A Tree-Structured LFSR Synthesis Scheme for Pseudo-Exhaustive Testing of VLSI Circuits [1]

W. B. Jone, J. C. Rau, S. C. Chang, and Y. L. Wu*

Department of CS & IE, National Chung-Cheng University, Chiayi, Taiwan, R.O.C

Department of CS & CE, The Chinese University of Hong-Kong, Shatin, N.T., Hong Kong*

Abstract

This paper presents a new test architecture, called Tree-LFSR/SR, to more effectively generate pseudo-exhaustive test patterns for combinational VLSI circuits. Instead of using a single scan chain, the proposed test architecture routes a scan tree driven by the LFSR to generate all possible input patterns for each output cone. The new test architecture is able to take advantages of both signal sharing and signal reuse. The benefits are: (1) the hardware overhead can be greatly reduced by saving routing area and XOR circuits, and (2) the difficulty of test architecture synthesis can be eased by accelerating the searching process of appropriate residues. The Tree-LFSR/SR configuration is then extended, if necessary, by adding XOR networks to deal with more complex input-output relations. An efficient method to directly synthesize the XOR network is also included. Experimental results obtained by simulating combinational benchmark circuits are very encouraging.

1. Introduction

Basically, Built-In Self-Testing (BIST) of logic circuits can be divided into two major categories: *random* testing and *exhaustive* testing [1][2][3]. Exhaustive testing is performed by employing a linear feedback shift register (LFSR) to apply all possible input combinations to the circuit under test (CUT), and then observing the output responses. Since all input combinations are applied, no fault model is required to estimate the fault coverage. However, test time might be intolerable if the number of circuit inputs is large, and circuit partitioning which incurs extra hardware and performance degradation is unavoidable. Exhaustively testing a circuit by partitioning to reduce the test time is called *pseudo-exhaustive* testing [4]-[6].

To achieve the goal of pseudo-exhaustive testing, it is sometimes necessary to partition the CUT by inserting *bypass storage cells* (*bscs*) – which are transparent in the normal mode and act as pseudo-inputs and pseudo-outputs in the test mode – such that each output cone depends on no larger than k (≤ 25, mostly) inputs during the test process. For example, if there is an (n, m, r) circuit where n (m) is the number of inputs (outputs) and r is the maximal dependency of the CUT, then s bscs can be inserted into the circuit that becomes $(n+s, m+s, k)$ in the testing mode with $r \geq k$. Basically, circuit partitioning methods for pseudo-exhaustive testing can be categorized to two classes: constrained partitioning strategy (CPS) and un-constrained partitioning strategy (UPS). In the case of CPS, the CUT is partitioned into several disjoint segments [7]. However, it is not necessary that the segments are disjoint for UPS [8]-[11]. Strategies to find the minimal number of bscs for the purpose of partitioning the given circuit as pseudo-exhaustive testable can be found in [7]-[11].

After circuit partitioning, it is possible to pseudo-exhaustively test the entire circuit using several test sessions where each test session applies at most 2^k patterns to exhaustively exercise some of the output cones. Hence, the entire test length is the summation of each test length used in all test sessions. Certainly, if the number of test sessions can be reduced to one, then the test length for pseudo-exhaustively testing the modified $(n+s, m+s, k)$ circuit can potentially be minimal, i.e. 2^k. A straightforward method for reducing the number of test sessions to one is to employ an LFSR/SR of degree k [12]. That is, all PIs and bscs of the CUT are reconfigured as an LFSR/SR chain (in test mode) of degree k for generating all possible 2^k patterns to pseudo-exhaustively test the circuit in one test session. Subject to the complex input-output relations, it is not an easy task to construct a desirable LFSR/SR chain for the CUT. Some variants of LFSR/SR such as LFSR/XOR [13]-[15], reconfigurable LFSR/SR [16], permuted LFSR/SR [16], convolved LFSR/SR [16], and cell-reordering LFSR/SR [17], constant-weight LFSR [18], linear-code LFSR [19], condensed LFSR [20], and cyclic-code LFSR [21] have been proposed to alleviate this difficulty. Though encouraging results have been re-

[1] This work was partially supported by National Science Council, Taiwan, R.O.C., Contract NSC 87-2215-E-194-007

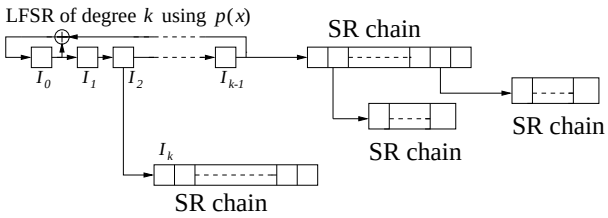

Fig. 1: A Tree - LFSR/SR test architecture.

ported using the above approaches, high hardware overhead or very long derivation time for primitive polynomial selection and test architecture synthesis might occur for some circuits.

In this work, we propose a test architecture, called Tree-LFSR/SR, to more effectively generate all pseudo-exhaustive test patterns in one test session. By taking advantages of both signal sharing and signal reuse, a *scan tree* is employed to implement the pseudo-exhaustive test generator for the CUT, instead of just using a single scan chain. Based on this test architecture, several tree branches which in fact are constructed using bscs can share the same residues to satisfy the *linear independency* constraint [12]. The results are then extended to include XOR networks, if necessary, to deal with more complicated input-output relations. An efficient method is also provided to construct the required XOR network that is part of the Tree-LFSR/SR test architecture. The proposed Tree-LFSR/SR test architecture has two major advantages: (1) the routing area of scan tree and the XOR circuits are smaller than those of traditional methods, and (2) the derivation time for appropriate residues (and thus the test architecture which is able to pseudo-exhaustively test the CUT) can be greatly reduced because of the tree configuration.

