An Approach for Redesign for Testability at the RT Level

Haidar M. Harmanani and Salam Harfoush

Keywords— **BIST Insertion, Test Synthesis**

Abstract

This paper presents a new approach for redesign for testability at the Register-Transfer Level (RTL). The method identifies hard to test parts in RTL designs that were synthesized, either *manually* or *automatically* using a high-level synthesis tool. The design is modified by *inserting* additional registers that are active during test mode. The insertion process is followed by a *test selection* process that uses functional test metrics in order to minimize test overhead. Finally, *test scheduling* is performed in order to minimize the overall test time and the number of test sessions. The system outputs a VHDL description of the resulting testable data path along with the test plan.

I. Introduction

The complexity of VLSI circuitry has complicated the design and test of digital circuits. Recent trends in synthesis have been moving synthesis higher in the design hierarchy. *High-level synthesis* has emerged as a good approach for top down design methodology [5]. Though some researchers have recently integrated design and test synthesis [2], [6], [8], [11], most other researchers have ignored testability considerations at the system level. Add to that the numerous designs that are *manually* synthesized without testing considerations.

Design for testability (DFT) techniques [1] emerged as a solution that aims at efficient and cost effective testing by enhancing the controllability and observability of the circuit under test; the control and observation of the circuit under test are central to implementing its test procedures. Within DFT, Built-In Self-Test (BIST) was proposed with test generation and response analysis occurring on-chip. The advantage for BIST is that designs can be "tested per clock," potentially enhancing test application time, delay testing, and defect coverage. However, it often requires a higher hardware overhead and induces more delay during normal operation mode.

A. Background

This paper discusses testing within the scope of pseudorandom BIST. In order to test a kernel using the BIST methodology, every input port has to be fed by a *Test Pattern Generation Register* (TPGR) which is based on autonomous Linear Feedback Shift Register (LFSR). Every output port must feed a *Multiple Input Signature Register* (MISR) that evaluates the test responses . In our work, we determine the fault coverage using logic level fault simulation. One of the difficulties in implementing BIST techniques is the register self-adjacency problem[1], arising due to structures similar to the one shown in Figure 1(a). In this circuit, it is not possible to assign R_2 as both a TPGR and an MISR in the same test session. Some researchers used the MISR outputs, which are essentially random vectors, as random patterns [15]. The problem in this approach is that errors in the MISR propagate erroneous test patterns that are applied to the ALU, which then tend to produce errors in the BILBO[2]. This problem can be rectified using a *concurrent built-in logic-block observation* (CBILBO) register [1]. The CBILBO register can operate simultaneously as an MISR and a TPGR. The disadvantage of the CBILBO is that it is very costly in area (about 1.75 times the size of a BILBO [2]) and induces more delay during normal operation mode.

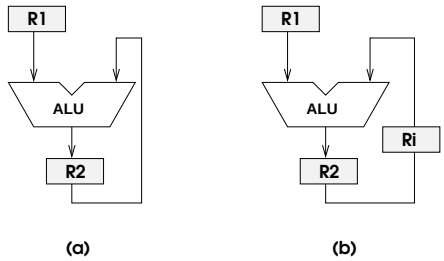

Fig. 1. a) Non-observable ALU due to self-adjacency, b) Testable ALU after test insertion

Hudson [12] showed that when self-adjacent registers are configured as test pattern generators, the additional feedback that made the register self-adjacent can greatly reduce its ability to retain error information as a signature analyzer. Kim [15] showed that random patterns generated by signature registers are rarely repeated when the number of test patterns is relatively small compared to the number of possible patterns concluding that signatures registers can be used as test pattern generators. However, we note that this is not the case with self- adjacent registers since, when configured as signature registers, they have an increased probability of aliasing.

B. Related Work

Many researchers have addressed how to automatically generate testable designs while minimizing test overheads. Lin, Njinda and Breuer [19] proposed a system based on

Department of Computer Engineering & Science, Lebanese American University, P.O. Box 36, Byblos, Lebanon
S. harfoush is with The Central Bank of Lebanon

[1] A register is self-adjacent if an output of that register feeds through combinational logic and back into itself.

