Optimized Reseeding by Seed Ordering and Encoding

Ahmad A. Al-Yamani, *Member*, *IEEE*, Subhasish Mitra, *Member*, *IEEE* and Edward J. McCluskey, *Life Fellow*, *IEEE*

***Abstract*— Mixed mode Logic BIST applies both pseudorandom test patterns and deterministic test patterns (from an ATPG tool) to the combinational portion of the circuit under test. Each scan test cycle consists of (1) shifting a test pattern into the scan chains, (2) capturing the response to that pattern and (3) shifting the captured response out of the scan chains. The shifting of the test pattern out of the scan chains is overlapped with shifting in the next test pattern. The pattern shifted into the scan chains comes from the output of the PRPG (pseudo random pattern generator); this pattern is determined by the initial state or *seed* of the PRPG (contents of the PRPG at the beginning of the shifting operation). In a pseudorandom cycle, the initial state is the final state (last PRPG contents) from the previous cycle. The initial state of a deterministic cycle is shifted into the PRPG either from an ATE or from an on-chip BIST controller.**

This paper describes techniques to minimize the number of deterministic seeds that must be used: the number of seeds determines the required storage either on the ATE or the chip being tested. These techniques interleave pseudorandom and deterministic cycles rather than first applying all of the pseudorandom cycles and then the deterministic cycles. The decision of when to change from a pseudorandom cycle to a deterministic cycle is made by comparing the final state of the pseudorandom cycle with previously generated ATPG patterns or by carrying out fault simulation on the final state. Which deterministic pattern is chosen for the deterministic cycle influences critically the remainder of the test. A methodology for doing this is described. In addition to interleaving test cycles it is possible to use partial cycles in which the PRPG operates for a few clocks without loading the scan chains. This allows a new seed to be present without loading the seed from the ATE or controller. As might be suspected this reduces the number of stored seeds at the penalty of more complexity in the control sequence. These techniques were simulated and compared with conventional reseeding for some ISCAS 89 benchmarks. Improvements varied between 25% and 85% in the required seed storage.

***Index Terms*—Built-In Self-Test (BIST), Reseeding, Seed ordering, Test set compression.**

Manuscript received May 27, 2003. This work was done at Stanford CRC and was supported by LSI Logic under contract No. 0098218, by NSF under contract number CSL-FY00-28 and by King Fahd University of Petroleum and Minerals.

A. A. Al-Yamani is with Stanford Center for Reliable Computing, Stanford, CA 94305 USA, phone: 650-723-3801; fax: 650-725-7411; e-mail: alyamani@crc.stanford.edu.

S. Mitra is with Intel Corporation (e-mail: subhasish.mitra@intel.com).

E. J. McCluskey is with Stanford Center for Reliable Computing, Stanford, CA 94305 USA; e-mail: ejm@crc.stanford.edu.

I. Introduction

In BIST, on-chip circuitry is used to provide test vectors and to analyze output responses. One possible approach for BIST is pseudorandom testing using a linear feedback shift register (LFSR) [1]. Among the advantages of BIST is its applicability while the circuit is in the field and its test data volume reduction.

Many digital circuits contain random-pattern-resistant (r.p.r.) faults that limit the fault coverage of pseudorandom testing [2].The r.p.r. faults are faults with low detectability (Few patterns detect them). Several techniques have been suggested for enhancing the fault coverage achieved with BIST. These techniques are: (1) Modifying the circuit under test (CUT) by test point insertion or by redesigning the CUT [2], [3], (2) Weighted pseudorandom patterns, where the random patterns are biased using extra logic to increase the probability of detecting r.p.r. faults [4], [5] and (3) Mixed-mode testing (aka top-off) where the circuit is tested in two phases. In the first phase, pseudorandom patterns are applied. In the second phase, deterministic patterns are applied to target the undetected faults [6], [7], [8]. The techniques we present are mixed mode techniques that insert deterministic patterns between the pseudorandom patterns.

Modifying the CUT is often not possible due to performance or intellectual property issues. Weighted pseudo-random sequences require multiple weight sets that are typically stored on-chip. Mixed mode testing is done in several ways; one way is to apply deterministic test patterns from a tester. Another technique is to store the deterministic patterns (or their seeds) in an on-chip ROM. There needs to be additional circuitry to apply the patterns in the ROM to the circuit under test.

Another mixed-mode technique is mapping logic [9]. The strategy is to identify patterns in the original set that don't detect new faults and map them by hardware into deterministic patterns. Reseeding refers to loading the LFSR with a seed that expands into a precomputed test pattern. We presented a technique for built-in reseeding (encoding the seeds in hardware) in [10]. The technique combines mapping logic and reseeding, and is based on running the LFSR in pseudorandom mode after every seed load. Engineering changes in the circuit require re-synthesizing the mapping logic of [9] and the built-in reseeding circuit of [10].

The contributions in this paper are: (1) **A seed ordering algorithm** that reduces the number of seed loads and seed storage by up to 80%. The algorithm is based on exploiting the algebraic properties of the PRPG to increase the number of patterns generated from one seed, and hence reduce the seed storage. (2) **A seed encoding algorithm** that encodes a seed in a vector that corresponds to the number of cycles needed to reach it. The encoding vector is much smaller than the seed itself. (3) **A seed matching technique** that tests if a given PRPG state can generate a test pattern or not. The technique is applicable to LFSRs and cellular automata with and without phase shifters. This technique exploits the unspecified bits to improve the ordering and encoding algorithms.

