IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received October 20, 2021, accepted November 28, 2021, date of publication December 1, 2021,
date of current version December 13, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3132188

An Integrated Digital System Design Framework With On-Chip Functional Verification and Performance Evaluation

GERMAN CANO-QUIVEU, PAULINO RUIZ-DE-CLAVIJO-VAZQUEZ, MANUEL J. BELLIDO-DIAZ, DAVID GUERRERO-MARTOS, JULIAN VIEJO-CORTES, AND JORGE JUAN-CHICO

Department of Electronic Technology, University of Seville, 41012 Seville, Spain

Corresponding author: German Cano-Quiveu (germancq@dte.us.es)

This work was supported in part by the Ministerio de Industria y Competitividad of Spain under Project TIN2017-89951-P (BootTimeIoT), and in part by the European Regional Development Fund (ERDF). The work of German Cano-Quiveu was supported by VI Plan Propio de Investigacion y Transferencia de la Universidad de Sevilla (VI-PPITUS).

ABSTRACT This paper introduces a design and on-chip verification framework for IPCores in FPGA platforms. The methodology of the proposed framework is based on the development of a high level software model, an HDL description of the IPCore and the verification of the system under test by the Autotest Core, an on-chip verification core developed for this framework. The test pattern generation is done at the high level in software and used throughout the design and verification process. HDL simulation results can then be compared to on-chip results and get performance measurements from the Autotest Core. The Off-line testing is possible by using standard low-cost Flash storage (SD card). The proposed framework and methodology applied to PRESENT and SPONGENT cryptographic algorithms has shown over two orders of magnitude better performance than commercial tools like Xilinx's VIO and a hardware footprint of the verification cored below 3% of the available FPGA resources.

INDEX TERMS FPGA, framework, HDL, IoT, IPCore, on-chip, performance, verification.

I. INTRODUCTION

Nowadays embedded systems such as those found in smartphones, smart cities, medical devices, home automation or security systems are part of our daily life. Thus the Internet of Things (IoT) is one of the most active fields of research. It has been estimated than there will be more than 64 billion IoT devices connected in 2026 [1]. The increasing IoT demand has created new market opportunities, specifically in FPGA that is expected to reach a value of 7.5 billion $ by 2030 [2]. This has hardened time-to-market constraints, which is a problem despite the availability of external Intellectual Property Cores (IPCores) and high-level-software techniques and tools such as MyHDL [3], PyRTL [4], CocoTB [5] or VIvado HLS [6]. The impact is remarkably severe in terms of verification and validation, since these processes take a large portion of the development time. Because of this, bug implementations are not uncommon. In 2020 only 17% of the projects were in production with no known bugs [2]. Therefore, it is imperative to speed up the verification process. That is a challenging task, since circuit verification is a complex issue and a field of study of its own. A crucial task included in verification is functional verification. The objective of functional verification is to make sure the system carries out the task it was designed to. In the case of a digital system, during functional verification, it is fed with a set of input vectors and the corresponding outputs are compared with precomputed free-fault values. Obviously, for the most part this cannot be carried out comprehensively since the size of the input space grows exponentially with the input vectors length. A set of pseudo-random input test patterns of manageable size is then chosen so that, if the system produces the corresponding outputs, then the probability of a fault in the design is low. The device or algorithm generating these patterns is called a Test Pattern Generator (TPG), and the system fed by them is the Core Under Test (CUT). If the verification fails, inspecting the internal signals of the system is then convenient in order to find the source of the flaw. If the procedure or device employed to carry out the verification makes it possible to inspect such internal signals, then it is said to provide white-box-testing [7]. In contrast, only the external signals are monitored in black-box-testing.

The associate editor coordinating the review of this manuscript and approving it for publication was Liang-Bi Chen.

One of the reasons why verification takes so much time and resources is that it must be carried out at every stage of the development cycle. Automated procedures can apply the same test patterns at every stage in order to minimize this time. If the system has been described using a Hardware Description Language (HDL), this can be carried out at the earliest stages through simulation. However, simulation is very time consuming, making it infeasible when the size of set of test patterns is high. Also, some flaws can only be detected once a physical instance of the CUT is available. Verifying a physical instance can be faster and more reliable, but it requires additional devices. For example, dedicated apparatus called Automated Test Equipment (ATE) can be used to test a physical instance. ATE can be very powerful, but also expensive and bulky. Because of that, in field verification with ATE can be difficult or impossible. Alternatively, the CUT and the verification device can be in the same board or even the same Integrated Circuit (IC). Several Verification IPCores (VIP) are available for this. Obviously, On-Chip verification devices can be expedited by using Reconfigurable Hardware (RH) such as FPGA. An example of On-Chip verification devices are Internal Logic Analysers (ILA). ILAs are less expensive than external logic analysers and do not require an out-chip to monitor signals. However, they consume a remarkable share of the IC resources. Another example is the Built-In-Self-Test (BIST) [8], that is a mechanism that permits a device to feed itself with a fixed set of input patterns and compare the corresponding outputs or a hash of the outputs with a precomputed stored value. Usually, BIST provides poor control on the test patterns. It is suitable when it is necessary to check the system's reliability along its lifetime once it has been deployed, but not in previous stages.

In this paper, the authors introduce a new framework to design, test and measure the performance of an arbitrary digital system by using automated TPG an On-Chip open source verification device called Autotest Core. This core provides high throughput during hardware verification, performance measurements and immediate test/fail results. The framework provides functional verification at every stage of the development cycle, making it possible to reuse the set of test patterns. Additionally, It is a general purpose framework that can be applied to any implementation technology. The TPG can be chosen according to what is most in accordance with the CUT, and there are no restrictions in the size of the set of test patterns.

II. RELATED WORK

The main advantage of using FPGA devices for the design and implementation of digital systems is the possibility to carry out hardware verification during the design process itself, even if it is intended to be implemented in an Application Specific Integrated Circuit (ASIC). Because of this, FPGA plays an important role onto the hardware functional verification of critical parts or even the whole system. To perform this On-Chip functional verification, FPGAs vendors provide several tools. For example, Xilinx provides black-box-testing tools such as the Virtual Input/Output (VIO) core [9] as well as white-box-testing tools such as ChipScope [10]. The first one is a customizable IPCore that can both monitor and drive internal FPGA signals in real time, while the latter is an ILA. In [11] the methodology to carry out white-box-testing on hardware using Chipscope is described. It details several benefits, but also points out the limited number of samples that can be recorded in the internal block RAMs. Other authors have developed alternatives to the use of ILAs in order to solve their drawbacks. An example of this is the educational works presented in [12] and [13] used to implement telematic FPGA laboratories. In [12] the ILA is replaced by an integrated microprocessor that controls the operation of the CUT (Circuit Under Test), feeds it with input patterns, stores its output and internal signals and sends them outside of the FPGA. The work presented in [13] is more specific because it exclusively verifies the On-Board execution of a program loaded into an embedded microcontroller. A Finite State Machine (FSM) is used to control the operation. One of the main drawbacks of the ILAs is the reduced number of data that can be stored due to the size of the block RAMs of the FPGAs. Different alternatives have been proposed to solve this problem. In [14] the use of a microcontroller (picoblaze [15]) is proposed. The signals that are considered necessary to verify the operation of the CUT are captured and sent out in real-time. This system is suitable for the functional verification of IPs that operate at low speed and generate results in a permanent and constant way over a long period of time. In [16] a start-stop system is applied so that when the internal memories are full, the operation is stopped, the data is extracted, and the system is restarted again. This is done by controlling the system clock signal, stopping and reactivating it as the memories fill up and are downloaded. The work that we have been discussing so far focuses on the On-Chip functional verification process itself.

