Rapid prototyping and verification of hardware modules generated using HLS

Julián Caba[1], João M.P. Cardoso[2], Fernando Rincón[1], Julio Dondo[1], and Juan Carlos López[1]

[1] University of Castilla-La Mancha, Ciudad Real 13071, Spain,
`julian.caba@uclm.es`
[2] Faculty of Engineering, University of Porto, Porto 4200-465, Portugal
`jmpc@fe.up.pt`

Abstract. Most modern design suites include HLS tools that rise the design abstraction level and provide a fast and direct flow to programmable devices, getting rid of manually coding at the RTL. While HLS greatly reduces the design productivity gap, non-negligible problems arise. For instance, the co-simulation strategy may not provide trustworthy results due to the variable accuracy of simulation, especially when considering dynamic reconfiguration and access to system busses. This work proposes mechanisms aimed at improving the verification accuracy using a real device and a testing framework. 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). In addition it is possible to change some of those parameters, such as clock speed rate, and check the behavior of a hardware component into an overclocking or underclocking scenario. Our on-board testing flow allows faster FPGA iterations to ensure the design intent and the hardware-design behavior match. This flow uses a real device to carry out the verification process and synthesizes only the DUT generating its partial bitstream in a few minutes.

Keywords: FPGA, verification, high-level synthesis, co-simulation

1 Introduction

High-Level Synthesis (HLS) tools are focused on reducing the design gap, as well as the complexity of hardware digital design [1]. Ultimately, using this kind of tools software engineers would be able to accelerate their applications with reconfigurable hardware. In addition, with HLS hardware engineers can work at higher abstraction levels when requirements can be fulfilled without the low-level manual hardware design. Thus, high-level programming languages, such as C or C++, are becoming widespread to describe and speed up user algorithms providing some goodness such as adaptability to changes or shorter *time-to-market* [2]. As a consequence, engineers are able to build different versions of their hardware modules in a short time according to the requirements of the project.

Although the use of HLS tools has boosted hardware design productivity, they entail some issues which have not been solved yet. 1) Most HLS tools include a co-simulation strategy in order to check the correctness of hardware designs described in a high-level programming language, reusing the software test into a co-simulation environment (software tests + simulator tool). However, when one synthesizes the generated RTL and runs these tests in a real device, the behavior may not match the expected results and tests may fail. This is because the co-simulation environment does not take into account some physical aspects (e.g., routing paths, resource locations). 2) Performing in-hardware testing using a real device is a hard task and typically implies the conversion of tests according to the deployment platform. Moreover, the engineer has to build a custom hardware platform, which entails an important development time, so the product's *time-to-market* may be affected. 3) The report given by HLS tools is not fully accurate, since it always reports the worst case. Sometimes that worst case is not present in practice and designs could work with higher clock frequencies, for example. 4) Although HLS vendors provide some techniques, such as pragmas, in order to drive the translation process, engineers loss control over generated designs. Engineer's experience plays an important role to generate desired designs. In addition, the visibility is lower when one works at higher abstraction levels.

Because of these limitations, current HLS tools and design flows are not enough to fully verify and perform accurate design space exploration of designs. Tests must be re-written and a handmade verification platform must be built for each hardware design. In addition, simulations close to real scenarios would be desirable in order to ensure the correctness of the hardware design generated from a high-level programming language, both considering the effect of physical parameters and reducing the limitations imposed by HLS tools.

The main contribution of this work is an approach to extend HLS verification, based on software testing techniques, but including design deployment on real hardware as part of the process. Besides the verification of the correctness of the design, the methodology can be used to explore the different physical constraints, and its behavior under different conditions than those used as restrictions during the synthesis process. To achieve our main objective our approach provides the following aspects:

- A *testing framework* to check Design Under Test (DUT) behavior following the point of view of software testing frameworks and using macro assertions.
- Bring *physical parameters* into verification, tests support the configuration of certain physical parameters to obtain results from a real scenario. For instance, tests may configure the clock rate in accordance with the profiling of HLS tools or overclocking the solution generated by these tools.
- An *on-board platform* to handle the verification process on real hardware, providing real timing results, and able to configure the clock rate using software instructions in order to observe the design behavior under overclocking/underclocking conditions. This platform is remotely accessible and should decouple 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 development flow, which is implemented on the architecture presented in Section 4. In Section 5, we present some results using our approach in the context of a case study. Finally, Section 6 summarizes this paper and proposes directions for future work.

