9th IEEE Workshop on RTL and High Level Testing (WRTLT'08), pp.45-50, Nov. 2008.

Enhancement of Test Environment Generation for Assignment Decision Diagrams

Hideo Fujiwara[1], Chia Yee Ooi[2], and Yuki Shimizu[1]

[1]Graduate School of Information Science
Nara Institute of Science and Technology
Kansai Science City, Nara, 630-0192 Japan
{fujiwara, yuki-s}@is.naist.jp

[2]Faculty of Electrical Engineering
University of Technology Malaysia
Skudai Johor, 81300 Malaysia
ooichiayee@fke.utm.my

Abstract *— In this paper, we consider the problem of test environment generation for functional register-transfer level (RTL) circuits using an assignment decision diagram (ADD). For the test environment generation method using ADDs, Ghosh et al. [10] and Zhang et al. [11] previously proposed symbolic processing based algorithms which utilize a set of nine-valued and ten-valued algebras, respectively, to perform symbolic justification and propagation of test environment objectives. However, it is known that there are many cases that their methods fail to generate a test environment even if it exists. This paper presents a new method for test environment generation with enhanced symbolic processing (justification and propagation) rules in order to generate more test environments than previous methods.*

Keywords — Test environment, RTL test generation, high-level testing, assignment decision diagrams

1. Introduction

The problem of test generation is a basic, essential issue in the area of testing [1][2]. In the combinational test generation problem, efficient algorithms have been developed where high fault efficiency can be achieved even for large combinational circuits. However, large sequential circuits require long test generation time and achieving high fault efficiency is quite difficult. To improve the fault efficiency, researchers started to propose the test generation methods for targeting the design at higher level [3] – [11].

Ghosh et al. proposed a test generation method for functional RTL circuits which was described in a cycle-accurate behavior [10]. In their work, test environments are generated for each functional module of a given functional RTL circuit described in an *assignment decision diagram* (ADD) using the justification/propagation rules comprising nine symbols of nine-valued algebra [12]. A test sequence is then formed by substituting the corresponding test patterns for the test environment. However, regardless of the existence of some test environments, the proposed nine-valued algebra cannot always generate the test environments. To overcome this drawback, Zhang et al. upgraded the nine-valued algebra to a ten-valued algebra by taking the signal line value range into consideration. This algebra can generate much more test environments [11]. Nevertheless, there are still many test environments that could not be generated by this method.

This paper extends the method proposed by Zhang. We propose a test environment generation method which can generate any test environment that can be produced by Ghosh's and Zhang's methods, as well as many test environments which cannot be produced by the previous two methods.

The outline of the paper is as follows. Section 2 describes the terminology to be used in this paper. Section 3 defines the test environment problem. This is followed by the section that describes the justification/propagation rules introduced by Ghosh and Zhang. Section 5 explains an experiment conducted on a set of benchmark circuits and shows the result of comparing our justification/propagation rules and those from previous works. Finally, Section 6 concludes the paper.

2. Preliminaries

2.1 ADD

ADD is an acyclic graph that consists of a set of nodes that can be categorized into four types: read node, write node, operation node and assignment decision node (ADN), and a set of edges which contain the connectivity information between two nodes (Fig. 1). A read node represents a primary input port, a storage unit or a constant while a write node represents a primary output port or a storage unit. An operation node expresses an arithmetic operation unit or a logic operation unit while an ADN selects a value from a set of values that are provided to it based on the conditions computed by the logic operation units. If one of the condition inputs becomes true, the value of the corresponding data input will be selected. Although ADD was essentially introduced as an internal representation in the high-level synthesis process, it can be used to describe a functional RTL circuit, the controller part and the data path part of which are homogeneously represented.

