FPGA-Assisted Assertion-Based Verification Platform

Nurita Mohamad[1,2], Chia Yee Ooi[1], Jwing Teh[2], Norlina Paraman[3], Hasliza Hassan[4] and Nordinah Ismail[1]
[1]*Malaysia-Japan International Institute of Technology, Universiti Teknologi Malaysia, Kuala Lumpur, Malaysia.*
[2]*Intel Programmable Solutions Group Technology Centre, Plot 6 Bayan Lepas Technoplex, Penang, Malaysia.*
[3]*School of Electrical Engineering, Universiti Teknologi Malaysia, UTM Johor Bahru, Johor, Malaysia.*
[4]*Faculty of Electrical and Electronic Engineering, Universiti Tun Hussein Onn Malaysia, Batu Pahat, Johor, Malaysia.*
nuritamohamad89@gmail.com

***Abstract*— In this paper, field programmable gate array (FPGA)-assisted verification platform is devised to enhance the assertion-based verification methodology to address the issues of high demand of integrated circuit with the advanced features to be delivered to market within tight Time-To-Market. The concept of SystemVerilog Assertion (SVA) checker generator is introduced to translate non-synthesizable verification coding into hardware so-called assertion checker in Verilog. A lookup table, which comprises of SVA operators mapped to their corresponding synthesizable Verilog coding was developed to generate assertion checker, which produces a single bit 1 when the assertion fails. Collection module implemented using a memory block and an arbiter was devised to be simple and fast enough to collect assertion results from the assertion checker. Since assertion checker can produce assertion result at any time, an arbiter is required to act as an interface between assertion checker and collection module. Case studies have been conducted on the proof-of-concept designs, which are the first-in-first-out (FIFO), up-down counter and Context Adaptive Variable Length Coding (CAVLC) to evaluate the effectiveness of the proposed FPGA-assisted verification platform. In the case studies, we have shown that the proposed FPGA-assisted verification platform works correctly. Besides, we also evaluated the method in area utilizations (ALMs). It has been proven that simulation-based verification time can be reduced for as much as 50% for complexity of VLSI design. Thus, implementing assertions using hardware such as FPGA becomes a solution to alleviate issue of long simulation time.**

***Index Terms*—Arbitration; Assertion-based Verification; Assertion checker; FPGA; System Verilog Assertion; Verilog.**

I. INTRODUCTION

With design ability lagging behind the fabrication ability, electronic production is facing the risk of product re-spins. In this case, verification can be considered as one of the ways to address this issue. Verification is a process that ensures the implemented device is matched with the product intended to define for the device before sending the device for manufacturing. Verification requirements have increased exponentially due to the increased complexity of hardware designs. Furthermore, it has become increasingly critical in the product development cycle, requiring effort at least 70% of the system on chip (SoC) development cycle [1, 2]. When market windows become increasingly tight, high-quality verification is essential for successful chip prototype delivery to reduce the probability of delayed tape-out and re-spin [3]. However, verification ability is still far lagging behind the fabrication. Thus, there is a rising edge of the verification gap, in which the new acceleration methodology needed to cover up these issues and optimize the productivity gap. Assertion-based verification (ABV) was then introduced to reduce the gap.

In ABV, assertions are used to ensure the design fulfils its given specification. Assertions are additional statements that are bound together with Device-under-Verification (DUV) to check the design behavior [4]. This improves the observability of verification compared to the conventional verification, which allows only observation at DUV's output. The role of assertions in ABV intensely improves the efficiency of detecting bugs and monitors the behavior for a set of given input stimuli in verification [5, 6]. In addition, ABV is widely gaining acceptance in the industry because it has been proved that simulation-based verification time can be reduced for as much as 50% when using ABV because it directly gives feedbacks for the correctness of a design property or behavior specified [7]. In fact, it improves the observability of verification as can be seen in the successful identification of many corner case bugs [8, 9]. Moreover, ABV allows reusability of coding framework [10]. The increasing complexity of VLSI design demands a large number of complex assertions to be simulated so verification needs longer simulation time. Thus, implementing assertions using hardware such as field programmable gate array (FPGA) becomes a solution to alleviate issue of long simulation time [11–14].

Hardware-assisted verification can be divided into emulation and prototyping. Some research suggested synthesizing assertions for verification using hardware emulation [15-18]. Examples of emulator in the market are Palladium from Cadence, Veloce from Mentor Graphic and Zebu from Synopsys. Hardware emulation is usually used when the circuit design is complex because hardware implementation can exploit the parallel nature of a logic circuit to verify DUV that is mapped onto the hardware. Author in [18] debugged the design in FPGA-based prototyping board with emulation mode and control the functionality of synthesized assertions in PSL, monitors and checker. However, the emulator environment is only capable to provide user's viewer based on GUI software such as to observe changing of debug symbols and it does not mention how to capture the assertion result. Thus, user needs to observe assertion manually and this action might create a possibility of assertion losses. Some issues on coverage measurement have been identified, one of which is little debug capability when using an FPGA-based prototype since signal as well as assertions cannot be observed using a waveform viewer [19]. Although using scan-chain techniques

can help user in coverage measurement, signals must propagate through the entire chain until they reach the output before we can observe the bug that happens in the scan register.

It has been proved in [19] that emulation could be 82% faster than the regular register transfer level (RTL) simulation. Unfortunately, when logic design becomes larger, the area utilized in hardware emulation becomes bigger. Thus, there is a trade-off between the speed of the verification and logic utilization. Another approach to solve the performance issues is using emulator platform based on commercial FPGA, which enables execution at high speed but limited in terms of flexibility. Besides, emulator is costly in the market. In addition, the generated hardware must consume the least amount of hardware resources. Yet, emulator has a limitation, whereby it cannot run at full speed and connect the same system as the final design [20]. Thus, the number of engineers required to run the emulation at the same time increases as more emulators are needed to complete the complex design.

Prototyping is a hardware representation of a design which is often created using FPGAs that operates at the target system speeds. FPGA prototyping is one of the platform for system integration and verification. Software integration is depending on the quality of the test case applied during simulation and the ability in writing the assertions for discovering any bugs in the design [21]. By using FPGA prototyping, the assertions can only be asserted when it is necessary and can utilize fewer hardware resources. This method may increase the observability of the design and reduce time for debugging [22].

