# Test Strategies for BIST at the Algorithmic and Register-Transfer Levels

Kelly A. Ockunzzi
IBM Microelectronics
Burlington, VT 05452
(802) 769-8757
ockunzzi@us.ibm.com

Chris Papachristou
EECS Dept., Case Western Reserve University
Cleveland, OH 44106
(216) 368-5277
cap@eecs.cwru.edu

## ABSTRACT

The proposed BIST-based DFT method targets testability problems caused by three constructs. The first construct is reconvergent fanout in a circuit behavior, which causes correlation. The second construct, control statements, also cause correlation, and relational operations degrade observability. The third construct is random-pattern-resistant RTL modules, which cannot be tested effectively with random patterns. Test strategies are presented that overcome the testability problems by modifying the circuit behavior. An analysis and insertion scheme that systematically identifies the problems and applies the strategies is described. Experimental results from seven examples show that this scheme improves fault coverage while minimizing the impact on area and critical delay.

**Categories and Subject Descriptors:** B.8.1 [**Performance and Reliability**]: Reliability, Testing, and Fault-Tolerance.

**General Terms:** Design, Theory

**Keywords:** design-for-test, built-in self-test, test synthesis

## 1. INTRODUCTION

Design-for-test (DFT) methods offer an economical and effective solution to the problem of testing circuits for manufacturing defects. As technology improvements enable the production of more complex circuits, circuit design moves to higher levels of abstraction. Considering testability early in the circuit design allows more opportunities to improve the testability of a design while minimizing the impact on performance and area.

Our approach applies pseudorandom built-in self-test (BIST) and we assume a scan-based approach will not be used. We examine a circuit in its functional RTL form, which includes high level and register-transfer level (RTL) information. For example, a control data flow graph that has been scheduled and bound to RTL modules describes circuit behavior and hardware usage.

We focus on three constructs that cause testability problems, combining ideas from [10, 11]. The first construct is reconvergent fanout in the circuit behavior, which causes correlation. Correlation has a detrimental effect on fault coverage because it restricts test patterns and masks fault effects. The second construct, control statements, can cause correlation as well. In addition, observability through a relational operation is poor because the response is always one bit wide. The third construct involves random-pattern resistance. Pseudorandom techniques assume all RTL modules in the circuit are random-pattern testable. However, some modules are tested more effectively with specific patterns or patterns with a particular characteristic.

We propose a set of test strategies to overcome the testability problems caused by these constructs. Our objective is to improve testability while minimizing the impact on area and performance. We present an analysis and insertion scheme to systematically locate problems and implement the strategies accordingly. Both data-flow and control-flow intensive behaviors are considered. The strategies modify the control signals for the datapath, so the controller may be affected but the datapath is not. Results from seven example behaviors show that our method improves fault coverage and generates minimal overhead.

The basis of our approach is delivery of high quality test patterns to each RTL module from the primary input and delivery of response patterns to the primary output. The patterns are propagated via other RTL modules and existing connections in the circuit. This idea comes from a deterministic approach called hierarchical test generation [9, 7]. Test patterns and responses are derived for each module, then propagated unmodified by using identity operations of the RTL modules. [1, 5, 12, 8] create a test environment for each module, which consists of a justification path and a propagation path. [4] extends the technique to pseudorandom BIST. Our approach uses the one-to-one property of operations, instead of the identity property, to propagate patterns. We do not derive a test environment for each RTL module.

## 2. BACKGROUND

Our work uses the *test behavior* approach [2]. A test behavior is derived from the original circuit behavior. The result is a circuit that implements both behaviors with minimal test control logic. A pseudorandom pattern generator supplies test patterns to the primary input. A signature analyzer collects response patterns at the primary output and compresses the patterns into a signature. The circuit is exercised according to its behaviors.

A *fault* is a manufacturing defect in a circuit. At the gate level, these faults are modeled as lines that are stuck-at-1 or stuck-at-0. Fault coverage is determined as the percentage of detected stuck-at faults from all possible stuck-at faults. At the RTL, faults manifest themselves as incorrect values. Fault detection involves sensitization and propagation. A fault is sensitized when an incorrect response is produced at the fault location. The incorrect value is a


DAC 2001, June 18-22, 2001, Las Vegas, Nevada, USA.

