SYNTHESIS OF MAPPING LOGIC FOR GENERATING 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, California 94305-4055

ABSTRACT

During built-in self-test (BIST), the set of patterns generated by a pseudo-random pattern generator may not provide a sufficiently high fault coverage. This paper presents a new technique for synthesizing combinational mapping logic to transform the set of patterns that are generated. The goal is to satisfy test length and fault coverage requirements while minimizing area overhead. For a given pseudo-random pattern generator and circuit under test, there are many possible mapping functions that will provide a desired fault coverage for a given test length. This paper formulates the problem of finding a mapping function that can be implemented with a small number of gates as a one of finding a minimum rectangle cover in a binate matrix. A procedure is described for selecting a mapping function and synthesizing mapping logic to implement it. Experimental results for the procedure are compared with published results for other methods. It is shown that by performing iterative global operations, the procedure described in this paper generates mapping logic that requires less hardware overhead to achieve the same fault coverage for the same test length.

1. INTRODUCTION

Pseudo-random pattern testing is used in built-in self-test (BIST) because of its low hardware overhead. A linear feedback shift register (LFSR) or cellular automaton (CA) can be used to generate the pseudo-random patterns. LFSR's and CA's have simple structures which require small area overhead, and they can also be used as output response analyzers thereby serving a dual purpose. BIST techniques such as circular BIST [Krasniewski 89] and BILBO registers [Konemann 79] make use of these advantages to reduce overhead. Unfortunately, the pseudo-random patterns that are generated do not always provide a 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 [Eichelberger 83] or by redesigning it [Touba 94], and the other is to add logic to "bias" the patterns that are generated. As illustrated in Fig. 1, one approach for biasing the pseudo-random patterns is to add mapping logic at the output of the pseudo-random pattern generator to transform the original set of patterns produced by the generator into a new set of patterns that provides the desired fault coverage. Note that this is similar to weighted pseudo-random pattern testing [Bardell 87] except that the "weight" logic is generalized to be any function (not just those that weight signal probabilities). The architecture in Fig. 1 has the following advantages: it allows parallel test pattern application ("a test per clock"), it is easy to insert into an existing design (the mapping logic is simply placed between the pattern generating circuit and the CUT), and it can be used with any pattern generating circuit (e.g., LFSR, CA, BILBO register, etc.). This paper presents a new method for synthesizing the mapping logic so that it satisfies fault coverage requirements while minimizing area overhead.

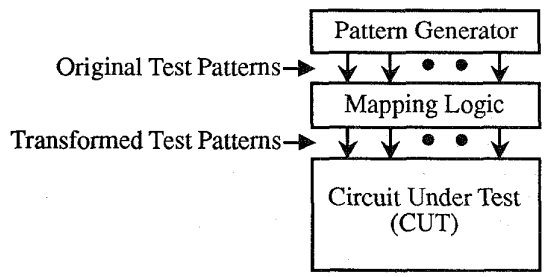

Figure 1. Block Diagram for Generation of Transformed Patterns

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*. For a given original pattern set and CUT, there are many possible mapping functions that will produce a transformed pattern set that provides the desired fault coverage. Finding a mapping function that can be implemented with a small number of gates is a challenging problem. In the past, the set of mapping functions considered was limited to those that weight signal probabilities at the inputs of the CUT thereby generating *weighted pseudo-random patterns* [Schnurmann 75], [Bardell 87], [Pomeranz 93], [Hartmann 93]. The advantage of using these mapping functions is that they can be determined through

INTERNATIONAL TEST CONFERENCE

probabilistic techniques and can be implemented with a reasonably small number of gates. Much research has been done on determining weights sets for a given CUT [Wunderlich 88], [Waicukauski 89], [Muradali 90], [Miranda 93]. Many circuits require multiple weight sets to satisfy test length and fault coverage requirements [Wunderlich 88]. Thus, the area overhead of the mapping logic required to implement the weight sets can be substantial.

Recent research considers a broader class of mapping functions in an attempt to reduce the area overhead required by the mapping logic. Two papers that describe methods for designing mapping logic were presented at the 1995 IEEE VLSI Test Symposium: [Touba 95] and [Chatterjee 95]. In [Chatterjee 95], a procedure is given for designing mapping logic that maps patterns in the original pattern set into new patterns that detect hard-to-detect faults. The heuristic of minimizing the number of inputs to the combinational unit for each CUT input is used to choose the mapping function. In [Touba 95], a procedure is described for designing mapping logic that is based on a special class of transformations called "cube mappings." Each cube mapping transforms a set of patterns in the original pattern set that doesn't detect any new faults into a new set of patterns that detects hard-to-detect faults. The mapping logic is designed to implement a set of cube mappings that is selected to satisfy fault coverage requirements. The advantages of using cube mappings are that sets of patterns are easier to decode than single patterns, and 100% fault coverage can be guaranteed because only patterns that don't detect any new faults are transformed. A procedure is described in [Dufaza 95] for designing mapping logic for a mixed-mode scheme. An LFSR is used to generate pseudo-random patterns, and then it is reconfigured as a ring counter to generate deterministic patterns through a network of OR gates.