The previous contributions make it possible to carry out functional verification on a physical prototype. However, the verification methodology must be comprehensive, covering all phases of the design process. Also, the synergy between the verification process carried out at each stage is essential since it improves their quality and reduces the design time. This holistic approach is expedited by VIPs. These are IPCores specifically designed to check the functionality of specific protocols, interfaces or functionalities both at a discrete level and in combination with other IPCores. An example of this type of VIP is presented in [17]. In some holistic verification methodologies, the input patterns and obtained results are reused in later design stages. To this end, a high-level software programming language is used to get the first description of the system. This makes it possible to take advantage of the power of software programming language to carry out functional verification. In [18] a methodology of this type is proposed based on the high-level description tools of Xilinx (Vivado HLS) that automates functional verification

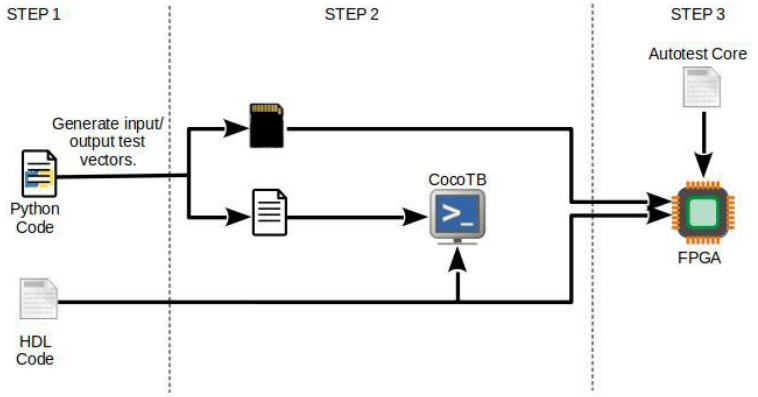


FIGURE 1. Design flow overview.

at Register Transfer Level (RTL), simulation and hardware levels.

In [19], the authors propose a co-validation design flow using the continuous integration methodology with tools such as Jenkins [20], GitHub [21] and Docker [22]. In this flow, a high-level software model of the system is used to verify each implementation. Another co-validation design flow is presented in [23]. In this paper, the authors introduce a python framework based on CocoTB called DUTILS to design SoCs. DUTILS is a python framework based on CocoTB which uses python to migrate high-level code to HDL. It uses a software model reference and from that creates by iterative steps the HDL model. Then, both are tested by the same test patterns, achieving an HDL module tested in a simulation-level. Reference [24] introduces another tool called LastLayer that makes it possible to create a C code simulation interface from an HDL description. In order to generate the interface, the HDL code must be adapted to a C library. According to the authors, this task is quite simple. Performing white-box-testing with the simulation interface is also straightforward, and the simulation interface is the same regardless of the implementation.

III. DESIGN FLOW OVERVIEW

From the analysis of previous work in the field of IPCore verification, three main questions arise that must be considered in any verification methodology:

1) **Test pattern generation.** As previously mentioned, most times an exhaustive test will not be possible, and a set of test patterns of manageable size must be carefully selected. Test pattern generation is a complex issue that is an active research field in its own, and is outside the scope of this paper.
2) **Functional verification**. The behaviour of the implementation fed by the selected test patterns is simulated at logical level. This has a high computational cost, and usually requires a remarkable share of the whole verification time.
3) **Hardware verification**. A physical instance of the implementation is fed by the selected test patterns.

Addressing these three main issues requires a comprehensive design, verification, and implementation procedure that begins with a high-level design and ends with a hardware verification of the implementation. The proposed approach is a fully integrated design and functional verification framework that consists of three main steps as shown in Fig. 1:

A. SOFTWARE LEVEL: DEVELOP A SOFTWARE MODEL OF THE SYSTEM

The design flow begins with the development of a high-level software model that describes the functionality of the system. The Python programming language is used for this task. The simplicity, clear structure and extensive availability of high-level programming libraries, makes it possible to write the software model in a fraction of the time required to design the hardware and verify it, taking advantage of the great facilities for the test that the Python language has. The software model also has the purpose to help the developer get a better understanding of the core functionality, making it possible to speed up the HDL design. The software model is then fed with a suitable set of test vectors, and the corresponding outputs are stored to be used as a reference in later steps. The proposed framework is flexible and does not impose a mechanism to get this set of test vectors. The developer can use test patterns or a TPG provided by a third party or write his own TPG. As previously mentioned, test pattern generation techniques are outside the scope of this paper.

B. RTL LEVEL: WRITE AND VERIFY THE HDL DESCRIPTION

First, the HDL description has to be developed. Then, the behaviour of the HDL description is simulated by using the CocoTB tool (RTL simulation). CocoTB is a co-verification Python tool that provides white-box testing. Specific python test code is written for critical parts of the HDL description, such as internal registers or counters at each clock cycle.

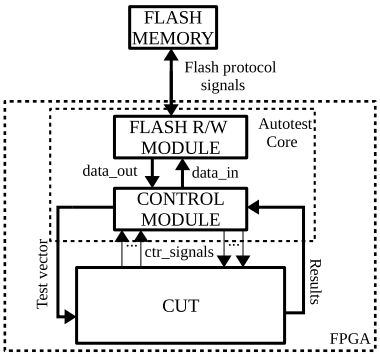


FIGURE 2. Proposed On-Chip functional verification system.

The overall functional verification of the whole HDL description is done by feeding CocoTB with a subset of the test patterns generated in step 1 and comparing its outputs with those generated by the software model.

C. HARDWARE LEVEL: VERIFICATION AND PERFORMANCE TESTING OF THE HARDWARE

In the last step, the system is implemented in a RH chip, alongside the so called Autotest Core, that will carry out black-box-testing. The Autotest Core is connected to the CUT inputs and outputs, as shown in Fig. 2. The test vectors and results previously generated by the software model are stored in a low-cost Flash memory device (microSD card). The Autotest Core then checks the functionality and performance of the CUT by feeding it with the test vectors, and comparing its outputs against the expected results. Every mismatch is reported and stored back in the microSD card alongside the performance measurements for further off-line analysis.

IV. AUTOTEST CORE

The proposed On-Chip functional verification system is shown in Fig. 2. In this Figure are three components; the flash memory which stores all the test records, the CUT which is the target to test on-board and finally the Autotest Core. The Autotest Core gets the test records from the flash memory by the Flash R/W Module using the SPI protocol. Once a test record is retrieved, it is processed by the Control Module to feed the CUT. To achieve it, the Control Module must be in charge of the control signals of the CUT, such as the reset signal, in order to get the CUT outputs which are compared with the expected ones. Lastly, the data collected from the CUT are send it from the Control Module to the Flash R/W module which writes it back into the flash memory.

