社団法人 情報処理学会 研究報告
IPSJ SIG Technical Reports

2008－SLDM－135（2）
2008／5／8

Checker Circuit Generation for SystemVerilog Assertions in Prototyping Verification

Mengru WANG† and Shinji KIMURA†

† Graduate School of Information Production and Systems, Waseda University 2-7 Hibikino, Wakamatsu, Kitakyushu, Fukuoka 808-0135 Japan

E-mail: mengru@ruri.waseda.jp

Abstract: Reduction of verification period is the crucial problem in the recent LSI designs, and prototyping/emulation technologies are used for the reduction. Assertion-Based Verification (ABV) has been paid attention to check design errors at run time in simulation, and it has become an important to combine ABV with the prototyping. In the manuscript, we discuss about a generation method of checker circuit for SystemVerilog Assertions (SVA's). SVA is one of standard method to describe assertions in ABV. In the checker circuit generation, we focus on the hardware cost reduction.

Keyword *Assertion-Based Verification, SystemVerilog Assertion*

1. INTRODUCTION

As the advances in LSI process technologies, we can integrate tons of transistors in one chip, but at the same time we suffer from the design verification of such circuits. Assertion-Based Verification (ABV) is recognized as one of effective pre-silicon validation methods especially for the protocol verification and the design reuse.

In AVB, assertions showing the correct circuit behavior are described and their satisfaction is checked in the simulation. There have been proposed several assertion description methods: Forspec (Intel) [6], Sugar/PSL (IBM/Accelera) [7], OVA (Synopsys), etc. Recently, SystemVerilog Asserion was proposed as a part of SystermVerilog language, and has been accepted as a standard of the assertion description language.

Basically, assertion descriptions are to check the correctness of the circuits' behavior in simulation, the synthesizability of them is not considered. However with the recent progress of Field Programmable Gate Arrays (FPGA's), the check of assertions under prototyping environment becomes an important issue. A basic strategy of SVA synthesis has been proposed in [1], in which SVA expressions are handled in 4 groups respectively, Simple Sequence Expression (SSE), Interval Sequence Expression (ISE), Complex Sequence Expression (CSE) and Unbounded Sequence Expression (USE) and checker circuits are built based on 3 sequence expression blocks, namely, SE block, Delay block and Interval delay block. In [2], some typical checker circuits have been described, such as checker circuits for clock cycle delays and implication operators. These proposed methods only discuss SVA checker circuit generation for each kind of SVA syntax separately. Hardware cost issue has not been addressed.

The checker circuits generated based on SVA also consume hardware resources on FPGA during emulation, for large designs, the checker circuits might dissipate such a large amount of hardware that it is even impossible to build the prototype of the design. In this paper, we discuss the hardware cost of SVA checker circuit generation in a holistic way and show a possibility of optimizing checker circuits in order to reduce the hardware cost of SVA synthesis by sharing hardware among similar checker circuits. We also propose a new checker circuit for clock cycle delay in SVA by using counter circuits instead of register arrays, thus the hardware cost can be further reduced.

The rest of the paper is organized as follows: Section 2 shows the basic syntax of System Verilog Assertions and previous methods of generating checker circuits. Section 3 describes the basic idea of a counter-based generating method for delay operation and also a hardware reduction method by sharing hardware of checker circuits. Section 4 gives some experimental results to show the efficiency of the proposed methods. Section 5 gives the conclusion and some further work.

2. PRELIMINARIES

2.1. Assertion Based Verification

An assertion is an expression that is capable to indicate certain behavior. Assertions are usually used to debug by finding can't happen errors. When used in hardware description language (HDL) designs, an assertion checks for specific behavior and displays a message if it occurs. Assertions are generally used as monitors checking for bad/good behavior. For our purposes, an assertion is a statement about a specific function or property that is expected to hold for in a design.

Assertion-based verification, as is shown in Fig. 1, is obtained by extending the capabilities of assertions by employing some strategy that uses them as a central target for a variety of verification methods. Assertion-based verification is the use of assertions for the efficient validation of a specification collection by the synergistic application of simulation, formal verification, and semi-formal verification.

