A BIST Scheme for RTL Controller-Data Paths Based on Symbolic Testability Analysis *

Indradeep Ghosh† Niraj K. Jha† Sudipta Bhawmik‡

† Department of Electrical Engineering
Princeton University, Princeton, NJ 08544

‡ Engineering Research Center
Lucent Technologies, Princeton, NJ 08542

Abstract

This paper introduces a novel scheme for testing register-transfer level controller/data paths using built-in self-test (BIST). The scheme uses the controller netlist and the data path of a circuit to extract a test control/data flow (TCDF) which consists of operations mapped to modules in the circuit and variables mapped to registers. This TCDF is used to derive a set of symbolic justification and propagation paths (known as test environment) to test some of the operations and variables present in it. If it becomes difficult to generate such test environments with the derived TCDFs, a few test multiplexers are added at suitable points in the circuit to increase its controllability and observability. This test environment can then be used to exercise a module or register in the circuit with pseudorandom pattern generators which are placed only at the primary inputs of the circuit. The test responses can be analyzed with signature analyzers which are only placed at the primary outputs of the circuit. Every module in the module library is made random-pattern testable, whenever possible, using gate-level testability insertion techniques. Finally, a BIST controller is synthesized to provide the necessary control signals to form the different test environments during testing, and a BIST architecture is superimposed on the circuit. Experimental results on a number of industrial and university benchmarks show that high fault coverage (>99%) can be obtained with our scheme in a small number of test cycles at an average area (delay) overhead of only 6.4% (2.5%).

1 Introduction

Due to the increasing complexity of integrated circuits, BIST is emerging as a popular testing technique for large and complex designs. BIST has several advantages over other test approaches [1], [2], *e.g.*, it gets rid of the need to do test generation and enables at-speed testability. Previously, BIST techniques at the logic level have targeted better test pattern generators [3], and reduction in test overheads [4]. In this paper, the BIST problem is targeted at the register-transfer level (RTL). Owing to the drastic reduction in the number of circuit elements one needs to tackle at the RTL, the problem becomes more tractable. At the RTL, there have been some efforts to reduce test overheads by using a testability analysis scheme based on randomness and transparency of modules [5]. However, the complexity of this method becomes prohibitive as the bit-width increases. A BIST scheme for testing data paths of data-flow intensive RTL circuits was presented in [6]. A scheme for testing the controller was presented in [7]. However, in this work the class of circuits was restricted with the requirement that at most one multiplexer/bus exists along any register-to-register transfer path. Also, the scheme only dealt with register loads and multiplexer select lines coming out of the controller, and did not consider many other types of control lines like ALU select lines *etc*. Another BIST scheme targeting data paths of data-flow intensive RTL circuits was presented in [8]. It reduced the complexity of testability analysis by concentrating on subspace state coverage. However, since it relies on arithmetic units for test generation and compaction, its applicability to control-flow intensive circuits is limited.

In this paper, we present a BIST scheme for testing a controller-data path RTL circuit without imposing any major design restrictions. The only assumptions we make are that the circuit have a separate data path and controller, and a reset state be present in the controller. We can handle both data-flow intensive and control-flow intensive circuits. The scheme works by replacing RTL modules in the circuit which are random-pattern resistant with random-pattern testable versions. If the circuit is originally designed with a module library composed of random-pattern testable modules, then this step is unnecessary. Next, the control and data flow of the circuit are extracted form the controller/data path using the technique presented in [9]. This is termed as test control/data flow (TCDF). It consists of operations mapped to modules in the circuit and variables mapped to registers. The TCDF is used along with the functional information of modules to obtain a set of justification and propagation paths called *test environments* for some operations/variables in them [10]. The test environment of an operation (variable) guarantees the existence of a test path from the primary input ports of the circuit to the inputs of the module (register) to which the operation (variable) is mapped and a path from the output of the module (register) to a primary output port of the circuit. If it becomes difficult to generate such test environments with the existing TCDFs, a few test multiplexers are added at suitable points in the circuit to increase its controllability and observability [11]. Note that the methods presented in [9]-[11] are targeted towards deterministic test generation, not BIST. Since the search for a test environment is done symbolically, it is very fast and needs to be done only once for each module or register in the circuit. The test environment can then be used to exercise a module or register in the circuit with pseudorandom pattern generators (PRPGs) which are placed only at the primary inputs of the circuit. The test responses can be analyzed with multiple-input signature registers (MISRs), which are only placed at the primary outputs of the circuit. The test environments are also analyzed to see if primary input registers can themselves be converted to PRPGs and primary output registers converted to MISRs to reduce overheads further.

In Section 2, the procedure is explained with examples. The BIST architecture is discussed in Section 3. The method is formalized in Section 4. The experimental results and conclusions are in Sections 5 and 6, respectively.

2 Motivational Examples

In this section, we illustrate the complete RTL BIST scheme with a few examples.

2.1 Making RTL modules random-pattern testable

If elements present in the RTL design library are not individually random-pattern testable, then they can degrade the fault coverage of an RTL circuit. In practice, most RTL elements are random-pattern testable. For example, adders, subtracters, shifters, multiplexers, registers, and buses are all random-pattern testable. Even 32-bit versions of these elements are fully tested with less than 50 pseudorandom vectors. A 32-bit multiplier is completely tested

*Acknowledgments: This work was partially supported by NSF under Grant No. MIP-9319269.