Regards to the submodules of the Autotest Core, the Control Unit it is detailed below. However, the Flash R/W module is a modified version of the *minsdhost* module described in [25]; the differences are:

1) It is written in SystemVerilog instead of VHDL.
2) It supports class 10 microSD cards.
3) It implements the write command (CMD24 [26])
4) It implements the read multiple blocks command (CMD18 [26])

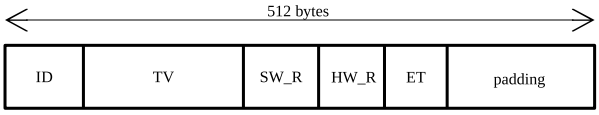


FIGURE 3. Record stored in the flash memory in the single block mode.

The Autotest Core description must be configured for each CUT. The Control Module has two operation modes suitable for different combinations of CUT and dataset. Both modes are different enough to provided an HDL implementation for each one. In order to agilize the design cycle template codes has been created and are available in the authors' github [27]. Therefore, when the designer chooses the dataset for the CUT then the appropriate template will be selected creating a synthesis of the full system.

A. SINGLE BLOCK MODE

This is the mode used to verify and measure the performance of any core whose input can be read in a single clock cycle, i.e. all the input bits are provided in parallel. The record corresponding to each test vector is stored within a single block of the flash memory using the format shown in Fig. 3. A block can only contain data for a single record, and test vectors to the same core are stored in a list of contiguous blocks.

Each record includes the following fields:

- `ID`: Every core to be tested is assigned a 32-bit identifier. The identifier of the core corresponding to the test vector is stored in this field. This makes it possible to store test vectors of different cores in the same flash memory.
- `TV`: This is the test vector to be applied to the CUT.
- `SW_R`: The expected output computed by the software model is stored in this field.
- `HW_R`: The output of the CUT is stored in this field.
- `ET`: The number of clock cycles employed by the CUT to generate the output is stored in this 64-bit field.
- `padding`: The size of this field is chosen so the size of the whole record is 512 bytes, i.e. the size of a block of the flash memory. Its content is not relevant since it is never read nor written.

The flow diagram for the single block mode is shown in Fig. 4. `initial_block`, `IPCUT_ID` and `max_timer` are parameters of the Autotest Core code template, while `current_b` and `errors` are local variables. `initial_block` is the number of the block containing the first test vector of the CUT, `IPCUT_ID` is the core identifier of the CUT and `max_timer` is an upper bound on the number of clock cycles required by the CUT to compute the output. Respecting the local variables, `current_b` encodes the number of the block containing the current input vector to be applied, while `errors` is the number of wrong outputs found. As previously mentioned, we used the seven segment displays available in our FPGA prototyping board to show `errors` in runtime.

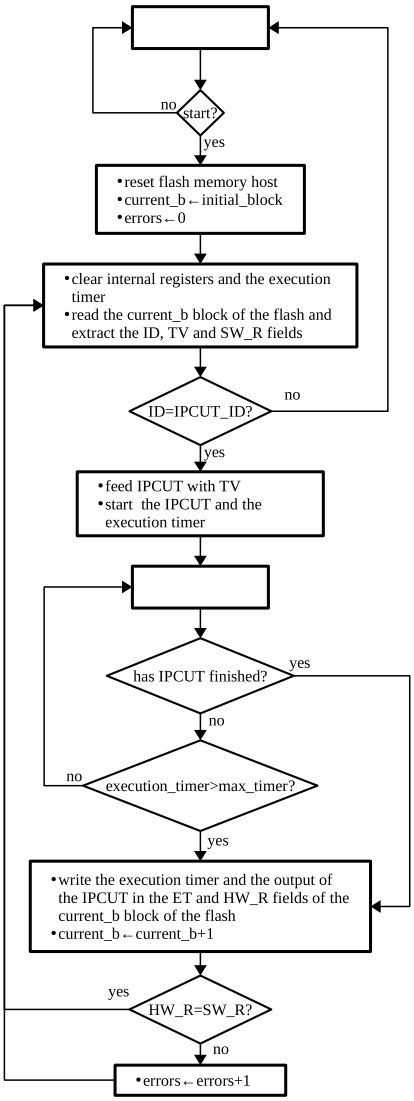


FIGURE 4. Flow diagram for the single block mode.

The behaviour of the Control Module in the single block mode is to wait until the start signal is activated. Then, it enters into a setup stage where the Flash R/W module is reset as well as all counters and registers. After that, the loop begins to check each test record. First, the test record fields are stored from the Flash memory into internal registers. Then, it makes a comparison between the ID field with a pre-stored signature, if both values are different then the Control Unit is finished, else the CUT is fed with the inputs from the test record and waits until an end signal is generated from the CUT. Due to the nature of the black-box testing, a timer has been introduced to be able to continue even if an internal state of the CUT is stalled. This timer also gives us the execution time that takes the CUT to process the data. Then, the output from the CUT is compared with the expected output, if both values are different then an internal counter for errors is incremented. Once the output and the execution time from the CUT are stored in the memory flash together with the test record, an internal register is updated to the upcoming

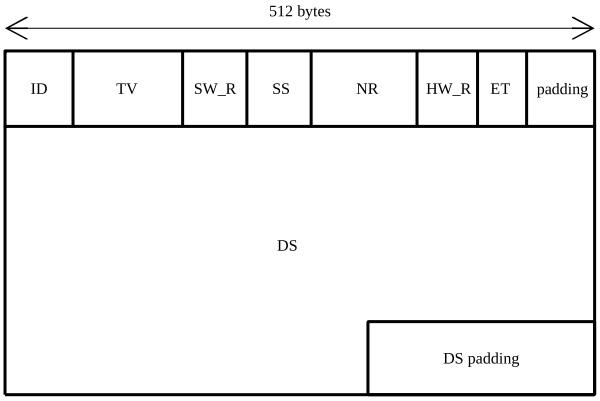


FIGURE 5. Record stored in the flash memory in the multiple block mode.

block in the memory flash where the next test record can be found and the loop begins once again.

B. MULTIPLE BLOCK MODE

This is the mode to be used to verify and measure the performance of cores used to process a data stream of variable size such as checksum generators and cryptographic hash function implementations. The Autotest Core code template for this mode has an additional parameter labeled `word_size` that is the size in bytes of the words used to feed the CUT. As shown in Fig. 5, for each test pattern a record with the following fields is stored in the flash memory:

- `ID`: Every core to be tested is assigned a 32-bit identifier. The identifier of the core corresponding to the test vector is stored in this field. This makes it possible to store test patterns of different cores in the same flash memory.
- `TV`: If the input of the CUT has fixed-size fields, they are stored here. For example, it can be the key used by message authentication code generators or digital signature implementations.
- `SW_R`: The expected output computed by the software model is stored in this field.
- `SS`: The size in bytes of the data stream is stored in this 64-bit field.
- `NR`: The number of the first flash block used to store the record of the following test pattern, if any, is stored in this 32-bit field.
- `HW_R`: The output of the CUT is stored in this field.
- `ET`: The number of clock cycles employed by the CUT to generate the output is stored in this 64-bit field.
- `padding`: The size of this field is chosen so the size of the whole record (except the data stream) is 512 bytes, i.e. the size of a block of the flash memory. Its content is not relevant since it is never read nor written.
- `DS`: The data stream is stored in this field.
- `DS padding`: The size of this field is chosen so the size of the data stream is a multiple of 512 bytes, i.e. the size of a block of the flash memory. Its content is not relevant since it is never read nor written.

