BIST RESEEDING WITH VERY FEW SEEDS

Ahmad A. Al-Yamani[1], Subhasish Mitra[2] and Edward J. McCluskey[1]

[1]*Center for Reliable Computing*
Stanford University, Stanford , CA
{aaa, ejm}@crc.stanford.edu

[2]*Intel Corporation*
Sacramento, CA
subhasish.mitra@intel.com

Abstract

Reseeding is used to improve fault coverage of pseudo-random testing. The seed corresponds to the initial state of the LFSR before filling the scan chain. The number of deterministic seeds required is directly proportional to the tester storage or hardware overhead requirement. In this paper, we present an algorithm for seed ordering to minimize the number of seeds required to cover a set of deterministic test patterns. Our technique is applicable whether seeds are loaded from the tester or encoded on chip. Simulations show that, when compared to random ordering, the technique reduces seed storage or hardware overhead by up to 80%. The seeds we use are deterministic so 100% SSF fault coverage can be achieved. Also, the technique we present is fault-model independent.

1. Introduction

An advantage of built-in self-test (BIST) is its low cost compared to external testing using automatic test equipment (ATE). In BIST, on-chip circuitry is included to provide test vectors and to analyze output responses. One possible approach for BIST is pseudo-random testing using a linear feedback shift register (LFSR) [McCluskey 85]. Among the other advantages of BIST is its applicability while the circuit is in the field.

Many digital circuits contain random-pattern-resistant (r.p.r.) faults that limit the coverage of pseudo-random testing [Eichelberger 83].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 can be classified as: (1) Modifying the circuit under test (CUT) by test point insertion or by redesigning the CUT [Eichelberger 83, Touba 96], (2) *Weighted pseudo-random patterns*, where the random patterns are biased using extra logic to increase the probability of detecting r.p.r. faults [Eichelberger 89, Wunderlich 90] and (3) *Mixed-mode testing* (aka *top-off*) where the circuit is tested in two phases. In the first phase, pseudo-random patterns are applied. In the second phase, deterministic patterns are applied to target the undetected faults [Koenemann 91, Hellebrand 95, Touba 00]. The technique we present is a mixed mode technique based on inserting deterministic patterns between the pseudo-random patterns.

Modifying the CUT is often not possible due to performance restrictions or intellectual property reasons. Weighted pseudo-random sequences require multiple weight sets that are typically stored on-chip. Mixed mode testing is done in several ways; one is to apply deterministic test patterns from a tester. Another technique is to store the deterministic patterns (or the 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* [Touba 95]. 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 present a technique for built-in reseeding (encoding the seeds in hardware) in [Alyamani 03]. The technique combines mapping logic and reseeding, and is based on running the LFSR in pseudorandom mode after every seed reload because there is a chance more faults will be detected.

Our contributions in this paper are: a seed ordering algorithm that minimizes the number of seed loads. The algorithm is based on exploiting the algebraic properties of the LFSR. The previous work in [Koenemann 91, Hellebrand 95, Rajski 98 and Krishna 01] and many others assume one seed per pattern. Our technique increases the number of patterns generated from one seed significantly, and hence reduces the hardware required for encoding the seeds. Previous algorithms for embedding multiple patterns into a single-seed sequence [Lempel 95 and Fagot 99] have much higher computational complexity and are impractical for reasonable size circuits. Our algorithm is fault model independent.

In Sec. 2 of this paper, we review the related literature. In Sec. 3, we present the seed ordering algorithm. In Sec. 4, we present our seed calculation algorithm. In Sec. 5, we summarize our built-in reseeding scheme. Section 6 shows the simulation results and Sec. 7 concludes the paper.

2. Related Work

Konemann presented 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 [Koenemann 91]. In [Hellebrand 95], a scheme was presented for using Multiple-polynomial LFSRs (MP-LFSRs). The authors used MP-LFSRs to reduce the probability of linear dependence.

[Rajski 98] presented a reseeding technique that improves the encoding efficiency by using variable-length seeds together with an MP-LFSR. In [Krishna 01], the authors presented a scheme where the contents of the LFSR are incrementally modified instead of modifying them all at once. The technique achieves higher encoding efficiency than regular reseeding.