The Tree-LFSR/SR construction method mainly consists of three algorithms: backbone generation, tree growing, and XOR-tree generation. The first two processes are detailed in Sections 2 and 3, respectively. Section 4 gives the XOR-tree generation algorithm that extends the Tree-LFSR/SR architecture to deal with more complicated input-output relations. An effective XOR network synthesis strategy is also provided. Experimental results obtained by simulating combinational benchmark circuits are given in Section 5. Conclusions and future research are presented in Section 6.

2. Backbone Generation

Here, we shall introduce a new pseudo-exhaustive test scheme, called tree-structured LFSR/SR (Tree-LFSR/SR), to overcome the linear dependency (LD) problem encountered in constructing the desirable LFSR/SR test architecture. Assume that output O depends on inputs I_0, I_1, I_2, ..., I_{k-1}, based on the work of [12], O can be exhaustively exercised by an LFSR/SR structure if and only if the k residue classes $[x^{I_j} \ mod \ P(x)]$ are linearly independent over $GF(2)$. Note that I_j is the stage number (counted from left to right at the LFSR/SR structure) of the jth input affecting O, and $P(x)$ is the primitive polynomial used to construct the LFSR. For the purpose of simplification, R_{I_j} is used to represent residue $[x^{I_j} \ mod \ P(x)]$. For example, in Fig. 1, we have $I_0 = 0$, $I_1 = 1$, ..., $I_{k-1} = k-1$, and $I_k = 3$. Further, the residue of I_k in Fig. 1 is equal R_3 since the stage number of I_k is 3. Thus, each output cone must satisfy the LD constraint that is imposed by the input-output relations.

The Tree-LFSR/SR construction is guided by coordinates which give relative position of logic components. Each bsc under consideration (BUC) is connected to a nearest tree branch, if the linear independence (LI) relationship is not violated, to minimize the routing area and circuit delay. For simplicity, both X and Y coordinates are assigned integers, and each of the PIs, POs, gates, and bscs is assigned an ordered pair of coordinates (x, y).

Example 1: Consider the circuit of Fig. 2(a), by inserting bscs, it can be partitioned from $(7, 2, 6)$ to $(10, 5, 4)$. We can assign proper coordinates to all components as shown in Fig. 2(b). From this figure, it can be found that the logic components are evenly distributed to the entire coordinate space in a natural manner. The x-coordinates increase from left to right along the x-axis, while the y-coordinates increase from top to bottom along the y-axis.

Definition 1: The *distance* of two grid points, i and j, is the sum of $|x_i - x_j| + |y_i - y_j|$ where $|x|$ represents the absolute value of x.

Tree-LFSR/SR is a novel pseudo-exhaustive test pattern generation architecture that contains a simple LFSR with several SR chains. Each of the SR chains, L, can recursively grow any number of SR chains from the stages of L, and the entire configuration looks like a tree. Compared with a real tree, the LFSR portion of the Tree-LFSR/SR structure corresponds to the tree root, while the SRs portion to the tree branches. A Tree-LFSR/SR is of degree k if the LFSR is constructed using a primitive polynomial of degree k as shown in Fig. 1. There is possibility that the tree configuration can not resolve the LD problem, so it may be necessary to employ XOR gates to pseudo-exhaustively test all output cones. The way of finding a small number of XOR gates to fulfill the LI requirements will be presented in Section 4.

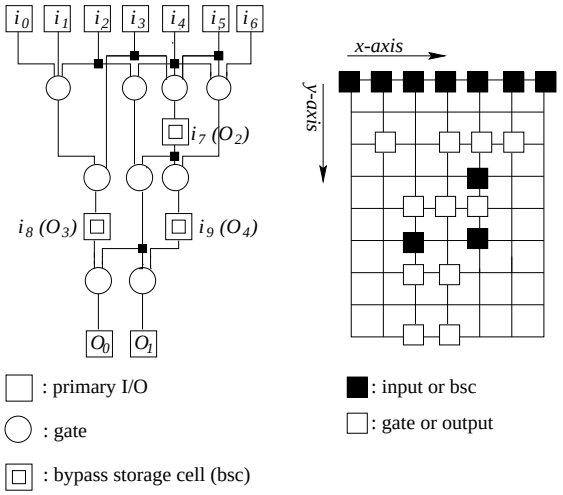

Fig. 2: (a) A (10, 5, 4) circuit ;
(b) The coordinates of the components in (a).

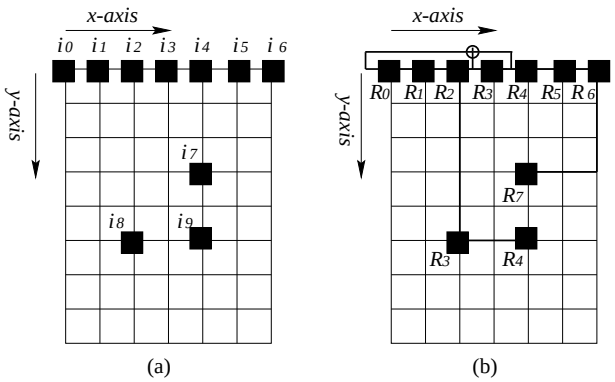

Fig. 3: (a) The simplified version of Fig. 2(b).
(b) Residue assignments and the Tree-LFSR/SR test architecture for (a).