[2] A BILBO is a test register that can act as a TPGR and as an MISR but in different test sessions.

the BILBO methodology. The system constructs all possible embeddings for each kernel[3] and determines the compatibility between each two embeddings. Two embeddings are compatible if they do not have resource conflicts and can be executed concurrently. The system generates next representative designs for the testable design space using a branch and bound procedure. Each representative design has its own test time and area overhead and the designer uses these data to choose between representative designs. An expert system helps the designer in the selection process. Once the designer makes his selection, a modification process is carried out to add the test hardware to the circuit under test. Kim, Tront and Ha [15] developed the BIDES knowledge-based expert system for test insertion in RTL designs using the BILBO methodology. The design process consists of an initial design and subsequent redesign steps that are repeated until an acceptable solution is obtained. Techniques in AI planning are employed for backtracking in the redesign steps. A family of testable designs can be produced via user interactions. The system works in a local search fashion and lacks a global view of the design space. Craig, Kime and Saluja [4] proposed optimal and sub-optimal procedures for scheduling the execution of tests associated with a testable design in order to minimize test time. They formulated the test scheduling problem for kernels having equal test lengths as a *clique covering problem* which is NP-hard [10]. For the unequal test length problem, a transformation is used that partitions the tests for the kernels into equal length sub-tests, and the procedure for the equal test length problem is then employed. It is obvious that if the longest test length is much greater than the shortest test length, the problem becomes intractable.

C. Problem Description and Significance

The problem we address in this paper is as follows: Given an RTL description of a datapath, the purpose of the redesign for testability method is to improve its testability by: 1) inserting additional registers, active during test mode only, if necessary; 2) schedule the resulting structure into the minimal number of test sessions so as to reduce the overall test time.

In order to reduce test penalty and ensure the datapath structural testability, it is necessary to automate the BIST insertion process. We solve the BIST insertion problem in two stages. In the first stage, necessary registers are inserted in the datapath so as to guarantee the datapath structural testability (section 2). In the second stage, datapath registers configurations are explored in order to improve the datapath cost while trading-off with test time and quality. The resulting datapath, including inserted test structures, is finally synthesized in VHDL and fed to a logic synthesis tool.

This paper is organized as follows: in section 2 we describe the test insertion process. Section 3 describes the selection process while section 4 describes our scheduling approach. Results are presented and discussed in section 5. We conclude with remarks in section 6.

[3] An embedding is defined as the structure formed of a kernel and its associated Pseudorandom Pattern Generators (PRPGs) and Signature Analyzer (SA).

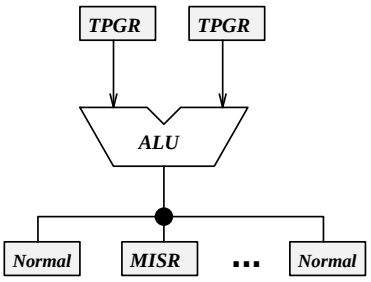

Fig. 2. Testable Functional Block (TFB)

II. BIST Test Points Insertion

The insertion process is based on the notion of *structural testability*, introduced in our earlier work [11]. The key element of the structural testability model is the *Testable Functional Block* (TFB) shown in Figure 2. A TFB consists of an ALU and a set of input and output registers. There are at least two registers at the input ports of a TFB that can be configured as TPGRs during test mode. The output port of a TFB is connected to a set of registers, one of which is configured as an MISR in test mode. Although a basic TFB consists of two TPGRs and one MISR, it should be noted that these BIST registers maybe shared by other TFBs in the datapath. This means that some of these registers at the TFB output port maybe configured as BILBO registers. Furthermore, some other non-BIST registers at the TFB output port may have to be configured as TPGRs if they control the input port of other TFBs. A datapath that consists of TFBs is *structurally testable* [11].

