






Article

FPGA-Based Solution for On-Board Verification of Hardware Modules Using HLS

Julián Caba * , Fernando Rincón , Jesús Barba , José Antonio de la Torre  and Juan Carlos López 

School of Computer Science, University of Castilla-La Mancha, 13071 Ciudad Real, Spain; fernando.rincon@uclm.es (F.R.); jesus.barba@uclm.es (J.B.); joseantonio.torre@uclm.es (J.A.d.I.T.); JuanCarlos.Lopez@uclm.es (J.C.L.)

* Correspondence: julian.caba@uclm.es

Received: 24 September 2020; Accepted: 26 November 2020; Published: 30 November 2020



Abstract: High-Level Synthesis (HLS) tools provide facilities for the development of specialized hardware accelerators (HWacc). However, the verification stage is still the longest phase in the development life-cycle. Unlike in the software industry, HLS tools lack testing frameworks that could cover the whole design flow, especially the on-board verification stage of the generated RTL. This work introduces a framework for on-board verification of HLS-based modules by using reconfigurable systems and Docker containers with the aim to automate the verification process and preserve a clean testing environment, making the testbed reusable across different stages of the design flow. Moreover, our solution features a mechanism to check timing requirements of the HWacc. We have applied our solution to the C-kernels of the CHStone Benchmark on a Zedboard, in which the on-board verification process has been accelerated up to four times.

Keywords: Field-Programmable Gate Arrays (FPGA); high-level synthesis; on-board verification; testing; docker

1. Introduction

Today's advanced embedded system designs pose a challenge for teams that have to deal with tight deadlines. A productive methodology for designing specialized hardware accelerators (HWacc) is that where more time is spent at higher levels of abstraction, verification time is minimized and productivity is maximized. In this sense, High-Level Synthesis (HLS) makes the designer to focus his effort on designing and verifying the right system, as well as make designers to explore faster the design space for value-added solutions.

Focusing on verification tasks, HWacc developed using HLS must be verified at each abstraction level to assess its correctness. Therefore, once the behavior of the C model is verified (simulation), the generated RTL model has to also be verified to check whether it is functionally equivalent (through co-simulation). Then, after going through the place & route process, the synthesized RTL may be mapped on a reconfigurable fabric to be verified again (on-board verification) [1]. These three verification stages are considered pre-silicon activities [2], whose objective differs from post-silicon activities, which creates a gap between both; the first is driven by models, whilst the second ensures that the silicon works properly under actual conditions. In contrast, early pre-silicon activities omit important details of the physical layer, and designers are forced to use prototyping platforms, such as FPGAs. These devices have a similar complexity to post-silicon activities and make control, observation and debugging difficult to accomplish [1].

Unfortunately, commercial HLS tools only cover the first two phases of pre-silicon activities (see dotted box in Figure 1). On the one hand, commercial tools such as Vivado HLS from Xilinx or

Catapult HLS from Mentor use the cosimulation and smoke testing strategies respectively to perform an RTL verification by instrumenting the testbench with wrappers that do not add additional code to the generated logic model. On the other hand, academic HLS tools, such as LegUp, provide some facilities to debug the HWacc in-situ but instrumenting the C code [3], which adds new paths in the generated RTL and, hence, modifies the component under test.

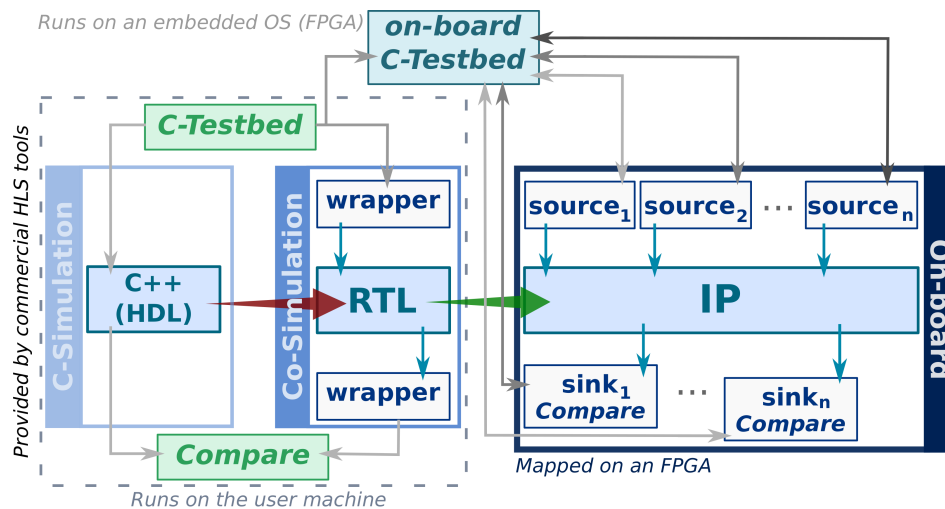


Figure 1. Big picture of the proposed approach.

In this article, a novel solution to mitigate the impact of the verification processes in modern FPGA-SoC design is introduced, but without altering the structure nor the component behaviour. This proposal combines techniques that are familiar to software engineers (unit testing) and seamlessly integrates in current design workflows supported by commercial HLS tools (see Figure 1). According to [1] verification activities account for, roughly, 60% of the development cycle, and it is exacerbated by industrial verification trends [4]. In this sense, the focus of this work is on providing a solution to reduce the gap between pre-silicon and post-silicon verification activities when it comes to the use of HLS tools. This proposal is based on a set of tools and an on-board verification infrastructure for HLS-based modules. This paper has four main contributions:

- A framework that automatically verifies a HWacc across abstraction levels, starting from their high-level description by using Docker images.
- An automatic translation of the test cases from a timeless model to a timed one that allows designers to assess the timing development of a HWacc.
- A set of timing and checking pragma directives that modify and verify the signal protocol, respectively. Such artifacts bring the facilities to explore an enriched set of testing scenarios that HLS tools cannot carry out.
- A demonstration and a statistical comparison of the proposed hardware verification framework.

This paper is organized as follows. Section 2 describes the related work. In Section 3, we present the hardware verification framework that instruments HWacc for validation purposes, in which an FPGA-based platform is used to carry out such a process through Docker containers. In addition, we explain how the HWacc is verified across the abstraction levels by using the same testbed. Section 4 presents an analysis of our hardware verification framework and compares the proposal presented in this paper with others. Finally, Section 5 summarizes the major outcomes of our proposal.

2. Related Work

There is a wide range of techniques and strategies to check the behavioral equivalence of HLS and synthesized RTL design models.

2.1. Instrumenting the High-Level Description

There are several works that include debugging synthesizable functions in order to find errors, so designers can observe what it is happening inside the HWacc. Goeders et al. introduce in [5] a signal-tracing technique that records signals and restores them offline. Pinilla et al. in [6] propose the inclusion of monitors at HLS level that allow designers to retrieve timing values. In this sense, the proposal provided by Choi et al. in [7] brings a solution less intrusive than the one proposed in [6] due to the use of tiny monitors inside the HWacc that measure the elapsed time between functional units. In any case, this kind of solution modifies the original HWacc high-level model by introducing synthesizable artifacts to instrument them. Hence, new route paths appear in the HWacc under test that will not be present in the final release. This fact only makes it worse the pre-silicon and post-silicon verification gap.

2.2. In-Hardware Verification

FPGA-based prototyping is an extended method for on-board verification and early software development. In this sense, there is a set of works that propose FPGA-based platforms to verify hardware designs. Han et al. in [8] propose a debugging system for FPGA prototyping based on Network-on-Chips. Sarma and Dutt in [9] present a design library along with an FPGA-based platform to build and verify the adaptive computing using the CPSoS paradigm. Podivinsky et al. in [10] propose an automated FPGA-based platform that includes a set of synthesizable artifacts of Universal Verification Methodology (UVM), whilst other, such as the scoreboard, run outside the FPGA, either way this proposal applies the UVM principles. Therefore, building, testing and deploying a HWacc manually involve repetitive work plus a large knowledge of the system and tools. As a conclusion, automating everything is the way forward.

2.3. Test Cases

Hardware design implies rewriting test cases to assess the consistency of the design across the different stages of the development flow. Thus, the definition of test cases is as important as the design of the HWacc because the validity of the final product depends on them. Portable Test and Stimulus Standard (PSS) from Accellera is a Domain-Specific Language (DSL) that can be used to create a single representation of stimuli [11] and use it at the different implementations. Edelman and Ardeishar in [12] present a novel solution that provides reusable test cases in UVM-based environments. In this sense, our approach provides a solution to reuse the C-tests across abstraction levels, by aligning our solution with the nature of HLS.

