JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 16, 719-731 (2000)

Testability Improvement by Branch Point Control for Conditional Statements With Multiple Branches

SYING-JYAN WANG AND CHIA-CHUN LIEN
Institute of Computer Science
National Chung-Hsing University
Taichung, Taiwan 402, R.O.C.
E-mail: sjwang@cs.nchu.edu.tw

High-level test synthesis (HLTS) methodologies have attracted much many research interest in recent years as digital design has moved to higher levels of abstraction. Conditional statements in behavioral descriptions tend to produce testability problems and have to be taken care of in the early stage of the design cycle. In this paper, we present an HLTS methodology for the Built-In Self-Test (BIST) environment. Our methods modify conditional case statements in the original design so as to control the number of test patterns applied to modules being tested. As a result, the number of required test patterns can be greatly reduced. This method is especially useful when there is a wide variance in the number of random test patterns required for functional units. Experimental results show that our methods achieve a high degree of fault coverage with a much smaller number of test patterns while the area and time overheads are negligible.

Keywords: VLSI, high-level test synthesis, behavioral statement, BIST, conditional branch

1. INTRODUCTION

The fast increase in VLSI density has created a great challenge for the design and testing of VLSI circuits. Given the complexity of current VLSI circuits, it is very difficult to control or observe signals inside a chip, which in turn makes circuit testing difficult. Design-for-testability (DFT) methodologies have thus, been developed to reduce the cost and improve the quality of VLSI testing. The two most well-known and widely used DFT techniques are the scan design and Built-In-Self-Test (BIST) techniques.

The complexity of circuits is also pushing digital circuit design toward higher levels of abstraction. CAD tools that accept and optimize designs specified in the Register-Transfer Level (RTL) have been used for years while high-level synthesis, which translates designs specified in the behavioral domain to the structural domain (RTL), is becoming popular. In order to consider testability issues in the early stage of a design cycle, high-level test synthesis has been studied intensively in recent years.

High-level synthesis for testability (HLTS) tries to transform a design description into another equivalent one with the same functionality and improved testability [1]. The original design may be specified with either RTL or behavior or description. Many HLTS techniques have been proposed, including techniques at the RT level [2-8] and behavioral level [9-13].

Received July 3, 1999; revised December 23, 1999 & March 28, 2000; accepted May 5, 2000.
Communicated by Kuen-Jong Lee.

Behavioral synthesis for testability can be focused on ATPG [9] or BIST [10-13]. These techniques try to find behavioral statements that may cause testability problems and modify the statements for better testability. Conditional statements are the ones that are most likely to cause testability problems. These statements include conditional *loop* statements [9], *if-then-else* statements [13], and *case* statements [16].

Conditional *case* statements are commonly used in behavioral and RTL descriptions to provide multiple branch points. This kind of statement may decrease testability in the BIST environment, as the probability that a branch will be taken may not reflect the testability of corresponding module. In other words, some modules may not be provided with enough test patterns while others may be given too many.

In this paper, we present methods to deal with the aforementioned testability problem in the BIST-based environment. Our methods modify the design specified in the RTL level so that the testing time can be greatly reduced while the area and time penalty remains negligible. This paper is organized as follows. The testability problems caused by case statements in the BIST environment are discussed in section 2, and our approach to this problem is presented in section 3. We have synthesized circuits according to the presented methods, and the experimental results are given in section 4. The results show that our methods effectively solve the testability problem with insignificant extra cost. Furthermore, the overhead diminishes as the data path becomes wider. Concluding remarks are given in section 5.

2. TESTABILITY PROBLEMS

The Built-In Self-Test (BIST) is a widely used DFT technique [14]. In a circuit with BIST, test patterns are generated on chip, and output responses are also analyzed on chip. In order to achieve this goal, the BIST structure reconfigures part of the functional circuit as a test pattern generator (TPG) and another part as an output response analyzer (ORA). The rest of the circuit consists of the circuit under test (CUT). The TPG is usually made up of a linear feedback shift register (LFSR). The test patterns generated by the TPG are fed to the CUT while the output responses are collected and analyzed by the ORA. The most popular ORA design is the multi-input signature register (MISR), which is also an LFSR. Fig. 1 shows a simple illustration of a general BIST structure.

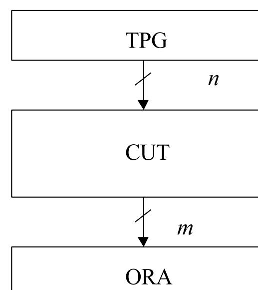


Fig. 1. The BIST structure.

