International Journal of Electrical and Computer Engineering (IJECE)
Vol. 7, No. 4, August 2017, pp. 2192~2205
ISSN: 2088-8708, DOI: 10.11591/ijece.v7i4.pp2192-2205

Functional Verification of Large-integers Circuits using a Cosimulation-based Approach

Nejmeddine Alimi[1], Younes Lahbib[2], Mohsen Machhout[4], Rached Tourki[5]
[1]Faculty of Sciences of Tunis, University of Tunis El Manar, 2092 El Manar Tunis, Tunisia
[2]National Engineering School of Carthage, University of Carthage, 2035 Charguia II Tunis, Tunisia
[1,2,4,5]Electronics and Micro-Electronics Laboratory (E. µ. E. L), Faculty of Sciences of Monastir, University of Monastir, 5000 Monastir, Tunisia

Article Info

Article history:

Received Nov 28, 2016
Revised Apr 26, 2017
Accepted May 10, 2017

Keyword:

Assertion-based verification
Co-simulation
Cryptography
Hardware description language
High-level verification
Large-integer
Matlab/Simulink

ABSTRACT

Cryptography and computational algebra designs are complex systems based on modular arithmetic and build on multi-level modules where bit-width is generally larger than 64-bit. Because of their particularity, such designs pose a real challenge for verification, in part because large-integer's functions are not supported in actual hardware description languages (HDLs), therefore limiting the HDL testbench utility. In another hand, high-level verification approach proved its efficiency in the last decade over HDL testbench technique by raising the latter at a higher abstraction level. In this work, we propose a high-level platform to verify such designs, by leveraging the capabilities of a popular tool (Matlab/Simulink) to meet the requirements of a cycle accurate verification without bit-size restrictions and in multi-level inside the design architecture. The proposed high-level platform is augmented by an assertion-based verification to complete the verification coverage. The platform experimental results of the testcase provided good evidence of its performance and re-usability.

Corresponding Author:

Nejmeddine Alimi,
Electronics and Micro-Electronics Laboratory (E. µ. E. L),
Faculty of Sciences of Monastir, University of Monastir,
Avenue de l'Environnement - 5000 Monastir, Tunisia.
Email: nejmeddine.alimi@fst.utm.tn

1. INTRODUCTION

Large-integer arithmetic is a set of operations like addition, multiplication, modular reduction, *etc* that involves integers larger than the native word size of the general purpose processors, typically, 64-bit. Depending on the target application requirements, integer operands may have 163-bit, 192 bit, 512-bit, 1024-bit of length, and more. One place where large integers are used is cryptography, especially in the public-key family like RSA [1] and Elliptic Curve Cryptography [2], [3]. Large integers are also used in complex research, high performance computing (HPC) and computational algebra. Large integers operations know a continuous development in mathematical algorithms [4-6]. Hardware-based implementations of such algorithms have proved to be more efficient than equivalent software's programs in terms of speed and resources usage. This is mainly due to exploring new design architectures [7-12]**.** Such designs are generally written in hardware description languages (HDLs).

To verify that a design works as intended, two technologies are commonly used; Simulation-based verification and formal verification. The simulation-based verification is the technique generally used for complex designs. Formal verification, which consists in mathematically checking the functional correctness of the design, is generally used to verify small designs and corner cases. However in the last decade, formal verification tools have seen their capacity to verify more complex designs improved to some extent, in part, because of its coupling with simulation (dynamic formal verification) and the standardization of some assertion languages. The goal was to make a complementary technology to the simulated-based one so that the overall verification methodology could be enhanced.

Regarding simulation-based verification, running testbench in an HDL simulator is the common approach to verify hardware designs and HDL packages (e.g. VHDL, Verilog, *etc*.) provide a range of functions intended to help writing testbenchs. But, to the best of our knowledge, among those packages as well as functional verification frameworks (e.g. Specman, Jove, *etc*.), there is no dedicated Application Programming Interface (API) supporting large-integers operations. A workaround consists on verifying against equivalent program written at a high-level language. Such programs are run on softwares called Computer Algebra Systems (CAS) that supports a non-limited precision like MAPLE, MATHEMATICA and the GMP library. In addition to CAS, there exist a number of domain-specific libraries like Crypto++ and MIRACL that supplement traditional high level programming languages with large-integer support to target specific domains like cryptography. Although using CAS and specific libraries to verify HDL designs may meet the functional verification purpose for very basic and unit-level designs, it remains insufficient for more complex designs. In fact, because the verification flow is disjoined (DUV and CAS are not ran simultaneously), the verifcation and interaction with the Design Under Verification (DUV) is limited. On the other hand, the large-integer data to be used as stimuli to DUV and CAS has to be constant and stored beforehand. Therefore, guided testbenchs techniques with dynamic updated stimuli cannot be applied.

Co-simulating DUV and its Reference Model requires an efficient communication between the high-level testbench and the HDL simulator. In this context, some works have been done. For example, the cosimulation of VHDL designs and a C-based testbench using the Foreign Language Interface (FLI) provided by ModelSim simulator was proposed in [13]. Similar projects based on FLI and/or PLI (for Verilog) and written in other high-level languages (e.g. Python) were proposed in [14] and [15]. However because such languages are architecture limited size, large-integer support in not supported natively. In the other hand, formal verification techniques for large-integer HDL were applied in simple cases in [16], [17]. Despite their proved performance, those frameworks remain insufficient to verify large-integer HDL designs of certain complexity in standalone. In another hand, some works on large-integers using Matlab/Simulink, the powerful pair of numerical computing and simulation softwares, have been conducted in the design field. As examples, in [18]–[20] authors speeded up hardware implementations of cryptographic designs by modelling the schemes in Simulink and generating synthesizable HDL using dedicated tools like HDL coder. Examples of working around the size restriction has been reported in [18],where authors divided the large operands into smaller size to take advantage of hardware DSP's multiplication capabilities in the target FPGA. In the same context, in [19], authors used specific multiplication algorithm with a property of splitting up operands into small size words. While in [20], authors bounded the operand sizes to ordinary bit-length to optimize the HDL code generation in order to achieve efficient throughput. In verification, Matlab was separately used to verify ECC (Elliptic Curve Cryptography) designs in [21] and [22] but no details on the evaluation process or the interfacing with the HDL design were given.

Three challenges are still to take for designs involving large-integers: how to support a hardware design testbench without size restriction? How to perform verification for complex designs where operations run at different levels, and how to set the verification structure to verify the full design? In this paper, which is a revised and extended version of the work presented in [23], we try to draw a path for a solution to those challenges by introducing a high-level simulation-based verification platform based on Matlab and Simulink. Besides generating stimuli and monitoring the verification flow, large integer's transactions and processing are supported within the proposed platform. The platform features a high level generation of testbench, a cross-level and a cycle-accurate verification. Furthermore, Matlab's support for large-integer, using its Variable Precision Integer Arithmetic (VPI) package, is exploited. To complete the verification of a given design, the control logic part of a DUV is verified formally using the same HDL simulator.

The rest of the paper is organised as follows, section 2 details the proposed platform where the verification structure, data transformation across stages and the process of settings and controlling the platform are explained. In section 3, a detailed testcase is given to illustrate the working of the platform followed by results and discussion. Finally, a conclusion with future works ends the paper.