3. Materials and Methods

Achieving high productivity in verification processes is desirable for design teams. In this regard, designers must use C-based executable models to decrease simulation times, whilst the rest of the design flow must be automated. Thus, designers could spend more time designing value-added solutions instead of building verification systems/platforms that are only valid for testing purposes and usually little or no reusable. When it comes to HLS-based design, designers only deal with a specification of the HWacc in a High-Level Language (HLL), such as C or C++, which is tuned and optimized until the design meets the requirements. To this end, designers must describe test cases to validate the HWacc and, then, analyse the reports produced by the HLS tool to decide the optimizations that will be applied in the next iteration. On top of this, the nature of HLS follows a timeless model because of the use of programming languages that do not include timing features. In this sense, tests are coded with the same programming languages and, thus, they do not allow to change the signal timing or interface protocol. These characteristics are desirable since they allow to stress the HWacc and cover more scenarios.

This work is featured by the use of testing frameworks, such as Unity [13], to facilitate the description of the test cases and keep up a robust test structure [14], whilst signal timing requirements are expressed as pragma directives. The rest of the verification tasks, co-simulation and on-board verification stages, are automated by our hardware verification framework, which is able to interpret the timing requirements that have been included by the designer into the test cases. For example, our framework is able to check the temporary behavior of the HWacc that C-Simulation and Co-Simulation steps cannot, because of the nature of their timeless model. Thus, this proposal follows the principles of high-level productivity design, which drives the designer's effort towards designing the ideal solution.

Following the design flow of the HLS-based high productivity design methodology, that has been extended by our hardware verification framework, the first transition (red arrow in Figure 1) translates the C specification into the RTL code. The RTL model is instrumented by the HLS tools with a set of wrappers for verification purposes. Stimuli are extracted from the execution of the software simulations and stored for later use in the Co-Simulation and on-board verification stages. The second transition (green arrow in Figure 1) creates the bitstream from the RTL generated in the previous stage. At this point, the HWacc is automatically instrumented with hardware components, source and sink, that replace the wrappers of the Co-Simulation stage to exercise and compare the results of the HWacc, respectively. The source components exercise the HWacc under test, whilst the sink components compare the results obtained with the expected values. Previously, the test cases written to verify the correctness of the C/C++ model are automatically adapted by our verification framework, so they can be reused during on-board testing without any extra work required from the designer.

3.1. On-Board Verification Environment

The proposed on-board verification environment is based on FPGA-based hybrid platforms, whose architecture is divided into two parts: the Processing System (PS) where test cases are executed; and the Programmable Logic (PL) where the HWacc is instrumented with source and sink components. A fourth component is responsible for orchestrating the verification process in the PL part.

3.1.1. Instrumenting the HWacc

To make the testing process transparent to the software and, at the same time, independent of the hardware platform, the HWacc interfaces with the infrastructure through source and sink components. These components provide or consume data to/from the HWacc, respectively, as Figure 1 shows. In this sense, we consider the source components to be the initiators because they feed the HWacc by writing values into data input port, whilst the sink components are the targets because they ingest data from an output port of the HWacc. In the current version of the verification platform, only AXI-Stream interfaces to the HWacc are supported, since it is the interface most commonly used [15,16] in this type of accelerators. Even though other types of interfaces or protocols are out of the scope of this work, but they could be adopted following a similar approach. Each AXI-Stream interface of the HWacc is connected either to a source or sink component, i.e., the hardware verification platform must provide as many of these components as the HWacc needs.

Source Instrumenting-Component

The source exercises the HWacc through an AXI-Stream packet emulating the behavior of the actual producer component in the final system. The stimuli and timing/signal attributes associated to a source are read from a pre-assigned memory space in DDR memory and, then, stored in the internal buffer of the source component. Thus, the source is inherently capable of buffering data and understands the behavior of not mandatory signals, such as TLAST signal of AXI-Stream protocol, and the time in which the signal must be set/cleared. Each TDATA value is associated to a *last* and a *delay* values; the former marks the boundary of a packet (TLAST signal), while the latter establishes the number of cycles that the TVALID signal must be low before set it (delay). In brief, source components

can generate AXI-Stream packets with delays, something that cannot be done by HLS tools because of their timeless model, i.e., only ideal testing scenarios are feasible with HLS tools, where there are no data delays or protocol handshaking is always fulfilled. In consequence, through our verification platform, designers can explore an enriched set of testing scenarios where the HWacc is exercised to check for protocol correctness at cycle level through timing pragma directives included in the test cases. Again, HLS tools cannot handle these scenarios and, therefore, are not able to ensure that the HWacc would work correctly under no ideal scenarios. Our verification framework is able to understand such pragmas embedded in the test cases, which, in turn, are translated from a timeless model to a timed one by adding delays (if they are specified) between stimuli. Thus, developers can observe and analyze the adverse effects of delays in the HWacc under test. The timing pragma directives that alter the timing behavior of the AXI-Stream signal set, except for the TDATA signal, are the following:

- **rand:** Generates random values.
- **none:** No signal check. Ideal packet is generated.
- **<file>:** Binds the signal values to the contents of a file.

From a technical point of view, data is organized in N -bit + 32-bit packets, where N is the width of the TDATA signal (expanded to multiples of 32-bit words) and the following 32-bits encode the attributes of such TDATA (delay, last, ...). These packets are stored in the off-chip memory without padding (no gap) between them, so it is minimized the waste of space due to internal fragmentation when the word size of TDATA differs from 32 bits. For the sake of simplicity and ease of debugging of the prototype at signal level, this parameter is fixed to 32 bits (see Figure 2b): (1) 16 bits to specify the delay (TREADY) and; (2) the other 16-bits are devoted to code current (e.g., TLAST) and future attributes that could be added (i.e., TKEEP). These sizes can be adjusted, reducing the overhead due to side-channel information requirements.

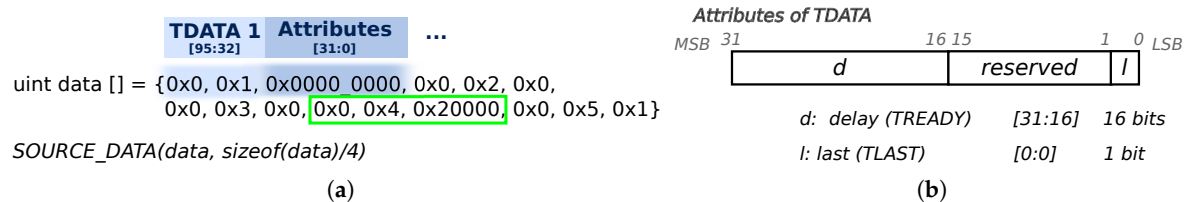


Figure 2. Data organization of source component. (a) High-level test function; (b) Attributes of TDATA.

In the end, the source component must deserialize the data it receives from the DDR. The logic in charge of the interpretation of the byte stream is implemented by an FSM (Finite State Machine). This FSM separates the different parts (TDATA and its attributes) of a stimuli packet. Figure 2a represents the software function that copies in source’s memory address space the stimuli corresponding to five samples. The values of vector data [] are automatically generated by our verification framework out of the trace of the C-simulation and with the help of the timing pragma directives.

Figure 3 depicts the layout in the off-chip memory of the same five stimuli samples, after the execution of the SOURCE_DATA function. Then, the memory data are read by the source component, temporarily stored in the local FIFOs and finally resulting in a specific sequence of signals to exercise the DUT. In this example, one sample has been bounded with a green rectangle, so the reader can easily trace it through Figures 2a and 3.

As it can be seen, despite the fact TDATA size is 64-bit, the alignment of the TDATA samples do not have to be forced to 64-bit, no padding is added and, therefore, no useless bytes are left in memory. Furthermore, the width of the input AXI Memory interface in the source component, through which the stimuli are received, may vary depending on the platform on which the on-board testing infrastructure will be implemented. The control logic module is the main responsible for interpreting the streaming of stimuli and store them in the corresponding FIFOs. This fact makes it possible to separate the

data memory access from the testing issues, but achieving a high bandwidth. Thus, using one target platform or another does not imply a change in the testing model, but it requires a change in how data are transferred from off-chip memory (see Section 3.1.4).