This paper presents a more effective method for finding a mapping function to minimize overhead. The problem of choosing a mapping function is formulated as one of finding a minimum rectangle cover in a binate matrix. A heuristic procedure involving EXPAND, IRREDUNDANT, and REDUCE operations (analogous to what is used in ESPRESSO [Brayton 84]), is used to minimize a rectangle cover that corresponds to a mapping function that can be implemented by a small amount of mapping logic. By performing global operations, the procedure is able to find better mapping functions thereby synthesizing mapping logic that requires less hardware overhead than other methods.

The paper is organized as follows: In Sec. 2, a method for specifying a mapping function that satisfies test length and fault coverage requirements is described. In Sec. 3, it is shown how the mapping logic can be constructed from bit-fixing transformations that correspond to a set of rectangles in a binate matrix. In Sec. 4, the procedure for selecting a mapping function that minimizes area overhead is described. In Sec. 5, the process of synthesizing the mapping logic is explained. In Sec. 6, experimental results are presented and compared with previously published results. Sec. 7 is a conclusion.

2. SPECIFYING A MAPPING FUNCTION

This section describes a procedure for specifying a function that maps the original pattern set into a new pattern set. The mapping function is specified in a way that guarantees that it will produce a new pattern set that achieves a desired fault coverage for a given pattern generating circuit and test length.

The first step is to simulate the pattern generating circuit for the given test length to generate the original pattern set. Then fault simulation is performed on the CUT for the original pattern set to identify the undetected faults and the set of patterns that caused faults to be dropped (i.e., be detected for the first time). An automatic test pattern generation (ATPG) tool is then used to obtain *test cubes* (i.e., test patterns in which the unspecified inputs are left as don't cares) for the undetected faults.

Given a set of test cubes for the undetected faults, the original set of patterns, and the set of patterns that dropped faults, a mapping function can be specified by assigning to each test cube a pattern in the original pattern set that didn't drop any faults. Each original pattern that caused a fault to be dropped is mapped to itself, each original pattern assigned to a test cube is mapped to the test cube, and the remaining patterns are don't cares. Fig. 2 shows an example of specifying a mapping function that maps the original pattern set into a new transformed pattern set that provides a 100% fault coverage. The original patterns that caused faults to be

	Original Patterns $x_1 x_2 x_3 x_4 x_5 x_6$		Transformed Patterns $x_1 x_2 x_3 x_4 x_5 x_6$
Patterns That Drop Faults	0 1 0 0 1 1	→	0 1 0 0 1 1
	0 1 1 0 0 0	→	0 1 1 0 0 0
	1 0 1 1 0 1	→	1 0 1 1 0 1
	0 1 0 1 1 0	→	0 1 0 1 1 0
	1 0 1 1 0 1	→	1 0 1 1 0 1
Patterns Assigned to Test Cubes	1 0 1 0 1 1	→	0 0 1 0 X 1
	0 1 1 1 0 1	→	X 0 1 1 0 0
	1 0 0 1 1 1	→	1 X X 0 1 X
Unassigned Patterns	1 0 0 1 1 0	→	X X X X X X
	0 0 1 1 0 1	→	X X X X X X
	0 1 0 1 1 1	→	X X X X X X
	1 1 0 1 0 0	→	X X X X X X
	1 0 0 0 0 1	→	X X X X X X
	1 1 1 0 0 1	→	X X X X X X
	0 0 1 0 1 0	→	X X X X X X

Figure 2. Example of Specifying a Mapping Function.

dropped are mapped to themselves to ensure that all of the faults that were detected by the original pattern set are also detected by the transformed pattern set. The patterns that were assigned to test cubes are mapped to a pattern that matches the test cube so that all of the faults that are not detected by the original pattern set will be detected by the transformed pattern set. These mappings are sufficient to ensure that the transformed pattern set will detect all faults, so it doesn't matter what the remaining original patterns are mapped to.