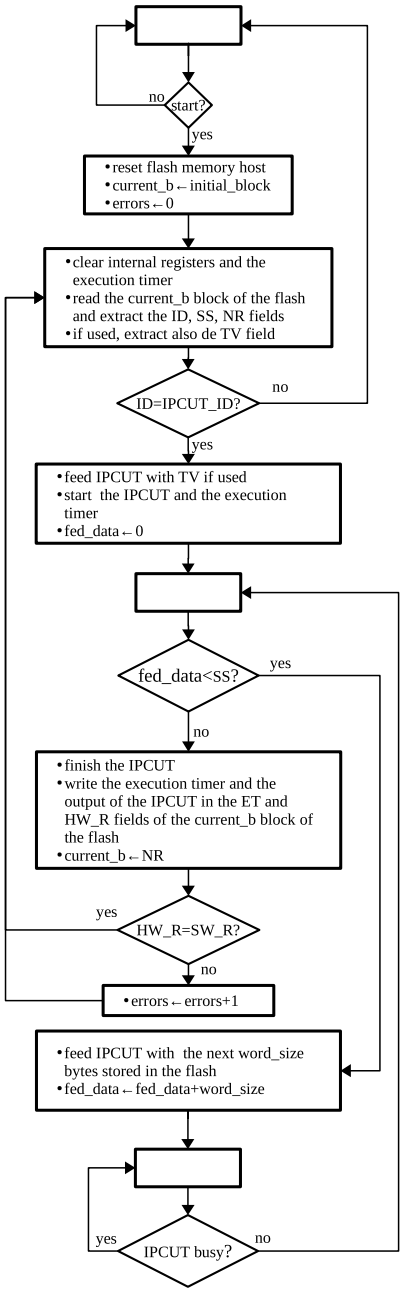


FIGURE 6. Flow diagram for the multiple block mode.

The flow diagram for this mode is quite similar to the previous one as shown in Fig. 6.

The behaviour of the Control Module in the multiple block mode is quite similar to the single block mode. Both make the same steps at the beginning until the way to feed the CUT. To feed the CUT in this mode, a chunk of data is read and then applied to the CUT which generates a new internal state. This process continues until all bytes stored have been applied. Then, the Control Module activates an end feed signal to the CUT which processes the data and when it finishes, it activates an end signal. It is important to clarify that a timer has been placed in the Control Unit as well as in the single mode. However, to get more accuracy in the result of execution time of the CUT, the timer is paused when the Flash R/W retrieves the data. Once the output is generated, it is stored together with the execution time of the CUT into the flash memory. If the generated output is different from the expected output, then the error counter is increased. After that, an internal register is updated with the value stored in the test record which indicates the upcoming block in the flash memory where the next test record is found. Lastly, the loop begins once again.

V. RESULTS

To validate the proposed approach and to estimate its performance, the following aspects have been evaluated: (i) Ease of integrating Autotest Core in the functional verification process of different systems; (ii) Resources used by the distinct operation modes; And (iii) Performance results (total execution time used to process a set of input test patterns in single and multiple block modes). Furthermore, these aspects have been compared with Xilinx VIO, an alternative method to perform black-box-testing on-board. In addition, one of the examples presented has been compared with a specific BIST solution detailed in [28]. However, this comparison can be only performed with the first example due to the specificity of the BIST solution.

A. EASE OF INTEGRATION

To demonstrate the ease of integrating Autotest Core in the functional verification process of any system, this approach has been used to design and verify a variety of IPCores available at the authors' GitHub repository. Most of these IPCores are hardware implementations of lightweight cryptographic algorithms:

- Block Ciphers: PRESENT and Twofish.
- Stream Cipher: Trivium.
- Hash Functions: Hirose PRESENT and SPONGENT.

These hardware implementations are perfectly adapted to the IoT perspective, where most devices are highly resource constrained and hardware implementations of traditional cryptographic algorithms cannot be afforded in terms of resources and power consumption. Therefore, once verified, these IPCores will be used to provide adequate security to different IoT devices that are being developed.

In order to organize them, a folder structure was used for each IPCore. The folders are:

- python_code: This folder contains the Python implementation of the IPCore functionality. In addition, it contains the TPG (Python script) which generates the test patterns.
- hdl_code: This folder contains the SystemVerilog implementation of the IPCore.
- cocotb_files: This folder contains a Python file which acts as a testbench file. This file is used for testing the SystemVerilog implementation with the test patterns generated by the TPG. A Makefile needed by CocoTB is also included in this folder.

- Hardware_verification_files: This folder contains two subfolders:
  - -- fusesoc_files: This subfolder contains a core file alongside the SystemVerilog top file. The core file collects all the files that FuseSoc [29] requires to generate the specific project files, depending on the selected Electronic Design Automation (EDA) tool: Vivado (Xilinx), Quartus (Intel), etc. As a result, this tool builds the bitstream file for the chosen hardware platform, which includes the IPCore as CUT and the Autotest Core as the functional verification core.
  - -- microSD_script_files: This subfolder contains all the necessary scripts to analyze the microSD card, once the Autotest Core has finished.

Moreover, this repository includes a user guide, which details the process of Autotest Core integration. Thus, it can allow designers to use this core to verify their own system.

The fact of providing all this information is twofold. Firstly, to demonstrate that Autotest Core is a generic approach that enables the functional verification of a wide range of systems quickly and easily, regardless of the technology used. And secondly, to facilitate that all the results presented in this paper can be contrasted.

Specifically, the obtained results in the functional verification process of two IPCores is shown in this paper: PRESENT block cipher and SPONGENT hash function.

The PRESENT IPCore is a SystemVerilog implementation of the PRESENT symmetric block cipher [30] with a block size of 64 bits and a key size of 80 bits. First, the HDL code of this block cipher was verified comparing the software results with the CocoTB simulation results. Later, the Autotest Core was adapted to the PRESENT cipher; and since inputs can be provided in parallel, the single block mode was used to verify this IPCore. The test vectors (`TV` record) have the following format:

- key: the 80-bit encryption/decryption key
- text: a 64-bit plaintext or ciphertext
- mode: a 1-bit control signal that selects the PRESENT Core mode to cipher (0-value) or decrypt (1-value)

The SPONGENT IPCore is a SystemVerilog implementation of the SPONGENT hash function [31]. This implementation may be fed with an arbitrary amount of input data. SPONGENT has an N-bit output where N can be 88, 128, 160, 224 or 256 bits. Therefore, a multiple block mode was used in order to test this IPCore. Input data is provided in the `DS` record; the `TV` record is empty.