The insertion problem then becomes one of inserting registers in order to transform the datapath into a structurally testable *one*. The idea is to modify the datapath in order to create TFBs by: 1) breaking self-loops that are feeding a module output back to its input, and 2) inserting additional registers to cover every port in the datapath with at least one register, to be converted in the next stage into a test point.

Based on the above, there are two general cases that may require test insertion for testability enhancements. In the first case, a CLB output feeds immediately an input port of one or more other CLBs, either directly or through a multiplexer as shown in Figure 3(a). In this case, a register must be inserted between both CLBs if no other register covers this port. The inserted register would generate patterns for one of the CLBs and compress the signature for the other one. The second case is due to register self adjacency problem shown earlier in Figure 1(a).

The above cases could feed into CLBs directly or through a multiplexer. For a non-trivial circuit, there are many paths to check and thus a lot of possibilities to make it testable; hence, the difficulty in the above problem. It should be noted that not all self-adjacent registers create a testing problem as shown in Figure 3(b) and 3(c). If an ALU has two or more output registers, one of which is self-

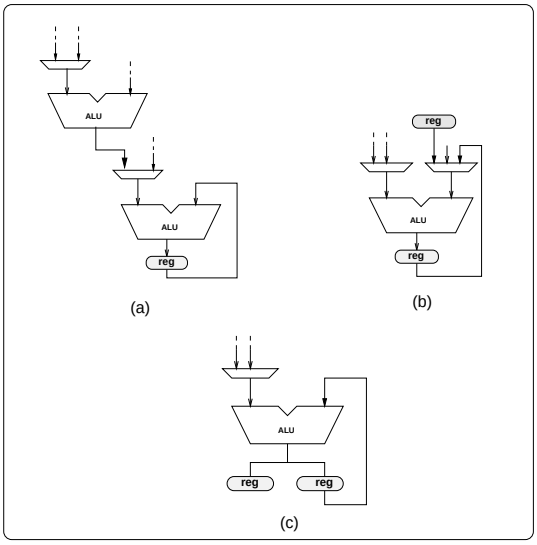

Fig. 3. Insertion Cases

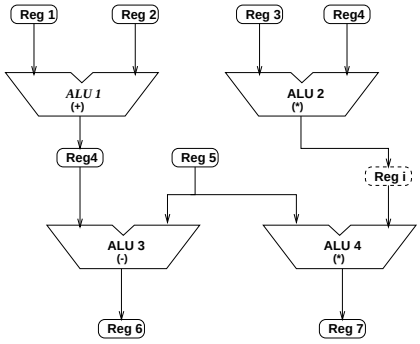

Fig. 4. Data path example

adjacent, then the self-adjacent register may be configured as a TPGR to the ALU and one of the other output registers would be configured as an MISR (Figure 3(c)). The insertion algorithm, shown in Figure 5, has a worst case run time in the order of $O(n^3)$, where n is the number of components in the input circuit.

To illustrate our method we will use the simple datapath example in Figure 4. In order to be able to test ALU_2 under BIST, we need to insert an additional register, Reg_i, in order to compress the test patterns. Reg_i may also be used to generate random patterns for ALU_4.

III. BIST Test Points Selection

Once the datapath has been ensured to be structurally testable, test points must be selected. There are two conditions that should be satisfied:

1. The TPGRs at the input ports of a single ALU cannot be the same due to correlation problems.
2. A TPGR cannot be used as an MISR for the same ALU in order to avoid the self-adjacency problem.

An initial and straightforward selection is to configure all registers connected to primary inputs as TPGRs, and registers connected to the primary outputs as MISRs. The rest of the registers are configured as BILBOs. This selection is possible since every datapath port is covered by at least one register due to its structurally testability. Obviously, this will result in a datapath characterized with high fault coverage but which incurs additional hardware overhead and delay. Another approach is to reduce the test hardware overhead by considering modules functionality, using the test metrics developed in [3]. Thus, the selection process will result in two possible extreme solutions though it is possible to explore in between additional testable designs by adding or removing test points. It should be noted that other test metrics, such as the ones in [6], [7], [9], could also be used in this case in our method.