The above schemes assume that seeds are either applied from an external tester or stored in an on-chip ROM. They also encode a single seed per test pattern. The technique in [Alyamani 03] presents a scheme to encode the seeds in hardware compared to storing them. By doing so, the chip doesn't need external testing and doesn't need a ROM with the associated decoding and loading circuitry. When seeds are encoded in hardware, finding the minimal set of seeds will reduce the hardware needed for the seeds. The technique presented in this paper tries to exploit the algebraic properties of the LFSR and the don't care bits in the patterns to encode the maximum number of patterns per seed. The don't care bit result in additional degrees of freedom in solving the linear system of equations for seeds

In [Lempel 95], an analytical method was presented 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. In [Fagot 99], a simulation scheme for calculating initial seeds for LFSRs was presented. The scheme is based on simulating several sequences and picking the one that includes the maximum number of ATPG vectors.

In [Koenemann 00], 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. The SSDS stores a sequence of numbers corresponding to useful and useless sequence lengths. Using additional control logic, the useless patterns are not loaded from the PRPG to the scan chain of the device under test. The SSDS reduces the storage needed for the test patterns. The control logic that skips the useless patterns reduces the test application time.

Koenemann's technique is optimized for reducing test application time. On the other hand, our seed ordering technique is optimized for reducing the number of seed reloads. By reducing the number of reloads, the seed storage is reduced if seeds are stored on or off chip and the area overhead of the reseeding circuitry is reduced if seeds are encoded in hardware as in [Alyamani 03]. In our technique, we don't skip useless patterns for two reasons: (1) If full BIST is assumed, the test application time is not as big a worry as is the case with ATE. (2) Our ELF35 experiment shows the effectiveness of non fault dropping patterns in catching unmodeled faults.

3. Seed Ordering

The BIST architecture we assume is shown in Figure 1. Our technique is applicable with any number of scan chains and any phase shifter. The results shown in Sec. 6 are for a single scan chain per circuit. However, the only difference if multiple scan chains were used is in seed calculation. The way we calculate the seeds is explained in Sec. 4. The seeds are then either loaded from the tester or encoded in hardware on chip as explained in Sec. 5. If the LFSR 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 their corresponding faults are already covered with the pseudorandom patterns. Relying on randomness may not exploit the benefit of this scheme maximally. For such a scheme to work optimally, we should figure out the order of the seeds that will result in the minimum number of seed loads into the LFSR.

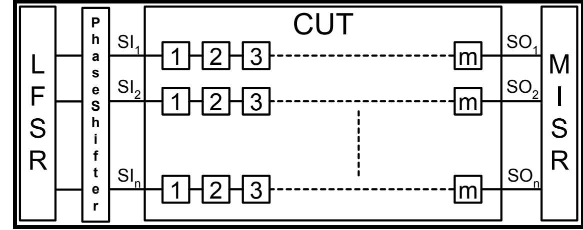

Figure 1: Multiple scan chains with a phase shifter

Our seed ordering algorithm starts with pseudorandom patterns to detect the easy faults and then generates deterministic test patterns fort the undetected faults. It randomly picks a deterministic test pattern and encodes it into a seed. We refer to the seed as the *initial state* (s^0) of the LFSR. By running the LFSR for m cycles, where m is the length of the scan chains, the seed expands into the desired test pattern into the scan chains. We refer to the state of the LFSR after the seed is expanded in the scan chains as the *final state* (s^{m+1}). If the final state of the LFSR 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 an LFSR 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 LFSR at time t. $s^{t+1} = s^t H$, where H is called the transition matrix for the LFSR.

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

The ordering algorithm we present is based on looking ahead in the LFSR sequence by finding $s^{m+1}(i)$ for seed i and trying to find whether it matches any of the other seeds $s^0(j)$, where $j \neq i$. If they match, then $s^0(j)$ doesn't need to be loaded into the LFSR. If a match is not found we can search for a match with $s^{d(m+1)}(i)$, where $1 \leq d \leq d_{max}$. The d_{max} is a parameter that corresponds to the number of scan shifts we are willing to continue running the LFSR in pseudorandom mode before loading the next seed.