The basic idea behind the Tree-LFSR/SR test scheme can be well explained using the following example.

Example 2: Consider the $(10, 5, 4)$ circuit in Fig. 2(a) and the corresponding coordinates of the circuit components in Fig. 2(b). To concisely demonstrate the tree configuration process, the gates and primary outputs have been removed from the circuit, and only primary inputs and bscs remain as shown in Fig. 3(a). For pseudo-exhaustively testing the circuit, we can construct a Tree-LFSR/SR of degree 4 with primitive polynomial $P(x) = x^4 + x^3 + 1$ as shown in Fig. 3(b). It is obvious that i_3 and i_8 share the same test signal, since both of them are the next stage of i_2, that is, i_2 is repeatedly attached. More interestingly, both i_4 and i_9 also have the same residue R_4 under this configuration. With the residues assigned, no LI violation occurs and the entire circuit of Fig. 2(a) can be pseudo-exhaustively tested.

Algorithm 1: $Backbone(C, P)$
C : an (n, m, k) circuit with component coordinates;
P : a primitive polynomial of degree k;
begin
  assign residues R_0 through R_{k-1} to primary inputs $(0, 0)$ through $(k-1, 0)$, respectively;
  link primary inputs $(0, 0)$ through $(k-1, 0)$ as an LFSR which forms the initial backbone B;
  $tail$ = primary input $(k-1, 0)$;
  while (there exists an untagged input, h, such that $h_y \geq tail_y$) **begin**
    $tail_{new}$ = select a non-routing-back and untagged isolated input that is shortest to $tail$;
    if $tail_{new}$ = NULL **then return** B;
    else begin
      if LIcheck $(tail_{new}, P)$ = LEGAL **then begin**
        add $tail_{new}$ to B;
        $tail = tail_{new}$;
        remove all tags;
      end else tag $tail_{new}$;
    end
  end
  return B;
end

The entire backbone generation process is described in Algorithm 1, and the basic idea can be described by Example 3 followed.

Example 3: Consider the $(10, 5, 4)$ circuit shown in Fig. 2(a). Again, we use primitive polynomial $P(x) = x^4 + x^3 + 1$ as the LFSR feedback function. The example is to show the backbone generation process by applying Algorithm 1. First, residues $R_0 - R_3$ are assigned to inputs $i_0 - i_3$ and the resultant LFSR forms the initial ver-

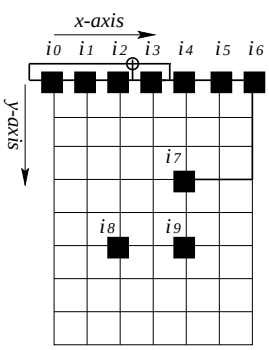

Fig. 4: The backbone of the Tree - LFSR/SR architecture for Fig. 3(a).

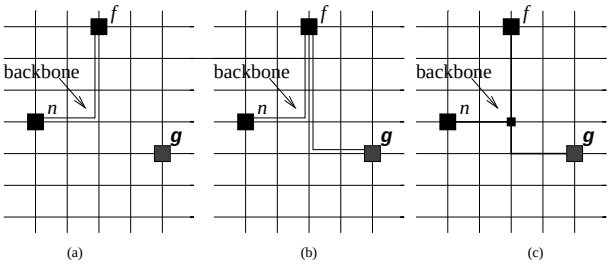

Fig. 5: (a) The backbone and an isolated input g.
(b) Direct connection between g and f.
(c) A more efficient connection between g and f.

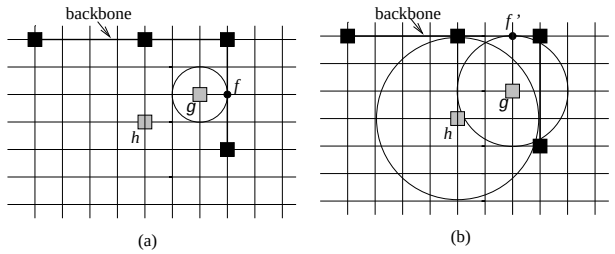

Fig. 6: (a) Hit point determination for g.
(b) The next shortest distance .

sion of the backbone whose tail is i_3 (see Fig. 4). Then, R_4 is assigned to the nearest isolated input, i_4, and LI violation is immediately checked. In this configuration, R_4 can be assigned to i_4 without causing LI violation. Similarly, R_5, R_6, and R_7 can be assigned to i_5, i_6, and i_7, respectively. Now, the new tail is i_7 and the nearest isolated input is i_9. To shorten the routing overhead, it is natural to connect i_9 to the tail, i_7, of the backbone such that i_9 will have residue R_8. Unfortunately, this will cause LI violation because $\{R_2, R_4, R_7, R_8\}$, the input residues of output O_1, are not linearly independent with respect to primitive polynomial $P(x) = x^4+x^3+1$. Thus, i_9 is not allowed to be connected to i_7, and neither is i_8. Finally, the backbone generation process terminates with inputs i_8 and i_9 left isolated as shown in Fig. 4.

