Deterministic Circular Self Test Path

Ke WEN\textsuperscript{1,2}, Yu HU\textsuperscript{1}, and Xiaowei LI\textsuperscript{1}

\textsuperscript{1}Advanced Test Technology Laboratory, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100080
\textsuperscript{2}Graduate University of Chinese Academy of Sciences, Beijing 100080
\{wenke, huyu, lxw\}@ict.ac.cn

Abstract

CSTP (Circular Self Test Path) is an attractive technique for testing sequential circuits in the nanometer era because it can easily provide at-speed test. But the area overhead of CSTP is high if random-pattern-resistant faults need to be reliably tested. This paper presents a deterministic CSTP (DCSTP) structure that embeds pre-computed test data with predictable area overhead. Experimental results on ISCAS’89 benchmarks indicate that complete fault coverage can be obtained with low area overhead, especially for large circuit.

Keywords: Circular Self Test Path, Deterministic, State Skipping

1. Introduction

Along with the scale and speed of integrated circuits continue to increase in the nanometer era, built-in-self-test approaches that can provide high fault coverage with low area overhead and performance penalty are needed. In the literature, various BIST techniques yielding high fault coverage were proposed. Most of the techniques embedded the pre-computed test data \( T_D \) in an on-chip BIST circuit. Since \( T_D \) was generated by an ATPG (Automatic Test Pattern Generator) tool, the fault coverage is deterministic. This is why such BIST techniques are called deterministic BIST. The circuit that regenerates \( T_D \) can be broadly classified to two categories: ROM-based (Read Only Memory) and PRPG-based (Pseudo Random Pattern Generator).

A straightforward way to use ROM is to store \( T_D \) in ROM. However, it is impractical in the cases of huge test data volume as the area overhead of ROM is intolerable. \cite{1} proposed to compress \( T_D \) with Huffman coding or Comma coding firstly, and then stored the encoded test data \( T_E \) in ROM. During self-testing process, \( T_E \) was decoded by a finite-state machine (FSM). Because the data volume of \( T_E \) was much smaller than that of \( T_D \), the area overhead of ROM and FSM was less than the area overhead of directly storing \( T_D \) in ROM.

PRPG-based deterministic BIST is more popular than ROM-based deterministic BIST. Among the PRPG-based deterministic BIST techniques, there are two subcategories, according to whether the PRPG is additional circuitry or not.

When using additional circuitry such as LFSR (Linear Feedback Shift Register), counter and CA (Cellular Automation) as PRPG, extra control circuitry is added to alter the pseudo random test data to deterministic test data. Approaches using LFSR include LFSR-reseeding \cite{2}, Bit-flipping \cite{3}, Bit-fixing \cite{4}, deterministic BIST with Reconfigurable Interconnection Network \cite{5}, EDT \cite{6} and DBIST \cite{7}, while \cite{8} using both folding counter and LFSR. To address the memory consumption and computing time issues of Bit-flipping and Bit-fixing, which were usually cubical even exponential, \cite{9} proposed a BDD-based (Binary Decision Diagram) algorithm to alter the pseudo random test data. To reduce the number of seeds loaded in the LFSR or CA, \cite{10} optimized the reseeding procedure by ordering and encoding seeds. For the above mentioned schemes whose test stimulus (test response) needs to be serially shifted in (out) scan chains, they are test-per-scan BIST.

CBIST (Circular BIST) \cite{11} and CSTP (Circular Self Test Path) \cite{12} belong to the other subcategory that utilizes the on-chip memory elements such as flip-flops and latches to generate test data. As shown in Figure 1(a), the memory elements, primary inputs and primary outputs were firstly replaced by CSTP scan cells \cite{12} shown in Figure 1(b), and then the scan cells were connected to form a long circular chain. In the BIST mode, a scan cell was fed by the XOR (exclusive-OR) of its normal functional input and the output of the scan cell that preceded it in the chain. The dashed line shows the circular chain in Figure 1(a). For CBIST and CSTP, a test stimulus was generated by a previous test response. Because the test stimulus is generated every test cycle, CBIST and CSTP is test-per-clock BIST. The circular chain is a feedback shift register which acts both as the response compactor and the random pattern generator. \cite{13} proposed a state skipping technique for CBIST, which broke out limit cycles and test pattern correlations, and transformed present test response to a deterministic test stimulus. Since the insertion of state
skipping logic is hill-climbing, CBIST with state skipping can systematically provide high fault coverage.

