

Integrated Test of Interacting Controllers and Datapaths

Mehrdad Nourani

The University of Texas at Dallas

and

Joan Carletta

The University of Akron

and

Christos Papachristou

Case Western Reserve University

In systems consisting of interacting datapaths and controllers and utilizing built-in self test (BIST), the datapaths and controllers are traditionally tested separately by isolating each component from the environment of the system during test. This work facilitates the testing of datapath / controller pairs in an *integrated* fashion. The key to the approach is the addition of logic to the system that interacts with the existing controller to push the effects of controller faults *into the data flow*, so that they can be observed at the datapath registers rather than directly at the controller outputs. The result is to reduce the BIST overhead over what is needed if the datapath and controller are tested independently, and to allow a more complete test of the interface between datapath and controller, including the faults that do not manifest themselves in isolation. Fault coverage and overhead results are given for four example circuits.

Categories and Subject Descriptors: B.5.3 [**Register-Transfer-Level Implementation**]: Reliability and Testing—*Built-in Tests*

General Terms: Design, Reliability

Additional Key Words and Phrases: built-in self-test, register transfer level, synthesis-for-testability

1. INTRODUCTION

Systems consisting of interacting datapaths and controllers are typically designed by synthesizing and testing the datapath and controller independently, even though the two operate as an inseparable pair. This separation can cause difficulties in testing; even if the datapath and controller are designed such that they are 100%

Current addresses are: M. Nourani, Dept. of Electrical Engineering, The University of Texas at Dallas, P.O. Box: 830688, EC 33, Richardson, TX 75083-0688; J. Carletta, Dept. of Electrical Engineering, The University of Akron, Akron, OH, 44325-3904; C. Papachristou, Dept. of EECS, Case Western Reserve University, 10900 Euclid Avenue, Cleveland, OH 44106-7071.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works, requires prior specific permission and/or a fee.

© 2000 by the Association for Computing Machinery, Inc.

testable taken separately, when the two are taken in combination the testability may be severely degraded [Dey et al. 1995]. In addition, separating tests for the datapath and the controller may result in neglecting the control/status signals used to communicate between the two. Moreover, some faults can be seen only when modules interact with each other, such as faults due to phenomena like crosstalk and reflection [Bakoglu 1990], faults that create signal slew among cores receiving the same signal [Sparmann et al. 1995], and faults that cause excessive power consumption in the circuit [Nourani et al. 1997].

Few, if any, synthesis tools address the issue of how to test the datapath and controller of an interacting pair in an integrated way. The main goal of our work is to test a controller-datapath pair realistically and quickly, without neglecting the signals used for communicating between the two. The key to a successful integrated system test of a controller-datapath pair is to provide a method to propagate and observe the effect of certain controller faults *through the datapath*, so that they can be observed at the datapath registers rather than at the controller outputs. In this way, we can avoid the test hardware overhead associated with observing the controller outputs directly. This approach will test the interconnects between datapath and controller more effectively than would separate tests of datapath and controller. Moreover, our approach can detect certain type of redundant controller faults that, although they may not affect the overall system functionality of the controller-datapath pair, do have deleterious effects on the system, such as increased power consumption. At the same time, our approach will not substantially increase the overall system cost.

The basis of our technique is the addition of a small finite state machine (FSM) that interacts with the main controller FSM, for the purpose of making controller faults observable at the datapath registers. This state machine is designed to work independent of implementation details, the design-for-testability technique, and the design tools used to synthesize the controller and datapath.

1.1 Related Work

There are many well-known problems in controller optimization. Work presented in [Devadas and Newton 1989], [Ashar and Devadas 1991] and [Lagnese and Thomas 1989] apply finite state machine (FSM) decomposition techniques to improve controller area or performance. The importance of state assignment is discussed in [DeMicheli et al. 1984] and [Devadas et al. 1988], among others. Recently, the effect of controller design on power consumption has been explored in [Landman and Rabaey 1995]. The work of [Benini and DeMicheli 1994] uses special state assignments to reduce power, while [Benini et al. 1994] adds some combinational logic to the original controller to avoid inactive state transitions.

For self-testable designs based on BIST (Built-In-Self-Test), research involving controllers focuses on test plan and test scheduling [Abadir and Breuer 1985] [Kime and Saluja 1982] [Jone et al. 1989]. [Hellebrand and Wunderlich 1994] uses additional test registers to implement the system function supporting self-testable pipeline-like controller. The MMC control scheme in [Breuer and Lien 1988] is able to test a chip in a module via a boundary scan bus. A local dedicated test controller is discussed in [Joersz and Kime 1987] to reduce the overall test overhead. Some other heuristics and examples are [Eschermann and Wunderlich 1990], which uses

special state assignment and feedback polynomial, [Mukherejee et al. 1991], which uses one-hot encoding, and [Breuer et al. 1988], which employs microprogrammed and hard-wired implementations of the controller. The method proposed in [Kuo et al. 1995] adds some additional edges to FSM to make the corresponding architecture testable. The authors of [Hsu and Patel 1995] note that some FSMs are not easily controllable because they require a long synchronizing sequence, and propose a method to improve FSM testability.

None of these approaches use a unified model to test the controller-datapath pair. Instead, datapath and controller are tested separately in different test sessions. For these approaches, the basic test scheme is similar to what [Bhatia and Jha 1994] proposed; the controller output signals are multiplexed with some or all of the datapath primary outputs, thus making them directly observable. Observing the controller and datapath faults separately, in general, implies more test time (due to separate test session) and more overhead (due to direct observation of each). [Dey et al. 1995] observed that even when the controller and datapath are 100% testable separately, the combination of them has usually much lower coverage. This degradation, in their opinion, occurs due to the correlation and dependency between the control signals. Then, to improve testability the authors propose to redesign the controller by breaking the correlation between the control signals.

1.2 Organization of Paper

This paper is organized as follows. Section 2 presents a system model for testing a datapath/controller pair. Issues central to the testing of controllers are presented in Section 3. These issues include a classification of the types of faults in the controller, and the impact that the controller faults have on the nonfunctional aspects of the system of manufacturing and power. Section 4 details our solution for integrated datapath/controller testing. Experimental results are shown in Section 5, and concluding remarks are in Section 6.

2. MODEL

In our system model, introduced in [Carletta and Papachristou 1995], the datapath is represented behaviorally by a *data flow graph* (DFG), in which nodes represent operations such as addition and multiplication, and edges represent the transfer of data. Structurally, the datapath consists of arithmetic logic units (ALUs), multiplexers, registers, and busses, and is responsible for all data computations. We assume that the datapath is composed of functional blocks like that shown in Figure 1. Behaviorally, the controller is viewed as a state diagram that specifies the time steps in which the various operations in the data flow graph are done. For this work, controllers are implemented structurally as finite state machines using random logic.

The traditional approach to testing a controller-datapath pair is shown in Figure 2. The pair is completely split, and the two parts are tested independently. If the controller and datapath can be tested at different times, multiplexers may be used to share the test resources; for example, one TPGR and one multiplexer can be used instead of the two TPGRs shown on the figure. For traditional designs, for which design-for-testability decisions are made for one component without thinking about how the component will be used in the context of the pair, this approach

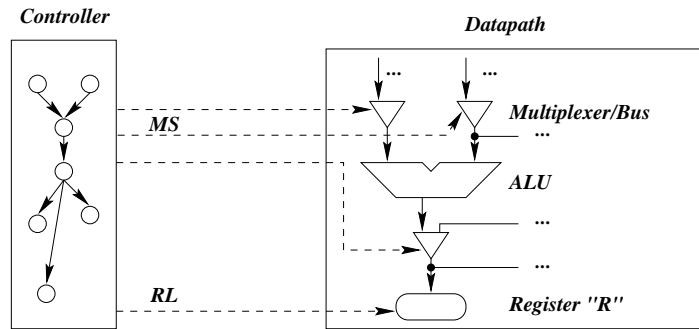


Fig. 1. One functional block defining our datapath style.