3. Tree Growing

As described in Example 3, the backbone generation process may leave some inputs isolated, i.e., they cannot be assigned proper residues. Thus, it is necessary to adopt additional processes to embed these isolated inputs into the backbone in such a way that each input is assigned an appropriate residue so as to pseudo-exhaustively test the entire circuit. The basic idea of tree growing is to connect an isolated input g into a stage f of the backbone (Fig. 5(a)) with two objectives: (1) the routing distance between f and g must be as small as possible, and (2) the addition of g as f's next stage will not cause any LI violation. Thus, this process is similar to branch growing from the trunk of a tree. It should be noticed that, instead of connecting g directly to stage f (Fig. 5(b)), we may attach g to the wire driven by f (Fig. 5(c)) for the purpose of saving routing area.

Initially, the distance from each isolated input to the backbone is calculated as shown in Algorithm 2. Since all circuit components are placed on a two-dimensional coordinate system filled with grid points, the distance from an input g to the backbone is defined as the shortest *linear* distance from g to a grid point f on the backbone as shown in Fig. 6(a). Additionally, grid point f is said to be *hit* by g. And f can be either an LFSR or SR stage, or a wire segment driven by an LFSR or SR stage s. In the latter case, of course, f inherits the residue of stage s.

After the distance for each isolated input has been derived, the consideration focuses on the one, say g, with the shortest distance (than all other isolated inputs) to the backbone at grid point f. This situation is indicated in Fig. 6(a) and Algorithm 2, where α corresponds to g and β to f. If, unfortunately, the added input (g) can not satisfy the LI constraint, then the (f,g) pair must be discarded from the candidate list. The next shortest distance for g must be calculated by excluding both f and its corresponding wire from further consideration as shown in Fig. 6(b), and the entire distance list must be sorted to reflect the newest situation. Again, a pair of isolated input and hit grid point with shortest distance is selected as the candidate for further consideration. The above process is repeated until all isolated inputs have been included into the tree or no inputs can be further considered as shown in Algorithm 2. If no isolated input exists after the tree growing process, then the circuit has been well reconfigured to allow pseudo-exhaustive testing of the entire circuit by one test session. Our experimental results demonstrate that Tree-LFSR/SR structures can be efficiently established to pseudo-exhaustively test most benchmark circuits. However, if the tree growing process ends up with some inputs isolated or intolerably long tree branches, then XOR gates must be employed to eliminate the difficulties as discussed in the next section.

Example 4: As shown in Fig. 4, the previous example (Example 3) terminates by leaving isolated inputs i_8 and i_9 whose distances to the backbone are 4 and 2, respectively. According to the *Treegrowing* algorithm, we have i_9 selected as the next stage of i_7 since they have the smallest distance when compared with other pairs. Unfortunately, growing a branch from i_7 to i_9 causes LI violation and this is not allowed. The distance from i_9 to

Algorithm 2: $Treegrowing(C, P, B, I)$
C: an (n, m, k) circuit with component coordinates;
P: a primitive polynomial of degree k;
B: backbone;
I: isolated input list;
begin
  distance(C); /* calculate the distance
  form each isolated input to the backbone */
  while (I is not empty) **begin**
    (α,β) = shortest();
    /* select the shortest-distance (α,β) pair, where
    α is an isolated input and β is a backbone wire */
    if β = NULL **then return** B and I;
    if LIcheck(α,P) = LEGAL **then begin**
    /* check if α can act as the next stage of β */
      remove α from I;
      link α to β and update B;
      distance(C);
      /* update the distance for each isolated input*/
    end else begin
      remove (α, β) pair out of further consideration;
      distance (C);
    end
  end
  return B and I;
end

the backbone is then updated from 2 to 3 with hit point (5,3) which represents a wire from i_6 (with residue R_6). It appears that connecting i_9 to i_6 is a good choice, unfortunately, LI violation occurs when i_9 is assigned R_7. Once more, the distance of i_9 to the backbone is updated from 3 to 6 which is larger than that (4) of i_8 to the backbone. Thus, i_8 is considered by hitting the backbone at i_7, i.e., residue R_8 is assigned to i_8. However, this assignment also violates the LI constraint. Hence, we change the distance of i_8 to the backbone from 4 to 5 with hit point (2, 0), i.e., i_2. Consider again the updated distances of i_8 and i_9 to the backbone. Clearly, we select i_8 as the next stage of i_2, that is, residue R_3 is assigned to i_8. Fortunately, the new residue assignment does not violate the LI constraint, and a branch has grown from i_2 to i_8. This leads the distance from i_9 to the new backbone equal 2. So, i_9 is connected to i_8 and the LI constraint is not violated, either. Finally, the backbone has grown to a tree as shown in Fig. 3(b), and the goal of pseudo-exhaustively testing the circuit by one test session has been achieved.

4. XOR-Tree Generation and Polynomial Selection

For a given (n, m, k) circuit, probably, we might fail

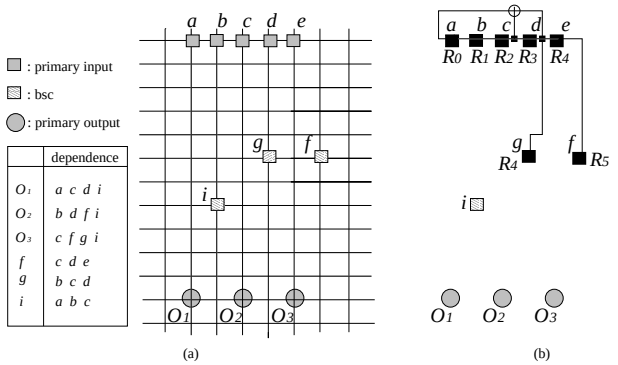

Fig. 7: (a) A (8, 6, 4) circuit.
(b) Node isolation after the backbone and tree-growing process.