In Sec. II of this paper, we review the related literature. In Sec. III, we present the seed ordering algorithm. The seed encoding technique is presented in Sec. IV. In Sec. V, we present the seed calculation scheme. Section VI shows the simulation results and Sec. VII concludes the paper.

II. Related Work

Reseeding was first presented in [6] by Koenemann as a technique for coding test patterns into PRPGs of size S_{max}+20, where S_{max} is the maximum number of specified bits in the ATPG patterns. By adding 20 to S_{max} as the size of the PRPG, the probability that test patterns with $S \leq S_{max}$ specified bits cannot be coded into seeds drops to 1 in a million [6]. The S_{max}+20 requirement was reduced to S_{max}+4 in [7] using multiple-polynomial LFSRs (MP-LFSRs) to reduce the probability of linear dependence. The techniques presented in this paper are based on [6] but can be modified to rely on [7].

In [11], a reseeding technique was presented to improve the encoding efficiency (specified bits/seed size) by using variable-length seeds together with an MP-LFSR. In [12], the authors presented a scheme where the contents of the LFSR are incrementally modified instead of modifying them all at once. Both techniques achieve higher encoding efficiency than regular reseeding. [13] presented a scheme for eliminating the boundaries between test patterns to improve the encoding efficiency using a simple modification to the BIST architecture. [14] presented a technique for optimizing BIST by ordering the polynomials in MP-LFSRs, which resulted in high encoding efficiency for the seeds.

The above schemes assume that seeds are either applied from an external tester or stored in an on-chip ROM. The technique in [10] presents a scheme to encode the seeds in hardware. Whether seeds are encoded in hardware or stored in a tester or a ROM, reducing the number of seeds will reduce the hardware or storage needed for the seeds.

In [15], a technique was presented for generating multiple patterns from a single seed if the equations corresponding to all of the care bits in the multiple patterns are satisfiable. The embedded deterministic test (EDT) presented in [16] uses incremental (continuous flow as in [12]) decompression to avoid repeated computations and to improve the encoding efficiency. Both [15] and [16] try to control the specified bits generated by the ATPG algorithm to provide efficient encoding, which can be utilized to improve most of the previous work on reseeding.

The ordering technique, first presented in [17], exploits the algebraic properties of the PRPG and the don't care bits in the patterns to encode the maximum number of patterns per seed. The encoding technique, first presented in [18], tries to encode a given seed by the number of additional clock cycles needed to reach it. In this paper, we discuss both techniques with details and explain their similarities and differences.

The seed ordering technique we present here is different from previous work on reseeding in that it allows a variable number of pseudorandom patterns between the deterministic patterns. The seed encoding technique allows a variable number of clock cycles between those deterministic patterns. Unlike the previous techniques mentioned, the two techniques are not limited by the care bits because pseudorandom patterns are inserted between seed loads as explained in Sec. III and Sec. IV. This means that a seed could be generated for the maximum number of patterns as in [15] and then a few pseudorandom patterns applied to put the LFSR in a state that can generate another pattern. By optimizing the order of the patterns as in Sec. III, this technique can be efficiently used. Also, the techniques in [15] and [16] involve controlling the ATPG patterns generation in a way not available except to tool vendors.

Seed ordering was addressed in [19] with an analytical method for computing a single seed for random pattern resistant circuits based on discrete logarithms. The complexity of the algorithm depends on the number and size of the prime factors of 2^n-1, where n is the LFSR size. A simulation scheme for seed ordering appeared in [20] for calculating initial seeds for LFSRs. The scheme is based on simulating several sequences and picking the one that includes the maximum number of ATPG vectors. The ordering technique we present is a hybrid technique that avoids high complexity and unnecessary simulations.

In [21], a technique for skipping useless patterns is presented. The technique is based on having a Seed Skip Data Storage (SSDS) inside the tester. Fault simulation is performed to identify the useful (fault dropping) and useless (non fault dropping) sequences of patterns. Using additional control logic, the useless patterns are not loaded from the PRPG to the scan chain of the device under test. The SSDS technique reduces the test application time. The problem with this technique is the requirement to store all sequences that don't drop faults and the long test time requirement. Our seed encoding technique allows a trade off between test time and test storage. Also, our encoding technique starts with specific test patterns instead of relying completely on pseudorandom patterns. It also avoids the additional control logic to run the SSDS.

III. Seed Ordering Algorithm

The BIST architecture we assume is shown in Fig. 1. If the PRPG runs in pseudorandom mode after loading the seeds, there is a chance that some of the other seeds may not need to be loaded. This is because the corresponding faults are already covered by the pseudorandom patterns. For such a scheme to work efficiently, we should optimize the order of the seeds to reduce the number of seed loads into the PRPG while covering all deterministic test patterns to reduce seed storage.

Our seed ordering algorithm tries to get the PRPG to go through the states (seeds) that produce the desired test patterns without having to explicitly load such seeds into the PRPG. The algorithm starts with pseudorandom patterns to detect the easy faults and then generates deterministic test patterns for the undetected faults. The algorithm tries to make the PRPG reach to states that can generate some of the deterministic test patterns generated by ATPG. This is done by running the PRPG in pseudorandom mode between deterministic patterns. The additional pseudorandom patterns serve two purposes: (1) they may get the PRPG in a state that can generate some of the deterministic patterns (2) they pseudorandom patterns themselves can detect additional fault because the algorithm applies a capture cycle after each pseudorandom pattern.

