTPUT(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 that measures the time elapsed by a operation. By default 1
`TEST_ASSERT_TIME_XX(expected)`	Compares the time obtained and expected value in accordance with the comparison operator

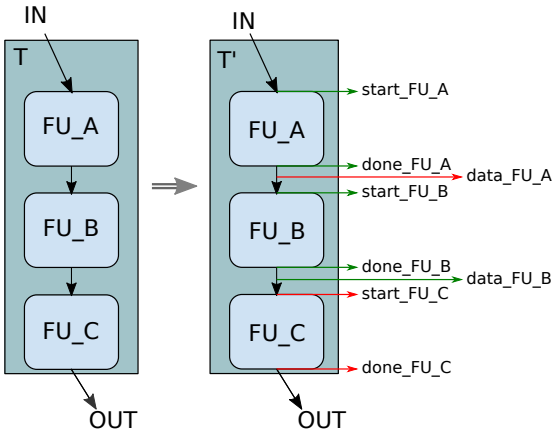

Figure 1: Original DUT vs Modified DUT

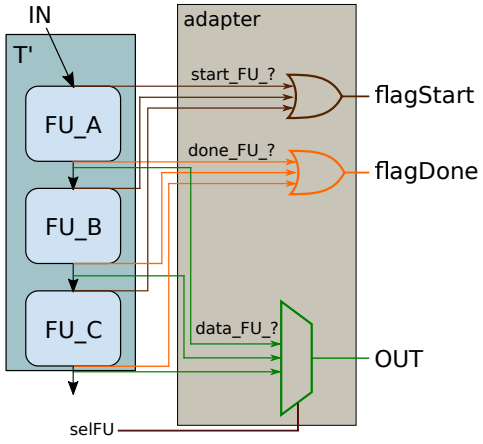

Figure 2: Overview of generated adapter and the communication with the original DUT

duplicate the output of each sub-block (or FU) which contain a DUT and route that output to the module interface and to the next FU in order to not disturb the original path (see Figure 1). Therefore, our proposal exercises the DUT from the original entry point, but the output point changes in accordance with the FU under test. Figure 1 shows an example related to verification of FUs. This example is based on a top-function (`T`) which contains three nested FUs (`FU_A`, `FU_B` and `FU_C`), note each FU depends on the previous one. Thus, if one wants to verify the intermediate `FU_B` block, he should include some extra code to bridge the output of `FU_B` block to the output point (right side of Figure 1). Our solution duplicates the output of each FU except the last FU and adds two additional signals, start and done, at the beginning and ending of each FU respectively. Start signal is activated during one cycle before executing the FU's tasks, whereas done signal is activated during one cycle when FU ends its tasks.

Therefore, verification engineers can check intermediate FUs, but the return values of the FU under test must be routed to outside and manage them. To manage the new signals which contain the hardware design, we provide an adapter that is directly connected to those extra signals and to a hardware component (*Test Manager* component, see Section IV) enabling hardware introspection. Figure 2 illustrates a general overview of the communication between the modified DUT (`T'`) and the generated adapter. The start and done signals are grouped by `OR` gates while data signals (output data of each FU) are selected by a multiplexer in accordance with the `selFU` signal. Notice that the adapter is more complex and the figure shows an abstraction of its behavior, moreover, the adapter is generated automatically from HLS tools.

Now, one can exercise the DUT delimiting the FU under test. Listing 2 shows a new test which verifies the `FU_B` block instead of the whole DUT. In accordance with Figure 2, green arrows are analyzed during the execution of the test that listed in Listing 2, while red arrows are ignored and they do not interfere in this test. The configuration macros are similar that the previous test (Listing 2), but in this case we use `RCUNITY_CONF_FU` macro to measure the time elapsed by the `FU_B` block, this macro annotate the value 2 (line 5 of Listing 2), because FU_B block is the second block to be executed and we check the time elapsed in that FU during the test (line 9 of Listing 2). Further, we can use `RCUNITY_CONF_FU_INPUT` and `RCUNITY_CONF_FU_OUTPUT` macros with the values 1 and 2 respectively to measure the time elapsed by `FU_A` and `FU_B` blocks, thus we are able to retrieve the latency from

the beginning of the stimulation to the end of FU_B block execution. In this scenario, the *moduleDUT* invocation returns the values related to the FU_B block, thus we might compare them with some golden intermediate values (line 10 of Listing 2).