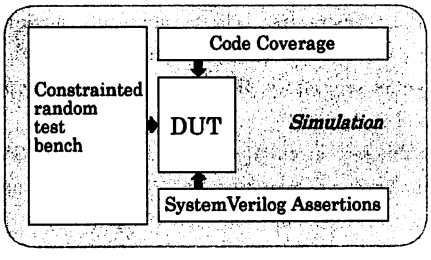

Fig.1. Assertion-Based Verification with SVA

Limited by the speed of simulation, SVA has not been widely adopted in large scale SoC designs. In order to accelerate the ABV on hardware emulation, an issue of converting SVA into synthesizable HDL code has been addressed.

2.2. SystemVerilog Assertion Syntax

In this section, we introduce basic syntax of SVA. The basic SVA building block is called *Sequence Expression (Seq)*, all SVA properties are based on sequence expressions. The simplest sequences are signals or Boolean expressions and more complex sequences consist of simple sequences and various operators, like temporal operators, etc, which is shown as follows:

```
seq::= [delay_range] seq {delay_range seq}
       /seq and seq
       /seq or seq
       /seq intersect seq
       /seq within seq
       /seq throughout seq
       /'(' seq ')'
       /seq [consecutive_repeat]
       /seq [Boolean_repeat]
       /seq {',' seq_match_item} [consecutive_repeat]
       /seq {',' seq_match_item} [Boolean_repeat]
```

where delay_range, consecutive_repeat, Boolean_repeat and seq_match_item are show below:

```
delay_range ::= '##' constant_expression
              /'##' '[' const_range_expression ']'

const_range_expression ::= constrant_expreesion : '$'
              /constrant_expression : constrant_expression
Boolean_repeat ::= consecutive_repeat
              /nonconsecutive_repeat
              /goto_repeat
consecutive_repeat::= '[' '*' const_range_expression']'

nonconsecutive_repeat::='[''='const_range_expression']'

goto_repeat::= '[' '->' const_range_expression ']'
```

The structure of an SVA property is as follows:

```
prop::= seq
      /event_control prop
      / '(' expression ')'
      /not prop
      /prop or prop
      /prop and prop
      /seq |=> prop
      /seq |-> prop
      /if '(' expression ')' prop [ else prop ]
```

where |-> and |=> are called overlapped implication operator and non-overlapped implication operator, respectively. Implication is equivalent to an if-then structure. The left hand side of the implication is called the "antecedent" and the right hand side is called the "consequent". The antecedent is the *gating* condition. If the antecedent succeeds, then the consequent is evaluated. If not, then the property is assumed to succeed by default, which is called a "vacuous success" and no error messages are displayed. Implication construct can be used

only with property definitions. It cannot be used in sequences. The difference of these two implication operators is that the evaluation of the consequent starts at the same clock cycle at which the antecedent matches or one clock cycle later.

2.3. Previous Work on SVA Synthesis

A divide-and-conquer approach for synthesis SVA sequences has been proposed in [1]. The basic idea is to break the sequence expression as a sequence of expressions concatenated with the corresponding cycle delay. They generate checker circuit for each small sequence and then connect them so that these checker circuits can perform as the actual sequence expression. In [2], basic cycle delay and implication circuits are given, as shown in Fig. 2.

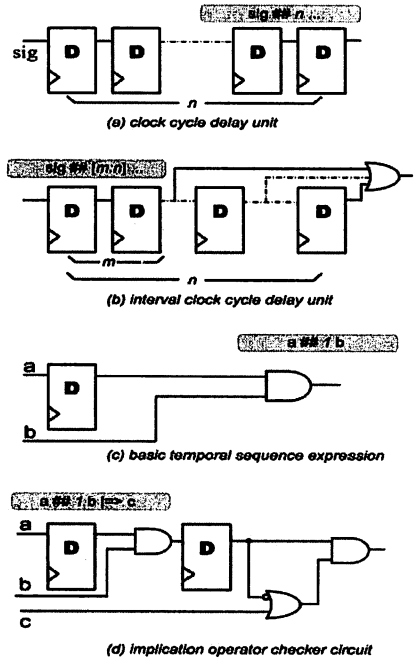

Fig.2. Basic checker circuit generation

The cycle delay is implemented by using register arrays, as shown in (a), while interval delay circuit is described in (b), with all the outputs of registers in the time window connected by an OR gate. Sub-figure (c) shows the basic sequence expression with timing relation. In (d), the checker circuit for implication operator is shown.

These typical structures of checker circuits can be used to form much more complex checker circuits. On the other hand, the divide-and-conquer approach focuses on the basic block of SVA checker circuit but neglects the holistic optimization. For complex designs, very large sets of SVA properties are required for the complete verification; therefore, hardware resources are consumed heavily by the divide-and-conquer approach. An optimization method is needed to handle the hardware cost issue from a holistic view.

