SHiLA: Synthesizing High-Level Assertions for High-Speed Validation of High-Level Designs

Mohammad Riazati
Mälardalen University
Västerås, Sweden
mohammad.riazati@mdh.se

Masoud Daneshtalab
Mälardalen University
Västerås, Sweden
masoud.daneshtalab@mdh.se

Mikael Sjödin
Mälardalen University
Västerås, Sweden
mikael.sjodin@mdh.se

Björn Lisper
Mälardalen University
Västerås, Sweden
bjorn.lisper@mdh.se

***Abstract*—In the past, assertions were mostly used to validate the system through the design and simulation process. Later, a new method known as assertion synthesis was introduced, which enabled the designers to use the assertions for high-speed hardware emulation and safety and reliability insurance after tape-out. Although the synthesis of the assertions at the register transfer level is proposed and implemented in several works, none of them can be adopted for high-level assertions. In this paper, we propose the SHiLA framework and a detailed implementation guide by which assertion synthesis can also be applied to the high-level design processes. The proposed method, which is fully tool independent, is not only an enabler to high-speed assertion-assisted simulation but can also be used in other scenarios that need assertion synthesis, as it has the minimum possible effect on the main design's performance.**

Keywords— Assertion, Synthesis, HLS, High-Level Design, High-Speed Validation

I. Introduction

High-Level Synthesis (HLS) enables designers to develop the design faster and easier in higher-level languages such as C. Design and verification are two main steps before any digital product becomes available to be sent to the market. Researchers and engineers have been continuously trying to facilitate and expedite this process. On the design side, the most recent and significant step was the introduction of the high-level synthesis, which enabled the designers to implement their desired system in a more natural and perceivable manner. Although HLS emerged as a remarkable advancement in digital design technology, on the verification side, there has been no significant achievement to accompany this design level. One of the main verification methods in the literature and industry is Assertion-Based Verification (ABV) [1].

Fortunately, assertion statements are already supported by the ANSI C language standard [2]. The two most common practices of using assertions are using them as pre-conditions or post-conditions. Pre-conditions and post-conditions are used to specify and check the assumptions or conditions before and after a region of a program, respectively. These usages not only expedite the simulation and verification process but also improve the readability and understandability of the program. More precisely, assert is a macro usually defined in the assert.h header file.

The C/C++ assert statement's implementation in assert.h is in a way that it is executed only during the debug time. When the release version is being built, the assert statement is eliminated during the preprocessing phase. A similar scenario happens when the C language is used for high-level modeling of the digital circuits. When the designer is simulating (and testing) the design using a testbench, assertions are active and generate corresponding messages whenever a fault, incorrect action, unexpected behavior, undesired input or output, etc. is detected. However, during the synthesis process, assertions are ignored (or may cause synthesis failure as a result of being a non-synthesizable construct, if they are not eliminated during the pre-processing step).

Assertions were basically intended to be used as software internal checkpoints. But, transforming them to the hardware checkers and putting them along with the main design body will be valuable in several scenarios described as follows:

A) High-speed simulation: in this scenario which is also known as emulation or hardware-assisted simulation, instead of simulating the design on the computers, alternatively, the design is synthesized to the FPGA and input vectors and sequences are applied to the hardware. By monitoring the outputs, the designer can examine the functional correctness of the design. In this case, if the assertions are also synthesized to the FPGA, the observability of the design's internal behavior as well as the ability to cover the corner cases significantly increases. This method also has a side benefit, which is known as synthesis result assurance. If assertions are synthesized to the hardware, especially if they are used as the pre-conditions of a specific code region, they will help the designers not only with detecting the possible inequality between pre-synthesis and post-synthesis versions but also with pinpointing its origin.

B) Safety assurance: in safety-critical applications, like avionics systems, a slight unpredicted or unexpected behavior (such as bit-flips) may result in disastrous consequences, including loss of human life or financial damage. The inclusion of the assertions in the final chip will help the chip environment to realize the chip malfunction. This can start several countermeasures, e.g., deactivation of the current module and activation of the back-up one.