to generate the desirable Tree-LFSR/SR by employing a certain primitive polynomial of degree k, i.e., there exist some isolated inputs remaining disconnected after the backbone generation and tree growing processes have been completed. For example, we desire to construct the Tree-LFSR/SR configuration using primitive polynomial, $P(x) = x^4 + x^3 + 1$, for the purpose of pseudo-exhaustively testing the circuit shown in Fig. 7(a). After both backbone generation and tree growing algorithms are applied, the result illustrated in Fig. 7(b) tells that input i is still isolated. Once this situation takes place, the original objective of pseudo-exhaustively testing the (n, m, k) circuit in one test session using 2^k patterns can not be achieved. Sometimes, it is quite difficult to find a primitive polynomial by which we can successfully construct the desirable Tree-LFSR/SR for a given CUT without leaving any inputs isolated. For this reason, we shall not attempt to overcome the problem of rescuing isolated inputs by brutally identifying a primitive polynomial for the CUT, if better alternatives can be used.

Further investigation into the tree-growing failure problem reveals that an input becomes isolated at the end of the backbone generation and tree growing processes, if the input can not act as the next stage of any register node on the generated (incomplete) Tree-LFSR/SR test architecture. Assume that input N is left isolated and the largest (indexed) residue implemented by the Tree-LFSR/SR configuration is R_i. As will be shown later, input N can always be assigned a residue R_j, $j > i$, to resolve the LD problem. Thus, by continuously assigning R_{i+1}, R_{i+2}, ... to N, sooner or later, we will have N given a residue which enables N to satisfy the LI requirement. By this concept, all isolated inputs can be connected into the Tree-LFSR/SR which is then able to pseudo-exhaustively exercise the entire CUT. Intuitively, R_j can be generated and assigned to

N by inserting $d = j - i - 1$ dummy cells between N and the register node (or its wire equivalent) which provides R_i. Unfortunately, the hardware overhead might not be tolerable if d is large, so a better solution by synthesizing R_j using several *existing* stages must be found. The following lemmas and theorems will prove this fact, and a method to effectively synthesize the desirable residue using a limited number of existent register stages will be given.

Lemma 1: Given a simple LFSR/SR chain which is constructed from a primitive polynomial $P(x)$ of degree k, the first $2^k - 1$ stages of the chain generate all possible residues R_j, $0 \leq j \leq 2^k - 2$; that is, the remainders of x^j's divided by $P(x)$ are all different [11].

Many interesting properties can be found, if the residues are spanned into a k-dimensional linear space. By definition, m vector sets, S_1, ..., S_m, are linearly independent provided that each of the S_i contains linearly independent vectors.

Lemma 2: Given two linearly independent vector sets, S_1 and S_2, in a k-dimensional vector space with $r = max(rank(S_1),\ rank(S_2))$, there exists another set containing at most $k - r$ vectors that can be added into S_1 and S_2 without causing S_1 and S_2 to be linearly dependent.

Proof: Omitted. □

The above lemma can be extended to the general case. That is, in a k-dimensional linear space, if we have m linearly independent vector sets, S_1, ..., S_m, and $r = max(rank(S_1),\ ...,\ rank(S_m))$, then there exists a vector set containing at most $k - r$ vectors that can be added into these sets without causing S_1, ..., S_m to be linearly dependent.

Theorem 3: If the largest residue implemented by the Tree-LFSR/SR configuration is R_i and input node N is isolated, then there exists a residue R_j, $j > i$, that can be assigned to N without causing N to be linearly dependent.

Proof: Omitted. □

Theorem 4: Given a residue R_j, $j > k$, resulting from dividing x^j by a primitive polynomial of degree k, there exists a set of residues $\{R_{j_1}, R_{j_2}, ..., R_{j_n}\}$ such that

$$R_j = R_{j_1} + R_{j_2} + ... + R_{j_n}$$

where $0 \leq j_i < k$ for $1 \leq i \leq n$, and "+" stands for modulo-2 addition.

Proof: Omitted. □

From Theorem 4, we know that there indeed exists a residue R_j that can be assigned to the isolated input without causing the LD problem. Intuitively, we only have to insert $d = j - i - 1$ dummy cells between the isolated node N and the register node (or its wire equivalent) providing R_i. However, the hardware overhead might not be torerable if d is too large. In fact, the desirable residue R_j can be synthesized using a subset of the residues taken from the LFSR portion based on Theorem 4. The number of required XORs is one less than the number of non-zero terms consisting of R_j as shown in Theorem 4. Unfortunately, implementing R_j by taking only the combination of LFSR stages into consideration might not be economical, as the SR stages are unnecessarily wasted. Hardware overhead can be further saved if the SR stages are also involved in the R_j substitution process. Based on this concept, an effective residue replacement technique will be developed. Theorem 6 below establishes the cornerstone for further discussions.