There are many possible mapping functions depending on how the original patterns are assigned to the test cubes. The amount of logic required to implement each possible mapping function varies greatly. Thus, the problem of minimizing the mapping logic involves careful selection of the mapping function.

3. MAPPING LOGIC MINIMIZATION

In order to map pattern X into pattern Y, each bit in pattern X that differs from the corresponding bit in pattern Y must be "fixed" so that it matches. For example, if the pattern *0010* is being mapped into the pattern *1000*, then the first bit must be fixed to a *'1'*, and the third bit must be fixed to a *'0'*. So for original patterns that are mapped into different patterns (i.e., those assigned to test cubes), the mapping logic must fix the value of some of the bits so that the original pattern matches the test cube. An example of "bit-fixing" logic is shown in Fig. 3. There is some "bit-fixing function" that is active for some set of original patterns (in this case, *0010*, *0101*, and *0111*). When the bit-fixing function is active, it fixes some of the outputs to a specific logic value (in this case, x_1 = *'1'*, x_3 = *'0'*, x_4 = *'0'*). So in this example, the bit-fixing logic maps the original patterns *0010*, *0101*, and *0111* into the patterns *1000*, *1100*, and *1100*, respectively. The strategy for minimizing the mapping logic is to select the mapping function in a way that minimizes the number of bit-fixing functions that are required. This section describes a technique for finding a set of bit-fixing functions to implement a given mapping function, and based on this technique, the next section gives an iterative procedure for selecting a mapping function that minimizes the number of bit-fixing functions.

The problem of finding the minimum number of bit-fixing functions required to implement a given mapping function can be formulated as one of finding a minimum rectangle cover in a binate matrix (a similar idea to what is done in [Brayton 87]). A binate matrix B, where $B_{ij} \in \{0,1\}$, is formed in which each test cube is represented by a row. There is a complemented and uncomplemented column corresponding to each input in the test cube. If an input in a test cube is a *'0'*, then its corresponding complemented and uncomplemented column entries are set equal to 1 and 0, respectively. If an input in a test cube is a *'1'*, then its corresponding complemented and uncomplemented column entries are set equal to 0 and 1, respectively. If an input in a test cube is a don't care (*'X'*), then its corresponding complemented and uncomplemented column entries are both set equal to 1. An entry in B is stared if it corresponds to a bit difference between the assigned original pattern and the test cube. A *rectangle* in B is a subset of rows R and a subset of columns C such that $B_{ij} = 1$ for all $i \in R$ and $j \in C$. A rectangle in B corresponds to a common bit-fixing function among the outputs of the mapping logic. In the example in Fig. 3, the rectangle that

Original Patterns					Test Cubes			
x_1	x_2	x_3	x_4		x_1	x_2	x_3	x_4
1	1	1	1	→	0	0	1	1
0	0	1	0	→	X	0	0	0
0	1	0	1	→	1	X	X	0
0	1	1	1	→	1	1	0	X

B-Matrix

x_1'	x_2'	x_3'	x_4'	x_1	x_2	x_3	x_4
1*	1*	0	0	0	0	1	1
1	1	1*	1	1	0	0	0
0	1	1	1*	1*	1	1	0
0	0	1*	1	1*	1	0	1

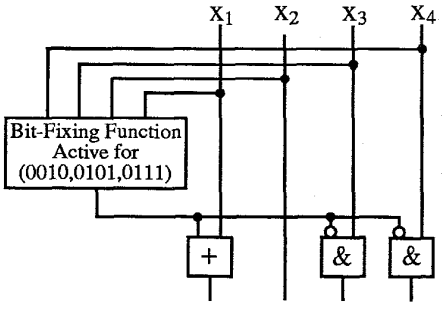

Figure 3. Example of a Rectangle in the B-Matrix and its Corresponding Bit-Fixing Logic

is shown corresponds to a bit-fixing function that is active if any of the assigned patterns that correspond to the rows in the rectangle, i.e., *0010*, *0101*, and *0111*, are applied to the mapping logic. When the bit-fixing function is active, it fixes the outputs that correspond to the columns in the rectangle to a specific logic value, i.e., x_1 = '1', x_3 = '0', x_4 = '0'. Thus, the rectangle in Fig. 3 corresponds to a transformation in which the original patterns *0010*, *0101*, and *0111* are mapped into the patterns *1000*, *1100*, and *1100*, respectively, which detect the faults corresponding to the test cubes *X000*, *1XX0*, and *110X*, respectively. So each rectangle in *B* corresponds to a transformation that can be implemented by bit-fixing logic that forces specific logic values at a set of outputs. In order for the mapping logic to transform all of the assigned patterns so that they match their respective test cubes, each bit difference (which corresponds to a stared entry in *B*) must be contained in a rectangle that is implemented by the mapping logic. The mapping logic can be designed by finding a set of rectangles that covers all of the stared entries in *B* and then constructing bit-fixing logic to implement the transformations corresponding to those rectangles.