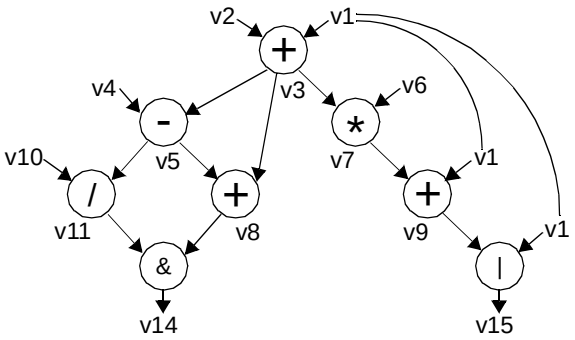



**Figure 1. FACET behavior with reconvergent fanout.**

*fault effect*. To detect a fault, its fault effect must be propagated to an observable point. *Controllability* is the ability to sensitize the fault, and *observability* is the ability to propagate the fault effect.

An *F-path* is a data channel with a one-to-one transformation from its primary input to its primary output [3]. The data channel may have other inputs and outputs, which can be data or control signals. Modules that have *F*-paths and are connected serially have an overall *F*-path [3]. An *F*-path preserves the observability of faults because it allows fault effects to propagate from the primary input to the primary output of a data channel. Test patterns applied to the other inputs must be independent of the patterns applied to the primary input, but these other inputs are not required to remain fixed during test.

# 3. THREE CONSTRUCTS

This section describes three constructs that cause testability problems.

## 3.1 Reconvergent Fanout

Reconvergent fanout is illustrated in Figure 1 with the data flow graph for the FACET behavior. A variable that drives more than one operation, such as *v3* in Figure 1, is a *fanout point*. When two or more fanout paths converge at the inputs for an operation, such as the addition operation for *v8* in Figure 1, that operation is a *convergence point*.

Reconvergent fanout introduces correlation at the convergence point. Correlation affects both controllability and observability, and can be measured with the correlation metric [10]. Correlation at the inputs to an operation restricts the test patterns that can be applied to that operation [2]. Because of the restricted input patterns, the response patterns for this operation are restricted as well. Correlation decreases observability by masking fault effects. If the inputs to an operation are not independent, then the operation is not guaranteed to have an *F*-path.

At the gate level, reconvergent fanout can make some stuck-at faults undetectable. To ease the complexity issue of modern circuits, we focus on breaking correlation at the RTL. The reconvergent fanout present in the behavior will remain once the circuit is synthesized to the gate level. By modifying the circuit at the RTL we can improve the fault coverage at the gate level.

## 3.2 Control Statements

Control statements such as *if-then-else* and *while* statements can introduce several testability problems. As discussed in [11], branch probabilities determine test session length for each branch of a control statement. Correlation occurs when variables are used

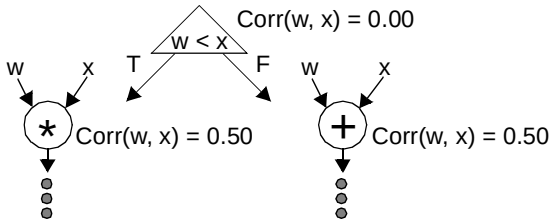
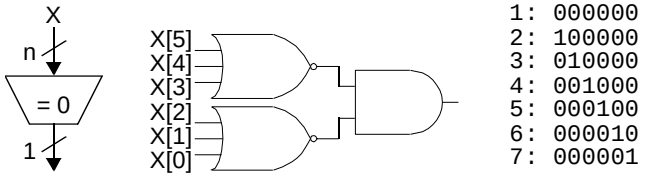



**Figure 2. Conditional statement with correlation.**




**a. RTL module.** **b. Gate-level description when *n* = 6.** **c. Test patterns when *n* = 6.**

**Figure 3. Specific test patterns required.**

in both the conditional expression and the branches of a control statement, because the conditional expression determines which branch is executed. [11] demonstrates how correlation affects test pattern quality for the branch operations, using the randomness and expected state coverage metrics to quantify the degradation.