Permissions to make digital/hard copy of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DAC 98, San Francisco, California
(c) 1998 ACM 1-58113-049-x/98/06 ..$5.00

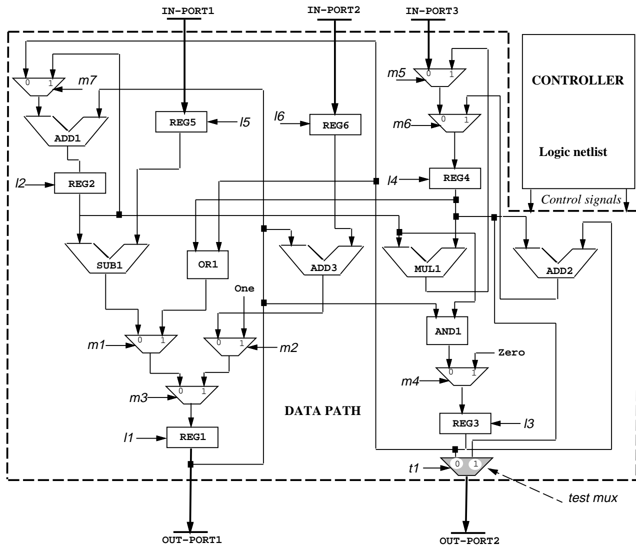

Figure 1: The RTL circuit for *Tseng*

INPUT	STATE		OUTPUTS													
reset	PS	NS	l1	l2	l3	l4	l5	l6	m1	m2	m3	m4	m5	m6	m7
1	ANY	S0	1	0	1	1	1	1	0	1	1	1	0	0	0
0	S0	S1	0	1	0	0	0	0	0	0	0	0	0	0	0
0	S1	S2	1	0	0	1	0	0	0	0	0	0	1	0	0
0	S2	S3	1	1	0	1	0	0	0	0	1	0	0	1	1
0	S3	S4	1	0	1	0	0	0	1	0	0	0	0	0	0
0	S4	S0	0	0	0	1	1	1	0	0	0	0	0	0	0

Figure 2: Controller state table of *Tseng*

with approximately 500 pseudorandom vectors. The real bottleneck lies with the comparators which are highly random-pattern resistant. Hence, in order to boost the random pattern testability of these modules, we resort to some logic-level techniques. This can be done with the help of some test points, *i.e.*, extra control/observe points [2]. For the scheme we have devised, a less-than (equal-to) comparator can be made fully random-pattern testable with only one (*n/8*) extra control and one extra observe point [12]. These extra control and observe points are connected to PRPGs and MISRs, respectively. Thus, they do not require any extra test pins when the comparator is part of an RTL circuit.

2.2 Extracting control/data flow information

Figure 1 shows the RTL circuit of benchmark *Tseng* taken from [13]. This particular RTL implementation was synthesized by the *Genesis* high-level synthesis system [10]. The shaded test multiplexer is not a part of the original circuit and will be discussed later.

The first step in the BIST process is to extract the TCDF using the RTL data path and controller netlist. For this, the procedure presented in [9] is used. First, the controller state table is extracted from the controller netlist. This can be done by a state machine extraction program, *e.g.*, SIS [14]. It starts from the controller reset state and extracts the input/output and state transition information of all reachable states. The extracted controller state table of *Tseng* is shown in Figure 2.

The basic idea behind extracting the TCDF from the RTL circuit is to extract operations executed in each cycle and keep track of variables present in each register or latch. We identify all registers that load in the first cycle by analyzing the load signals of all the registers in the data path from the outputs in the state table. If a load signal is 1 in any cycle, then the register loads in that cycle. In our example, in cycle 1, all registers except *REG2* load. If there are any latches, they load by default in each cycle. We next analyze the multiplexer tree that feeds each of the registers or latches and check if any input port is connected to the register/latch input in the first cycle. The multiplexer tree configuration in any cycle is obtained by looking at the values of the select signals of those multiplexers in the state table. We find that *IN-PORT1*, *IN-PORT2* and *IN-PORT3* are connected to *REG5*, *REG6*, and *REG4*, respectively, in the first cycle. Hence, three variables are born in these three registers. We call them *i1*, *i2* and *i3*, respectively. A variable is live

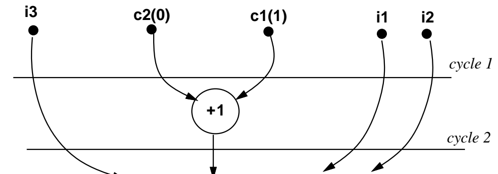

Figure 3: Partial TCDF after analyzing cycle 2 of *Tseng*

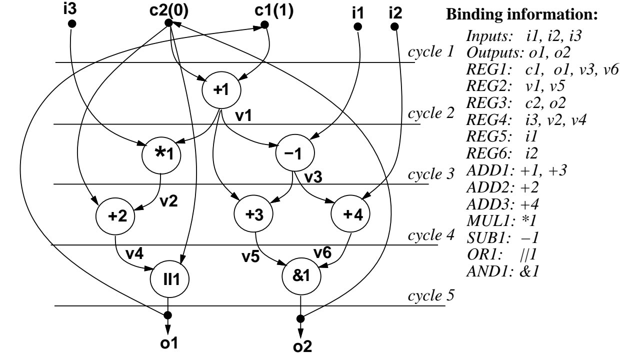

Figure 4: The TCDF for *Tseng*

until its register loads again. Thus, the variable-to-register binding information is developed in each cycle and the variables bound to a particular register noted. On the other hand, we find that in the first cycle, *REG1* and *REG3* are connected to hardwired constants. We name these constants *c1* and *c2* , respectively.

