Transformed Pseudo-Random Patterns for BIST

Nur A. Touba and Edward J. McCluskey

Center for Reliable Computing
Departments of Electrical Engineering and Computer Science
Stanford University, Stanford, CA 94305-4055

ABSTRACT

This paper presents a new approach for on-chip test pattern generation. The set of test patterns generated by a pseudo-random pattern generator (e.g., an LFSR) is transformed into a new set of patterns that provides the desired fault coverage. The transformation is performed by a small amount of mapping logic that decodes sets of patterns that don't detect any new faults and maps them into patterns that detect the hard-to-detect faults. The mapping logic is purely combinational and is placed between the pseudo-random pattern generator and the circuit under test (CUT). A procedure for designing the mapping logic so that it satisfies test length and fault coverage requirements is described. Results are shown for benchmark circuits which indicate that an LFSR plus a small amount of mapping logic reduces the test length required for a particular fault coverage by orders of magnitude compared with using an LFSR alone. These results are compared with previously published results for other methods, and it is shown that the proposed method requires much less overhead to achieve the same fault coverage for the same test length.

1. Introduction

One of the requirements for built-in self-test (BIST) is on-chip test pattern generation. Some circuit, called a *test pattern generator*, is needed to generate test patterns for the circuit under test (CUT). For a given test length, the test pattern generator must be able to generate test patterns that provide a high fault coverage. A linear feedback shift register (LFSR) is commonly used as a test pattern generator because it provides two advantages: (1) it has a simple structure requiring small area overhead, (2) it can also be used as an output response analyzer thereby serving a dual purpose. BIST techniques such as circular BIST [15] and BILBO registers [13] make use of these advantages to reduce overhead. Unfortunately, the pseudo-random test patterns that are generated do not always give high enough fault coverage for a reasonable test length. There are two ways to solve this problem. One is to increase the fault detection probabilities in the CUT by inserting test points [11] or by redesigning it [20], and the other is to augment the LFSR with additional logic to improve the patterns that are generated. This paper presents a new approach for the latter.

Given an LFSR that doesn't provide high enough fault coverage when used as a test pattern generator, one possible solution is to simply try a different seed or different characteristic polynomial. Lempel *et al.* [16] presented an analytical method for finding a good seed for an LFSR with a given characteristic polynomial. Results in [16] indicate, however, that seed selection cannot reduce the test length by more than a factor of 10. The LFSR must be augmented by additional logic if this reduction is not sufficient. Three general approaches that have been proposed for doing this are as follows:

1. Mixed-Mode: Logic is added to generate deterministic patterns to detect faults that the pseudo-random patterns miss. Many methods have been proposed for generating deterministic patterns on-chip [2,3,7,8,10]. In general, however, substantial overhead is required.
2. Multiple Seeds/Reconfigurable LFSR: Logic is added to periodically reseed the LFSR or change its characteristic polynomial. Techniques have been developed for finding seeds and characteristic polynomials that will generate tests for the hard-to-detect faults [9,14,21]. The seeds and characteristic polynomials need to be stored on-chip.
3. Weighted Patterns: Logic is added to bias the pseudo-random patterns towards those that detect the hard-to-detect faults [12,18,19,22]. Multiple weight sets are usually required for an acceptable test length [23]. The weight sets need to be stored on-chip.

This paper presents a new approach for augmenting an LFSR, or any other pattern generating circuit, to produce a desired fault coverage for a given test length. No storage of deterministic patterns, seeds, characteristic polynomials, or weight sets is required. In fact, no additional sequential logic needs to be added. As illustrated in Fig. 1, a purely combinational logic block is added between the pattern generating circuit and the CUT to map the original set of patterns into a new transformed set of patterns that provides the desired fault coverage. The original set of patterns produced by the pattern generating circuit for a given test length will be referred to as the *original pattern set*, and the set of patterns that is produced at the output of the mapping logic block will be referred to as the *transformed pattern set*. The strategy is to identify patterns in the original pattern set that don't detect any new faults and then map them into patterns that detect the hard-to-detect faults. The key is to design the mapping logic so that it uses only a small number of gates. This is accomplished by using the special class of mappings described in Sec. 2. Given a pattern generating circuit, a procedure is described in Sec. 3 for designing mapping logic to produce transformed patterns that satisfy test length and fault coverage requirements. The goal of the procedure is to minimize the number of gates required in the mapping logic.

The test pattern generator architecture in which a pseudo-random pattern generator is followed by a transform network to produce "biased" patterns is not new. However, previous methods have only considered using a transform network that either weights or correlates signal probabilities. This paper considers a broader class of transformations. Whereas the transformations used in weighted pattern testing are uniformly applied to some number of patterns per weight set, the transformations used here are applied only to selected sets of patterns.