C) Security: a design can be equipped with assertions in order to detect various attacks such as control-flow and data-injection attacks. In case such assertions fire when the generated hardware is performing the intended tasks, this firing signal can be utilized either to stop execution or to take a countermeasure.

In this paper, we mainly focus on high-speed simulation. Nonetheless, our method is applicable to any safety/security mechanism that is capable of using assertions.

As mentioned earlier, ANSI C assert statements are not synthesized by the traditional synthesis tools, unless a specific method is adopted. As we will discuss in the related work section, the previously proposed methods significantly affect the performance of the design, rely on a specific synthesis tool, or are applicable only to a subset of C designs.

As our key contribution, we propose a complete framework, which instead of letting the assert statements be eliminated by the synthesis tool, converts them to synthesizable constructs to accompany the generated hardware without affecting the performance of the design. We implemented a code analyzer and manipulator to extract and eliminate the assertions, inject the required auxiliary codes and automatically generate the synthesizable assertion modules. The main advantages of this work are threefold as follows:

- The proposed mechanism is entirely tool-agnostic so that it can be used along with any existing synthesis tools, either open-source or commercial.
- The assertion modules are generated such that their effect on the design's timing and performance is minimal.
- Assertions are implemented as separate modules so that any future improvements like optimization or consolidation is easily possible for future researchers or tool producers.

The remainder of this paper is organized as follows. In Section 2, previous work on assertion synthesis is reviewed. Then in Section 3, the overall structure and detailed implementation of the proposed mechanism is explained. Experimental results are given in Section 4. Some technicalities are discussed in Section 5, and finally, the paper concludes in Section 6.

II. Related Work

There are a lot of research works on the synthesis of the assertions. We categorize them as the synthesis of RTL assertions, and the synthesis of high-level assertions. A large body of works has focused on the synthesis of the RTL assertions, where some recent works are provided here. However, only a few papers deal with the high-level assertion synthesis, and we try to cover all of them.

A. Synthesis of assertions at the register transfer level

A lot of assertion languages and libraries are used along with the RTL design. OVL, SVA, PSL, and VHDL assert statements are the most covered ones in the literature. There are also some less prevalent ones like OVA, Intel ForSpec, IBM Sugar, Motorola CBV.

The synthesis of RTL assertions has been the topic of various research works, and we will address some of the latest ones here. Morin-Allory et al. [3] developed a prototype tool called SyntHorus2 that generates a synthesizable RTL design from the input PSL specifications. Wenzl et al. in [4] also provided a tool for conversion from PSL to synthesizable VHDL/Verilog code as output. In [5] the architecture of a System Verilog Assertions (SVA) synthesis compiler is presented, which converts SVA assertions to equivalent synthesizable Verilog code. Taatizadeh and Nicolici in [6], presented an emulation framework in which the assertions are synthesized using the existing assertion synthesis tools for PSL and SVA; the framework is used to detect bit-flips during post-silicon validation.

B. *Synthesis of the assertions in high-level designs*

The simplest and most straightforward way to synthesize the assertions is that the assertions are just removed from the source design, and after synthesizing the assertion-free design, assertions are appended to the generated RTL. Although this method seems natural to apply, it has several disadvantages. The first problem is that the designer should find the corresponding signal for each of the variables and objects used as the input expression of the assert statement in the high-level source in the generated register-level source; usually, the variable and signal names in the post-synthesis RTL code is different from their names in the high-level design. It is even likely that the HLS eliminates the signal during the optimization phases. The second point is that whenever the designer alters the high-level design, this step should be repeated manually. Moreover, any changes in the HLS system, such as utilizing another HLS from another vendor or even any updating by the same vendor, requires the user to repeat this process.