2 Related Work

During the last decade, FPGA vendors have introduced new HLS tools in order to reduce the complexity of hardware design, thus allowing engineers to work at higher abstraction levels. However, problems arise when engineers try to check the correctness of their designs. The accuracy of the simulation depends on the different levels of abstraction used during the modelling of a SoC [3]. The highest simulation effort is required at High-Level Modelling, but it usually results in inaccuracies. On the other hand, the prototype is the perfect environment to carry out the verification process, but it usually is very hard and costly.

One of the advantages of working at high levels of abstraction is that verification tasks, such as writing Non-Regression Tests (NRT), can be performed much easier (see, e.g., [4]), and can benefit from the use of higher productivity tools such as testing frameworks. Some research efforts (see, e.g., [6] and [5]) propose formal methodologies, such as UVM (Universal Verification Methodology) [8]. As this methodology entails a complex and hard effort, researchers have focused on techniques to make it simpler and reduce the effort needed ([6] and [5]). As randomized stimuli are considered to be undesirable and difficult for inexperienced engineers, authors [7] propose to reuse the C test cases in the UVM environment. Thus, tests are kept over all the verification levels. Nevertheless, high abstraction techniques are focused on testing the functional behavior of a design, applying a verification methodology or not, into a particular scenario because most of those tests do not take physical parameters into account.

On the other hand, other approaches propose the use of real devices for verifying hardware designs. For instance, [9] introduces a hardware verification platform in order to overcome RTCA/DO-254 verification challenges. However, their solution requires the synthesis of the whole system. Conversely, [10] includes partial reconfiguration and validates synthesized hardware designs with the original model, using a black-box verification approach. In the same context, [11] proposes a verification framework to verify hardware designs directly in an FPGA instead of using simulations. They do not consider the synthesis process and use a *Network Storage* which contains the bitstreams, the input test vectors and the golden reference. It is clear that using a real device for the verification and exploration of the performance of a design entails new challenges, such as building a platform, its synthesis, or communication between the test framework and the DUT, among others.

Indeed, from high-level simulation to in-hardware simulation there is an important gap. In order to narrow that gap, some efforts provide intermediate approaches overcoming several physical parameters from HLS tools. The authors in

[12] consider multi-cycling in HLS contexts and use software profiling to guide multi-cycling optimizations. In [13], they perform multi-cycle optimization on chained functional operations. Their approach couples HLS and logic synthesis synergistically so multi-cycle paths can be identified and optimized coherently across both behavioral and logic levels.

Our approach relies on a mixture of high abstraction levels with low ones to ensure the correctness of hardware designs generated from a high-level programming language through HLS, allowing physical modifications, such as clock rate and check the design with overclocking/underclocking configurations.

3 On-board testing flow

Our proposed testing flow is based on the three typical steps. The flow starts from C or C++ code written by the user, plus some functional tests. In the **software domain**, including embedded systems, testing frameworks to verify the software artifacts are very widespread, so we propose the use of *Unity* testing framework [14]. *Unity* can easily be expanded building new macros. Most frameworks have a variety of assertions which are meant to be placed in the test to verify the production code. For instance, the `TEST_ASSERT_EQUAL(expected, value)` macro checks the equality between the expected value and the one returned.

Once the module is tested in a purely software domain, we consider that an HLS tool is used to translate the high-level code into an RTL description. Although the flow could be easily adapted to any other tools such as LegUp [1] or AUGH [11], in this work we use Vivado HLS from Xilinx [15]. At this stage, we can run the tests again without any change (**co-simulation stage**).

The third step generates the configuration file (partial bitstream) and deploys it to the **on-board** platform. In order to carry out an in-hardware testing process, our platform contains a dynamic area and we use Dynamic Partial Reconfiguration (DPR). Hence, we provide a reference synthesized platform with some checkpoints in order to synthesize only the logic that is programmed into the dynamic area. For this, we include a TCL script following the Xilinx approach for the overall generation process management. This TCL script imports a reference platform and adds RTL description files, customizing a new configuration. Summarising, our TCL scripts automate the partial bitstream generation process from high-level descriptions. Thus, developers do not have to build a new platform or regenerate their IPs, moreover, our approach allows testing designs in a real devices in few minutes (FPGA iterations) as our case study shows.