4. SELECTING A MAPPING FUNCTION

In the previous section, it was shown how mapping logic can be constructed by finding a set of rectangles that cover all of the stared entries in *B* where each rectangle corresponds to some bit-fixing logic. Based on this correspondence, the problem of minimizing the mapping logic can be formulated as one of finding a minimum set of rectangles that cover all of the stared entries in *B*. The set of stared entries in *B* depends on which original pattern is assigned to each test cube. Therefore, the problem of selecting a mapping function to minimize the amount of mapping logic corresponds to assigning original patterns to each test cube in a way that minimizes the number of rectangles required to cover all the resulting stared entries. A heuristic procedure for solving this problem is described in this section.

4.1 Procedure for Minimizing Mapping Logic

The procedure begins with a set of rectangles, *R*, that covers all the stared entries in the *B*-matrix. Then it iterates using an EXPAND, IRREDUNDANT, REDUCE sequence analogous to what is used in ESPRESSO [Brayton 84] to reduce the number of rectangles. The key step is that after the set of rectangles is made prime and irredundant by the EXPAND and IRREDUNDANT procedures, the mapping function is changed by reassigning original patterns to test cubes in order to eliminate the need for some of the rectangles in *R*. This process continues until no further reduction in the number of rectangles in *R* is achieved. The steps of the procedure are described below:

1. Form the *B* matrix from the set of test cubes.
   Each test cube corresponds to a row in the *B*-matrix.

2. Do an initial assignment of the original patterns to test cubes based on minimizing the number of bit differences.
   Each test cube is compared with the set of unassigned patterns in the original pattern set, and the pattern that differs in the smallest number of bits is assigned to the test cube. This minimizes the total number of stared entries in the *B*-matrix.

3. Let each stared entry in the *B*-matrix be a rectangle in the set of rectangles, *R*.
   This ensures that the initial set of rectangles covers all of the stared entries.

4. EXPAND(*R*) - expand each rectangle in *R* so that all the rectangles are prime rectangles.
   A *prime rectangle* is a rectangle that is not contained in another rectangle. The EXPAND procedure uses the heuristic of expanding each rectangle so that it covers as many other rectangles as possible.

5. IRREDUNDANT(R) - eliminate rectangles in *R* that are covered by other rectangles in *R*.
   The IRREDUNDANT procedure goes through each rectangle in *R* from the smallest to the largest and checks to see if the rectangle covers any stared entries that are not covered by any other rectangles. If not, then the rectangle can be removed without uncovering any stared entries, so it is eliminated from *R*.

6. Reassign the original patterns to the test cubes based on eliminating as many rectangles in *R* as possible.
   An attempt is made to eliminate the need for some rectangles in *R* by changing the location of the stared entries. This is done by reassigning the original patterns to the test cubes. For each rectangle in *R*, an attempt is made to eliminate all of the stared entries that it alone covers by reassigning the original patterns to the test cubes. A new original pattern can be assigned to a test cube provided the resulting stared entries are all covered by rectangles in *R*. If a new assignment can be found such that some rectangle in *R* no longer covers any stared entries, then that rectangle can be removed from *R*.

7. REDUCE(*R*) - reduce each rectangle in *R* as much as possible while still covering all stared entries in the *B*-matrix.
   Each prime rectangle in *R* is reduced as much as possible so that it can be either re-expanded in the next iteration or implemented by the mapping logic.

8. Loop back to step 4 if the number of rectangles in *R* has decreased.
   The procedure keeps looping back until no further reduction in the number of rectangles in *R* is obtained.

The procedure selects the mapping function by reassigning original patterns to test cubes in a way that minimizes the number of rectangles in *R*. When the procedure is complete, the mapping logic can be synthesized so that it implements the transformations corresponding to each rectangle in *R*; this is explained in Sec. 5.