***P**:* set of patterns
***S**:* set of calculated seeds (empty initially)
Calculate the seed $s^0(1)$ of pattern 1 and add it to ***S***
while (all patterns not covered)
  for $i = 1$ ***to*** d_{max}
    Find a seed $s^0(k)$ whose final state $s^{d(m+1)}(k)$ can be an initial state for any pattern $p \in$ ***P***.
    $i = i + 1$
  endfor
    if $s^0(k)$ is found, ***P*** = ***P*** − {p}
    else, Calculate $s^0(p)$ for p and, ***S*** = ***S*** + {$s^0(p)$}
endwhile

Algorithm 1: Seed ordering algorithm

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}. If a match is found, then that seed does not need to be stored or loaded. Computing $s^{d(m+1)}$ from s^0 involves a single matrix multiplication.

The algorithm for matching the final state of the LFSR with a seed for another pattern is explained in Sec. 4.

Table 1: Example LFSR Sequence

Cycle	Q 1	Q 2	Q 3	Q 4	Cycle	Q 1	Q 2	Q 3	Q 4
0	1	0	0	0	8	1	1	0	1
1	1	1	0	0	9	0	1	1	0
2	1	1	1	0	10	0	0	1	1
3	1	1	1	1	11	1	0	0	1
4	0	1	1	1	12	0	1	0	0
5	1	0	1	1	13	0	0	1	0
6	0	1	0	1	14	0	0	0	1
7	1	0	1	0	15	1	0	0	0

Given a set of seeds and a user-specified d_{max}, the best ordering is the ordering that will result in the minimum number of seeds that need to be loaded into the LFSR. To find an ordering that minimizes the seed loads, we organize the patterns in sequences. The sequence includes seeds that occur within d_{max} distance from one another and cover multiple ATPG patterns.

Algorithm 1 is the seed ordering algorithm. The output is a set of seeds that need to be loaded into the LFSR to expand into the desired patterns. After every seed is loaded, we run the LFSR for d_{max} scan cycles to generate all patterns that this seed can generate within d_{max}. Algorithm 1 helps in choosing which seeds to load so that we know that other patterns will be covered by just running for d_{max} cycles after every seed.

As an example, take a 4-stage LFSR whose polynomial is $f(x) = x^4 + x^3 + 1$. The LFSR sequence is shown in Table 1. Assume that the seeds are 0111, 0101, 1001 and 0001, which correspond to cycles 4, 6, 11 and 14 of the LFSR sequence shown. If d_{max} used is 3 then we will need to load S4 and S11. If d_{max} used is 5, we will only need to load S4 only. The example is for parallel BIST (test per clock) for simplicity. For serial BIST (test per scan), we need to include the cycles needed to fill the scan chains into account while finding the match in the ordering algorithm. Those cycles are taken care of by raising the transition matrix H to the power m+1 as shown earlier, where m is the length of the scan chains. This means that we try to match the contents of the LFSR after multiples of m+1.

Algorithm 1 reorders the seeds to reduce the seeds to be loaded given d_{max}. By reducing the number of seeds, the algorithm reduces the hardware overhead if the seeds are encoded in hardware. If the seeds are loaded from a tester, the algorithm reduces the storage needed.

It's possible to find the best order of the seeds by simulating all possible permutations. However, this is prohibitively time consuming.

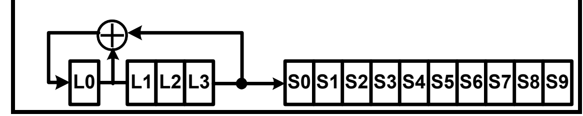

Figure 2: A 4-stage LFSR connected to a chain

4. Seed Calculation

Figure 2 shows an example for a 4-stage LFSR connected to one scan chain with 10 flip-flops. This figure 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 the multiple scan chains.