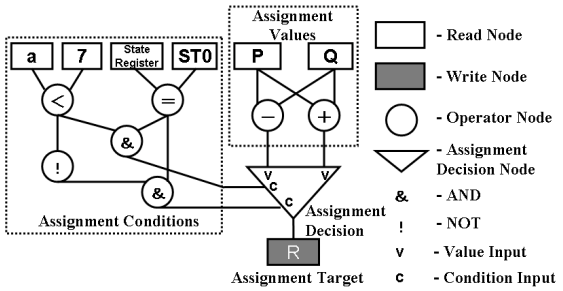

Fig. 1 ADD (Assignment Decision Diagram)

2.2 Test Environment

When a node *N* is under test, the testability of the node is guaranteed if (a) any value can propagate from a read node corresponding to a primary input port to the input of *N*, and (b) the value at the output of *N* can propagate to a write node corresponding to a primary output port. The paths which allow (a) and (b) to occur are called *justification path* and *propagation path*, respectively. Justification and propagation can be done through symbolic processing that utilizes nine-valued algebra. The series of symbols obtained from the symbolic processing that activates justification and propagation paths is known as the *test environment* for the node under test.

For a given node under test, its test sequence is generated by first extracting a test pattern from the *test set library* and by substituting the test pattern for the test environment. The test set library is obtained beforehand by first simply taking a gate-level circuit whose functionality is the same as that of the node under test, then generating the test patterns for all faults in the circuit using a combinational ATPG algorithm. In the case where the node is synthesized into a circuit which is different, fault simulation must be performed to check the fault efficiency of the test patterns.

2.3 Methods of Ghosh et. al and Zhang et. al

The nine symbols of Ghosh's nine-valued algebra, each of which can be assigned true or false, are as follows:

- *Cg*(*v*): variable *v* can be set to any value.
- *C0*(*v*): variable *v* can be set to 0.
- *C1*(*v*): variable *v* can be set to 1.
- *Ca1*(*v*): all bits of variable *v* can be set to 1's.
- *Cq*(*v*): variable *v* can be set to a constant.
- *Cz*(*v*): variable *v* can be set to high impedance *Z*.
- *Cs*(*v*): state variable *v* can be set to a specific state.
- *O*(*v*): any fault effect at variable *v* can be observed.
- *O'*(*v*): fault effect of *D'* can be observed for a single bit variable v.

To generate a test environment, first an objective like the one shown in Fig. 2 has to be set. In order to achieve the test environment objective, the test sequence for each ADD can be generated through the following two phases using the justification/ propagation rules shown in Figs 3 and 4:

***Phase 1*:** Generate the test environment of the node under test.

***Phase 2*:** Generate the test sequence of the node under test by substituting the test patterns of the gate-level circuit corresponding to the node under test for the test environment.

The procedure to generate the test environment in Phase 1 is as follows:

***Step 1*:** Find the propagation path from the node under test to a primary output using a single path sensitization method. If the path does not exist or all the paths have been traversed, terminate the procedure.

***Step 2*:** Propagate the output response of the node under test (propagation symbol *O*) to the primary output by assigning the symbols on the side inputs of the propagation path that follows the propagation rules.

***Step 3*:** Assign *Cg* to the input of the node under test.

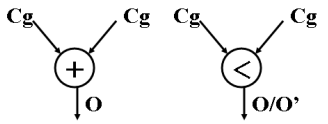

Fig. 2 Test Environment Objectives

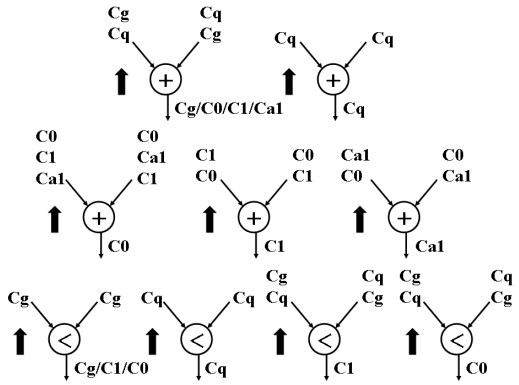

Fig. 3 Justification Rules for Addition and Comparison Nodes

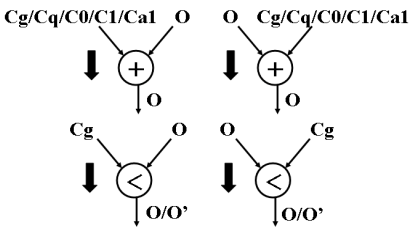

Fig. 4 Propagation Rules

***Step 4*:** Perform implication to detect any conflicts earlier.

***Step 5*:** All the test environment objectives are justified using the branch and bound method.

***Step 6*:** If all the test environment objectives have been justified, the test environment is produced and the procedure is completed. If we cannot achieve all the objectives and there are still propagation paths which are not yet searched, go back to Step 1.