There are only a few works on the synthesis of high-level assertions. The first paper was published in 2010 by Curreri et al. [7] in which the C assert statement was converted to an IF statement. The condition of the IF statement was the complement of the assert condition. Authors admitted that this conversion results in significant performance downgrade and hardware overhead. The source of this problem is that the IF statement is still in the original code, which affects the synthesis process, and more precisely, the scheduling and binding steps of the HLS. The proposed method in [8] is aimed to work with Impulse C, which is a proprietary subset of the C programming language with parallel programming support. The tool, which is not available anymore at the time of writing this paper, only supported that version of the C language (not the standard ANSI C). In [9], another synthesis method for high-level assertions is presented. This method reads and manipulates the Control Data Flow Graph (CDFG) of the design. It identifies and tags all assertion branches in the graph and extracts them. Afterward, the result VHDL/Verilog modules are generated. As the proposed method requires access to the HLS source, it cannot be employed in commercial HLS tools. Besides, the assertions are still embedded in the CDFG, and thus, the performance of the generated hardware is affected. Authors in [10] defined extensions to the Java language to support assertions. Then, they defined some APIs to send the result of the assertion alerts from the hardware part to the software part. The extended language is only supported by Maxeler [11], and the generic and standard synthesis tools cannot synthesize the design. Besides, the method is only applicable to the designs that a software part accompanies the hardware. In [12], HLS verification was studied, but no methodology or practical assertion synthesis mechanism is proposed.

As discussed in this section, none of the existing assertion synthesis methods has all the following features:

- Applicable to the standard C language (ANSI C)
- Being tool independent so that a designer is not limited to a specific (likely a proprietary) tool.
- Design's critical path and performance is not affected.

In this paper, we are proposing an automatic process that has all the above features together.

III. SHiLA FLOW

Our proposed high-level assertion synthesis method begins by receiving the High-Level Design (HLD) in C/C++ language, in which the assert statements are embedded. We assume that although the assert statement is not a synthesizable construct, the expression used as the assertion parameter is synthesizable. Non-synthesizable expressions may comprise dynamic memory allocation function calls or pointers that cannot be resolved statically. The overall flow of SHiLA is illustrated in Fig. 1. The input HLD is then analyzed to find the assert statements. During the analysis and manipulation stage, assert statements are extracted from the code. The output of this stage will be two separate sets of HLD source files. The first one is the modified design, in which no assert statement exists, and some additional statements and auxiliary control signals and ports are implanted in the design. The second set of modules contains the high-level synthesizable version of the assertions. For each assert statement in the HLD, an assertion module is generated. This stage is later explained in detail. Assuming that the input HLD (ignoring the assert statements) is synthesizable, the two outputs of this step are synthesizable HLDs. As a result, they can be fed into a high-level synthesis tool. The flow is entirely tool-agnostic, which enables the users to utilize every existing high-level synthesis tool. By synthesizing the generated modules, one assertion-free IP for the main design and a set of synthesizable assertion IPs are created. These IPs are then all connected to each other and synthesized.

As stated earlier, in the code analysis and manipulation stage, hereafter referred to as pre-processing, the input HLD is processed, and one assertion-free HLD and several assertion modules are generated. Fig. 2. shows an overview of the process in this step. As shown, we assume the input HLD consists of three significant elements: ports (shown in blue), several C code sections (shown in black), and several assert statements scattered throughout the design (shown in green). Except for the C code sections, which are kept untouched, the two other parts should be analyzed and manipulated during the pre-processing phase. In the rest of this section, we will cover the implementation details of the pre-processing stage.

A. *Creating assertion modules*

Assertion modules are designed in the ANSI C language. Fig. 3 shows a pseudo-code of an assertion implementation.

The "enable" input determines if the assertion module should function at a specific time or not. An assert statement is used at a particular point of the design. It is necessary to inform the assertion module that the execution of the main design has reached that particular point, and the required data are ready and valid to be collected and evaluated. It is indicated by a Boolean (one bit) input.

The validity of the assertion outputs is indicated by the one-bit ready signal. At the beginning of the assertion execution, the value of zero (false) is assigned to this output. It remains false until the execution has finished, and the result becomes ready.