The seed ordering algorithm starts by picking the deterministic test pattern with the largest number of specified bits and encoding it into a seed. We refer to the seed as the *initial state* ($s(0)$) of the PRPG. By running the PRPG for m cycles, where m is the length of the longest scan chain, the seed expands into the desired test pattern in the scan chains. We refer to the state of the PRPG after the seed is expanded in the scan chains as the *final state* ($s(m+1)$). If the final state of the PRPG can expand into one of the remaining test patterns, then we don't need to load a seed for that pattern.

Let's start with a PRPG whose characteristic polynomial is: $f(x) = c_n x^n + c_{n-1}x^{n-1} + \cdots + c_1 x + c_0$. Let $s(t)$ be the state of the PRPG at time t. $s(t+1) = s(t) \times H$, where H is the *transition matrix for the PRPG*.

By associativity of matrix multiplication, $s(t+1) = s(0)H^{t+1}$. If the length of the longest scan chains is m, and the i^{th} seed is given by $s_i(0)$, then the contents of the PRPG after the scan chains are loaded is given by $s_i(m+1)$.

Seed ordering is shown in Algorithm 1. The algorithm starts by generating test patterns for all faults (or for faults that are not detected pseudorandomly.) These test patterns can be generated using any ATPG tool. The algorithm then calculates seeds for one (or multiple) patterns at a time. The order in which patterns are chosen for seeds to be calculated is what the algorithm is optimizing. The algorithm is based on looking ahead in the PRPG sequence by finding $s_i(m+1)$ for seed i and trying to find whether it can be used as a seed $s_j(0)$ to generate pattern j, where $j \neq i$. If that is the case, then we don't need to load a seed for pattern j into the LFSR. If a match is not found we can search for a match with $s_i(d(m+1))$, where $1 \leq d \leq d_{max}$. The parameter d_{max} corresponds to the number of scan shifts we are willing to continue running the PRPG in pseudorandom mode before loading the next seed. Due to the pseudorandom patterns applied, some of the original deterministic patterns may be unnecessary because their corresponding faults are already covered. The algorithm finds such patterns by running fault simulation after loading them and if they don't detect additional faults, the algorithm drops them. The parameter d_{max} can be controlled from the tester. A high-level description of the steps of the ordering algorithm is shown in Fig. 2.

Given H, we precompute $H^{(m+1)}$, $H^{2(m+1)}$,…, $H^{d(m+1)}$, …. We keep multiplying $H^{d(m+1)}$ by the current seed $s(0)$ to get $s(d(m+1))$ until we find a match with one of the other seeds or d exceeds d_{max}. Computing $s(d(m+1))$ from $s(0)$ involves a single matrix multiplication. As shown in Sec. V, matching a final state of the LFSR with a deterministic pattern involves a vector-matrix multiplication. Doing so d_{max} times for $|P|$ pattern has a complexity of $O(d_{\max} \times |P| \times n^2)$, where *n is the size of the PRPG.* Since $H^{d(m+1)}$ is computed once, it requires $m+d_{max}$ matrix-matrix multiplications which has a complexity of $O((d_{\max} + m) \times n^3)$ where *m is the length of the longest scan chain*. The complexity for calculating a seed and generating the matrix of equations is shown in Sec. V.

The output of the algorithm is an ordered set of seeds that need to be loaded into the PRPG to expand into the desired patterns together with the d value for each pattern.

```
Algorithm1 (Seed Ordering Algorithm)
P: set of patterns
S: set of calculated seeds (empty initially)
Calculate the seed $s_1(0)$ of one or multiple
patterns and add it to S
while (all patterns not covered)
  for i = 1 to $d_{max}$
    Find a seed $s_k(0)$ whose final state $s_k(d(m+1))$
    can be an initial state for any set of
    patterns $Q \subset P$.
    i = i + 1
  endfor
  if $s_k(0)$ is found, P = P – Q
  else
    Calculate $s_R(0)$ for pattern $R \subset P$
    S = S + {$s_R(0)$}
endwhile
```

IV. Seed Encoding Algorithm

In this section, instead of only checking the final state of the PRPG to see if it can produce another deterministic test pattern, we try to improve the chances of finding a matching state by looking at the intermediate states of the PRPG, i.e., while shifting. This way, the test time can also be reduced by avoiding shifting a full pattern or several patterns in the scan chains before finding a useful seed.

If the final state of the PRPG doesn't match another seed, we can clock the PRPG a few times until we reach a match with one of the seeds. The first pattern to be encoded is chosen as the pattern with the maximum number of care bits. This way, the number of choices for the first seed is reduced. Also, the don't cares in the other patterns are exploited efficiently by trying to generate those patterns from the final state of the PRPG. To save runtime, we only generate one seed for the first pattern. We also stop at the first match of any other pattern with the final state of the PRPG.

Our seed encoding technique allows the PRPG to run for a variable number of cycles for different seeds. (Normally, it runs for a number of cycles equal to that of the longest chain before a capture cycle.) Our technique achieves this generalization by representing a seed by the number of clock cycles required to reach it. The following explains changes to the BIST architecture to implement the method.