In cycle 2, we identify the operations that take place. For each module in the circuit, we find out the operand selected at each of its input ports by analyzing the multiplexer trees at its inputs. For example, the left input of the adder, *ADD1*, is connected to *REG3* and the right input to *REG1*. We check if both these registers have some live items in them. In this case, they do and they are *c2* and *c1*, respectively. In addition, we check if any of the registers (there may be many due to fanout) at the output of the module load at the end of the second cycle, and if the multiplexer configuration at the input of these registers is such that the output of the module is selected. In case of *ADD1*, *REG2* indeed loads at the end of the second cycle. Hence, we create a new variable *v1*, in *REG2*, and an operation in the TCDF, $c1 + c2 = v1$. We also label this operation as *+1*. Similarly, we analyze all other modules in order to obtain the set of operations that execute in the second cycle, as shown in Figure 3 (here the initial values of the constants are also shown in brackets). By repeating this procedure for five cycles, we generate the TCDF shown in Figure 4. In this figure, constants *c1* and *c2* are the initial values of the loop outputs. This method is general enough to handle operation chaining, multi-cycling, and structural pipelining.

2.3 Test environment generation

We use the obtained TCDF to generate *test environments* for all the RTL elements in the circuit. Before we can explain the test environment generation procedure, we need to define a few terms. The general controllability C_g of a TCDF variable is the ability to control it to any arbitrary value from the system primary inputs. The observability O_v of a TCDF variable is the ability to observe any arbitrary value at it at a system primary output. Similarly, we can define C_q (controllability to some arbitrary constant q). C_1 (controllability to 1), C_0 (controllability to 0), and C_{a1} (controllability to the all-1s vector) are special cases of C_q. The verifiability V of a TCDF variable is the ability to verify its value by either controlling or observing it. Controllability, observability and verifiability are all Boolean parameters and only take the values of 1 or 0, depending on whether the variable has the corresponding ability or not [10]. Next, we add another field to these parameters, which designates the cycle when the particular property of a variable is desired. Hence, $C_g(2)$ of a TCDF variable means that we need to control that variable to an arbitrary value in cycle 2.

Let us take the example of operation *+4*. The test environment

of this operation will be used to test adder *ADD3* in the RTL circuit and is obtained as follows. For operation *+4*, we need $C_g(\alpha)$ of *v3*, $C_g(\alpha)$ of *i2* and $O_v(\alpha+1)$ of *v6*. $O_v(\alpha+1)$ of *v6* is trivially achieved since *v6* is mapped to *REG1* which is connected to primary output *OUT-PORT1*. $C_g(\alpha)$ of *i2* is transformed to $C_g(\alpha-2)$ of *i2* which is trivially satisfied as *i2* is a primary input variable. $C_g(\alpha)$ of *v3* is transformed through operation *–1* to $C_g(\alpha-1)$ of *v1* and $C_q(\alpha-1)$ of *i1*. $C_g(\alpha-1)$ of *v1* is transformed through operation *+1* to either $C_g(\alpha-2)$ of *c1* and $C_q(\alpha-2)$ of *c2*, or $C_g(\alpha-2)$ of *c2* and $C_q(\alpha-2)$ of *c1*. Since either objective is unachievable as both *c1* and *c2* are constants, we backtrack and transform $C_g(\alpha)$ of *v3* to $C_q(\alpha-1)$ of *v1* and $C_g(\alpha-1)$ of *i1* through operation *–1*. $C_g(\alpha-1)$ of *i1* is transformed to $C_g(\alpha-2)$ of *i1* which is trivially satisfied as *i1* is a primary input variable. $C_q(\alpha-1)$ of *v1* is transformed to $C_q(\alpha-2)$ of *c1* and $C_q(\alpha-2)$ of *c2*. Both of these are now satisfied and, hence, all objectives for testing *+4* are fulfilled. Next, we examine the test environment and assign a value of 1 to the lowest symbolic cycle value, which is $(\alpha-2)$, and calculate all the cycle values accordingly. Thus, if we control *i1* and *i2* in cycle 1 and observe *v6* in cycle 4, we have tested *+4* with a test vector applied at the primary inputs. From the BIST point of view, this means that if we place a pair of pseudorandom vectors at *PI-port1* and *PI-port2* in cycle 1 derived from PRPGs, and analyze the output at *OUT-PORT1* in cycle 4 with an MISR, we have tested adder *ADD3* with a pseudorandom pattern.

2.4 Test multiplexer insertion

Sometimes, it may not be possible to achieve a test environment for an RTL element using the existing behavior of the circuit. In the example *Tseng*, *ADD2* is symbolically untestable since operation *+2* mapped to *ADD2* does not have a test environment. This is because of the reconvergent fanout of constant *c2*. (Note that *c2* is a constant in the first iteration of the CDFG only, but it can be controlled to an arbitrary value at the beginning of the second iteration). *c2* needs to be non-zero for providing general controllability to the left input of *+2*, but also needs to be zero for propagating its output *v4* through ||1. In such cases, we solve the problem by adding a test multiplexer, as shown in Figure 1. Thus, variable *v4*, which is mapped to *REG4*, is observable at the primary output *OUT-PORT2*. The procedure for adding test multiplexers is given in [11]. It utilizes a badness count for TCDF variables to identify testability bottlenecks in the TCDF, and thus the RTL circuit to which the TCDF is mapped. It then inserts appropriate test multiplexers at those points. This procedure tries to avoid the critical path, whenever possible, while adding the test multiplexers. Since a large number of solutions typically exist for overcoming a testability bottleneck, avoiding the critical path is usually possible. The select signals of these test multiplexers come from a BIST controller which is described later.