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

```
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 }
```

Our proposal contains an important restriction. It is limited to those solutions 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 particular vertex and return to that vertex. Therefore, loops between FUs are not supported at all in our proposal. We only can measure the time between one FU and other or even the same FU but verification engineers must know the position of each FU. For instance, imagine our three FUs are placed inside a loop and we want to measure the time elapsed by FU_B block from the beginning to the second iteration. In this case, we might use RCUNITY_CONF_FU_INPUT with the value 1 and RCUNITY_CONF_FU_OUTPUT with the value 5.

IV. Hardware verification platform

Nowadays, most HLS developers has to build their own custom hardware verification platform for verifying the modules generated by HLS tools on handmade. In order to allow an in-hardware verification or on-board verification which is one of our goals, we provide a hardware verification platform. Our verification platform relies on a SoC platform which integrates an FPGA and a hardcore-processor in the same device (hybrid FPGAs). An overview of that platform is shown in Figure 3, in our case, we use a ZedBoard from Xilinx. In addition, Figure 3 shows the communication between the hardware verification platform and a developers workstation. Both actors exchange messages using zeroC Ice communication middleware.

The ZedBoard device allows an easy communication between the FPGA logic part and the hardcore-processor using standard communication mechanisms. In our case, an AXI bus connects PS and PL sides, tunneling a master-slave communication, where PS is the master and the components programmed in PL are slaves. In addition, this device is connected to a computer network via Ethernet in order to share the verification platform, thus we convert a hybrid FPGA into a remote testing service.

A. Programmable Logic Architecture

Figure 3 (right side) illustrates the Programmable Logic (PL) architecture layout of our hardware verification platform in the ZedBoard. The DUT is deployed in this side and it is connected with the Processing System (PS) side through two FIFO channels which bridge the AXI data between the DUT and the PS whenever hardware address matches with the address where the *dpr_bridge* component is mapped, thus the data is stored into an input FIFO when the operation is a write. On the other side, when the operation is a read, the *dpr_bridge* component retrieves data from the output FIFO and sends it through the *AXI* bus. This communication mechanism is a typical configuration of most accelerators where they read a stream of input data and produce an output data stream. Anyway, this interface could be replaced by another protocol/interface such as *AXI-Stream* or *AXI-Lite*, or we could even add our own protocol over a streaming channel. Thus, tests can run in software domain and exercise the DUT from that domain.

In order to reuse the verification platform in future projects (reusability challenge), the DUT is deployed on a dynamic reconfigurable area while the rest of the layout does not change, thus the verification platform uses Dynamic Partial Reconfiguration feature (DPR). Hence the PL layout is divided into two parts, a static one which contains those components that do not change independently of the DUT and the DUT, which can be part of a vision algorithm, voice filter, cryptographic algorithm, etc. One advantages of the use of DPR is that it reduces the synthesis process and improves synthesis tasks [19] due to working with partial bitstreams. Furthermore, it is possible to adapt an FPGA to different scenarios, by just modifying the functionality or the performance of some related tasks [19], enabling the reusability of our verification platform.

To perform deployment tasks, the verification platform needs a reconfiguration engine: the *zipFactory* component. This component programs a partial bitstream (new DUT) into the dynamic reconfigurable area available in our verification platform without microprocessor intervention. Firstly, bitstream data should be stored into a memory address of the DDR and then the *zipFactory* component retrieves with an internal DMA these data which is stored into an internal small buffer, handling them as batched data. The ICAP bandwidth is 32-bits, which matches the internal buffer width. The component knows the type of 32-bit word sent to the ICAP at each transaction, thus when the 32-bit word is the desynchronization word the reconfiguration process is finished, although we attach some NOPs to flush the command pipeline properly.

The *Test Manager* component performs a variety of hardware tasks which are not feasible from software or result in a poor accuracy where they are performed in the software domain. Each hardware tasks is directed from software using the macros listed in Table I.

- It resets the dynamic area (or DUT) in order to assure that the DUT starts from a well-known state.
- It manages the clock enable signal during the cycles depicted 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 time we annotate a 0 value.