Ghosh et. al's procedure does not allow the justification of two arbitrary values by just one variable. Zhang et. al [11] resolves the problem by considering signal line value range where symbol *Cp* is added to the algebra.

3. Test Environment Generation Problem

To efficiently generate a test sequence that provides a high fault efficiency, it is important to be able to generate test environments for as many nodes as possible. This section redefines the test environment generation problem. Furthermore, the completeness of the test environment algorithm and the justification/ propagation rules of the test environment are discussed.

Definition 1 (*Test environment generation problem*)

Input: An ADD, a test environment objective, and a set of justification/propagation rules

Output: (a) Decision (yes/no) on whether there exists any test environment that satisfies the test environment

objective and the set of justification/propagation rules; (b) Test environment if one exists.

The rules of Ghosh and Zhang (referred to as *Rule G* and *Rule Z*, respectively in the following text) cannot generate test environments in many cases even if they exist. The next section describes the extension of the justification/propagation rules that can generate as many as possible test environments including those which cannot be generated by Rule G and Rule Z.

4. Enhancement of Justification/Propagation Rules

4.1 Rule M (Multiple-Path Sensitization)

Rule G and Rule Z use the propagation rules that are based on the *single-path sensitization* concept. Let's consider the ADD of Case [a] in Fig. 5. The addition node at the lower part of the figure receives the fault effect *O* at both of its inputs due to the reconvergent path. Rule G and Rule Z cannot propagate the fault effect to the output of the addition node since they did not consider multiple-path sensitization.

Therefore, we extend the justification/ propagation rules such that the fault effects from different paths can be propagated to a primary output. Let's consider Case [a] again. The symbolic processing of $O+Cq$ and $2O+Cq$ can be produced by applying the operation of the node. By introducing this new symbolic processing, the fault effects propagating from two different paths can reach the primary output.

In this paper, we will refer to this new justification/propagation rules based on multiple-sensitization as *Rule M*. Its justification rules are the same as that of Rule Z whereas its propagation rules are extended to include the new symbols of $kC1$, kO, k_1O+k_2C1, $kO+Cq$, where k, k_1 and k_2 are constants. The new propagation rules for an addition node are demonstrated in Fig. 6. Similar new propagation rules can be derived for the subtraction node and the logic operation node. Different from the previous rules, symbol *O* in Rule M represents the same fault effect.

4.2 Coverage Relation between Rules G, Z and M

Let T_G, T_Z, *and* T_M denote the sets of test environments generated by Rule G, Rule Z and Rule M, respectively for a given ADD. The examples in Fig. 5 and Fig. 7 illustrate cases where no test environment exists under Rule G and Z but a test environment can be generated by Rule M. We have the following theorem.

Theorem 1: For any node *N* of an ADD, sets of test environments T_G, T_Z, and T_M which are generated by Rule G, Rule Z and Rule M, respectively, have the coverage relation of $T_G \subset T_Z \subset T_M$.

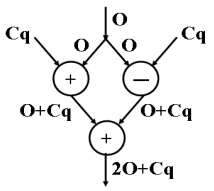

Fig. 5 Case [a]

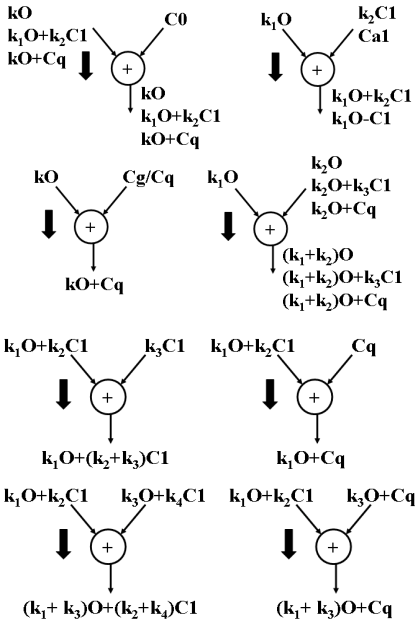

Fig. 6 Propagation Rules of Rule M

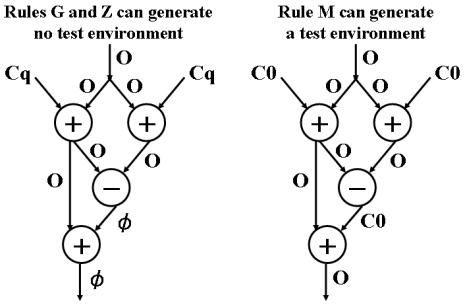

Fig. 7 Case [b]