The BIST structure discussed above may encounter some testability problems under some special circumstances. For example, consider the simple ALU structure shown in Fig. 2. The ALU has two inputs coming from registers R_1 and R_2, and the output is stored in R_3. Furthermore, the function executed by the ALU is controlled by the control input R_4, which is usually generated by the control circuit.

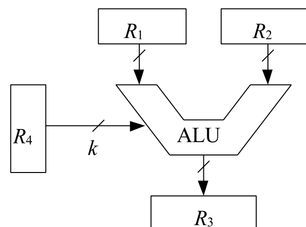


Fig. 2. A simple ALU.

In order to exhaustively test the ALU, registers R_1, R_2, and R_4 should become a test pattern generator which generates an exhaustive number of test patterns for the ALU. With exhaustive testing, all non-redundant stuck-at faults in the circuit under test (ALU in this example) are guaranteed to be detectable. The only problem is that the number of test patterns is usually prohibitive.

In order to reduce the number of test vectors needed and thus reduce the test time, either pseudorandom testing or pseudoexhaustive testing is most helpful. With these strategies, the registers can be configured in various ways. For example, during the BIST session, R_1 and R_2 may become, or the contents in them may come from, a single TPG while register R_4 is an independent TGP. When pseudorandom test patterns are applied, fault simulation is usually conducted first to decide the number of test patterns needed to achieve a required level of fault coverage.

Now suppose that the number of different functions implemented in the ALU is N. From the point of view of pseudorandom testing, the ideal case is $N = 2^k$, where each function corresponds to exactly one control code. In this case, when we apply L test patterns to the ALU, each functional unit is exercised by about $L/2^k$ patterns. However, in real circuits, it is possible that we will have $N < 2^k$. In this case, a functional unit may be exercised by more than one control code, and some control codes may be unused (i.e., they do not activate any functional unit). This will probably lead to some testability problems discussed below.

- Suppose that there are unused control codes (stored in register R_4 as shown in Fig. 2). In this case, some test cycles will be wasted during BIST sessions since all the patterns will be generated in R_4 but none of the functional units will be activated for the unused codes.
- Suppose that functional unit FU_i can be activated by m_i different control codes, where $2^k > m_i \geq 1$. Therefore, the number of test patterns accepted by FU_i will be around $L \times m_i/2^k$ during a BIST session.

The above problems can be illustrated with the example shown in Fig. 3. Fig. 3(a) shows a piece of Verilog code, and Fig. 3(b) shows a block diagram of the circuit synthesized from the code in Fig. 3(a).

In order to implement BIST in this circuit, registers B and C become a single PRPG during the testing time, and register A becomes an independent PRPG. Let both registers B and C be m-bit wide; in all there are 2^{2m} different input combinations.

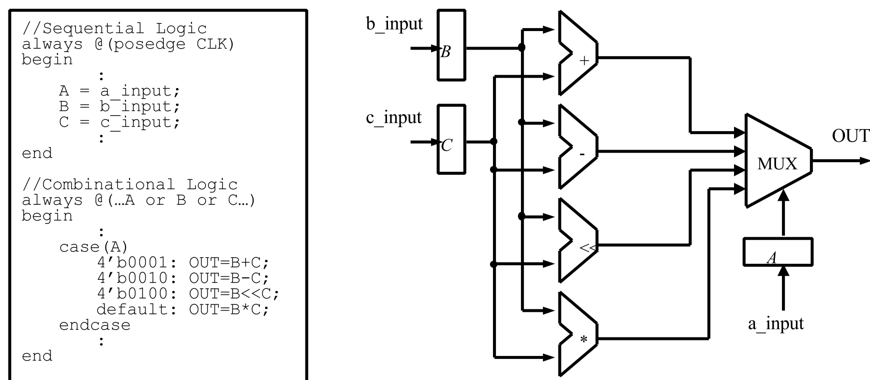


Fig. 3. (a) A description with a case statement, (b) the corresponding circuit.

Among the four functional units, only the multiplier can be activated by 13 different control codes while each of the other three units can be activated by exactly one control code. Therefore, with randomly generated control code in register *A*, the probability that the multiplier will be activated is 13/16 while the probability that any one of the other three units will be activated is 1/16.