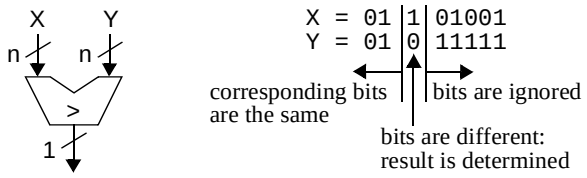
Figure 2 illustrates another correlation example. The variables *w* and *x* are independent for the relational operator in the conditional expression. However, *w* and *x* are correlated in each branch, with a correlation value of 0.50 for the multiplication and addition operations. The variables are correlated because of the restrictions imposed by the conditional expression: in the true branch *w* is less than *x*, and in the false branch *w* is greater than or equal to *x*.

Finally, control statements add relational operations to the behavior, which are implemented as comparators in the datapath. A comparator converts multi-bit inputs to a one-bit response, which is output to the controller as a status signal. Comparators do not have *F*-paths, so fault effects that appear at the inputs to a comparator are not easily observable through the comparator. Moreover, this status signal has no direct access to an observable point, so detecting faults within the comparator is difficult.

## 3.3 Random-Pattern Resistance

For some modules, random test patterns do not detect all faults effectively or efficiently. A module might require specific test patterns, and the probability of randomly generating these input patterns might be very low. Other modules may require test patterns with some characteristic that is difficult to generate randomly. These modules are *random-pattern resistant*. Fault coverage for these modules depends on the actual patterns generated and applied during the test.

For example, modules that implement relational operations become more random-pattern resistant as the bit width ($n$) of their inputs increases. The comparator shown in Figure 3a determines whether *X* is equal to zero. Figure 3b shows the gate-level description, generated by Synopsys, for this module when the bit width is six. The structure for other values of $n$ is similar. This module requires $n$+1 specific patterns to be fully tested. *X* must be set to zero and to the $n$ values where only one bit is set to '1', as shown in Figure 3c for a bit width of six. No other test patterns will detect




**a. RTL module.** **b. Compare process.**

**Figure 4. Test patterns with particular characteristic required.**

the faults in this module. If the probability distribution of *X* is uniform, then each of the test patterns will be randomly generated with probability $2^{-n}$. As *n* increases, the likelihood of generating any specific pattern decreases exponentially.

A second comparator, shown in Figure 4a, determines whether *X* is greater than *Y*. This module compares *X* and *Y* on a bit-by-bit basis, starting with the most significant bits and working towards the least significant bits. If the corresponding bits in *X* and *Y* are the same, then the process continues to the next pair of bits. As soon as the bits are different, the module can output the result and the remaining bits are ignored. Figure 4b illustrates this process for a bit width of eight.

For this module, the logic for the higher bits is exercised more frequently than the logic for the lower bits. The most significant bits are compared for all possible test patterns, but the least significant bits are compared only when all other corresponding bits are the same. Assuming uniform probability distributions, the probability that *p* corresponding bits are the same is $2^{-p}$. As bit width increases, the opportunity to exercise the logic for the lower bits decreases exponentially. Instead of specific test patterns, this module requires some test patterns with the characteristic that corresponding higher bits in *X* and *Y* are the same.

From experimentation, we have determined that comparators with eight-bit-wide input variables or smaller can be easily tested with random patterns.

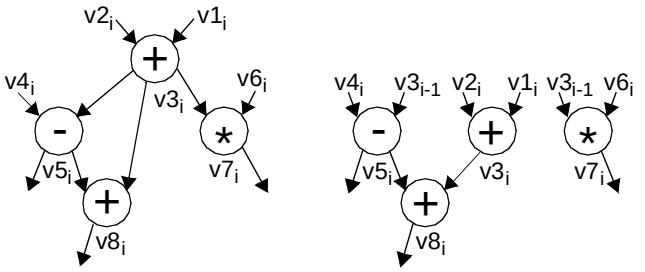
# 4. APPROACH

This section presents test strategies to overcome the testability problems caused by the three constructs.