ALU	First TPGR	Second TPGR	MISR
ALU_1	Reg_1	Reg_2	Reg_4
ALU_2	Reg_2	Reg_3	Reg_i
ALU_3	Reg_4	Reg_5	Reg_6
ALU_4	Reg_5	Reg_i	Reg_7

TABLE I
Initial test mapping for the data path example

We use two test metrics, *randomness* and *transparency*. A module is random if its output is random enough to act as a random pattern generator for all modules connected to its output port. On the other hand, a module is transparent if it can pass the faults generated by other modules to its MISR. Thus, we remove a TPGR if it is at the output port of a module whose responses are random. In the same token, an MISR is removed if the faults can be propagated through an intermediate module to another MISR without a loss in randomness. For example in Figure 4, ALU_3 is transparent; therefore we can use register reg_6 as an MISR for ALU_1 as well as for ALU_3. On the other hand, if an ALU is random, then its output maybe used as random patterns for other ALUs. In Figure 4, ALU_1 is random, and therefore its output may be used to generate random patterns for ALU_3.

A. BIST Selection Algorithm

The selection algorithm starts by generating all possible mappings of the datapath ALUs with all different possible TPGRs and MISRs. For the data path example of Figure 4, the initial test mapping is shown in Table I.

A further reduction of test points is explored next using functional test metrics [3]; thus, additional mappings are added to list of test mappings (Table I). Note that in some cases, by removing a test point due to test metrics, a register maybe saved if this register was inserted during the insertion phase. The new mappings after the randomness and transparency metrics are shown in Table II.

Once all possible mappings have been generated, the system removes TPGRs that are at the same input ports of an ALU due to the correlation condition. Furthermore, registers that are TPGRs and MISRs for the same ALU in the same session are removed due to self-adjacency. The updated list of registers is next sorted according to the number of times a register is used in the test mapping. The

■ **Input:** A list of all datapath ALUs and the registers at their ports.
■ **Output:** A list that contains all ALUs and their possible TPGRs and MISRs.

```
Check the connection at every ALUs port in the list
    if it is a self-adjacent output register and there is no other output register then
      insert a register
    if it is an ALU that directly feeds into the input port then
      insert a register between both ALUs.
    else
      if another ALU is fed at the output port by the current ALU
        if the current ALU has other output register(s)
          if the register is self-adjacent then
            insert a register
        else
          insert a register
    if it is a multiplexer at the input port and if the inputs of the MUX do not include registers then
      insert a register
    else
      if the inputs to the MUX include only one self-adjacent register then
        insert a register
    if the MUX is at the output port of the ALU
      if the ALU has a self-adjacent register at its output port
        then insert a register
      else
        if the ALU has no registers at the output port
          if at the output of the MUX there is a self-adjacent register then
            insert a register
        else if there is no registers at the output of the MUX then
          insert a register
```

Fig. 5. Test Insertion Algorithm

ALU	First TPGR	Second TPGR	MISR
ALU_1	Reg_1	Reg_2	Reg_4
ALU_2	Reg_2	Reg_3	Reg_i
ALU_3	Reg_4	Reg_5	Reg_6
ALU_4	Reg_5	Reg_i	Reg_7
ALU_3	Reg_1	Reg_5	Reg_6
ALU_3	Reg_2	Reg_5	Reg_6
ALU_4	Reg_5	Reg_2	Reg_7
ALU_4	Reg_5	Reg_3	Reg_7
ALU_1	Reg_1	Reg_2	Reg_6

TABLE II
TEST MAPPING FOR THE DATA PATH EXAMPLE

final test plan is selected by choosing the mappings with the highest weight. For the example datapath in Figure 4, the highest weighted register is Reg_2 resulting in the final list of selected mappings shown in Table III.