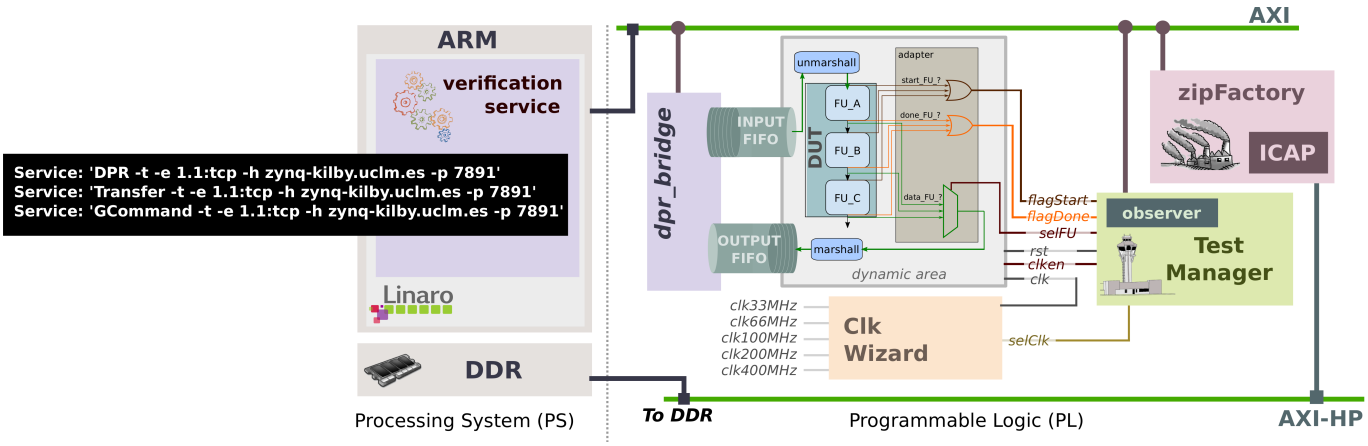

Figure 3: Overview of our hardware verification platform

- It sets the clock rate of the dynamic part and those modules that interact with it. The available clock speed rates are: 33MHz, 66MHz, 100MHz, 200MHz and 400MHz. This feature provides a flexible environment allowing engineers to reuse the verification platform in future projects, also when an amount of versions of a particular module generated by an HLS tool works at different clock rate. Therefore, verification engineers do not need to build a new verification platform o synthesize the whole design in accordance with the profiling clock rate, hence they have only to annotate which clock rate will be used.
- It measures the time elapsed by each FU or between FUs. The Test Manager component observes the start and done signal of each FU and determines the number of cycles to complete an FU or several consecutive FUs. It works increasing an internal counter which plays the role of a chronometer. The internal counter is triggered by start signals of FUs, whereas it is stopped by done signals. The RC-Unity testing framework allows engineers to set the FU under test in order to measure the number of cycles that takes to complete its tasks or set the initial and final FU in order to measure the number of cycles that takes to complete the tasks done between both FUs.
- It configures the DUT's datapath in order to route the return values of an FU to the output point.

B. Processing System

Figure 3 (left side) shows the PS architecture of our hardware verification platform. The PS contains an ARM processor which runs an embedded operating system, Linaro Ubuntu distribution. To use the internal components of the verification platform, we deploy three services based on ZeroC Ice over this OS. For instance, when a partial bitstream is ready we can send its data to the FPGA using the *transfer* service which stores the data in a memory address. Then we can use the *dpr* service to deploy the partial bitstream on the dynamic area. Therefore, we provide the mechanism to share transparently our hardware verification platform, increasing its availability and accessibility. Finally, we may run the test on-board mode (step 3 of flow proposed), in this case the *GCommand* service translates the communication messages into AXI messages whose hardware address matches that of the DUT.

From outside of the verification platform, one may use some wrappers of these three services in order to marshall and unmarshall the data into ZeroC Ice messages. These services are shown as functions from the testing framework point of view. Indeed, the *moduleDUT* function in Listings 1 and 2 serializes stimuli into zeroC Ice messages and deserializes data retrieved from the FPGA too. Notice that stimuli and result are streaming data that will be retrieved and stored from the input FIFO and to the output FIFO respectively.