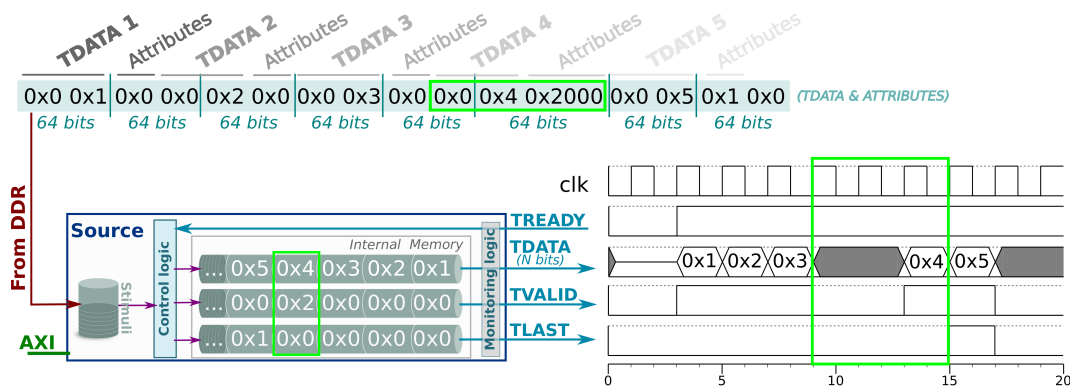


Figure 3. Block and timing diagram example of a source component.

Concerning the time diagram, the delay value of the first sample must be 0, that is, the AXI-Stream signals are assigned to their corresponding values when the handshaking enables it (TREADY is set). The boundary of a packet can be marked by the value 0x0001 in the last attribute of the last sample, however, designers can indicate as many of last indicators as they need, to set the TLAST signal at will. Figure 3 also illustrates an AXI-Stream transfer with delays, in which the fourth value is delayed 2 cycles (highlighted in green).

Although the signal set of AXI-Stream protocol has been limited to the TREADY, TDATA, TVALID and TLAST signals, the rest of them, such as TKEEP, could be included following the same approach. In the example shown in Figure 2b, the 32-bit word, in which the attributes are encoded, contains some unused bits, so the TKEEP can be managed as a new attribute (see Figure 4). To this end, the control logic module must be modified to interpret this new attribute, the rest of the modules of the component work as they have been doing.

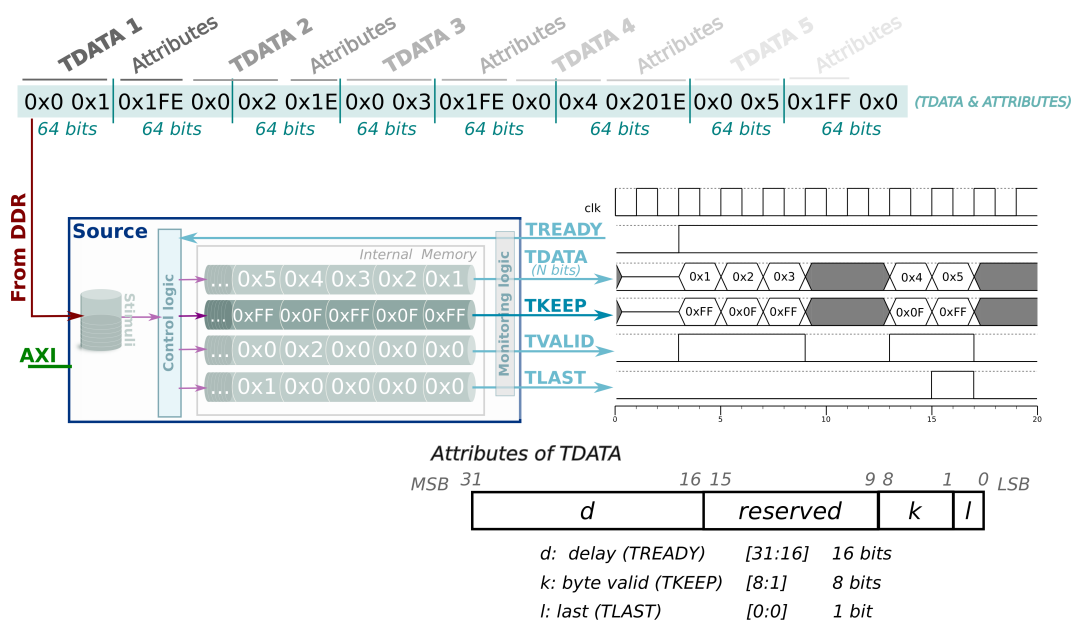


Figure 4. Block and timing diagram example of source component (with TKEEP).

Sink Instrumenting-Component

The sink components compare at run-time the output of the HWacc with the expected values and, also, check the clock cycle in which the output is generated. Thus, our verification framework moves part of the comparison process from test cases to the sink components. To do so, the reference output values are stored in the sink component according to the two following rules.

- The delay value indicates the maximum number of cycles between the current TDATA and the previous one, i.e., the time window in cycles in which the TVALID signal could be set. When the delay is 0xFFFF, it means that TDATA can be received at any time, and it is not checked.
- The rest of attributes contain an extra bit that depicts if such attribute must be checked. For example, the TLAST signal has two bits, one to specify the expected value and the other to point out the action that must be performed. A value of 0 in this second bit means that the attribute must be checked, whilst 1 indicates that the attribute is not considered during the test process.

Like source components, sink components read the reference output vectors and the attributes associated to each data value from external memory (DDR). The sink components also store a trace of failure when the HWacc output mismatches the expected one. Such trace contains an error flag to identify the source of the error: 001 for TDATA error; 010 for TLAST error; 100 for TVALID error. The sink components are also capable of detecting multiple failures in the same sample by an OR operation. For example, 011 means that TLAST and TVALID were wrong.

Figure 5 illustrates an example of the sink component with two timing diagrams: current values on the left (AXI-Stream packet with delays caused by the HWacc and by the TREADY signal) and expected values on the right (AXI-Stream packet without delays). HLS tools would consider both diagrams correct, since co-simulation technology is not able to detect delays or violations of the protocol. By using the proposed verification framework, the designer is able to check the value of TVALID signal and control the TREADY signal to force specific scenarios and look for wrong behaviours of the HWacc. From a technical point of view, the delay attribute of the first expected sample must be assigned to the reserved value 0xFFFF for synchronization purposes. The first sample is the beginning of a transfer and the sink component does not have a reference signal-time arrival from a previous sample. Also, the first sample (0x1) is correct because the value of the data is the expected one and the TLAST flag is not set. In this case the delay is not taken into account because it is the first data. However, the second sample (0x2) arrives one cycle later than expected, so the sink component notifies this failure by denoting the error type found (highlighted in red). The sink component keeps looking for mismatches and, in the actual example, another failure takes place in the last sample. The expected delay must be one, but the current delay is two because the TREADY signal is not set during one cycle (highlighted in orange). The third and fourth samples (0x3 and 0x4) are not failures because the arrival time is correct; we take as the reference the arrival time of the previous samples.

In addition, the sink component handles the TREADY signal to disable the output of the HWacc in order to force borderline scenarios. For example, the internal storage (such as FIFOs and BRAMs) of the HWacc can collapse and, depending on the implementation, the HWacc could either discard values, which may lead to TDATA and TLAST errors, or block the HWacc, which produces TVALID errors. It is worth mentioning that such kind of use cases are impossible to be emulated with the testing facilities provided by current HLS tools. Figure 5 also shows how the sink component handles the TREADY signal as it is specified in the *readyBehavior* array. Starting with 0, each element in the array represents the number of clock cycles the value of TREADY is maintained before complementing it. In this example, after two cycles, it is set during six cycles and then it is cleared for one cycle, causing a failure in the last sample (highlighted in orange).

Furthermore, both components, source and sink, provide more information, such as the number of samples consumed and produced by the HWacc, respectively, and the rate at which the data is consumed and produced. Thus, designers can determine if the speed rate achieved is enough to validate the HWacc or it must be optimized.