4.2 Example

An example will be shown to illustrate the steps of the procedure. The initial set of rectangles corresponding to a *B*-Matrix is shown in Fig. 4. There is one rectangle for each stared entry. The EXPAND procedure is performed on the initial set of rectangles to expand them into prime rectangles. Fig. 5 shows the prime rectangles that are generated by the EXPAND procedure. The IRREDUNDANT procedure is performed to eliminate any rectangles that can be removed without uncovering any stared entries. The result of the IRREDUNDANT procedure is shown in Fig. 6 -- one rectangle was eliminated. Then an attempt is made to reassign the original patterns that are matched with each test cube in order to change the location of the stared entries so that rectangles can be eliminated. The result of the reassign step is shown in Fig. 7. By assigning the pattern *0111* to the test cube *110X* (therefore replacing the pattern *1011*), the stared entries for the last row in the rectangle are changed such that a rectangle can be eliminated. The REDUCE procedure is performed to reduce the size of the rectangles as much as possible without uncovering any stared entries. The result of the REDUCE operation is shown in Fig. 8 -- one rectangle was reduced in size. The procedure can then be repeated on the resulting set of rectangles to try to further minimize it.

Original Patterns					Test Cubes				B-Matrix							
x_1	x_2	x_3	x_4		x_1	x_2	x_3	x_4	x_1'	x_2'	x_3'	x_4'	x_1	x_2	x_3	x_4
1	1	1	1	→	0	0	1	1	1*	1*	0	0	0	0	1	1
0	0	1	0	→	X	0	0	0	1	1	1*	1	1	0	0	0
0	1	0	1	→	1	X	X	0	0	1	1	1*	1*	1	1	0
1	0	1	1	→	1	1	0	X	0	0	1*	1	1	1*	0	1

Figure 4. Initial Set of Rectangles.

Original Patterns					Test Cubes				B-Matrix							
x_1	x_2	x_3	x_4		x_1	x_2	x_3	x_4	x_1'	x_2'	x_3'	x_4'	x_1	x_2	x_3	x_4
1	1	1	1	→	0	0	1	1	1*	1*	0	0	0	0	1	1
0	0	1	0	→	X	0	0	0	1	1	1*	1	1	0	0	0
0	1	0	1	→	1	X	X	0	0	1	1	1*	1*	1	1	0
1	0	1	1	→	1	1	0	X	0	0	1*	1	1	1*	0	1

Figure 5. Set of Rectangles after EXPAND.

Original Patterns					Test Cubes				B-Matrix							
x_1	x_2	x_3	x_4		x_1	x_2	x_3	x_4	x_1'	x_2'	x_3'	x_4'	x_1	x_2	x_3	x_4
1	1	1	1	→	0	0	1	1	1*	1*	0	0	0	0	1	1
0	0	1	0	→	X	0	0	0	1	1	1*	1	1	0	0	0
0	1	0	1	→	1	X	X	0	0	1	1	1*	1*	1	1	0
1	0	1	1	→	1	1	0	X	0	0	1*	1	1	1*	0	1

Figure 6. Set of Rectangles after IRREDUNDANT.

Original Patterns					Test Cubes				B-Matrix							
x_1	x_2	x_3	x_4		x_1	x_2	x_3	x_4	x_1'	x_2'	x_3'	x_4'	x_1	x_2	x_3	x_4
1	1	1	1	→	0	0	1	1	1*	1*	0	0	0	0	1	1
0	0	1	0	→	X	0	0	0	1	1	1*	1	1	0	0	0
0	1	0	1	→	1	X	X	0	0	1	1	1*	1*	1	1	0
0	1	1	1	→	1	1	0	X	0	0	1*	1	1*	1	0	1

Figure 7. Set of Rectangles after Reassigning Original Patterns (*1 0 1 1* replaced by *0 1 1 1*).

Original Patterns					Test Cubes				B-Matrix							
x_1	x_2	x_3	x_4		x_1	x_2	x_3	x_4	x_1'	x_2'	x_3'	x_4'	x_1	x_2	x_3	x_4
1	1	1	1	→	0	0	1	1	1*	1*	0	0	0	0	1	1
0	0	1	0	→	X	0	0	0	1	1	1*	1	1	0	0	0
0	1	0	1	→	1	X	X	0	0	1	1	1*	1*	1	1	0
0	1	1	1	→	1	1	0	X	0	0	1*	1	1*	1	0	1

Figure 8. Set of Rectangles after REDUCE.

5. SYNTHESIZING THE MAPPING LOGIC

After the mapping function has been selected and the set of rectangles that covers all the stared entries in the *B*-matrix has been minimized, the mapping logic can be synthesized. This is best explained with an example. In Fig. 9, the bit-fixing functions corresponding to the two rectangles in Fig. 8 are specified. For each rectangle, the assigned original patterns corresponding to the rows in the rectangle are placed in the on-set since the bit-fixing function must be active for those patterns. The patterns that drop faults and the patterns that are assigned to other test cubes should not be transformed because otherwise the fault coverage may be reduced, so the bit-fixing function shouldn't be active for those patterns, therefore they are added to the off-set. The on-set and off-set specify the bit-fixing function and can be passed to a logic synthesis tool to generate a logic implementation.