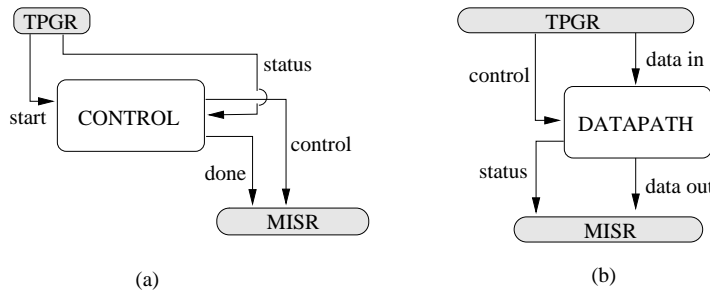


Fig. 2. Separate testing: (a) for the controller; and (b) for the datapath.

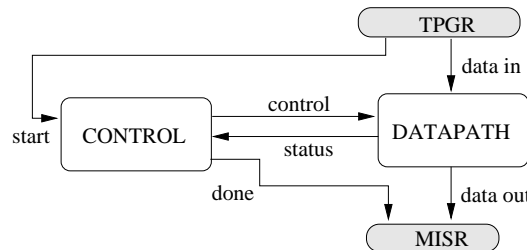


Fig. 3. Completely integrated testing of datapath and controller.

has the advantage that it tests the components as the designers intended. Therefore, fault coverage for individual components will be as high as the design allows. However, this approach is undesirable because it does not test the interface between controller and datapath, and because it requires a large amount of insertion. These disadvantages are addressed by a completely integrated approach, as shown in Figure 3. In an integrated approach, the controller-datapath pair is treated as an inseparable unit, and the two parts are tested simultaneously.

One motivation for treating a controller-datapath pair as an integrated system is shown in Figure 4. The figure shows a controller and datapath, with a single control line extending from one to the other. In an independent test of the controller, this line would be tapped so that the output of the controller could be observed

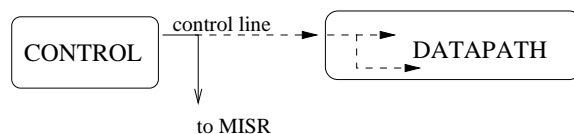


Fig. 4. Illustration of disadvantage of separate testing.

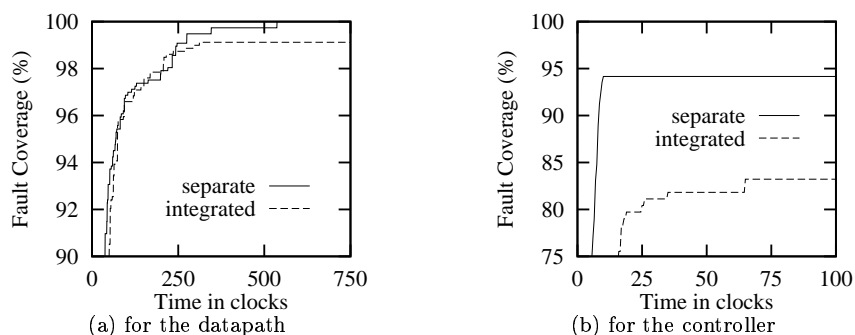


Fig. 5. Fault coverage curves for the datapath and the controller of a differential equation solver, when tested separately and when tested together as pair.

directly by an MISR. Even though such an arrangement allows good observation of the controller, there is still a segment of the control line, shown on the figure as a dotted line, that can not be observed. This control line extends far into the datapath, and may control multiple registers and / or multiplexers. Even if part of the segment is tested during the test of the datapath, it is difficult to ensure coverage of the complete line. In particular, testing the line as a series of segments may miss problems due to phenomena like crosstalk, reflection and signal skew that show up only during operation of the overall system. Note that if the controller and datapath are laid out in separate blocks, the control lines may be of significantly longer length than other wires, and may therefore be more susceptible to faults. By doing an integrated test, for which we observe the controller *through* the datapath, we ensure that we test the *entire* control line.

Figure 5 uses a differential equation solver as an example to compare separate and integrated testing of a controller-datapath pair. Separate fault coverage curves are shown for the datapath and controller. As can be seen from the curves, fault coverage is degraded when the datapath and controller are tested together; this is because controllability and observability of the control and status lines are reduced; in the integrated test, these lines are no longer directly accessible. The goal of this research is to achieve a high quality integrated test by overcoming these difficulties.

This work concentrates exclusively on the test of the controller in an integrated environment. The focus is on enhancing the observability of the control lines through the datapath through the addition of some extra logic to the system. The work complements previous work, reported in [Carletta and Papachristou 1997], that develops a scheme under which the datapath can be tested in an integrated way. In that work, the datapath is exercised according to its normal behavior even during test; guidelines based on high-level testability metrics are given for modify-

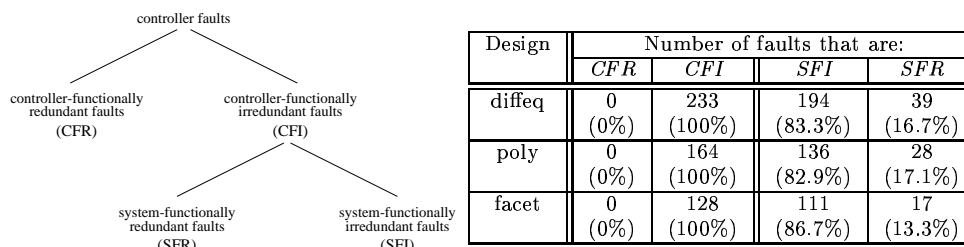


Fig. 6. Classification of controller faults, with percentage of faults belonging to each category in example controllers.

ing the datapath to ensure that the quality of such a test is sufficient. These two pieces of work can be used together to ensure a full integrated system test.

3. CONTROLLER FAULT ANALYSIS

We classify stuck-at faults internal to the controller into several groups as shown in Figure 6 [Carletta et al. 1999]. The first division is based on whether a fault affects the functionality of the controller. By functionality, we mean the input-output behavior of the *synthesized* controller as it operates in normal mode. Faults that never affect the output of the synthesized controller in normal mode are *controller-functionally redundant* or *CFR*. CFR faults can not be detected even by direct observation of the controller outputs during normal mode operation. Detection of these faults may require, for example, the application of transitions that the designer left unspecified, perhaps because some of the states of the controller are unused, or because for some states some input combinations will never occur. The work in [Fummi et al. 1995] shows that controller resynthesis can be used to remove these faults if they are a concern.

Note that a stuck-at fault inside the controller may affect the controller outputs in a sequential way, causing the controller outputs to change only in one or more control steps in the controller state diagram. The other kind of fault, which we call *controller-functionally irredundant* or *CFI*, affects the output of the controller in at least one time step when the controller is running in normal mode. Faults of this kind will be caught in an independent test of the controller, for which we operate the controller in normal mode and observe the controller outputs directly.