2. THE PROPOSED PLATFORM

2.1. Overview

The design methodology of the platform follows the Simulation-based approach, where stimuli are generated, applied to the DUV and responses are compared to the expected ones. Typical verification framework based on high level design language includes a stimuli generator, a Reference model (also referred to as Golden Model) which is usually written at a higher level of abstraction, and a comparator. We abstract the functional description of the platform into three flows, i.e., control flow, data flow and verification flow, as shown in Figure 1. The Control flow controls the process of verification through the platform. It fixes the settings; i.e. the parameters of the blocks constituting the platform, delay times, sampling times, *etc*. Data flow represents the transformations that data undergoes, starting from the generation of large-integer operands, passing through the input adapter, the DUV, the output adapter and, finally, entering the comparison/checking blocks. The third flow, Verification flow, verifies the functional correctness of the DUV. We chose to build the verification flow around two complementary simulation-based verification approaches: testbench and assertion-based verification. We guide the testbench via a verification structure with considerations of a coverage plan. When testbench is launched, outputs of DUV and reference model are compared. Results are then transferred to a Scoreboard to be analyzed. We write assertions in Property Specification Language (PSL) [24], standard assertion language, inside the DUV and represent a precise description of the DUV's behavior. Note that we chose PSL for property description as it's in widespread use in industry and compatible with many hardware description languages. PSL assertions are checked by the HDL simulator during the simulation. The assertions verification results (pass/fail) are also sent to the scoreboard to be analyzed and new stimuli are generated in the next testbench according to the updated functional coverage.

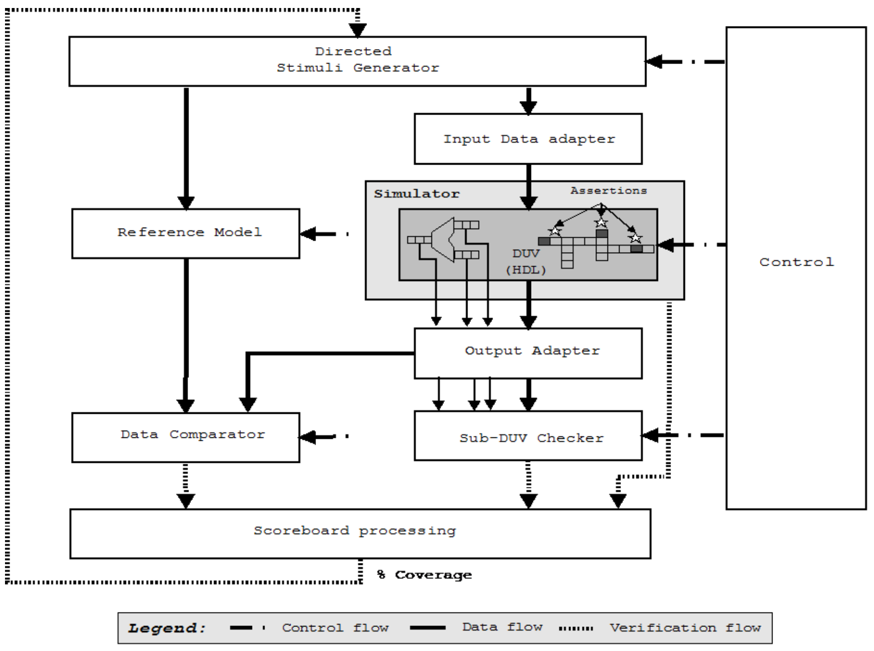

Figure 1. Functional Description of the platform.

2.2. The Functional Verification Process

The purpose of the "Functional" verification process is to verify that the DUV matches its specification. This process should verify that the implemented functions behave correctly. The verification technology used is the simulation-based verification, more precisely a cosimulation between Matlab/Simulink and ModelSim, and simulated assertions written in PSL.

Globally, we followed a coverage-driven random-based verification approach. The level of verification can be of unit/sub-unit or cores/blocks level and two simulation-based verification techniques are used jointly, depending on the partition of DUV being verified. In fact, a common practice in the integrated circuits design community is to divide designs into datapath and control logic (Figure 2). Because of their differences, appropriate verification schemes can be applied to each. Datapath units which involve large-integers processing can be verified using the Matlab/Simulink testbench where large-integers are supported as will be detailed in the next section. Datapath usually consists of uniform arrays of cells, such as bits in a register file, slices in an adder and so on. The remaining logic is regarded as control logic.

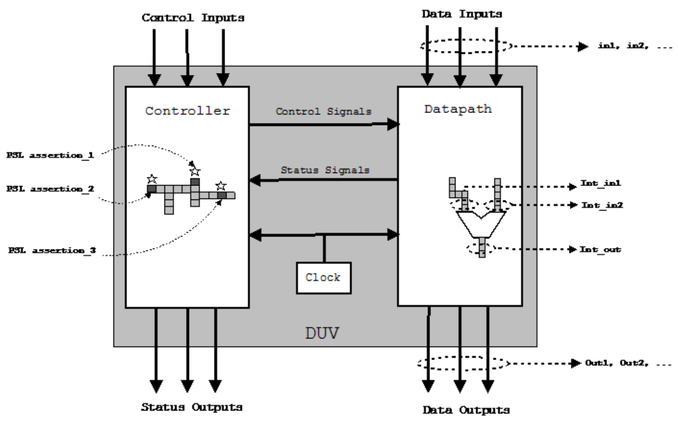

Figure 2. Datapath and control logic partition for verification

An advantage of using HDL cosimulation with Matlab/Simulink testbench is the possibility of cross-level datapath verification, as will be more detailed later. This means that data's output of different hierarchical level can be probed and compared in run-time against Matlab models. On the other hand, control logic can be accurately specified by properties and assertions, and thus is verifiable using PSL. The DUV's control logic is specified by a set of proprieties written in PSL assertions.

The verification structure is the set of Matlab Function Blocks within the platform in charge of the verification plan. Figure 3 represents the architecture of the "Verification structure". The latter is divided in two block sets, connected to form a loop with the rest of the platform. The first set is composed of stimuli generation blocks while the second is composed of analysis blocks ("Comparator", a "Checker" and a "Scoreboard"). Within the first, the Data output of the DUV (Z_DUV), is verified against the Reference Model output (Z_Ref), the result of the comparison is transferred to the Scoreboard. According to the feedback, the first set generates new stimuli corresponding to the next coverage step and/or to the next property to be verified. The DUV is here a modular arithmetic operation. The objective is to ensure that DUV matches its specification.

The functional verification approach is a white-box approach (i.e. the HDL design is known to the verifier). Because of sampling time difference, control signals and data were assigned to separate blocks (Figure 3). Both blocks are driven by a block called "Testbench Scenario Update". The role of the latter is to update control signals and data according to verification coverage.