Fig. 10 shows an implementation for the set of rectangles in Fig. 8. The bit-fixing functions were derived as shown in Fig. 9. Each bit-fixing function forces the logic value at the outputs corresponding to the columns in its rectangle. When bit-fixing function 1, $(x_1\ x_2)$, is active, it forces x_1 = '0' and x_2 = '0'. This is implemented by adding AND gates to x_1 and x_2. When bit-fixing function 2, $[x_1'\ (x_2' + x_4)]$, is active, it forces x_1 = '1', x_3 = '0' and x_4 = '0'. This is implemented by adding an OR gate to x_1, and AND gates to x_3 and x_4.

Patterns that Drop Faults: *0100, 0110, 1011*

Patterns Assigned to Test Cubes: *1111, 0010, 0101, 0111*

Bit-Fixing Function 1:
- On-Set = *1111*
- Off-Set = *0100, 0110, 1011, 0010, 0101, 0111*
- Synthesized Logic $<x_1, x_2, x_3, x_4> = x_1\ x_2$

Bit-Fixing Function 2:
- On-Set = *0010, 0101, 0111*
- Off-Set = *0100, 0110, 1011, 1111*
- Synthesized Logic $<x_1, x_2, x_3, x_4> = x_1'(x_2' + x_4)$

Figure 9. Bit-Fixing Functions Corresponding to Set of Rectangles in Figure 8

One option for disabling the mapping logic during system operation is to AND in a test mode line with each bit-fixing function; this is shown in Fig. 10. An important issue is the delay during system operation. The delay through the mapping logic can be optimized by a logic synthesis tool. However, if the delay is unacceptable for some of the inputs, then another option is to bypass the mapping logic for those inputs during system operation by using multiplexors.

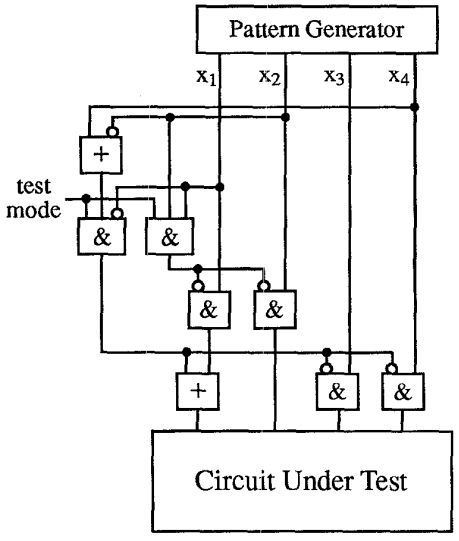

Figure 10. Implementation of Mapping Logic for Set of Rectangles in Figure 8

6. 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 [Brglez 85] and ISCAS 89 [Brglez 89] benchmark circuits that require over a million test patterns. 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).

6.1 Comparison with Weighted Pseudo-Random Pattern Methods

Table 1 compares the "rectangle mapping" method described in this paper with weighted pseudo-random pattern methods. The fault coverage is the same for all methods: 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, the second column shows the number of primary inputs, and the third column shows the test length for pseudo-random pattern testing using an LFSR alone. Then results are given for 3 different methods plus the rectangle mapping 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 [Hartmann 93] to reflect a static CMOS technology: *(0.5)(n)* GE's for a *n*-input NAND or NOR, *(2.5)(n-1)* GE's for a *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 estimate that is computed as follows:

Multiple Weight Sets: The weight sets from [Bershteyn 93] 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 [Pomeranz 93]. 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 [Pomeranz 93] (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 [AlShaibi 94]. 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 [AlShaibi 94].