In a usual logic BIST architecture, a *bit counter* is used to count the bits shifted into the scan chains and choose when to disable the Scan Enable (SE) signal for capturing. One way to implement this is to have the bit counter loaded with the value that corresponds to the length of the longest scan chain for every pattern. The length of the scan chains is stored in a register and loaded into the bit counter with every pattern. Our technique is based on running the PRPG for a number of cycles to reach the desired seed. To implement this, we need to load the bit counter register with different values corresponding to the number of cycles before the next capture. Unloading the scan chains starts right after the capture cycle. So, for encoded seeds, there are extra cycles after unloading the response to pattern *i* and before capturing the response for pattern *i*+1. In case if the output is compressed with a MISR, the extra cycles should be taken into account in calculating the signature.

Based on our seed encoding technique, we have two types of seeds, seeds that need to be loaded into the PRPG, *loaded seeds*, and seeds that can be reached by continuing to run the PRPG for additional cycles after loading the scan chains, *encoded seeds*. We use the name encoded seeds because these seeds are encoded into the number of cycles the PRPG needs to run to reach them.

Seed size: How efficient is this encoding? Why not just load all seeds? This question can be answered by a simple example. Take a circuit of 10,000 flip flops that has 10 scan chains of length 1000 each. If the maximum number of care bits in the test patterns is 500 (5%), we need a PRPG of size 520 [6]. Since the length of the scan chain is 1000, the bit counter needs to have only 10 bits. So, by encoding the seed into the number of cycles to reach it we get a 98% (52×) reduction in seed storage. Even if we decide to run the PRPG for up to 1000 additional cycles before reaching the next desired seed, then impact on the size of the bit counter is a single bit.

Seed loading time: In terms of test length, for the example above, loading the PRPG with a new seed takes 520 cycles. Loading the bit counter register takes 11 clock cycles. This means that we can search for a match with another seed in up to 508 cycles while saving on the seed loading time and at the same time saving on tester storage. We make sure in the experiments that the number of additional shifts is smaller than the difference between the size of the original seed and the size of the encoded seed. This way, the loading time is not increased by the additional shifts compared to the original seed loading time.

An enhancement over seed encoding is to utilize the state of the PRPG to generate any useful test patterns. By useful, we mean a pattern that can detect any previously undetected faults as opposed to a specific pattern generated by an ATPG tool. By utilizing all such states, we can greatly increase the flexibility of which PRPG states to use to generate useful test patterns and accordingly reduce the number of seeds required. It is true that many of the patterns generated this way may detect fewer faults than a pattern that was ATPG generated. However, the fact that we don't have to load a seed for such a pattern greatly reduces the storage requirement and the test time. A very important characteristic of this technique presented is that it creates interaction between the seed selection process and the ATPG process to optimize the test procedure based on BIST-related knowledge. Treating ATPG as an independent process of the BIST structure in the circuit does not enable such optimization. This technique will be published in a different paper.

V. Seed Calculation

The LFSR in Fig. 3 will be used as an example for illustrating the equation generation technique. We start with a simple example and then we explain the technique for multiple scan chains.

For every flip-flop in a given scan chain, there is a corresponding equation in terms of the bits of the LFSR. Let's label the scan chain flip-flops by $S_0 \rightarrow S_{m-1}$ where m is the size of the scan chain. Also, let's label the stages of the LFSR by $L_0 \rightarrow L_n$-1 where n is the size of the LFSR. In the example above, the equations for the n most significant flip-flops of the scan chain are: $S_9 = L_3$, $S_8 = L_2$, $S_7 = L_1$, and $S_6 = L_0$ because after n clock cycles the bits of the seed end up in the most significant bits of the scan chain. The reader is invited to verify the remaining equations:

$$S_5 = L_0 \oplus L_3 \quad S_4 = L_0 \oplus L_2 \oplus L_3 \quad S_3 = L_0 \oplus L_1 \oplus L_2 \oplus L_3$$
$$S_2 = L_1 \oplus L_2 \oplus L_3 \quad S_1 = L_0 \oplus L_1 \oplus L_2 \quad S_0 = L_1 \oplus L_3$$

We can represent the above equations by an $m\times n$ matrix (E) in which the rows correspond to the scan chain flip-flops and the columns correspond to the LFSR stages. An entry (i,j) is 1 if and only if L_j appears in the equation of S_i.

The equations of $S_0 \rightarrow S_{m-1}$ represent the linear relations between the scan chain flip-flops and the PRPG flip-flops. If we know the contents of the PRPG at a given point in time e.g. after loading the scan chains, we can check if these contents will generate a given test pattern by checking the linear relations between the PRPG contents and the care bits of the test pattern. If those relations are satisfied, we know that the contents of the PRPG will generate the desired pattern. The equations can be taken directly from the matrix E for the corresponding care bits of the test pattern. If the final state of the LFSR ($s(m+1)$) satisfies all equations for a given pattern then we know that the final state is a seed for this pattern. Accordingly, we don't have to load an extra seed for that pattern. By only trying to satisfy the care bits of a given pattern, we utilize all the don't care bits in that pattern to improve the chances of using a final state as a seed for such a pattern.

We extract the vectors that correspond to the care bit equations from the E matrix and then multiply those vectors with final state of the LFSR. If the result of this multiplication matches the care bit in the test pattern, then the final state is a valid seed for the desired test pattern.