The above problem was formulated as a set-covering problem and solved in a near optimal approach using a greedy technique. The selection algorithm, shown in Figure 6, has a worst case run time of $O(a^2r^2)$ where a is the number of ALUs in the input circuit and r is the number of registers in the initial circuit.

IV. The Scheduling Process

The final step is the scheduling of different datapath components into different test sessions. The goal of this step is to minimize the number of test sessions by maximizing the number of ALUs tested in the same test session. However, there are conditions that restrict two ALUs from being tested in parallel. These conditions are:

1. If two ALUs have the same MISR, then they cannot be tested at the same time.
2. If an ALU's MISR is another ALU's TPGR then these two ALUs cannot be tested at the same time, unless the register in concern is a CBILBO, which is not used at this stage in our system

The scheduling process is divided into two main steps that resolve the first and the second condition. In the first step, ALUs that have the same MISR are assigned to different test sessions. Each ALU is also assigned a weight that is equal to the number of times its MISR is used by other ALUs. For the mappings of Table II, ALU_1 and ALU_3 have the same MISR, and thus they are assigned to different test sessions with a weight of two. ALU_2 and ALU_4 are assigned to the first test session with weights of one. The second phase deals with the second condition. In this phase, every ALU's TPGRs is compared with every other ALUs MISR within the same test session. If they are equals then the ALU with the smallest weight is moved to

ALU	First TPGR	Second TPGR	MISR
ALU_2	Reg_2	Reg_3	Reg_i
ALU_3	Reg_2	Reg_5	Reg_6
ALU_4	Reg_5	Reg_2	Reg_7
ALU_1	Reg_1	Reg_2	Reg_6

TABLE III
FINAL SELECTED MAPPINGS

```
■ Input: A list of all possible test mapping of the datapath ALUs
■ Output: Minimum cost test registers mapping for the datapath.

For every module in the datapath
{
    if the ALU is random apply the randomness check
    if the ALU is transparent apply the transparency check
}
Sort remaining registers in descending order according to their occurrence in the mapping.

for every register in the list do
{
    get the first mapping that contains the register at any one of its input ports
    for all the mappings of the same ALU that contain this register
        add the weights of the selection points.
        pick the highest weighted mapping.
    pick the ALU that corresponds to the above mapping.
    remove all other mappings that contain this ALU in the selection list.
}
```

Fig. 6. The Greedy Set Cover Selection Algorithm

```
■ Input: Minimum cost test registers mapping for the datapath, Mapping[n]
■ Output: Sub-optimal test schedule for the datapath

// Two ALUs with the same MISR may not be scheduled in the same session
repeat {
    CurrentSession ← 1
    CurrentMapping ← Mapping[1]
    Session(Mapping[1]) ← 1
    for every other mapping X in list, if its MISR is the same as CurrentMapping
        CurrentSession ← CurrentSession +1;
        Session(X) ← CurrentSession;
        Adjust the weight of all the mappings whose session has been changed to be CurrentSession.
} until there are no more sessions
// Two ALUs with the TPGRs of one is the MISR of the other may not be scheduled in the same session.
for every mapping i
{
    for every other mapping j
        if i and j are scheduled in the same session and if either TPGR is equal to the MISR of the other mapping
        {
          if (weight(i) < weight(j))
            Session(i) ← weight(i) + 1
          else
            Session(j) ← weight(j) + 1
        }
}
```

Fig. 7. The test scheduling algorithm

another test session. For the data path example in Figure 4, this results with the following test sessions:

Session 1: ALU_3, ALU_2, ALU_4

Session 2: ALU_1

In the second step every ALU's TPGR is compared to every other ALUs MISR within the same session. If they are equal, then the ALU with the least weight is moved to another test session. The final schedule for the example, after using test metrics, is shown in Table IV.

The scheduling algorithm, shown in Figure 7, has a worst case run time equal to $O(a^2)$ where a is the number of ALUs in the input circuit.