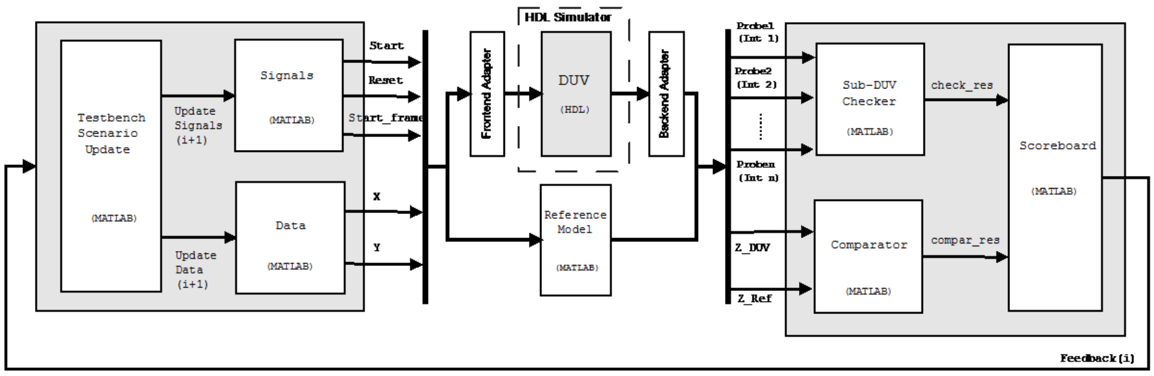

Figure 3. The verification structure

Inside the Data block, a Random Number Generator (RNG) produces constrained random large-integer data (X,Y operands) using a VPI seed value. An overview of the functioning of the platform's blocks is given in Table 1. The comparator receives data from DUV and Reference Model, converts it to VPI data type, makes the comparison, and finally transfers the result to the Scoreboard. Sub-DUV Checker checks the functional correctness of internal operations of DUV. It receives the probed metadata from the DUV, converts it to VPI data type, calculates the internal operation bit-wise as a reference operation, makes the comparison and finally transfers the result to the Scoreboard. As illustrated in Figure 3, the comparator and the Sub-DUV Checker update the scoreboard with results as long as the cosimulation is going on. The control logic verification, not represented in Figure 3, is done with PSL assertions simulated in the HDL Simulator.

Table 1. Functioning of platform blocks

Phase	Block	Corresponding pseudo-code

Stimuli preparation and Generation	Test-bench Scenario Update	new_seed=Compute_seed(feedback(i)) //constraint on seed to hit specific data interval
		Do{
		CRLI_1 = randint(vpi(new_seed)); (*) // CRLI = Constrained Random Large-Integer
		CRLI_2 = randint(vpi(new_seed));
		OK = Verify_seed (new_seed, CRLI_1,CRLI_2); } // to verify that RLI is constrained as desired
		While (NOT (OK)) // RLIs do not match the desired interval.

	Signals	Reset <= '1'; When T = T0, Reset <= '0'; *// T0 = n0 * CLK Cycles* When T = T1, Start_frames <= '1' *// T1 = n1 * CLK Cycles* When T = T'1, Start_frames <= '0'; *// T'1 = T1 + 1 CLK Cycle*

	Data	X<= VPI_to_binary_matrix(CRLI_1);
		Y<= VPI_to_binary_matrix(CRLI_2);

	Signals	When T = T_2, Start <= '1' // $T_2 = n_2$ * CLK Cycles When T = T'_2, Start <= '0'; // $T'_2 = T_2$ + 1 CLK Cycle
Reference Model		X_vpi<= binary_matrix_to_VPI(X); *// inside the Reference Model* Y_vpi<= binary_matrix_to_VPI(Y); *// inside the Reference Model* Z_Ref<= Reference_Model(X_vpi,Y_vpi); *// inside the Reference Model*
Verification execution and analysis	Compar-ator	Compare_res <= (Z_DUV == Z_Ref)?

	Sub-DUV Checker	Op_i,…,Op_k<= binary_matrix_to_VPI(Probe_i,…,Probe_k); // Operands of a selected internal operation are probed from DUV
		Int_Matlab_output<= Internal_operation(Op_i,…,Op_k); // The equivalent Matlab operation is calculated.
		Int_DUV_output<= binary_matrix_to_VPI(Probe_n); // The result of the internal operation is also probed
		Check_res<=(Int_DUV_output ==Int_Matlab_output)? // results are compared

	Property assertions (**)	… psl assert_done_pulse : assert always({done} |-> next {!done} abort !RST_N) @ (posedge CLK);-- signal DONE is a pulse of one clock cycle …
	Score-board	Data_ coverage=Measure_coverage(Compare_res,Check_res,CRLI_(coverage_step), Total_Assertions_coverage) feedback(i+1)=Compute_new_testbench_scenario(Data_ coverage);

* randint() is a random and uniformly distributed VPI number.
** "Property Assertions" is not a block of the platform; it runs in HDL simulator.

2.3. Large-integer Data Processing

Large-integers data processing is an important part of the platform. Processing is carried out in Matlab, Simulink and simulated hardware. We assume that the platform, shown in Figure 4, verifies the operation f: $Z = f(X,Y)$, where: X, Y and Z are three large-integers. Control signals are reset and start. Done is an output signal that indicates the end of the DUV's operation. Simulation control is handled by Simulink.

The co-simulation stage (stage 5 in Figure 4) contains the Reference Model and the DUV. The Reference Model is the DUV's equivalent model written in Matlab inside a Matlab Function block. The DUV is represented by the HDL Cosimulation block. The DUV's simulator (ModelSim) is launched and linked to Simulink using a Matlab code based on a TCL script. When communication is established, the simulator functions as the server and Simulink as a client. The HDL simulator responds to simulation requests it receives from the Simulink Client. The communication between the HDL Simulator and Simulink is done through the HDL Verifier™ tool. The maximum length of integer data types supported by HDL Cosimulation Block in Simulink is 128-bits. To work around this limitation, the DUV was masked in an HDL wrapper that stacks the received data frames into a logic vector that matches the input data size of the DUV and vice-versa for the output data. Two kinds of adapters were used in the platform (Frontend and Backend adapters). The first one adapts data and control signals received in Matlab/Simulink formats to DUV supported formats. Within this stage, each data matrix is converted into a sequence of scalars using the Simulink's block "Unbuffer". The Unbuffer unbuffers an M-by-N input into a 1-by-N output (Figure 5(a)). That is, inputs are unbuffered row-wise so that each matrix row becomes an independent time-sample in the output. As example, a 192-bit data fits into a 24-by-8 matrix, and the Unbuffer Block will unbuffer the 24-by-8 input into an 8-bit length vector. Then, each data is converted to standard logic vector via the “Data Type Conversion” block. The Backend adapter adapts data and signals from DUV to the Comparator and Checker blocks. In this stage, the HDL block output data is re-buffered into a decimal matrix using a Simulink block “Delay Line”. The latter performs the reverse task of the “Unbuffer Block”, rebuffering a sequence of Mi-by-N matrix inputs into a sequence of Mo-by-N matrix outputs (Figure 5(b)).