The number of test vectors required for each unit depends on the actual circuit implementation of the unit. If all the units are designed separately, which is the case for ALU and many datapath designs, the number of test vectors required is known. If the whole circuit is to be synthesized using tools, then there are two ways to decide the number of test vectors required. We can either estimate the number of test vectors needed for each unit, or we can synthesize all the units separately. Although the number of test vectors for separately synthesized units may be different from the number of vectors needed when all the units are synthesized as a whole, it usually serves as a very good estimator.

We synthesize the four functional units separately with an 8-bit data-path (i.e., both registers *B* and *C* have 8-bit width). We apply pseudorandom test patterns to each unit and then conduct fault simulation to see how many random patterns a unit requires. The results are shown in Table 1.

In Table 1, we give the size of each unit (Gate Count), the number of faults in the unit (#Faults), and the number of deterministic test patterns obtained by the fault-oriented test generation algorithm. To each unit is applied random test patterns, and the number of applied test patterns (#Applied Test Patterns), the number of detected faults corresponding to the applied patterns (#Detected Faults), and the calculated fault coverage (Fault Cov.) are all listed in the table. From the results shown in Table 1, it should be obvious that the number of random patterns required to achieve 100% fault coverage may vary widely for different units, and that the number of test patterns required is not directly related to the size of the given unit. In particular, the shifter unit is very resistant to random test patterns.

From Table 1, we can see that we need no more than 4460 (=30+30+400+4000) random test patterns to achieve 100% fault coverage if all the units can be tested separately. However, since the distribution of test vectors is controlled by register A, and only 1/16 of all the test vectors are applied to the shifter, many more vectors are required. Let the total number of test vectors required to achieve 100% fault coverage be *T*. Thus, we have:

Table1. Number of random test patterns required for each unit.

Circuit	Gate Count	#Inputs	#Faults	#Required Deterministic Test Patterns	#Applied Random Patterns	#Detected Faults	Fault Cov. (%)
Adder	74	16	226	9	30	226	100
Subtractor	94	16	236	12	30	236	100
Multiplier	676	16	1476	37	50	1658	98.7
					100	1673	99.6
					200	1679	99.9
					400	1680	100
Shifter	67	16	192	27	50	126	65.6
					100	145	75.5
					200	163	84.9
					400	171	89.0
					800	181	94.3
					1000	181	94.3
					2000	188	97.9
					4000	192	100

(1) $T/16 \geq 30$ for the adder,
(2) $T/16 \geq 30$ for the subtracter,
(3) $T/16 \geq 4000$ for the shifter, and
(4) $13T/16 \geq 400$ for the multiplier.

As a result, we need $T \geq 64000$ random test patterns, in which more than 59540 patterns, or 93% of all applied patterns, are wasted because of the way the test patterns are distributed.

A partial solution to this problem is to rearrange the corresponding control codes such that more test patterns can be applied to the unit that is most difficult to test. Whenever this approach is possible, the number of test vectors can be greatly reduced. For example, we may rewrite the case statement in Fig. 3(a) as follows:

```
case (A)
        4'b0001: out = B + C;
        4'b0010: out = B – C;
        4'b0100: out = B * C;
        default:   out = B << C;
endcase
```

Fig. 4. A modified Verilog code.

With this modification, the shifter will be applied with 13/16 of all the generated test patterns. Thus, equations (3) and (4) above will be changed to

(3a) $T/16 \geq 400$ for the multiplier, and
(4a) $13T/16 \geq 4000$ for the shifter.

For this new design, we shall have $T \geq 6400$ from (3a), which is much better than the previous results. However, we still have to apply 1940 more test vectors compared to the original set of test patterns, which increases the test set by 43.5%. Furthermore, this approach is not always possible. Thus, we need to use a systematic approach to solve this problem. Our solution will be proposed in the next section.

3. TESTABILITY ENHANCEMENT

From the discussion given above, it is clear that we should be able to distribute test vectors to functional units according to their requirements in order to improve testability under the BIST environment. In this section, we will present our approach to this problem. The efficiency of this method will be presented in section 4.

3.1 Branch Point Selection

The testability problem described in section 2 appears because the condition code (stored in register *A*) is generated by a PRPG. As a result, we have very little control over which functional unit is actually selected. In order to solve this problem, we must be able to freely select the unit to be tested in test mode. This requires a different way to design the BIST control circuit.