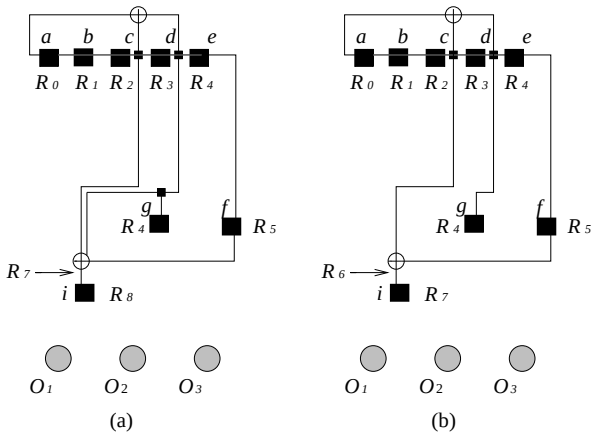

Fig. 8: (a) Residue replacement without considering bsc.
(b) A correct residue replacement with bsc considered.

Lemma 5: Under $GF(2)$, if $x^j + Q(x) \equiv 0 \bmod P(x)$, then $x^j \equiv Q(x) \bmod P(x)$.

Proof: Omitted. □

Theorem 6: Under $GF(2)$, consider a degree-k primitive polynomial, $P(x) = x^k + \Sigma_{i=k-1}^{0} a_i x^i$, where $a_i \in \{0, 1\}$. If residue R_j $(j > k)$ is the remainder of dividing x^j by $P(x)$ and $d = j - k$, then

$$R_j = \Sigma_{i=k-1}^{0}\ a_i R_{i+d}.$$

Proof: Omitted. □

Consider again the circuit shown in Fig. 7(b) with input i remaining disconnected, when primitive polynomial $P(x) = x^4 + x^3 + 1$ is used. Starting from R_7 to obtain a target R_j, $j > 6$, we immediately find that residue R_7 can be assigned to the isolated input i without causing the LD problem. By Theorem 6, the target residue R_7 can be replaced by $R_6 + R_3$ under $P(x) = x^4 + x^3 + 1$,

and we can use both R_6 and R_3 to synthesize R_7. Unfortunately, R_6 is not existent and must be further substituted by R_5 and R_2. Finally, residue R_7 is synthesized by $R_2 + R_3 + R_5$ and assigned to the isolated input i as illustrated in Fig. 8(a). Using the Tree-LFSR/SR configuration with XOR-network added, the circuit can be pseudo-exhaustively tested.

As shown in Fig. 8(a), R_7 has been implemented by an XOR circuit, and register stage i immediately following the output of the XOR network will implement R_8. Thus, the above residue replacement process must be performed by synthesizing an XOR network to realize R_{j-1}, which is the residue immediately preceding the target one. Then, adding a register stage to the output of the XOR network will give the target residue R_j. The example of Fig. 7(a) can be repeated by synthesizing R_6 using existent residues. By applying Theorem 6, we can have R_6 synthesized by taking the module-2 addition of both R_5 and R_2 which are already existent. If the output of the XOR network is connected to a register stage whose output is then fed to input node i, the target residue R_7 is accurately generated as shown in Fig. 8(b). The reason for not directly taking the XOR output is to avoid any circuit instability caused by the XOR tree delay. The detail of the process to find a proper residue for an isolated node is given in Algorithm 3.

5. Simulation Results

To evaluate the performance of the proposed tree-structured LFSR synthesis scheme, simulation software has been developed using C as the high-level language on a SUN Sparc workstation. The program implements three major algorithms (backbone, tree-growing, and xor-tree-generation) which are integrated into the TLS (Tree-LFSR/SR) algorithm shown in Algorithm 4. Ten ISCAS85 combinational benchmark circuits are then used to test the program. Before applying the TLS algorithm, we use a quite efficient strategy, similar to [11], to partition the benchmark circuits such that each of them has a maximum dependency smaller than or equal to 20. The circuit partitioning results are given in Table 1, and bscs are added to satisfy the maximum dependency requirement. For example, originally, C432 has 36 inputs and 7 outputs, and its maximum dependency is 36. However, after circuit partitioning, the number of inputs (including the psudo-inputs induced by bscs) has increased to 56, and the number of outputs (including the pseudo-outputs) has increased to 27. Thus, we know that 20 bscs are inserted while the maximum dependency has been successfully reduced from 36 to 20. It should be noted that the number of bscs inserted by

Algorithm 3: $Xor - Tree - Generation(C, P, T, I)$
C: an (n, m, k) circuit with component coordinates;
P: a primitive polynomial of degree k;
T: LFSR/SR tree;
I: isolated input list;
begin
  while (I is not empty) **begin**
    i = the largest indexed residue, R_i, implemented by T;
    distance(); /*calculate the distance from each isolated input to the LFSR/SR tree*/
    N = the isolated input which is nearest T;
    $i = i + 1$;
    while (R_i can not be assigned to N) $i = i + 1$;
    $i = i$ - 1;
    initialize the residue queue (Q) to be empty;
    insert i to Q;
    $R = \emptyset$;
    repeat do
      i = dequeue (Q);
      replace R_i guided by P to get a set of residues, R';
      add elements in R' to R with duplicated residues eliminated;
      if (there exist residues not implemented) **then**
        remove theses residues from R,
        and insert them to Q;
    until (Q is empty);
    assign the modulo-2 addition of the residues in R (after adding a register) to N;
    add N to T and remove N from I;
    if (I is not empty) **then**
      (T,I) = $treegrowing(C,P,T,I)$;
  end
end

Algorithm 4: $TLS(C, P)$
C : an (n, m, k) circuit with component coordinates;
P : a primitive polynomial of degree k;
begin
  $(B, I) = backbone(C, P)$;
  if I is not empty **then**
    $(B, I) = treegrowing(C, P, B, I)$;
  if I is not empty **then**
    $xor - Tree - Generation(C, P, B, I)$;
end

Table 1: Circuit partitioning.