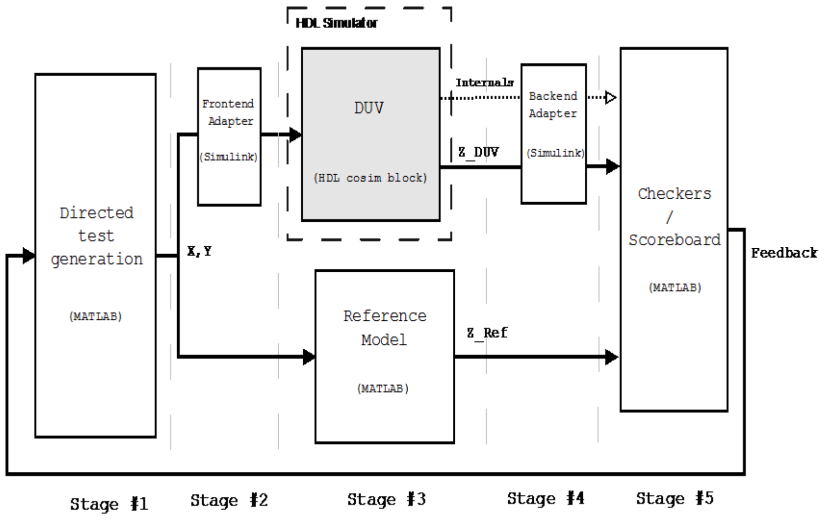

Figure 4. Block diagram of the proposed verification platform

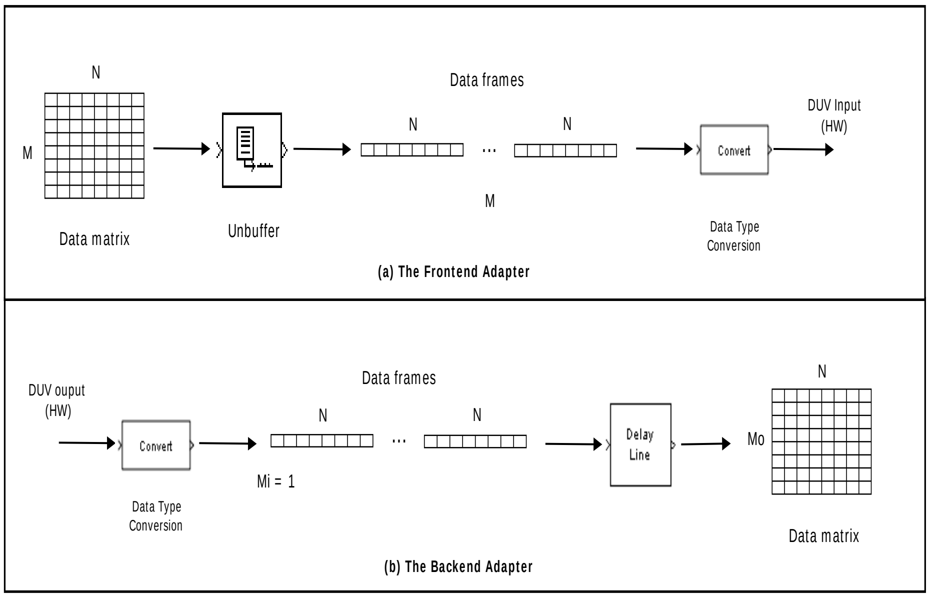

Figure 5. The adapters

An attention should be given to the reading time of the “Delay Line” output so that the data matrix can be read entirely. In fact, the DUV wrapper, detailed in a latter paragraph, was designed to send each of Z_DUV frames at every clock’s positive edge starting from instant when the output signal “done” is on and according to the “Delay Line” Block functioning, the entire matrix representing Z_DUV can be read by the Comparator Block at time T calculated in formula 2:

$$T=Time_{(done=1)}+Nbr\_of\_Z\_frames*T_{(Z\_sample\_period)} \quad (2)$$

Where: $\underline{\text{Time}_{(\text{done}=1)}}$ is the time when done is set to '1', $\underline{\text{Nbr}_{\text{of Z frames}}}$ is the total number of Z frames outputted from HDL Block, and $\underline{T_{(\text{Z sample period})}}$ is the sampling time of Z.

The Figure 6 illustrates the communication between Simulink and the DUV via the wrapper. As shown, Inputs (reset, start_frames, start, X, Y) are stimuli from Matlab/Simulink, while Outputs (Done, Z_DUV, Probe1… Probe n) are results sent back to Matlab/Simulink for comparison and internal checking. The start_frames is an extra input to the DUV Wrapper to control the reception of frames from Data block.

The role of the Wrapper is to handle a cycle-accurate transfer of data between Simulink and the DUV without modifying the latter's description. The wrapper , written in VHDL, is based on an Input converter and two Output converters. One dedicated to DUV's result, the other to internal signals (Figure 7).

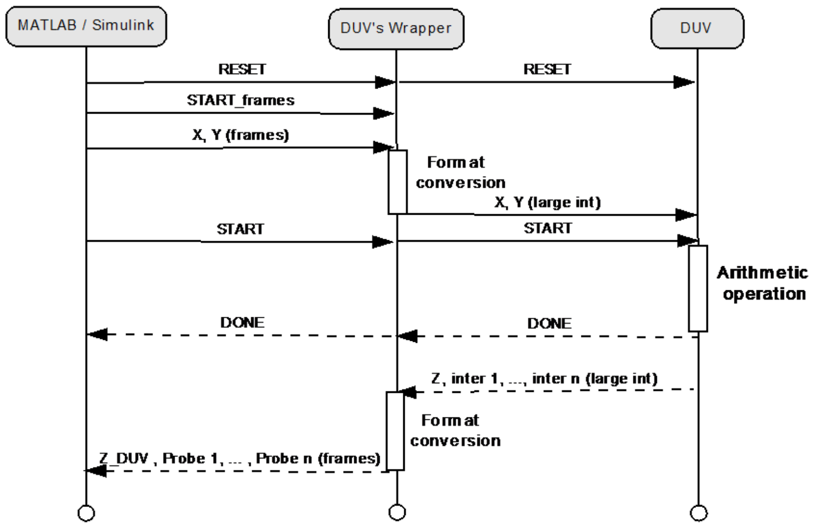

Figure 6. UML sequence diagram of Simulink – HDL block communication

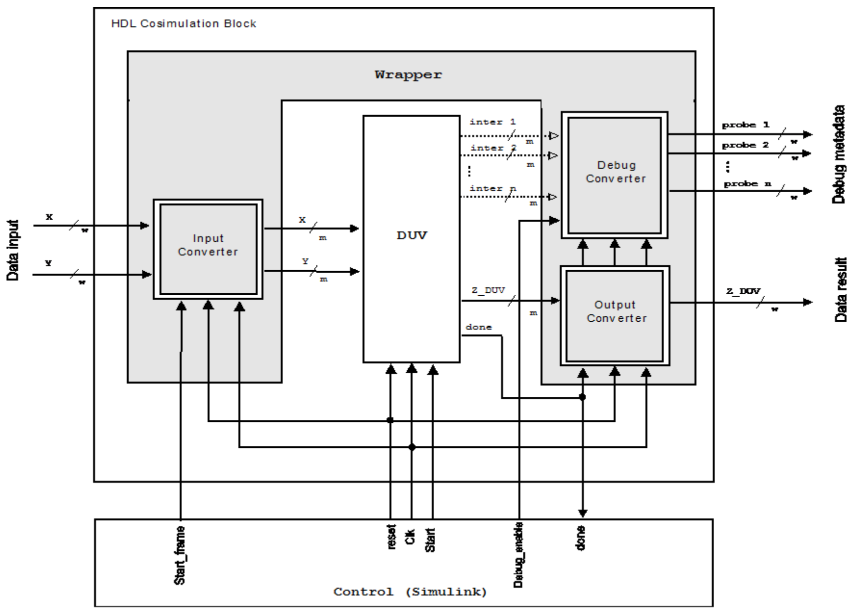

Figure 7. The HDL Cosimulation block data flow

As illustrated in Figure 8(a), The "Input Converter" module receives data (X,Y) from Matlab/Simulink, stacks the w-bit length frames (f_i) into Standard logic vector. The m-bit matching the size of the expected DUV input data size (f_0 to f_k frames) are extracted ("*unpadding*" operation). When the

"Output Converter" module receives the result, bits are added to the logic vector to bring it to the required size (f_{k+1} to f_n frames) ("*padding*" operation). Then, the logic vector is sent in w-bit frames to the next stage (Figure 8(b)). Similarly, the "Debug Converter" module brings DUV's internal signals to the next stage.