To map assertions onto emulator or prototype, we first need to synthesize assertions into Hardware Description Language (HDL). Therefore, hardware assertion checker generator tool has become an attractive solution. There are commonly hardware verification languages to express the assertions such as property specification language (PSL) and SystemVerilog Assertion (SVA). The assertion checker generation is typically a synthesizer to generate HDL that describes the given assertions behavior to be used in hardware-assisted verification [23]. FOCs is a tool developed by IBM to automatically generate checkers for design properties by synthesizing property specification in RTCL into finite state machine in VHDL [24]. Thus, FoCs does not process assertions written in any standard verification language. Authors in [23] have developed a checker generation called MBAC, which deals with a complete set of PSL properties. MBAC can automatically generate hardware assertion checkers in VHDL from the given assertions to be used for hardware emulation. In MBAC checker generation process, a PSL property is written in temporal logic equation. Boolean logic part is first transformed into edges of automata whereas temporal logic part is transformed into states in the automata. Each state is then mapped to a flip-flop whereas each edge is mapped to a combinational logic when synthesizing the automata into checker circuit. One drawback is that a property that consists of n repetition counts or repetition range of n will be synthesized into a circuit with at least n flip-flop. However, this could be further optimized by using counter as count of n could be realized by counter consisting of $\lceil \log_2 n \rceil$ flip-flops. This means the area consumed by verification should not be large to ensure the size of the design is not large. Note that $\lceil \log_2 n \rceil < n$.

Besides PSL, assertion checker based on SVA is also important because SVA has verification features not found in PSL. Different from PSL, SVA is a part of SystemVerilog language that can be written directly into SystemVerilog design and testbenches [25]. This allows faster identification of design bugs. Furthermore, SVA inherits the expression language of SystemVerilog, including its data types, expression syntax and semantics. Intuitively, translating SVA to synthesizable Verilog codes is easier since the languages are similar.

II. METHODOLOGY

A. FPGA-Assisted Verification Platform

In our verification platform, we proposed FPGA-assisted verification platform, as shown in Figure 1. It consists mainly of two tools; SVA checker generator and SVA signal extractor. SVA checker generator is to synthesize the assertions required by DUV into hardware that is so-called Assertion Module using modular approach whereas SVA signal extractor is to generate necessary signals to connect DUV and the Assertion Module. Both the DUV and the Assertion Module are assumed to be at register-transfer level (RTL). Meanwhile, we also proposed the collection module to collect assertion results from the assertion module. Collection module is composed of a memory block as storage and an arbiter to interface between the memory and the assertion module. SVA signal extractor is also responsible to generate signals that connect verifiable hardware module (combining both DUV and assertion module) and the collection module.

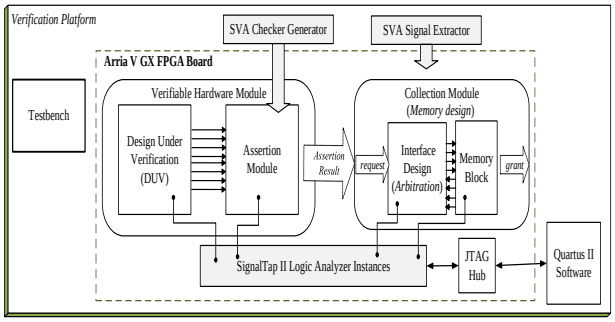


Figure 1: Proposed verification platform consist of verifiable hardware module and collection module

B. SVA Checker Generator

SVA describes SystemVerilog behavior of a given DUV for verification purpose. Unfortunately, it is only for simulation-based verification because these syntax, operator and system function are not synthesizable. Our approach is to synthesize all the required assertions into RTL hardware called assertion checkers. Thus, the given verification file should be translated to synthesizable Verilog coding.

Referring to [5], syntax of a property declaration and sequence declaration are described as follows:

property_declaration ::=
propertyproperty_identifier[property_formal_list]';'
{ property_decl_item }
property_spec ';'
endproperty[':' property_identifier]

sequence_declaration ::=
sequencesequence_identifier[sequence_formal_list] ';'

```
    { assertion_variable_declaration }
    sequence_expr ';'
endsequence[':'sequence_identifier ]
```

The property uses formal mathematical techniques to describe some behavior of a design to be verified [20]. Sequence is the building block of properties and concurrent assertions where the evaluation is based on clock semantic that ignores all the glitch occurrence [5]. Since both the property and the sequence are non-synthesizable, they are transformed into hardware modules by the proposed SVA checker generator. In the *property_formal_list* (resp. *sequence_formal_list*), which lists variables involved in the *property* (resp. *sequence*), SVA checker generator adds one additional output named *property_identifier'_out'* (resp. *sequence_identifier'_out'*) during the generation process. This additional output is to indicate whether the *property* (resp. *sequence*) is fulfilled (resp. occurring) or not when the assertion is being monitored. The *property_decl_item a*nd the *assertion_variable_declaration* are the declaration of local variables required by the property and sequence, respectively. *Property_spec* consists of event control, such as clocking event and asynchronous reset, and property expression. Both clocking event and asynchronous reset are translated to Verilog coding of clocking event and asynchronous reset. *sequence_expr* denotes sequence expression. Both property expression and sequence expression are composed by SVA operators and system function. SVA checker generator translates these operators and functions to RTL synthesizable Verilog modules.

Table 1 is the lookup table that summarizes a set of important and frequently used SVA operators and system functions, and their corresponding synthesizable Verilog coding used in SVA checker generation. *s1_out* (resp. *p1_out*) is the signal that indicates whether *s1* (resp. *p1*) occurs or not whereas *op_i* denotes the Boolean signal that represents a coverage result for the operator or system function. *temp_clock* is used in some operators with *posedge* as a temporary clock direction because an actual clock signal name and its triggering direction could only be known from the complete property declaration's clocking event given a DUV. Note that the inversion of coverage result will provide the assertion result. The synthesis detail of each operator or system function is elaborated in the following subsections.

Different from automata concept in MBAC [5], the concept of counter in translating *consecutive repetition* and *goto repetition* is used to reduce the number of flip-flops required. For temporal delay *s1##N s2*, shift registers is used to store the occurrence of *s1* and *s2* for *N* clock cycles and occurrence of *s1* at clock *N-1* is compared with the occurrence of *s2* at clock *0*.