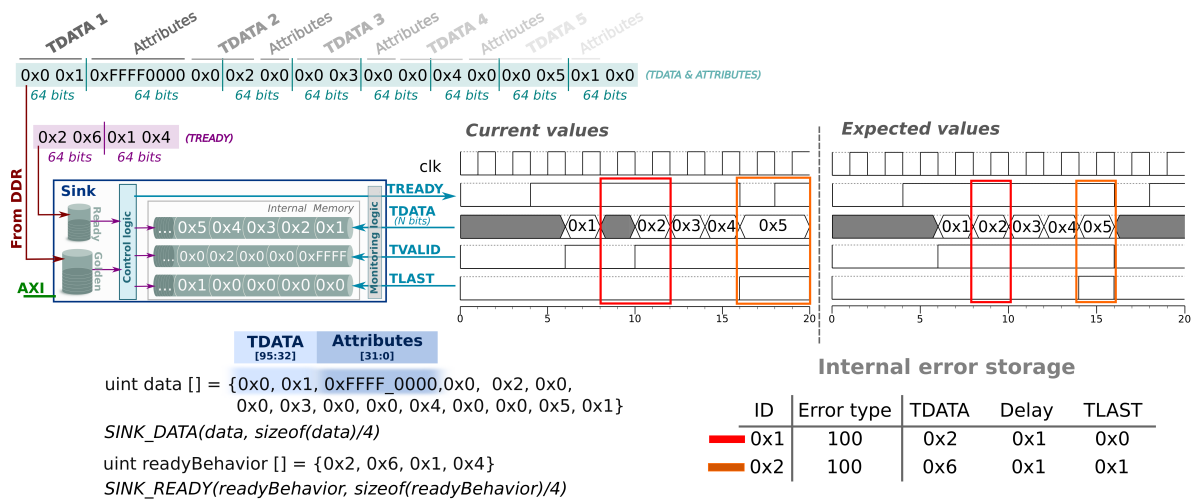


Figure 5. Block and timing diagram example of sink component.

3.1.2. Mapping the HWacc

The source and sink components isolate the HWacc from the rest of the platform. This configuration allows to apply partial reconfiguration techniques to significantly speed up the synthesis process of the whole design. Thus, the on-board verification platform defines a Reconfigurable Partition (RP) where various HWacc can be mapped as Reconfigurable Modules (RM) by using the Dynamic Partial Reconfiguration (DPR) feature present in some FPGAs architectures [17]. In this sense, pre-built parts (checkpoints) of a previous project, which have been synthesized, are reused to reduce the synthesis time of a new HWacc in order to make the FPGA iteration faster. In the case that a HWacc does not fit in the RP, because there are not enough FPGA resources to allocate its logic, the DPR feature is not valid. In any case, the process to generate the configuration file (partial and full bitstreams) is completely automated by our verification framework by means of TCL scripts.

3.1.3. Orchestrating the Verification Process

All components are connected to an orchestrator component, whose objective is to oversee the verification process in the PL part, following the implementation shown in [18,19]. The orchestrator can be controlled, in turn, by the software part of the test case, which runs on the PS part. The designer of the test cases can make it use of some high-level test functions provided as a library:

- The RCUNITY_RESET resets all components in order to assure that the verification environment starts from a well-known state.
- The RCUNITY_START(c) enables the HWacc during c cycles and then halts the system, but the information stored in the source and sink components can be accessed by the test cases. The HELD_HIGH value is reserved to permanently activate the HWacc.

3.1.4. Hardware Verification Environment

In this section, we describe the big picture of the hardware verification platform and an overview of the test flow on the hardware side. The software side is detailed in Section 3.2, where it is explained the extension made on a well-known software testing framework so it can be play the role of test manager in the proposed on-board verification infrastructure.

Figure 6 shows an overview of the architecture layout of the hardware verification platform presented in this manuscript. Such architecture contains an RP (Reconfigurable Partition) in which the HWacc is placed during the place & route stage (see Section 3.1.2). In addition, there are several hardware components around it that allow to proceed with the verification process as follows.

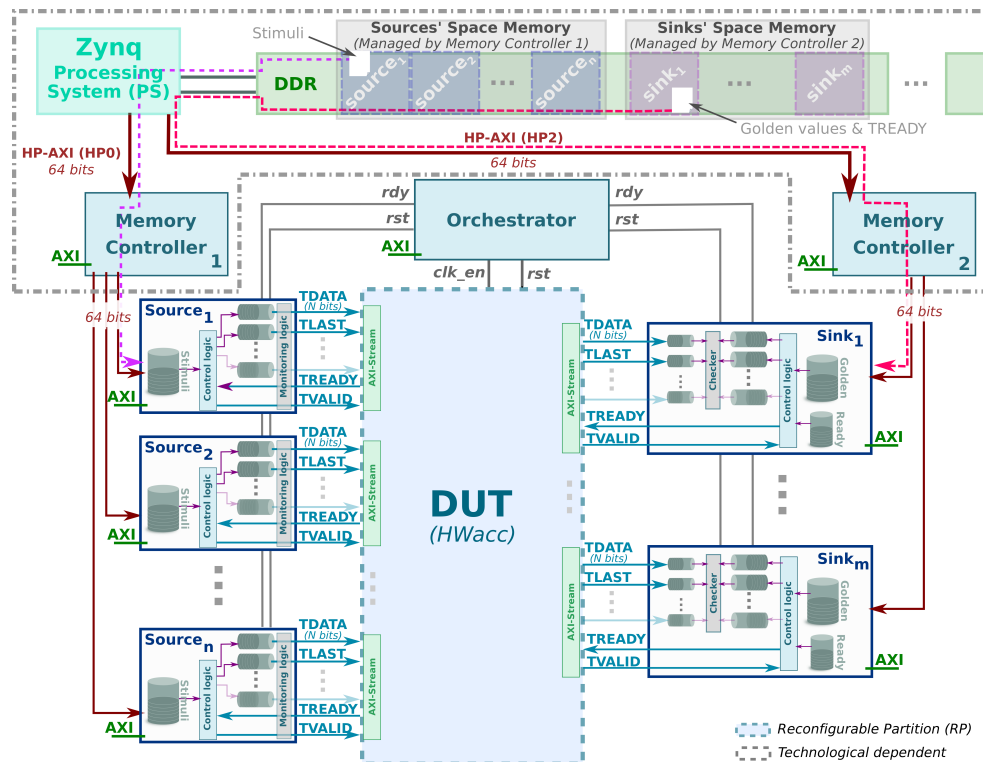


Figure 6. Overview of hardware verification platform implemented on a Zynq-7000 FPGA-SoC.

Firstly, the HWacc (DUT) must be developed using HLS and verified at RTL level by using the facilities provided by the HLS tool. During this step, the functional simulation (timeless) of the C or C++ model takes place, producing a set of intermediate files with the trace of the inputs and outputs. Thus, our hardware verification framework is able to automatically extract the stimuli and the golden values from the files generated by the HLS tool. Besides, the timing annotations, which were included in the test cases through the timing pragma directives, are processed and combined with the values extracted from the timeless model. The result is stored in different regions of the off-chip memory (DDR) in which has been divided and assigned to each source and sink component at design time.

Then, the memory controller components read from the DDR the stimuli and golden values as they are needed. That is, the movement of data from DDR memory is made on demand. Since each source and sink component is independent of each other, they are able to issue read requests to the memory controller independently. The memory controller will be responsible for dynamically scheduling the access to the DDR, trying to minimize the waiting times and maximize bandwidth. To help in this goal, each source and sink component is configured with a threshold value that is used to trigger a new request to the memory controller. If the internal buffer empties below such threshold, the source or sink asks for a new block of data to the memory controller (see pink and red dotted arrows in Figure 6).

It is worth noticing that each source/sink may have a different read pattern of the stimuli/golden data samples. This is highly dependent on the specific application and traffic pattern. Therefore, it is the designer who sets a collection of architectural parameters for such infrastructure at design time so as to get the best tradeoff between the use of resources and the total bandwidth achieved (e.g., size of the internal memory buffers, size of the read request to memory or value of the threshold, etc.). The designer is free to make the best decision based on the aforementioned memory

traffic characteristics for each channel. So, for example, one memory controller can be the hub for five components whilst another can handle just two. Everything depends on the estimated aggregated memory traffic that is expected to be generated by the set of sink or source components that connect to a single memory controller.

The execution of the test will start as soon as the source and sink components have enough data to operate. To this purpose, the orchestrator component freezes the HWacc (i.e. disables the DUT clock) and waits until these components are ready to start. Previously, the orchestrator must be configured as is explained in Section 3.1.3.

Once the hardware verification stage has been finished, the test case retrieves information from the source and sink components concerning metrics about the execution of the test such as the speed rate of each AXI-Stream interface or the number of samples exchanged through such interface. For the sink components, the test case can enquire about the number of errors or mismatches (see Section 3.2). Thus, developers can dispense the use of timing diagrams for analysis purposes, which results in higher levels of productivity.