## 4.1 Techniques for Reconvergent Fanout

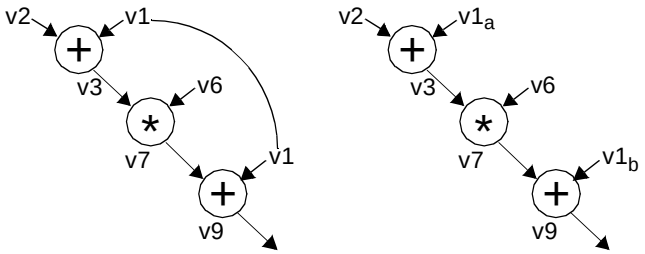
One way to break correlation caused by reconvergent fanout is to insert a *delay load* to create an *F*-path *across time*. As the behavior is executed, registers within the datapath store values from the previous execution until they are loaded with the new values for the current execution. If we delay the load signal for a register, then the old value is used in the current execution instead of the new value. The new value is not lost, however, because it is used in the next execution. Load signal delays were introduced by [7].

Figure 5 demonstrates the delay-load insertion using the FACET example from Figure 1. Part of the original behavior is shown in Figure 5a. Figure 5b illustrates the test mode behavior, where the load signal for the register that stores *v3* is delayed one clock cycle. During the *i*th execution of the behavior, *v5* depends on the current *v3* ($v3_i$) in normal mode. This value is used to compute *v8*. In test mode, *v5* and *v7* are computed from the *v3* generated in the previous execution ($v3_{i-1}$). However, *v8* is computed from the *v3* and *v5* generated in the current execution ($v3_i$ and $v5_i$). Correlation is broken because $v3_{i-1}$ and $v3_i$ are not the same value.




**a. Normal mode.** **b. Test mode: *v3* delayed.**

**Figure 5. Delay-load insertion technique.**




**a. Normal mode.** **b. Test mode: load *v1* again.**

**Figure 6. Multiple-load insertion technique.**

Inserting a *multiple load* breaks reconvergent fanout when a primary input of the circuit is involved. The register storing the primary input can be loaded more than once during the current execution, which allows a new value to be introduced into the computations. This technique assumes the test pattern generator at the primary input generates new patterns every clock cycle.

Figure 6 demonstrates the multiple-load insertion. Figure 6a shows part of the data flow graph from Figure 1 and the modified behavior is shown in Figure 6b. In normal mode both inputs for the *v9* computation depend on *v1*. In test mode, the register that stores *v1* is loaded again in control step 2. The first value for *v1* ($v1_a$) is used to compute *v3*, and the second value for *v1* ($v1_b$) is used to compute *v9*. Since $v1_a$ and $v1_b$ are two different values, the correlation between *v7* and *v1* is broken.

These techniques are implemented by modifying the controller. An input signal to the controller indicates whether the circuit is in test mode or normal mode. In normal mode, the controller produces the control signals for the datapath according to the circuit behavior. In test mode, the normal mode behavior is followed except when delay loads and multiple loads are required to break the correlation. The state machine is altered to produce the new load and select signals for the affected registers and multiplexers. Alternatively, delay elements could be inserted on the control signals to implement the delay loads.

## 4.2 Techniques for Control Statements

The delay-load and multiple-load techniques can be used to break the correlation caused by control statements. For the example shown in Figure 2, suppose the load signal for *w* is delayed. The *w* computed in the previous execution is used to compute the conditional expression. Then the *w* computed in the current execution is loaded. This value is used for either the addition or multipli-

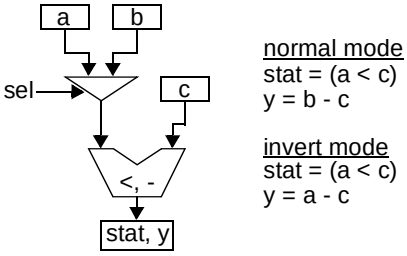



**Figure 7. Invert-mode insertion technique.**

cation operation, depending on which branch is taken. As a result, the correlation value for these operations is now 0.