We further divide the controller-functionally irredundant faults into two sub-groups, based on whether a fault affects the functionality of the datapath/controller pair *as a system*. *System-functionally irredundant* or *SFI* faults, are those faults that change the input-output behavior of the system as a whole. Some faults in the controller clearly affect the function performed by the datapath; for example, a fault whose effect changes a “care” specification of a multiplexer select line will cause an operation to be done on incorrect data, thereby affecting a change in the results of the computation. *System-functionally redundant* or *SFR* faults are those faults that do not affect the input-output behavior of the system, even though they did affect the input-output behavior of the controller. One example of a system-

functionally redundant fault is a fault that affects bits of the controller output only in time steps when those bits are “don’t care” specifications. For example, a fault may affect a multiplexer select line only in those time steps when the multiplexer is idle. In time steps when no register driven by a multiplexer is loaded, the select line for the multiplexer is a “don’t care”, and the multiplexer does not take part in any register-to-register transfer. Depending on how the controller was synthesized, the select line will be either a 0 or 1. Although the actual value of the select line will make a difference in terms of what signals are propagating locally in the area of the multiplexer, these signals are never written to any register, are never used in computation, and therefore do not affect the function performed by the datapath. If some fault in the controller causes the select line value to change, there is no way to observe the change through the datapath, and the function of the datapath is not affected.

The difference between our fault classification model and that of [Fummi et al. 1995] is subtle, but important. In [Fummi et al. 1995], faults are distinguished in terms of the controller functionality as specified by the designer, whereas we consider the functionality of the *synthesized* controller. In the designer’s specification, some outputs may be unspecified, whereas in the synthesized version specific values have been chosen for all the outputs as a byproduct of the synthesis. In [Fummi et al. 1995], any fault that affects a “care” specification at the controller output is viewed as irredundant. However, some system-functionally redundant faults affect even the “care” specifications for the outputs of the controller. In some sense, these faults are due to redundant logic within the system. Even if the controller and datapath have no redundancy when considered separately, the combined system may have redundancy. For example, suppose that some fault in the controller causes a register to be loaded in a time step when it should not be loaded. If the extra load overwrites some important part of the computation, it will be detectable. However, the extra load may write into a register that is not currently holding a computation value, or that is holding a computation value that will not be used again. In this case, the extra load will not affect the functionality of the datapath. It is possible to determine whether a fault that causes a change on a register load line during a time step is system-functionally redundant by analyzing the lifespans of the variables bound to the register. If no variable is alive during that time step or if the extra load serves only to re-do a previous computation, the fault is system-functionally redundant.

Table 6 shows how the faults in three example controllers, presented more completely in the results section, break down into categories. For these controllers, about 15% of faults in the controller are system-functionally redundant. Synthesizing the controller so that SFR faults do not exist is not trivial; in [Carletta et al. 1999], we show that the key to removing SFR faults is a careful consideration of the meaning of “don’t care” specifications in the context of the controller-datapath pair, and requires an analysis of the lifespans of the variables bound to registers in the datapath. Controllers specified in other ways are likely to contain SFR faults. In particular, controllers for systems utilizing gated clocks, which designate that registers be loaded only when necessary to save the results of a computation, are very likely to contain a significant number of SFR faults.

<i>Inputs</i>			<i>Power [μW]</i>	
<i>Clock</i>	<i>Data Lines</i>	<i>Load Line</i>	<i>Latch</i>	<i>Register</i>
Stopped	Random or Fixed	Random or Fixed	0.07	0.07
Running	Random	Stuck-at-0	56	66
		Random	130	151
		Stuck-at-1	176	221
Running	Fixed	Stuck-at-0	26	25
		Random	68	96
		Stuck-at-1	97	155

(a)

<i>Fault Presence</i>	<i>Power [μW]</i>	<i>% increase</i>
fault-free	1376	-
fault 1	1403	2.0%
fault 2	1498	8.9%
multiple effects	2413	75.4%

(b)

Table I. Power consumption in μW atts for (a) four bit storage components; and (b) a four bit implementation of a differential equation solver in the presence of SFR faults.

3.1 Power and other non-functional effects of SFR faults

The synthesis method used for the controller impacts how many and what kind of controller-functionally redundant and system-functionally redundant faults exist in the controller [Carletta et al. 1999]. The presence of these faults does not affect the functionality of the system. However, these faults may cause undesirable non-functional effects. Whether detection of these faults is important depends on the concerns of the designer, and groups responsible for manufacturing, reliability, and quality assurance.

One non-functional effect of system-functionally redundant faults is increased power consumption. Excessive power consumption may be undesirable in its own right, and can also cause degradation in system performance as the chip heats up. An SFR fault that causes harmless but unnecessary loading of “garbage” values into a register will result in unnecessary power consumption in the register and any combinational logic driven by the register. In essence, such a fault undermines the gated clock scheme used for low power design.

To show the extent of effects on power, we measured the dynamic power consumption of 4-bit storage elements. For implementation, we used components in the 0.8-Micron VCC4DP3 datapath library [VLSI Technology 1993] in the COMPASS Design Automation tools [Compass Design Automation 1993]. We then ran the COMPASS toolset with the “power enable” switch on to report the average power consumption for a large number of random patterns. Table I (a) shows an experiment to measure the power consumption. “Fixed” means that we have fixed that signal to a randomly selected value and kept it unchanged for the entire simulation process. “Random” means that the signal is driven with random patterns. Note that even when the data input to the storage element is fixed, there is a considerable amount of power consumption in the component due to the “clock” signal. A system-functionally redundant stuck-at-1 faults on the load line will cause a dra-

matic increase in power consumption. For random data inputs, power consumption in a latch rises from 56 to 176 μ Watts, and for fixed inputs power consumption rises from 26 to 97 μ Watts. This is 200-300% increase in power of one latch in the worst case and $|\frac{130-176}{130}| = 35.4\%$ to $|\frac{68-97}{68}| = 42.6\%$ on the average.

The above simulation was for a non-embedded storage element. To verify that the same phenomenon occurs when storage elements are embedded in a circuit, we have repeated the power simulation for a complete design that implements a differential equation solver. Here, we are careful to inject only faults that are system-functionally redundant; throughout the experiment, the functionality of the datapath remains the same as in the fault-free case. Table I(b) summarizes the result. Fault 1 and 2 correspond to two different system-functionally redundant single stuck-at-1 faults on two specific register load lines. The presence of fault 1 causes a 2% increase in overall power consumption, while the presence of fault 2 causes a 9% increase. The column labelled “multiple effects” reflects a worst case scenario for this particular example, in which registers load as often as possible without disrupting datapath functionality. In this scenario, multiple registers load multiple times, and the increase in power consumption is a dramatic 75%.

In another example of an undesirable non-functional fault effect, the presence of the fault may be an indication of some manufacturing problem. Taking an example from [Aitken 1995], one manufacturing problem seen in real integrated circuits is cracks in insulation layers. Over time, metal migrates into the cracks, forming shorts. A system-functionally redundant fault caused by this manufacturing problem may indicate more serious problems to come, as more shorts form, and therefore be worth detecting.