It must be mentioned that although Figure 6 shows an overview of our hardware verification platform, implemented on a Zynq-7000 FPGA-SoC (Digilent Inc., Pullman, WA, USA), the proposed solution abstracts the verification process from the final target platform. One of the major concerns is the facilities offered by the selected platform to access external memory (where the stimuli will be stored), which may differ in efficiency, services, architecture, etc. In order to overcome this technological challenge, the hardware verification platform isolates the source and sink component from the specificities of the target platform. Thus, the memory controllers are the components that depend on the technology used and are custom implemented to achieve the highest possible performance (see grey dotted box in Figure 6). In this platform, the internal memory of the source components is filled by a memory controller, whilst the internal memory of the sink components is flushed to DDR by another memory controller. From a technical point of view, the memory controllers, implemented on a Zynq-7000 FPGA-SoC, are connected to the off-chip memory through the PS using different HP-AXI buses, HP0 and HP2, that allows them to work in parallel. This fact is possible because there are two physical buses that connect the off-chip memory with the SoC, whilst the Zynq-7000 FPGA-SoC organize them into two pairs of logical buses, i.e., HP0 and HP1 use one physical bus, whereas HP2 and HP3 use the other one. In addition, each memory controller is connected to the PS through two independent AXI-Interconnect components in order to remove the bus arbitration (not depicted in Figure 6). Thus, memory controllers are able to perform several burst transfers of 16 64-bit words to fill, as fast as technology allows, the internal memories of the hardware instrumenting components.

Internally, one memory controller instantiates an AXI Data Mover IP [20] from Xilinx, which could be regarded as the component that implements the core logic of a DMA engine. The AXI Data Mover receives commands through a FIFO-style port that trigger memory transfers. The logic surrounding the AXI Data Mover IP instance is responsible for issuing commands to the AXI Data Mover anytime the data is needed (source) or ready to be written in memory (sink). Figure 7 sketches the block diagram of the memory controller for the Zynq architecture. In the represented use case, three source components are connected to the same memory controller. Each source component has its own FIFO, whose data is being consumed by the DUT. If the amount of data falls below a specific level (configurable for each one), the control logic issues a read command to the data mover. In addition, the control logic plays the role of arbiter whenever two or more FIFOs run out of data at the same time or while there is a pending memory transfer. On top of a configurable definition of the FIFO almost empty thresholds that triggers new read commands, it is also possible to adjust at run time parameters of the memory transfers such as the length of a single data beat or the total length of the burst. This is crucial to minimize the dead cycles when transferring the data between memory and the source components and depends on the specific traffic patterns for each source or sink.

Regardless of the specific technology used to implement the memory controller, it is worth noticing that this component is key in pursuing the goal of keeping the testing infrastructure independent of the target platform. Thus, the main benefit is that sources and sinks are independent of the memory interface used and its parameters (data width, access protocol, etc.).

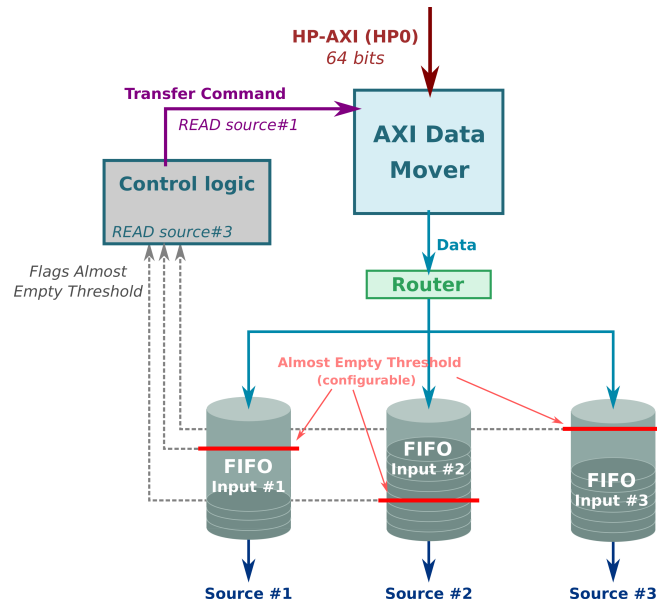


Figure 7. Overview of the memory controller for the ZynQ-7000 FPGA-SoC.

For example, the EK-U1-ZCU104-G (Zynq UltraScale+) board provides completely different hardware facilities as to the access to external memory compared to the ZedBoard (Zynq-7000 SoC). This fact allows us to define a new strategy, while maintaining the same architecture, based on the use of the memory controller. In this new scenario, the best option is based on the use of a single memory controller that provides access to the off-chip DDR memory. All source and sink components of the DUT are connected to this version of the memory controller. The main difference is the protocol access to the DDR, through the user interface protocol of a MIG (Memory Interface Generator) IP [21]. In this sense, the MIG IP replaces the AXI Data Mover for the ZCU104 board and thus the controller must be modified. However, the rest of the testing infrastructure is isolated from this change, making it easier and fast to move from one prototyping platform to another.

3.1.5. Distribution through Docker Images

In order to facilitate the distribution of our hardware verification framework, a set of Docker images have been built. The set of images are warehoused in what is called a Docker Hub. The use of Docker technology greatly facilitates the work of the final users of our hardware verification platform. The final user only has to download a Docker image from the Docker Hub to their computers and create a container (see Figure 8) where they can launch the HWacc verification process. A Docker container packages up libraries, tools, environment variables, code, etc. of our framework and all its dependencies so the verification runs quickly and reliably from one computing environment to another. In this regard, it is considered a lightweight, standalone, package of software that provides an isolated and clean environment to test the HWacc.

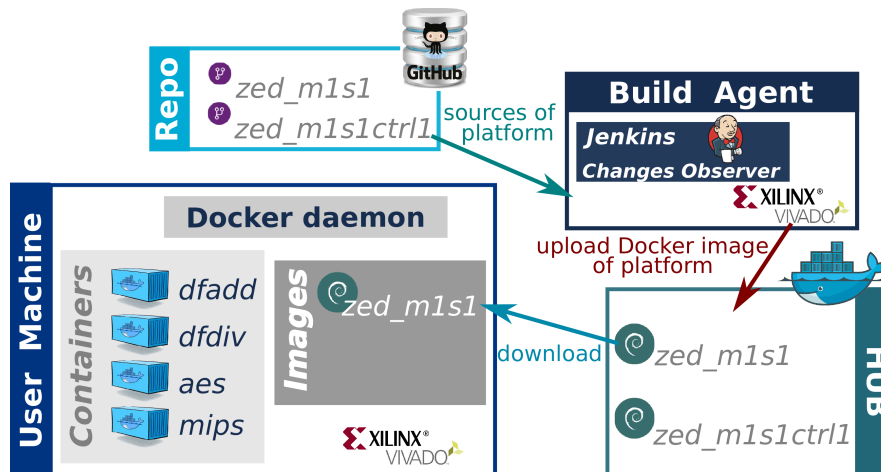


Figure 8. Building Docker images and distribution.

Docker images can be built and released by anyone. The process to create a new Docker image is automated. To this end, the developer of an image must link a GitHub repository, which contains the source code of our hardware verification platform, with a Jenkins Agent (Build Agent in Figure 8). This agent is the main responsible for building the Docker image by synthesizing the static components of the hardware verification platform that will be deployed in the on-board verification stage. Thus, the user of the image is freed from the time consuming task of generating the bitstream for the whole design, only needing to synthesize the DUT that will be deployed in the Reconfigurable Partition (see Figure 6 and Section 4). Finally, the Build Agent automatically uploads the Docker image to the Docker Hub.

3.2. Test Cases across Abstraction Levels

In this section, it is shown how our proposal allows verification activities across abstraction levels, using as the driving use-case the implementation of a HWacc for the *dfadd* CHStone benchmark kernel on a Xilinx ZedBoard FPGA. Following the design flow, designers must start with the description of the test cases to verify the high-level model of the HWacc. Thus, Listing 1 shows one of the forty test cases written for the *dfadd* kernel, which performs a sum operation of two 64-bit words and return the result into another 64-bit word. This test case includes two timing pragma directives that randomly alter the timing behavior of the TVALID signal of the source component and the TREADY signal of the sink component (lines 2 and 3 of Listing 1, respectively).

Apart from the set of timing pragma directives that modifies the timing behaviour of the AXI-Stream signals, there are three checking pragma directives that allow designers to verify the requirements of HWacc that are not feasible in C-Simulation because its timeless model or are not automated in Co-Simulation in which designers must manually analyze the generated timing diagram.

- **delay:** Check that the delay of the HWacc is within a range of cycles, which is depicted by the variables *max* and *min*, whose aim is to denote the maximum and minimum number of cycles to complete the operation, respectively.
- **inputs:** Check the number of input values that the HWacc has consumed. The designer must depict the number of inputs as a range through the variables *max* and *min*.
- **outputs:** Check the number of output values that the HWacc has produced. The designer must depict the number of outputs as a range through the variables *max* and *min*.