Session	ALU	TPGR1	TPGR2	MISR
1	ALU_2	Reg_2	Reg_3	Reg_i
1	ALU_3	Reg_2	Reg_5	Reg_6
1	ALU_4	Reg_5	Reg_2	Reg_7
2	ALU_1	Reg_1	Reg_2	Reg_6

TABLE IV
FINAL TEST PLAN FOR THE DATA PATH EXAMPLE

V. Results

In order to validate the Redesign for Testability (ReTest) approach, we attempted four benchmark examples that were found in the literature. The examples were automatically generated using *High-Level Synthesis* tools. In what follows, the experimental procedure is first explained and then results are presented.

A. Experimental Procedure

In order to validate our system shown in Figure 8, various published designs for benchmark circuits were attempted. However, due to the lack of VHDL descriptions for these designs, they were manually captured in struc-

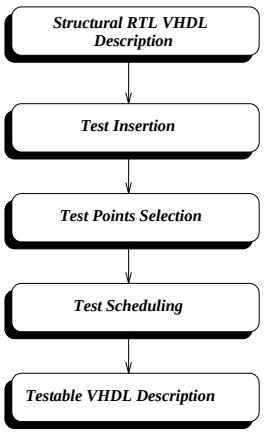

Fig. 8. ReTest System Description

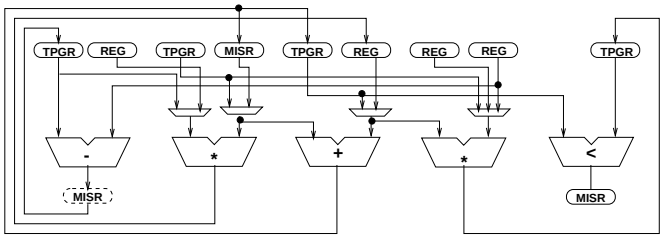

Fig. 9. Datapath from ARYL/LYRA

tural VHDL. ReTest translated the designs, expressed in structural VHDL, to ISCAS89 format. The circuits were then fault graded, before and after *test insertion*, using the HOPE fault simulator [18]. For the initial designs (before the test insertion), all registers at the input ports were configured as TPGRs, the registers at the output ports were configured as MISRs. The remaining registers were configured as BILBOs. The fault simulator was fed random patterns generated using the COMPASS LFSR compiler. Four designs were attempted for this paper, all derived from the literature. Detailed results summary are shown in Table VI.

A.1 Example 1: ARYL and LYRA

The first attempted example was synthesized from behavioral description using ARYL and LYRA [13]. The data path for this example is shown in Figure 9 with the selected test registers. Only one register was inserted for this example in order to improve the testability of the subtractor. The datapath was scheduled in three test sessions. The schedule for this example is shown in Table V. The fault coverage of the redesigned circuit improved to 96.41%, from 87.94%, an improvement of 10.92%. The fault simulation time improved from 9.73 seconds to 6.67 seconds. Fault simulation results are shown in Figure 10.

A.2 Examples 2 and 3: Differential Equation

The second example is the HAL differential equation popularized by Paulin [21]. Two designs that were generated by [3], [22] are presented. One register was inserted in the first design improving its testability from 58.53% to 98.11%. The simulation time also improved from 39.7 secs to 4.41 secs. The example was scheduled in four test sessions. For the second design, one register was also inserted. The fault coverage improved from 87.97% to 98.71%; however, the fault simulation time did not improve by much (only 0.6 secs). The example was scheduled in three test sessions. Fault simulation results for [22] are shown in Figure 11.