Table 1
Lookup Table for SVA Checker Generation

No.	SVA Operators and System Function	Synthesizable Verilog Coding

i	Consecutive repetition: s1 [*N:M]	always @(posedge temp_clock)
		begin
		if (s1_out==1'b0)
		begin
		count = 2'b0;
		enable =1'b0;
		end
		if (s1_out==1'b1)
		enable =1'b1;
		if (count<M && enable)
		count = count+ 2'b01;
		else count=count;
		if (count>=N && count <=M)
		op_i=s1_out;
		else op_i=1'b0;
		end

ii	Goto repetition: s1[->N:M]	parameter N = 2; parameter M = 4;
		reg enable;
		
		always @(posedge temp_clock)
		begin
		if (s1_out==0)
		enable = 1'b0;
		else if (s1_out ==1)
		enable =1'b1;
		if (count<=M && enable==1'b1)
		count = count+2'b01;
		if (count>=N && count<=M)
		op_i = s1_out;
		else
		op_i = 1'b0;
		if (count == M)
		count = 2'b0;
		else if (count == 2'b0 &&
		enable == 1'b1)
		op_i = 1'b1;
		end

iii	Temporal delay: ##N , ##[N:M], Eg: s1##N s2	always @(posedge temp_clock)
		begin
		s1_reg = {s1_out, s1_reg[N-1:1]};
		s2_reg = {s2_out, s2_reg[N-1:1]};
		if (s1_reg[0] && s2_out &&
		(s2_reg [N-2:0] == { (N-1) {1'b0} }))
		op_i=1'b1;
		else
		op_i=1'b0;
		end

iv	And : s1 and s2	always @(posedge clock)
		begin
		if (s1_off && s2_off)
		begin
		temp_s1 <= 0; end
		else if (s1_out)
		temp_s1 <= s1_out;
		else
		temp_s1 <= temp_s1;
		end
		always @(posedge clock)
		begin
		if (s1_off && s2_off)
		begin
		temp_s2 <= 0; end
		else if (s2_out)
		temp_s2 <= s2_out;
		else
		temp_s2 <= temp_s2;
		end
		always @(posedge clock)
		begin
		if (s1_off && s2_off) begin
		comb_out <= 0; end
		else if (s1_on && s2_on)
		comb_out <= s1_on & s2_on;
		else
		comb_out <= comb_out;
		end
		assign op_i = temp_s1 & emp_s2 &
		comb_out;

v	Intersection: s1 intersect s2	always @(posedge clock)
		begin
		if (s1_off && s2_off) begin
		match_start <= 0; end
		else if (s1_on && s2_on)
		match_start <= s1_on & s2_on;
		else
		match_start <= match_start;

No.	SVA Operators and System Function	Synthesizable Verilog Coding
		op_i <= s1_out & s2_out & match_start; end
vi	Condition: if(expr) p1 if(expr) p1 else p2	assign op_i=(expr)?p1_out:1'b0; assign op_i=(expr)?p1_out:p2_out;
vii	Overlapping Implication: s1 |-> p1	always @(posedge clk) op_i <= |s1_out && |p1_out;

viii	Non-Overlapping Implication:
	s1|=> p1
		always @ (posedge clk)
		begin
		s1_prev <= s1_out;
		op_i <= |s1_prev && | p1_out;
		end

ix	System function:
	$past (A)
		always @ (posedge clk)
		op_i <= temp_A;
		assign temp_A = A ? A :1'b0;

x	System function:
	$stable (A)
		always @ (posedge clk)
		old_A <= A;
		assign op_i=(A==old_A)?1'b1:1'b0;

Consecutive repetition *s1 [*N:M]* is a repetition of sequence *s1* for *N* times or between *N* to *M* times. The repetition can hold any number of cycles including zero [5]. As shown in Table 1, *s1[*N:M]* is synthesized into a counter that has input *s1_out* to indicate the occurrence of sequence *s1* and produces the output *op_i* whenever the occurrence of *s1* is at N^{th} to M^{th} time. The number of bits for the counter is $\lceil \log_2 N \rceil$, which is smaller than *N*. The counter will only be enabled to start counting when *s1* has occurred. The counting continues whenever a new occurrence of *s1* takes place while the count is less than *M*. Note that *M* and *N* are declared as parameters of the counter. For example, *s1[*2:4]* specifies that *s1* is repeating for 2 to 4 times.

Go to repetition *s1[->N:M]* is a repetition of sequence *s1* for between *N* to *M* times. The repetition might be found at the end of the Boolean expression because it is non-consecutive cycles [5]. As shown in Table 1, *s1[→N:M]* is synthesized into a counter that operates based on input *s1_out* that indicates the occurrence of *s1* and produces the output *op_i* whenever the occurrence of *s1* is between N^{th} to M^{th} time. The number of bits for counter is $\lceil \log_2 N \rceil$. The counter will only be enabled to start counting when *s1* has occurred. The counting continues whenever a new occurrence of *s1* takes place while the count is less than or equal to *M*. Note that *M* and *N* are declared as parameter of the counter. For example, *[-> 2:4]* specifies that *s1* is repeating for 2 times, 2 to 4 times or 3 to 4 times.

Temporal delay operator *s1##N s2* specifies that when *s1* occurs on the current clock tick, *s2* must occur on the N^{th} subsequent clock tick. As shown in Table 1, *s1##N s2* is synthesized into a pair of registers, namely *s1_reg* and *s2_reg*, of length *N* each, which store the occurrence of *s1* and *s2*, respectively. Note that *s1_out* and *s2_out* feed to shift registers *s1_reg* and *s2_reg* respectively. When *s1_out* is high and the content of *s2_reg* is zero, this indicates the occurrence of *s1##N s2*, so it produces output *op*_i. For example, *s1##4 s2* specifies that *s2* must happen after *s1* occurs after 4 clock cycles.

Operator *s1 and s2* is a match operator of two sequences that occur at the same time but may end at different time. Under the coverage, the expression of this operator is only true when the last sequence has finished [5]. As shown in Table 1, *s1 and s2* were synthesized based on *s1*'s starting signal *s1_on*, *s1*'s ending signal *s1_off*, *s2*'s starting signal *s2_on* and *s2*'s ending signal *s2_off*.

Intersect operator *s1 intersect s2* is a match operator of two sequences that occur and end at the same time. Intersect operator is similar to *and* operator, but it is more similar to logical *and* in Verilog. As shown in Table 1, *s1 and s2* were synthesized based on *s1*'s starting signal *s1_on*, *s1*'s ending signal *s1_off*, *s2*'s starting signal *s2_on* and *s2*'s ending signal *s2_off*.

Operators *if (expr) p1* and *if (expr) p1 else p2* are the conditional operators. These operators are similarly with the procedural *if()* statement in Verilog [5]. As shown in Table 1, *if (expr) p1* and *if (expr) p1 else p2* were synthesized into a conditional Verilog operator that operates according to condition of *expr* that indicates the evaluation on *p1*. *p1* and *p2* denote a property and *expr* denote an expression given in the design. For example, *if (expr) p1* specifies that *p1* evaluates to true if Boolean expression *expr* has occurred or else it evaluates to false or zero. Meanwhile, *if (expr) p1 else p2* specifies that *p1* evaluates to true if Boolean expression *expr* has occurred or else *p2* evaluates to true.