4. A DESIGN SOLUTION

This section describes a solution to the controller testing problem that adds a small finite state machine (FSM) to the system. This FSM “piggybacks” onto the original controller, interacting with the controller in such a way that the effects of all controller-functionally irredundant (CFI) faults, both system-functionally irredundant and redundant (SFI and SFR), within the controller are pushed into the data flow, where they can be observed at the outputs of the datapath registers. The goals of our scheme are as follows:

- The scheme should work with any existing (ad-hoc or systematically synthesized) controller / datapath pair without architectural change.
- The overhead for the scheme should be less than the overhead required for the separate test scheme.
- The scheme should complement any test schemes for the controller and datapath indicated by the designer, making it possible to detect SFR and interface faults.
- *All faults* observable directly at the controller outputs, whether system-functionally redundant or irredundant, should be made observable at the datapath registers under the scheme.

The key to the method is to push the effects of controller faults from the controller-datapath interface into the datapath registers. The conditions under which this can be done successfully are explored in Section 4.1. Section 4.2 shows the implementation details of our scheme, and explains how the finite state machine added to the

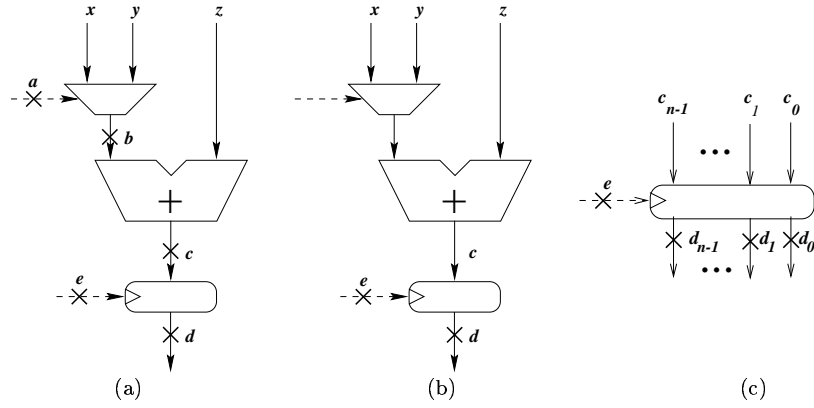


Fig. 7. Propagation of a fault effect on a control line into the data flow (a) for multiplexer select lines; (b) for register load lines; and (c) closer view of register load line case.

system ensures that the necessary conditions are present to observe faults through the datapath registers. In Section 4.3, we show how observation costs can be reduced by observing a single bit of each pertinent datapath register, rather than the whole register.

4.1 Propagation of Controller Faults

In this section, we discuss how to propagate the effect of controller-functionally irredundant (CFI) faults within the controller through the datapath. Any CFI fault will cause at least one output of the controller to change in at least one time step of the control schedule. Barring a detailed gate level analysis of the controller, if we want to be sure to catch all CFI faults within the controller, we must be sure that we can detect any change in a control line during any one time step. The key to our approach is to ensure that changes in the control lines cause changes in the data flowing through the datapath. In what follows, we consider multiplexer select line fault effects, register load line stuck-at-1 effects and register load line stuck-at-0 effects separately.

Figure 7(a) shows a fault effect on the select line of a multiplexer (at point a). The fault causes the wrong path through the multiplexer to be selected in some time step of the control schedule. In that time step, for example, the multiplexer may pass the incorrect value y instead of the correct value x . This will be noticeable at point b as long as $x \neq y$. In turn, the ALU performs the operation $y + z$ instead of the correct $x + z$, and the effect of the fault propagates further into the datapath, to point c . To preserve the fault effect and propagate it to point d , the register must be loaded in the same time step; otherwise, the result of the erroneous operation $y + z$ is never written, and is therefore lost.

Figure 7(b) illustrates a fault effect on a register load line (at point e). Suppose first that the fault causes the load line to be stuck-at-0 in some time step; in this case, the register is not loaded when it should be. Thus, it keeps its old value $c(t - 1)$, rather than obtaining a new value $c(t)$. This will be noticeable at the output of the register (at location d) only if $c(t - 1) \neq c(t)$, i.e., only if the missed load would have written a new value into the register. Assuming that the system

is designed so that redundant computations are not done, this should be the case for at least some of the test patterns.

If a register load line is stuck-at-1 in some time step, the register is loaded when it should not be. This is noticeable at the register output (at location d) only if the new value inadvertently loaded is different from the old value. Referring to Figure 7(b), we see that there are a number of ways for this to happen. First, of all, the multiplexer select line could have changed value since the last time the register was loaded, so that the operand of the ALU comes from a different source. This is noticeable as long as the new source supplies a different data value from the old source (i.e., $x \neq y$). Alternatively, the value of x itself may be changed; since x is coming from another register in the datapath, it is possible that a new value has been written to x . In this case, the inadvertent load will cause $x(t) + z$ to overwrite the correct value, $x(t - 1) + z$.

For all of the faults discussed, if the register is not a primary output register, then multiplexer selects and register loads in subsequent time steps must serve to propagate the erroneous value at the register output to an observable point. For example, in the case of a register load line being stuck-at-1 in some time step, the inadvertent load caused by the fault will be noticed only if the value of the register is used at least once *after* the inadvertent load takes place.

Note that Figure 7 is solely an example; our method is not restricted to this specific architectural style. Having multi-level multiplexers or fanouts at the outputs of arithmetic logic units, multiplexers, or registers does not invalidate the above argument. We elaborate on this shortly, after we explain how the controller faults are observed.

4.2 Implementation

The purpose of the FSM that we add to the system is to allow us to detect the changes in the controller output value by looking at the outputs of the datapath registers, rather than directly at the controller outputs themselves. We justify the method by looking at a single functional block of the datapath (as shown in Figure 1) in a single time step $T = i$. We would like to be able to detect any change in the multiplexer select lines or register load line during this time step, with the following requirements:

- The justification should not depend on the content of register loaded at a time step before i .
- The scheme should work regardless of the values of MS and RL .

The method works by freezing the original controller to expand the time step into two time steps. In the first of the two steps, a *known* value that is *different* from what it is supposed to be under normal operation is loaded into the register. This is accomplished by complementing the multiplexer select lines and loading the register. In the second of the two steps, control signals for normal operation are produced, and the original controller is unfrozen so that it will make the transition to the next normal mode state. This is illustrated in Figure 8. As the figure shows, when testing the controller, a time step that normally would produce the control signals (MS_i, RL_i) is expanded into two time steps, one which produces $(\overline{MS}_i, 1)$, and one which produces (MS_i, RL_i) . Note that this has the side effect

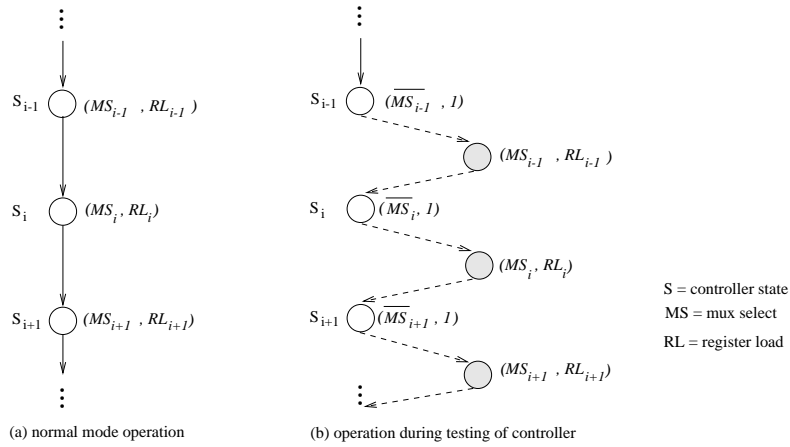


Fig. 8. State diagrams illustrating how the added FSM interacts with the original controller.