4.3 Rule E (Essential Justification/Propagation)

Rule M can neither justify the control symbol *Cg* nor propagate the observing symbol of *O* for modulo operation node. *Cg* cannot be justified because the output of the modulo operation node cannot be set to all of the possible 2^n values at the input of modulo operation node under the assumption of the input and the output having the same bit width. Furthermore, symbol *O* cannot be propagated because there is a possibility of a fault at the input of modulo operation node being masked.

However, even though variable *v* cannot be set to all of the possible 2^n values, it is sufficient if variable *v* can be justified to any value which is essentially justifiable for a targeted ADD. This set of possible values is called *range of values*. Based on this new finding, we derive two new symbols *Cg** and *O** from *Cg* and *O* for test environment generation.

- *Cg*(v)*: Variable *v* can be set to any value which is *essentially* possible.
- *O*(v, p)*: Error at variable *v* which is *essentially* observable can be observed through path *p*.

By adding these new symbols to Rule M, we derive a new justification/propagation rules called *Rule E* (E is the first letter of term '*Essentially*'). The following shows the relationship between symbols *Cg* and *O* with symbols *Cg** and *O**.

$$Cg(v) \Rightarrow Cg^*(v)$$
$$O(v) \Rightarrow \exists p, O^*(v, p)$$

It is sufficient to consider the justification of *Cg*(v)* only as the control symbol of a test environment objective during the test environment generation since *Cg(v)⇒Cg*(v)*. On the other hand, instead of the symbol *O(v)*, the propagation of *O*(v, p)* for any possible propagation path *p* should be considered since the contraposition of O(v) ⇒ ∃p, O*(v, p) holds. Fig. 8 depicts an example of a test environment objective.

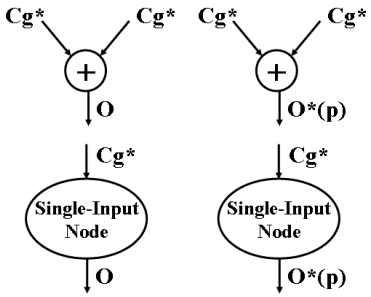

Fig. 8 Test Environment Objectives

Figs 9-11 show justification/propagation rules for addition nodes, modulo operation nodes and single-input nodes, respectively. A single-input node represents either a node that originally has a single input, or a 2-input node with a constant assigned to one of its input. Examples of the latter include addition nodes with one of its inputs assigned with one, modulo operation node that calculates mod 128, etc.