V. Experimental Results

Our experiment results have been obtained under a GNU/Linux environment using a Xilinx ZedBoard platform to implement our approach. By default, the platform works at 100 MHz, but the dynamic area and other components can modify this clock speed rate at runtime. As a case study in this paper, we select one based on the 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. The algorithm implementation is divided into different steps: gamma and color normalization, gradient computation, block normalization among other. In our case study, the step chosen is the vector normalization block with l^2-*norm* normalization factor [20]. The solution has been developed in the C programming language using Vivado HLS v15.4 [21].

The input and output of l^2-*norm* factor is 16 pixels, a 4x4 window. The l2norm layout is divided into three FUs like our previous example, but in this case their names are `FU_sum`, `FU_scale`, `FU_mult`. These FUs are sequentially executed and each one depends on the previous FU, thus it forms up a DAG with three vertices and two edges. Applying our approach the original DUT is converted into four vertices (one per FU plus the adapter) and five edges, the two original edges plus an edge from each FU to the adapter vertex. Figure 4 shows the signal waveforms of l^2-*norm* algorithm

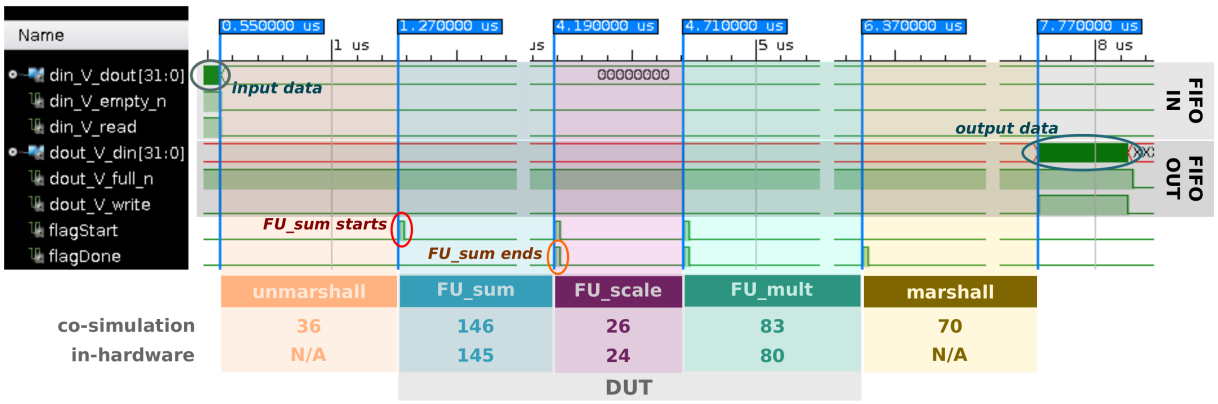

Figure 4: Waveform of l^2-*norm* test and timing comparison

in a complete co-simulation and a comparison of timing results (depicted in cycles) obtained from co-simulation and in-hardware domain applying our solution. In the in-hardware scenario, the timing penalty introduced by our solution is substracted, displaying accurate timing results.

Firstly, before exercising the DUT an unmarshalling task takes place, as well as datapath configuration task, thus the design does not need evaluate at runtime if the data goes to the next FU or to output point. The unmarshalling task is performed independently of our approach because of the custom interface used. Note the interface of our dynamic area is modelled with two FIFOs which is the most popular interface in hardware accelerators. Then, the first FU (`FU_sum`) is exercised and the `flagStart` signal is high activated during one cycle. Once `FU_sum` block finishes its execution, the `signalDone` is high activated annotating that moment, at the same cycle the `signalStart` is high activated because the second FU (`FU_scale`) starts an so on with the rest of FUs. Finally, the result values are marshalled before send it. This last operation is done with or without our approach.

Unfortunately, our approach includes an overhead. The hardware overhead incurred by hardware introspection generated by the approach proposed in this paper is analyzed according to the original DUT. Table II shows the resource overhead of our approach. In addition, we compare the profiling generated by Vivado HLS tool with the report provided by Vivado P&R tools. We can observe the profiling of the HLS tool contains some inaccuracies. About latency overhead, we can conclude that our approach introduces one cycle per each FU because the management of start and done signals. Note that the latency values have been retrieved from HLS report and our hardware verification platform.

Table II: Comparison between Vivado HLS and After *Place & Route* reports (l^2-*norm* algorithm)

	Vivado HLS tool		After P&R phase	
	Original	RC-Unity	Original	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.