Another technique, proposed in [11, 6], breaks correlation and changes branch probabilities. The conditional expression is augmented by inserting an AND, an OR, or an XOR operation. One input to this operation is the output of the conditional expression, and the other input comes from a pseudorandom pattern generator. In test mode, this augmentation allows better control of test session length and test pattern quality for the branches. In normal mode, the extra input is set so that the conditional expression alone determines which branch is taken. This technique modifies the status signal sent to the controller.

To improve testability for relational operations, we recommend binding a relational operation to a subtractor or adder. Faults in the ALU can then be detected via the subtraction or addition operations present in the circuit behavior. This binding also creates an opportunity to better observe the inputs to a comparator. A second test mode, called *invert mode*, allows the comparator inputs to be observed through the subtractor instead. Invert mode inverts multiplexer select signals.

The sample datapath shown in Figure 7 illustrates the technique. Faults in the multiplexer and fault effects that appear at its inputs are difficult to observe because of the "less than" operation. The ALU has an *F*-path only when (*b* - *c*) is computed. To improve observability, invert mode inverts the select signal for the subtraction operation. The multiplexer passes *a* instead of *b*, so the subtraction operation is (*a* - *c*). For the "less than" operation, the select signal is not inverted and the multiplexer passes *a*. Invert mode must be a separate test mode because the (*b* - *c*) subtraction is eliminated. This subtraction is tested in the "regular" test mode, during which the multiplexer select signal is not inverted.

Invert mode is implemented by modifying the controller. The multiplexer select signals are inverted as needed by altering the state machine or by directly inserting inverter gates on the control signals. An input signal to the controller indicates whether the circuit is in invert mode.

## 4.3 Techniques for Random-Pattern Resistance

The general approach for testing random-pattern-resistant modules is to manipulate the circuit behavior so the desired test patterns are applied to these modules. These modifications become one or more new test modes, and each mode fulfills an objective. The modes are distinguished by codes input to the controller. The controller, modified to implement each test mode as well as normal mode, then executes the test behavior indicated by the code.

To demonstrate the techniques for testing random-pattern-resistant modules, the algorithm shown in Figure 8 is used. This algorithm uses a series of subtraction operations to determine the

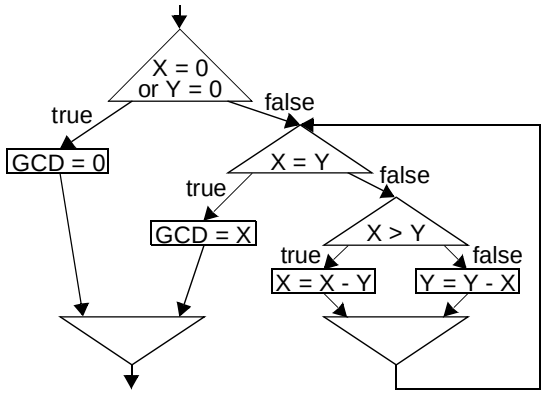



**Figure 8. GCD algorithm with random-pattern resistance.**

greatest common divisor (GCD). The bit width of the variables is twelve, so the relational operations are random-pattern resistant.

Sometimes the behavior itself is effective in testing random-pattern-resistant modules. For the GCD example, the (*X* > *Y*) comparison requires some test patterns where the corresponding higher bits of *X* and *Y* are the same. In this case, the algorithm provides patterns with this requirement. As the behavior is executed, the values for *X* and *Y* decrease, so that more of the higher bits for both variables are set to '0'. These smaller values allow more of the gate-level comparator logic to be exercised and tested.

Other times, however, the circuit behavior needs to be modified to allow desired test patterns to reach a module. In the GCD example, both the (*X* = *Y*) and the (*X* > *Y*) comparisons require test patterns where *X* equals *Y*. However, the (*X* = *Y*) comparison receives many test patterns where *X* and *Y* are not equal, and only a few test patterns where the variables are equal. Moreover, the nature of the GCD algorithm does not allow test patterns where *X* equals *Y* to reach the (*X* > *Y*) comparison.