This stage introduces testing novelities, the tests can be modified to annotate the physical parameters, such as the clock rate to be used or the number of cycles the clock enable must be set active (lines 4 and 5 of Listing 1.1). To carry out these tasks we have extended the *Unity* testing framework with the inclusion of a number of configuration macros (see Table 3). In addition, to ensure that sequential processes of our DUT starts in a known state, we add a special macro that resets the whole module (DUT) before exercising it (line 6 of Listing 1.1).

Table 1. Configuration macros of *Unity* extension.

Macro	**Description**
`UNITY_RESET`	Sends a reset signal to the DUT (dynamic area)
`UNITY_START`	Enables the dynamic area during the cycles depicted in `CONFIGURE_UNITY_CLK_EN` macro
`CONFIGURE_UNITY_HW_ADDR(addr)`	Configures the hardware address where is mappped the *Hardware Manager* component. By default `0x41000000`
`CONFIGURE_UNITY_IGNORE_INPUT(words)`	Configures the input 32-bit words that should be ignored. By default 1
`CONFIGURE_UNITY_IGNORE_OUTPUT(words)`	Configures the output 32-bit words that should be ignored. By default 1
`CONFIGURE_UNITY_CLK_EN(cycles)`	Configures the number of cycles that the `clken` signal is active-high. 0 means that the `clken` signal is always active-high
`CONFIGURE_UNITY_CLK_RATE(clk)`	Configures the clock frequency. Allowed inputs are: 33, 66, 100, 200 and 400. By default 100 MHz

On-board verification entails another important problem: the communication between different domains. As in this case the module is running in a real device, our approach includes a special function for the transfer of stimuli between the testing environment and the hardware module, the input and output sizes are 16 32-bit words in accordance with a 4x4 window (line 7 of Listing 1.1). Following the testing framework philosophy, we have also added new macros to measure the time in clock cycles spent by the module (line 8 of Listing 1.1). Therefore, Unity extension keeps the same testing technology independently of the module abstraction level, and therefore, the development status. Indeed, macros are added to tests when one wants to check or configure some physical parameters.

Listing 1.1. Example of on-board test with *Unity* extension

```
 1  void
 2  test_module(){
 3  #ifdef HW_TEST
 4    CONFIGURE_UNITY_CLK_EN(200); // 200 cycles active-high
 5    CONFIGURE_UNITY_CLK_RATE(100); // 100MHz
 6    UNITY_RESET(); // Reset Module
 7    result = moduleDUT(stimuli)
 8    TEST_ASSERT_TIME_LT(50); // Checking time
 9  #else
10    result = moduleDUT(stimuli);
11  #endif
12    for(int i=0; i!=16; i++)
13      TEST_ASSERT_EQ(reference[i], result[i]);
14  }
```

4 Architecture overview

One of our goals is to provide an on-board platform for testing the modules generated by HLS tools, and improve the static analysis provided by them. Our on-board platform relies on a SoC platform which integrates an FPGA and a hardcore-processor in the same device. In our case, we use a Zedboard from Xilinx [16].

This platform allows an easy communication between the hardcore-processor and the logic part. In addition, the Zedboard is connected to a computer network via Ethernet in order to communicate with the workstation where hardware modules have been developed using HLS tools. The testing framework is remote too, thus an FPGA plays the role of a remote service.

4.1 Hardware Architecture

Fig. 1 shows the Programmable Logic (PL) architecture layout of our on-board platform environment in the Zedboard. The module to verify, the DUT, is programmed in this side and its communication with the Processing System (PS) side is done through two FIFO channels connected to an AXI interface. This AXI bus connects both PS and PL sides, tunneling a master-slave communication, where PL is the master and the components programmed in PL area are slaves.