Although the area overhead of the state skipping logic in [13] was low as shown in its experimental results, it will deteriorate when the scale of the circuit-under-test increases. In this paper, we propose a deterministic CSTP (DCSTP) structure to obtain high fault coverage with predictable low area overhead. With the proposed structure, not only the fault coverage, but also the area overhead is predictable. And the procedure to implement the DCSTP is not time-consuming, so it can be applied to large circuit.

This paper is organized as follows. Section 2 presents preliminary concepts of CBIST and state skipping, and then we theoretically analyze the area overhead of the technique proposed in [13]. In Section 3, the proposed DCSTP structure along with the state skipping insertion procedure is described, and then its area overhead and computing complexity are analyzed. Experimental results on ISCAS’89 benchmark circuits are reported in Section 4. Finally, Section 5 concludes the paper.

2. Preliminary

CBIST and CSTP have the advantages of shorter test time, lower area overhead, and simpler control logic than test-per-scan BIST. However, there are three structural constraints that limit the fault coverage of CBIST and CSTP [14]:

1) Limit cycling

2) Test pattern correlation

3) Random-pattern-resistant faults

Approaches such as flip-flop reordering [14], initial state selection [15][16] can target one or two of the three constraints, but the remained constraints still limit the fault coverage. In order to reliably yield high fault coverage, [13] proposed a state skipping design named CBIST to overcome the three constrains systematically.

Figure 1(a) The architecture of CSTP/CBIST

Figure 1(b) The CBIST Cell [11]

2.1. CBIST with State Skipping

The main idea of [13] is to add a small amount of “state skipping logic” that causes the circular chain to skip from a state that can not detect any fault to a state which can detect new faults. Meanwhile, the state skipping logic can also break the correlation between test patterns and jump to states that detect random-pattern-resistant faults.

Figure 2 shows an example of a circular chain with state skipping logic. The grayed XOR gates are inserted for state skipping, the AND gate with two inputs Q_3 and Q_4 is the decoding logic that activates the state skipping XORs. For normal CBIST without inserting state skipping logic, when the circular chain reaches state 1101, its next state is 0101. However, in order to break limit cycling, break test pattern correlation or go to a state that can detect new random-pattern-resistant faults, the next state is expected 0011. Therefore, the two-input AND gate and the two grayed XOR gates are inserted. When the circular chain reaches the state 1101, the output of the AND gate is 1, which makes 0011 as the skip-to state. And 0101 is the skipped state.

To find the longest state sequence that precedes the skipped state, [13] describes the relationship between the preceding states and the skipped state with a Boolean conflict matrix. Then the inputs of the AND gate are obtained by solving the minimum column cover problem. Unfortunately, the minimum column cover problem is NP-complete. Therefore, heuristic technique is proposed in [13] to achieve high fault coverage. However, the area overhead is relatively high. We will theoretically analyze
2.2. Area Overhead Analysis

Assume there are \( N \) flip-flops in the circuit-under-test, state skipping happens \( M \) times. The skipped state is denoted as \( S \), and its direct preceding state is \( P \). To reach the state \( P \), we need to run BIST \( L+1 \) cycles.

The area overhead is increased by the decoding logic and the inserted XOR gates. The decoding logic consists of multiple-input AND gates. As the multiple-input AND gate must output 1 at the state \( P \) but output 0 at any states preceding \( P \), the AND gate is the product of \( \alpha \) literals, where \( \alpha \in [0, \min(NL), N] \). A literal is a Boolean variable or its complement. For \( M \) times of state skipping, totally \( M \) number of AND gates is needed.

For the area overhead of the inserted XORs, since one time of state skipping needs to insert one XOR gate for every flip-flop that will change its state, \( \beta \) XOR gates will be inserted in the circular chain, here \( \beta \in [1, N] \). Therefore, during the test procedure, there are totally \( \gamma \) XORs inserted, here \( \gamma \in [M, NM] \).