Implication operator *s1 |-> s2* and *s1 |=> s2* respectively are overlapping and non-overlapping implication. The concept of implication is similar to *if()* statement [5]. As shown in Table 1, *s1_out |-> p1_out* were synthesized to a logic that produces output of *op_i* when input *p1_out* is followed by the occurrence of *s1_out* immediately. Property *p1_out* at the right-hand side (RHS) of the implication is called a consequence. It is a test condition for sequence *s1_out* as the antecedent at the left-hand side (LHS) to evaluate the operation to true. On the other hand, *s1_out |=> p1_out* start evaluation if *p1_out* happens at the next cycle later.

System function *$past* is a function that evaluates the previous expression value. *$past* returns true in the current clock cycle if the previous expression value is true in the previous clock cycle. For example, *$past(A)* is true if *A* has occurred in the previous cycle.

System function *$stable* is a function that evaluates the previous expression value and the current expression value. Previous expression value must be the same as the current expression value in order for *$stable* to return true. For example, *$stable (A)* is true when previous value of *A* is similar to *A*. As shown in Table 1, *$stable (A)* was synthesized into a logic that produces output *op_i* when *A*'s old value, *old_A* is the same as value of *A*.

C. Assertion Monitor Synthesis

Assertion monitor functions to store the result of assertion whenever the assertion is checked in the hardware. It allows user to retrieve the assertion results during or after the verification. We proposed to synthesize the assertion monitor into the collection module so that assertion results could be stored directly into hardware instead of being passed to the host computer that normally takes longer time.

1) Collection Module Design Process

Figure 2 shows the operation of collection module's operation. The proposed approach collected the assertion results from the assertion module through arbiter and stored them in a memory. Note that if there are *n* assertion outputs from the assertion module, each output has unique index *i* where $0 \leq i \leq n-1$ .The concept of arbitration was used to design the memory interface in the collection module because the occurrence of assertions is unpredicted and there maybe

multiple assertions occur at one time. The arbiter decides which assertion result has the highest priority and issues a grant signal accordingly. Thus, when an assertion output O_i is produced and at the same time, receives a grant signal in the current cycle, its index *i* which represents the identity of the granted assertion will be encoded and stored in the memory. The process is terminated after *n* assertions have occurred or the user interrupts to read the memory.

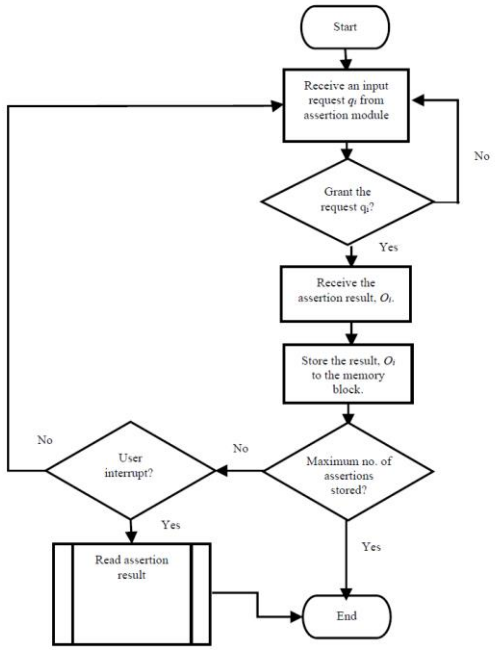


Figure 2: Operation of collection module

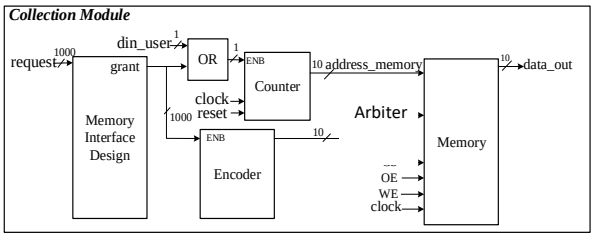


Figure 3: Proposed *N*-bits architecture of the memory design as collection module

Collection module consists of an arbiter, memory block and other components as shown in Figure 3. The illustrated design is able to receive and store at least 1000 assertion results. The concept of arbitration was used to design the memory interface in the collection module because the occurrence of assertions is unpredicted and there maybe multiple assertions occur at one time. The arbiter decides which assertion result has the highest priority and issues a grant signal accordingly. Counter is used to count the number of assertion results that have been stored and to generate memory address at which assertion results are stored. Encoder functions to encode grant signal of 1000 lines to 10 lines, which represents a 10-bit assertion location or identity namely *data_in* to represent assertion result. *data_out* sends out assertion result to another output peripheral of FPGA whenever the memory is read. OR gate functions to enable the counter when a grant signal is generated or when user intends to read the memory which can be enabled through *din_user*. *din_user* can interrupt the memory to read out the assertion result at *data_out* when necessary. In scan-chain approach, as discussed in the literature review, the assertion result is shifted out batch by batch or one by one after the completion of the verification process. Nevertheless, this proposed approach allows interruption or termination of the verification process anytime subject to user's decision on when to read the assertion results. For instance, user may wish to abort the verification after detection of a critical bug.

2) Oblivious Arbiter with Blocking Logic

We augment the oblivious arbiter with block logic to be used to arbitrate the assertion results. The blocking logic functions to block the same assertion from being stored again in the memory. By avoiding serving the same assertion, it minimizes the chance of failing to store an assertion result that occurs at the same time with other assertion checked for the first time.

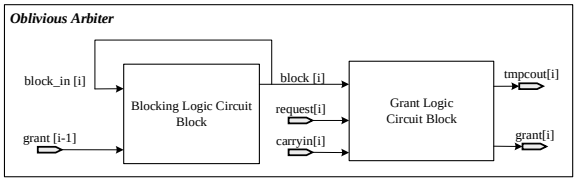


Figure 4: Slice circuit of the memory interface architecture; a) Oblivious arbiter architecture for slice *i*; b) Round-Robin arbiter architecture for slice *i*

Figure 4 illustrates the proposed arbiter architecture. It consists of blocking logic and grant logic. The function of each port and its internal signals are illustrated below:

- request: input to the arbiter, which receives signal from assertion module output.
- grant: output from the arbiter that indicates which assertion results to be stored.
- carryin: signal that acknowledges the arbiter to consider current request to be granted.
- tmpcout: signal that acknowledges the arbiter to consider next request to be granted.
- block: signal that acknowledges the block logic to block the request if it has been served previously.