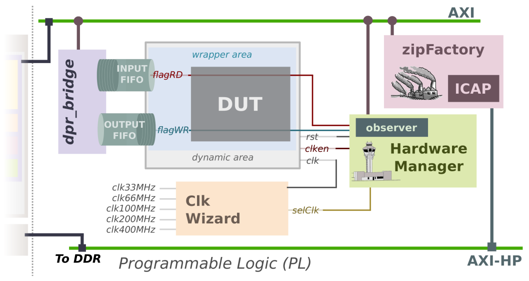


Fig. 1. Programmable Logic architecture layout

The PL side is divided into two parts, a static one which contains those components that do not change independently of the Design Under Test (DUT), and the DUT, which could be a part of an image filter, a cypher algorithm, etc. The DUT is also deployed into a dynamic reconfigurable area, while the rest of the layout does not change. One of the advantages of the use of DPR is that it reduces the synthesis process and improves synthesis tasks [17] due to working with partial bitstreams. Another benefit motivated by DPR is the fact that engineers can dynamically insert new functionality without redesigning the whole system or moving to a bigger device. Furthermore, it is possible to adapt an FPGA to different scenarios, by just modifying the functionality or the performance of some related tasks [17].

The *zipFactory* component programs (physically deploys) a partial bitstream into the dynamic area available in our on-board platform without the microprocessor intervention. It retrieves bitstream data from the DDR which would have been previously stored into a memory location. This component is able to recognize a desynchronization bitstream word, so it stops reading at that moment. The retrieved data 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.

As we mentioned above, the DUT is connected to an AXI bus through two FIFO channels. This 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. The current platform bridges the AXI data whose 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. Besides, a clock enable signal has been added in the dynamic area interface in order to manage when the DUT should be active. This enable signal does not affect the *dpr_bridge* component, so it is able to carry out its tasks independently: fill the input FIFO and empty the output FIFO. Indeed, a *start macro* is executed during the *moduleDUT* function to enable the DUT.

The *Hardware Manager* component carries out the following hardware tasks (each task is controlled from the test using the macros mentioned in the previous section) that would not be feasible or would result in a poor accuracy when performed in the software testing environment.

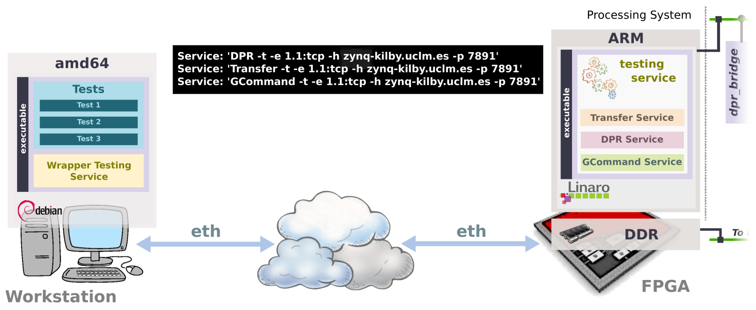
- It resets the dynamic area in order to assure that internal signals of our DUT start from a well-known state. In addition, one can send a reset in the middle of a DUTs operation in order to know its behavior in that scenario.
- It sets active-high the clock enable signal during the cycles related to the configuration of the test (e.g., line 5 of Listing 1.1). It is active-low until the macro `UNITY_START()` is called upon. Engineers can set the number of cycles that the clock enable must be active-high using the `CONFIGURE_UNITY_CLOCK_EN(cycles)` macro.
- It sets the clock rate of the dynamic part and those static modules that interact with it at runtime, such as *clk_rd* of input FIFO. The available clock speed rates are: 33MHz, 66MHz, 100MHz, 200MHz and 400MHz. Thus, engineers have only to indicate which clock rate will be used through the `CONFIGURE_UNITY_CLK_RATE(clk)` macro. This feature provides a flexible environment in which engineers do not need to build a new platform or modify a previous one for the verification of their designs.
- It measures the time elapsed by the tasks performed by a DUT. The *Hardware Manager* component observes transactions between the input/output FIFOs and the dynamic part as a spy. Thus, this component is able to know how many transactions take place between both parts and when they happen. It works increasing an internal counter which plays the role of a chronometer. This chronometer is triggered by transactions that take place between the input FIFO and the DUT (input transactions), whereas it is

stopped by transactions that take place between the DUT and the output FIFO (output transactions). The extension of *Unity* testing framework allows us to configure the number of input transactions and output transactions that may be ignored. Therefore, the *Hardware Manager* component must be configured from the test denoting the number of 32-bit words to be ignored during input transactions and, on the other side, the number of 32-bit words of output transactions which must also be ignored. By default, both values are one and we can explicitly configure them in the test source code, e.g., Listing 1.1, the following two configuration macros `CONFIGURE_UNITY_IGNORE_INPUT(words)` and `CONFIGURE_UNITY_IGNORE_OUTPUT(words)`.