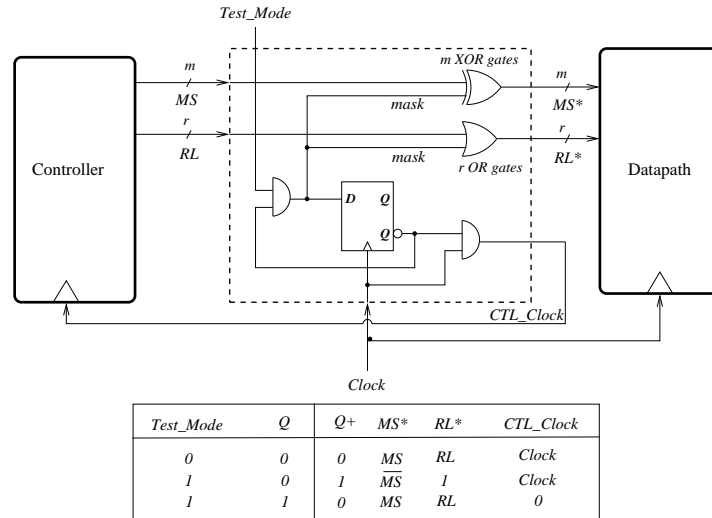


Fig. 9. One possible logic implementation of the FSM added for our scheme.

of slowing down the execution of the control schedule by a factor of two. The logic implementation for the FSM needed to effect these changes is quite inexpensive. Figure 9 shows one possible implementation.

We now elaborate on the role of the added FSM in allowing the detection of CFI faults within the controller. This is best described by Figure 10, which details different cases. Note that this table shows typical active components (see Figure 1) at time step $T = i$. However, for simplicity the subscript i is not shown in the figure. $[R]$ and $[\overline{R}]$ denote the content of register R when multiplexer select lines are MS and \overline{MS} , respectively. Also, MS_f denotes a multiplexer select for which at least one bit is faulty due to a controller fault, and \overline{MS}_f denotes its complement. $[R]_f$ and $[\overline{R}]_f$ denote the content of register R in these two situations.

	Fault-free Circuit		Faulty Circuit			
	\overline{MS}	MS	$s-a-0$ on RL	$s-a-1$ on RL	$s-a-0$ or -1 on MS	
MS_i	\overline{MS}	MS	\overline{MS}	MS	\overline{MS}_f	MS_f
RL_i	1	1	1	0	1	1
Content of register "R"	$\overline{[R]}$	$[R]$	$\overline{[R]}$	$\overline{[R]}$	$\overline{[R]}_f$	$[R]_f$
Result when register "R" is observed			$\xrightarrow{s-a-0 \text{ on } RL \text{ is detected}}$		$\xrightarrow{s-a-0 \text{ or } s-a-1 \text{ on } MS \text{ is detected}}$	

(a) for the case in which "R" is supposed to load.

	Fault-free Circuit		Faulty Circuit			
	\overline{MS}	MS	$s-a-0$ on RL	$s-a-1$ on RL	$s-a-0$ or -1 on MS	
MS_i	\overline{MS}	MS	\overline{MS}	MS	\overline{MS}_f	MS_f
RL_i	1	0	1	0	1	0
Content of register "R"	$\overline{[R]}$	$\overline{[R]}$	$\overline{[R]}$	$\overline{[R]}$	$\overline{[R]}_f$	$\overline{[R]}_f$
Result when register "R" is observed			$\xrightarrow{s-a-1 \text{ on } RL \text{ is detected}}$		$\xrightarrow{s-a-0 \text{ or } s-a-1 \text{ on } MS \text{ is detected}}$	

(b) for the case in which "R" is not supposed to load

 Fig. 10. The effect of interaction between the controller and piggyback on a typical register R at time step $T = i$.

Figure 10 shows how all controller faults that cause changes at the controller outputs (MS and RL lines) in the given time step can be observed by checking the content of register R . The figure is split into two cases:

Case 1. In the fault-free system, the register loads a new value at time step i , i.e., $RL = 1$. Part (a) of the figure shows the contents of register R in the fault-free case, and in the presence of three different kinds of fault effects: stuck-at-0 on RL , stuck-at-1 on RL , and stuck-at-0 or 1 on MS . The arrows show when a difference in the contents of the register indicates that a fault will be detected. From the figure, it is easily seen that any fault which causes RL to be stuck-at-0 or MS to be stuck-at-0 or 1 in this time step will be detected. Note that if a fault causes RL to be stuck-at-1 only in time steps during which (like this one) RL is supposed to be a '1', the fault is controller-functionally redundant, and not targeted by this technique.

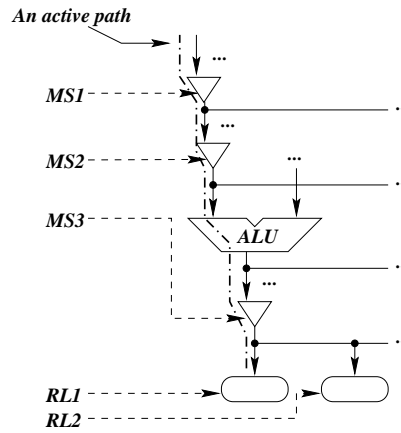


Fig. 11. An active path with multi-level multiplexers and fanouts.

Case 2. In the fault-free system, the register does not load a new value ($RL = 0$) at time step i . Results for this case are shown in part (b) of the figure. The arrows show that any fault which causes RL to be stuck-at-1 or MS to be stuck-at-0 or 1 in this time step will be detected. Note that if a fault causes RL to be stuck-at-0 only in time steps during which RL is supposed to be a '0', the fault is controller-functionally redundant, and not targeted by this technique.

As mentioned earlier, our method in pushing controller faults into the data flow is not restricted to any specific architectural style. In fact, in Figure 10, MS_i refers to all select lines of multiplexers that forward data in time step $T = i$. For example, in Figure 11, where we show the active path through the data flow in step $t = i$, we have $MS = \{MS1, MS2, MS3\}$. \overline{MS} indicates that all select lines of all three multiplexers (including the two multi-level multiplexers) are complemented; this will have the desired effect, that is, it will forward incorrect data to the ALU. Similarly, RL_i refers to all registers loading data from the ALU, whether directly or indirectly (through muxes). In Figure 11, for example, $RL = \{RL1, RL2\}$. Having more fanouts on the multiplexer/ALU outputs or having more registers driven by an ALU could even be beneficial for testing, since the effect of a fault that is traveling from controller to datapath at time step $T = i$ has the potential to influence more components, and more erroneous values are loaded for checking. This feature stems from the fact that *all* multiplexer select lines are complemented and *all* storage elements are loaded in the additional control states.

4.3 Observation of controller faults

An important point is that when the effect of the fault moves from a control line to the data flow, it moves from a single bit line to a multi-bit bus. The fault effect may be seen on one or more lines of the bus. Figure 7(c) shows the transfer of the fault effect from the register load line into an n -bit wide data flow, with individual bits of the data flow shown. In moving the register load line fault effect from location e to location d , which data bits the fault effect will change on the bus at location d depends on the specific values of the data. It is noticeable in some bit of the data

if $x \neq y$, but more specifically, it is noticeable in bit i of the data bus at location d if bit i of x is not equal to bit i of y . From a practical standpoint, if we test the datapath using a reasonably large number of random patterns, it will be possible to observe control line faults without observing *all* the bits of the datapath bus; a single bit will suffice, because, for any bit i that we choose, there are likely to be at least some patterns that cause a change in bit i .