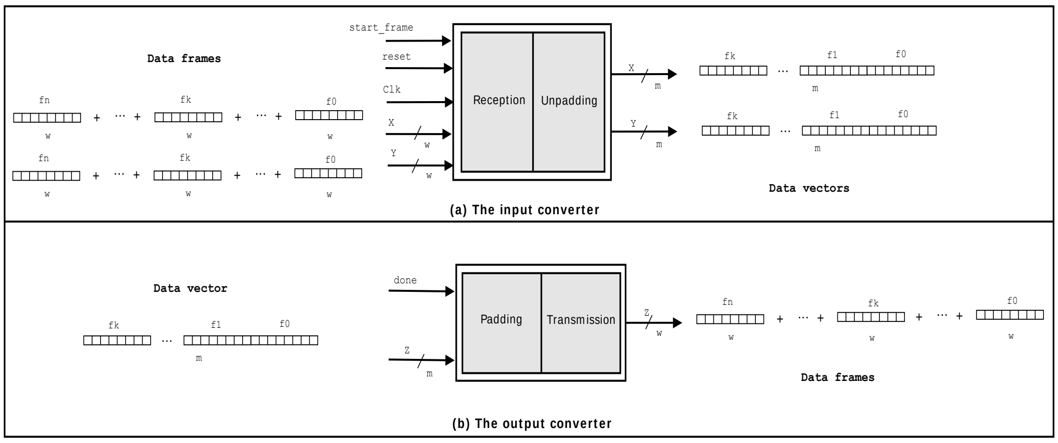

Figure 8. The DUV's Wrapper units.

2.4. Platform Control, Settings and Execution

An essential side of the platform is the control and settings. Platform control consists in controlling the execution of the testbench by scheduling the stimuli to the DUV/Reference Model and the outgoing signals/data to be verified. The challenge here is to synchronize the Matlab/Simulink blocks, which are inherently untimed, with an RTL-level design running in an event-based simulator (ModelSim). The Platform Control process is abstracted in the timed finite state machine (TFSM) represented in Figure 9.

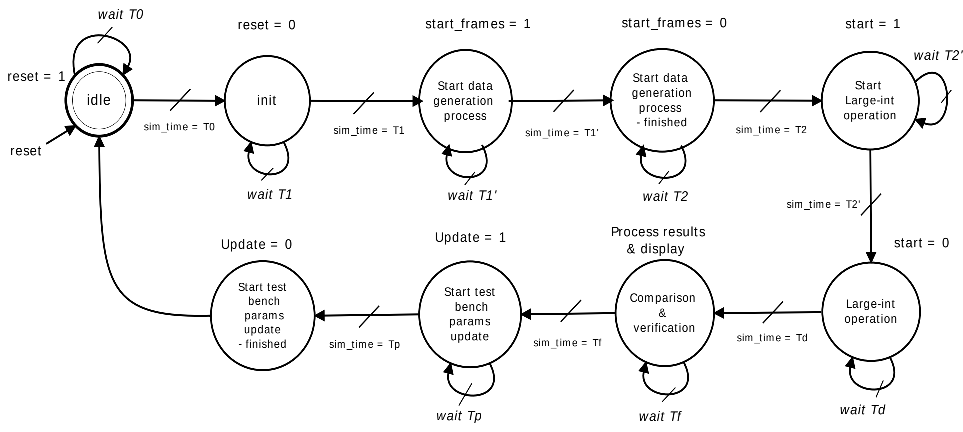

Figure 9. The finite state machine of the Platform Control

Because Matlab Function Block, and Matlab language in general, is untimed, the timing and the delivery of the data is controlled by the HDL simulator (when Matlab Function Block is located after the DUV) and/or by the Block's sampling time setting (when Matlab Function Block is located forward). Using a Simulink Digital Clock, the stimuli (control signals and data) are generated in specific simulation times. The transition delay times between the TFSM states are presented in Table 2.

Platform settings are the settings of parameters related to each block of the platform. That is, the Simulink blocks parameters (Unbuffer, Delay Line, *etc.*) and the sampling times for Matlab function blocks (Verification structure blocs). A graphical user interface (GUI) was developed to facilitate this task.

In addition to the choice of Simulink blocks and algorithms inside Matlab Function blocks, the functioning of the platform relies on the timing settings. In fact, for each block, a sample time needs to be specified. In Simulink, the sample time of a block is a parameter that indicates when, during simulation, the block is active and if appropriate, updates its internal state. For HDL cosimulation Block, a sample time has to be set for each input/output. Sample times of platform blocks were set to HDL clock period except the "reference model" whom the execution depends on the triggering signal received from the Stimuli Generator Block. The sample time of Data block can be calculated using formula 5 derived from formula 4:

$$UBO\_Sample\_Time = UBI\_Sample\_Time / Data\ Matrix\_rows\_Nbr \quad (4)$$

Where UBO is "Unbuffered Block Output"

$$UBI\_sample\_time = DUV\_clk\_period * Data\ Matrix\_rows\_Nbr \quad (5)$$

Where UBI is "Unbuffered Block Input"

As UBI is connected to Data block, the value of Data block's sample time is the same as that of UBI. Because the Unbuffer only accepts fixed-size input, output of Data sub-block cannot be set to variable size type. Therefore, data bit-size (operands) has to be set manually by user for different operands bit-size. In practice, Sample time can be set automatically by working around the restriction of the Unbuffer. To do so, the size of all data transferred between blocks in the platform are chosen as a constant that holds all standard sizes of finite-field commonly used in Public-key Cryptography (for instance, 1024). Therefore, the Number of data matrix rows is also a constant (fixed size-input) and the Sample Time of UBI, becomes only DUV's Clock dependent. The choice of a unique size simplifies the transmission/reception of data inside each block and only bits corresponding to operands real size (e.g. 192-bit) are used. This task is done by the Wrapper as it brings operands to the required size by the padding and unpadding processes previously detailed. User has to place the VHDL design code(s) inside a specific folder for co-simulation, adjust wrapper's parameters to the size of the HDL design operands and connect wrapper's probes (outputs) to desired DUV's internal signals. The Output Data Adapter, represented by the Simulink block "*Delay Line*", executes the reverse task of the Unbuffer Block. That is, it transforms a sequence of data frames into a matrix.

When verification is launched, PSL assertions test results are processed in Matlab workspace. Then, results are carried to Scoreboard along comparator and checker results. The Scoreboard computes the new verification coverage and generates a feedback summarizing the coverage. According to the feedback, a new testbench scenario and parameters targeting the uncovered assertions and/or datapath logic not yet verified are set. Then, the next testbench will be ready to run.