Registers may be similarly targeted for test environment generation by generating a test environment for any one variable mapped to it. Test environments can also be generated for multiplexers. However, test multiplexers are added to overcome test environment bottlenecks for modules only, since they become an overkill for the other RTL elements. The fault coverage is not compromised because of this, as will be evident from the experimental results. From the above discussion, it is clear that the test environment of an RTL element guarantees that a test path exists from the primary input(s) to the input(s) of the element and from the output of the element to a primary output. We will be utilizing this fact for the BIST scheme.

2.5 Modifications for control-flow intensive circuits

We explain the modifications for control-flow intensive designs with the help the *Barcode* example shown in Figure 5. This circuit is used as a preprocessor to read barcodes printed on objects. The main problem in extending the approach outlined before lies in the status signals fed by the data path to the controller. Now the TCDF for an operation is usually input vector dependent as the data and control flow of the data path change according to the input vectors applied. Hence, in order to generate a fixed TCDF for an operation during the test mode, the status inputs of the controller

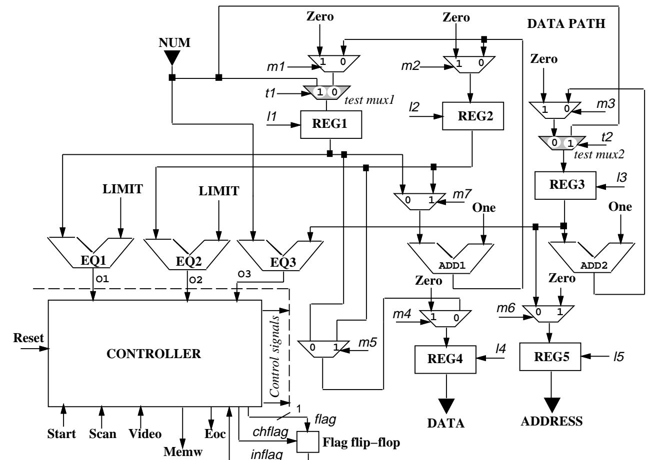

Figure 5: RTL circuit of *Barcode*

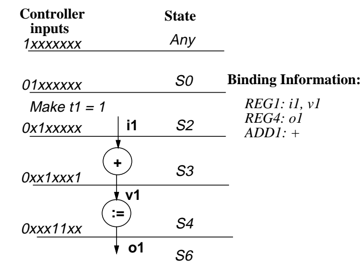

Figure 6: TCDF for testing ADD1 in *Barcode*

need to be controlled. The TCDF generated for the incrementer, *ADD1*, is shown in Figure 6. In the figure, the (:=) operation denotes a register-to-register transfer. Here, since there is no direct input possible from the primary input to the module input, the extraction procedure does a shortest path analysis on the controller state transition graph, and creates a test path by placing the test multiplexer *test mux1* as shown in Figure 5 [9]. Then, the TCDF is generated along with the controller input signals and the test multiplexer select signals required to generate such a TCDF. During the BIST mode, these signals are fed to the controller from a BIST controller. Thus, a TCDF for *ADD1* is guaranteed. After this, test environment generation can proceed as explained in Section 2.3. A single TCDF is usually not enough to test all data path elements in a control-flow intensive RTL circuit.

2.6 Synthesizing a BIST controller

The BIST controller is instrumental in guaranteeing the existence of test environments for all the RTL elements in the test mode. Let us take the example of *Barcode*. The external inputs of the controller, the status signals and the test multiplexer select signals required to test module *ADD1* are shown in Figure 7(a). The signals required to test module *ADD2* are shown in Figure 7(b). The BIST controller can be implemented as a finite-state machine (FSM) that just generates all these input signal sequences one after another. Hence, a trivial way of synthesizing the BIST controller would be to just append the input signal sequences required for the test environments of all the RTL elements, and derive an FSM for those many cycles, as shown in Figure 7(c). In the test mode, each RTL element is tested with pseudorandom vectors from the PRPGs placed at the primary inputs, and the MISRs at the primary outputs, in a sequential fashion. When the BIST controller steps through all its states, it resets to the first state and each RTL element continues getting its pseudorandom vectors, and so on. However, this trivial approach can lead to large overheads. To reduce the BIST controller overheads, we try to merge the input signal sequences for various test environments. For example, the input sequences of *ADD1* and *ADD2* can be merged to test both the modules simultaneously with the same pseudorandom vector at the primary inputs,

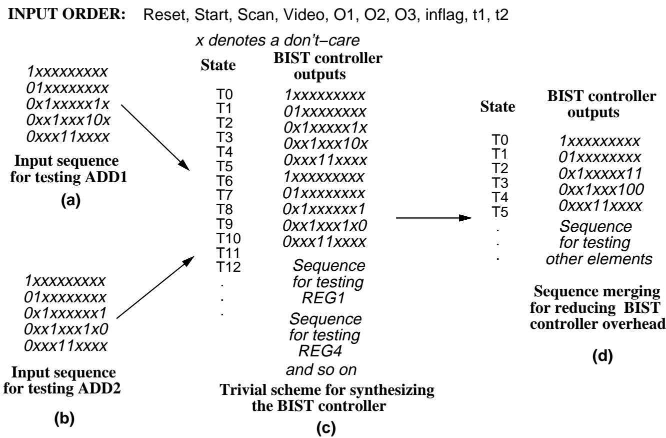

Figure 7: BIST controller synthesis for *Barcode*

as shown in Figure 7(d). This reduces the number of states in the BIST controller, and consequently the overhead.