According to the system explained above, the following are the first 3 rows of E for the example above:

$$E_{0-2} = \begin{array}{cc} \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix} & \begin{matrix} S_0 \\ S_1 \\ S_2 \end{matrix} \\ \begin{matrix} L_0 & L_1 & L_2 & L_3 \end{matrix} & \end{array}$$

As an example, for the LFSR shown in Fig. 3, assume that the ATPG tool generates the following two patterns: (X0X1X10XXX, 0XXX1XXXXX). To generate the seed for the 1st pattern, we solve the following system:

$$\begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} L_0 \\ L_1 \\ L_2 \\ L_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

In this example, the rows of the matrix correspond to S_1, S_3, S_5, and S_6, respectively. The solution of the system is $s_0(0)$ = 0111. The next step is to calculate $s_0(m+1)$, which is the value of the contents of the LFSR after the chain is loaded with the pattern. We have that $s_0(m+1) = s_0(0)\times H^{(m+1)}$. Since $m = 10$, $s_0(m+1) = 1000$. Now, we check if this state ($s_0(m+1)$) can generate the second pattern. We need to verify that the bits of $s_0(m+1)$ satisfy the equations for the care bits of the second pattern. This is achieved if the following condition is satisfied:

$$\begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \end{bmatrix} \times \left(s_0(m+1)\right)^T = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \Rightarrow \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \end{bmatrix} \times \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Since the condition is satisfied, the final state of the LFSR after loading the 1st pattern is a valid seed for the 2nd pattern. With more than 2 seeds, the ordering algorithm explained in Sec. III should be used to exploit this efficiently.

If the number of specified bits in more than one test pattern is less than $S_{max} - 20$, then a single seed could be calculated to generate those multiple test patterns as it was discussed in [15]. The final state of the PRPG after loading all of the deterministic test patterns can be checked against the remaining test patterns. It can also be checked whether this final state can generate multiple patterns. Packing multiple test patterns to be generated by a single seed reduces the number of don't care bits and so increases the restrictions and reduces the chances of having the final state of the PRPG generate such test patterns. On the other hand, it reduces the number of test patterns to be dealt with.

In the case of multiple scan chains, if the architecture shown in Fig. 3 above is used, there will be undesired correlation between the bits of patterns that are shifted into the scan chains. Because of this correlation, the outputs of the LFSR stages must go through a phase shifter as shown in Fig. 1. The phase shifter for a given scan chain is a linear sum of some stages from the LFSR. Since the phase shifter is nothing but a linear mapping, the same technique can be applied because we know the linear relation between scan chain flip-flops and the phase shifter output as well as the linear relation between the phase shifter and the PRPG. For a given scan chain i, the phase shift vector is a selection vector $p^i = \begin{bmatrix} p^i_{n-1} & p^i_{n-2} & \cdots & p^i_2 & p^i_1 & p^i_0 \end{bmatrix}$ where p^i_j is one if the XOR feeding scan chain i has the output of stage j of the LFSR as one of its inputs. For example, if scan chain 2 is fed by an XOR gate that adds L_0, L_2 and L_3 then $p^2 = \begin{bmatrix} 1 & 0 & 1 & 1 \end{bmatrix}$ is the phase shifting vector for that scan chain.

The algorithm that generates the equations for a scan chain S that is fed through a phase shifter p^s starts with the selection vector p^s. The algorithm starts by assigning the selection vector p^s to the last row of E. The other rows are generated bottom up by multiplying the transition matrix H by the following row of E.

Since the seed calculation algorithm is entirely based on the transition matrix H and the phase shift vector, it is directly applicable to cellular automata as in the example in Fig. 4. In this example, assume that the CA is connected to a scan chain at the output of stage 6 of the CA. Also assume that the scan chain has 9 stages $S_0 \rightarrow S_8$. The equation for the deepest stage of the scan chain is $S_8 = L_5$. The following matrix shows the equations for the first three flip-flops in the scan chain of the example above:

$$E = \begin{array}{cc} \begin{bmatrix} 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 \end{bmatrix} & \begin{matrix} S_0 \\ S_1 \\ S_2 \end{matrix} \\ \begin{matrix} L_0 & L_1 & L_2 & L_3 & L_4 & L_5 \end{matrix} & \end{array}$$

Algorithm 2 is used to generate the equation matrix E_s. This algorithm can be used with any number of scan chains. The algorithm works with any PRPG and phase shifter.

```
Algorithm2 (Generating equations matrix $E_s$ for scan
chain s fed through a phase shifter)
m: depth of the scan chains
n: size of PRPG
$p^s$: the phase shifter for the current scan chain
$E_s$: equations matrix for the current scan chain
$E_s$-Generator (m, n, h, $E_s$)
  E(m-1) = $p^s$
  for i=m-2 to 0
    Ei = H × $(E(i+1))^T$
  endfor
end
```

Note that generating the E matrix requires solving systems of linear equations for all flip flops. So, the complexity of that process is $O\left(k \times n^3\right)$, where k is the number of flip-flops in the circuit and n is the size of the PRPG. The matching process involves a vector-matrix multiplication and a vector-vector comparison per pattern. So, its complexity is $O(|P| \times n^2 + n)$.

VI. Simulation Results

We performed our experiments on some ISCAS 89 benchmarks. The characteristics of the benchmarks we used are shown in Table I. The table shows the number of primary inputs, primary outputs, flip-flops in the scan chain, and in the LFSR. It also shows the cell-area of the circuits in LSI library cells area units. The library used for technology mapping is LSI Logic's G.Flex library, which is a 0.13 μ library.