The proposed blocking logic in Figure 4 is designed to block the *request* which has already been served from being served again in the next cycle onward. In other words, the assertion result is served once only to be kept to the memory after it happens. Initially, signal *block[i]* is set to high which allow any initial *request[i]* that issues a request to have a *grant* permission. When signal *grant[i]* goes high, the blocking logic produces low *block[i]* such that permission is not given to the same *request[i]* to have grant anymore. For example, when condition *grant[0]* is high, *block[0]* which is also high activates the blocking function. *block[0]* is then changed to low.

The grant logic behaves such that when there are many *requests* to the arbiter, they will be served according to the priority set and block signal. Only one request is granted in one cycle. Initially, the active-low input *block* is set to high to have the request granted based on the set priority. The first *carryin* in the circuit is set to high as an initial value, for example, the 4-bit request. When condition *request[3]* is issued, a grant is given since *block[3]* is initially set to high, which means no blocking. Otherwise, other *request[N-1]* can have their grant based on signal *block[N-1]*.

D. SVA Signal Extractor

In System Verilog language, it provides a bind feature that links internal nets or wires of DUV to the assertion module, as illustrated in Figure 5 a) for verification purpose. When the assertion module is implemented as hardware, these nets and wires need to be transformed into new outputs of DUV, which are also new inputs to the synthesized assertion module as illustrated in Figure 5 b). To automate the transformation, SVA signal extractor is developed to identify the required signals involved in the assertion statements using the Perl script. These signals could be inputs or internal wires of DUV, which connect DUV to the assertion modules. Besides, outputs of assertion modules that produce assertions result are also generated by SVA signal extractor and they are connected from assertion module to the collection module automatically.

Figure 5 c) illustrates the SVA signal extractor that first reads the Verilog files of the original DUV and the synthesized assertion module. Then, it creates new signals to replace internal signals linked by bind feature to connect DUV to assertion module. Besides, it generates collection module and necessary signals that connect assertion module to the collection module based on the number of assertion results derived by the synthesized assertion module.

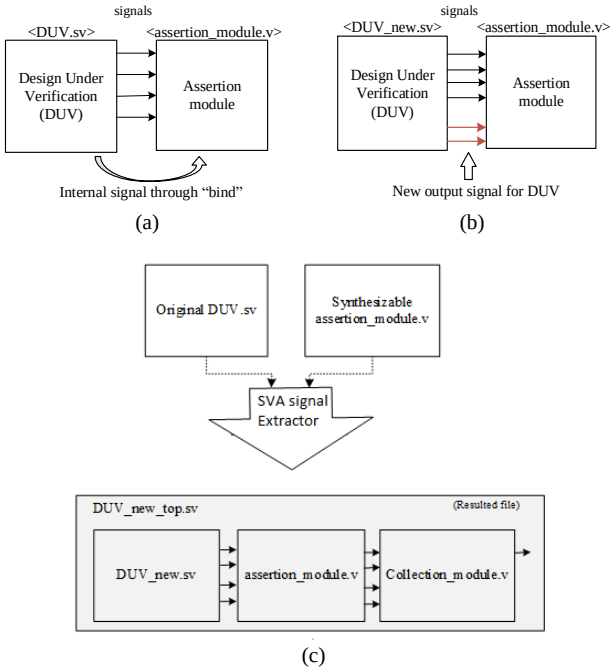


Figure 5: a) Internal signal through bind that connected DUV and assertion module, b) New output signal required for hardware verification, and c) Structure arrangement of the script directory

Figure 6 shows the first step of SVA signal extraction, which generates the signals that connect DUV to the synthesized assertion module. Firstly, the SVA signal extractor reads the DUV, assertion file and sub modules of DUV to identify common signals between DUV and assertion module. While identifying the common signals, the signal directions (input or output), ports, data type and internal nets and pins are extracted and kept in an array. Ports between DUV and assertion module are compared based on their names similarity to check whether a port is common to connect DUV to assertion module or not. If an input port of assertion module does not exist in the DUV, a new output port is created in the DUV. This is to convert the DUV's internal signals, which are originally connecting to the assertion module through bind feature, into DUV's output signal fed to assertion module. Then, the existing output ports and the new output port(s) are included in the output list of the new DUV. After the DUV file is processed to fulfill the signals requirement, the remaining unchanged content of the original DUV is copied as the content of the new DUV. During this process, conditional compiler directive such as *`ifdef, `else, `elseif, `endif,* and *`ifndef* in the original DUV file are removed and eliminated except *`include* directive line, because those syntaxes are non-synthesizable in the hardware.

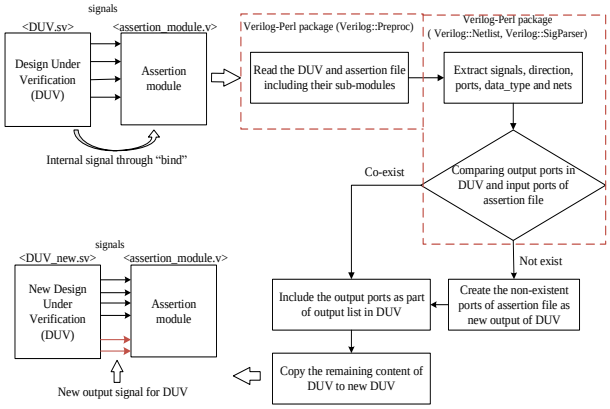


Figure 6: Flow chart of main script to produce new DUV file with new output port

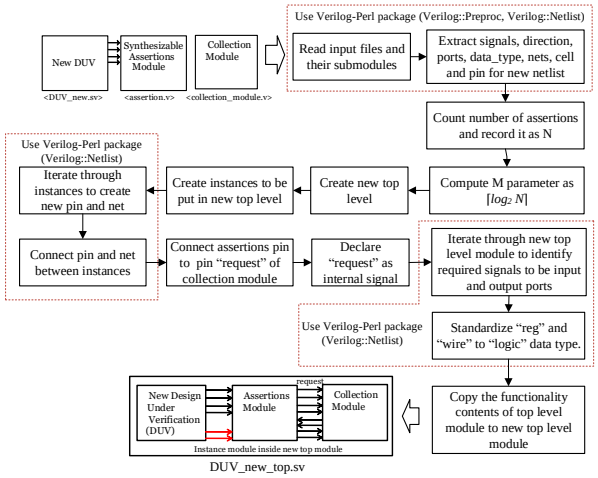


Figure 7: Flow chart of Script 2 to produce new top DUV with new instances

Figure 7 illustrates the second step of SVA signal extraction, which is generating the collection module and its relevant signals that connect assertion module with the collection module. The number of assertions is counted based on the keyword *output* in the assertion file. It will be recorded as *N*. Next, parameter *M* of collection module is computed as $\lceil \log_2 N \rceil$ where *M* is the number of address bits of the memory that is supposed to store the assertion results. Then, new top level Verilog file is created. This empty object will be filled with the signals extracted earlier by the first step and new parameters *M* and *N* besides instances of DUV, assertion module and collection module together with their nets and pins. In generating the collection module, signal *request* is assigned as a fixed

signal name which receives an assertion result signal to be fed to the arbiter in the collection module. Signal *request* is declared as an internal signal for the top level file. Finally, all the unchanged functionality content of the DUV and the assertion module are written to the new top level file. The new top level module file consists of new DUV, assertion and collection instances.