The compaction technique that we use here is similar to the ones used in test vector compaction for automatic test pattern generation. First, a test environment and the corresponding controller and test multiplexer input sequence for an untested RTL element are generated. Such a sequence typically has many don't-care values for unspecified signals, as shown in Figure 7. Using this initial sequence as a constraint, static test compaction techniques are used to merge other input sequences with it to come up with compact sequences. During compaction, we use existing test set compaction techniques [15], [16].

During test environment generation for a module, we also check if other RTL elements like registers and multiplexers get tested in the process. For example, while testing module *ADD1* in *Barcode*, we see that registers *REG1* and *REG4* also get tested with the same test environment. This is because in the test environment, there exists at least one variable mapped to each of these registers which is controlled from a primary input and observed at a primary output. In such cases, separate test environments are not generated for testing these registers. This is useful in bringing down the BIST controller overhead further.

3 Imposing a BIST Architecture

Once the BIST controller is synthesized, the test hardware is integrated with the original RTL circuit to facilitate its testing during BIST mode by imposing a BIST architecture on it. A low-overhead solution to this problem is shown in Figure 8. In this figure, the added test hardware is shaded grey. (We assume one extra pin to provide the *Test* mode signal.) The controller outputs are multiplexed with a data path primary output port to facilitate testing of the controller [10]. This multiplexer does not result in a delay overhead, as the delay of a multiplexer or even a multiplexer tree at the output of a primary output register is usually much less than the clock period.

The status signals are made directly observable by multiplexing the status signal lines from the data path with an output port, and made controllable during BIST mode by feeding the status inputs of the controller from the BIST controller. The BIST controller is activated during test mode by the *Test* pin. Its *reset* signal is connected to the controller *reset*. The BIST controller feeds: (i) the *select* signals of the test multiplexers that are added to the circuit, (ii) two bits S_0 and S_1 that control the select signals of multiplexer **M** and the output multiplexer, respectively. In the normal mode, the test-multiplexer select lines, the *Test* signal, and signals S_0 and S_1 should be 0.

The data path and the controller are tested separately. In the test mode, the *Test* pin is set. While testing the data path, the BIST controller gives a suitable control flow which is achieved by controlling the controller input signals to a desired sequence. Signals S_1 and S_0 are both set to 0 for testing the data path elements other than the ones that produce the status signals for the controller. At

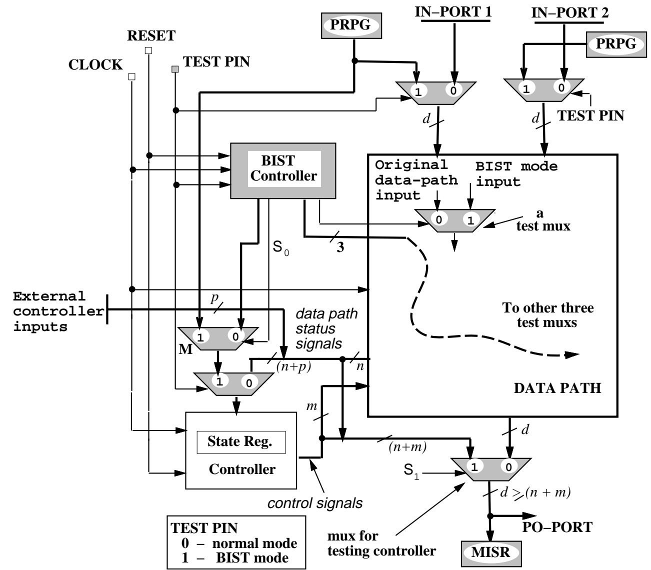

Figure 8: The BIST architecture

specified cycles, we need to reconfigure the data path using the test multiplexers. The BIST controller also provides these select signals. While testing the data path elements that produce the status signals, such as comparators, we need to observe the status signals coming out of the data path to get full observability of the element outputs. In this case, the BIST controller makes signals S_1 and S_0, 1 and 0 respectively. At each cycle, pseudorandom vectors are fed into the primary input ports from PRPGs and the outputs captured at the primary output ports with MISRs. While testing the controller, we can directly control the controller inputs using the PRPG at the primary input port and directly analyze controller outputs by the MISR at the primary output port. The BIST makes S_1=S_0=1 to obtain this particular configuration. In some rare cases, it might so happen that the sum of the number of controller output bits and the status output bits $(n+m)$ exceeds the number of output bits of the data path (d). In that case, the MISR width at the output port has to be increased to capture these extra bits.

It is frequently possible to convert existing input registers (registers connected to primary input ports) to PRPGs and output registers (registers connected to primary output ports) to MISRs, instead of adding extra PRPGs and MISRs. This reduces the overhead even further. However, this can only be done if the occurrence of a circular path can be avoided in the BIST mode. In other words, the input and output registers should not be involved in any test environment in a circular path [17]. For example, in the RTL circuit of *Tseng*, input registers *REG5* and *REG6* can be converted to PRPGs, whereas in the RTL circuit of *Barcode*, output registers *REG4* and *REG5* can be converted to MISRs.