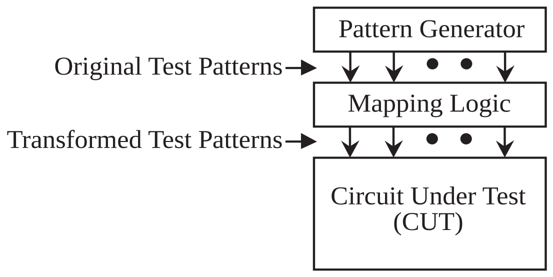

Figure 1. Block Diagram for Generating Transformed Patterns

2. Cube Mapping

In the method described in this paper, a special class of mappings, which will be called *cube mappings*, are used to map the original pattern set into a transformed pattern set. Each cube mapping is uniquely specified by a "source" cube and an "image" cube where each cube is a product of literals in the input space of the CUT. Each original pattern that is contained in the source cube is mapped into a new pattern that is contained in the image cube. In the following definitions, a cube in an input space with n variables will be represented by a vector in $\{0,1,X\}^n$ where a '*0*' indicates that the variable appears complemented in the cube, a '*1*' indicates that the variable appears uncomplemented in the cube, and an '*X*' indicates that the variable doesn't appear in the cube.

Definition 1: For a circuit with n primary inputs, let $A = (a_1,\dots,a_n) \in \{0,1\}^n$ be an input pattern and let $C = (c_1,\dots,c_n) \in \{0,1,X\}^n$ be a cube, then A is *contained* in C if $\forall_j\,[\,(a_j = c_j)$ *or* $(c_j = \text{'}X\text{'})\,]$.

Definition 2: For a circuit with n primary inputs, let $A = (a_1,\dots,a_n) \in \{0,1\}^n$ be an input pattern, then a *cube mapping*, $M: \{0,1\}^n \to \{0,1\}^n$, *with source cube* $S = (s_1,\dots,s_n) \in \{0,1,X\}^n$ and *image cube* $I = (i_1,\dots,i_n) \in \{0,1,X\}^n$ is defined as follows:

$M_{S\to I}(A) = B = (b_1,\dots,b_n) \in \{0,1\}^n$ where

if A is contained in S then if i_j = 'X' then $b_j = a_j$ else $b_j = i_j$

else if A is not contained in S then $b_j = a_j$

An example of a cube mapping is shown in Fig. 2. The source cube $a_1'a_2\,(0,1,X)$ contains the patterns *0 1 0* and *0 1 1*. These two patterns are mapped into new patterns that are contained in the image cube $a_2'a_3\,(X,0,1)$ by setting $a_2 = 0$ and $a_3 = 1$. Hence both patterns are mapped into *0 0 1*.

The method described in this paper involves finding some set of cube mappings, $\{M_{S_1\to I_1},\dots,M_{S_n\to I_n}\}$, that can be used to map the original pattern set into a transformed pattern set that provides the desired fault coverage. The advantage of using cube mappings is that they can be implemented with a small amount of logic. In Fig. 2, the logic required to implement a cube mapping is shown. One AND-gate is needed to decode the input patterns that are contained in the source cube, and one two-input AND or two-input OR gate is needed for each literal in the image cube to perform the mapping. The mapping can be disabled during normal operation by simply adding an input to the decoding AND-gate (labeled "test mode" in Fig. 2).

Original $a_1a_2a_3$		Transformed $a_1a_2a_3$
0 0 0		0 0 0
0 0 1		0 0 1
0 1 0	$\to$	0 0 1
0 1 1	$\to$	0 0 1
1 0 0		1 0 0
1 0 1		1 0 1
1 1 0		1 1 0
1 1 1		1 1 1

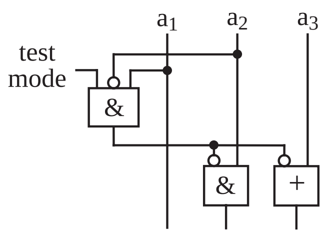

Figure 2. Cube Mapping with Source Cube $a_1'a_2\,(0,1,X)$ and Image Cube $a_2'a_3\,(X,0,1)$

3. Procedure for Selecting Cube Mappings

Given a pattern generating circuit, a test length, and a fault coverage requirement, a procedure is described in this section for finding a set of cube mappings that will map the original pattern set into a transformed pattern set that satisfies the fault coverage requirement. The procedure involves generating cube mappings one at a time until the resulting transformed pattern set gives a high enough fault coverage.