Related to the VIO verification, the VIO IPCore can be easily configured and integrated using the Vivado Design Suite; and it is used to drive data into your design and read data from your design through the JTAG port. To automize the process of sending and reading data, an external software application is usually used. Unlike the Autotest Core, VIO is technology dependent and can only be implemented on Xilinx FPGAs.

```
import numpy as np
for k in range(0,N):
  # random TV fields
  key=np.random.randint(0,2**63-1,1,dtype=np.int64)
  text=np.random.randint(0,2**63-1,1,dtype=np.int64)
```

Listing 1. TPG simplified for PRESENT.

```
import random
for i in range(0,N):
  state = spongent_sw.initial_state()
  for k in range (0,stream_size):
    data_stream = random.randint(0,255))
    state=spongent_sw.feed_data(data_stream,state)
    microSD.write(data_stream)
```

Listing 2. TPG simplified for SPONGENT.

B. TPG SELECTION

As mentioned earlier, the TPG selection can be chosen by the developer. In our particular case, the goal is to test the proposed framework. Therefore, the TPG selection falls into the background. A TPG which generates a number of random test records has been used for simplicity. However, theoretically this particular TPG method has a high fault coverage assuming the single stuck-at fault model when applied to crypto-cores [32].

The following pieces of code are simplified versions of the python script used as TPG in these results, the full code can be accessed in the authors' github at the paths:

- block_ciphers\present_cipher\python_code\gen_testbench.py
- hash_functions\spongent_iter\python_code\gen_testbench.py

Listing 1 is the code for the PRESENT which has three TV fields. However, the mode field can be only 0, 1, therefore in each iteration of the TPG we create two records, one for each value of the mode field. Listing 2 is the code for the SPONGENT which only needs to generate the DS values. Each time data from the Stream is generated, it is used to feed the software model which updates the hash state. Once it is finished the random data is stored in the microSD.

C. RESOURCES

Regards to the hardware implementation on FPGA, both IPCores were implemented in a Nexys4DDR development board from Digilent [33], which included a Xilinx Artix7 XC7A100T-1CSG324C [34] FPGA chip. Besides this, in order to compare the amount of hardware resources used for our approach with those obtained in the PRESENT BIST work presented in [28], the PRESENT IPCore was also implemented in a Genesys2 development board which included a Xilinx Kintex7 XC7K325T FFG900-2 [34]. Similar results were obtained for both FPGAs. In this way, we assumed the same values is the Present BIST work for both boards in order to compare the results.

In table 1 the amount of resources used for three different alternatives in order to verify a PRESENT block cipher core (Autotest, VIO and the PRESENT BIST work) are shown.

TABLE 1. FPGA resources on Xilinx Artix7 XC7A100T-1CSG324C with the total percentage of the FPGA resources used by different functional verification cores in order to verify a PRESENT block cipher core.

Verification Core	Slices		Flip Flops		LUT's		BRAM's	
	No.	%	No.	%	No.	%	No.	%
Autotest Core (Single block mode)	334	2.11	764	0.60	1023	1.61	0	0
Xilinx VIO core	443	2.79	1655	1.31	842	1.33	0	0
BIST architecture proposed in [28]	33*	0.21	14*	0.01	163*	0.26	0*	0

* : indicates estimated values.

TABLE 2. FPGA resources on Xilinx Kintex7 XC7K325T FFG900-2 with the total percentage of the FPGA resources used by different functional verification cores in order to verify a PRESENT block cipher core.

Verification Core	Slices		Flip Flops		LUT's		BRAM's	
	No.	%	No.	%	No.	%	No.	%
Autotest Core (Single block mode)	332	0.65	764	0.19	1021	0.50	0	0
Xilinx VIO core	438	0.86	1655	0.41	842	0.41	0	0
BIST architecture proposed in [28]	33	0.06	14	0.003	163	0.08	0	0

TABLE 3. FPGA resources on Xilinx Artix7 XC7A100T-1CSG324C with the total percentage of the FPGA resources used by different functional verification cores in order to verify a SPONGENT88 hash function core.

Verification Core	Slices		Flip Flops		LUT's		BRAM's	
	No.	%	No.	%	No.	%	No.	%
Autotest Core (Multiple block mode)	418	2.64	813	0.64	1362	2.15	0	0
Xilinx VIO core	400	2.52	1469	1.16	833	1.31	0	0

TABLE 4. FPGA resources on Xilinx Kintex7 XC7K325T FFG900-2 with the total percentage of the FPGA resources used by different functional verification cores in order to verify a SPONGENT88 hash function core.

Verification Core	Slices		Flip Flops		LUT's		BRAM's	
	No.	%	No.	%	No.	%	No.	%
Autotest Core (Multiple block mode)	406	0.80	792	0.19	1339	0.66	0	0
Xilinx VIO core	411	0.81	1469	0.36	834	0.41	0	0

The results indicated that the BIST implementation is the option that uses the least hardware resources. This is because the PRESENT cipher core is reused as a TPG, therefore, being a specific solution. However, it can be seen that Autotest and VIO options are similar in this category. These approaches are generic solutions and the resource constraints are not critical since both cores use less than a 3% of the total slices. Similarly in table 2 the BIST implementation is the most efficient solution at the cost of being the more specific. Respect the other solutions although they are more expensive in resources, both are below the 1% of slices for the Xilinx Kintex7 XC7K325T FFG900-2 FPGA.

In tables 3 and 4 the amount of resources used by two different functional verification cores (Autotest and VIO) in order to verify a SPONGENT hash function core with an 88-bit output are shown. It can be seen that both cores have a low impact on the total FPGA resources, occupying less than 3% of the total slices for the Xilinx Artix7 XC7A100T-1CSG324C and less than 1% of the slices for the Xilinx Kintex7 XC7K325T FFG900-2. Therefore, in terms of hardware resources, they are generic solutions that allow designers to verify these types of systems.

Finally, as described in the previous sections, the Autotest Core read the input tests and expected results from the Flash memory device, so the whole functional verification process could be carried out within the FPGA. Regards to VIO core, an external script in TCL format was used to send the test vectors to the CUT through a JTAG port. This script also took care of receiving the output from CUT, comparing it with the expected result and storing all verification results in a file. Since the input tests were read one by one, the use of BRAMs was not necessary as observed in Tables 1,2 and 3. Therefore, hardware resources are highly optimized for both functional verification cores (Autotest and VIO) and operation modes (single and multiple block modes).

D. PERFORMANCE RESULTS

In this section, performance results will be presented. In the context of this paper, performance will be measured as the time dedicated to process a set of test patterns.