For every flip-flop in the 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:

$S_5 = L_0 \oplus L_3$ $\qquad$ $S_4 = L_0 \oplus L_2 \oplus L_3$

$S_3 = L_0 \oplus L_1 \oplus L_2 \oplus L_3$ $\qquad$ $S_2 = L_1 \oplus L_2 \oplus L_3$

$S_1 = L_0 \oplus L_1 \oplus L_2$ $\qquad$ $S_0 = L_1 \oplus L_3$

We can represent the above equations by an $m \times n$ matrix 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. According to this system, the following matrix shows the equations for all the flip-flops in the scan chain of the example above:

$$E = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{matrix} S_0 \\ S_1 \\ S_2 \\ S_3 \\ S_4 \\ S_5 \\ S_6 \\ S_7 \\ S_8 \\ S_9 \end{matrix}$$

$$\quad L_0 \quad L_1 \quad L_2 \quad L_3$$

In the case of multiple scan chains, if the architecture shown in Figure 2 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.

The phase shifter makes sure that the sequence fed into chain j is apart from the sequence that goes into chain i by at least the depth of chain i.

The phase shifter for a given scan chain is a linear sum of some stages from the LFSR. The phase shifter for chain i can be represented by:

$p^i = \left[p^i_{n-1} \quad p^i_{n-2} \quad \cdots \quad p^i_2 \quad p^i_1 \quad p^i_0 \right]$. 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.

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. This algorithm can be used with any phase shifter and any number of chains.

Let's now revisit the concept of matching the final state of the LFSR with a seed for one of the remaining patterns. We are trying to match the final state of the LFSR with a seed that will expand into one of the ATPG patterns. An ATPG pattern is represented by a set of equations in terms of the LFSR stages. The equations are actually restrictions that need to be satisfied so that the test pattern is generated from a given seed. 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.

In other words, there are degrees of freedom in solving for the seed. The degrees of freedom are caused by the fact that the number of equations is less than the number of unknowns. To solve such a system, we use Gaussian reduction to extract the pivots from the variables. The non-pivot variables can be assigned arbitrary values. The pivot variables are linear combinations of the non-pivot variables, so they are fixed. By assigning all the non-pivot variables in one of the remaining seeds the same values in $s^{(m+1)}$, we increase the chance of finding a match between $s^{(m+1)}$ and that seed.

As an example, for the LFSR shown in Figure 2, assume that the ATPG tools generates the following two patterns: (X0X1X10XXX, 0XXX1XXXXX). To generate the seed for the 1st pattern, we solve the following system:

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

The solution of the system is $s^0(0) = 0111$. The next step is to calculate $s^{(m+1)}(0)$, which is the value the contents of the LFSR after the chain loaded with the pattern. $s^{(m+1)}(0) = s^0(0) \times H^{(m+1)}$. Since $m = 10$, $s^{(m+1)}(0) = 1000$. Now, we solve the following system to generate the seed for the other pattern:

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

In the above system, we can choose L_0 and L_1 to be the pivots. By doing so, L2 and L3 become free variables that we can assign any arbitrary value to, and L_0 and L_1 are given by:

$L_1 = L_3 \oplus 0 \qquad L_0 = L_2 \oplus L_3 \oplus 1$

So, if we assign 10 to L_2L_3, the second seed will be $s^0(1) = 0010$. However, if we assign 00 to L_2L_3, the second seed will be $s^0(1) = 1000$, which matches $s^{(m+1)}(0)$. This means that the 2nd seed doesn't need to be loaded. We can continue running the LFSR and it will expand to put the 2nd pattern in the scan chain.

5. Built-In Reseeding

In built-in reseeding, the operation of the reseeding circuit is as follows: the LFSR starts running in autonomous mode for sometime according to the reseeding algorithm [Alyamani 03]. Once it is time for reseeding, a seed is loaded into the LFSR, which then goes back to the autonomous mode and so on and so forth until the desired coverage is achieved. The new seed is loaded by putting the LFSR in the state that precedes the seed value, so that at the next clock pulse, the new seed is in the LFSR.