III. RESULT AND DISCUSSION

A. Implementation of FPGA-Assisted Verification Platform

Given a DUV and its verification file, SVA checker generator provides the assertion checkers whereas SVA signal extractor extracts signals required to connect DUV, assertion module and collection module. On top of that, necessary new inputs and outputs are added to the new top level Verilog file. To evaluate the proposed FPGA-assisted platform in the actual environment, the new top-level file that consist of new DUV, assertion module and collection module is implemented to FPGA. Then, the resource utilization is evaluated and discussed in the next section. Besides, we also demonstrate the verification platform using FPGA and SignalTap II analyzer.

1) Area Effectiveness of SVA Checker

The proposed SVA checker generator was experimented and tested using case studies below:

i. Up-down counter;
ii. Single FIFO; and
iii. Multiple FIFO.

FPGA area consumed by the assertion module resulted from our proposed SVA checker generator is compared with the results from MBAC for each case study. There are four assertions, namely ASR_1, ASR_2, ASR_3 and ASR_4 designed to verify the up-down counter, as described in Table 2. We derived the synthesized assertion checkers of these four assertions based on the concept of our proposed SVA checker generator and existing MBAC tool, respectively. For each assertion in Table 2, the first column describes the SVA coding for the assertion. The second column shows the synthesizable Verilog coding for the assertion which was generated by MBAC tool. *s[i]s* is the next state signal whereas the signal *s[i]sq* is the present state of the automata, which functions as an intermediate representation that stores bit representing sequence result, and *ASR_[k]* is the assertion signal or assertion result. The third column shows the synthesizable Verilog coding generated by the proposed SVA checker generator. However, for assertion that involves repetition such as *ASR_4*, the number of flip-flops in the SVA checkers generated by MBAC tool consists of at least *n* to keep track of the sequence that involves a repetition of *n*. This could be further optimized by using counter as repetition of *n* could be counted and tracked by counter consisting of $\lceil log_2 n \rceil$ flip-flops. Thus, the area consumed by verification-related hardware could be reduced, which is especially essential when the design is large. MBAC tool synthesizes *ASR_4* that involves repetition count of 10 using 10 flip-flops. It is different with our proposed SVA checker generator, which implements a 4-bit counter using four flip-flops for the same assertion.

Table 2
SVA and Synthesizable SVA in Verilog from MBAC Tool and SVA Checker Generator

Assertion in SVA	MBAC	Proposed SVA Checker Generator
ASR_1: assert property (@(posedge clk) (!en_ud && !en_load) |=> stable(cnt));	always @(posedge clk) s1<=cnt; assign s2 = s1 == cnt; always @(posedge clk) if (`MBACRPS reset) s3sq<=3'h4; else s3sq<=s3s; assign s3s={1'b1,((! en_ud) && (! en_load)), (s3sq[1] && !(s2))}; always @(posedge clk) if (`MBACRPS reset) ASR_1<=0; else ASR_1 <= (s3s[0]);	always@(posedge clk) old_A <= cnt; assign B = (cnt === old_A)? 1'b1 : 1'b0; always@(posedge clk) begin A <= !en_ud && !en_load; yy <= |A && |!B; end assign ASR_1 = !reset && yy;
ASR_2: assert property (@(posedge clk) (en_load) |-> ##1 (cnt == $past(load)));	always @(posedge clk) s4<=load; always @(posedge clk) if (`MBACRPS reset) s5sq<=3'h4; else s5sq<=s5s; assign s5s={1'b1, en_load, (s5sq[1] && !((cnt == s4)))}; always @(posedge clk) if (`MBACRPS reset) ASR_2<=0; else ASR_2 <= (s5s[0]);	assign temp_load = load ? load :1'b0; always @ (posedge clk) past_B<= temp_load; assign B = (cnt == past_B); always @ (posedge clk) begin A <= en_load; yy <= |A && |!B; end assign ASR_2 = yy && !reset;
ASR_3: assert property (@(posedge clk) (!en_load) |-> ##1 (!(cnt == ~$past(cnt) && cnt[width-1]==cnt[0]));	always@(posedgeclk) s6<=cnt; always@(posedge clk) if (`MBACRPS reset) s7sq<=3'h4; else s7sq<=s7s; assign s7s={1'b1,!(en_load),(s7sq[1] && ((cnt == (~s6)) && (cnt[width-1] == cnt[0])))}; always @(posedge clk) if (`MBACRPS reset) ASR_3<=0; else ASR_3 <= (s7s[0]);	always @ (posedge clk) begin A <= !en_load; yy <= |A && |!B; end always @ (posedge clk) past_B <= cnt; assign B = ((cnt == past_B)&& (cnt[7] == cnt[0])); assign ASR_3 = yy && !reset;

ASR_4:
 	
assert property (@(posedge clk) not (!en_load && !en_ud)[*10]);
	always @(posedge clk)
	if (‘MBACRPS reset)
	s8sq<=11’h400;
	else
	8sq<=s8s;
	assign s8s={1’b1,((!en_load)&&(! en_ud)), (s8sq[9] && ((! en_load) && (! en_ud))), (s8sq[8] && ((! en_load) && (! en_ud))), ... (s8sq[1] && ((! en_load) && (! en_ud)))};
	always @(posedge clk)
	if (‘MBACRPS reset)
	ASR_4<=0;
	else
	ASR_4 <= (s8s[0]);
		assign A = !en_load && !en_ud;
		always@(posedge clk)
		if (reset)
		counter <=0;
		else if (A)
		counter<=counter +1;
		assign ASR_4=(counter > 10 && A) ? 1'b1 : 1'b0;

Similar experiment has been conducted on single FIFO and multiple FIFO. Table 3 summarizes FPGA area utilized by synthesized assertions for all the case studies. ALMs

represent total logic elements, and FF represents the number of flip-flops in the FPGA. Assertion checkers produced by MBAC tool and the proposed method were compiled and simulated using Quartus II simulator targeted on Altera Arria V FPGA 5AGTFD7K3F40I3 board. The syntaxes were verified using VCS simulator and no error was found. Based on the results depicted in Table 3, using the proposed SVA checker generator, the logic utilization and total registers in the design has reduced on average by 12.1% and 23.4% respectively. The improvement was mainly due to the reduction of states in the SVA checkers that involves sequences of repetition and delay. The reduction of states is realized by the reduction of flip-flops to $\lceil log_2\ n \rceil$ flip-flops from n also leads to the greater improvement achieved in the reduction of total registers consumed in the FPGA (average 23.4%).