3.1 Overview of Procedure

The steps in the procedure are as follows:

1. Simulate the pattern generating circuit for the given test length to generate the original pattern set.

2. Evaluate the fault coverage and identify undetected faults.
3. If the fault coverage is high enough, then the procedure is complete.
4. Otherwise, add a cube mapping.
5. Compute the resulting transformed pattern set and loop back to step 2.

In step 4, a cube mapping is added to improve the fault coverage. A method for selecting which cube mapping to add during this step will be described in detail. The method involves first selecting a source cube and then selecting the image cube. To illustrate the method, a simple example of finding mapping logic for testing the 5-input ISCAS 85 benchmark circuit *C17* will be used. Assume that the *C17* circuit is to be tested using a pseudo-random generator and 100% fault coverage is required with a test length of 10. The first steps of the method are done for the example and the results are shown in Fig. 3: the original pattern set is obtained, and fault simulation is done revealing that 5 out of 18 faults are left undetected. Now the task is to select a cube mapping that will produce a transformed pattern set that will detect the undetected faults; this is the subject of the next two subsections.

C17 EXAMPLE

Fault Coverage Requirement: 100%

Test Length Requirement: 10

Original Pattern Set: 00111, 11011, 10111, 10110, 11010, 00101, 11100, 01010, 10100, 00100

Fault Coverage = $\frac{13}{18}$ = 72.2%

Figure 3. Original Pattern Set and Fault Coverage for *C17* Example.

3.2 Selecting a Source Cube

Each pattern in the original pattern set that is contained in the source cube will be transformed into a new pattern. In order not to reduce the fault coverage, it is important to choose a source cube that does not contain all of the patterns in the original pattern set that detect some fault f; otherwise the transformed pattern set may not contain a test pattern for fault f. On the other hand, in order to maximize the potential of the mapping for increasing fault coverage, the source cube should contain as many patterns as possible in the original pattern set so that the transformed pattern set will contain as many new patterns as possible. Thus the strategy for selecting the source cube is to find a large cube that doesn't contain all of the test patterns in the original pattern set for some fault.

In order to avoid selecting a cube that contains all of the test patterns in the original pattern set for some fault, it is necessary to know which patterns in the original pattern set detect each fault. To find the whole set of patterns that detect each fault, fault simulation without fault dropping would be required. Results in [17] indicate that fault simulation time can be increased by up to a factor of 50 if fault dropping is not used. If fault dropping is used, then the fault detection information is limited to one pattern for each fault (the first pattern that detected the fault). However, this is enough information to choose the source cube. For each detected fault, there must be at least one test pattern that is not contained in the source cube. This requirement can be satisfied if the source cube is chosen such that it doesn't contain any of the patterns that caused faults to be dropped during fault simulation.

Let F be a Boolean function equal to the sum of the minterms corresponding to each pattern that caused a fault to be dropped. Then finding a cube that doesn't contain any pattern that caused a fault to be dropped is equivalent to finding an implicant in F'. Finding an implicant in F' that is as large as possible can be solved using binate covering. A binate matrix is formed in which each column corresponds to a literal and each row corresponds to a pattern that caused a fault to be dropped. A minimum binate column covering for the resulting matrix is then computed and expressed as a cube C with each literal corresponding to a binate column in the solution. The source cube is then computed by complementing each literal in C. The source cube will then have the property that it doesn't match any pattern that caused a fault to be dropped, and therefore it is guaranteed to not contain all of the patterns in the original pattern set that detect some fault. Binate covering is an NP-complete problem, however, there are good heuristic algorithms for it (e.g., [6]).

For the *C17* example, the original test patterns that caused faults to be dropped are listed in Fig. 4. These patterns are formed into a binate matrix and a minimum binate column cover is found. The source cube is computed by complementing each literal in the minimum binate column cover. The source cube has the property that it doesn't contain any of the patterns that caused faults to be dropped.

C17 EXAMPLE

Patterns that Drop Faults <a,b,c,d,e>: 00111, 11011, 10111, 10110, 00101

a′	a	b′	b	c′	c	d′	d	e′	e
1	0	1	0	0	1	0	1	0	1
0	1	0	1	1	0	0	1	0	1
0	1	1	0	0	1	0	1	0	1
0	1	1	0	0	1	0	1	1	0
1	0	1	0	0	1	1	0	0	1

A Minimum Binate Column Cover: b′ e

Selected Source Cube: b e′

Figure 4. Source Cube Selection for *C17* Example.