The experiments were designed such that pseudorandom patterns are applied first to detect easy faults. Then, test patterns are generated for the undetected faults and the seeds are calculated for the test patterns.

Two experiments were conducted; one to evaluate seed ordering and another one to evaluate seed encoding. Section A and B present the results for seed ordering and seed encoding.

A. *Seed Ordering*

In our seed ordering technique, we try to reorder the seeds such that we need to load the minimal number of seeds into the LFSR to generate the whole test set.

Table II shows the number of test patterns generated for the undetected faults. The seed per pattern column shows the number of seeds required for 100% SSF coverage if a single seed per pattern is assumed as it is the case in most of the previous work. The table shows the number of seeds that need to be stored with our technique. The reduction varies from 8 to 84%.

Table III shows a comparison between the storage requirements for [6], [7], [11], [12], and our seed ordering technique. The data used are taken from the tables published in [12] based on their implementation of all techniques.

Table IV shows the test time (given by the number of scan patterns applied) when regular top off is used compared to the test time when our seed ordering technique is used. The test time increases by a factor of 1.3 to 3.5 in all cases. Notice that seed ordering is also applied as top-off so that the comparison is fair.

Table V shows the number of seeds that need to be loaded with three arbitrary orderings of the seeds. It also shows the reduction gained by using our ordering compared to the average number of seeds with arbitrary orderings. The reduction varies from 5% to 80%. The three arbitrary orderings are: (1) the original order given by the ATPG tool, (2) the reverse order, and (3) taking every other seed (i.e., all odd numbered seeds followed by all even seeds).

B. *Seed Encoding*

In this section, we present results for the seed encoding technique. We cannot evaluate the technique in terms of the number of seeds only. This is because we have two types of seeds, loaded seeds, which have the size of the PRPG, and encoded seeds, which have the size of the bit counter. In order to avoid ambiguity, we compare the storage in terms of the number of bits required for different seeds.

Table VI shows the seed storage needed for the seed per pattern scheme and the seed storage needed for our seed encoding scheme. The storage is calculated by multiplying the number of loaded seeds by the PRPG size and the number of encoded seeds by the bit counter size. The table also shows the reduction gained by seed encoding. The reduction varies from 25% to 86%. Seed encoding is orthogonal to all previous techniques. It can be combined with any of the techniques in [7], [11], or [12] for improved efficiency.

Table VII shows a seed loading time comparison between our seed encoding technique and regular top-off technique, in which pseudorandom patterns are applied followed by deterministic patterns for the undetected faults. Note that the comparison is fair because both techniques are applied after the easy faults are detected with pseudorandom patterns. The loading time is shown as the number of clock cycles needed to load the PRPG with useful seeds. The two techniques then need the same time to fill the scan chains with test patterns. The table shows that seed encoding had in many cases comparable loading time to plain top-off. It also had lower loading time in several cases. This is done while significantly saving storage requirements as shown in Table VI. Notice that this experiment was mainly optimized to save storage by applying pseudorandom patterns between deterministic test patterns. Savings in storage can be traded for test time by avoiding additional pseudorandom patterns.

VII. Conclusion

We presented a seed ordering technique based on the transition matrix of the PRPG and efficiently exploiting the don't care bits in the test patterns. The simulation experiments showed that the number of seeds required for 100% SSF coverage is reduced by up to 80% when our ordering technique is used compared to arbitrary ordering. The technique avoids high complexity like previous analytical solutions and avoids long simulation times like previous simulation techniques.

We also presented a scheme for representing a seed by the number of PRPG cycles required to reach it. The simulation experiments showed that the storage needed is reduced by 25%-85% when our encoding technique is used compared to storing a single seed per pattern.

The main characteristics that make our techniques effective are: (1) Exploiting the linearity of the LFSR and the associativity of matrix multiplication to avoid simulation. (2) Avoiding unnecessary computation to reduce the complexity of ordering the seeds. (3) Exploiting the don't cares in test patterns, and (4) Applying pseudorandom patterns intelligently between deterministic patterns to reduce the number of seeds required to be loaded.

REFERENCES

[1] E. J. McCluskey, "Built-In Self-Test Techniques," *IEEE Design & Test of Computers*, pp. 21-28, Apr. 1985.

[2] E. B. Eichelberger, and E. Lindbloom, "Random-Pattern Coverage Enhancement and Diagnosis for LSSD Logic Self-Test", *IBM Journal of Research and Development*, Vol. 27, No. 3, pp. 265-272, May 1983.

[3] N. A. Touba, and E. J. McCluskey, "Test Point Insertion Based on Path Tracing," *Proc. of VLSI Test Symposium*, pp. 2-8, 1996.

[4] E. B. Eichelberger, E. Lindbloom, F. Motica, and J. Waicukauski, "Weighted Random Pattern Testing Apparatus and Method," *US Patent 4,801,870*, Jan. 1989.

[5] H.-J. Wunderlich, "Multiple Distributions for Biased Random Test Patterns," *IEEE Transactions on Computer-Aided Design*, Vol. 9, No. 6, pp.584-593, Jun. 1990.

[6] B. Koenemann, "LFSR-Coded Test Patterns for Scan Designs," *Proc. of European Test Conference*, pp. 237-242, 1991.

[7] S. Hellebrand, S. Rajski, S. Tarnick, S. Venkataraman and B. Courtois, "Built-in Test for Circuits with Scan Based on Reseeding of Multiple-Polynomial Linear Feedback Shift Registers," *IEEE Trans. on Comp.*, Vol.44, No.2, pp. 223-233, Feb. 95.