Session	ALU	TPGR1	TPGR2	MISR
1	ALU_2	Reg_7	Reg_1	Reg_2
1	ALU_5	Reg_9	Reg_7	Reg_{10}
2	ALU_1	Reg_7	Reg_1	Reg_i
2	ALU_3	Reg_3	Reg_7	Reg_2
3	ALU_4	Reg_7	Reg_3	Reg_i

TABLE V
FINAL TEST PLAN FOR THE ARYL/LYRA EXAMPLE

A.3 Example 4: TMS32010

The last example is the TMS32010, used by [16] as well as by other researchers for comparison purposes. Two registers were inserted improving the fault coverage by 71.24%, from 25.97% to 97.21%. The fault simulation time improved by 18.05 seconds, from 22.81 sec to 4.77 sec. The example was scheduled in two test sessions. Fault coverage results are shown in Figure 12.

VI. Conclusion

A method for redesign for testability at the register-transfer level was presented. The method improves circuits testability through *test registers insertion*. The method is followed by a *test selection* and a *test scheduling* algorithm. What distinguishes our approach is the use of *functional test metrics* in order to tradeoff datapath area and delay with test quality expressed in terms of fault coverage and test time. The method was implemented and several benchmarks circuits were attempted. The results show clear improvemnet in test time as well as in fault coverage.

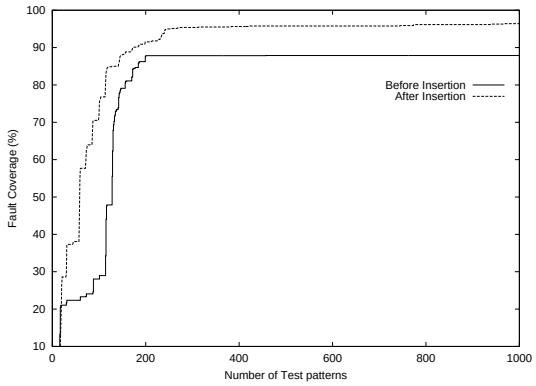

Fig. 10. Fault simulation results using ARYL/LYRA

Example	#Registers Inserted	Test Register Types			# Test Sessions	Fault Coverage		Test Time Improvement (s)
		TPGR	MISR	BILBO		Initial	Improved	
ARYL/LYRA [13]	1	4	3	0	3	87.94%	96.41%	3.1
Differential Equation [3]	1	4	2	1	3	87.97%	98.71%	35.29
Differential Equation [22]	1	11	3	5	4	58.53%	98.11%	0.6
TMS32010 [16]	2	1	3	1	2	25.97%	97.21%	18.05

TABLE VI
RESULTS SUMMARY

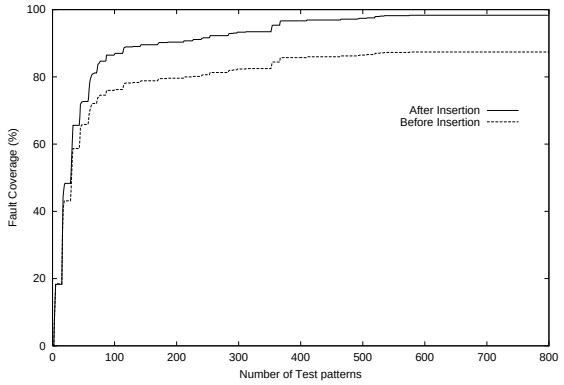

Fig. 11. Differential Equations fault simulation results using Splicer

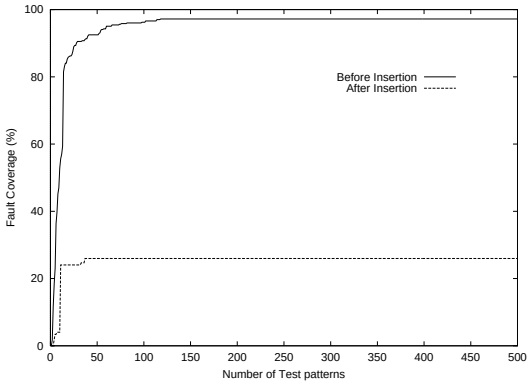

Fig. 12. Fault simulation results using TM32010 Example

REFERENCES

[1] M. Abramovici, M. Breuer, A. Friedman, *Digital Systems Testing and Testable Designs*, Computer Science Press, 1990.