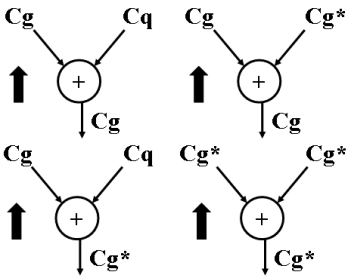

Fig. 9 (a) Justification Rules of Addition Node

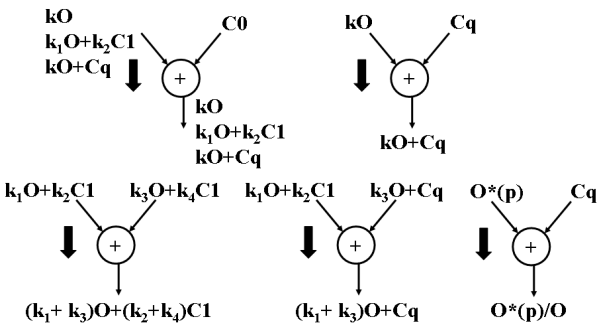

Fig. 9 (b) Propagation Rules of Addition Node

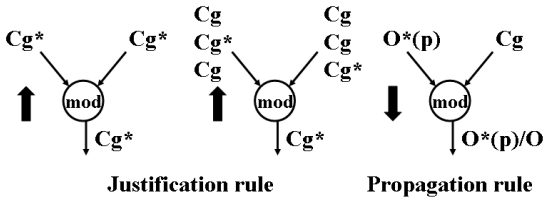

Fig. 10 Justification/Propagation Rules of Modulo Operation Node

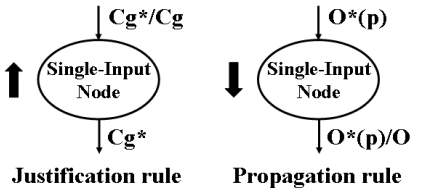

Fig. 11 Justification/Propagation of Single Input Node

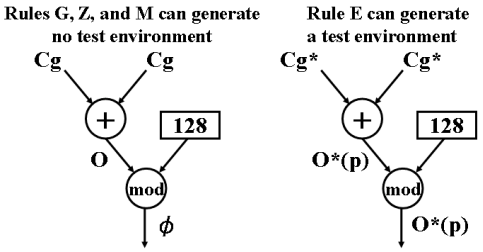

Fig. 12 Case [c]

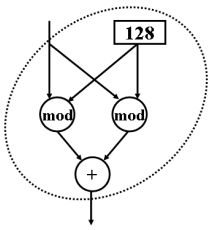

Fig. 13 Single Input Node by Composition

Justification rules of *Cg** for addition nodes and multiplication nodes are similarly defined based on the justification rules of *Cg*. For modulo operations, justification of *Cg** is possible though *Cg* is not justifiable. For single-input nodes, justifiability of *Cg* depends on the types of operation after realization of the single-input node while the justification of *Cg** is always possible.

As shown in Figures 9-11, the propagation rules of observing symbol *O* are similar to that in Rule M while the propagation rules for observing *O** are based on single-path sensitization concept. This is to guarantee the completeness of the newly defined Rule E. In other words, by limiting the propagation of *O** using single-path sensitization concept, the test environments generated include the propagation sequence that can observe any errors which are essentially observable.

4.4 Coverage Relation between Rule M and Rule E.

It is obvious that test environments generated by Rule M can also be generated by Rule E. Moreover, Case [c] in Fig. 12 illustrates an example where no test environment exists under Rule M but one exists under Rule E. Therefore, we have the

following theorem.

Theorem 2: For any node N in an ADD, the relationship between T_M and T_E, which are the sets of test environments generated by Rules M and E respectively, can be defined as $T_M \subset T_E$.

4.5 Single-Input Node by Composition

When two controlling symbols *Cg* set at the inputs of the addition node in Fig. 13 are being justified, the junction at the input side has two *Cg*s which conflict with each other and thus the test environment generation fails. For such a situation, we should regard all the nodes in the part marked by the dotted line as being merged into one single-input node considered as a node under test. By setting *Cg* at the input and *O* at the output as the test environment objectives, the test environment can be generated. At the input of the addition node, any values which are essentially justifiable can be controlled by the test environment that satisfies these test environment objectives.

Furthermore, let's consider the propagation of observing symbol *O* from the left input of the example circuit in Fig. 13. In this case, observing symbol *O** is needed to allow propagation through modulo operation nodes but it is constrained by the concept of single-path sensitization and thus propagation operation fails at the addition node. However, observing symbol *O** can propagate to the output of the node if a single-input by composition is considered.

By introducing the concept of *single-input node by composition*, much more test environments can be generated using Rule E.

5. Case Study

To compare the effectiveness of Rule G, Rule Z, Rule M and Rule E, test environment generation algorithms based on each rule have been applied on two ADDs shown in Figures 14 and 15 and comparisons have been done. Test environment coverage, which is used as an evaluation measure in the case study, is defined as follows. Let TEC(N_i) denote Test Environment Coverage for node N_i. Let TEC denote Test Environment Coverage.