By default the *min* variable of all checking pragma directives is 1. The test case listed in Listing 1 includes a checking pragma (line 3 of Listing 1) that restrict the time to complete the operation of the *dfadd* kernel, it must be completed 32 cycles, i.e., the test case will check that the *adder64* function is done within 1 and 32 cycles, thus if the time completion is greater, the test will fail.

Listing 1: Test case for *dfadd* kernel.

```

1 void test_a_n1_b_0_z_n1(){
2     #pragma FIL_SRC TVALID=rand
3     #pragma FIL_SNK TLAST=none TVALID=none TREADY=rand
4     #pragma FIL_CHK delay max=32
5     float64 result = adder64(0xBFF0000000000000,0x0);
6     TEST_ASSERT_EQUAL_INT64(result, 0xBFF0000000000000);
7 }

```

Once the writing of the test cases is complete, our framework automatically generates the test runner, a function that orchestrates the execution of all of them in the software side, following the guidelines of Unity testing framework [13]. Then, the source files are compiled and executed to validate the behaviour of the C code (C-Simulation). In order to validate the RTL model, the HWacc HLS model is synthesized and Co-Simulated, using the same test runner and testbed. Listing 2 shows the Co-Simulation report after execute the same test cases that were executed in C-Simulation; the C-Simulation report is omitted because it is similar to the Co-Simulation one.

Then, in order to perform the on-board verification, our framework launches the synthesis process to obtain the bitstream and adapts the test cases to the new environment. Listing 3 shows how the test case generated from the one in Listing 1 looks like. No new code is needed to be written by the designer, and the stimuli are taken from the test cases.

Listing 2: Co-Simulation report of *dfadd* kernel.

```

...
// RTL Simulation : 39 / 40 [75.00%] @ "5055000"
// RTL Simulation : 40 / 40 [100.00%] @ "5255000"
...
ut_dfadd.cc:437:test_a_n1_b_0_z_n1:PASS
ut_dfadd.cc:444:test_a_n2_b_1p5_z_n0p5:PASS

-----
40 Tests 0 Failures 0 Ignored
OK

```

Nonetheless, the designer can get control over some features of the test process by using different timing pragma directives (e.g., lines 2 and 3 of Listing 1). The first one sets a random delay between stimuli, thus the second TVALID signal is set 16 cycles after the first one because the value 0x00010010 (lines 4 and 5 of Listing 3) indicates that the second value is the last one (0x0001) and it must be set after 0x0010 clock cycles. It must be mentioned the stimuli is composed by two 64-bit words, which are represented as four 32-bit words in the *data_in_1* array of Listing 3, followed by the timing notations, delay and last attributes. Meanwhile, the latter timing pragma directive establishes a random behavior of TREADY signal (line 3 of Listing 3); it will be set low the first 5 clock cycles, high during 2 cycles, low the next 10 cycles and so on. In addition, the same pragma also selects the signals to be checked and its mode, in this case neither the TLAST signal nor the TVALID signal are checked because the reserved value 0xFFFF is present in the golden vector (lines 6 and 7 of Listing 3).

Listing 3: On-board test case.

```

1 void test_a_n1_b_0_z_n1()
2 {
3     unsigned int readyBehavior[4] = {5,2,10,32};
4     unsigned int data_in_1[8] = {0xBFF00000, 0x0, 0x0,
5                                 0x0, 0x0, 0x00010010};
6     unsigned int data_out_1[4] = {0xBFF00000, 0xFFFFFFFF,
7                                 0x0, 0};
8
9     RCUNITY_RESET();
10
11    SOURCE_DATA(data_in_1, sizeof(data_in_1)/4);
12    SINK_DATA(data_out_1, sizeof(data_out_1)/4);
13    SINK_READY(readyBehavior, sizeof(readyBehavior)/4);
14
15    RCUNITY_START(HELD_HIGH);
16
17    PRINT_SINK_SOURCE_INFO();
18
19    TEST_ASSERT_LE_INT(32, SINK_DELAY());
20
21    SINK_ASSERT_PASS();
22 }

```

Furthermore, the designer can add other testing features that are only feasible in the on-board verification by using checking pragma directives, e.g., line 4 of Listing 1. This pragma is translated into an assertion that assesses that the time completion of the HWacc is less or equal than the specified in the *max* variable (line 19 of Listing 3). In addition, the `SINK_ASSERT_PASS` higher-test function (line 21 of Listing 3), which is automatically included in the test cases, assesses three rules; firstly, the number of failures is zero, secondly, the number of input and output samples is at least one, and thirdly, the delay is greater than one. If all are true, the test will pass. Thus, the default requirements imposed by the checking pragma directives are checked.

Later, newly test cases are compiled to get an executable file that runs on the embedded ARM processor in order to exercise the HWacc. Listing 4 shows the report after execute them that displays the number of exchanged samples by the HWacc. In this case, the HWacc processes 2 words of 64-bit width and returns a words of 64-bit width, which are the return value of the *dfadd* kernel. Alongside the exchanged samples, the report also displays the speed rate of transactions, which had taken place between instrumentation components (sources and sinks) and the HWacc, as well as the delay in cycles to complete the kernel operation.

Listing 4: On-board report of *dfadd* kernel.

```

[SOURCE INFO]   Exchanged Samples: 2
[SOURCE INFO]   Digest Rate (words/cycle): 0.512639
[SINK INFO]      Exchanged Samples: 1
[SINK INFO]      Produce Rate (words/cycle): 0.285714
[SINK INFO]      Not failures found
[INFO DELAY]    29
ut_filTest.c:1311:test_test_a_n1_b_0_z_n1:PASS
-----
40 Tests 0 Failures 0 Ignored
OK

```

In order to show that our verification framework detects mismatches between golden values and output ones, the data of the golden output has been increased by one and the checking pragma directive related to the delay is reduced by 10 cycles (`#pragma FIL_CHK delay max=22`); this change will produce a failure. Listing 5 shows the stack of the new report that displays the trace of failure found by the sink component. Such trace, whose identifier is 0, shows that the error is located in the data because the value of the error type is 001. Meanwhile the time completion exceeds the requirement depicted via timing pragma directive, which is shown as a failure message. It is worth mentioning that the delay that shows the stack could be different that the one displayed by the line labeled as `info delay`; the first refers to the time in which the output data is ready and it is individual for each output value, whereas the second is the overall time to process the input values, i.e., from the reading of the first input to the writing of the last output.

Listing 5: On-board report of *dfadd* kernel (failure found).

```
[SOURCE INFO] Exchanged Samples: 2
[SOURCE INFO] Digest Rate (words/cycle): 0.512639
[SINK INFO] Exchanged Samples: 1
[SINK INFO] Produce Rate (words/cycle): 0.285714
[SINK INFO] 1 failures found
Showing stack!!!
[SINK INFO] ID: 0
[SINK INFO] Error type: 1
[SINK INFO] DATA: 0xbff0000000000001
[SINK INFO] DELAY: 0x1C
[SINK INFO] TLAST: 1
[INFO DELAY] 29
ut_filTest.c:1373:test_test_a_n1_b_0_z_n1:FAIL: Delay out of the range (29)
ut_filTest.c:1373:test_test_a_n1_b_0_z_n1:FAIL
-----
40 Tests 1 Failures 0 Ignored
FAIL
```

4. Discussion

In this paper, we present a detailed description of our hardware verification framework that leverages the use of HLS tools for FPGA-based projects. As it was fully discussed in Section 3, the proposed framework extends the features of HLS tools by including the on-board verification stage to complete the FPGA-based design flow. In this section, we would like to also provide a comprehensive analysis of the suitability of the proposed framework and compare it to other solutions.

The statistics in Table 1 are gathered from the execution of test cases related to each kernel of the CHStone Benchmark [22]. The test cases have been Co-Simulated in Vivado HLS and executed on a Xilinx ZedBoard, using our hardware verification framework. Both the minimum and maximum delay for each test case are shown. It is noteworthy the significant error detected in the times obtained for some of the CHStone kernels, which ranges from 5% to 66%. Focusing on the *dfs_in* kernel, which is the one with the highest relative error, the error increases with the number of test cases. It has been observed that the HLS tool builds a queue of stimuli, one entry per input to the HWacc. The latency of a single execution is measured starting from the time the stimuli for such test case is inserted in the queue. Therefore, this fact produces wrong timing profiling, leading the designer to wrong conclusions. Our solution measures the elapsed time just when the stimuli are read by the HWacc, and hence, it provides accurate timing results.

Table 1. Latency statistics from CHStone kernels.