Therefore, the area overhead can be evaluated as \( M\alpha+\gamma \), which is in the interval of,

\[
M\leq \text{area overhead} \leq (\min(NL)+N)M \leq 2NM \quad (1)
\]

3. Deterministic CSTP

3.1. Deterministic CSTP Structure

Although CBIST with state skipping [13] can achieve high fault coverage, its area overhead is relatively high. Hence we propose DCSTP to reduce the area overhead while keep the high fault coverage. DCSTP also embeds the pre-computed test data in the BIST logic, which is similar to other deterministic BIST techniques. DCSTP generates pseudo random test patterns with DCSTP chain. When new faults cannot be detected, the state of the DCSTP chain jumps from the pseudo random state to a deterministic state. We design a decoding logic for state skipping. Figure 3 illustrates the DCSTP structure.

In Figure 3, the decoding logic part consists of a counter and a state decoder. The counter records how many states the circuit has transited. The state decoder decodes the current state of the circuit. The outputs of the State Decoder are minterms of the outputs of the counter. For example, \( \text{Sel}_2 = \overline{C_3} \cdot C_2 \cdot C_1 \) is the minterm of \( C_1, C_2 \) and \( C_3 \). The number of the state decoder outputs depends on the number of skipped states. When different states need the same DCSTP cell flips to transit, the selection signals are ORed to activate the XOR gate. For example, \( \text{Sel}_1 \) and \( \text{Sel}_2 \) are ORed to activate the XOR gate connected to \( Q_1 \).

An example is given to illustrate how the DCSTP works. The circuit is shown in Figure 3, and the state sequence of CSTP and DCSTP is shown in Figure 4. The first column is the index of the state sequence, and the second column is the state sequence running in CSTP. The circuit of CSTP will run into a cycle at the fourth pattern, and can not detect any more faults. But DCSTP can activate state skipping logic to jump out of the cycle and then detect more faults. For the third state, the output of the counter is 011, so the decoding logic outputs 11 to let the circuit jump out of the cycle to detect new faults.

In the second column of Figure 4, \( \text{Sel}_1 \) and \( \text{Sel}_2 \) are given for each state. \( \text{Sel}_1 \) is the value of \( D_i \) when \( B_0 = 0 \), and \( \text{Sel}_2 \) is the value of \( D_i \) when \( B_0 = 1 \). For example, when \( B_0 = 1 \), \( \text{Sel}_2 \) is 1011, so the XOR gate is activated.

Figure 5 shows an implement of the DCSTP cell with the decoding logic. The operation mode of the DCSTP cell with decoding logic is described in Table 1. The “X” entry means the value of the control signal is don’t-cared. We can see in DCSTP mode, when the decoding logic asserts \( \text{Sel}_i \), the value of \( D_i \) is \( \overline{Q}_{i-1} \odot Z_i \), hence the state sequence is altered by the decoding logic.

The DCSTP chain is tested by shifting 0011 sequence through the DCSTP chain in Shift mode. The Counter is used only in the Normal DCSTP and Transitional DCSTP.
mode, not in the System mode. It doesn’t influence the function of the circuit. So we just ensure that the Counter works well in its function during BIST operation including both the Normal DCSTP and Transitional DCSTP mode. If the counter does not function correctly, the State Decoder would decode the state wrongly, and would change the state sequence of the DCSTP chain. The State Decoder and the ORs fed by the outputs of the State Decoder can be test by itself during BIST operation. If there is a stuck-at fault either at the input or output of the State Decoder or ORs, it would change the normal state sequence of the DCSTP chain. After final state of the DCSTP chain is shifted out, the fault will be detected.

Table 1 Operation mode of DCSTP cell with decoding logic

<table>
<thead>
<tr>
<th>B₀</th>
<th>B₁</th>
<th>Sel</th>
<th>Z</th>
<th>Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>Reset</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>X</td>
<td>Z₁</td>
<td>Shift</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>X</td>
<td>Z₁</td>
<td>System</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>Q₁,₁ ∘ Z₁</td>
<td>Normal DCSTP</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Q₁,₁ ∘ Z₁</td>
<td>Transitional DCSTP</td>
</tr>
</tbody>
</table>

Compared with conventional CSTP or circular BIST, the DCSTP does not add any delay in the functional path, because the state skipping logic is added in the DCSTP chain, not in the functional path.

Next we will demonstrate DCSTP reduces area overhead by the decoding logic design.

3.2. Area Overhead Analysis

Let \( U \) denotes the total number of states the circuit transited in the DCSTP procedure, to record these \( U \) states, we use a \( v \)-bit counter, where \( v = \lceil \log_2 U \rceil \). To decode each state skipping address given by the counter, a multiple-input AND gate is needed, which is the production of \( v \) literals. For \( M \) times state skipping, the number of AND gates is \( M \). In addition, there are \( \eta \) multiple-input OR gates that let different states activate the inserted XORs, here \( \eta \in [0, N] \). The OR gate is the production of at most \( M \) literals.