[8] N. Touba, and E. J. McCluskey, "Altering Bit Sequence to Contain Predetermined Patterns," *US Patent 6,061,818*, May, 2000.

[9] N. A. Touba, and E. J. McCluskey, "Synthesis of Mapping Logic for Generating Transformed Pseudo-Random Patterns for BIST," *Proc. of ITC*, pp. 674-682, 1995.

[10] A. A. Al-Yamani, and E. J. McCluskey, "Built-In Reseeding for Built-In Self Test", *Proc. of VLSI Test Symposium*, Apr. 2003.

[11] J. Rajski, J. Tyszer and N. Zacharia , "Test Data Decompression for Multiple Scan Designs with Boundary Scan," *IEEE Trans. on Comp.*, Vol. 47,#11, pp. 1188-1200, Nov. 98.

[12] C. V. Krishna, A. Jas, and N. Touba, "Test Vector Encoding Using Partial LFSR Reseeding" *Proc. of International Test Conference*, pp. 885-893, 2001.

[13] E. Volkerink, and S. Mitra, "Efficient Seed Utilization for Reseeding based Compression," *Proc. of VLSI Test Symposium*, Apr. 2003.

[14] S. Venkataraman, J. Rajski, S. Hellebrand, and S. Tarnick, "An Efficient BIST Scheme Based on Reseeding of Multiple Polynomial Linear Feedback Shift Registers," *Proc. of ICCAD*, Vol. 9, No. 6, pp. 572-577, 1993.

[15] S. Hellebrand, B. Reeb, S. Tarnick, H. Wunderlich, "Pattern Generation for a Deterministic BIST Scheme," *IEEE International Conference on Computer Aided Design*, pp. 88-94, Nov. 95.

[16] J. Rajski, J. Tyszer, M. Kassab, N Mukherjee, R. Thompson, K. Tsai, A. Hertwig, N. Tamarapalli, G. Mrugalski, G. Eide, and J. Qian, "Embedded Deterministic Test for Low Cost Manufacturing Test," *Proc. of International Test Conference*, pp. 301-310, 2002.

[17] A. A. Al-Yamani, S. Mitra, and E. J. McCluskey, "BIST Reseeding with Very Few Seeds", *VLSI Test Symposium*, Apr. 2003.

[18] A. A. Al-Yamani, and E. J. McCluskey, "Seed Encoding for LFSRs and Cellular Automata", *Design Automation Conference*, June 2003

[19] M. Lempel, S. Gupta and M. Breuer, "Test Embedding with Discrete Logarithms," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 14, No. 5, pp. 554-566, May 1995.

[20] C. Fagot, O. Gascuel, P. Girard and C. Landrault, "On Calculating Efficient LFSR Seeds for Built-In Self Test," *Proc. of European Test Workshop,* pp. 7-14, 1999.

[21] B. Koenemann, "System for Test Data Storage Reduction," *US Patent 6,041,429,* 2000.

FIGURES

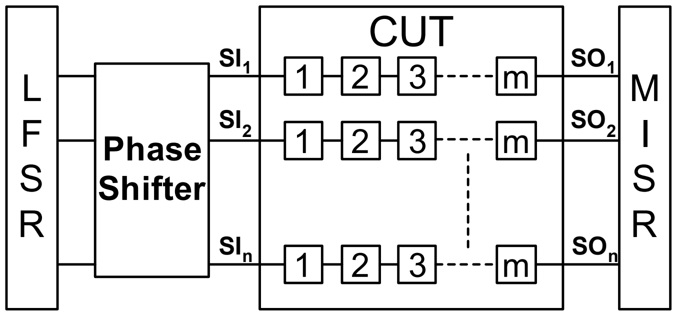

Fig. 1. Multiple scan chains with a phase shifter and equal length scan chains

1. *Run pseudorandom patterns till no improvement in coverage.*
2. *Generate ATPG patterns.*
3. *Pick the ATPG pattern with maximum number of care bits, encode it into a seed and drop it from the test set.*
4. *Find the final state of the PRPG after loading the test pattern and applying the capture cycle.*
5. *If the final state can generate another ATPG pattern or if a sequence of pseudorandom patterns (with capture cycles) can lead to such a state, drop that pattern. Else, pick another ATPG pattern and encode it into a seed.*
6. *If the test set is not empty, go to step 3. Else, exit.*

Fig. 2. High-level description of seed ordering algorithm

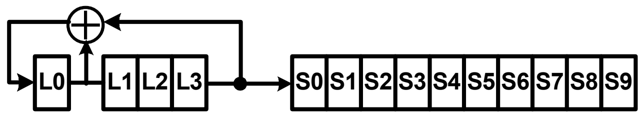

Fig. 3. A 4-stage LFSR connected to a chain

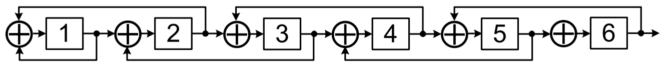

Fig. 4. A 6-stage cellular automaton.