In order to spy the transactions between both FIFOs and the DUT, our platform adds two special signals: *flagRD* and *flagWR*. In our case study both are connected to the input FIFO and output FIFO, respectively. However, both signals can be connected to another sources, modifying the wrapper which adapts the dynamic area interface to the DUT interface.

4.2 Software Architecture

Fig. 2 shows the PS architecture of our on-board platform, the developer's workstation side and the communication between them. Messages are sent through a computer network using the zeroC Ice communication middleware [18].



```
Service: 'DPR -t -e 1.1:tcp -h zynq-kilby.uclm.es -p 7891'
Service: 'Transfer -t -e 1.1:tcp -h zynq-kilby.uclm.es -p 7891'
Service: 'GCommand -t -e 1.1:tcp -h zynq-kilby.uclm.es -p 7891'
```

Fig. 2. Communication overview

On the FPGA side, the PS contains an ARM processor which runs an embedded operating system (in the experiments we use an OS based on Linaro Ubuntu distribution from a SD card). Over this OS, we deploy three services that enables the use of internal hardware components from outside. For instance, when the partial bitstream is ready we can send it to the FPGA. We choose the network as the channel to carry out this task since the FPGA can be a shared resource, thus we provide the mechanism to share transparently a hardware resource, increasing its availability and accessibility. In addition our reconfiguration engine (*zipFactory* component) is faster than other proposals as shown in the results section. Hence, to send a partial bitstream we may use the *transfer* service, that stores the data sent into a default memory address. Then we may trigger a reconfiguration process through the *DPR* service. 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.

In the engineer's workstation, one may use software wrappers of these three services in order to marshall and unmarshall the messages. These services are shown as functions from the testing framework point of view. Indeed, the `moduleDUT` function in Listing 1.1 serializes stimuli into zeroC Ice messages. In addition, `moduleDUT` function deserializes data retrieved from the FPGA too. Note that *stimuli* and *result* are streaming data that will be retrieved and stored from the input FIFO and to the output FIFO respectively.

5 Use Case

Our approach has been developed under a GNU/Linux environment, and has been implemented on a Xilinx Zedboard platform. We use here a case study based on the histogram of oriented gradients (HOG). By default, the platform works at 100 MHz, but the dynamic area and other components can modify this clock speed rate at runtime. 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 several normalization factors and solutions, e.g., single or double precision [19]. Each solution has been developed in the C programming language and using Vivado HLS in its version 2015.4.

We consider two block normalization factors: l^2-*norm* and l^2-*hys* with a 4x4 window as input and output. Thus, the input and output of both algorithms are 16 pixels. Table 2 shows the hardware resource comparison between the results reported by Vivado HLS and the results after place and route process for both algorithms. In addition, we considered some improvements in order to achieve higher performance without a high hardware resource cost. For instance, the difference between *original* and *improved* versions are not very far in hardware resources, whereas the latency is much lower in the second version. The *improved* solution contains some *pragma* in order to pipeline and unroll the solution. Moreover, we consider implementations with single and double floating-point precisions.