4 The RTL BIST Scheme

In this section, we formalize our RTL BIST scheme. The complete scheme is summarized in Figure 9. First, we extract a controller state table from the controller logic netlist. This can be done by a state machine extraction program starting from the reset state of the controller. Then we select an untested RTL data path element. We extract a TCDF that can be used to test this element along with the controller input sequence required to generate it. This is done with the help of the procedure given in [9] using the RTL data path and the controller state table. Next, we perform a symbolic testability analysis (STA) of the TCDF and try to obtain test environments of as many untested data path elements as possible. This is done using the symbolic justification and propagation techniques presented in [10] and [11]. This procedure may add additional test multiplexers to the data path to increase the controllability and observability of the circuit at certain points where there are testability bottlenecks.

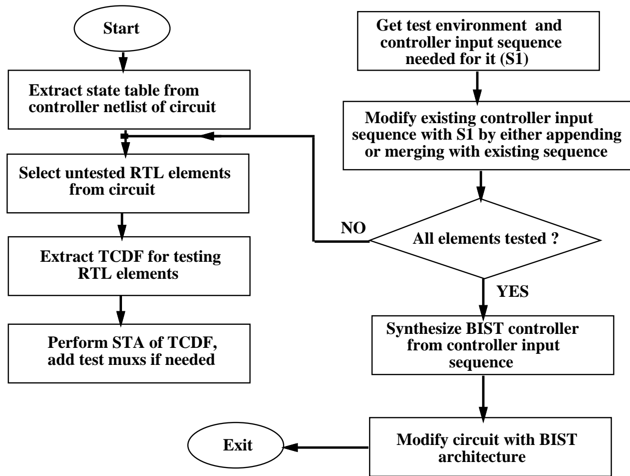

Figure 9: The RTL BIST Scheme

Table 1: Circuit size and DFT hardware statistics

Circuit	Bit-width	# lits.	# FFs	# test muxs	CPU time (sec.)
Tseng	16	5246	100	1	27.5
Paulin	16	6826	116	1	3.5
GCD	16	1329	50	0	0.0
Barcode	16	1518	84	2	1.3
Amvabs	12	1456	40	1	0.8
Amvdual	12	2056	46	1	0.9

The input sequence at the select signals of these test multiplexers needed for the particular test environment is noted. The controller input sequence and the test multiplexer select signal sequence are stored. If possible, they are merged with existing sequences generated for testing other data path elements. The above process is repeated until all data path elements are tested. Then a BIST controller is synthesized from the stored input sequences. Test environments are analyzed to detect the occurrence of circular paths. Input and output registers, for which circular paths are not a problem, are converted to PRPGs and MISRs, respectively. Otherwise, extra PRPGs and MISRs are added along with test multiplexers at the primary input ports and primary output ports of the circuit only. Finally, the circuit is integrated with the BIST components using the BIST architecture described in the previous section.

5 Experimental Results

In this section, we present experimental results obtained by applying our BIST scheme to six example circuits. Among these, *Tseng* and *Paulin* are data-flow intensive circuits literature [13]. *GCD* and *Barcode* are control-flow intensive circuits taken from [18]. *Amvabs* and *Amvdual* are two subcircuits of an MPEG decoder that was designed in the industry. The area and delay results were obtained after technology mapping the gate-level implementation of the RTL circuits using the *stdcell2_2.genlib* cell library in the SIS [14] logic synthesis system.

Table 1 shows the characteristics and specifications of these circuits. The BIST overheads are reported in Table 2 and the testability results are shown in Figure 10. In Table 1, Column 2 shows the bit-width of the circuit data paths. In Columns 3 and 4, the literal-counts of the original technology-mapped circuit and the number of flip-flops are given, respectively. The number of test multiplexers added to the data path by the DFT procedure is given in Column 5. Note that this number just shows the number of extra multiplexers in the data path and does not include the multiplexers in the BIST architecture which are added by default to all circuits, and which have been taken into account while calculating the overheads in Table 2. In Column 6, the CPU time required for symbolic testability analysis is given in seconds. This is on a SparcStation 20 with 256 MB DRAM. Since the analysis is symbolic, it is independent of the bit-width of the data path. Hence, the testability analysis time would remain the same even for 32-bit wide data paths. This is a major advantage over many previous testability analysis schemes [5] where the CPU time for analysis explodes with an increase in the data path bit-width.

Table 2: DFT hardware placement overheads

Circuit	*Area*			*Delay*		
	Orig.	*Mod.*	*Ovhd.* (%)	*Orig.* (ns)	*Mod.* (ns)	*Ovhd.* (%)
Tseng	58654	60531	3.2	70.2	71.8	2.3
Paulin	72345	74443	2.9	74.3	75.7	1.9
GCD	16734	18893	12.9	55.3	56.9	2.9
Barcode	20516	23573	14.9	34.2	35.9	5.0
Amvabs	19564	21168	8.2	29.7	30.2	1.7
Amvdual	28765	31728	10.3	32.3	32.9	1.9

In Column 2 of Table 2, the original area of the circuits after technology mapping is given. This is a relative figure obtained from the layouts of the standard cells used, and hence has no units. Column 3 shows the area after the circuits have been modified by the BIST architecture and test hardware. The area overhead is mainly due to the extra PRPGs and MISRs, the BIST controller, the *Test* pin, the extra logic for making the comparators random-pattern testable, and test multiplexers in the data path and BIST architecture. The percentage overheads are given in Column 4. In Columns 5, 6 and 7, the corresponding figures for delay are provided. The delay represents the clock period in *nanoseconds*. The average area and delay overheads are 6.4% and 2.5%, respectively (the average is calculated based on total area/delay of all the examples for the original and modified cases). The area and delay overheads are much smaller than conventional BIST techniques due to the utilization of functional information and existing data and control flows of the circuit in the test mode.