$$TEC(N_i) = \frac{\begin{array}{c}Number\ of\ effective\ test\ patterns\ for\ N_i \\ with\ respect\ to\ generated\ test\ environment\end{array}}{Number\ of\ test\ patterns\ for\ N_i}$$

$$TEC = \frac{Sum\ of\ TEC(N_i)\ for\ all\ N_i}{Total\ number\ of\ nodes}$$

Tables 1 and 2 show the results for Circuits Ex.1 and Ex.2. In each table, columns from left indicate the test generation methods, test environment coverage, fault coverage, test sequence length, test generation time, fault simulation time, and total computation time, respectively. The test generation methods in the first column include RTL test generation methods using Rule G, Rule Z, Rule M and Rule E, respectively, as well as gate-level test generation methods that have the time limits of 1 second per fault and 100 seconds per fault. We used Synopsys' TetraMax for gate-level ATPG, running on SunFire X4100 (CPU 3.0GHz, Memory 16GB).

For all of the circuits, Rule M has better test environment coverage than that of Rule G and Rule Z. Among all the rules, Rule E shows the highest test environment coverage. The third column shows the fault coverage obtained from the fault simulation of the synthesized circuits using the generated test sequence. The correlation between fault coverage and test environment coverage is high. Rule M shows better fault coverage compared to Rule G and Rule Z. In addition, Rule E has the highest fault coverage among all the rules. Compared to Rule G and Rule Z, Rule E generates much more test environments and test sequences that provide much higher fault coverage. For the performance comparison between RTL ATPG and gate-level ATPG, RTL ATPG with Rule E achieves higher fault coverage compared to gate-level ATPG with limit of 1 second per fault. Furthermore, the total computation time of RTL ATPG with Rule E is 1/887 and 1/131 of the gate-level ATPG with limit of 1 second per fault for Circuit Ex.1 and Circuit Ex.2, respectively. When the abortion time is extended to 100 seconds for gate-level ATPG, higher fault coverage can be achieved. However, the CPU time rose to more than 10,000 seconds. From the above case study, we can summarize that RTL ATPG with Rule E is able to generate the test sequences whose fault coverage is equal to or higher than gate-level ATPG with much shorter processing time.

6. Conclusion

This paper studied the test environment generation problem for each module in a functional RTL circuit represented in ADD. Ghosh et. al [10] and Zhang et. al [11] introduced the test environment generation methods using nine-valued algebra and ten-valued algebra symbolic processing, respectively. This paper proposed an effective and efficient test environment generation method by extending the existing symbolic processing rules (justification/propagation). Our proposed method can generate more test environments in addition to the test environments generated by the previous methods. The set of test environments generated by our proposed rules covers the set of test environments generated by the previous rules. Moreover, the test environment coverage obtained in the case study showed the effectiveness of the proposed rule, Rule E. Compared to gate-level ATPG, our RTL ATPG with Rule E can drastically reduce the test generation time without sacrifice of fault coverage.

Acknowledgements This work was supported in part by Japan Society for the Promotion of Science (JSPS) under Grants-in-Aid for Scientific Research (B) (No. 20300018).

References

[1] H. Fujiwara, *Logic Testing and Design for Testability*, MIT Press, 1985.

[2] M. Abramovici, M. A. Breuer, and A. D. Friedman, *Digital Systems Testing and Testable Design*, IEEE Press, 1990.

[3] B. T. Murray and J. P. Hayes, "Hierarchical test generation using precomputed tests for modules," *IEEE Trans. Computer-Aided Design*, vol. 9, pp. 594–603, June 1990.