We now explore this argument more quantitatively. Suppose that the patterns being written to a register are random with a uniform distribution and uncorrelated in time. Let $c_i(t)$ denote bit i of the t th pattern written to the register. When we observe the bit i of the register output, we will detect an error that affects the register load line whenever $c_i(t) \neq c_i(t-1)$. In other words, the fault will escape detection only if $c_i(t) = c_i(t-1)$ for *all* patterns written to the register $t = 1, 2, \dots, N$. Assuming that $c_i(t)$ is a random signal, this will happen with probability $\frac{1}{2^N}$, since $c_i(t)$ must take on the same value (0 or 1) for N patterns, and the probability that it takes a particular value in a given pattern is $\frac{1}{2}$. Therefore, under these assumptions, the probability that the fault escapes detection drops exponentially with the test session length, and is quite small even for short test sessions. Note that if all bits of the register output were observed, the probability of the fault escaping detection would be even smaller; for an n bit register, the probability would be $\frac{1}{2^{nN}}$. We acknowledge that in practical circuits, the ALUs in particular influence the randomness of signals and the assumption that $c_i(t)$ remains random is often not valid for all or some bits of a signal. For example, if an ALU multiplies its input by 4, the first two bits of the output remain zero all the time, and their probability will be far from the ideal value of $\frac{1}{2}$.

Empirically, we have observed that for the majority of arithmetic and logic ALUs, randomness reduction does not invalidate our argument, and almost any bit can be used for observation. This can be seen for an example system in the fault coverage curves of Figure 12, which show the effects of observing a single bit (either the most significant bit or the least significant bit of each register) versus observing all datapath register bits. The curves corresponding to single bit observation rise a bit more slowly than the full observation curve, but do reach the same final fault coverage. One can easily perform a behavioral simulation of the register transfer level datapath to find out which bit(s) can be successfully used for observation. In previous work [Harmanani et al. 1994], we presented a randomness analysis of a dataflow graph based on entropy¹. The simulation tool analyzes the behavior and computes the randomness of each bit (and overall signals) generated by the ALUs.

5. EXPERIMENTAL RESULTS

In this section, we demonstrate our approach using several example circuits. The circuits have been synthesized from high level descriptions using the SYNTTEST synthesis system [Harmanani et al. 1992]. The output of SYNTTEST is a register transfer level datapath and state diagram controller. Logic level synthesis is done using the ASIC Synthesizer from the COMPASS Design Automation suite of

¹Entropy of a binary signal X is defined as: $I_X = \sum_{i=0}^{2^{|X|}-1} p_{X,i} \log_2 \frac{1}{p_{X,i}}$, where $|X|$ denotes the bit width of X , and $p_{X,i}$ denotes the probability that X is in state i ($i = 0, 1, 2, \dots, 2^{|X|} - 1$).

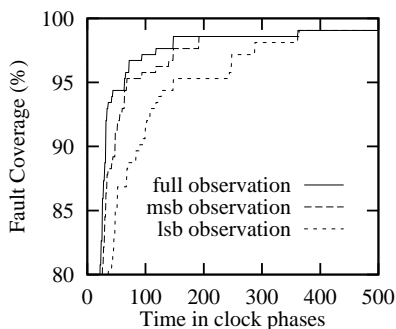


Fig. 12. Fault coverage curves for the proposed test scheme for the controller in a differential equation solver when all datapath register bits are observed, versus observing only the most or least significant bit of each register.

tools [Compass Design Automation 1993], using a finite state machine implementation for the controller and based on a 0.8-micron CMOS library [VLSI Technology 1993]. The test pattern generation registers (TPGRs) necessary for built-in self-test (BIST) are synthesized using COMPASS's Test Compiler. Fault coverage curves are found for the resulting logic level circuits using AT&T's GENTEST fault simulator [AT&T 1993]. GENTEST uses a single stuck-at fault model. The probability of aliasing within the MISRs is neglected, as are faults within the TPGRs and other test circuitry. Although the datapath and controller are tested together, we have separated out the fault coverage curves for the controller to clarify the results.

We work with four example circuits, all with eight bit wide datapaths. The first evaluates the third degree polynomial $ax^3 + bx^2 + cx + d$. Our second example implements a differential equation solver and is a standard high level synthesis benchmark [Gajski et al. 1992]. Our third example is another high level benchmark known as the FACET example [Gajski et al. 1992]. Finally, the fourth example is the well known *fifth order elliptical filter* from [Kung et al. 1985]. For the basis of comparison, we show fault coverage and transistor count results for three different test schemes:

Together Test. Corresponds to a completely integrated test of datapath and controller, for which no additional hardware is added internal to the system. For this case, we drive the inputs of the system from the TPGR, and observe the system outputs, but make no changes at the datapath/controller interface.

Piggyback Test. Corresponds to our new test scheme for facilitating integrated controller/datapath test. We add the piggyback finite state machine at the interface between datapath and controller, then we drive the inputs of the system from the TPGR, and observe one bit of all datapath registers.

Separate Test. Corresponds to an independent test of the controller separate from the system. In this case, we drive the inputs of the controller from the TPGR, and observe the controller outputs directly.

Transistor counts given are for an entire system under a given test scheme, and include the datapath, controller, and any other circuitry necessary for the test. For

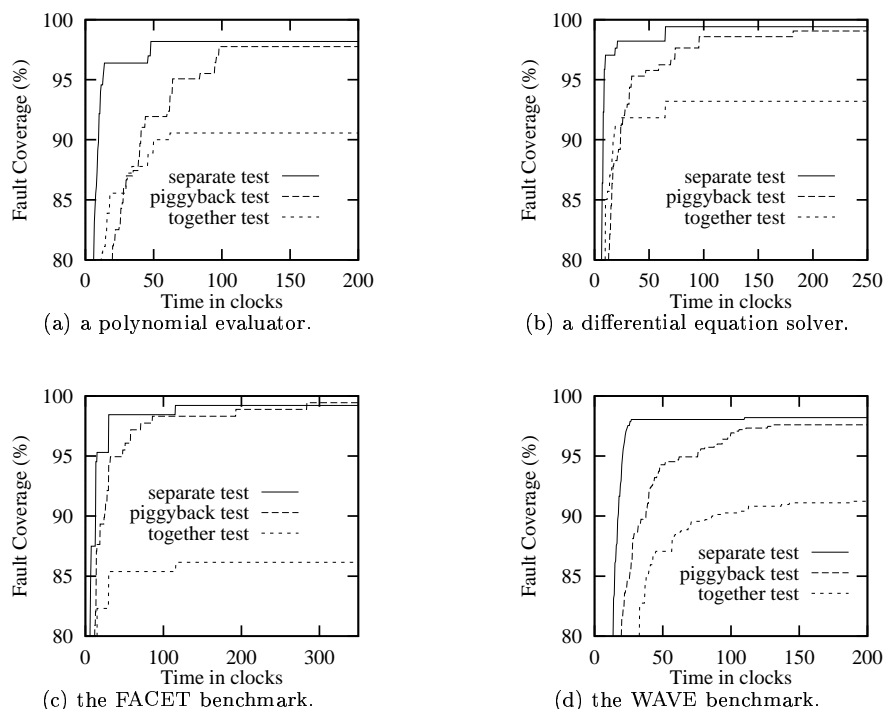


Fig. 13. Fault coverage curves for the controllers of four example circuits.

the “piggyback” test scheme, this includes the transistor count for the added finite state machine.