Table 3 timing and frequency summary of each solution that Table 2 shows. The table campares the HLS report with the on-board report. The maximum frequency is controlled using the `UNITY_TIME_CLK_RATE` macro, while the latency, throughput and execution time are measured from the *flagRD* and *flagWR* signals which are connected to FIFO channels and are activated when the first transaction happens. All versions are tested with the same hardware platform, generating each partial bitstream, sending them through the network using the

Table 2. Comparison between Vivado HLS and after Place and Route reports

		Solution	HLS report				P&R report			
			BRAMs[2]	DSPs	FFs	LUTs	BRAMs[2]	DSPs	FFs	LUTs
l^2-*norm*	32b[1]	Original	0	5	1904	2628	0	5	1710	1764
		Improved (II=1, Factor=2)	6	5	2039	2777	6	5	1942	1886
	64b[1]	Original	4	14	6403	8123	4	14	6124	5942
		Improved (II=1, Factor=2)	10	14	6834	8415	6	14	6758	6373
l^2-*hys*	32b[1]	Original	0	5	2129	3160	0	5	1851	2040
		Improved (II=1, Factor=2)	4	5	2334	3416	2	5	2144	2320
	64b[1]	Original	6	14	6828	9149	6	14	6343	6312
		Improved (II=1, Factor=2)	12	14	7460	9685	8	14	7250	7036

[1]Float-Point precision. [2]BRAM18K.

services deployed in the FPGA where, they finally are exercised. Contrasting the results in Tables 2 and 3, we can observe some differences between the results provided by the HLS tool and the real results provided by our platform.

Table 3. Summary of latencies, execution times and throughput

		Solution	HLS		On-board				
			Latency[2]	Max. Freq.[3]	Freq. Used[3]	Latency 1^{st}Output[2]	Latency[2]	Throughput[2]	Exec Time/ Freq. Used
l^2-*norm*	32b[1]	Original	329	122	200	295	385	385	1925
		Improved (II=1, Factor=2)	141	114	200	111	171	155	855
	64b[1]	Original	507	116	200	457	577	577	2887
		Improved (II=1, Factor=2)	166	60	100	152	242	210	2420
l^2-*hys*	32b[1]	Original	602	122	200	568	658	658	3290
		Improved (II=1, Factor=2)	234	112	200	219	279	263	1395
	64b[1]	Original	910	122	200	860	980	980	4900
		Improved (II=1, Factor=2)	316	60	100	302	392	360	3920

[1]Float-Point precision. [2]In clock cycles. [3]In MHz.

On the other hand, we can change the clock period in the HLS tool in order to force it to fulfill the specified time requirement. Then we can use our platform independently of the clock period target to ensure that the design works, or we can even overclock or underclock the DUT.

Engineers can use our platform and design flow based on TCL scripts to verify their designs, or new design versions modifying the target clock period or adding new optimizations through *pragma* directives, in a real device waiting a few minutes. Any of the solutions of our case study takes about 2 min. and 30 sec. to generate one partial bitstream from its high-level description. In addition, the processes for configuring the FPGA and sending the stimuli take only three seconds. Thus, engineers can make faster FPGA iterations to ensure their designs intention and their hardware-designs behavior match.

Our approach allows measuring the real throughput of streaming designs and the related time of each result. For instance, the l^2-*norm original* solution of our case study for single precision version takes 295 cycles to generate the first result (*execution time*), as shown in Table 2. The output rate of this version is 6 cycles, thus the second pixel is written at cycle 301 and so successively to complete the output window. Therefore, this solution takes 385 cycles to process an input window of 16 pixels (*latency*). In this solution, the *throughput* is 385 cycles because we do not apply any improvement to build a dataflow.

One important part of our platform is the reconfiguration engine. This component enables to run tests very fast since the hardware module may be sent to an FPGA through the network. Comparing our approach with other controllers (e.g., the ones presented by [20], [21], [22] and [23]) we achieve a configuration rate about 387.59MB/s, very close to the theoretical (400MB/s), using the half of resources - *zipFactory* core uses 272 FFs, 586 LUTs and 2 BRAMs - that the best solution [22].

6 Conclusion

In this paper we presented a development flow and on-board platform to provide unit testing to hardware modules, considering both functionality and timing issues. Our approach is well-suited for both software and hardware developers. We propose the use of software macros embedded in test cases to program physical parameters such as operating clock frequency. Hardware modules depicted in a high-level programming language are translated into a programmable file automatically through some TCL scripts. In addition, our proposal provides a remote and transparent dynamic reconfiguration service, offering FPGA as a verification service. Thus, we can exercise a design under test (DUT) remotely, breaking down the test from the hardware prototype.