Table 2. Time Periods of the TFSM

Delay time symbol	Significance	Value
T_0	Time to wait before starting a new test	$T_0 = n_0$ * CLK Cycles
T_1	Time to wait before Starting Data generation process	$T_1 = n_1$ * CLK Cycles
T_2	Time to wait before Starting Data generation process	$T_2 = n_2$ * CLK Cycles
T_d	Time to wait before Done = 1	$T_d = n_d$ * CLK Cycles
T_f	Time to wait before Feedback is ready	$T_f = n_f$ * CLK Cycles
T_p	Time to wait before Testbench Parameters are updated	$T_p = n_d$ * CLK Cycles
$T_{x'}$	One clock cycle after T_x	$T_x' = T_x$ + CLK Cycle

2.5. Verification Platform with FPGA in-the-loop

Another aspect of reusability of the proposed platform is the possibility to switch from HDL cosimulation to real hardware testing while keeping the same verification platform. This option was tested with the "Hardware-in-the-loop" (HIL) option provided by Simulink for FPGA boards equipped with Gigabit Ethernet port (an Altera DE2-115 board with Cyclone IV EP4CE115 FPGA was used). This way enables controlling and verifying a design (a modular multiplier, more details in section 3) running on FPGA from the Matlab/Simulink platform with the design's real execution time (Figure 10). However, this came at a cost as that internal verification (Sub-DUV Probing) becomes inaccessible due to the FPGA development's tool restrictions on design's coding style and wrapping.

To conclude this section, Table 3 gives a comparison between the present work and similar works from literature. The Table shows that the proposed platform while sharing some features with other works (supported HDL, cosimulation, etc.), stands out with more powerful HVL, unrestricted large-integer support and adaptability to HIL.

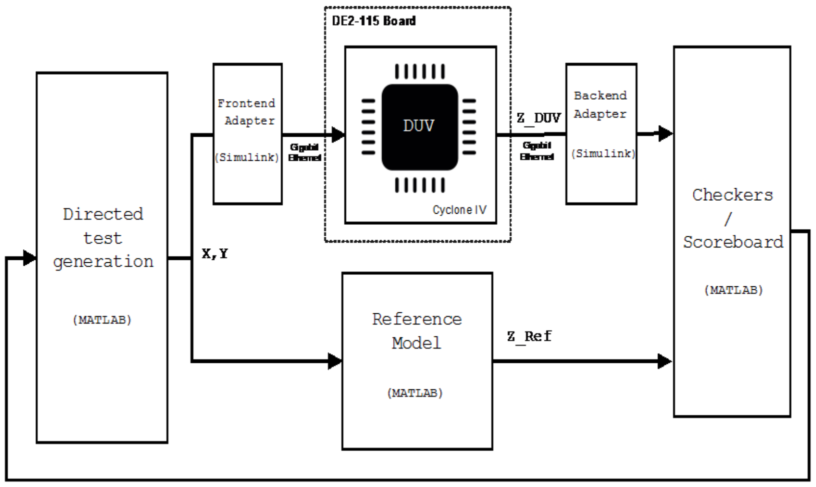
DE2-115 Board
Frontend Adapter (Simulink)
Gigabit Ethernet
DUV
Cyclone IV
Z_DUV
Gigabit Ethernet
Backend Adapter (Simulink)
Checkers / Scoreboard (MATLAB)
Directed test generation (MATLAB)
X,Y
Reference Model (MATLAB)
Z_Ref

Figure 10. Verification Platform with FPGA-in-the-loop

Table 3. Comparison with similar verification platforms

Verification environment	HW Verification Language (HVL)	Supported HDL	Interfacing with Simulator	Cosimulation with VHDL simulator	Large-integer support	DUV in Hardware
[13]	C	VHDL	FLI	Yes	Limited	No
[14]	Python	VHDL/Verilog	FLI/VPI	Yes	Limited	No
[15]	Python	Verilog	VPI	No	Limited	No
[25]	Python	Verilog	VPI	No	Limited	No
[26]	Ruby	Verilog	VPI	No	Limited	No
This Work	Matlab/Simulink	VHDL/Verilog	HDL Verifier® + Wrapper	Yes	Unlimited	Yes (HIL)

VPI : Verilog Procedural Interface, **FLI** : Foreign Language Interface,
PLI : Procedural Language Interface , **HIL** : Hardware-in-the-loop.

3. CASE STUDY & RESULTS

As case study of the platform, we consider the operation Z = *f(X,Y)* , where *X*, *Y* and *Z* are three large-integers. Control signals are *reset* and *start*, while *Done* is an output indicating the end of the operation. The goal is to evaluate the cost of the bit-size, the number of assertions and the internal signal probing on the platform.

3.1. Large-integer Arithmetic Background

Large-integer arithmetic has a variety of applications in cryptography. Among these, AES, RSA and ECC. As illustrated in the Figure 11, ECC schemes are based on Point operations, primarily on the point multiplication and also on the operations on which it point multiplication relies, i.e. point addition and doubling. In turn, those point operations are made on finite-fields arithmetic, a particular field of large-integers. This implies that finite-field arithmetic are determinant to design an efficient elliptic curve cryptosystem. Finite-field arithmetic is the arithmetic of integers modulo a large prime p. Arithmetic in a finite-field is different from standard integer arithmetic and all operations performed in the finite-field result in an element within that field. Three kinds of fields that are used for efficient implementation of ECC systems are prime fields (F_p), binary fields (F_{2^m}), and optimal extension fields (F_{p^m}). Those fields were extensively studied and this has resulted in numerous algorithms. Finite-field arithmetic is a practical example of large-integer arithmetic usage and is the cornerstone of cryptographic schemes such as ECC.

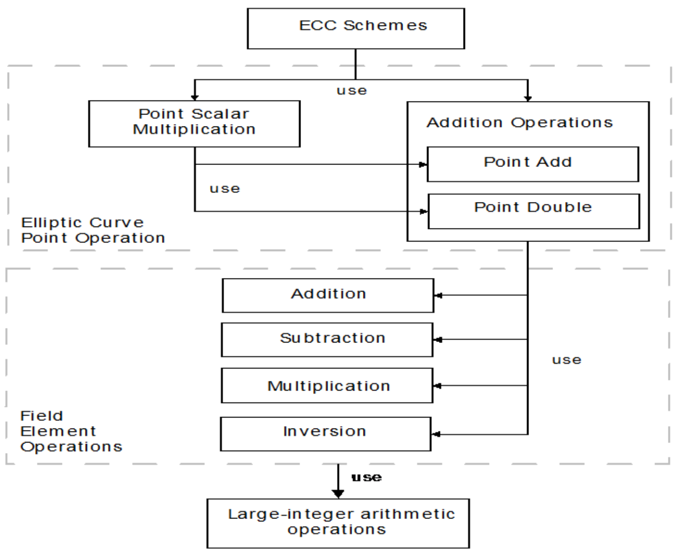

Figure 11. Hierarchy of required underlying operations

3.2. The DUV

A hardware implementation of a Finite-field multiplication algorithm called the “Double, add, and reduce” (DAR) multiplier [27] was used as a DUV. The DAR multiplier is based on the Interleaving Multiplication Algorithm [28]. Given a k-bit natural x and a natural y the product $z = x\ .\ y$ can be computed as follows formula 6:

$$x.y = (x_{k-1}\, 2^{k-1} + x_{k-2}\, 2^{k-2} + \ldots + x_0\, 2^0).y \tag{6}$$

The latter can also be expressed as in formula 7:

$$x.y = (\ldots\ ((0.2 + x_{k-1}\ y\ )2 + x_{k-2}\ y)2 + \ldots + x_1 y)2 + x_0 y \tag{7}$$

If all operations (addition and doubling) are executed mod m, the result is $product = x\ .y \bmod m$. The corresponding (left to right) algorithm, written in ADA syntax, is presented in Listing 1. The function “mod_m_addition(x, y, m, k)” computes $x + y \bmod m$; x, y, and m being k-bit numbers, according to the binary mod m Addition. This unit of the datapath represents an internal large-integer operation. In the design, operands were set to recommended sizes (192, 384, 512, 1024) for cryptographic use by the NIST [29].