[4] K. Roy and J. A. Abraham, "High-level test generation using data flow descriptions," in *Proc. Eur. Conf. Design Automation*, Mar. 1990, pp. 480–484.
[5] J. Lee and J. H. Patel, "An architectural level test generator for a hierarchical design environment," in *Int. Symp. Fault-Tolerant Computers*, June 1991, pp. 44–51.
[6] C. H. Cho and J. R. Armstron, "B-algorithm: A behavioral test generation algorithm," in *Proc. Int. Test Conf.*, Oct. 1994, pp. 968–979.
[7] F. Corno, P. Prinetto, and M. S. Reorda, "Testability analysis and ATPG on behavioral RT-level VHDL," in *Proc. Int. Test Conf.*, Oct. 1997, pp. 753–759.
[8] S. Bhatia and N. K. Jha, "Integration of hierarchical test generation with behavioral synthesis of controller and data path circuits," *IEEE Trans. VLSI Syst.*, vol. 6, pp. 608–619, Dec. 1998.
[9] S. Chuisano, F. Corno, and P. Prinetto, "RT-level TPG exploiting highlevel synthesis information," in *Proc. VLSI Test Symp.*, Apr. 1999, pp. 341–346.
[10] I. Ghosh, and M. Fujita, "Automatic test pattern generation for functional register-transfer level circuits using assignment decision diagrams," *IEEE Trans. Computer-Aided Design*, Vol. 20, No. 3, pp. 402-415, March 2001.
[11] L. Zhang, I. Ghosh, and M. Hsiao, "Efficient sequential ATPG for functional RTL circuits," in *Proc. International Test Conference*, pp. 290-298, September 2003.
[12] V. Chaiyakul, D. D. Gajski, and L. Ramachandran, "High-level transformations for minimizing syntactic variances," in *Proc. Design Automation Conference*, pp. 413-418, June 1993.

Table 1. Circuit Ex.1

Method	TEC (%)	Fault Coverage (%)	Test Length (cycles)	Test Generation Time (sec)	Fault Simulation Time (sec)	Total Time (sec)
RTL ATPG (Rule G)	39.51	77.18	254	0.066	0.18	0.246
RTL ATPG (Rule Z)	39.51	77.18	254	0.066	0.18	0.246
RTL ATPG (Rule M)	83.95	90.09	804	0.119	0.54	0.659
RTL ATPG (Rule E)	96.30	90.14	846	0.174	0.55	0.724
GL ATPG (1sec/fault)	---	82.75	398	---	---	617.72
GL ATPG (100sec/fault)	---	91.47	597	---	---	13951.41

Table 2. Circuit Ex.2

Method	TEC (%)	Fault Coverage (%)	Test Length (cycles)	Test Generation Time (sec)	Fault Simulation Time (sec)	Total Time (sec)
RTL ATPG (Rule G)	28.57	53.99	35	0.003	0.01	0.013
RTL ATPG (Rule Z)	28.57	53.99	35	0.003	0.01	0.013
RTL ATPG (Rule M)	28.57	53.99	35	0.003	0.01	0.013
RTL ATPG (Rule E)	90.48	88.80	210	0.008	0.01	0.018
GL ATPG (1sec/fault)	---	88.65	56	---	---	2.24
GL ATPG (100sec/fault)	---	88.65	56	---	---	137.66

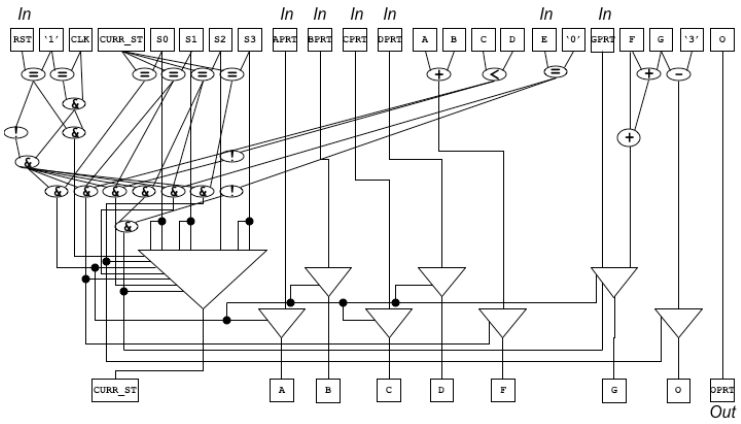

Fig. 14 Circuit Ex.1

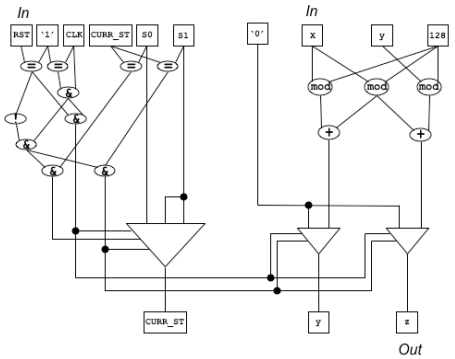

Fig. 15 Circuit Ex. 2