Kernel	Num Tests	Co-Simulation		On-Board Verification	
		Min *	Max *	Min *	Max *
<i>adpcm</i>	2	54	348	11	339
<i>aes</i>	2	3002	6712		4526
<i>blowfish</i>	1	0	29,239		29,243
<i>dfadd</i>	40	8	20	8	19
<i>dfdiv</i>	21	9	317	8	160
<i>dfmul</i>	20	8	23	8	19
<i>dfsin</i>	36	22	6466	21	2166
<i>gsm</i>	1		3647		3646
<i>jpeg</i>	1		405,962	385,030	405,963
<i>mips</i>	1		3536		3535
<i>motion</i>	1	0	215	0	216
<i>sha</i>	1		127,437		127,436

* In clock cycles.

Furthermore, our verification framework obtains the number of samples consumed and generated by the HWacc under test. This information is relevant to determine the correctness of a design, because although the output will match with the reference values, the number of samples consumed could be wrong. Moreover, the verification framework also measures the rate of consumption and data generation by the HWacc. Table 2 lists the rates obtained using two strategies: no delays (testing identical to Co-Simulation) and randomized delays in TVALID and TREADY signals, i.e., the handshaking of the AXI-Stream protocol is randomly modified to observe the behavior of the HWacc. It is worth mentioned the output rates are lower in the most kernels when delays are introduced. However, there are two particular cases (*jpeg* and *sha*) where the verification framework has not been able to obtain the rates because such kernels are blocked when the stream packets contain delays. The Verification IP from Xilinx [23] allows this kind of packets through SystemVerilog but only in a Co-Simulation strategy. The time requirements is critical in some projects because they must meet hard timelines, i.e., the delays can alter the behavior of the HWacc or even result in wrong time responsiveness.

Table 2. Rates from CHStone kernels.

Kernel	No Delays				Random Delays			
	Source		Sink		Source		Sink	
	Samples	Rate ¹	Samples	Rate ¹	Samples	Rate ¹	Samples	Rate ¹
<i>adpcm</i>	100	1	50	1	100	0.521	50	0.345
<i>aes</i>	16	1	16	1	16	0.712	16	0.37
<i>blowfish</i>	6	0.0006	10	0.27	6	0.0001	10	0.127
<i>dfadd</i>	2	1	1	1	2	0.523	1	0.65
<i>dfdiv</i>	2	1	1	1	2	0.48	1	0.71
<i>dfmul</i>	2	1	1	1	2	0.526	1	0.789
<i>dfsin</i>	1	1	1	1	1	0.749	1	0.834
<i>gsm</i>	80	0.5095	84	0.33	80	0.312	84	0.252
<i>jpeg</i>	8192	1	8192	0.379	213	0.0	0	-
<i>mips</i>	8	0.6153	9	0.36	8	0.415	9	0.236
<i>motion</i>	23	0.3285	12	0.273	23	0.228	12	0.173
<i>sha</i>	4096	1	4096	1	145	0.0	0	-

¹ In words per clock cycle.

Unfortunately, the hardware instrumentation for debugging purposes entails a cost in terms of FPGA resources. Table 3 shows the breakdown of resource usage of our solution (see last row), which has been divided into the two types of hardware instrumentation components: source and sink.

Although other works use a different FPGA technology, as well as their experimental results were analyzed with other benchmark than ours, we have compared them focusing on the percentage of the hardware resources used, which are also listed in Table 3.

Table 3. Breakdown of hardware resource use.

	FPGA Family	LUTs	FFs	BRAMs	DSPs	
J.P. Pinilla et al. [6]	NA	NA (0.3%)	NA (10.5%)	-	-	
Y.K. Choi et al. [7]	Virtex-7	170 (0.16%)	-	4 (0.27%)	0 (0%)	
K. Han et al. [8]	Ultrascale	1048 (0.47%)	1401 (0.73%)	-	-	
A. Kourfali et al. [24]	Zynq-7000	1006 (1.88%)	1542 (1.46%)	-	-	
Our	Zynq-7000	Source	679 (1.27%)	973 (0.92%)	3 (2.14%)	0 (0%)
		Sink	967 (1.81%)	1450 (1.35%)	7 (5%)	0 (0%)

It is worth mentioning that our hardware instrumentation components are the ones that need more resources because several reasons. On the one hand, our components contain all logic necessary to operate with the HWacc and with the PS side, whilst the works [6,24] do not include the storage resources in their analysis. These works follow the same strategy, internal signals are routed to the top of the HWacc under test in order to check them later, hence they must be stored on-chip memory or off-chip memory, but unfortunately that information is not provided. In the same line, J. Goeders and S.J.E. Wilton [5] automatically route the internal signals to the on-chip memory, and then store the values of such signals when a store instruction is executed, i.e., a snapshot of the internal signals are stored several times to be analyzed by a software debugger later. However, this strategy entails an area overhead up to 25.4% over the original code [5]. Moreover, these works provide an intrusive solution worsening the frequency of the HWacc, whereas our solution does not cause this effect. On the other hand, Y.H Choi et al. [7] only displays the latency of each function, while K. Han et al. [8] checks that their own protocol is correct in a Network on a Chip (NoC) by increasing a counter when the message is processed by one node of the network.

None of the works analysed provide an efficient way of generating stimuli like our approach does, i.e., our hardware verification framework is capable to model the signal timing behavior from a timeless model. Moreover most works do not check the behavior of the HWacc thorough a standard protocol, such as AXI-Stream. In this regard, the output interfaces of the HWacc can be blocked by the sink components setting the TREADY signal, whilst the works [5,6,24] depends on the depth of the memory in which the output values are stored. In addition, our framework bases the on-board verification stage on the use of pre-built parts of an FPGA-based project and the DPR feature, which makes a reduction of the synthesis time to obtain the (partial) bitstream. Table 4 shows the synthesis time of each kernel. Unfortunately, a set of kernels does not meet the available hardware resources of the RP, but our approach is capable to synthesize the kernel without the DPR feature. Timing savings reach a factor of four in some kernels, and hence the FPGA-loop iteration is considerably reduced.

From the energy consumption point of view, the use of a Docker-based solution and pre-synthesized parts speeds up the (partial) bitstream generation as well as reduces the computing needs. This fact makes that the same host machine will be able to execute several Docker containers in parallel to obtain the corresponding bitstream files. Figure 9 compares the consumption-power (Figure 9a) and consumption-memory (Figure 9b) between our approach and the Xilinx toolchain using its GUI mode for the *dfadd* kernel. The CPU consumption-power is roughly 10% better when our approach is applied, besides our solution takes less time to perform a synthesis process than the GUI mode of Xilinx, so the CPU cores and memory are quickly released, around an average of 7 min per synthesis thanks to run the synthesis process in a Docker container in batch mode. The experiment had been synthesized in a GNU/Linux environment onto a CPU i7-3770 @ 3.4 GHz and 16 GB of RAM.

Table 4. Timing Savings.

Kernel	Fit in RP	DPR Mode	No DPR Mode	Factor
<i>adpcm</i>	No	-	1629.24 s	-
<i>aes</i>	Yes	470.86 s	1536.96 s	3.26
<i>blowfish</i>	Yes	505.27 s	1560.92 s	3.08
<i>dfadd</i>	Yes	413.045 s	1716.41 s	4.15
<i>dfdiv</i>	Yes	424.667 s	1635.39 s	3.85
<i>dfmul</i>	Yes	378.252 s	1719.695 s	4.54
<i>dfsin</i>	No	-	1919.572 s	-
<i>gsm</i>	No	-	1874.575 s	-
<i>jpeg</i>	No	-	2016.65 s	-
<i>mips</i>	Yes	401.24 s	1640.12 s	4.08
<i>motion</i>	No	-	1776.53 s	-
<i>sha</i>	Yes	357.56 s	1615.25 s	4.51
AVERAGE				3.92

In addition to the computational benefits that Docker brings, it adds a well-maintained test environment avoiding the maintain difficulties, such as environment configurations, that provides value add to our hardware verification framework. Thus, engineers can, on their own development machine, spin up exactly the services and dependencies that would be available in production, but with their own hardware design deployed on top. In addition, engineers rarely have time to spend days or weeks “integrating” changes and testing them in a secure environment. Docker increases the productivity of development teams by allowing them to check the changes made in their designs through an automated and clean testing environment, which can be integrated with a Jenkins environment to reach continuous integration feature.