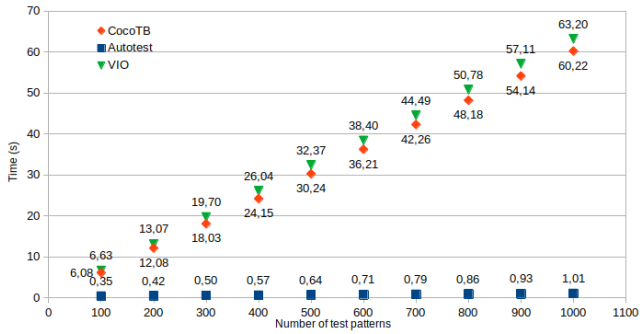


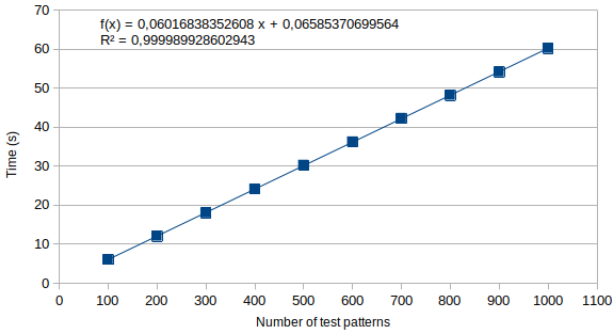
FIGURE 7. **PRESENT IPCore. Execution times, partial up to 1000 test patterns.**

As described in Section III, first, the behaviour of the HDL description was simulated using the CocoTB software tool. Next, was the On-Chip functional verification procedure. Therefore, three different alternatives were analyzed: CocoTB software tool, Autotest Core and Xilinx VIO core. All the software executions were executed in a computer with an AMD Ryzen 7 2700 Eight-Core Processor and 32 GB of RAM memory.

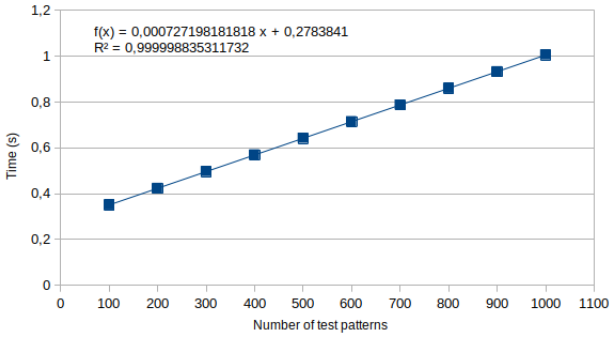
First, we can see the functional verification results of PRESENT block cipher are shown. Fig. 7 depicts execution times, partial up to 1000 test patterns (500 encrypts and 500 decrypts). This figure shows a similar performance for CocoTB software tool and VIO core. This is because the script used to communicate to the computer with the VIO core was running in software, so most of the time was spent reading test vectors from a file, exchanging data through the JTAG port and storing all functional verification results in another file. In this way, the potential of doing On-Chip functional verification using the VIO core was reduced in terms of communication with the computer. As seen in Fig. 7, this fact does not occur when Autotest Core is used, since all operations (reading of the test patterns from the SD card, pattern processing, etc.) are fully performed in hardware. This statement implies that Autotest Core can process a large number of test patterns per unit of time and perform a more thorough functional verification of systems, compared to the other alternatives.

From the data shown in Fig. 7, trend lines and coefficients of determination can be calculated for each alternative: CocoTB simulation tool, Autotest Core, and Xilinx VIO core (Fig. 8(a), 8(b), and 8(c), respectively). In these figures, the line slopes represent the average time per pattern, and the line constants represent the initialization times.

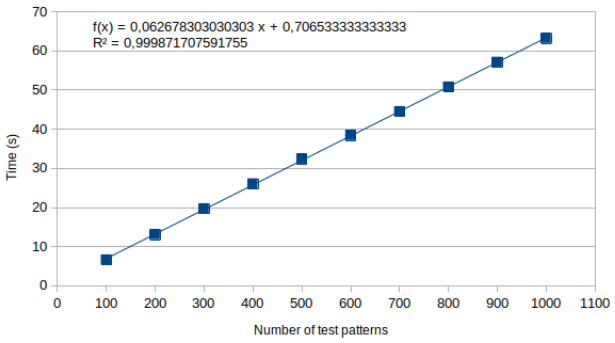
To get a better understanding of the results, Fig. 9 has been added. In this figure, it has been compared the average execution time (ns) taken to test one record. In addition, it has been included the time that the PRESENT cipher Core takes to encode or decode data in order to be able to compare both the time dedicated to the CUT, and the complete system. The values of the Y-axis are in logarithm base 10 to get a quick sight of the differences in orders of magnitude. The results show that the PRESENT BIST implementation presented



(a)



(b)



(c)

FIGURE 8. **PRESENT IPCore. Trend lines and coefficients of determination for each alternative: (a) CocoTB, (b) Autotest Core, and (c) Xilinx VIO core.**

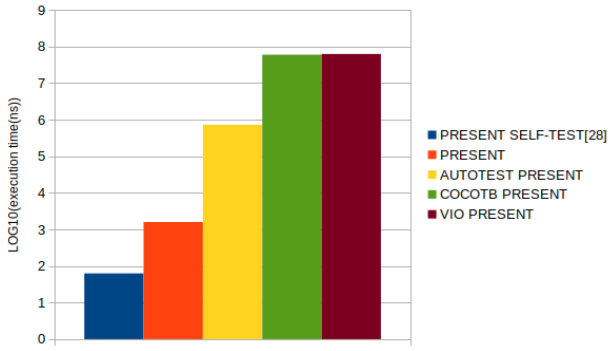


FIGURE 9. **PRESENT solution average execution time for one record in log10 scale.**

in [28] has the better performance with an order of $10^{-8}s$ by far of the second best that is Autotest Core with 10^{-4}

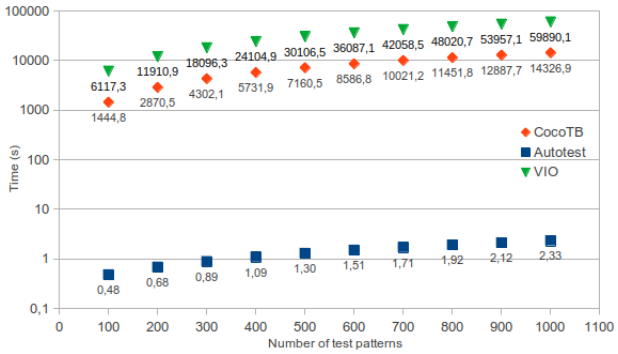


FIGURE 10. SPONGENT88 IPCore. Execution times, partial up to 1000 test patterns. The input data has a size of 1024 bytes.

TABLE 5. PRESENT IPCore. Number of test patterns processed in one hour.

	Number of test patterns
CocoTB	59831
Autotest Core	4950125
Xilinx VIO core	57425

and the 10^{-2} of VIO. However, the clock signal used in the BIST solution is five times greater than the one used in the other cases, 500Mhz in opposition of 100Mhz. The conclusion with these results is that specific solutions are better in performance, but our framework is a better generic solution to test a broad set of Cores.

To demonstrate the power of Autotest Core, Table 5 shows the number of test patterns processed in one hour by each alternative. As seen in Table 5, the number of test patterns processed by Autotest Core exceeds the results obtained by CocoTB and VIO core by two orders of magnitude; speeding up the functional verification process. In addition, the type of functional verification that Autotest Core performs has two additional benefits. Firstly, that performing such a thorough hardware functional verification can allow designers to detect errors occurring after several hours of testing under certain established conditions. And secondly, it can check the maximum operating frequency of the system and measure the processing time of the system at that frequency. In this way, it has been verified this IPCore works properly at a maximum frequency of 400 MHz, getting a processing time of 390 ns.