VI. Conclusion

This paper has presented a hardware testing framework, RC-Unity, for in-hardware verification of the hardware modules generated by HLS tools, considering both functionality and timing issues. Our approach is well-suite for software or hardware developers. We propose the use of a variety of configuration/control macros to program physical parameters, such as operating clock frequency, of our verification platform. Moreover, RC-Unity framework provides some timing macros to check the time elapsed by the hardware design and allows engineers to make hardware introspection, checking intermediate results, or in other words the return values of an FU and the time that takes that specific FU to perform its tasks. RC-Unity framework is able to check the time elapsed from an FU to another later FU.

In addition, our proposal provides a remote and transparent dynamic verification service. Engineers can exercise a DUT remotely, breaking down the test from the hardware prototype. The verification platform uses DPR feature of FPGAs which enables its reusability in future projects. Thus, engineers only need to configure the verification platform in accordance with the DUTs requirements, either engineers can verify hardware designs in the context of an overclocking or underclocking environments.

Future work will be targeted to take out intermediate results from software simulations and integrate them into the verification process automatically. In addition, we will raise the visibility of internal signals using synthesizable hardware assertions, because RTL description generated by HLS tools exacerbated the trace of signals in a simulator.

Acknowledgments

This work is supported in part by Spanish Government under projects `REBECCA (TEC2014-58036-C4-1R)` and `PLATINO (TEC2017-86722-C4-4-R)`.

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, and J. Anderson, "From software to accelerators with LegUp high-level synthesis," in *2013 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, Sept 2013, pp. 1–9.

[2] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-Level Synthesis for FPGAs: From Prototyping to Deployment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 473–491, April 2011.

[3] W. Chen, S. Ray, J. Bhadra, M. Abadir, and L. C. Wang, "Challenges and Trends in Modern SoC Design Verification," *IEEE Design Test*, vol. 34, no. 5, pp. 7–22, Oct 2017.

[4] L. D. Luna and Z. Zalewski, "FPGA level in-hardware verification for DO-254 compliance," in *2011 IEEE/AIAA 30th Digital Avionics Systems Conference*, Oct 2011, pp. 7D5–1–7D5–5.

[5] 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," in *2010 International Conference on Communications, Circuits and Systems (ICCCAS)*, July 2010, pp. 891–895.

[6] A. Wicaksana, A. Prost-Boucle, O. Muller, F. Rousseau, and A. Sasongko, "On-board non-regression test of HLS tools targeting FPGA," in *2016 International Symposium on Rapid System Prototyping (RSP)*, Oct 2016, pp. 1–7.

[7] M. K. You and G. Y. Song, "Case study : Co-simulation and co-emulation environments based on System; SystemVerilog," in *TENCON 2009 - 2009 IEEE Region 10 Conference*, Jan 2009, pp. 1–4.

[8] L. Feng, Z. Dai, W. Li, and J. Cheng, "Design and application of reusable SoC verification platform," in *2011 9th IEEE International Conference on ASIC*, Oct 2011, pp. 957–960.

[9] A. Organization, "Standard Universal Verification Methodology Class Reference Manual, Release 1.1," Accellera, Tech. Rep., 2011.

[10] Y. N. Yun, J. B. Kim, N. D. Kim, and B. Min, "Beyond UVM for practical SoC verification," in *2011 International SoC Design Conference*, Nov 2011, pp. 158–162.

[11] J. Podivinsky, M. imkov, O. Cekan, and Z. Kotsek, "FPGA Prototyping and Accelerated Verification of ASIPs," in *2015 IEEE 18th International Symposium on Design and Diagnostics of Electronic Circuits Systems*, April 2015, pp. 145–148.

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

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

[14] J. Goeders and 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*, vol. 36, no. 1, pp. 83–96, Jan 2017.

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

[16] J. P. Pinilla and 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)*, Dec 2016, pp. 109–116.

[17] J. Curreri, G. Stitt, and 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)*, April 2010, pp. 1–8.

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

[19] W. Lie and W. Feng-yan, "Dynamic Partial Reconfiguration in FPGAs," in *2009 Third International Symposium on Intelligent Information Technology Application*, vol. 2, Nov 2009, pp. 445–448.

[20] N. Dalal and 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, June 2005, pp. 886–893 vol. 1.

[21] Xilinx, "Vivado Design Suite User Guide: High-Level Synthesis (UG902)," Xilinx Inc., Tech. Rep., 2017.