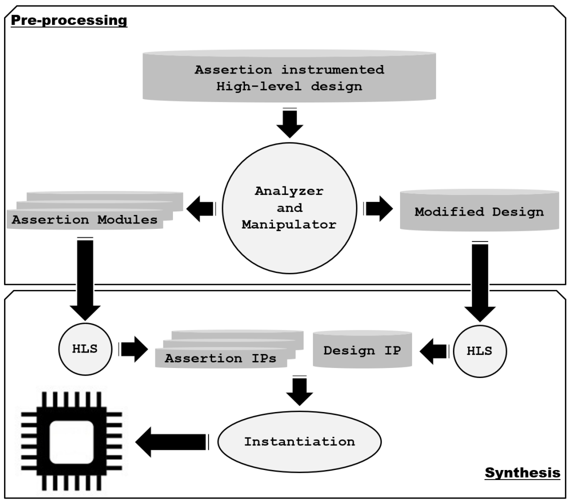

Fig. 1. SHiLA flow from HLD to Device

As high-level designs are described through sequential bodies, once the assertion is activated, the program continues running while the assertion is running in parallel. Execution of the next statements in the main design's body may affect the variables' values that are used by the assertion. For this reason, the assertion module should retain the values at the very first stage of its execution. To guarantee that, a local copy of all the variables is created as the opening statements in the assert module by defining local correspondences for each of the input variables and assigning the inputs to these new local variables. It is worth noting that as there is no dependency between these assignment statements, all of them are executed in parallel and in one clock cycle after synthesis.

The result of the validation is assigned to the output. The value of zero (false) indicates that no error has been detected. Contrarily, the value of one (true) means that checking the inputs based on the determined correctness conditions has failed. Note again that this value is valid only if the ready signal is true.

B. *Assertion firing*

After checking the correctness of the assertion expression, the assertion module generates the firing result. It will be false if the assertion input expression is evaluated as correct and will be true otherwise. The design validators can use this signal in hardware-assisted simulation environments and safety or security applications may use it to start counter actions such as restarting the system, blocking access to the system, etc.

A firing signal is generated for each of the assertions. However, it should be considered that for designs with lots of assertions (and firing signals consequently) due to a possible shortage of output ports of the target device, the user may opt to adopt an encoder circuit, or even an OR gate if the user is only concerned about the failure of at least one assertion. Another useful method, which is slower, needs fewer ports, and meanwhile, provides the exact firing location, is utilizing a scan chain to send out the assertion firing results sequentially.

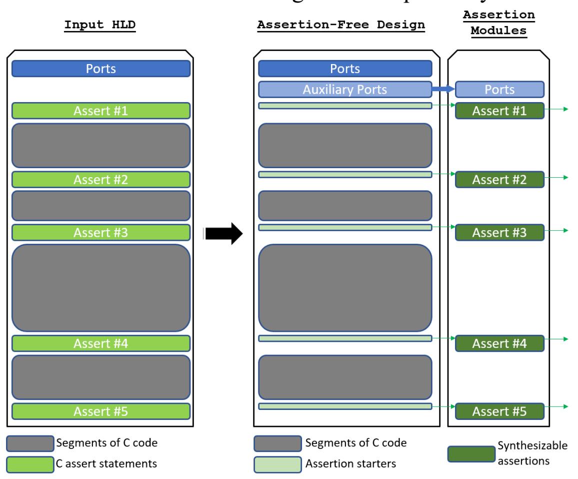

Fig. 2. code analysis and manipulation stage of SHiLA

```
function assert_id
  inputs: bool enable, input_data_list
  outputs: bool ready, bool fired

  create local copies of input_data_list;
  if (!enable) do nothing;
  else
    ready = false;
    if (condition)fired = false; //No Firing
    else fired = true; //Firing
    ready = true;
```

Fig. 3. High-level synthesizable version assertion

C. *Auxiliary ports*

Assertion modules and the high-level design are implemented as two distinct IPs. Therefore, all the variables used in the assertion input expression should be defined as the main design's ports. As illustrated in Fig. 2, they are all connected to the inputs of the assertion modules. These ports will not use any of the physical I/O ports as they are just internal connections when two IPs (main design and assertion modules) are connected to each other. Another consequence of adding these signals as the design's interface ports is to prevent the HLS from eliminating them during the optimization stages.