Secondly, the functional verification results of SPONGENT hash function will be shown. Fig. 10 depicts execution times, partial up to 1000 test patterns. The input data has a size of 1024 bytes. Trend lines and coefficients of determination for each alternative are shown in Fig. 11(a), 11(b), and 11(c). Regarding the VIO core, Fig. 10 shows that when the CUT must be fed with an arbitrary amount of input data (only 1024 bytes in this scenario), the performance of this core is very low; even CocoTB presents a better performance. This is because the VIO core is designed to replace or augment board-level I/O components such as status indicators and low-bandwidth controls (LEDs, buttons or DIP switches); but

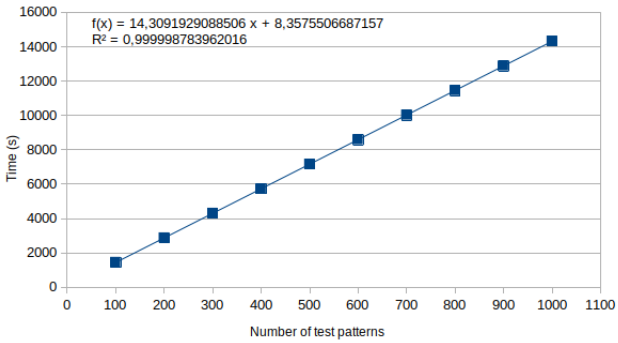


FIGURE 11. SPONGENT88 IPCore. Trend lines and coefficients of determination for each alternative: (a) CocoTB, (b) Autotest Core, and (c) Xilinx VIO core.

it is not optimized to perform this type of hardware functional verification. Fig. 10 also shows Autotest Core can verify this type of IPcores in a reduced time. Besides, according to the Fig. 11(a), 11(b), and 11(c), the number of test patterns processed by Autotest Core exceeds the results obtained by CocoTB and VIO core by four orders of magnitude. This means that in scenarios where a larger number of test patterns or larger input data have to be processed, Autotest Core still offer a good performance.

In the same line with the previous example, Fig. 12 has been added. This figure shows the execution time (ns) to process one test record in the different solutions, alongside the time taken by the Core to generate the hash value.

TABLE 6. Power consumption.

Complete System	Dynamic Power(mW)	Static Power (mW)	Total Power (mW)
PRESENT Autotest Core	39	97	137
PRESENT VIO	15	84	99
PRESENT BIST [28]	63	83	146
SPONGENT88 Autotest Core	46	97	143
SPONGENT88 VIO	18	84	102

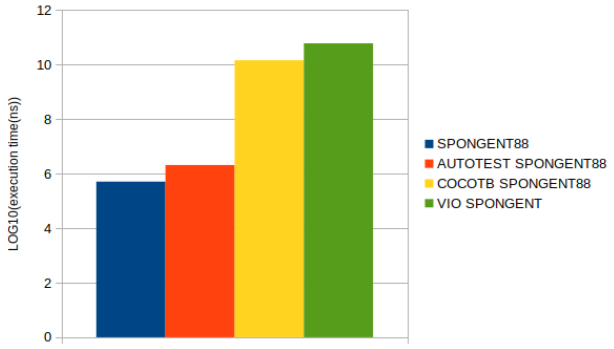


FIGURE 12. SPONGENT88 solution average execution time for one record in log10 scale.

The logarithm base 10 in the results is to compare the orders of magnitude. It is clear that Autotest Core has an advantage over VIO by four orders of magnitude, so the Autotest Core has an order of $10^{-3}s$ and VIO has an order of $10^{1}s$.

Finally, the performance of the SPONGENT IPCore has been determined by Autotest Core. In this way, this IPCore works properly at a maximum of 400 MHz, getting a processing time of 126,75 μs.

E. POWER CONSUMPTION

In this section, the power consumption will be presented. These results are an estimation performed by Vivado Power Analyzer tools.

With the values presented in table 6 we can establish that all the proposed solutions have the same order of consumption ($10^{-2}W$), being that one assumable in the IoT field.

VI. CONCLUSION

The verification framework proposed in this paper is adequate to design and test IPCores in a fast and reliable way by combining test pattern generation using a software model, HDL simulation, on-chip testing and performance measurements.

TPG generation, including expected results, is done at a high level in software and used throughout the whole verification process by the HDL model and the on-chip verification core (Autotest Core) greatly reducing verification time.

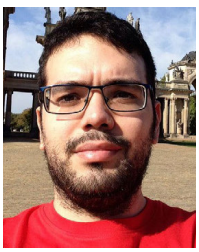
PRESENT and SPONGENT cryptographic cores and random pattern generation have been used to test the framework functionality successfully. It has been shown that the Autotest Core hardware footprint is below 3% of the total slices available in standard FPGA chips while the ability to read patterns and store results in low-cost standard Flash memory (SD card) provides a very flexible and cost-effective verification system.

Comparison to commercial alternatives like Xilinx's VIO shows that the Autotest Core performance is at least two orders of magnitude faster in addition to be able to do off-line testing without the support of an external system (computer), making it a great alternative to standard tools.

REFERENCES

[1] *IoT Infrastructure Technology & Connectivity Explained—Business Insider*. Accessed: Dec. 2020. [Online]. Available: https://www.businessinsider.com/iot-infrastructure-technology?IR=T
[2] H. D. Foster, "Quantifying FPGA verification effectiveness," *Verification Horizons*, vol. 16, no. 3, 2020.
[3] *MyHDL*. Accessed: Oct. 2020. [Online]. Available: http://www.myhdl.org/
[4] *PyRTL by UCSBarchlab*. Accessed: Oct. 2020. [Online]. Available: https://ucsbarchlab.github.io/PyRTL/
[5] *Introduction—COCOTB 1.1 Documentation*. Accessed: Aug. 2020. [Online]. Available: https://cocotb.readthedocs.io/en/latest/introduction.html
[6] Xilinx. (2018). *Vivado High-Level Synthesis*. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html
[7] S. Nidhra, "Black box and white box testing techniques—A literature review," *Int. J. Embedded Syst. Appl.*, vol. 2, no. 2, pp. 29–50, Jun. 2012.
[8] C. E. Stroud, *A Designer's Guide to Built-in Self-Test* (Frontiers in Electronic Testing), 1st ed. New York, NY, USA: Springer, 2002.
[9] Xilinx. *Virtual Input/Output v3.0 LogiCORE IP Product Guide Vivado Design Suite*. [Online]. Available: http://www.xilinx.com
[10] *ChipScope Integrated Logic Analyzer (ILA)*. Accessed: Jul. 2020. [Online]. Available: https://www.xilinx.com/products/intellectual-property/chipscope_ila.html
[11] K. Arshak, E. Jafer, and C. Ibala, "Testing FPGA based digital system using Xilinx ChipScope logic analyzer," in *Proc. 29th Int. Spring Seminar Electron. Technol.*, May 2006, pp. 355–360.
[12] D. Garijo and R. Senhadji, "CcLAB: A tool for remote verification of FPGA-based circuits," *IEEE Latin Amer. Trans.*, vol. 14, no. 3, pp. 1115–1121, Mar. 2016.
[13] K. Saksida and A. Trost, "Remote laboratory for testing processor cores in FPGA device," in *Proc. 37th Int. Conv. Inf. Commun. Technol., Electron. Microelectron. (MIPRO)*, May 2014, pp. 172–177.
[14] J. Viejo, J. I. Villar, J. Juan, A. Millan, E. Ostua, and J. Quiros, "Long-term on-chip verification of systems with logical events scattered in time," *Microprocessors Microsyst.*, vol. 36, no. 5, pp. 402–408, Jul. 2012, doi: 10.1016/j.micpro.2012.02.005.
[15] *PicoBlaze 8-bit Microcontroller*. Accessed: Jan. 2021. [Online]. Available: https://www.xilinx.com/products/intellectual-property/picoblaze.html#overview
[16] H. U. H. Khan and D. Göhringer, "FPGA debugging by a device start and stop approach," in *Proc. Int. Conf. ReConFigurable Comput. FPGAs (ReConFig)*, Nov. 2016, pp. 1–6.
[17] S. Harutyunyan, T. Kaplanyan, A. Kirakosyan, and H. Khachatryan, "Configurable verification IP for UART," in *Proc. IEEE 40th Int. Conf. Electron. Nanotechnol. (ELNANO)*, Apr. 2020, pp. 234–237.
[18] J. Caba, F. Rincón, J. Barba, J. A. De La Torre, J. Dondo, and J. C. López, "Towards test-driven development for FPGA-based modules across abstraction levels," *IEEE Access*, vol. 9, pp. 31581–31594, 2021.
[19] L. Beaulieu, O. Weppe, B. Le Ludec, and F. Lebeau, "Co-verification design flow for HDL languages: A complete development methodology," in *Proc. 24th IEEE Int. Conf. Electron., Circuits Syst. (ICECS)*, Dec. 2017, pp. 530–533.