3.3 Selecting an Image Cube

Once the source cube has been selected, the remaining task is to select the image cube. The goal in selecting the image cube is to transform the patterns that are contained in the source cube into new patterns that detect as many of the undetected faults as possible. The patterns contained in the source cube are mapped into patterns contained in the image cube. The strategy that is used for selecting the image cube is to find some good candidate image cubes and compute how many undetected faults would be detected if each was used. The candidate image cube that gives the highest fault coverage is then selected as the image cube.

Deterministic test patterns for the undetected faults are used to guide the selection of candidate image cubes. The unnecessary input assignments in the test patterns are left as don't cares (*X's*) thereby forming *test cubes* for each fault. The test cubes are obtained using an automatic test pattern generation (ATPG) tool. If the intersection of the image cube and the test cube for fault *f* is non-empty, then the image cube contains test patterns for fault *f*, and therefore fault *f* can be potentially detected in the transformed pattern set. So it is important to choose candidate image cubes that have non-empty intersections with as many test cubes as possible. This is done using rectangle covering similar to what is done in multilevel logic optimization to find cube factors [5]. A binate matrix *B* is formed in which each test cube is represented by a row. The complemented and uncomplemented literals corresponding to each don't care input in a test cube are both set equal to 1 (this is different from finding cube factors where they are both set equal to 0). A rectangle in *B* corresponds to a cube that has a non-empty intersection with the test cubes covered by the rectangle (this is different from finding cube factors where a rectangle corresponds to a common cube between the cubes covered by the rectangle).

One approach for selecting candidate image cubes would be to simply use each of the prime rectangles in *B* (i.e., each rectangle not contained in another rectangle). However, for circuits with large numbers of primary inputs, the number of prime rectangles becomes prohibitive. So the strategy that is used instead is to begin with a prime rectangle that covers as many test cubes as possible (i.e., is the same height or taller than all other prime rectangles). The cube corresponding to this rectangle is used as the initial candidate image cube. Subsequent candidate image cubes are then obtained by incrementally adding literals to the initial candidate image cube; this corresponds to incrementally adding columns to the initial rectangle. The columns are selected based on maximizing the number of test cubes covered by the resulting rectangle (i.e., maximizing its height). The procedure is as follows:

1. The initial candidate image cube is set equal to a prime rectangle in *B* with maximum height.

The initial candidate image cube will then have the property that it has a non-empty intersection with as many test cubes as possible. Thus, it will contain test patterns for as many undetected faults as possible.

2. Compute the transformed pattern set based on the candidate image cube.

The transformed pattern set is computed for the cube mapping specified by the previously selected source cube and the candidate image cube.

3. Determine how many undetected faults are now detected in the transformed pattern set.

This requires fault simulation of the undetected faults.

4. If the number of faults detected is larger than that of the best candidate seen so far, then mark this candidate as the best candidate.

The goal in choosing the image cube is to detect as many faults as possible, so only the best candidate is kept.

5. Add a column to the current rectangle to form a new rectangle that is as tall as possible.

The goal of this step is to find a smaller candidate image cube that has the potential to detect as many faults as possible. A literal is added to the current candidate image cube based on maximizing the number of test cubes that the resulting candidate image cube has a non-empty intersection with.

6. If the number of rows covered by the resulting rectangle is less than or equal to the number of faults detected by the best candidate, then select the best candidate. Else, loop back to step 2.

The next candidate image cube will have a non-empty intersection only with the test cubes covered by the rectangle and hence its potential for detecting faults is limited by the number of rows. If it is not possible for the next candidate to detect more faults than the best candidate, then the best candidate is selected as the image cube.

7. Expand the image cube as much as possible without reducing fault coverage.

A gate is needed for each literal in the image cube, so if some of the literals can be removed without reducing the fault coverage, then this results in a hardware savings. This can be done by removing one literal at a time from the image cube and computing the resulting fault coverage. If the fault coverage remains the same, then the literal is not needed.

For the *C17* example, the test cubes for the 5 undetected faults are listed in Fig. 5. These test cubes are formed into a binate matrix, and the first candidate image cube is set equal to $a'e$ which corresponds to a rectangle with maximum height. The transformed pattern set is computed and fault simulation of the undetected faults is done revealing that only one of them is detected. The *d* column is then added to the rectangle because it maximizes the height of the resulting rectangle. The second candidate image cube is then set equal to $a'de$. The transformed pattern set is computed and fault simulation of the undetected faults is done revealing that 3 of them are detected. Since the number of rows in the next rectangle will be less than or equal to the number of faults detected for the second candidate image cube, the selection procedure terminates and the selected image cube is *a′d e*. Removing any of the literals from the image cube reduces the fault coverage, so the image cube is not expanded.