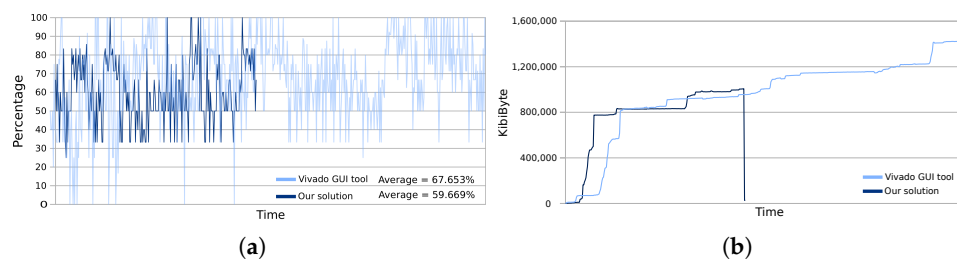


Figure 9. CPU and Memory usage to obtain the bitstream of *dfadd* kernel. (a) consumption-power; (b) consumption-memory.

5. Conclusions

This work presents a solution that extends the HLS tools to perform on-board verification activities following a productive methodology. Particularly, we propose an integral approach that covers all abstraction levels with the aim to preserve the same programming language that is used to describe the behavior of hardware modules. In this sense, the verification process is understood by both hardware and software engineers and avoids the task of observing the signals on the simulator viewer to find the mismatches. Moreover, our Docker-based solution plus the use of scripts opens the complete automation of verification activities related to reconfigurable systems, as well as, keeps up a clean testing environment through the containers.

The experimental results show that the proposed approach makes timing savings by the use of our verification framework, that means it is possible to further launch several runs in parallel with isolate and safe spaces. This opens up future directions towards the continuous integration of reconfigurable systems through specialized software, such as Travis.

Author Contributions: Conceptualization, J.C.; Investigation, J.C. and F.R.; Methodology, J.C.; Software, J.C.; Validation, J.C.; Writing—original draft, J.C. and J.B.; Writing—review & editing, J.C., F.R., J.B., J.A.d.I.T. and J.C.L. All authors have read and agreed to the published version of the manuscript.

Funding: This paper is partially supported by European Union’s Horizon 2020 research and innovation programme under grant agreement no. 857159, project SHAPES (Smart & Healthy Ageing through People Engaging in Supportive Systems). It is also founded by the Ministry of Economy and Competitiveness (MINECO) of the Spanish Government (PLATINO project, no. TEC2017-86722-C4-4-R) and the Regional Government of Castilla-La Mancha under FEDER funding (Symblot project, no. SBPLY-17-180501-000334).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Chen, W.; Ray, S.; Bhadra, J.; Abadir, M.; Wang, L.C. Challenges and Trends in Modern SoC Design Verification. *IEEE Des. Test* **2017**, *34*, 7–22. [CrossRef]
2. Mishra, P.; Morad, R.; Ziv, A.; Ray, S. Post-Silicon Validation in the SoC Era: A Tutorial Introduction. *IEEE Des. Test* **2017**, *34*, 68–92. [CrossRef]
3. Jamal, A.S. An FPGA Overlay Architecture Supporting Rapid Implementation of Functional Changes during On-Chip Debug. In Proceedings of the 2018 28th International Conference on Field Programmable Logic and Applications (FPL), Dublin, Ireland, 27–31 August 2018; pp. 403–4037.
4. Foster, H.D. Trends in functional verification: A 2014 industry study. In Proceedings of the 2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC), San Francisco, CA, USA, 8–12 June 2015; pp. 1–6. [CrossRef]
5. Goeders, J.; Wilton, S.J.E. Signal-Tracing Techniques for In-System FPGA Debugging of High-Level Synthesis Circuits. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2017**, *36*, 83–96. [CrossRef]
6. Pinilla, J.P.; Wilton, S.J.E. Enhanced source-level instrumentation for FPGA in-system debug of High-Level Synthesis designs. In Proceedings of the 2016 International Conference on Field-Programmable Technology (FPT), Xi’an, China, 7–9 December 2016; pp. 109–116. [CrossRef]
7. Choi, Y.K.; Cong, J. HLScope: High-Level Performance Debugging for FPGA Designs. In Proceedings of the 2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), Napa, CA, USA, 30 April–2 May 2017; pp. 125–128. [CrossRef]
8. Han, K.; Lee, J.; Lee, W.; Lee, J. A Diagnosable Network-on-Chip for FPGA Verification of Intellectual Properties. *IEEE Des. Test* **2019**, *36*, 81–87. [CrossRef]
9. Sarma, S.; Dutt, N. FPGA emulation and prototyping of a cyberphysical-system-on-chip (CPSoC). In Proceedings of the 2014 25th IEEE International Symposium on Rapid System Prototyping, New Delhi, India, 16–17 October 2014; pp. 121–127. [CrossRef]
10. Podivinsky, J.; imková, M.; Cekan, O.; Kotásek, Z. FPGA Prototyping and Accelerated Verification of ASIPs. In Proceedings of the 2015 IEEE 18th International Symposium on Design and Diagnostics of Electronic Circuits Systems, Belgrade, Serbia, 22–24 April 2015; pp. 145–148. [CrossRef]
11. Accellera. Portable Test and Stimulus Standard. 2018. Available online: <https://www.accellera.org/downloads/standards/portable-stimulus> (accessed on 24 September 2020).
12. Edelman, R.; Ardeishar, R. UVM School - I want my C tests. In Proceedings of the Design and Verification Conference and Exhibition, Napa, CA, USA, 13 February 2014; pp. 1–8.
13. Karlesky, M.; VanderVoord, M.; Williams, G. A simple Unit Test Framework for Embedded C. Technical Report, Unity Project. 2012. Available online: <http://www.throwtheswitch.org/unity> (accessed on 24 September 2020).
14. Martin, R.C. *Clean Code: A Handbook of Agile Software Craftsmanship*, 1st ed.; Pearson: London, UK 2008.
15. Benevenuti, F.; Kastensmidt, F. *Analyzing AXI Streaming Interface for Hardware Acceleration in AP-SoC Under Soft Errors*; Springer: Cham, Switzerland, 2018; pp. 243–254. [CrossRef]
16. Kachris, C.; Falsafi, B.; Soudris, D. *Hardware Accelerators in Data Centers*, 1st ed.; Springer Publishing Company, Incorporated: Berlin/Heidelberg, Germany, 2018.
17. Lie, W.; Feng-yan, W. Dynamic Partial Reconfiguration in FPGAs. In Proceedings of the 2009 Third International Symposium on Intelligent Information Technology Application, NanChang, China, 21–22 November 2009; Volume 2, pp. 445–448. [CrossRef]

18. Caba, J.; Cardoso, J.M.P.; Rincón, F.; Dondo, J.; López, J.C. Rapid Prototyping and Verification of Hardware Modules Generated Using HLS. Applied Reconfigurable Computing. Architectures, Tools, and Applications. In Proceedings of the 14th International Symposium, ARC 2018, Santorini, Greece, 2–4 May 2018; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2018; Volume 10824, pp. 446–458. [[CrossRef](#)]
19. Caba, J.; Rincón, F.; Dondo, J.; Barba, J.; Abaldea, M.; López, J.C. Testing framework for on-board verification of HLS modules using grey-box technique and FPGA overlays. *Integration* **2019**, *68*, 129–138. [[CrossRef](#)]
20. Xilinx. AXI DataMover v5.1. Available online: https://www.xilinx.com/support/documentation/ip_documentation/axi_datamover/v5_1/pg022_axi_datamover.pdf (accessed on 25 November 2020).
21. Xilinx. Xilinx Memory Interface Generator. Available online: <https://www.xilinx.com/support/answers/34243.html> (accessed on 25 November 2020)
22. Hara, Y.; Tomiyama, H.; Honda, S.; Takada, H.; Ishii, K. CHStone: A benchmark program suite for practical C-based high-level synthesis. In Proceedings of the 2008 IEEE International Symposium on Circuits and Systems (ISCAS), Seattle, WA, USA, 18–21 May 2008; pp. 1192–1195.
23. Xilinx. AXI Verification IP-Xilinx Product Specification (DS941). Technical Report, 2019. Available online: https://www.xilinx.com/support/documentation/ip_documentation/zynq_ultra_ps_e_vip/v1_0/ds941-zynq-ultra-ps-e-vip.pdf (accessed on 24 September 2020).
24. Kourfali, A.; Fricke, F.; Huebner, M.; Stroobandt, D. An integrated on-silicon verification method for FPGA overlays. *J. Electron. Test. Theory Appl.* **2019**, *35*, 173–189. [[CrossRef](#)]

Publisher’s Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



© 2020 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<http://creativecommons.org/licenses/by/4.0/>).