Figure 3(b) shows the structure of the LFSR and its interaction with the reseeding circuit. For our technique, we use muxed flip-flops as shown in the figure. By activating the select line of a given mux, the logic value in the corresponding LFSR stage is inverted.

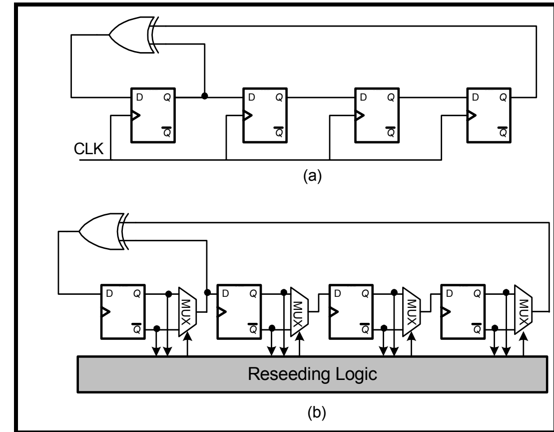

Figure 3: Reseeding circuit connection to the LFSR: (a) A standard 4-stage LFSR (b) 4-stage LFSR with reseeding circuit.

Figure 4 shows where the reseeding circuit fits in a system level view of a circuit with an LBIST controller, which includes the additional control circuitry added for logic BIST. Complete details about the architecture and synthesis for built-in reseeding are given in [Alyamani 03].

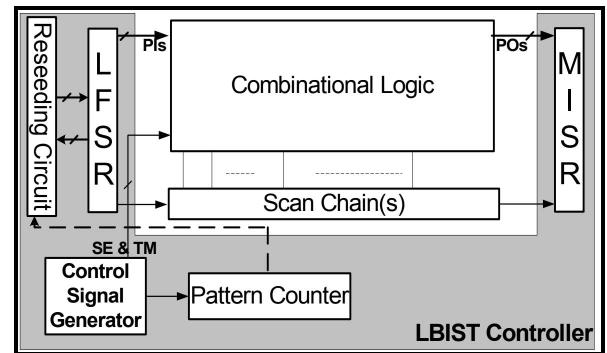

Figure 4: Reseeding circuit in a system view of BIST environment.

6. Simulation Results

We performed our experiments on some ISCAS 89 benchmarks. The characteristics of the benchmarks we used are shown in Table 2. 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 G.Flex library, which is a 0.13 μ library.

The experiment was designed such that pseudo random 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.

In [Koenemann 91, Hellebrand 95, Rajski 98, Touba 00, and Krishna 01] a single seed per pattern is assumed. In our technique, we try to generate multiple patterns with the same seed by reordering the seeds such that we need to load the minimal number of seeds into the LFSR.

Table 2: ISCAS Circuits Used for the Experiments.

Circuit Name	PIs	POs	Scan Chain Size	LFSR Size	Cell Area
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 3: Number of Seeds for Our Technique Compared to Seed per Pattern.

Circuit	Cell Area	Seed/ 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 3 shows the number of test patterns generated for the undetected faults. The seed/pattern column shows the number of seeds that need to be stored or mapped 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 or mapped with our technique. As shown in the table, the reduction varies from 8 to 84%.

Table 4 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.

Table 4: 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 5 shows the number of seeds that need to be loaded with three random orderings of the seeds. It also shows the reduction gained by using our ordering compared to the average number of seeds with random ordering. The reduction varies from 5% to 80%. The three random ordering 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).

Table 6 shows the area overhead needed for built-in reseeding [Alyamani 03] if the random orderings are used. It also shows the reduction in area overhead gained by using our ordering compared to the average overhead with random ordering. The reduction varies from 7% to 84%.

7. Conclusion

We presented a seed ordering technique based on the algebraic properties of the LFSR and exploiting the degrees of freedom in solving the seed equations.

The simulation experiments showed that the area overhead and the storage needed are reduced by up to 80% when our ordering technique is used compared to random ordering. The main characteristics that make our technique 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 finding the best order of the seeds, and (3) Exploiting the degrees of freedom in solving for the seeds to match them with the current contents of the LFSR.