Listing 1 Double, add, and reduce (DAR) algorithm.

```
p := 0 ;
for i in 0 .. k-1 loop
   p := mod_m_addition(p, p, m, k);
 if x(k-i-1) = 1 then
  p := mod_m_addition(p, y, m, k);
 end if;
end loop;
product := p;
```

The datapath and a part of control logic corresponding to the hardware description of Algorithm 1 are shown in Figure 12. The DUV is an ideal case for the platform testing with internal large-integer operation and a distinct control units and datapath. In practice, in addition to the functional validation (comparing DUV against reference model), each partition modules were verified. For datapath, “Mod m Adder” module was the target of internal checking while the control logic units were verified with PSL assertions.

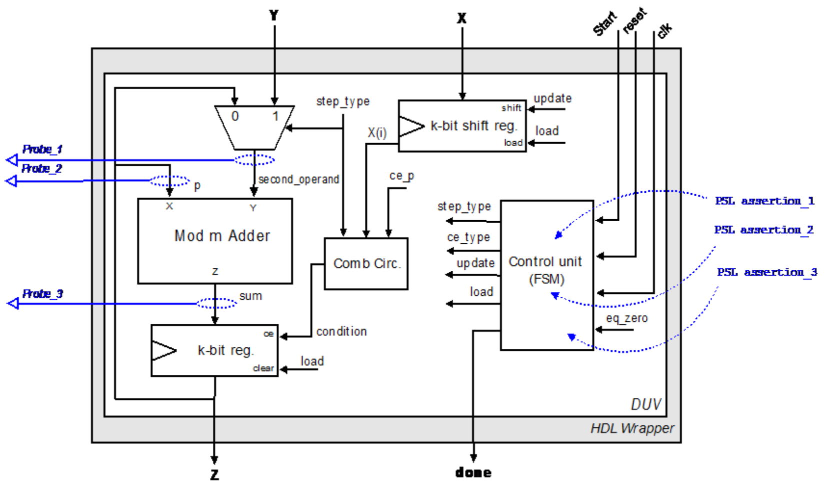

Figure 12. The DAR multiplier (DUV)

3.3. Tests, Results and Discussion

The DAR multiplier was verified with the platform on an 32-bit Intel Pentium Dual-Core processor (2,5 GHz, 2GB RAM, 2MB cache memory). The goal is to measure the cost over time of the bit-size, the number of assertions and the number of probes. A campaign of tests was carried out for each parameter.

Detailed execution times of the platform as a function of operands sizes were measured using Simulink's Profiler. The results are presented in Figure 13. The latter shows that the total recorded time increases quite linearly with operands size but is still acceptable for an operation on 1024-bit. For the four bit-sizes, the "HDL co-simulation block" occupies a small portion of the execution time (between 2.81 % and 5.18%) and remains rather constant. As can be seen in the same figure, the total recorded time is dominated by the initialization in average sizes (192 and 384). This aspect decreases in larger sizes in favor of the group of blocks " data, Scoreboard, and testbench Updater" reaching ≈40% of the total recorded time for 1024-bit size. This can be explained by the fact that the time used to the guided generation, adaptation and transmission of data stimuli increases with bit-size. It should be noted that in order to get a correct measure of bit-size cost, data across the platform was transmitted/received in the exact bit-size without using the padding/unpadding operations and related automated settings.

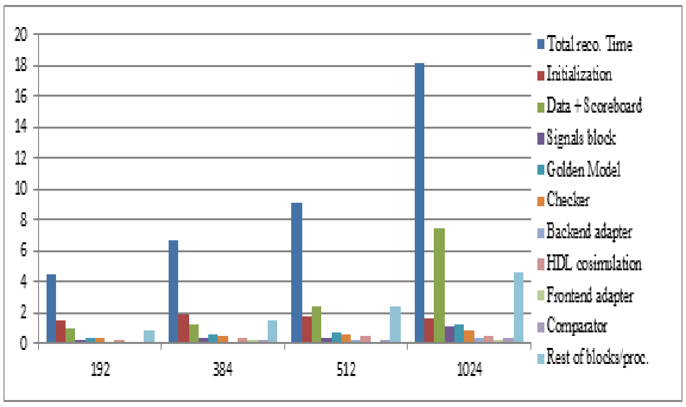

Figure 13. Execution time (in sec) of platforms' blocks

In order to evaluate the impact of the PSL assertions on the execution time of the platform, measurements of the latter function of the number of PSL assertions were made and the results, for the 4 sizes, were plotted in Figure 14. To reach a high number of assertions, the assertions of the testcase were replicated. As shown in the in the four curves, the number of PSL assertions checked during co-simulation is fairly stable for assertions below 40 PSL assertions. From 40 assertions, the execution time increases linearly but with a low slope for the 4 curves ($0.0052 < \alpha i < 0.0381$; i = 192; 384; 512; 1024). It can be concluded that the PSL assertions increase the execution time but the more the number gets larger (> 100), the more the impact on time is limited. It should be noted that the “steep slope” aspect of the curve is due to the high steps taken, on the horizontal axis, from the 200th assertion.

To evaluate the impact of the number of Probes on the execution time, measurements of the platform time for a fixed DUV bit-size (1024 bit) function of the number of probes were made and the results are shown in Figure 15. In this test, the outputs of the "HDL co-simulation" block, the number of sub-blocks of the "Backend Adapter" stage and the inputs of the "Checker" were adjusted to match the corresponding number of probes. According to results, when the number of probes increases, the execution time increases linearly but with a low slope ($\alpha \approx 0.4$). It can be said that the number of probes increases the execution time of the verification process but does not penalize it especially because a small number of probes is generally needed for verification.

Analysis of the three tests campaign results indicates that the parameters (bit-size, number of assertions and number of probes) has only a moderate impact on the execution time of the verification platform, thus justifying its efficiency.

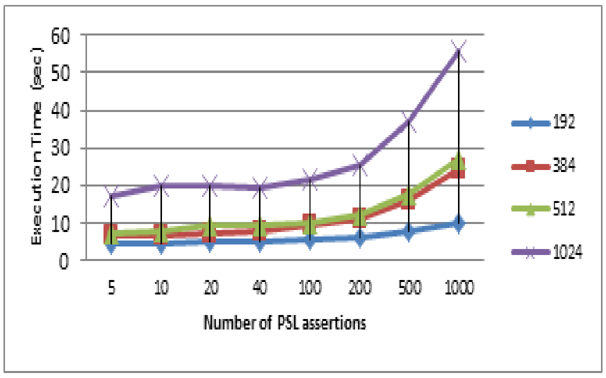

Figure 14. Platform's execution time (in sec) as a function of PSL assertions

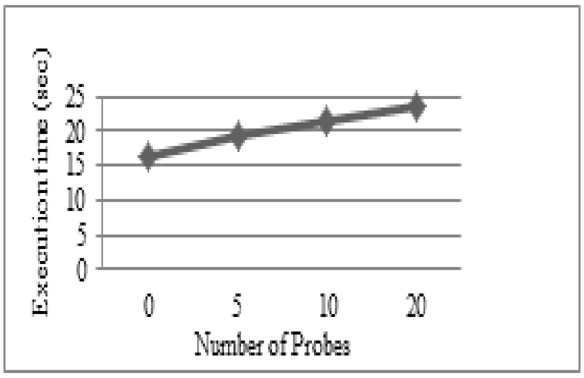

Figure 15. Execution Time (in sec) as a function of number of probes

4. CONCLUSION

In this paper, we have presented a novel platform intended to verify hardware large-integer based designs, the first one based on Matlab/Simulink to the best of our knowledge. We demonstrated that the proposed platform holds a number of interesting aspects for the task of verification. First, this is run time and cycle-accurate verification. Second, flexibility, where minor adjustments in Matlab/Simulink blocks parameters, different bit-length can be verified with a moderate impact on execution time. Besides, testbench scenarios are adjustable to meet desired verification coverage where datapath and control logic can be verified simultaneously and in different level of the design hierarchy. Third, reusability: In this paper, we developed testcase on finite-field arithmetic but we also tested the platform to verify a scalar multiplication (Figure 11) this proves that the platform is adapted to more complex systems like cryptographic primitives.