C17 EXAMPLE

Test Cubes for Undetected Faults: XX00X, X111X, 010X1, 0111X, X001X

a′	a	b′	b	c′	c	d′	d	e′	e		
1	1	1	1	1	0	1	0	1	1	Transformed Patterns: $M_{be'\rightarrow a'e}$	
1	1	0	1	0	1	0	1	1	1	First Candidate	$11010 \rightarrow 01011$
1	0	0	1	0	1	1	1	0	1	Image Cube: a′ e	$01010 \rightarrow 01011$
1	0	0	1	0	1	0	1	1	1	1 fault detected	$11100 \rightarrow 01101$
1	0	1	0	1	0	0	1	1	1		
										Transformed Patterns: $M_{be'\rightarrow a'de}$	
										Second Candidate	$11011 \rightarrow 01011$
										Image Cube: a′ d e	$01010 \rightarrow 01011$
										3 faults detected	$11100 \rightarrow 01111$

Selected Image Cube: a′ d e

Figure 5. Image Cube Selection for *C17* Example.

4. Hardware Implementation

After a set of cube mappings has been selected such that the test length and fault coverage requirements are satisfied, a gate implementation of the mapping logic can be easily constructed. This is best explained with an example. For the *C17* example, the steps for selecting the first cube mapping, $M_{be'\rightarrow a'de}$, were shown. This cube mapping causes 3 faults to be detected, but there are still 2 undetected faults remaining. So the same steps were used to select a second cube mapping, $M_{a'e'\rightarrow ab'c'}$. This cube mapping causes both of the remaining 2 faults to be detected. So the set of these two cube mappings satisfies the 100% fault coverage requirement. A circuit that implements these mappings is shown in Fig. 6. One AND gate is needed to decode each source cube. A "test mode" input is added to each decoding AND gate so that it can be disabled during normal operation. The first cube mapping, $M_{be'\rightarrow a'de}$, is implemented by adding an AND gate to *a*, and OR gates to *d* and *e*. The second cube mapping, $M_{a'e'\rightarrow ab'c'}$, is implemented by adding an OR gate to *a*, and AND gates to *b* and *c*. If a pattern is contained in both sources cubes, the the output of both decoding AND gates will go high. Then latter cube mappings must override previous cube mappings, so in this case the OR gate on *a* must be placed after the AND gate. Because the latter source cubes are always chosen so that they won't contain patterns that detect new faults, there is no concern that having the second cube mapping override the first cube mapping will reduce fault coverage.

An obvious concern about constructing a circuit structure by cascading gates is that the delay through the circuit will be a problem. Note that the circuit can be flattened and synthesized with logic synthesis tools to control delay. Also, the mapping logic can be bypassed during normal operation by using MUXes if necessary.

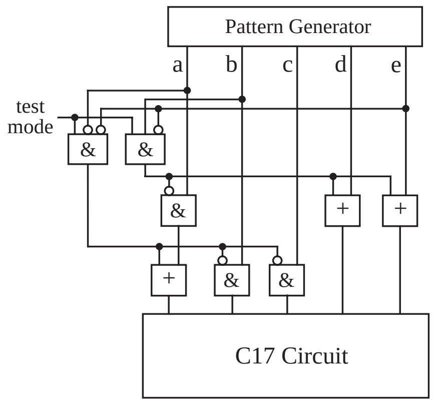

Figure 6. Gate Implementation of Cube Mapping Logic for C17 Example: $M_{be'\rightarrow a'de}$ and $M_{a'e'\rightarrow ab'c'}$

5. Experimental Results

The method described in this paper was used to generate mapping logic to reduce the pseudo-random pattern test length for some of the ISCAS 85 and ISCAS 89 benchmark circuits that require over a million test patterns.