In this section, we shall use the simple ALU modeled by the Verilog code shown in Fig. 4 as an example to demonstrate how our method works. In our method, all the functional units are tested sequentially. For example, for the case statement shown in Fig. 4, we test the adder first (with 30 test patterns), then the subtractor (30 patterns), multiplier (400 patterns), and shifter (4000 patterns). In other words, our method replaces register A with a special control circuit rather than a PRPG in test mode.

First, we generate a sequence of pseudorandom test patterns using a PRPG. Fault simulation is conducted to decide how many patterns are sufficient for each unit. The required number of test patterns and the corresponding branch point (that is, the last test vector applied to a functional unit) are shown in Table 2.

Table 2. Circuits under test and their branch point.

Circuit	Gate Count	#Test Patterns	Fault Cov. (%)	First Test Pattern	Last Test Pattern	Branch Point
Adder	74	30	100	1	30	1110000101101011
Subtractor	94	30	100	31	60	0011000001101010
Multiplier	676	400	100	61	460	1111110001011001
Shifter	67	4000	100	461	4460	1110111111100000

All 4460 patterns are required to achieve 100% fault coverage for the whole module. If we test the functional units in the order shown in Table 2, we will apply the first 30 patterns to the adder, patterns number 31 to 60 to the subtractor, patterns number 61 to 460 to the multiplier, and the remaining patterns to the shifter. As a result, patterns number 30, 60, 460, and 4460 are the branch points. Whenever a branch point is reached, the current unit under test (UUT) has been fully tested, and the next test pattern should be directed to the next unit. The last column in Table 2 shows the corresponding branch point for the four functional units.

It should be noted that the last branch point (i.e., test pattern number 4460) is not actually required in our method since the test process is finished once this point is reached. However, this point can be used to signal the end of the test sequence, and it should be helpful to simplify the design of the BIST controller.

3.2 Logic Minimization for Branch Points

The method discussed above is easy to implement. The control circuit compares the input test patterns with the branch points. Whenever a match is found, the following test patterns are directed to the next functional module. In general, if there are N functional units of an m-bit data-path, we need N equivalence comparators of size $2m$, which can be implemented with m 2-input XNOR gates and an m-input AND gate. This straightforward implementation is usually big in terms of circuit size and slow in terms of speed. With some elaboration, we are able to make the circuit both smaller and faster.

Whenever a branch point is met, we actually do not have to inspect every bit in this vector to decide whether this is the vector we need. Simpler logic can be obtained by checking the test patterns for a given functional unit. For example, consider an N-unit module, which requires t random test patterns. These patterns are denoted as $p_1, p_2, ..., p_t$, and these patterns are $2m$-bit. Suppose that test patterns for the i-th functional unit, denoted as U_i, start with p_l and end with p_m, where $1 \le l \le m \le t$. Simplification is possible with the following lemma.

Definition 1: In an n-variable function $f(x_1, x_2, ..., x_n)$, a *minterm* is a product term that contains each of the n variables as factors in either complemented or uncomplemented form.

A minterm can be represented with an n-bit vector $b_1b_2...b_n$, where $b_i \in \{0,1\}$.

Definition 2: A *cube* of an n-variable function $f(x_1, x_2, ..., x_n)$ is a product term that contains some of the variables in either complemented or uncomplemented form.

A cube can be represented with an n-bit vector $b_1b_2...b_n$, where $b_i \in \{0,1,x\}$ (x is a don't-care). Obviously, a cube contains one or more minterms. To be exact, a cube with c literals, where $c \le n$, is the logic sum of 2^{n-c} minterms.

Lemma 1: If the intersection of two cubes is empty (i.e., the two cubes do not contain the same minterms), then the two cubes must be different in at least one bit position.

Let $C_b(= b_1b_2...b_n)$ and $C_c(= c_1c_2...c_n)$ be two cubes. Lemma 1 states that if C_b and C_c do not contain common minterms, then there exists at least an index $\underline{i}$ such that $b_i = 1, c_i = 0$, or $b_i = 0, c_i = 1$. We shall refer to this difference henceforth as a *conflict*.

According to our method discussed in section 3.1, p_m is a branch point that will be compared with all the test patterns. Whenever p_m is met, the next test pattern p_{m+1} will be directed to unit U_{i+1}. With Lemma 1 described above, we are able to rewrite the condition for a branch to be checked as follows:

(a) Current unit under test is U_i, and

(b) the maximum cube C_{max} which satisfies $p_m \in C_{max}$ and $p_i \notin C_{max}$ ($l \le i \le m - 1$) is compared with the input patterns for a possible match.