Future work will involve improvements like reducing synchronisation and data transfer overhead, limiting the complexity of the wrapping module, and enabling internal verification in HIL. Furthermore, an interesting area of application of the platform that would need further efforts is the verification of designs under development, with possibility of replacing unachieved blocks with equivalent Matlab/Simulink models. Another extension of the platform, in the field of cryptanalysis, could be using Matlab’s data processing features to verify design robustness to side-channel and fault injection attacks in HIL.

REFERENCES

[1] R. L. Rivest, A. Shamir, and L. Adleman, “A method for obtaining digital signatures and public-key cryptosystems,” *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978.

[2] Victor S. Miller, “Use of Elliptic Curves in Cryptography,” in *Advances in Cryptology — CRYPTO ’85 Proceedings*, 1986, vol. 218.

[3] N. Koblitz, “Elliptic curve cryptosystems,” *Mathematics of Computation*, vol. 48, no. 177, pp. 203–203, Jan. 1987.

[4] Y. Kong, S. Asif, and M. A. U. Khan, “Modular multiplication using the core function in the residue number system,” *Applicable Algebra in Engineering, Communication and Computing*, Jul. 2015.

[5] M. T. Hamood and S. Boussakta, “Efficient algorithms for computing the new Mersenne number transform,”

Digital Signal Processing, vol. 25, pp. 280–288, Feb. 2014.

[6] G. X. Yao, J. Fan, R. C. C. Cheung, and I. Verbauwhede, "Novel RNS Parameter Selection for Fast Modular Multiplication," *IEEE Transactions on Computers*, vol. 63, no. 8, pp. 2099–2105, Aug. 2014.

[7] T. Wu, S. Li, and L. Liu, "Fast RSA decryption through high-radix scalable Montgomery modular multipliers," *Science China Information Sciences*, vol. 58, no. 6, pp. 1–16, Mar. 2015.

[8] S. Antão and L. Sousa, "A Flexible Architecture for Modular Arithmetic Hardware Accelerators based on RNS," *Journal of Signal Processing Systems*, vol. 76, no. 3, pp. 249–259, 2014.

[9] K. Jarvinen, V. Dimitrov, and R. Azarderakhsh, "A Generalization of Addition Chains and Fast Inversions in Binary Fields," *IEEE Transactions on Computers*, vol. 64, no. 9, pp. 2421–2432, Sep. 2015.

[10] G. D. Sutter, J. Deschamps, and J. L. Imana, "Efficient Elliptic Curve Point Multiplication Using Digit-Serial Binary Field Operations," *IEEE Transactions on Industrial Electronics*, vol. 60, no. 1, pp. 217–225, Jan. 2013.

[11] S. Suma and V. Sridhar, "Design of Multiplier for Medical Image Compression Using Urdhava Tiryakbhyam Sutra," *International Journal of Electrical and Computer Engineering (IJECE)*, vol. 6, no. 3, pp. 1140–1151, 2016.

[12] G. Arepalli and S. B. Erukula, "Secure Multicast Routing Protocol in Manets Using Efficient ECGDH Algorithm," *International Journal of Electrical and Computer Engineering (IJECE)*, vol. 6, no. 4, pp. 1857–1865, 2016.

[13] A. Pool, "Using ModelSim Foreign Language Interface for C – VHDL Co-Simulation and for Simulator Control on Linux x86 Platform," 2014.

[14] Potential Ventures, "cocotb : COroutine based COsimulation TestBench environment for verifying VHDL/Verilog RTL," 2014.

[15] B. Smith, T. Loftus, J. Greene, and X. Wu, "PyHVL, a verifcation tool," 2007.

[16] F. Farahmandi and B. Alizadeh, "Groebner basis based formal verification of large arithmetic circuits using Gaussian elimination and cone-based polynomial extraction," *Microprocessors and Microsystems*, vol. 39, no. 2, pp. 83–96, Mar. 2015.

[17] T. Pruss, P. Kalla, and F. Enescu, "Efficient Symbolic Computation for Word-Level Abstraction From Combinational Circuits for Verification Over Finite Fields," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 7, pp. 1206–1218, Jul. 2016.

[18] C. Siggaard, "Using MatLab to aid the implementation of a fast RSA processor on a Xilinx FPGA," in *Nordic MathWorks User Conference*, 2008.

[19] D. B. Cousins, K. Rohloff, C. Peikert, and R. Schantz, "An update on SIPHER (Scalable Implementation of Primitives for Homomorphic EncRyption) — FPGA implementation using Simulink," in *2012 IEEE Conference on High Performance Extreme Computing*, 2012, pp. 1–5.

[20] D. Cousins, J. Golusky, K. Rohloff, and D. Sumorok, "An FPGA Co-Processor Implementation of Homomorphic Encryption," in *2014 IEEE High Performance Extreme Computing Conference*, 2014.

[21] P. C. Realpe, V. Trujillo-Olaya, and J. Velasco-Medina, "Design of elliptic curve cryptoprocessors over GF(2^163) using the Gaussian normal basis," *Ingeniería e Investigación*, vol. 34, no. 2, pp. 55–65, Jul. 2014.

[22] A. Kaleel Rahuman and G. Athisha, "Reconfigurable Architecture for Elliptic Curve Cryptography Using FPGA," *Mathematical Problems in Engineering*, vol. 2013, pp. 1–8, 2013.

[23] N. Alimi, Y. Lahbib, M. Machhout, and R. Tourki, "Simulation-based verification of large-integer arithmetic circuits," in *2016 1st IEEE International Verification and Security Workshop, IVSW 2016*, 2016, pp. 19–24.

[24] Accellera, "Property Specification Language Reference Manual." 2004.

[25] Decaluwe and Jan, "MyHDL: a Python-based hardware description language," *Linux Journal*, no. 127, p. 5, 2004.

[26] S. N. Kurapati, "Specification-driven functional verification with Verilog PLI & VPI and SystemVerilog DPI," 2007.

[27] J.-P. Deschamps, *Hardware Implementation of Finite-Field Arithmetic*. McGraw Hill Professional, 2009.

[28] F. Rodriguez-Henriquez, N. A. Saqib, A. D. Pérez, and C. K. Koc, *Cryptographic Algorithms on Reconfigurable Hardware*. Springer Science & Business Media, 2007.

[29] E. Barker, W. Barker, W. Burr, W. Polk, and M. Smid, "NIST Special Publication 800-57, Recommendation for Key Management Part 1: General (Revision 3)," 2012.