Engineers only need to add a few physical parameters into their tests in order to check their hardware modules on a real device. Engineers should be able to modify the clock speed rate without a high effort, thus the correctness of their hardware modules can be verified in the context of an overclocking or underclocking environments. This can be very useful as it can test if a design works with a higher speed clock rate than the one reported by HLS tools, especially when all input values to be used do not trigger the worst case. In addition, in our approach the original software tests are kept as much as possible during the development life-cycle of hardware modules based on HLS.

Future work will be targeted to synthesizable hardware assertions, in order to enhance real-time verification capabilities in our platform and raise the visibility of internal signals, which are synthesized by HLS tools and are too difficult to be traced from 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 et al., *From software to accelerators with LegUp high-level synthesis*, International Conference on Compilers, Architecture and Synthesis for Embedded Systems, 2013.
2. J. Cong, B. Liu et al., *High-Level Synthesis for FPGAs: From Prototyping to Deployment*, Computer-Aided Design of Integrated Circuits and Systems, 2011.
3. L. Gong and O. Diessel, *Functional Verification of Dynamically Reconfigurable FPGA-based Systems*, Springter, 2015.
4. H. Hoffman, *Non-Regression Test Automation*, PNSQC, 2008.
5. J. Podivinsky, M. Simkova, O. Cekan and Z. Kotasek, *"FPGA Prototyping and Accelerated Verification of ASIPs"*, International Symposium on Design and Diagnostics of Electronic Circuits Systems, 2015.
6. Y.N. Yun, J.B. Kim, N.D. Kim and B. Min, *"Beyond UVM for practical SoC verification"*, International SoC Design Conference, 2011.
7. R. Edelman and R. Ardeishar, *"UVM SchmooVM - I want my c tests!"*, Design and Verification Conference and Exhibition, 2014.
8. Accellera Organization, *Standard Universal Verification Methodology Class Reference Manual, Release 1.1*, Accellera, 2011.
9. L. De Luna, Z. Zalewski, *FPGA Level In-Hardware Verification for DO-254 Compilance*, Digital Avionics Systems Conference (DASC), 2011.
10. Y. Iskander, S. Craven et al., *"Using partial reconfiguration and high-level models to accelerate FPGA design validation"*, International Conference on Field-Programmable Technology, 2010.
11. A. Wicaksana, A. Prost-Boucle et al.,*"On-board non-regression test of HLS tools targeting FPGA"*, International Symposium on Rapid System Prototyping, 2016.
12. S. Hadjis, A. Canis et al.,*"Profiling-Driven Multi-Cycling in FPGA High-Level Synthesis"*, Design, Automation, Test in Europe, 2015.
13. H. Zheng, S. T. Gurumani, L. Yang, D. Chen and K. Rupnow, *"High-level synthesis with behavioral level multi-cycle path analysis"*, FPL, 2013.
14. M. Karlesky, M. VanderVoord and G. Williams, *"A simple Unit Test Framework for Embedded C"*, Unity.
15. Xilinx Inc. *"Vivado Design Suite User Guide: High-Level Synthesis"*, Xilinx, 2014.
16. AVNET, *ZedBoard: Hardware User's Guide*, AVNET, 2014.
17. C. Kao, *"Benefits of Partial Reconfiguration"*, Xcell, 2005.
18. https://zeroc.com/
19. N. Dalal and B. Triggs, *"Histograms of Oriented Gradients for Human Detection"*, Computer Vision and Pattern Recognition (CVPR), 2005.
20. M. Hubner, D. Gohringer et al., *"Fast dynamic and partial reconfiguration Data Path with low Hardware overhead on Xilinx FPGAs"*, International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum, 2010.
21. P. Manet, D. Maufroid et al., *"An Evaluation of Dynamic Partial Reconfiguration for Signal and Image Processing in Professional Electronics Applications"*, EURASIP Journal of Embedded Systems, 2008.
22. K. Vipin and S. Fahmy, *"A high speed open source controller for fpga partial reconfiguration"*, International Conference on Field-Programmable Technology, 2012.
23. J. Tarrillo, F.A. Escobar, F. Lima and C. Valderrama, *"Dynamic Partial Reconfiguration Manager"*, Latin American Symposium on Circuits and Systems, 2014.