[20] *Jenkins*. Accessed: Nov. 2020. [Online]. Available: https://www.jenkins.io/
[21] *GitHub: Where the World Builds Software GitHub*. Accessed: Nov. 2020. [Online]. Available: https://github.com/
[22] *Empowering App Development for Developers | Docker*. Accessed: Nov. 2020. [Online]. Available: https://www.docker.com/
[23] M. Trapaglia, R. Cayssials, L. De Pasquale, and E. Ferro, "Flexible software to hardware migration methodology for FPGA design and verification," in *Proc. 10th Southern Conf. Program. Log. (SPL)*, Apr. 2019, pp. 39–44.
[24] L. Vega, J. Roesch, J. Mcmahan, and L. Ceze, "LastLayer: Toward hardware and software continuous integration," *IEEE Micro*, vol. 40, no. 4, pp. 103–111, Jul. 2020.
[25] P. Ruiz-de-Clavijo, E. Ostúa, M.-J. Bellido, J. Juan, J. Viejo, and D. Guerrero, "Minimalistic SDHC-SPI hardware reader module for boot loader applications," *Microelectron. J.*, vol. 67, pp. 32–37, Sep. 2017, doi: 10.1016/j.mejo.2017.07.007.
[26] SD Group. (2005). *SD Specifications Part 1: Physical Layer Specification. V1.10*. [Online]. Available: https://www.sdcard.org/downloads/pls/simplified_specs/part1_410.pdf
[27] *GitHub—Germancq/IPCores*. Accessed: Feb. 2021. [Online]. Available: https://github.com/germancq/IPCores
[28] Z. Haider, K. Javeed, M. Song, and X. Wang, "A low-cost self-test architecture integrated with PRESENT cipher core," *IEEE Access*, vol. 7, pp. 46045–46058, 2019.
[29] *GitHub—Olofk/Fusesoc: Package Manager and Build Abstraction Tool for FPGA/ASIC Development*. Accessed: Aug. 2020. [Online]. Available: https://github.com/olofk/fusesoc
[30] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. Robshaw, Y. Seurin, and C. Vikkelsoe, "PRESENT: An ultra-lightweight block cipher," in *Proc. Int. Workshop Cryptograph. Hardw. Embedded Syst.* Berlin, Germany: Springer, 2007, pp. 450–466.
[31] A. Bogdanov, M. Knežević, G. Leander, D. Toz, K. Varıcı, and I. Verbauwhede, "spongent: A lightweight hash function," in *Cryptographic Hardware and Embedded Systems—CHES 2011*, B. Preneel and T. Takagi, Eds. Berlin, Germany: Springer, 2011, pp. 312–325.
[32] G. Di Natale, M. Doulcier, M.-L. Flottes, and B. Rouzeyre, "Self-test techniques for crypto-devices," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 18, no. 2, pp. 329–333, Feb. 2010.
[33] *Nexys 4 DDR [Reference.Digilentinc]*. Accessed: Aug. 2020. [Online]. Available: https://reference.digilentinc.com/reference/programmable-logic/nexys-4-ddr/start
[34] Xilinx. (2010). *7 Series FPGAs Data Sheet: Overview (DS180)*. [Online]. Available: www.xilinx.com



GERMAN CANO-QUIVEU received the B.Sc. degree in computing engineering from the University of Seville, Spain, in 2015, where he is currently pursuing the Ph.D. degree with the Department of Electronics Technology. His research interests include bootloaders, the IoT, and SoC.

PAULINO RUIZ-DE-CLAVIJO-VAZQUEZ received the B.Sc. and Ph.D. degrees in computer science from the University of Seville, Spain, in 1999 and 2007, respectively. He was with the Institute of Microelectronics, Seville, part of the National Centre of Microelectronics, Spain, from 1998 to 2004. He has been with the Department of Electronics Technology, University of Seville, since 1999, as an Assistant Professor. His research interests include system-on-chip designs, digital signal processing, and embedded microprocessors architecture, areas to which he has contributed in international conferences and workshops.

MANUEL J. BELLIDO-DIAZ received the B.Sc. and Ph.D. degrees in physics from the University of Seville, Spain, in 1987 and 1994, respectively. He has been with the Department of Electronics Technology, University of Seville, since 1990, where he holds a post as a Professor.

DAVID GUERRERO-MARTOS received the B.Sc. and Ph.D. degrees in computer engineering from the University of Seville, Spain, in 2000 and 2012, respectively. Since 2002, he has been working as a Lecturer with the Department of Electronics Technology, University of Seville. He has published several papers in journals and conferences. His research interests include digital circuit synchronization, hardware implementation of numerical methods, and computer architecture.

JULIAN VIEJO-CORTES received the M.Sc. and Ph.D. degrees in computing engineering from the University of Seville, Spain, in 2004 and 2011, respectively. He currently works as an Assistant Professor with the Department of Electronics Technology, University of Seville, and has contributed several research papers to international journals and conferences in the area of digital signal processing and system-on-chip design.

JORGE JUAN-CHICO received the B.Sc. and Ph.D. degrees in physics from the University of Seville, Spain, in 1994 and 2000, respectively. He is currently an Associate Professor with the Department of Electronics Technology, University of Seville, where he is leading the Digital Research and Development Group. He has carried out research in the areas of metastability, delay modeling, timing and power simulation, and digital embedded systems.