Ckt	(n, m, k)	
	Before Partitioning	After Partitioning
c432	(36, 7, 36)	(56, 27, 20)
c499	(41, 32, 41)	(49, 40, 14)
c880	(60, 26, 45)	(75, 41, 20)
c1355	(41, 32, 41)	(49, 40, 14)
c1908	(33, 25, 33)	(47, 39, 19)
c2670	(233, 140, 122)	(262, 169, 20)
c3540	(50, 22, 50)	(118, 90, 20)
c5315	(178, 123, 67)	(225, 170, 20)
c6288	(32, 32, 32)	(87, 87, 20)
c7552	(207, 108, 194)	(296, 197, 20)

our circuit partitioning method is only slightly different from that of [11] in most cases.

Then, we apply the proposed TLS algorithm to the partitioned benchmark circuits with the results shown in Table 2 where columns 1 and 2 give the circuit names and their basic characteristics. The primitive polynomial employed for each circuit is shown in column 3. Here, we only depict the exponents of non-zero items. Column 4 provides the total number of inputs (including bscs) that are involved in the Tree-LFSR/SR test architecture, while columns 5 and 6 give the numbers of inputs that have been solved by the backbone generation and branch growing processes, respectively. The number of nodes remaining isolated is presented in column 7 and the number of XOR gates required to further embed these isolated nodes into the tree architecture is shown in column 8. The CPU time consumed for executing TLS using **SPARC 10** can be found in column 9 while column 10 gives the number of XOR gates required by [11].

From Table 2, it can be found that our test architecture requires much smaller number of XOR gates for each benchmark circuit when compared with the results of [11]. Potentially, the routing distance of the scan tree should also be smaller than that of the scan chain used by [11] because of the design freedom supported by the special feature of Tree-LFSR/SR. Results also demonstrate that the number of nodes remaining isolated after the backbone generation and tree growing processes is very small, i.e., most nodes will be embedded in the scan tree by the first two processes.

The CPU time consumed by each benchmark circuit simulation is quite small as seen in Table 2, and this can be confirmed by computing complexity analysis for each algorithm utilized. First, the backbone generation algorithm is straightforward and its computing complexity is $O(n)$ where n is the number of both PIs and bscs. Assume n_1 nodes have been included in the backbone while n_2 nodes remain isolated, theoretically, the computing complexity of the tree growing algorithm can be roughly estimated as $O(n_1 \cdot n_2)$. The reason is that each isolated node has the possibility of trying (to hit) all nodes in the backbone (the worst case analysis). Thus, the computing complexity is at most $O(n^2)$. Finally, the computing complexity of the XOR-tree generation algorithm is $O(i \cdot 2^k)$ where i is the number of isolated nodes and k is the degree of the employed primitive polynomial. Fortunately, the exponential (and also extreme) time complexity is rarely required since mostly the target residue R_j can be derived far before the extreme case is reached. Especially, most benchmark circuits can be solved by applying the backbone generation and tree growing processes only. We further analyze the CPU time consumed by C1908, C3540, and C6288 that need to perform the XOR-tree generation process. The results demonstrate that only 23% of the CPU time (0.56 secs) used by C1908 is for the XOR-tree generation, 17% (6.79 secs) by C3540, and 45% (115.6 secs) by C6288. We feel that the entire TLS algorithm is quite computationally cost-effective and its time complexity can be roughly estimated as $O(n^2)$.

Table 2: Simulation results for the LFSR synthesis scheme.

Ckt	(n, m, k)	Polynomial	Inputs	Inputs in backbone	Inputs in branches	Isolated Inputs	XOR	CPU (secs)	XOR [11]
c432	(56, 27, 20)	20 19 4 3 0	56	44	12	0	0	0.56	9
c499	(49, 40, 14)	14 9 8 3 0	49	48	1	0	0	0.54	11
c880	(75, 41, 20)	20 19 4 3 0	75	69	6	0	0	0.64	13
c1355	(49, 40, 14)	14 9 8 3 0	49	47	2	0	0	2.77	11
c1908	(47, 39, 19)	19 6 5 1 0	47	41	4	2	3	2.41	10
c2670	(262, 169, 20)	20 19 4 3 0	262	247	15	0	0	13.74	7
c3540	(118, 90, 20)	20 19 4 3 0	118	72	45	1	6	34.82	27
c5315	(225, 170, 20)	20 19 4 3 0	225	186	39	0	0	75.66	36
c6288	(87, 87, 20)	20 19 4 3 0	87	59	25	3	15	259.37	25
c7552	(296, 197, 20)	20 19 4 3 0	296	216	80	0	0	33.59	31

6. Conclusions and Future Research

We have presented a novel test generation architecture, called Tree-LFSR/SR, to pseudo-exhaustively test combinational networks in one test session. Basically, the proposed test architecture takes advantages of both signal sharing and signal reuse properties to minimize hardware overhead and derivation time of the test architecture. As discussed before, the process of constructing the desirable Tree-LFSR/SR mainly consists of three phases – backbone generation, tree growing, and XOR-tree generation. After the first two phases, there may be isolated inputs that can not be added into the Tree-

LFSR/SR configuration due to LI violation. If this occurs, we may employ XOR gates to overcome the encountered LD problem. We have also developed a method to directly determine the required XOR network, if the Tree-LFSR/SR configuration can not solve the LD problem alone. Experimental results obtained by computer simulation on benchmark circuits demonstrate that the Tree-LFSR/SR test architecture is indeed very powerful in generating pseudo-exhaustive test patterns.