Fault coverage results for the four examples are shown in parts (a), (b), (c) and (d) of Figure 13, respectively. Fault coverage is for the controller only. On the fault coverage graphs, the vertical axes show fault coverage as the percentage of controller faults detected, and the horizontal axes show time as a function of clock cycles. During each of the tests, the controller is run in normal mode; in the case of the polynomial, the schedule has five control steps, and so for the “separate” and “together” schemes, after, for example, 100 clocks, the controller has run through the schedule $100/5 = 20$ times. For the “piggyback” scheme, the action of the added finite state machine serves to slow the speed down to twice as slow, and so for the same 100 clocks the controller has run through the schedule 10 times. We see from this that one penalty of our approach is that it takes approximately twice as long for the fault coverage curves to saturate using the “piggyback” scheme. This effect is seen for all four of our example circuits. However, this is not a serious limitation, as the controller test is still quite short, especially when compared with the test for the datapath, which may easily be an order of magnitude longer [Carletta and Papachristou 1997].

As expected, the “together” test scheme results in very low fault coverage for the controller in all four examples. This is due to the fact that it is very difficult to observe the controller outputs through the datapath. For this scheme, all system-functionally redundant faults go undetected. On the other hand, the “sep-

d_{in}	the bitwidth of the input data
d_{out}	the bitwidth of the output data
r	the number of registers
m	the number of multiplexer select lines
s	the number of status lines

(a) parameters for system size

	<i>Together</i> <i>Test Scheme</i>	<i>Piggyback</i> <i>Test Scheme</i>	<i>Separate</i> <i>Test Scheme</i>
width of TPGR	$d_{in} + 1$	$d_{in} + 1$	$\max(r + m + d_{in}, s + 1)$
# of associated muxes	$d_{in} + 1$	$d_{in} + 1$	$r + m + s + d_{in} + 1$
width of MISR	$d_{out} + 1$	$\max(r + 1, d_{out})$	$\max(r + m + 1, s + d_{out})$
# of associated muxes	-	$\min(r + 1, d_{out})$	$\min(r + m + 1, s + d_{out})$
Other Gates	-	$m \text{ XOR}, r \text{ OR}$	-

(b) test circuitry required in terms of the parameters.

Table II. Test circuitry required for the three test schemes.

arate” test scheme does a good job of testing the controller. Because this scheme observes the controller outputs directly, this test scheme is capable of catching all controller-functionally irredundant faults, including the system-functionally redundant ones. We see from the curves that under the “piggyback” scheme, final fault coverages are very nearly as high as for the “separate” case. This indicates that the piggyback is successful at pushing the system-functionally redundant faults out into the datapath. Table II summarizes the test circuitry needed for each of the three test schemes in terms of key system parameters. It includes the TPGR, the MISR, and any muxes associated with them. For example, muxes are needed to control whether the data inputs to the datapath are coming from the TPGR (as in test mode) or from the system inputs (as in normal mode). If a single MISR is used to test both the datapath and the controller, muxes are needed to determine which component is driving the MISR at a given time.

Table III shows the relative sizes of systems implementing the three test schemes for our four example circuits. Overhead is given relative to the overhead for the “together” test scheme, since the “together” scheme represents the minimal amount of test circuitry that can be used for any BIST scheme. The “together” test scheme has the lowest area for test circuitry, since no attempt is made to observe extra controller or datapath lines. The drawback, however, is low fault coverage. At the other extreme, the “separate” test scheme has the highest fault coverage and highest observability, but also the highest area overhead, ranging from 10.8% to 23.9% for the examples shown. In the middle is the “piggyback” test scheme. its fault coverage is almost as good as the separate test scheme, but its area overhead is much lower, from 3.8% to 7.6%.

The overhead advantage of the “piggyback” scheme over the “separate” scheme arises from two sources. The first is reduced area requirements for the TPGR compared to the “separate” scheme, due to the fact that under the “separate” scheme, the TPGR must be at least wide enough to generate test bits for all inputs of the datapath, both data and control, at once. In contrast, for the “piggyback” scheme, the control lines are driven from the controller, so the TPGR must be only as wide as the input data, with one extra bit for the start input to the controller.

<i>Design</i>	<i>Together Test Scheme</i>	<i>Piggyback Test Scheme</i>	<i>Relative Overhead</i>	<i>Separate Test Scheme</i>	<i>Relative Overhead</i>
Poly	8186	8684	6.1%	9839	20.2%
Diffeq	9001	9357	4.0%	10136	12.6%
FACET	12966	13460	3.8%	14371	10.8%
WAVE	16022	17250	7.6%	19847	23.9%

Table III. Relative sizes of the systems for the three test schemes, in number of transistors, with overhead figures relative to the size of the system for the “together” test scheme.

<i>Design</i>	<i>Together</i>		<i>Piggyback</i>		<i>Separate</i>	
	<i>TPGR</i>	<i>MISR</i>	<i>TPGR</i>	<i>MISR</i>	<i>TPGR</i>	<i>MISR</i>
Poly	729	819	+0%	+0%	+43%	+0%
Diffeq	611	637	+0%	+0%	+65%	+0%
FACET	729	440	+0%	+50%	+76%	+100%
WAVE	1341	159	+0%	+50%	+65%	+525%

Table IV. Comparing TPGR and MISR overhead for three test schemes, in number of transistors for “together” test scheme and relative to “together” method for the other two test schemes.

There is a similar savings in the number of associated multiplexers. This often results in a significant area savings.

The second source is reduced area requirements for the MISR due to the indirect observation of the control lines through the datapath registers. The “separate” scheme requires an MISR wide enough to watch all outputs of the datapath or all outputs of the controller, whichever is wider, at once. Consider a system in which the controller outputs outnumber the datapath outputs. Our MISR must be at least as big as $r + m + 1$, where r is the number of registers and m is the number of multiplexer select lines. In the “piggyback” scheme, in contrast, we require an MISR wide enough to watch $r + 1$ bits. Thus, we see that the larger m is in comparison to r , the more we will save on area for the MISR. There is a similar savings associated with multiplexers necessary for sharing the same MISR when testing both the datapath and controller. Thus, for circuits in which that number of controller outputs outnumber the number of datapath outputs and m is reasonably large compared to r , we will see a significant reduction in the amount of observation circuitry needed. For other circuits, there may not be a significant reduction, and the amount of observation circuitry may even grow slightly. However, the other stated benefits, like detecting faults that would cause excessive power consumption, will still exist.

As can be seen from the results, our test scheme requires slightly more observation circuitry for two of the circuits: the differential equation solver, for which the number of datapath outputs outnumber the number of controller outputs, so that a savings in the MISR width is not possible; and the FACET example, which has eleven register load lines and only two multiplexer select lines. Here, m is too small relative to r to offset the addition of the logic needed for the finite state machine required for the “piggyback” approach. In general, for many designs the number of multiplexer select lines will outnumber the register load lines, especially when distributed multiplexors, buses with tri-state buffers and one-hot encoding for the

multiplexor control are used. For the other two circuits, use of the “piggyback” scheme does result in a reduction in the amount of observation circuitry.

6. CONCLUSION

This paper proposes a scheme for facilitating testing of datapath / controller pairs. It advocates testing the pair in an integrated way, rather than testing the datapath and controller completely independently by separating each from the system environment during test. The scheme adds a small finite state machine to the system that serves to enhance observability of the controller outputs, so that controller faults can be observed at the outputs of the registers of the datapath. Experimental results show that use of the scheme results in about one-third the test overhead than that required for a scheme in which datapath and controller are tested separately, with fault coverage that is as good, or nearly as good. In addition, for the integrated scheme, the control lines used to communicate between controller and datapath are more thoroughly tested.