TABLES

TABLE I
ISCAS CIRCUITS USED FOR THE EXPERIMENTS

Circuit Name	PIs	POs	Scan Chain Size	LFSR Size	Cell Area
s953	16	23	29	21	2,286
s1196	14	14	18	15	2,722
s1238	14	14	18	15	2,740
s1423	17	5	74	50	4,531
s1488	8	19	6	6	3,555
s1494	8	19	6	6	3,563
s5378	35	49	179	61	14,377
s9234	36	39	211	80	25,840
s13207	62	152	638	45	44,255
s15850	77	150	534	150	48,494
s38584	38	304	1426	200	115,855

TABLE II
NUMBER OF SEEDS FOR OUR TECHNIQUE COMPARED TO SEED PER PATTERN

Circuit	Cell Area	One seed per pattern	Our Tech.	Red. %
s1238	2,740	30	7	76.7
s1423	4,531	7	3	57.1
s1488	3,555	4	1	75.0
s1494	3,563	4	1	75.0
s5378	14,377	35	7	80.0
s9234	25,840	89	57	36.0
s13207	44,255	74	12	83.8
s15850	48,494	38	35	7.9
s38584	115,855	55	16	70.9

TABLE III
COMPARISON WITH PREVIOUS RESULTS FOR STORAGE REQUIREMENTS

Circuit	[6]		[7]		[11]		[12]		Seed Ordering	
	LFSR size	Total bits	LFSR size	Total bits	LFSR size	Total bits	LFSR size	Total bits	LFSR size	Total bits
s5378	38	1,140	18	570	18	658	38	502	61	427
s9234	81	11,178	61	5,332	61	5,364	81	5,013	80	4,560
s13207	44	6,908	24	3,925	24	3,609	44	3,008	45	540
s15850	58	9,686	38	6,513	38	5,927	58	5,204	150	5,250
s38584	75	4,650	55	3,472	55	3,158	75	2,942	200	3,200

TABLE IV
TEST TIME INCREASE FOR OUR TECHNIQUE COMPARED TO TOP OFF

Circuit	Test Time		Test Time Increase (x times)
	Top Off	Our Tech.	
s1238	414	832	2.0
s1423	487	672	1.4
s1488	228	288	1.3
s1494	228	288	1.3
s5378	1507	1920	1.3
s9234	1433	4992	3.5
s13207	1994	2688	1.3
s15850	1702	3904	2.3
s38584	1655	2624	1.6

TABLE V
NUMBER OF SEEDS NEEDED BY RANDOM ORDERING COMPARED TO OUR ORDERING

Circuit	Cell Area	Number of Seeds					Red.%
		Ord1	Ord2	Ord3	Ord.Avg.	Our Technique	
s1238	2,740	28	27	28	27.7	7	74.7
s1423	4,531	6	6	6	6.0	3	50.0
s1488	3,555	4	3	4	3.7	1	73.0
s1494	3,563	4	4	3	3.7	1	73.0
s5378	14,377	32	33	32	32.3	7	78.3
s9234	25,840	87	85	87	86.3	57	34.0
s13207	44,255	66	65	64	65.0	12	81.5
s15850	48,494	37	38	36	37.0	35	5.4
s38584	115,855	47	50	48	48.3	16	66.9

TABLE VI
SEED STORAGE NEEDED BY OUR TECHNIQUE COMPARED TO SEED PER PATTERN

Circuit	Circuit Cell Area	Seed per Pattern storage	Storage for Our Technique			Storage Red. %
			Loaded Seeds	Encoded Seeds	Total	
s953	2,286	987	189	112	301	69.50
s1196	2,722	1380	195	96	291	78.91
s1238	2,740	1455	180	72	252	82.68
s1423	4,531	850	450	0	450	47.06
s1488	3,555	534	24	54	78	85.39
s1494	3,563	510	18	66	84	83.53
s5378	14,377	2135	1586	9	1595	25.29
s9234	25,840	9360	6560	60	6620	29.27
s13207	44,255	14560	2400	160	2560	82.42
s35932	106,198	420	60	0	60	85.71

TABLE VII
SEED LOADING TIME NEEDED BY TOP-OFF COMPARED TO SEED ENCODING

Circuit	Seed Loading Time	
	Top Off	Seed Encoding
s953	8.4K	10.9K
s1196	7.1K	8.7K
s1238	7.2K	7.6K
s1423	24.9K	24K
s1488	1.9K	2.5K
s1494	1.9K	2.2K
s5378	91.9K	89.8K
s9234	116.9K	107.9K
s13207	94.6K	89.6K
s35932	129.1K	128.6K

CAPTIONS

Fig. 1. Multiple scan chains with a phase shifter and equal length scan chains

Fig. 2. High-level description of seed ordering algorithm

Fig. 3. A 4-stage LFSR connected to a chain

Fig. 4. A 6-stage cellular automaton.

TABLE I
ISCAS CIRCUITS USED FOR THE EXPERIMENTS

TABLE II
NUMBER OF SEEDS FOR OUR TECHNIQUE COMPARED TO SEED PER PATTERN

TABLE III
COMPARISON WITH PREVIOUS RESULTS FOR STORAGE REQUIREMENTS

TABLE IV
TEST TIME INCREASE FOR OUR TECHNIQUE COMPARED TO TOP OFF

TABLE V
NUMBER OF SEEDS NEEDED BY RANDOM ORDERING COMPARED TO OUR ORDERING

TABLE VI
SEED STORAGE NEEDED BY OUR TECHNIQUE COMPARED TO SEED PER PATTERN

TABLE VII
SEED LOADING TIME NEEDED BY TOP-OFF COMPARED TO SEED ENCODING