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

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

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

Table 1. Comparison with Weighted Pseudo-Random Pattern Methods

Circuit	Num	PRand	Multiple Weight Sets [Bershteyn 93]				3-Weight [Pomeranz 93]			Fix-Biased [AlShaibi 94]			Rectangle Mapping		
Name	Inputs	TLen	TLen	WS	FF	GE	TLen	FF	GE	TLen	FF	GE	TLen	FF	GE
s420	35	1.1M	532	4	≥2	350	NA	NA	NA	5K	18	>7	500	0	37
			1.8K	2	≥1	245							1K	0	28
s641	54	1.0M	593	3	≥2	459	NA	NA	NA	19K	20	>21	500	0	22
													10K	0	12
s838	67	>100M	893	5	≥3	770	NA	NA	NA	86K	19	>14	850	0	86
			17K	2	≥1	469							10K	0	37
C2670	233	4.6M	1.3K	9	≥4	4078	19K	5	1507	19K	54	>259	1K	0	218
			12K	3	≥2	1981	30K	5	1316				5K	0	121
C7552	207	>100M	2K	12	≥4	4554	47K	6	3003	191K	111	>658	10K	0	186
			69K	5	≥3	2380	72K	6	2475				50K	0	139

6.2 Comparison with Cube Mapping Method

Table 2 compares results for the rectangle mapping method presented here with results for the cube mapping method presented in [Touba 95]. The test length and gate equivalents are shown for each circuit. The fault coverage is 100% of detectable faults for both methods. The cube mapping method uses a greedy approach for selecting cube mappings based on simulation. The rectangle mapping method performs iterative operations to select the mapping function based on more general bit-fixing transformations. Because the transformations used by the rectangle mapping method are more general and the selection process is iterative as opposed to greedy, the rectangle mapping method finds better mapping functions that result in less hardware overhead.

Table 2. Comparison with Cube Mapping Method.

Circuit Name	Num Inp	PRand TLen	Cube Mapping [Touba 95]		Rectangle Mapping	
			TLen	GE	TLen	GE
s420	35	1.1M	500	48	500	37
			1K	38	1K	28
s641	54	1.0M	500	23	500	22
			10K	12	10K	12
s838	67	>100M	850	99	850	86
			10K	59	10K	37
C2670	233	4.6M	1K	260	1K	218
			5K	155	5K	121
C7552	207	>100M	10K	214	10K	186
			50K	183	50K	139

6.3 Comparison with Method in [Chatterjee 95]

Table 3 compares results for the rectangle mapping method with the results given in [Chatterjee 95]. The test length, fault coverage for detectable faults, and gate equivalents are shown for both methods. Note that the method in [Chatterjee 95] doesn't achieve 100% fault coverage for some circuits. The rectangle mapping method uses more global operations in selecting the mapping function than the method in [Chatterjee 95]. This results in mapping logic that requires less hardware overhead.

Table 3. Comparison with Method in [Chatterjee 95].

Circuit Name	Num Inp	PRand TLen	[Chatterjee 95]			Rectangle Mapping		
			TLen	Cov	GE	TLen	Cov	GE
C880	60	15K	1K	99.3%	36	1K	100%	27
C1355	41	4K	3K	100%	46	3K	100%	11
C1908	33	10K	4K	100%	66	4K	100%	12
C2670	233	4.6M	5K	99.8%	157	5K	100%	121
C3540	50	20K	4.5K	99.6%	16	4.5K	100%	13
C7552	207	>100M	8K	96.6%	302	8K	100%	256

7. CONCLUSIONS

On-chip generation of weighted pseudo-random patterns involves adding weight logic to the output of the pseudo-random pattern generator to increase the probability that the hard-to-detect faults will be detected. The method described in this paper improves upon that idea in two ways: (1) the weight logic is generalized to be any mapping function, and (2) a deterministic procedure is used to synthesize the mapping logic to *guarantee* detection of all faults. By using an iterative procedure involving global operations, the synthesis method described in this paper generates mapping logic that requires less hardware overhead than previous methods (as indicated by the experimental results). In addition to reducing the overhead required for BIST, it provides the following advantages: allows parallel test pattern application, is easy to insert into an existing design (the mapping logic is simply placed between a pattern generating circuit and the CUT), and can be used with any pattern generating circuit (e.g., LFSR, CA, BILBO register).

Results in [Ma 95] indicate that test sets in which each single stuck-at fault is detected multiple times provide high defect coverage. The procedure described in this paper can be easily extended to transform a pseudo-random pattern set into one that detects each single stuck-at fault at least n times by simply specifying additional transforms in the mapping function.

ACKNOWLEDGEMENTS

This work was supported in part by the Advanced Research Projects Agency under prime contract No. DABT63-94-C-0045, and by the Ballistic Missle Defense Organization, Innovative Science and Technology (BMDO/IST) Directorate administered through the Department of Navy, Office of Naval Research under Grant No. N00014-92-J-1782, and by the National Science Foundation under Grant No. MIP-9107760.