In Figure 10, the testability results for the original circuits (without any testability hardware) and the circuits augmented by our BIST scheme are shown. The fault coverage numbers are obtained by fault simulating the pseudorandom patterns obtained from the PRPGs (linear feedback shift registers were used as PRPGs) on the circuits using PROOFS [19]. The fault coverage numbers obtained by our method are above 99% for all the examples (with a minimum of 99.1% at 4,500 cycles for *Amvdual* and a maximum of 99.7% for *Paulin*), and the number of test cycles required to get over 99% fault coverage is never more than 4,000.

To check if most of the untested faults in the original circuits were random-pattern resistant, we increased the number of clock cycles to 40,000. By doing so, the increase in the fault coverage (compared to the fault coverage at 4,500 clock cycles) for the original circuits of *Tseng*, *Paulin*, *GCD*, *Barcode*, *Amvabs* and *Amvdual* was only 2.1%, 0.3%, 3.2%, 4.0%, 3.2%, and 5.5%, respectively. This points to the need of employing a BIST scheme, such as the one presented in this paper.

In order to see how different parameters vary with a change in bit-width, we evaluated the BIST architecture for 8, 16 and 32 bit wide data paths of the example *Tseng*. Figure 11 shows how area overhead (AO), delay overhead (DO), and number of cycles required to attain 99% fault coverage (#cycles_for_99%_fc) vary with bit-width. Similar results were obtained for other examples as well. The AO and DO curves are predictable since the testability hardware does not scale as fast as the data path size. The almost linear increase in #cycles_for_99%_fc is also very encouraging.

6 Conclusions

In this paper, we presented a new BIST scheme for testing RTL controller/data path circuits. The scheme uses functional information of modules and symbolic testability analysis techniques to obtain test environments of all the RTL elements in the circuit. The test environment of an element guarantees that the element can be fed by pseudorandom vectors provided at the primary inputs and its test response observed at a primary output. If it is not possible to generate the test environment of a module, then test multiplexers are added to the circuit at suitable points to increase the controllability and observability of the circuit. Then a BIST controller is