5.1 Comparison with LFSR Alone

Table 1 compares using only an LFSR and using an LFSR with cube mapping logic. It was assumed that the flip-flops in the ISCAS 89 circuits were configured as part of the LFSR during testing so that the circuits are tested like combinational circuits. The number of stages in the LFSR for each circuit was equal to the number of primary inputs plus the number of flip-flops. Patterns were applied in parallel to the circuit, i.e., a pattern was applied each clock cycle. Only detectable faults were considered in fault coverage calculations. Table 1 shows results for using an LFSR alone to generate the patterns. The fault coverage after 1K patterns, 10K patterns, and 50K patterns is shown, and the test length required for 100% fault coverage is shown (all circuits required over a million patterns). The method described in this paper was used to generate mapping logic to provide 100% fault coverage for test lengths of 1K, 10K, and 50K patterns using the same LFSR (same characteristic polynomial and same initial seed). The mapping logic was inserted between the LFSR and CUT. In Table 1, results are shown for the LFSR with the mapping logic. For each of the three test lengths, four things are shown: the number of cube mappings, the number of gates required to implement the mapping logic, the number of literals in the mapping logic (gate inputs), and the fault coverage achieved. When more than one cube mapping is required to achieve 100% fault coverage, results are shown for different numbers of cube mappings to show the possible tradeoffs between area and fault coverage. These results indicate that a small amount of mapping logic can dramatically reduce the random pattern test length. If the number of gates in the mapping logic is divided by the number of inputs in the CUT, then for all of the circuits, less than a gate per input is required to reduce the test length by 3 orders of magnitude or more. Note that the number of literals per gate (i.e., average gate fan-in) is very small as well. As the test length is increased, the amount of mapping logic required for 100% fault coverage goes down. It is very easy to trade off between test length, fault coverage, and hardware overhead.

5.2 Comparison with Prior Methods

There are three important factors in choosing a test pattern generator for BIST: test time, test quality, and hardware area. To evaluate the test pattern generators that are designed by the method in this paper, a comparison was made with other published results using three measures: test length (for test time), fault coverage (for test quality), and gate equivalents plus flip-flop count (for hardware area). Table 2 shows the comparison. The fault coverage is the same for all techniques: 100% of detectable single stuck-at faults. Parallel test pattern application ("a test per clock") is assumed for all techniques. The first column gives the circuit names, and the next column shows the test length for pseudo-random pattern testing using an LFSR. Then results are given for 3 different methods plus the proposed method. The test length and hardware overhead is shown for each method. In some cases, results are given for two different test lengths to show the tradeoff between test time and hardware overhead. The hardware overhead is the hardware required in addition to what is needed for pseudo-random pattern testing with an LFSR. Flip-flops and gates are counted separately. The gates are measured by gate equivalents (GE's) using the same method suggested in [12] to reflect a static CMOS technology: *(0.5)(n)* GE's for an *n*-input NAND or NOR, *(2.5)(n-1)* GE's for an *n*-input XOR, and *1.5* GE's for a 2-to-1 MUX (realized by transmission gates). The hardware overhead for each method is an

Table 1. Comparison of Testing with an LFSR Alone versus an LFSR plus Cube Mapping Logic

Circuit		LFSR Alone				LFSR plus Cube Mapping Logic											
		Fault Coverage			Test Len	1K Test Length				10K Test Length				50K Test Length			
Name	PI	at 1K	at 10K	at 50K	for 100%	maps	gates	lits	Cov	maps	gates	lits	Cov	maps	gates	lits	Cov
s420	35	75.3%	80.0%	87.4%	1.1M	1	12	25	95.2%	1	10	21	100%	1	7	15	100%
						3	24	52	99.1%								
						4	28	62	100%								
s641	54	94.5%	97.1%	97.6%	1.0M	1	17	36	98.1%	1	11	24	100%	1	8	18	100%
						2	31	64	99.2%								
						3	38	82	100%								
s838	67	78.1%	81.4%	82.7%	>100M	3	60	124	97.1%	1	25	51	93.6%	1	21	43	96.2%
						4	80	166	99.0%	2	39	81	99.1%	2	39	80	100%
						7	93	198	100%	4	55	118	100%				
C2670	233	87.9%	88.2%	88.4%	4.6M	2	77	156	96.1%	1	28	59	94.4%	1	32	65	96.1%
						5	169	343	99.2%	2	67	136	98.6%	2	66	134	99.1%
						12	252	520	100%	4	112	228	100%	4	95	194	100%
C7552	207	92.7%	95.0%	96.7%	>100M	5	201	421	98.0%	2	107	215	98.8%	2	98	198	98.9%
						12	416	879	99.0%	3	163	329	99.7%	4	117	238	99.9%
						21	548	1179	99.5%	6	211	428	100%	6	180	366	100%