Condition (a) given above is required since we want to find cube C_{max} by only considering test patterns p_l to p_m. This will lead to simpler logic; however, C_{max} may contain a pattern p_j where $j < l$ or $j > m$. Condition (a) is added to avoid this confusion.

Now the question is how to find the maximum cube C_{max}. C_{max} can be found with the help of Lemma 1. We shall demonstrate this procedure using the example shown in Fig. 5. Shown in the figure is a sequence of 4-bit vectors. Let the sixth vector be the target branch point. In order to find C_{max} for this branch point, we construct a conflict matrix, as shown on the right hand side of the figure. In general, for unit under test U_i whose test sequence consists of patterns from p_l and p_m, the conflict matrix consists of $m - l$ rows. Row i, where $l \le i \le m - l$, is obtained from bit-wise exclusive-OR operations between p_{l+i-1} and p_m. The conflict matrix is, thus, a 0-1 matrix, in which a "1" in row i of the matrix indicates a *conflict* between patterns p_{l+i} and p_m at the corresponding bit position.

Fig. 5. An example of logic minimization.

C_{max} is a maximum cube that conflicts with all the vectors from p_l to p_{m-1} in at least one bit position. This problem is equivalent to finding a minimum number of columns in the conflict matrix such that there is at least a "1" in each row by looking at theses selected columns. For example, consider the conflict matrix shown in Fig. 5. If we look at columns 3 and 4 only, we still can find at least a "1" in each row. As a result, columns 3 and 4 become a column covering for the matrix. This implies that we can identify the branch point (1101) by looking at the last two bits only; thus, C_{max} is xx01. It can be seen that none of the first 5 vectors belong of C_{max}. Thus, the problem of finding the maximum cube C_{max} is equivalent to solve the column covering problem for the conflict matrix. It should be noted that the seventh vector, which is 0101, in Fig. 5 also belongs to Cmax. However, when we reach this vector, the unit under test has been the next functional unit in the list; thus, condition (a) is not satisfied. Therefore, the control circuit will not generate a branch signal here.

We have applied the method to the circuits listed in Table II, and the results are summarized in Table 3. Although the inputs are 16-bit vectors, we can see that there are at most 6-bit care values in the corresponding maximum cubes. In Fig. 4, the control signal for the ALU is actually a one-hot code if we use 1000 for the shifter under test mode; thus, the number of lines to be checked in this part is 1. Note that for the multiplier, there are more than one candidate for its C_{max}, and that we only show some of them in the table. Also note that since the shifter is the last unit to be tested, we do not have to check for its branch point.

Table 3. Results of logic minimization.

Circuit	C_{max}	#lines	Control signal for UUT	#lines	#Total lines
Adder	xxxxxxxxx1xxxxx1	2	xxx1	1	3
Subtractor	x0xxxxxxxxxxxxx0	2	xx1x	1	3
Multiplier	11xx1xxxxxxx1x0x 11xxx1x0xxxx1x0x 11xxxxx0xxx11x0x . . .	6	x1xx	1	7
Shifter	-	-	-	-	-

The above method was used to modify the original Verilog model (first shown in Fig. 3 and then shown in modified form in Fig. 4) to improve its testability. The modified code is shown in Fig. 5. In the test mode, registers {C,B} are configured into a single PRPG while a branch point selection circuit is added to set the value in register *A* when a branch point is met. The check for a branch point consists of two parts: one is the maximum cube C_{max} for this branch point and the other is the content in register *A*. For example, the branch point for the adder is 1110000101101011 while its corresponding C_{max} is xxxxxxxxx1xxxxx1. The opcode for the adder is 0001 (one-hot code). Therefore, the branch point occurs when B[6] = 1, B[0] = 1, and A[0] = 1.

Unlike previous method outlined in section 2, the controller can be synthesized independent of the branch statement in this method. Thus, the number of test vectors required for each unit can be decided by the circuit implementation directly rather than based on estimation.