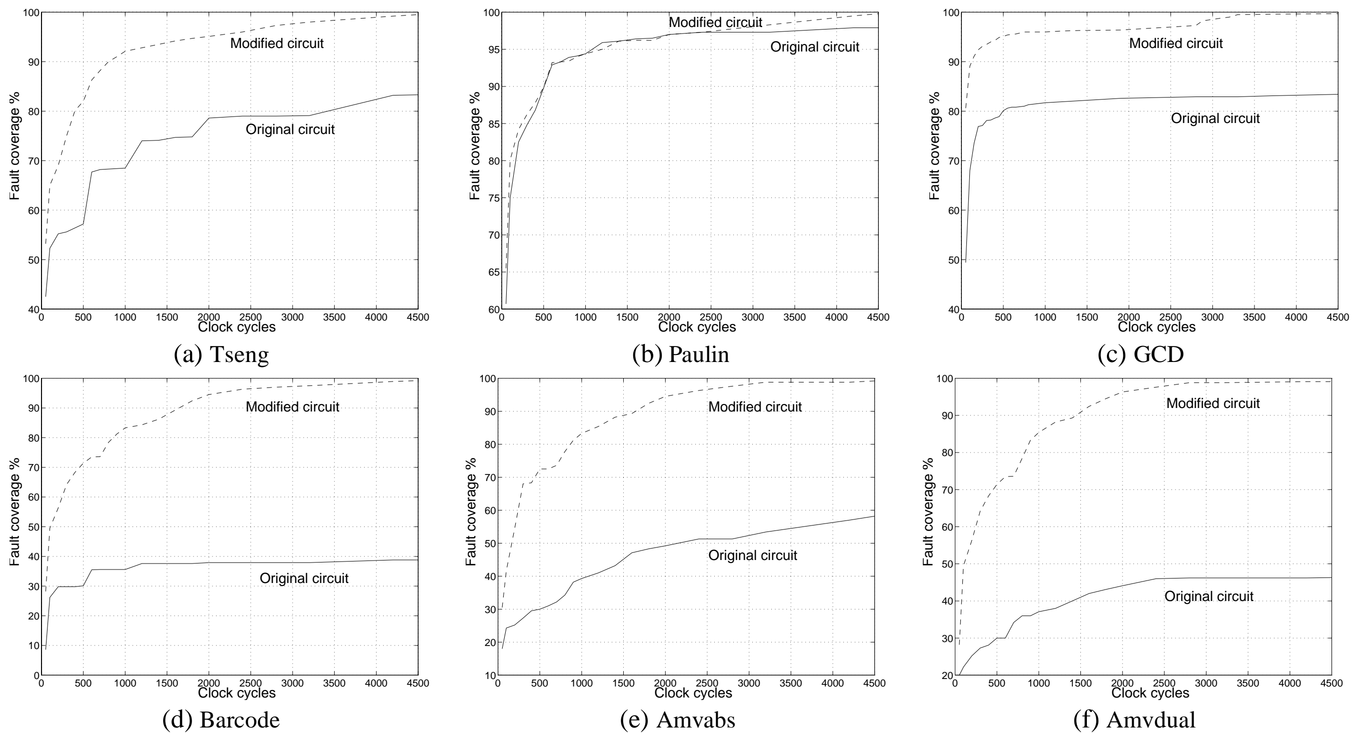

Figure 10: Fault coverage curves for the six example circuits

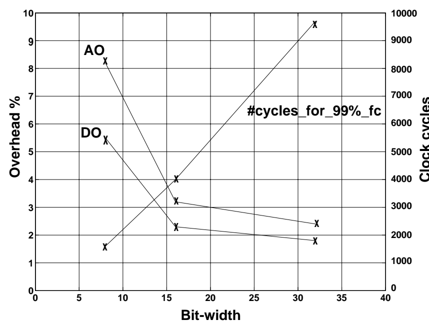

Figure 11: Effect of bit-width on the BIST parameters

synthesized to produce the necessary values of the status inputs of the controller in the test mode and the test multiplexer select signal values. Finally, PRPGs and MISRs are added at the primary inputs and primary outputs of the circuit respectively, and a BIST architecture imposed on the whole RTL circuit. The advantages of the scheme are: (i) comparatively low area and delay overheads, (ii) very high fault coverage, and (iii) low test application times.

References

[1] M. Abadir and M. Breuer, "Constructing optimal test schedules for VLSI circuits having built-in test hardware," in *Proc. Int. Symp. Fault Tolerant Computing*, pp. 165-170, June 1985.

[2] M. Abramovici, M.A. Breuer, and A.D. Friedman, *Digital Systems Testing and Testable Design*, IEEE Press, New York, 1990.

[3] H. Schnurmann, E. Lindblom, and R.G. Carpenter, "The weighted random test-pattern generator," *IEEE Trans. Comput.*, vol. C-24, pp. 695-700, July 1975.

[4] C.J. Lin, Y. Zorian, and S. Bhawmik, "PSBIST: A partial-scan based BIST scheme," in *Proc. Int. Test Conf.*, pp. 507-516, Oct. 1993.

[5] S.S.K. Chiu and C. Papachristou, "A built-in self-testing approach for minimizing hardware overhead," in *Proc. Int. Conf. Computer Design*, pp. 282-285, Oct. 1991.

[6] C. Papachristou and J. Carletta, "Test synthesis in the behavioral domain," in *Proc. Int. Test Conf.*, pp. 693-702, Oct. 1995.

[7] M. Nourani, J. Carletta, and C. Papachristou, "A scheme for integrated controller-data path fault testing," in *Proc. Design Automation Conf.*, pp. 546-551, June 1997.

[8] N. Mukherjee, M. Kassab, J. Rajski, and J. Tyszer, "Arithmetic built-in self test for high-level synthesis," in *Proc. VLSI Test Symp.*, pp. 132-139, May 1995.

[9] I. Ghosh, A. Raghunathan, and N.K. Jha, "A design for testability technique for RTL circuits using control/data flow extraction," in *Proc. Int. Conf. Computer-Aided Design*, pp. 329-336, Nov. 1996.

[10] S. Bhatia and N.K. Jha, "Behavioral synthesis for hierarchical testability of controller/data path circuits with conditional branches," in *Proc. Int. Conf. Computer Design*, pp. 91-96, Oct. 1994.

[11] I. Ghosh, A. Raghunathan, and N.K. Jha, "Design for hierarchical testability of RTL circuits obtained by behavioral synthesis," *IEEE Trans. Computer-aided Design*, vol. 16, pp. 1000-1001, Sept. 1997.

[12] I. Ghosh, N.K. Jha, and S. Bhawmik, "A BIST scheme for RTL controller-data paths based on symbolic testability analysis," Tech. Rep. CE-J98-005, EE Dept., Princeton Univ.

[13] L. Avra, "Allocation and assignment in high-level synthesis for self-testable data paths," in *Proc. Int. Test Conf.*, pp. 463-471, June 1991.

[14] E. Sentovich, K. Singh, C. Moon, H. Savoj, R. Brayton, and A. Sangiovanni-Vincentelli, "Sequential circuit design using synthesis and optimization," in *Proc. Int. Conf. Computer Design*, pp. 328-333, Oct. 1992.

[15] P. Goel and B.C. Rosales, "PODEM-X: An automatic test generation system for VLSI logic structures," in *Proc. Design Automation Conf.*, pp. 260-268, June 1981.

[16] I. Pomeranz, L.N. Reddy, and S.M. Reddy, "COMPACTEST: A method to generate compact test set for combinational circuits," *IEEE Trans. Computer-Aided Design*, vol. 12, pp. 1040-1049, July 1993.

[17] L.J. Avra and E.J. McCluskey, "Synthesis for scan dependence in built-in self-testable designs," in *Proc. Int. Test Conf.*, pp. 734-742, Oct. 1993.

[18] P.R. Panda and N.D. Dutt, "1995 high-level synthesis design repository," in *Proc. Int. Symp. System Level Synthesis*, pp. 170-174, Sept. 1995.

[19] T.M. Niermann, W.T. Cheng, and J.H. Patel, "PROOFS: A fast, memory efficient sequential circuit fault simulator," in *Proc. Design Automation Conf.*, pp. 535-540, June 1990.