References

- ABADIR, M. AND BREUER, M. 1985. Constructing optimal test schedules for VLSI circuits having built-in test hardware. In *Proc. International Symp. on Fault Tolerant Computing* (1985). 165–170.
- AITKEN, R. 1995. Finding defects with fault models. In *Proc. IEEE International Test Conf.* (1995). 498–505.
- ASHAR, P. AND DEVADAS, S. 1991. Optimum and heuristic algorithms for an approach to finite state machine decomposition. *IEEE Trans. Computer-Aided Design* 10, 3 (March), 296–310.
- AT&T. 1993. *User Manuals for GENTEST_S 2.0*. AT&T Bell Laboratories.
- BAKOGLU, H. 1990. *Circuits, Interconnections, and Packaging for VLSI*. Addison-Wesley Publishing Co., Reading, MA.
- BENINI, L. AND DEMICHELI, G. 1994. State assignment for low power dissipation. In *Proc. IEEE Custom Integrated Circuits Conf.* (1994). 136–139.
- BENINI, L., SIEGEL, P., AND DEMICHELI, G. 1994. Automatic synthesis of gated clocks for power reduction in sequential circuits. *IEEE Design and Test of Computers* 11, 4 (Winter), 32–40.
- BHATIA, S. AND JHA, N. 1994. Behavioral synthesis for hierarchical testability of controller/datapath circuits with conditional branches. In *Proc. IEEE International Conf. on Computer Design* (1994). 91–96.
- BREUER, M., GUPTA, R., AND LIEN, J. 1988. Concurrent control of multiple BIT structures. In *Proc. IEEE International Test Conf.* (1988). 431–442.
- BREUER, M. AND LIEN, J. 1988. A test and maintenance controller for a module containing testable chips. In *Proc. IEEE International Test Conf.* (1988). 502–513.
- CARLETTA, J., NOURANI, M., AND PAPACHRISTOU, C. 1999. Synthesis of controllers for full testability of integrated datapath-controller pairs. In *Proc. Design Automation and Test in Europe* (1999). 278–282.
- CARLETTA, J. AND PAPACHRISTOU, C. 1995. Testability analysis and insertion for RTL circuits based on pseudorandom BIST. In *Proc. IEEE Intl. Conf. on Computer Design* (1995). 162–167.
- CARLETTA, J. AND PAPACHRISTOU, C. 1997. Behavioral testability insertion for datapath/controller circuits. *Journal of Electronic Testing: Theory and Applications* 11, 1 (August), 9–28.
- COMPASS DESIGN AUTOMATION. 1993. *User Manuals for COMPASS VLSI V8R4.4*. Compass Design Automation, Inc.

- DEMICHELI, G., BRAYTON, R., AND SANGIOVANNI-VINCENTELLI, A. 1984. KISS: A program for optimal state assignment of finite state machines. In *Proc. ACM/IEEE International Conf. on Computer-Aided Design* (1984). 209–211.
- DEVADAS, S., MA, H., NEWTON, A., AND SANGIOVANNI-VINCENTELLI, A. 1988. MUSTANG: State assignment of finite state machines targeting multilevel logic implementations. *IEEE Trans. Computer-Aided Design* 7, 12 (December), 1290–1300.
- DEVADAS, S. AND NEWTON, A. 1989. Decomposition and factorization of sequential finite state machines. *IEEE Trans. Computer-Aided Design* 8, 11 (November), 1206–1217.
- DEY, S., GANGARAM, V., AND POTKONJAK, M. 1995. A controller-based design-for-testability technique for controller-datapath circuits. In *Proc. IEEE/ACM International Conf. on Computer-Aided Design* (1995). 534–540.
- ESCHERMANN, B. AND WUNDERLICH, H. 1990. Optimized synthesis of self-testable finite state machines. In *Proc. International Symp. on Fault Tolerant Computing* (1990). 390–397.
- FUMMI, F., SCIUTO, D., AND SERRA, M. 1995. Synthesis for testability of large complexity controllers. In *Proc. IEEE International Conf. on Computer Design* (1995). 180–185.
- GAJSKI, D., DUTT, N., WU, A., AND LIN, S. 1992. *High-Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, Boston, MA.
- HARMANANI, H., PAPACHRISTOU, C., CARLETTA, J., AND NOURANI, M. 1994. A method for testability insertion at the RTL – behavioral and structural. In *Proc. First International Test Synthesis Workshop* (1994). 45–49.
- HARMANANI, H., PAPACHRISTOU, C., CHIU, S., AND NOURANI, M. 1992. SYNTTEST: An environment for system-level design for test. In *Proc. ACM Conf. European Design Automation (EURO-DAC 92)* (1992). 402–407.
- HELLEBRAND, S. AND WUNDERLICH, H. 1994. An efficient procedure for the synthesis of fast self-testable controller structure. In *Proc. ACM/IEEE International Conf. on Computer-Aided Design* (1994). 110–116.
- HSU, F. AND PATEL, J. 1995. A distance reduction approach to design for testability. In *Proc. IEEE VLSI Test Symposium* (1995). 158–163.
- JOERSZ, R. AND KIME, C. 1987. A distributed hardware approach to built-in self test. In *Proc. IEEE International Test Conf.* (1987). 972–980.
- JONE, W., PAPACHRISTOU, C., AND PEREIRA, M. 1989. A scheme for overlaying concurrent testing of VLSI circuits. In *Proc. ACM/IEEE Design Automation Conf.* (1989). 531–536.
- KIME, C. AND SALUJA, K. 1982. Test scheduling in testable VLSI circuits. In *Proc. International Symp. on Fault Tolerant Computing* (1982). 406–412.
- KUNG, S., WHITEHOUSE, H., AND KAILATH, T. 1985. *VLSI and Modern Signal Processing*. Prentice-Hall, Englewood Cliffs, NJ.
- KUO, T., LIU, C., AND SALUJA, K. 1995. An optimized testable architecture for finite state machine. In *Proc. IEEE VLSI Test Symposium* (1995). 164–169.
- LAGNESE, E. AND THOMAS, D. 1989. Architectural partitioning for system level design. In *Proc. ACM/IEEE Design Automation Conf.* (1989). 62–67.
- LANDMAN, P. AND RABAAY, J. 1995. Activity-sensitive architectural power analysis for the control path. In *Proc. ACM International Symposium on Low Power Design* (1995). 93–98.
- MUKHEREJEE, D., NJINDA, C., AND BREUER, M. 1991. Synthesis of optimal 1-hot coded on-chip controllers for BIST hardware. In *Proc. ACM/IEEE International Conf. on Computer-Aided Design* (1991). 236–239.
- NOURANI, M., CARLETTA, J., AND PAPACHRISTOU, C. 1997. A scheme for integrated controller-datapath fault testing. In *Proc. ACM/IEEE Design Automation Conf.* (1997). 546–551.
- SPARMANN, S., LUZENBURGER, D., CHENG, K., AND REDDY, S. 1995. Fast identification of robust dependent path delay faults. In *Proc. ACM/IEEE Design Automation Conf.* (1995). 119–125.
- VLSI TECHNOLOGY. 1993. *0.8-Micron CMOS VSC450 Portable Library*. VLSI Technology, Inc.