REFERENCES

[AlShaibi 94] AlShaibi, M.F., and C.R. Kime, "Fixed-Biased Pseudorandom Built-In Self-Test for Random Pattern Resistant Circuits," *Proc. of International Test Conference*, pp. 929-938, 1994.

[Bardell 87] Bardell, P.H., W.H. McAnney, and J. Savir, *Built-In Test for VLSI: Pseudorandom Techniques*, New York: Wiley, 1987.

[Bershteyn 93] Bershteyn, M., "Calculation of Multiple Sets of Weights for Weighted Random Testing," *Proc. of International Test Conference*, pp. 1031-1040, 1993.

[Brayton 84] Brayton, R.K., G.D. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, Boston: Kluwer Academic Publishers. 1984.

[Brayton 87] Brayton, R.K., R. Rudell, A. Sangiovanni-Vincentelli, A.R. Wang, "Multi-Level Logic Optimization and The Rectangular Covering Problem," *Proc. of International Conference on Computer-Aided Design (ICCAD)*, pp. 66-69, 1987.

[Brglez 85] Brglez, F., and H. Fujiwara, "A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortan," *Proc. of International Symposium on Circuits and Systems*, pp. 663-698, 1985.

[Brglez 89] Brglez, F., D. Bryan, and K. Kozminski, "Combinational Profiles of Sequential Benchmark Circuits," *Proc. of International Symposium on Circuits and Systems*, pp. 1929-1934, 1989.

[Chatterjee 95] Chatterjee, M., and D.K. Pradhan, "A New Pattern Biasing Technique for BIST", *Proc. of VLSI Test Symposium*, pp. 417-425, 1995.

[Dufaza 95] Dufaza, C., H. Viallon, and C. Chevalier, "BIST Hardware Generator for Mixed Test Scheme," *Proc. of European Design and Test Conference*, 1995.

[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.

[Hartmann 93] Hartmann, J., and G. Kemnitz, "How to Do Weighted Random Testing for BIST," *Proc. of International Conference on Computer-Aided Design (ICCAD)*, pp. 568-571, 1993.

[Konemann 79] Konemann, B., J. Mucha, and G. Zwiehoff, "Built-in Logic Block Observation Technique," *Proc. of International Test Conference*, pp. 140-150, 1979.

[Krasniewski 89] Krasniewski, A., and S. Pilarski, "Circular Self-Test Path: A Low-Cost BIST Technique for VLSI Circuits," *IEEE Transactions on Computer-Aided Design*, Vol. 8, No. 1, pp. 46-55, Jan. 1989.

[Ma 95] Ma, S.C., P. Franco, and E.J. McCluskey, "An Experimental Test Chip to Evaluate Test Techniques: Experimental Results," *Proc. of International Test Conference*, 1995.

[Miranda 93] Miranda, M.A., and C.A. Lopez-Barrio, "Generation of Single Distributions of Weights for Random Built-In Self-Test," *Proc. of International Test Conference*, pp. 1023-1030, 1993.

[Muradali 90] Muradali, F., V.K. Agarwal, B. Nadeau-Dostie, "A New Procedure for Weighted Random Built-In Self-Test," *Proc. of International Test Conference*, pp. 660-668, 1990.

[Pomeranz 93] Pomeranz, I., and S.M. Reddy, "3-Weight Pseudo-Random Test Generation Based on a Deterministic Test Set for Combinational and Sequential Circuits," *IEEE Transactions on Computer-Aided Design*, Vol. 12, No. 7, pp. 1050-1058, Jul. 1993.

[Schnurmann 75] Schnurmann, H.D., E. Lindbloom, and R.G. Carpenter, "The Weighted Random Test-Pattern Generator," *IEEE Transactions on Computers*, Vol. C-24, No. 7, pp. 695-700, Jul. 1975.

[Touba 94] Touba, N.A., and E.J. McCluskey, "Automated Logic Synthesis of Random Pattern Testable Circuits," *Proc. of International Test Conference*, pp. 174-183, 1994.

[Touba 95] Touba, N.A., and E.J. McCluskey, "Transformed Pseudo-Random Patterns for BIST," *Proc. of VLSI Test Symposium*, pp. 410-416, 1995.

[Waicukauski 89] Waicukauski, J.A., E. Lindbloom, and O. Forlenza, "A Method for Generating Weighted Random Test Patterns," *IBM Journal of Research and Develop.*, Vol. 33, No. 2, pp. 149-161, March 1989.

[Wunderlich 88] Wunderlich, H.-J., "Multiple Distributions for Biased Random Test Patterns," *Proc. of International Test Conference*, pp. 236-244, 1988.