To increase the number of test patterns where *X* equals *Y*, the behavior is modified. In normal mode, the greater of *X* and *Y* is loaded during the subtraction step. In this test mode, however, *both* variables are loaded with the subtraction result. The rest of this test mode is the same as normal mode. By forcing *X* and *Y* to be equal, the (*X* = *Y*) comparison is tested more effectively. A second test mode forces *X* and *Y* to be equal, then allows these patterns to reach the (*X* > *Y*) comparison.

Finally, some modules require specific test patterns, such as the (*X* = 0) comparison in the GCD example. To generate the test patterns, the *X* register is replaced with a shift register that can be set to zero or to "100000000000". In one test mode, the controller resets the *X* register to zero. In another test mode the controller resets the *X* register to "100000000000", then shifts this value. Each reset or shift replaces the step where the register is loaded with a pattern from the test pattern generator. Two more test modes for the (*Y* = 0) comparison are similar.

# 5. METHOD

Our DFT method has two phases. In the first phase, testability problems due to reconvergent fanout and control statements are identified and resolved. In the second phase, random-pattern-resistant modules are identified and test schemes are devised to make these modules testable. The result is a circuit that implements the original behavior and one or more test behaviors.

To analyze reconvergent fanout, the correlation metric is computed for the input variables of operations at the convergence points. For each control statement, the randomness, expected state coverage, and correlation values are computed for the conditional variables. In addition, branch probabilities for each conditional statement and the average number of iterations for each loop statement are computed. If the metrics indicate a testability problem, then one of the techniques described in Sections 4.1 and 4.2 is inserted into the behavior. The metrics are recomputed and the behavior is examined again. The process is repeated until all problems are resolved. At this point, we have a circuit that implements the original behavior, a test behavior, and possibly an invert-mode behavior. All random-pattern testable faults should be covered.

Testability problems related to random-pattern-resistant modules are considered after all other problems have been identified and resolved. Often random-pattern-resistant modules can be identified by inspecting the circuit behavior or the datapath. Modules can be classified by their properties. For example, any comparator with twelve-bit-wide input variables or larger is random-pattern resistant. In other instances, examination of the fault coverage results reveals the modules that are difficult to test. Modifications that target other testability problems will not improve the fault coverage for random-pattern-resistant modules. In fact, modifications that increase the randomness of variables can actually decrease the testability of random-pattern-resistant modules. The list of undetected faults and the gate-level description of the module can be used to determine the specific patterns or pattern characteristics needed to test this module.

The behavior, the datapath, and the fault coverage results are inspected to identify any untestable random-pattern-resistant modules. For each random-pattern-resistant module in the datapath, a test scheme is devised based on the techniques described in Section 4.3. The behavior is modified to implement each test scheme. The procedure is complete when all random-pattern-resistant modules are testable. The result is a modified behavior with a normal mode and one or more test modes.

# 6. RESULTS AND DISCUSSION

We applied our DFT method to six example behaviors and one ITC'99 benchmark. The FACET and cubic polynomial behaviors are data dominated and have multiple instances of reconvergent fanout. The bit widths vary for each example so no bits are truncated. For the FACET example, the bit width ranges from five bits to twelve bits. The inputs for the cubic polynomial example are four bits wide, and the output is sixteen bits wide.

The next three behaviors are control dominated. The postage calculator example has three conditional statements. The package weight and the weight ranges are eight bits wide, and the price increments are four bits wide. The floating-point addition example has six conditional statements and one instance of reconvergent fanout. The exponents are four bits wide and the mantissas are eight bits wide. The ITC'99 benchmark example, b11, has seven conditional statements and two loop statements. The data input and output are six bits wide. The last two behaviors, GCD and integer division, are control dominated as well. Both examples have datapath modules that are random-pattern resistant. The bit width of the variables in both examples is twelve.

The test modifications proposed in this paper were implemented by altering the states in the controller state machine or by augmenting the status lines input to the controller. The control-dominated examples were synthesized so the comparators were bound to subtractors when possible, and invert modes were used when needed. For modules that did not have an *F*-path in the FACET, postage calculator, and floating-point addition examples, the datapath was modified so these modules would have an *F*-path in test mode. To supply the specific test patterns required by the "equals zero" comparators in the GCD and integer division examples, the input registers were replaced by shift registers, as described in Section 4.3.