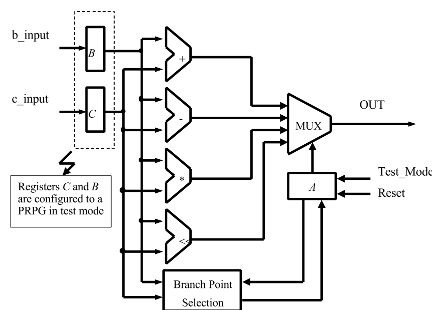
4. EXPERIMENTAL RESULTS

In order to show the effectiveness of the proposed methods, we conducted some experiments, and the results are given in this section. Based on the Verilog code shown in Fig. 3(a), we synthesized two different circuits whose characteristics are listed below. The modified Verilog code and corresponding circuit are given in Fig. 6.

1. Circuit I: Registers $\{C, B\}$ are configured into a single PRPG in the test mode while register *A* behaves as a 4-bit PRPG during test mode. This configuration is used in standard pseudorandom testing.

```
always @(B or C)//Combinational Logic
   begin
      k= B[0]^C[3]^C[5]^C[6]; //Feedback path of LFSR
      branch_point1=B[6]&B[0] &A[0]; //Cmax for branch point 1:xxxxxxxxx1xxxxx1
      branch_point2=~C[6]&~B[0]&A[1]; //Cmax for branch point 2:x0xxxxxxxxxxxxx0
      branch_point3=C[6]&C[3]&~C[1] &~B[7]&~B[2]&B[0]&A[2];
         //Cmax for branch point 3:x1xx1x0x0xxxx0x1
      branch_signal=branch_point1 | branch_point2 | branch_point3;
   end
always @(posedge CLK or posedge Reset) //Sequential Logic
         :
   if (Reset)
      (C,B,A)={0,1,1}; //seed for the PRPG is 1, the first UUT is the adder
   else
      if (Test Mode)
         {C[7], C[0:0], B[7], B[6:0]} = {k,C[7:1], C[0], B[7:11]};
            //In test mode, C and B are configured as a PRPG
         if (branch_signal)
               case (A)
                  4'b0001:A=4'b0010;
                  4'b0010:A=4'b0100;
                  4'b0100:A=4'b1000;
                  default: A=4'b0001;
               encase
      else
         {C,B,A}={C_input,b_input, a_input};
always @ (...A or B or C...) //Sequential Logic
   begin
         :
      case (A)
         4'b0001:OUT=B+C;
         4'b0010:OUT=B-C;
         4'b0100:OUT=B*C;
         default:OUT=B<<C;
      endcase
            :
   end
```

(a)



(b)

Fig. 6. Testability enhancement. (a) Verilog code, (b) block diagram.

2. Circuit II: Registers $\{C, B\}$ become a PRPG in the test mode, and register A is modified according to the method presented in section 3.

Two sets of circuits were synthesized. In the first set, the width of both registers B and C was 8-bit, and the results are shown in Table 4. The synthesis procedure has been described in section 3. In the second set, the width of both B and C was 16-bit, and the results are listed in Table 5. The circuits were synthesized using Design Analyzer from Synopsis. In the tables, both the combinational and noncombinational areas are provided.

Table 4. Area and Delay of synthesized circuits: 8-bit data registers *B* and *C*.

Circuit Type	#Test Patterns	Comb. Area	Noncomb. Area	Total Are	Area Overhead (%)	Delay	Delay Overhead (%)
I	Min: 6400 Max:64000	958	179	1138	0.00	29.12	0.00
II	4460	975	180	1154	1.4%	28.78	0.00

Table 5. Area and Delay of synthesized circuits: 16-bit data registers *B* and *C*.

Circuit Type	#Test Patterns	Comb. Area	Noncomb. Area	Total Area (%)	Area Overhead	Delay	Delay Overhead (%)
I	Max:3520000 Min:270770	3379	324	3703	0.00	61.17	0.00
II	220380	3395	323	3718	0.40%	61.17	0.00

In the 16-bit case (shown in Table 5), the shifter needed 220000 random test patterns to achieve100% fault coverage while the other three units could be fully tested with 380 random test patterns. When the shifter was tested with a probability of 1/16, we needed 3520000 patterns, in which 93.7% of the patterns were useless. If the shifter can be tested with 13/16 of all the patterns, then the total number of test patterns required is around 270770.

From the results shown in Tables 4 and 5, it can be seen that circuits can be modified using our method with negligible area overhead and no time penalty. The overhead is reduced as the circuit size increases, as can be seen by comparing the results shown in Tables 4 and 5.