Another category of the auxiliary ports is the assertion starters. As explained before, high-level assertions are valid only in a specific zone. Therefore, when the execution of the design reaches the validity point of one specific assertion, that assertion module should be notified. Notification is made through the enable port of the assertion modules, which were previously explained. For each assertion in the HLD, a single-bit output is added to the design. Every assertion in the HLD is replaced with an assertion starter, which assigns the value of true to this signal to activate that specific assertion.

D. *Integration and synthesis*

Finally, both modified HLD and the assertion modules are synthesizable. With the aid of the auxiliary ports, modules can be connected to each other and synthesized to the target device. The target device should be selected according to the required size of the whole design. A thorough discussion on this issue is later explained in Section V.

IV. Experimental Results

To confirm the performance of the proposed method, we provide two sets of results. First, with an example, we will explain how the insertion of the synthesizable version of an assertion, directly in the high-level design, will downgrade the performance of the design. We will show that this will result in a much longer execution in terms of the number of the required clock cycles to finish the execution of a slice of code. In the second experiment, we apply our method to two well-known benchmark sets and show how dramatically the simulation speed increases. Note that this experiment is conducted to demonstrate the effectiveness of the high-speed simulation technique, which we listed in the introduction section of this paper. Other usages can be achieved by the implementation of fault-tolerant or security mechanisms along with the introduced assertion synthesis mechanism, which is out of the scope of this paper.

To produce the experimental results, we chose a list of high-level designs from the WCET benchmarks [13] and CHStone benchmarks [14]. WCET benchmarks are set of C designs with various levels of complexity intended to analyze the worst-case execution time. CHStone benchmarks, on the other side, are mainly intended to evaluate high-level synthesis tools.

A. *Comparison with related work*

In this section, with a brief example, we will demonstrate how the insertion of the synthesizable version of the assertions directly into the design may downgrade the performance of the design. This method was proposed in [7]. It should be noted that although we introduced more methods in the related work section to synthesize the high-level assertions, as explained, [7] is the only one that is comparable to our method.

In this experiment, as an example, we used the Prime benchmark from the WCET benchmark set. We first synthesized the high-level design to a Xilinx Artix-7 FPGA using Xilinx Vivado HLS tool to obtain the minimum clock period for the synthesized design. Vivado HLS generates an RTL design in VHDL language. Using the ModelSim simulator, we simulated the generated VHDL code and realized how many clock cycles are required to finish the execution of the code. By multiplication of the clock period by the number of clocks, we obtain the total wall-clock time needed to execute the code. Then, we implemented a synthesizable version of an assertion. The assertion was intended to make sure that a specific value is not supplied as a parameter. We added the assertion inline as an IF statement as proposed in [7]. We followed the same approach for this assertion-equipped design to find the latency. Latency, or the execution time, is the number of the required clock cycles multiplied by the period of each clock. Table I shows the results. The clock period is similar in both versions as the critical paths are the same. The only difference is that the number of execution clock cycles spiked from 302 to 438, which is nearly 45% overhead in the latency. It is worth noting that this increase is a result of instrumenting the design with only one single assertion. Adding and invoking more assertions in this way can significantly deteriorate the performance.

Then, we applied our method and implemented the assertion as a separate module. Again, the same steps were taken. Again, the clock period is the same, but the number of execution clock cycles is much less than the inline method, although a few clock cycles more than the original design due to some communication overheads.