For the area overhead of the inserted XORs, DCSTP adds one XOR in front of the state skipping flip-flop. As the total number of the flip-flops is \( N \), \( \mu \) XOR gates are inserted in the circular chain, here \( \mu \in [1, N] \).

Therefore, the area overhead can be evaluated as \( c + Mv + \eta \mu \), which is in the interval of

\[
c + Mv + 1 \leq \text{area overhead} \leq c + Mv + MN + N
\]  

(2) is more tight than (1). Therefore, given a circuit design, the area overhead of DCSTP is easier to predictable than CBIST with state skipping. Note (1) has a smaller lower bound than (2), which means CBIST with state skipping can achieve less area overhead than DCSTP in some cases. But for the larger scale circuits, the upper bound of (1) will be greater than (2), which means DCSTP will predictably result in less area overhead along with increasing circuit scale.

Figure 6 illustrates an example of the area overhead intervals given in (1) and (2). Assume \( N \) is less than \( L \) in formula (1). State skipping happens 40 times. The counter is 16-bit, which is equivalent to 112 literals when synthesized with a 0.18\( \mu \)m CMOS library. We can see along with increasing \( N \), the upper bound of DCSTP area overhead is much less than that of CBIST with state skipping.

![Figure 6 Example of the area overhead intervals](image)

3.3. Procedure of state selection

From above mentioned theoretical analysis, the value of \( M \) and \( v \) impacts the area overhead. Both \( M \) and \( v \) is decided by the selection of skipped state and the skip-to state. In this subsection, the procedure of state selection is presented.

The main idea is to do fault simulation until there are \( w \) continuous states that do not detect any new faults, and then one state in the \( w \) continuous states is chosen to skip to a new state that can detect new faults. The decoding logic and a XOR gate are added after the fault simulation. The procedure is described below in detail.

1) Initialize the circuit. Firstly, the circuit-under-test is converted to CSTP form. Then the CSTP cells are initialized by shifting an initial seed in the CSTP chain. The counter is also initialized to 0. Go to 2).

2) Do fault simulation until no faults are detected by \( w \) continuous states, and add one to the counter after each state being simulated. The \( w \) continuous states
that do not detect new faults are not counted in. The continuous states that can not detect any new faults may be caused by limit cycling, correlation in patterns or random-pattern-resistant faults. A set $S = \{s_1, s_2, ..., s_n\}$ is used to record the w continuous states. Go to 3).

3) Select the skip-from state and the skip-to state. The skip-to state is selected from the set $T = \{t_1, t_2, ..., t_p\}$, which consists of the states that can detect new faults. $T$ is selected from the pre-computed test cubes (test data with X bits) generated by ATPG. A state pair $(s_i, t_j)$ presents the state skipping from $s_i$ to $t_j$. So there are $w \times p$ possible choices of state skipping. Here Hamming distance of $(s_i, t_j)$ is defined as the number of different bits between $s_i$ and $t_j$, except for the X bits in $t_j$. To minimize the area overhead, the minimum Hamming distance state pair is selected as the skip-from and skip-to state greedily, because the minimum Hamming distance state pair results in minimum number of flip-flops to skip and minimum overhead per state skipping. Alter the state of DCSTP from the skipped state to the skip-to state. The fault list and set $T$ are updated. The procedure of the state skipping design is hill-climbing. As the state skipping does not influence the faults that have been detected before, there is no need to resimulate the circuit. Go to 4)

4) If the fault coverage is not sufficient, Go to 2), else generate the state skipping logic and terminate the procedure.

The CPU time to implement the DCSTP is the sum of the time to do fault simulation in the BIST operation and the time to generate the state decoding logic. DCSTP and [13] conduct one-pass fault simulation in BIST operation, so they consume approximate time for fault simulation. But DCSTP need less time to generate the state decoding logic. This is because the complexity of generating decoding logic of DCSTP is linear and the generation of the decoding logic of [13] is NP-complete.