For each example, two circuits were produced. Each circuit consisted of a datapath and a controller. The original circuit implemented the original behavior with no test insertion. The modified circuit implemented the original behavior and the test behavior. Synopsys was used to compile the circuits, calculate area and critical delay, and run the fault simulations. For the fault simulations pseudorandom patterns were applied to the primary inputs of each circuit. The entire circuit was used in the fault simulation, and each of its behaviors was exercised. For each example, the same set of test patterns was applied to the original and modified circuit.

Table 1 summarizes the results for the seven examples. The number of gates, the critical delay, the total number of faults, the number of undetected faults, and the fault coverage are listed for each circuit. Area and delay overheads for the modified circuits are included in the table. The undetected fault counts include untestable faults; the FACET and cubic polynomial behaviors each have a couple untestable faults. The original and modified circuits for each example were simulated for the same number of clock cycles; the "Clock Cycles" column reports the clock cycle at which the last fault was detected for each circuit.

For each example, the test modifications improved the fault coverage. Although the fault simulations ran longer for some modified circuits, more faults were detected earlier in the test sessions for all modified circuits. Since we made no modifications to improve the testability of the controller, we did not expect to achieve 100% fault coverage. Nearly all remaining undetected faults were related to the controller. However, the test modifications did decrease the number of undetected controller faults.

The impact of our method on area and critical delay was less than 3.5% for most of the examples. For the modified cubic polynomial and GCD circuits, the critical delay decreased because Synopsys was better able to optimize the circuit. For the modified FACET circuit, observation of the remainder (from the division operation, to give this operation an *F*-path) caused most of the overhead. For the modified floating-point adder circuit, the modified datapath modules caused most of the area overhead.

Most of the very high area overhead in the modified GCD and integer division circuits was caused by the shift registers used to test the "equals zero" comparators. Although flip-flops can be designed and optimized with "set", "clear", and "shift" capabilities, only a basic flip-flop was available in the Synopsys library. A multiplexer was added to each register to implement the desired functions, which made the registers approximately 4.5 times larger.

# 7. CONCLUSION

Our contribution is a DFT method that addresses testability problems caused by reconvergent fanout, control statements, and random-pattern resistance. Behavioral modifications break correlation in the behavior, improve the observability of comparator

**Table 1. Experimental results.**

| Circuit | | Gates | | Critical Delay | | Clock Cycles | Faults | | Fault Coverage |
|---|---|---|---|---|---|---|---|---|---|
| | | count | overhead | ns | overhead | | total | undet. | |
| FACET | original | 1816 | | 114.69 | | 6825 | 1919 | 77 | 95.99% |
| | modified | 1889 | 4.02% | 129.32 | 12.76% | 11991 | 2079 | 30 | 98.56% |
| cubic | original | 1740 | | 75.20 | | 1933 | 1634 | 110 | 93.27% |
| | modified | 1759 | 1.09% | 75.08 | -0.16% | 2374 | 1670 | 48 | 97.13% |
| postage | original | 1187 | | 42.32 | | 14625 | 1162 | 86 | 92.60% |
| | modified | 1228 | 3.45% | 42.40 | 0.19% | 10917 | 1211 | 34 | 97.19% |
| adder | original | 1529 | | 45.29 | | 17969 | 1938 | 73 | 96.23% |
| | modified | 1663 | 8.76% | 45.64 | 0.77% | 15609 | 2189 | 33 | 98.49% |
| b11 | original | 835 | | 43.62 | | 12268 | 1154 | 95 | 91.77% |
| | modified | 853 | 2.16% | 44.24 | 1.42% | 18398 | 1189 | 49 | 95.88% |
| GCD | original | 979 | | 69.76 | | 18586 | 1107 | 102 | 90.79% |
| | modified | 1188 | 21.35% | 60.49 | -13.29% | 10810 | 1411 | 26 | 98.16% |
| divider | original | 1046 | | 34.76 | | 18416 | 1059 | 85 | 91.97% |
| | modified | 1194 | 14.15% | 35.39 | 1.81% | 10206 | 1296 | 35 | 97.30% |

inputs, and efficiently test random-pattern-resistant modules. These modifications affect the controller. The datapath might be modified if a module implements a function that is not one-to-one or if a module requires specific test patterns. To help test the comparators in a datapath, we suggest binding the relational operations to subtractors or adders.