The area overhead of both methods is also shown in table I to compare two methods. Even though the execution time of our method is significantly shorter, the area overhead of our method is slightly higher, as some logic sharing by the synthesis tools has not been possible in our method. Note that in this paper, we do not cope with the area overhead of the assertions. The designer is responsible for selecting the assertions while considering the importance and coverage of them. We have covered some aspects of the area overhead considerations in the discussion section.

TABLE I. Performance drop by synthesizing assertions inline

Design	Clock Period (ns)	Execution time (Clock cycles)	Execution time (ns)	Latency overhead	Area overhead (LUT/FF)
Original	8.47	302	2558	-	- / -
Assertion equipped [7]	8.47	438	3710	45.0%	13.7% / 3.21%
Assertion equipped [Proposed method]	8.47	306	2592	1.3%	14.3% / 3.43%

B. *High-speed simulation speed up results*

In the second round of experiments, we will explain how our proposed method will speed up the validation process. The following steps were taken to illustrate the effectiveness of the method:

1) Obtaining the normal execution time of the high-level code: In this step, we compiled and executed the assertion equipped design in Microsoft Visual C++ with sample inputs and obtained the total execution time. The results were collected on an Intel Core i5 1.6 GHz. To obtain a more realistic result for large designs with long testbenches, we had to extend the testcases. It should be noted that hardware-assisted simulation in general, and our proposed method in particular, is only useful for the designs with a long testing time.
2) Applying the method: We applied the method according to Section III. The output of this step was the synthesizable HLD accompanied by the assertion module. The whole system is synthesizable by any generic HLS tool as the designs were all synthesizable high-level designs and the assertion modules were also implemented in a synthesizable manner.
3) High-level synthesis of the system modules: We synthesized the whole system using the Xilinx Vivado HLS tool. It determined the minimum clock period required for the synthesized design on the FPGA. It will also generate the VHDL version of the design, which can be used in RTL simulation tools.
4) Simulation of the synthesized design: Here, we applied the same inputs as step one to the design and simulated the generated RTL design using the ModelSim simulation tool. The result of this step is the number of clock cycles required to finish the execution. By multiplication of the clock-cycle count by the clock period obtained from step three, we calculated the wall-clock time for the whole execution.

As shown in table II, the simulation time for the synthesized version of assertion-instrumented designs is considerably lower than the simulation time of the high-level design on a computer. For the Prime benchmark, which shows the least speed-up, the execution speed increased more than 100 times. For other cases, the speedup is almost between 200 and 500 times. The "lcdnum" benchmarks show a very large speed-up as high as almost 5000. Actually, this a very rare case that the design has been synthesized in a way that the execution takes only one clock cycle. In fact, this design was synthesized as a pure combinational circuit. Another important point that is worth being considered in this table is that there is no meaningful relation between the execution time on the PC and the achieved speed-up through hardware-assisted simulation.

V. Discussion

There are some notes to be considered for this method, which in this section, we try to cover.

Applying this method to a design certainly needs an extra area on the silicon. In the case of size limitation (especially if the assertions are synthesized and sent to the tape-out for a scenario other than hardware-assisted simulation), the user should prioritize and select the most important ones. Adding even a single assertion to the final chip that is being sent for the tape-out usually, not always, will result in area overhead. It is the cost of safety or security. However, it should be noted that in some specific cases, there is no area overhead cost. As an example, if you consider the number of logic cells of Xilinx Virtex-7 series FPGAs, they are as follows: 326,400; 412,160; 485,760; 554,240; 693,120; 979,200; 1,139,200; As can be seen, the number of logic cells increases by some sort of granularity. It means that if the original design does not fit into one FPGA and the designer decides to select a larger FPGA from the list, usually there are some unused logic cells (and Flip-Flops, etc.). These unused logic cells can be used for safety/security assurance through the assertion synthesis.

The second point to be noted is that although the method is defined and tested for ANSI C assertions, a similar methodology can be applied to sequential assertions in VHDL and SVA, though with minor modifications.