3. SVA CHECKER CIRCUIT GENERATION

3.1. Counter-Based method

In the existing methods, shift-registers are used for the delay evaluation. If the delay value is n, then they use n registers. It is not efficient for large n. The basic idea of the proposed method is to use a binary counter for the delay evaluation. With this, the number of registers becomes $\log(n)$ for delay value n. This is called the counter-based method.

The basic structure of the checker circuit is shown in Fig. 3. That shows a checker circuit for the following property.

a ## 1 b ## 1 c

The circuit includes a 2-bit counter, a decoder and the control logic of the counter. We control the count-up/reset of the counter by using the input signals and the output of the decoder. The counter counts up only when some input signal occurs at the corresponding clock cycle, otherwise the counter will be reset. For example, if *a* holds at clock 0, then we control the 2-bit counter to become 01, then we check whether b holds at clock 01. If so, we continue to count up to 10 and check the value of c. If c also holds at 10, then the property succeeds.

In FPGA prototyping, a logic element (LE) containing an LUT with 4 inputs / 1 output and a register is used to implement logic circuits. So, we use the number of LE's to measure the hardware cost.

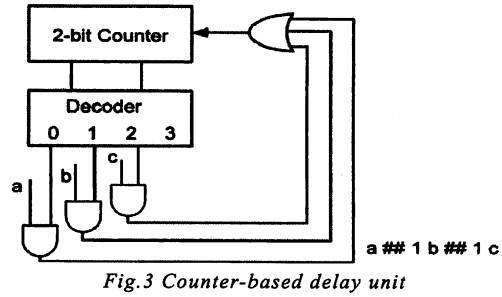

Fig.3 Counter-based delay unit

In Fig.3, 4 LE's are required to implement the counter-based checker circuit: 2 LE's for the counter and 2 LE's for the decoder and the control circuit with *a*, *b* and *c*, while the register-array based method only needs 3 LE's. Therefore, we should carefully check the feasibility when using the counter based method for the short delay, which will be discussed later.

For longer delays, the counter based method is expected to reduce the hardware cost compared with the previous method. An application example of the counter based circuit for *a ## 8 b ## 8 c |=> d ## 1 e* with 18 clock cycle delay is shown in Fig.4. The property is over 18 clocks. Since the time separations between signals are more than 1 clock cycle, we can not use just one OR gate to control the increment of the counter as in Fig.3. The increment logic in the figure includes registers to keep the counting-up states, and becomes a complex circuit. Once the signal *a* holds, we should count up during 8 clocks. Though the increment control logic of the counter becomes more complex, the checker circuit can be implemented with only 10 LE's which is smaller than the circuit with 18 LE's by the previous method. If the delay value becomes larger, we can reduce much more LE's in the checker circuit implementation.

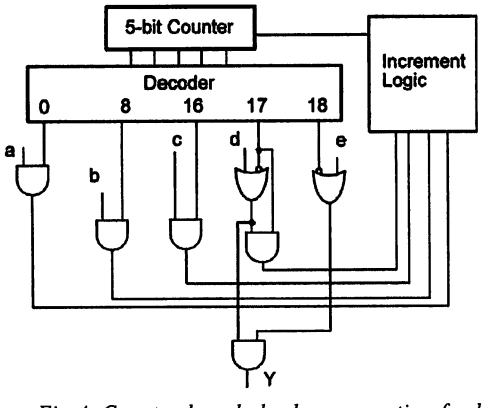

Fig.4. Counter-based checker generation for long delays

The counter based method is effective for large delay value, but seems not so effective for small delay values if we use the basic structure shown in Fig. 3. With devising the state coding in constructing the counter structure, we can reduce the hardware cost for small deval values.