Table 3
Area Utilization of DUV with Assertion Module Resulted from MBAC Tool and the Proposed SVA Checker Generator

DUV	No. of Assertions	MBAC Tool		Proposed SVA Checker Generator		% Improvement	
		ALMs	FF	ALMs	FF	ALMs	FF
Up-Down Counter	4	15	31	14	26	6.7	16.1
Single FIFO	10	46	83	38	60	17.4	27.7
Multiple FIFO	32	349	269	306	198	12.3	26.4
					Average	12.1	23.4

The FIFO design used in this experiment is able to store 16 words, each of which is an 8-bit data. It has one reset and clock domain. The design was verified with ten assertions. According to Table 3, MBAC tool and SVA checker generator were used to generate the corresponding assertion checkers and their area utilization was evaluated. They were executed using Quartus II with targeted Altera Arria V FPGA 5AGTFD7K3F40I3 board. One of the ten assertions was using repetition count. It showed that 17.4% of ALMs improvement has been achieved compared to MBAC tool. Besides, the total register was 23 flip-flops less than MBAC tool.

Next, we created bigger circuit by duplicating the single FIFO with 10 assertions for three times. The circuit consists of three FIFOs with multiple clock and reset domain. Two additional assertions were added to top level design of multiple FIFO on top of the 10 assertions used for each single FIFO module to make up 32 assertions totally. The additional assertions are:

i. ERROR_FIFO_ALL_SHOULD_BE_FULL to check if all three FIFOs are full.
ii. ERROR_FIFO_ALL_SHOULD_BE_EMPTY to check if all three FIFOs are empty.

The area utilization for synthesized circuits of the 32 assertions using the proposed SVA checker generator consumed less logics and registers compared to MBAC tool by 12.3% improvement of ALMs. In other words, the proposed tool achieved an improvement in terms of area utilization for the checker's size.

Table 4 tabulates the total area utilization when the circuits are also augmented with the collection module to store assertion results besides the assertion module. We added bigger design named CAVLC with the assertion module and the collection module to be evaluated with different number of assertions as shown in Table 4, which are 100, 500, and 1000 assertions, respectively. CAVLC is an important module in MPEG4 that performs context-adaptive variable-length coding. We varied the number of assertions and observed that the area utilization was dominated by both the assertion module and collection module; the more the assertions used to verify the DUV, the larger the area consumed and the growing trend is linear to the number of assertions.

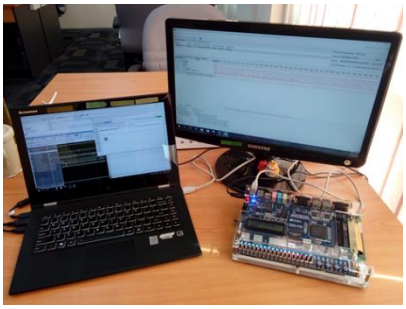
Table 4
Area Utilization the Proposed Verification Platform

DUV	No. of Assertions	*ALMs*	*FF*	*Pins*
Single FIFO	10	165	90	40
	13	182	102	40
Multiple FIFO	2	112	112	45
	29	466	265	105
	32	468	265	105
CAVLC	100	2,347	926	71
	500	4,985	1728	73
	1000	8,989	2729	74

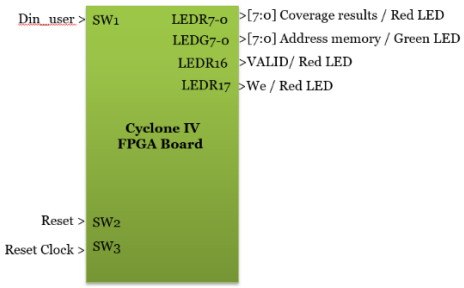
2) Verification Procedure

Figure 8a) shows the experimental setup used to demonstrate the verification procedure of the FPGA-assisted platform. It involves Ultrabook Lenovo with 64-bit operating system, monitor and FPGA device. The connection between Ultrabook and FPGA device were applied using USB blaster through JTAG programmer. The input and output peripherals were assigned as shown in Figure 8b):

- *Din_user*: a switch to interrupt the verification process to enable read mode to read coverage results from the collection module.
- *Reset*: switch to reset the registers. It is necessary before starting the evaluation.
- *Reset clock*: a switch to reset the system counter of the design that generates the memory address to store the coverage results.
- *Coverage results*: eight red LEDs to indicate 8-bit coverage results.
- *Address memory*: eight green LEDs to indicate 8-bit memory address of a location in the collection module, where the assertion result is stored.
- *VALID*: a red LED to indicate whether the output is valid or not.
- *We*: a red LED to indicate the write mode is enabled.



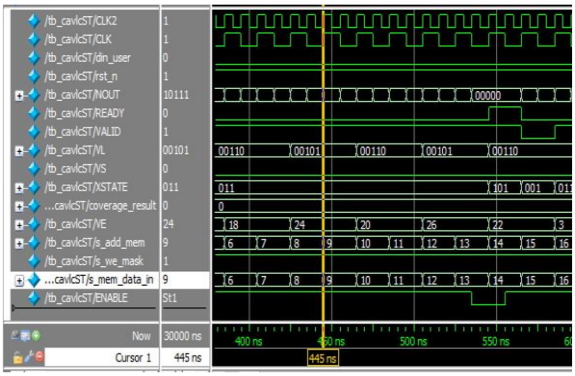
(a)



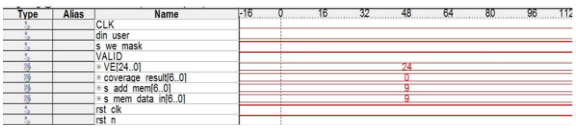
(b)

Figure 8: a) Experimental setup, and b) The structure of the input and output peripherals for CAVLC module

Initially, all the switches (sw1-sw3) were in off state to disable all the systems. Figure 9 shows the output signal value when the collection module is in write mode, where *din_user* is low, *we* is high and *coverage_result* is empty. The collection module stored the 9th coverage result of CAVLC design into the memory, while the *coverage_result* (the memory output) was empty at the given address since it was in write mode where the coverage result was being written. Hence, both captured waveforms from Figure 9(a) and Figure 9(b) agreed with each other.