Table 2. Comparison of Test Length and Required Hardware

Circuit	Random	Multiple Weight Sets [4]				3-Weight [18]			Fix-Biased [1]			Proposed Method		
Name	TLen	TLen	WS	FF	GE	TLen	FF	GE	TLen	FF	GE	TLen	FF	GE
s420	1.1M	532	4	≥2	350	NA	NA	NA	5K	18	>7	500	0	48
		1.8K	2	≥1	245							1K	0	31
s641	1.0M	593	3	≥2	459	NA	NA	NA	19K	20	>21	500	0	23
												10K	0	12
s838	>100M	893	5	≥3	770	NA	NA	NA	86K	19	>14	850	0	99
		17K	2	≥1	469							10K	0	59
C2670	4.6M	1.3K	9	≥4	4078	19K	5	1507	19K	54	>259	1K	0	260
		12K	3	≥2	1981	30K	5	1316				7K	0	114
C7552	>100M	2K	12	≥4	4554	47K	6	3003	191K	111	>658	10K	0	214
		69K	5	≥3	2380	72K	6	2475				50K	0	183

estimate that is computed as follows:
Multiple Weight Sets: The weight sets from [4] are used. The number of weight sets required is shown under the column *WS*. It is assumed that the best case occurs in which no stages have to be added to the LFSR to avoid correlation that increases test length. Thus, extra flip-flops are needed only to keep track of which weight set is being used. The logic required for each input to the CUT is conservatively estimated to be a total of 4 gates to generate the weighted signals and *WS* 2-to-1 MUXes to select the weighted signals based on which weight set is currently active.

FF's = log_2*(number of weight sets)*

GE's = *[4 + (1.5) (WS)] (number of inputs in CUT)*

3-Weight Method: This method was proposed by Pomeranz and Reddy in [18]. 3-gate modules are used to fix the value of certain inputs while random patterns are being applied thus forming "expanded tests". Extra flip-flops are needed to keep track of which expanded test is being used. The logic required by the 3-gate modules depends on the fan-in. One of the gates is a two-input gate, and the average fan-in for the other two is given in [18] (results are not available for the ISCAS 89 circuits).

FF's = log_2*(number of expanded tests)*

GE's = *(number of 3-gate modules) (1 + average fan-in)*

Fixed-Biased Method: This method was proposed by AlShaibi and Kime in [1]. It generates patterns using a weighted bit stream and fixing the value of some bits. A ROM is required to store configuration sequences that are periodically loaded during testing, but for sake of comparison, it is assumed that the configuration sequences are stored off-chip even though this would impact test time. A 17-stage LFSR plus some weight logic is used to generate the weighted bit stream. Each fixed bit requires one extra flip-flop, four 2-to-1 MUXes, and a two-input NAND gate; the number of fixed bits for each circuit is given in [1].

FF's = *17 + (number of fixed bits)*

GE's = *[(4)(1.5) + 1] (number of fixed bits)*

The proposed method requires no additional flip-flops, only combinational logic between the LFSR and the CUT. Assuming that flip-flops require 4 gate equivalents or more, the proposed method requires the least hardware overhead for a given test length compared with the other methods. In many cases, the proposed method reduces the test length significantly more than the other methods while using much less hardware.

Wunderlich proposed a generator of unequiprobable random tests (GURT) in [22] that requires very little hardware overhead but is limited to only one weight set. Hartmann and Kemnitz proposed a method in [12] that uses a modified GURT structure and described test pattern generators for *C2670* and *C7552* which require very little hardware overhead. However, these methods are not general methods because they use only one weight set and therefore are limited in their ability to reduce test length. For some circuits these methods will not be able to reduce the test length enough. The methods shown in the table are general methods in the sense that they can be used to reduce the test length for any circuit by basically any amount. It should also be pointed out that the order of the flip-flops in a GURT structure is greatly constrained and therefore can add substantial routing overhead. The proposed method, on the other hand, places no constraints on flip-flop ordering and allows the use of normal BILBO register cells.

6. Conclusions

The method described in this paper requires much less overhead than other general methods to achieve the same fault coverage for a given pseudo-random pattern test length. In addition to minimizing hardware overhead, the proposed approach has the following advantages:

1) Easy to insert into an existing design.
2) Fully compatible with BILBO registers.
3) Easy to trade off between test time, fault coverage, and hardware overhead.
4) No additional sequential logic is required.
5) Very simple control -- only one control line is needed (to indicate test mode).

Thus, the method described in this paper is very convenient to use in BIST designs to boost fault coverage. Mapping logic can be generated and seamlessly inserted into a BIST architecture.

In this paper, the problem of improving fault coverage during pseudo-random pattern testing was thought of as transforming a pseudo-random pattern set into a better one. This led to the use of a broader class of transformations than had been previously considered. Other transformations besides cube mappings are currently being investigated. More complex transformations hold promise for even greater improvement.

References

[1] AlShaibi, M.F., and C.R. Kime, "Fixed-Biased Pseudorandom Built-In Self-Test for Random Pattern Resistant Circuits," *Proc. of Int. Test Conf.*, pp. 929-938, 1994.