Even though we have developed a very powerful test architecture to pseudo-exhaustively exercise combinational circuits, unfortunately, we do not have efficient strategies to collect output responses and to compress them into signatures that cause the fault effect observed. A good design for an output response analyzer (ORA) must minimize the probability of error masking and faithfully report the test results. So, our future work is to develop a new ORA architecture which, when combined with Tree-LFSR/SR, will pseudo-exhaustively exercise combinational networks and faithfully report the test results. This is also the ultimate goal of built-in self-test (BIST).

References

1. M. Abramovici, M. A. Breuer and A. D. Friedman, *Digital System Testing and Testable Design,* Computer Science Press, New York, 1990.
2. V. D. Agrawal and S. C. Seth, *Test Generation for VLSI Chips,* IEEE Computer Society, 1988.
3. P. H. Bardell, W. H. McAnney and J. Savir, *Built-In Test for VLSI: Pseudorandom Techniques,* John Wiley and Sons, New York, 1987.
4. E. J. McCluskey, "Verification Testing - A Pseudoexhaustive Test Technique", *IEEE Trans. on Computers,* vol. **C-33**, no. 6, pp. 541-546, Jun. 1984.
5. E. J. McCluskey and S. Bozorgui-Nesbat, "Design for Autonomous Test", *IEEE Trans. on Computers,* vol. **C-30**, no. 11, pp. 866-875, Nov. 1981.
6. Z. Barzilai, J. Savir, G. Markowsky and M. G. Smith, "The Weighted Syndrome Sums Approach to VLSI Testing", *IEEE Trans. on Computers,* vol. **C-30**, no. 12, pp. 996-1000, Dec. 1981.
7. W. B. Jone and C. A. Papachristou, "A Coordinate Circuit Partitioning and Test Generation Method for Pseudo-Exhaustive Testing of VLSI Circuits", *IEEE Trans. on CAD,* vol. **14**, no. 3, pp. 374-384, Mar. 1995.
8. D. Kagaris, F. Makedon and S. Tragoudas, "A Method for Pseudo-Exhaustive Test Pattern Generation", *IEEE Trans. on CAD,* vol. **13**, no. 9, pp. 1170-1178, Sep. 1994.
9. M. W. Roberts and P. K. Lala, "An Algorithm for the Partitioning of Logic Circuits", *IEE Proc.,* vol. **131**, pt. E, no. 4, Jul. 1984.
10. S. N. Bhatt, F. R. K. Chung and A. L. Rosenburg, "Partitioning Circuits for Improved Testability", *Proc. of 4th MIT Conf. on Advanced Research in VLSI,* pp. 91-106, Apr. 1986.
11. R. Srinivasan, S. K. Gupta and M. A. Breuer, "An efficient Partitioning Strtegy for Pseudo-Exhaustive Testing", *Proc. of Design Autamation Conf.,* pp.242-248, Jun. 1993.
12. Z. Barzilai, D. Coppersmith and A. L. Rosenberg, "Exhaustive Generation of Bit Patterns in Discontiguous Positions with Applications to VLSI Self-Testing", *IEEE Trans. on Computers,* vol. **C-32**, no. 2, pp. 190-194, Feb. 1983.
13. S. B. Akers, "On the Use of Linear Sums in Exhaustive Testing", *Proc. of 15th Int'l. Symp. on Fault-Tolerant Computing,* pp. 148-153, Jun. 1985.
14. E. Wu and P. W. Rutkowski, "PEST - A Tool for Implementing Pseudo-Exhaustive Self Test", *Proc. of 1st European Design Automation Conf.,* pp. 639-643, Mar. 1990.
15. C.-I. H. Chen and J. T. Yuen, "Automated Synthesis of Pseudo-Exhaustive Test Generator in VLSI BIST Design", *IEEE Trans. on VLSI Systems,* vol. **2**, no. 3, pp. 273-291, Sep. 1994.
16. R. Srinivasan, S. K. Gupta and M. A. Breuer, "Novel Test Pattern Generators for Pseudo-Exhaustive Testing", *Proc. of International Test Conf.,* pp. 1041-1050, 1993.
17. D. Kagaris and S. Tragoudas, "Cost-Effective LFSR Synthesis for Optimal Pseudoexhaustive BIST Test Sets", *IEEE Trans. on VLSI Systems,* vol. **1**, no. 4, pp. 526-536, Dec. 1993.
18. D. T. Tang and L. S. Woo, "Exhaustive Test Pattern Generation with Constant Weight Vectors", *IEEE Trans. on Computer,* vol. **C-32**, no. 12, pp. 1145-1150, Dec. 1983.
19. D. T. Tang and C. L. Chen, "Logic Test Pattern Generation Using Linear Codes", *IEEE Trans. on Computers,* vol. **C-33**, no. 9, pp. 845-850. Sep. 1984.
20. L. T. Wang and E. J. McCluskey, "Condensed Linear Feedback Shifter Register (LFSR) Testing - A Pseudoexhaustive test technique", *IEEE Trans. on Computers,* vol. **C-35**, no. 4, 1986.
21. L. T. Wang and E. J. McCluskey, "Circuits for Pseudo-Exhaustive Test Pattern Generation", *Proc. 1986 IEEE Int. Test Conf.,* pp. 25-37, 1986.