Thirdly, if the high-level design consists of a hierarchy of the functions, the local variables of the functions which are used in the assertions inside that specific function, should also be defined as the auxiliary ports of the design. It can be achieved by adding these variables as the outputs of that functions and then the same action in the calling function, and so on. It should be done in a bubbling manner.

VI. Conclusion

In this paper, we proposed a methodology that enables the designers to synthesize the assertions in high-level designs without affecting their performance. It is helpful in various scenarios such as high-speed or hardware-assisted simulation and safety and security assurance. The experimental results showed that how the proposed method is more effective than merely embedding the assertions inline with the design and how this technique can significantly reduce the simulation time.

TABLE II. Execution results on PC vs. hardware-assisted simulation using SHiLA

Design	Execution Time (s) (on PC)	Clock Period (ns)	Execution Time (Number of Clock Cycle)	Execution Time (ns) (Hardware-Assisted)	Speed-up
prime	290	8.47	3,019,681,304	25,576,700,645	11.3
janne_complex	79	6.51	457,851,870	2,980,615,674	26.5
lcdnum	22	4.434	9,464,655	41,966,280	524.3
fibcall	98	7.88	328,463,091	2,588,289,157	37.9
sqrt	231	8.451	1,124,626,467	9,504,218,273	24.3
adpcm-encode	597	8.555	2,004,786,131	17,150,945,351	34.8
adpcm-decode	488	8.47	1,189,895,355	10,078,413,657	48.4

References

[1] Z. Ren and H. Al-Asaad, "Overview of Assertion-Based Verification and its Applications."

[2] Programming languages - C, 9899:1999, ISO/IEC.

[3] K. Morin-Allory, F. N. Javaheri, and D. Borrione, "Efficient and correct by construction assertion-based synthesis," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 23, no. 12, pp. 2890-2901, 2015.

[4] M. Wenzl, C. Fibich, P. Rössler, H. Taucher, and M. Matschnig, "Logic synthesis of assertions for saftey-critical applications," in 2015 IEEE International Conference on Industrial Technology (ICIT), 2015: IEEE, pp. 1581-1586.

[5] O. Amin, Y. Ramzy, O. Ibrahem, A. Fouad, K. Mohamed, and M. Abdelsalam, "System Verilog Assertions Synthesis Based Compiler," in 2016 17th International Workshop on Microprocessor and SOC Test and Verification (MTV), 2016: IEEE, pp. 65-70.

[6] P. Taatizadeh and N. Nicolici, "Emulation infrastructure for the evaluation of hardware assertions for post-silicon validation," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 25, no. 6, pp. 1866-1880, 2017.

[7] J. Curreri, G. Stitt, and A. D. George, "High-level synthesis techniques for in-circuit assertion-based verification," in 2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010: IEEE, pp. 1-8.

[8] "Impulce C." https://en.wikipedia.org/wiki/Impulse_C (accessed 18 June, 2019).

[9] A. Ribon, B. Le Gal, C. Jégo, and D. Dallet, "Assertion support in high-level synthesis design flow," in FDL 2011 Proceedings, 2011: IEEE, pp. 1-8.

[10] T. Todman and W. Luk, "In-Circuit Assertions and Exceptions for Reconfigurable Hardware Design," in Provably Correct Systems: Springer, 2017, pp. 265-281.

[11] "MaxCompiler." https://www.maxeler.com/products/software/maxcompiler/ (accessed 18 June, 2019).

[12] S. U. Park, T. P. Kim, M. Z. Lee, and Y. B. Cho, "Method of RTL Debugging When Using HLS for HW Design: Different Simulation Result of Verilog & VHDL," in 2018 International SoC Design Conference (ISOCC), 2018: IEEE, pp. 273-274.

[13] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The Mälardalen WCET benchmarks: Past, present and future," in 10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010), 2010: Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[14] Y. Hara, H. Tomiyama, S. Honda, and H. Takada, "Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis," Journal of Information Processing, vol. 17, pp. 242-254, 2009.