[2] Agarwal, V.K., and E. Cerny, "Store and Generate Built-In Testing Approach," *Proc. of FTCS-11*, pp. 35-40, 1981.

[3] Akers, S.B., and W. Jansz, "Test Set Embedding in a Built-In Self-Test Environment," *Proc. of Int. Test Conf.*, pp. 257-263, 1989.

[4] Bershteyn, M., "Calculation of Multiple Sets of Weights for Weighted Random Testing," *Proc. of Int. Test Conf.*, pp. 1031-1040, 1993.

[5] Brayton, R.K., R. Rudell, A. Sangiovanni-Vincentelli, A.R. Wang, "Multi-Level Logic Optimization and The Rectangular Covering Problem," *Proc. of Int. Conf. on Computer-Aided Design (ICCAD)*, pp. 66-69, 1987.

[6] Brayton, R.K., and F. Somenzi, "An Exact Minimizer for Boolean Relations," *Proc. of Int. Conf. on Computer-Aided Design (ICCAD)*, pp. 316-319, 1989.

[7] Daehn, W., and J. Muncha, "Hardware Test Pattern Generation for Built-In Testing," *Proc. of Int. Test Conf.*, pp. 110-113, 1981.

[8] Dandapani, R., J. Patel, and J. Abraham, "Design of Test Pattern Generators for Built-In Test," *Proc. of Int. Test Conf.*, pp. 315-319, 1984.

[9] Dufanza, C., and G. Cambon, "LFSR based Deterministic and Pseudo-Random Test Pattern Generator Structures," *Proc. of European Test Conf.*, pp. 27-34, 1991.

[10] Edirisooriya, G., and J.P. Robinson, "Design of Low Cost ROM Based Test Generators," *Proc. of VLSI Test Symp.*, pp. 61-66, 1992.

[11] 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.

[12] Hartmann, J., and G. Kemnitz, "How to Do Weighted Random Testing for BIST," *Proc. of Int. Conf. on Computer-Aided Design (ICCAD)*, pp. 568-571, 1993.

[13] Konemann, B., J. Mucha, and G. Zwiehoff, "Built-in Logic Block Observation Technique," *Proc. of Int. Test Conf.*, pp. 140-150, 1979.

[14] Konemann, B., "LFSR-Coded Test Patterns for Scan Design," *Proc. of European Test Conf.*, pp. 237-242, 1991.

[15] Krasniewski, A., and S. Pilarski, "Circular Self-Test Path: A Low-Cost BIST Technique for VLSI Circuits," *IEEE Trans. on Computer-Aided Design*, Vol. 8, No. 1, pp. 46-55, Jan. 1989.

[16] Lempel, M., S.K. Gupta, and M.A. Breuer, "Test Embedding with Discrete Logarithms," *Proc. of VLSI Test Symp.*, pp. 74-80, 1994.

[17] Pan, R., N.A. Touba, and E.J. McCluskey, "The Effect of Fault Dropping on Fault Simulation Time," *Tech. Report 93-5*, Center for Reliable Computing, Stanford Univ., Stanford, CA, Nov. 1993.

[18] Pomeranz, I., and S.M. Reddy, "3-Weight Pseudo-Random Test Generation Based on a Deterministic Test Set for Combinational and Sequential Circuits," *IEEE Trans. on Computer-Aided Design*, Vol. 12, No. 7, pp. 1050-1058, Jul. 1993.

[19] Schnurmann, H.D., E. Lindbloom, and R.G. Carpenter, "The Weighted Random Test-Pattern Generator," *IEEE Trans. on Computers*, Vol. C-24, No. 7, pp. 695-700, Jul. 1975.

[20] Touba, N.A., and E.J. McCluskey, "Automated Logic Synthesis of Random Pattern Testable Circuits," *Proc. of Int. Test Conf.*, pp. 174-183, 1994.

[21] Venkataramann, S., J. Rajski, S. Hellebrand, and S. Tarnick, "An Efficient BIST Scheme Based on Reseeding of Multiple Polynomial Linear Feedback Shift Registers," *Proc. of Int. Conf. on Computer-Aided Design (ICCAD)*, pp. 572-577, 1993.

[22] Wunderlich, H.-J., "Self-Test Using Unequiprobable Random Patterns," *Proc. of FTCS-17*, pp. 258-263, 1987.

[23] Wunderlich, H.-J., "Multiple Distributions for Biased Random Test Patterns," *Proc. of Int. Test Conf.*, pp. 236-244, 1988.