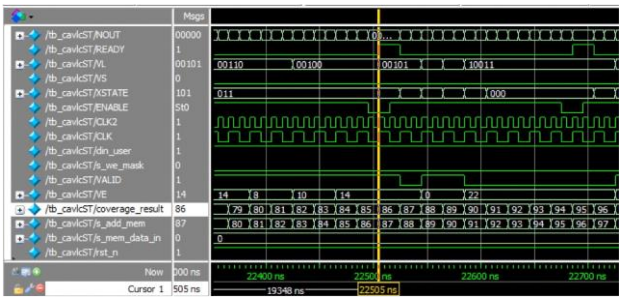
(a) Captured Modelsim waveform



(b) Captured SignalTap II analyzer waveform

Figure 9: Waveform comparison between simulation and real-time execution in hardware during write condition

For the second example, Figure 10 shows the waveform when collection module is in read mode where *din_user* is high, *we* is low and *coverage_result* is 86 in decimal. During the read mode, *coverage_result* was able to be read where the value of 86 corresponds to 86th coverage result captured from CAVLC design and stored at 86th memory address. Both captured waveforms in Figure 10(a) and (b) agreed with each other.



(a) Captured Modelsim waveform

(b) Captured SignalTap II analyzer waveform

Figure 10: Waveform comparison between simulation and real-time execution in hardware during read condition

IV. CONCLUSION

The proposed FPGA-assisted verification platform includes SVA checker generator, collection module and SVA signal extractor was successfully implemented. In addition, the proposed arbiter in the collection module selected the assertion result to be passed and stored to the memory. Implementing SVA signal extractor helps the user to identify the required signals between assertion module, collection module and DUV automatically. Experiment on FIFO and CAVLC have demonstrated that the proposed FPGA-assisted verification platform has been successfully realized.

ACKNOWLEDGMENT

This work is a joint research between Altera Corporation (Now is part of Intel as Intel Programmable Solution Group) and Malaysia-Japan International Institute of Technology, Malaysia (MJIIT) and supported by a research grant numbered 4B139 from Collaborative Research in Engineering, Science & Technology (CREST).

REFERENCES

[1] Foster HD. *Trends in Functional Verification : A 2014 Industry Study 2015*.

[2] Kuznik Ct, Mueller W, Le HM, Große D, Drechsler R. The System Verification Methodology for Advanced TLM Verification Categories and Subject Descriptors, 2011, pp. 313–322.

[3] Bamford N, Bangalore RK, Chapman E, Chavez H, Dasari R, Lin Y, et al. "Challenges in System on Chip Verification," *Seventh International Workshop on Microprocessor Test and Verification*, 2006, pp. 52–60.

[4] Boulâe M, Zilic Z. *Generating hardware assertion checkers*. Montreal: Springer Science + Business Media B.V, 2008.

[5] Foster H, Krolnik A, Lacey D. Assertion-based design. Boston: Kluwer Academic Publisher, 2005.

[6] Hutchison D, Mitchell JC. *Formal Methods for Hardware Verification*. Germany: Springer, 2006.

[7] Pierre L, Pancher F, Suescun R, Quévremont J, "On the effectiveness of assertion-based verification in an industrial context," *Proceedings of the 18th International Workshop on Formal Methods for Industrial Critical Systems*, Volume 8187, New York, NY, USA: Springer-Verlag New York, Inc, 2013.

[8] Abarbanel Y, Beer I, Gluhovsky L, Keidar S, "FoCs: automatic generation of simulation checkers from formal specifications," *Proc. 12th International Conference on computer aided verification*, 2000.

[9] Foster H, *Assertion-based verification: Industry myths to realities* (invited tutorial). Lecture Notes in Computer Science (Including

Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 2008, pp. 5–10.

[10] Bombieri N, Fummi F, Guarnieri V, Pravadelli G, Stefanni F, Ghasempouri T, et al, "On the reuse of RTL assertions," in *SystemC TLM verification*, 2014.

[11] Šimková M, Lengál O, Kajan M. HAVEN, *An open framework for FPGA-accelerated functional verification of hardware*. Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 2012, pp. 247–253.

[12] Boul M, Chenard J samuel, Zilic Z, *Assertion Checkers in Verification*, Silicon Debug and In-Field Diagnosis 2007.

[13] Lu Y, Zhu Y, *Assertion Synthesis : Enabling Assertion-Based Verification For Simulation*, Formal and Emulation Flows, pp. 1–7.

[14] Das S, Mohanty R, Dasgupta P, Chakrabarti PP, "Synthesis of system verilog assertions," *Proceedings-Design, Automation and Test in Europe*, DATE 2006, pp. 70–75.

[15] Chuang C. Lung, Liu C Nan J, "Hybrid Testbench Acceleration for Reducing Communication Overhead," 2011, pp. 40–51.

[16] Boul M, Zilic Z, "Incorporating efficient assertion checkers into hardware emulation," *Proceedings of the 2005 International Conference on Computer Design (ICCD'05)*, 2005.

[17] Koczor A, Matoga L, Penkala P, Pawlak A, "Verification approach based on emulation technology," *Proceedings of the 2016 IEEE 19th International Symposium on Design and Diagnostics of Electronic Circuits and Systems*, 2016.

[18] Tomas BJ, Jiang Y, Yang M, "SoC Scan-Chain verification utilizing FPGA-based emulation platform and SCE-MI interface," *International System on Chip Conference 2014*, pp. 398–403.

[19] Tong JG, Boulé M, Zilic Z, "Test compaction techniques for assertion-based test generation," *ACM Transactions on Design Automation of Electronic Systems*, 2013.

[20] Bailey B, Martin G, Arderson T, *Taxonomies for the Development and Verification of Digital Systems*, United States: Springer Science + Business Media B.V, 2005.

[21] Li Y, Wu W, Hou L, Cheng H, "A study on the assertion-based verification of digital IC," *International Conference on Information and Computing Science*, 2009, pp. 25–28.

[22] Boulé M, Zilic Z, "Automata-based assertion-checker synthesis of PSL properties," *ACM Transactions on Design Automation of Electronic Systems*, 2008.

[23] Pellauer M, Lis M, Baltus D, Nikhil R, *Synthesis of Synchronous Assertions with Guarded Atomic Actions*, 2005.

[24] Sonny AT, *OVL, PSL, SVA : Assertion Based Verification Using Checkers and Standard Assertion Languages*.

[25] Kumar Tala D. World of Asic 2014. http://www.asic-world.com/systemverilog/operators.html.