The controller modifications are independent of the bit width in the datapath, which reduces area overhead and performance degradation. The modified controller incorporates the original behavior and the test behavior, which reduces test complexity and allows the circuit to be tested at the designed speed. Our test strategies improve fault coverage, as shown by the results for seven example circuits.

Although the work presented in this paper is described in terms of control data flow graphs, our approach can be applied to other types of behaviors. In fact, the ITC'99 example behavior is written like a finite state machine in VHDL, without separate descriptions for the datapath and controller. Our method does not impose RTL design restrictions. Future work includes applying this approach to the system level, where the modules are interacting behaviors and the system behavior is known.

The analysis and insertion scheme can be applied either during or after high-level synthesis, because most of the modifications are made to the control and status signals. As a post-synthesis technique, the binding information for the datapath is used to guide the insertions. If correlation is considered during synthesis, the behavioral modifications are used to guide the binding decisions.

## 8. ACKNOWLEDGEMENTS

The first author did this work as a graduate student at CWRU. The work was supported by the National Science Foundation under Grant #CCR-9901160 and by an NSF Graduate Fellowship.

## 9. REFERENCES

[1] Bhatia, S., and N.K. Jha, "Integration of Hierarchical Test Generation with Behavioral Synthesis of Controller and Data Path Circuits", *IEEE Trans. on VLSI Systems*, Vol. 6, No. 4, pp. 608-619, Dec. 1998.

[2] Carletta, J.E., and C.A. Papachristou, "Behavioral Testability Insertion for Datapath/Controller Circuits", *J. of Electronic Testing: Theory and Appl.*, Vol. 11, No. 1, pp. 9-28, Aug. 1997.

[3] Freeman, S., "Test Generation for Data-Path Logic: The *F*-Path Method", *IEEE J. of Solid-State Circuits*, Vol. 23, No. 2, pp. 421-427, April 1988.

[4] Ghosh, I., N.K. Jha, and S. Bhawmik, "A BIST Scheme for RTL Controller-Data Paths Based on Symbolic Testability Analysis", *Design Auto. Conf.*, pp. 554-559, June 1998.

[5] Ghosh, I., A. Raghunathan, and N.K. Jha, "Design for Hierarchical Testability of RTL Circuits Obtained by Behavioral Synthesis", *IEEE Trans. on CAD*, Vol. 16, No. 9, pp. 1001-1014, Sept. 1997.

[6] Hsu, F.F., E.M. Rudnick, and J.H. Patel, "Enhancing High-Level Control-Flow for Improved Testability", *Intern. Conf. on CAD*, pp. 322-328, Nov. 1996.

[7] Karam, M., R. Leveugle, and G. Saucier, "Hierarchical Test Generation Based on Delayed Propagation", *Intern. Test Conf.*, pp. 739-747, Oct. 1991.

[8] Makris, Y., and A. Orailoglu, "Channel-Based Behavioral Test Synthesis for Improved Module Reachability", *Design, Auto., and Test in Europe Conf.*, pp. 283-288, March 1999.

[9] Murray, B.T., and J.P. Hayes, "Test Propagation Through Modules and Circuits", *Intern. Test Conf.*, pp. 748-757, Oct. 1991.

[10] Ockunzzi, K.A., and C.A. Papachristou, "Breaking Correlation to Improve Testability", *VLSI Test Symp.*, to appear, May 2001.

[11] Ockunzzi, K.A., and C.A. Papachristou, "Testability Enhancement for Control-Flow Intensive Behaviors", *J. of Electronic Testing: Theory and Appl.*, Vol. 13, No. 3, pp. 239-257, Dec. 1998.

[12] Ravi, S., G. Lakshminarayana, and N.K. Jha, "TAO: Regular Expression Based High-Level Testability Analysis and Optimization", *Intern. Test Conf.*, pp. 331-340, Oct. 1998.