Consider the following property:

a ## 1 b ## 1 c |=> d ## 1 e (1)

The checker circuit generated with the previous method is shown in Fig.5:

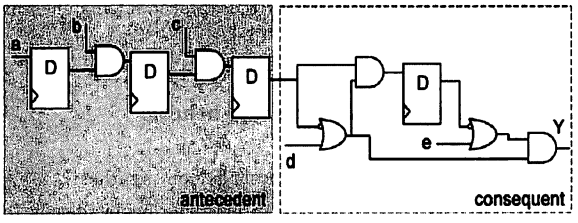

Fig.5 Previous checker circuit for property (1)

This checker circuit is constructed from 4 registers, and there are no overlap conditions, so the circuit can be modeled as a finite machine with 5 states. The state transition table is shown in Table. 1. According to the SVA syntax, if inputs signal *d* and *e* do not occur at S3 and S4 respectively, the output becomes 0, otherwise the output of the property is 1, as shown in Table. 2

Table.1 state description of property (1)

Current state	a	b	c	d	e	Next state
S0	1	0	0	0	0	S1
S1	0	1	0	0	0	S2
S2	0	0	1	0	0	S3
S3	0	0	0	1	0	S4
S4	0	0	0	0	1	S5

Table.2 output of property (1)

Current state	a	b	c	d	e	output
S3	0	0	0	d	0	d
S4	0	0	0	0	e	e

To generate the checker circuit using the counter-based method, we discuss the Boolean function according to the state machine. Assuming that signal c0, c1 and c2 are used to identify the states; the state transition can be described in the form of truth table in Table 3:

Table.3. State transfer of property (1)

Current state								Next state		
c2	c1	c0	a	b	c	d	e	c2	c1	c0
0	0	0	1	0	0	0	0	0	0	1
0	0	1	0	1	0	0	0	0	1	0
0	1	0	0	0	1	0	0	1	1	1
1	1	1	0	0	0	1	0	1	0	0
1	0	0	0	0	0	0	1	0	0	0

Note that c2 is 1 at S3 and S4. Based on the assignment, we can describe the "next state" function as follows:

Next c0 = c1'c0'a + c1 c0'c
Next c1 = c1'c0 b + c1 c0'c
Next c2 = c1 c0'c + c1 c0 d

The output Y is described as a function on c2, c1, d and e:

Y = (c2 c1 d' + c2 c1'e')'

With the assignment, all functions have less than 4 inputs, which means each of these functions can be implemented with one LE. The checker circuit by the counter based method uses only 4 LE's. Note that 5 LE's are used in the checker circuit by the previous method and we gain 20% hardware cost reduction.

The counter based method is effective for not only large delay values but also small delay values. For long delay property, the number of LE's can be reduced to log(*n*) instead of *n* by the previous method. For short delay property, the counter based method might reduce the number of LE's by devising the state encoding. The efficiency of this counter based method should be checked by using synthesis tools.

3.2. Hardware Sharing Between Properties

In this section, we discuss about the possibility of hardware sharing between properties. In previous works, one checker circuits is generated for each property, which implies that the hardware resource of checker circuits is proportional to the number of SVA properties. In practice, since several properties have similar structures, we would like to use the similarity of the properties to share hardware resources in order to reduce the total hardware cost.

The basic sharing is to share the matching part from the beginning point of several properties. For example, we can share *a ## 8* part of the following two properties. Note that *a ## 8* starts from the beginning in both properties.

Property 1: *a ## 8 b ## 8 c |=> d ## 1 e* (2)
Property 2: *a ## 8 f ## 8 c* (3)

We can also share the matching part from the ending point of properties in almost the same way.

Interval matching part like *b###8* and *f###8* in the above properties should carefully be processed. One method is to use flags keeping which value is entered. A simple example is shown in Fig.6 to show the basic idea for the previous properties.

In Fig.6, we first generate the checker circuit of property 1 using the counter-based method, then we add a OR gate to select signal *f* and *b*, so that the evaluation of the checker circuit continues if *f* or *b* holds at clock 8. A Register Logic block is also built to temporarily store the value of *f* and *b* at clock 8 and use these stored values to control the output of property 1 and 2 respectively. For example, after a holds at clock 0, we control the counter to count to clock 8, then the value of *f* and *b* are stored by the register logic block, if either *b* or *f* holds at clock 8, we continues the evaluation and check signal *c* at clock 16, if *c* also holds, we load the stored value of *f*, if it is one, then property 2 succeeds. We also check the stored value *b* at clock 18 to see if it is property 1 that is evaluating. By this method, we implement a checker circuit for two properties using the hardware resources for just one checker circuit.