The big win for our technique is that it provides a solution for the problem while avoiding high computational complexity like previous analytical solutions and avoiding long simulation times like previous simulation techniques.

We also presented an algorithm for generating the system of linear equations used to calculate the seeds for with any LFSR and phase shifter.

Acknowledgement

This work was supported by King Fahd University of Petroleum and Minerals and by LSI Logic under contract No. 16517.

Table 5: Number of Seeds Needed by Random Ordering Compared to Our Ordering.

Circuit	Cell Area	Number of Seeds					Reduction %
		Rand1	Rand2	Rand3	Rand.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 6: Area Overhead Needed by Random Ordering Compared to Our Ordering.

Circuit	Cell Area	Area Overhead %					Reduction %
		Rand1	Rand2	Rand3	Rand.Avg.	Our Technique	
s1238	2,740	39.6	37.0	42.7	39.8	9.9	75.1
s1423	4,531	5.8	4.8	4.6	5.1	2.1	58.8
s1488	3,555	2.1	2.0	1.6	1.9	0.3	84.2
s1494	3,563	2.1	2.0	1.6	1.9	0.3	84.2
s5378	14,377	13.0	16.6	15.6	15.1	3.1	79.5
s9234	25,840	14.4	13.8	14.4	14.2	13.2	7.0
s13207	44,255	2.0	1.8	1.9	1.9	0.3	84.2
s15850	48,494	4.1	3.7	4.2	4.0	3.5	12.5
s38584	115,855	1.5	1.4	1.5	1.5	1.2	20.0

References

[Alyamani 03] Al-Yamani, A., and E. J. McCluskey, "Built-In Reseeding for Built-In Self Test", *VLSI Test Symposium*, Apr. 2003.

[Bardell 87] Bardell, P.H., W. McAnney, and J. Savir, "Built-In Test for VLSI", John Wiley, New York, 1987.

[Eichelberger 83] Eichelberger, E. B., 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.

[Eichelberger 89] Eichelberger, E. B., E. Lindbloom, F. Motica, and J. Waicukauski, "Weighted Random Pattern Testing Apparatus and Method," US Patent 4,801,870, Jan. 1989.

[Fagot 99] Fagot, C., 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.

[Hellebrand 95] 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.

[Koenemann 91] Koenemann, B., "LFSR-Coded Test Patterns for Scan Designs," *Proc. of European Test Conference*, pp. 237-242, 1991.

[Koenemann 00] Koenemann, B., "System for Test Data Storage Reduction," US Patent 6,041,429, 2000.

[Krishna 01] Krishna, C. V., A. Jas, and N. Touba, "Test Vector Encoding Using Partial LFSR Reseeding" *Proc. of International Test Conference*, pp. 885-893, 2001.

[Lempel 95] Lempel, M., 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.

[McCluskey 85] McCluskey, E.J., "Built-In Self-Test Techniques," *IEEE Design & Test of Computers,* pp. 21-28, Apr. 1985.

[Rajski 98] Rajski, J., J. Tyszer and N. Zacharia , "Test Data Decompression for Multiple Scan Designs with Boundary Scan," *IEEE Transactions on Computers*, Vol. 47, No. 11, pp. 1188-1200, Nov. 98.

[Touba 95] Touba, N.A., and E.J. McCluskey, "Synthesis of Mapping Logic for Generating Transformed Pseudo-Random Patterns for BIST," *Proc. of ITC*, pp. 674-682, 1995.

[Touba 96] Touba, N.A., and E.J. McCluskey, "Test Point Insertion Based on Path Tracing," *Proc. of VLSI Test Symposium*, pp. 2-8, 1996.

[Touba 00] Touba, N. and E.J. McCluskey, "Altering Bit Sequence to Contain Predetermined Patterns," US Patent 6,061,818, May, 2000.

[Wunderlich 90] Wunderlich, H.-J., "Multiple Distributions for Biased Random Test Patterns," *IEEE Transactions on Computer-Aided Design,* Vol. 9, No. 6, pp.584-593, Jun. 1990.