5. CONCLUDING REMARKS

Conditional case statements may create testability problems under the BIST environment when the circuit is synthesized. In this paper, we have discussed how these statements affect the testability of synthesized circuits and presented a method to transform the original specification into a more testable design. This method effectively improves the testability of synthesized circuits with very low overhead, and the area and performance penalty diminishes as the circuit grows in size. This method is especially useful whenever there is wide variance in the number of random test patterns required for functional units.

REFERENCES

1. T. Thomas, P. Vishakantantaiah, and J. A. Abraham, “Impact of behavioral modifications for testability,” in *Proceedings of IEEE VLSI Test Symposium*, 1994, pp. 427-432.
2. V. Chickermane, J. Lee, and J. H. Patel, “A comparative study of design for testability methods using high-level and gate-level descriptions,” in *Proceedings of International Conference Computer-Aided Design*, 1992, pp. 620-624.
3. S. Bhattacharya, F. Brglez, and S. Dey, “Transformations and resynthesis for testability of RT-level control-data path specifications,” *IEEE Transactions on VLSI Systems*, Vol. 1, No. 3, 1993, pp. 304-318.
4. K. T. Cheng and V. D. Agrwal, “A partial scan method for sequential circuits with feedback,” *IEEE Transactions on Computer*, Vol. 39, 1990, pp. 544-548.
5. D. H. Lee and S. M. Reddy, “On determining scan flip-flops in partial-scan designs,” in *Proceedings of International Conference on Computer-Aided Design*, 1990, pp. 322-325.
6. V. Chickermane and J. H. Patel, “An optimization based approach to the partial scan design problem,” in *Proceedings of International Test Conference*, 1990, pp. 377-386.
7. S. Bhattacharya and S. Dey, “H-Scan: a high level alternative to full-scan testing with reduced area and test application overhead,” in *Proceedings of VLSI Test Symposium*, 1996, pp. 74-80.
8. S. Dey and M Potkonjak, “Non-scan design for testability of RT-level data paths,” in *Proceedngs of International Conference on Computer-Aided Design*, 1994, pp. 640-645.
9. F. F. Hsu, E. M. Rudnick, and J. H. Patel, “Enhancing high-level control-flow for improved testability,” *International Conference on Computer-Aided Design*, 1996, pp. 322-328.
10. C. A. Papachristou and J. Carletta, “Test synthesis in the behavioral domain,” *International Test Conference*, 1995, pp. 693-702.
11. S. Dey and M. Potkonjak, “Transforming behavioral specifications to facilitate synthesis of testable designs,” in *Proceedings of International Test Conference*, 1994, pp. 184-193.
12. A. Majumdar, R. Jain, and K. Saluja, “Incorporating testability considerations in high-level synthesis,” *Journal of Electronic Testing: Theory and Applications*, 1994, pp. 43-55.
13. K. A. Ockunzzi and C. A. Papachristou, “Testability enhancement for behavioral descriptions containing conditional statements,” in *Proceedings of International Test Conference*, 1997, pp. 236-245.
14. M. Abramovici, M. A. Breuer, and A. D. Friedman, *Digital System Testing and Testable Design*, W. H. Freeman and Company, 1990.
15. K. Kim, D. S. Ha, and J. G. Tront, “On using signature register as pseudorandom pattern generator in built-in self-testing,” *IEEE Transactions on Computer-Aided Design*, Vol. 7, No. 8, 1988. pp. 919-928.
16. C. Z. Yung and S.-J. Wang, “Behavioral synthesis for-testability for conditional statements with multiple branches,” in *Proceedings of Workshop on Computer Architecture, International Computer Symposium*, 1998, pp.15-21.

Sying-Jyan Wang (王行健) received a B.S. degree in electrical engineering from the National Taiwan University, Taiwan, R.O.C. in 1984, and a Ph.D. degree in electrical engineering from Princeton University in 1992. From 1989 to 1990 he was with AT&T Bell Laboratories, Holmdel, New Jersey. He was an associate professor of the National Chung-Hsing University, Taiwan, during 1992-1999. Currently he is a professor and the Chairman of Institute of Computer Science. His research itnerests include computer architecture, VLSI design, digital testing, and fault-tolerant computing.

Chia-Chun Lien (連家駿) received a B.S. degree in electronic engineering from the National Yunlin University of Science and Technology, Taiwan, R.O.C., in 1997, and a M.S. degree in Computer Science from National Chung-Hsing University, Taiwan, R.O.C. in 1999. He is currently with Via Technologies, Inc.