[2] L. Avra, "Allocation and Assignment in High-Level Synthesis for Self-Testable Data Paths," *Proc. ITC*, pp. 463-472, 1991.

[3] S. Chiu, C. Papachristou, "A Design for Testability Scheme with Applications to Data Path Synthesis," *Proc. 28th DAC*, pp. 271-277, 1991.

[4] G. Craig, C. Kime, and K. Saluja, "Test Scheduling and Control for VLSI Built-In Self-Test," *IEEE Trans. On Computers*, Vol. C-37, pp. 1099-1109, 1988.

[5] G. De Micheli, *Synthesis and Optimization of Digital Circuits,* McGraw Hill, 1994.

[6] M. Fernandez, P. Sanchez, E. Villar, "High-Level Synthesis with Testability Criteria," *Proc. Second Annual Atlantc Test Workshop,* pp. 381-390, 1993.

[7] M. Flottes, P. Pires, B. Rouzeyre, "Analyzing Testability from Behavioral to RT Level," *Proc. ED&TC*, pp. 1-8, 1997.

[8] C. Gebotys, M. Elmasri, "VLSI Design Synthesis with Testability," *Proc. 25th DAC,* pp. 16-21., 1988.

[9] M. Gentil, A. ElRhalibi, C. Durante, "A New High Level Testability Measure: Description Evaluation," *Proc. ED&TC*, 1994.

[10] M. Garey, D. S. Johnson, *Computer and Intractability*, W. H. Freeman, 1979.

[11] H. Harmanani, C. Papachristou, "An Improved Method for RTL Synthesis with Testability Trade-Offs," *in Proc. of the ICCAD,* pp. 30-37, 1993.

[12] C.L. Hudson, G.D. Peterson, "Parallel Self-Test With Pseudo-Random Test Patterns," *Proc. ITC*, pp. 954-963, 1987.

[13] C. Huang, Y. Chen, Y. Lin, Y. Hsu, "Data Path Allocation Based on Bipartite Weighted Matching," *Proc. 27th DAC*, pp. 499-504, 1990.

[14] W. Jone, C. Papachristou, M. Pereira, "A Scheme for Overlaying Concurrent Testing of VLSI Circuits," *Proc. 26th DAC*, pp. 531-536, 1989.

[15] K. Kim, D. Ha, J. Tront, "On Using Signature Registers as Pseudorandom Pattern Generators in Built-In Self-Testing," *IEEE Trans. CAD*, Vol. 8, pp. 919-928, 1988.

[16] K. Kim, J. Tront, D. Ha, "Automatic Insertion of BIST Hardware Using VHDL," *Proc. 25th DAC*, pp. 9-15, 1988.

[17] K. Kim, J. Tront, and D. Ha, "BIDES: A BIST design expert system," *JETTA*, Vol. 2, pp. 165-179, 1991.

[18] H. Lee and D. Ha, "HOPE: An Efficient Parallel Fault Simulator for Synchronous Sequential Circuits," *Proc. 29th DAC*, pp. 336-340, 1992.

[19] S. Lin, C. Njinda, M. Breuer, "Generating a Family of Testable Designs Using the BILBO Methodology," *JETTA*, Vol. 4, pp. 71-89, 1993.

[20] C. Papachristou, S. Chiu, H. Harmanani, "SYNTEST: a method for SYNThesis with self-TESTability," *Proc. ICCD*, pp. 458-462, 1991.

[21] P. Paulin, J.P. Knight, "Forced-Directed Scheduling for the Behavioral Synthesis of ASIC's", *IEEE Trans. CAD*, pp. 661-679, 1989.

[22] B. Pangrle, "Splicer: A Heuristic Approach to Connectvity Binding," *Proc. 25th DAC*, pp. 536-541, 1988.

[23] A. Stroele, H. Wunderlich, "Hardware-Optimal Test Register Insertion," *IEEE Trans. CAD*, pp. 531-539, 1998.