4. Experimental Results

We have conducted experiments on several ISCAS'89 benchmark circuits. The circuits are synthesized with a 0.18um digital CMOS standard cell library. Area overhead was estimated by DesignCompiler from Synopsys. The pre-computed test data for stuck-at fault was generated by ATLANTA [17]. All of the circuits were initialized by a random pattern. And the same size of patterns was applied to the CSTP-designed and DCSTP-designed circuit. After the desired fault coverage was achieved, the DCSTP chain scanned out the compacted response. And then the response was compared with the response of the fault-free circuit.

Table 2 shows the comparison of our DCSTP with the approach in [13]. The first column lists the name of the circuits. The “Pattern length” columns indicate the number of BIST patterns for these test technique. The “Coverage (%)” columns are the fault coverage of these test techniques. The fault coverage is for detectable faults. Both CBIST with state skipping and DBIST achieved 100% fault coverage. For most of the smaller circuits, the pattern length of CBIST with state skipping were shorter than that of DCSTP, but for larger circuits, from s526 to s13207, DCSTP needs less patterns to achieve 100% fault coverage.

The “Overhead (%)” column under DCSTP shows the area overhead that DCSTP more than CSTP. As the overhead is DCSTP vs. CSTP, the increased overhead of DCSTP is resulted by the decoding logic. For small scale circuits, the area overhead of DCSTP was higher than CBIST with state skipping. It shows our approach is not good for the small circuit, because the overhead of the counter in the decoding logic will be very high in the small circuit. However, from s5378 to s13207, the area overhead of DCSTP was less than or equal to CBIST with state skipping. Moreover, for the larger ISCAS'89 circuits such as s13207, s15850, s35932, s38417 and s38584, DCSTP steadily required less area overhead.

5. Conclusions

In this paper we presented a deterministic CSTP structure that provides the capability to provide high fault coverage with reasonable area overhead. A new decoding logic was proposed to reduce the area overhead. As the procedure to implement the DCSTP is not time-consuming, the DCSTP can be applied to large circuit. Experimental results on the ISCAS'89 benchmark circuits showed that the approach provided high fault coverage with reasonable area overhead.

6. Acknowledgement

This work was supported in part by National Natural Science Foundation of China under Grant No. 90607010 and 60576031, and in part by the Institute of Computing Technology, Chinese Academy of Sciences, under Grant No. 20056330 and 20056600-16.

7. References

[5] Li Lei, Chakrabarty Krishnendu, "Test set embedding for


---

**Table 2 Comparison of fault coverage and area overhead with [13]**

<table>
<thead>
<tr>
<th>Circuit Name</th>
<th>[13]</th>
<th>Our work</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CBIST</td>
<td>CBIST with state skipping</td>
</tr>
<tr>
<td></td>
<td>Pattern length</td>
<td>Coverage (%)</td>
</tr>
<tr>
<td>s208</td>
<td>1187</td>
<td>92.6</td>
</tr>
<tr>
<td>s298</td>
<td>650</td>
<td>99.2</td>
</tr>
<tr>
<td>s344</td>
<td>25</td>
<td>77.4</td>
</tr>
<tr>
<td>s382</td>
<td>4735</td>
<td>95.8</td>
</tr>
<tr>
<td>s420</td>
<td>50K</td>
<td>92.0</td>
</tr>
<tr>
<td>s510</td>
<td>2835</td>
<td>98.4</td>
</tr>
<tr>
<td>s526</td>
<td>8559</td>
<td>98.6</td>
</tr>
<tr>
<td>s641</td>
<td>50K</td>
<td>98.5</td>
</tr>
<tr>
<td>s1196</td>
<td>50K</td>
<td>98.9</td>
</tr>
<tr>
<td>s1423</td>
<td>46K</td>
<td>100</td>
</tr>
<tr>
<td>s5378</td>
<td>50K</td>
<td>86.0</td>
</tr>
<tr>
<td>s9234</td>
<td>50K</td>
<td>93.7</td>
</tr>
<tr>
<td>s13207</td>
<td>50K</td>
<td>99.0</td>
</tr>
<tr>
<td>s15850</td>
<td>44K</td>
<td>95.0</td>
</tr>
<tr>
<td>s35932</td>
<td>44K</td>
<td>95.0</td>
</tr>
<tr>
<td>s38417</td>
<td>83K</td>
<td>97.1</td>
</tr>
<tr>
<td>s38584</td>
<td>73K</td>
<td>95.8</td>
</tr>
</tbody>
</table>

__: the data is not provided in [13]