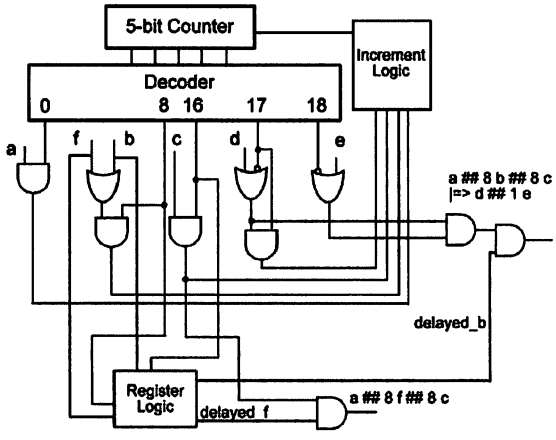

Fig.6. Hardware sharing between properties.

In reality, large parts of properties might be capable to share checker circuits for certain designs. Therefore, the hardware sharing method is expected to achieve large hardware cost reduction. The efficiency of this method has been checked experimentally.

4. EXPERIMENTAL RESULTS

The hardware costs of the counter-based method and hardware sharing method are examined using Altera Quartus II 7.1. We have applied the proposed method by hand to several properties and evaluated the checker circuits on EP1S25F1010C5 with Quartus II. Quartus II is

executed on a computer with an Intel Pentium(R) 4 2.66GHz CPU and 1GB memory.

For a property *a ## 1 b ## 1 c /=> d ## 1 e* mentioned in Section 3, the synthesis summary shows that the checker circuit generated by previous method costs 5 LUT and 4 registers, that is 5 logic elements totally, while the checker circuit of counter-based method costs 4 LUT and 4 register, 4 LE in total, which is a 20% reduction comparing to the previous method. For a property with longer delay, like a ## 8 b ## 10 c, the previous method costs 18 LE while counter based method costs only 9 LE, 50% hardware reduction is observed.

Hardware sharing between property *a ## 8 b ## 8 c /=> d ## 1 e* and property *a ## 8 f ## 8 c* has also been compared. The two checker circuits generated by previous method cost 18 LE and 16 LE respectively, therefore, 34 LE in total. The checker circuit with sharing only costs 10 LE. With the 70.6% reduction, the proposed hardware sharing method is proved to be effective on reducing hardware cost.

5. CONCLUSION

In this manuscript, a novel method of SVA checker circuit generation is proposed. For the cycle delay operation, we use counter circuits instead of register arrays, which reduces the hardware cost from O(n) to O(log(n)) for long delay, by our experiment, over 50% hardware reduction is obtained on a property with 18 clock cycle delay. For short delay properties, we also developed a effective method to reduce the number of related signals for each logic function so as to implement each function with less LUT's, about 20% of hardware reduction is observed for a property with 5 clock cycle delay.

By observing the SVA properties for many LSI designs, we notice that some SVA properties have similar structures. So we share hardware between these properties in order to further reduce the hardware cost for SVA synthesis. By introducing multiplexers into checker circuits, the checker circuit can dynamically switch between the evaluations of shared properties. Some experiments using synthesis tools are executed and about 70% hardware reduction is observed.

Our further work would be to develop an algorithm to automatically convert SVA properties into synthesizable Verilog code based on the counter-based method and hardware sharing strategy.

Acknowledgement

The work is supported in part by a fund from Toshiba, and by a grant of Knowledge Cluster Initiative implemented by Ministry of Education, Culture, Sports, Science and Technology(MEXT).

References

[1] Das S, Mohanty R, Dasgupta P, Chakrabarti P.P., "Synthesis of System Verilog Assertion", DATE 2006, Volume 2, pages: 1-6, Marth 2006.

[2] Boule M, Zilic Z, "Incorporating efficient assertion checkers into hardware emulation", ICCD 2005, pages: 221-228, Oct. 2005.

[3] Pellauer M., Lis M., Baltus D., Nikhil R, "Synthesis of synchronous assertions with guarded atomic actions", MEMOCODE 2005, pages: 15-24, July 2005.

[4] Srikanth Vijayaraghavan and Meyyappan Ramanathan, "A Practical Guide for SystemVerilog Assertions", Springer Science+Business Media Inc, ISBN 0387260498.

[5] Harry D. Foster, Adam C. Krolnik and David J. Lacey, "Assertion-Based Design", Springer Science+Business Media Inc, ISBN: 1402080271.

[6] Roy Armoni, Limor Fix, Alon Flaisher, "The ForSpec Temporal Logic: A New Temporal Property-Specification Language", Proceeding of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pages: 296-211, 2002.

[7] "IEEE Standard for Property Specification Language (PSL)", IEEE Std 1850-2005, papges: 0_1-143, 10.1109/IEEESTD.2005.97780.