Software-Based Self-Testing Methodology for Processor Cores

Li Chen, *Student Member, IEEE,* and Sujit Dey, *Member, IEEE*

***Abstract*—At-speed testing of gigahertz processors using external testers may not be technically and economically feasible. Hence, there is an emerging need for low-cost high-quality self-test methodologies that can be used by processors to test themselves at-speed. Currently, built-in self-test (BIST) is the primary self-test methodology available. While memory BIST is commonly used for testing embedded memory cores, complex logic designs such as microprocessors are rarely tested with logic BIST. In this paper, we first analyze the issues associated with current hardware-based logic-BIST methodologies by applying a commercial logic-BIST tool to two processor cores. We then propose a new software-based self-testing methodology for processors, which uses a software tester embedded in the processor memory as a vehicle for applying structural tests. The software tester consists of programs for test generation and test application. Prior to the test, structural tests are prepared for processor components in the form of self-test signatures. During the process of self-test, the test generation program expands the self-test signatures into test sets and the test application program applies the tests to the components under test at the speed of the processor. Application of the novel software-based self-test method demonstrates its significant cost/fault coverage benefits and its ability to apply at-speed test while alleviating the need for high-speed testers.**

***Index Terms*—At-speed testing, instructions, processors, self-test, structural test.**

I. Introduction

AS THE SPEED of microprocessors approaches the gigahertz range, at-speed testing is becoming increasingly critical. However, testers with speed matching the speed of gigahertz processors will be increasingly costly. According to [1], if the current testing techniques are to be continued, the test equipment cost can rise toward $20 million. Moreover, due to the inherent inaccuracy of testers, at-speed testing of high-speed processors may result in an unacceptably high yield loss of 48% by 2012. To ensure the economic viability of the industry to manufacture high-performance processors, alternative testing techniques are needed. Hence, the recent focus on self-testing—the ability of a circuit to test itself. By generating the required test patterns on-chip and applying the tests at the speed of the circuit, a gigahertz processor can test itself without relying on prohibitively expensive high-speed external testers.

Manuscript received May 30, 2000; revised September 3, 2000. This work is supported by the MARCO/DARPA Gigascale Silicon Research Center (GSRC). This paper was presented in part at the 18th IEEE VLSI Test Symposium, Motreal, QB, Canada, April 2000, and the 37th Design Automation Conference Los Angeles, CA, June 2000. This paper was recommended by Associate Editor R. Karri.

L. Chen and S. Dey are with the Electrical and Computer Engineering Department, University of California, San Diego, CA 92037 USA

Publisher Item Identifier S 0278-0070(01)01506-8.

One of the most widely researched self-testing techniques is built-in self-test (BIST) [2], which uses embedded hardware test generators and test response analyzers to generate and apply test patterns on-chip at the speed of the circuit, thereby eliminating the need for an external tester. Memory BIST has been commonly used for testing embedded memory components, as it performs well due to the deterministic nature of memory tests facilitated by the regular structure of memory components. Logic BIST, however, faces many challenges because it relies on the generation and application of pseudorandom test patterns.

1) For random-pattern-resistant circuits, the fault coverage achieved by pseudorandom testing may be low.
2) The insertion of the BIST circuitry used for generating and applying pseudorandom patterns may result in significant area and performance overhead.
3) The application of random test patterns often results in excessive power consumption in the BIST mode [3], possibly damaging the circuit under test.
4) The application of random test patterns may drive the circuit under test into nonfunctional mode in which the free flow of test data can be impeded by problems such as bus contentions. Thus, the circuit under test is required to be BIST-ready [4] (e.g., immune to problems such as bus contentions even when pseudorandom test patterns are applied).

To address the above issues, many solutions have been proposed. The low fault coverage due to random-pattern resistance may be improved with techniques such as deterministic BIST [5], [6] or weighted random patterns [7]–[9]. The BIST overhead may be reduced by generating pseudorandom test patterns using existing circuits such as accumulators [6], [10], [11], embedded processors [5], or any sequential circuits in general [12]. In scan-based BIST, the test overhead may also be reduced by partial scan [13]. The excessive power consumption in the BIST mode may be reduced by techniques such as test scheduling [3], reducing input activities [14], or filtering nondetecting vectors [15]. BIST readiness may be achieved by design changes [4], [16], [17].

The feasibility of logic BIST on industrial circuits has been demonstrated in [4] and [17] in which different approaches were used to overcome the random-pattern resistance of complex circuits. In particular, [4] uses test points, which come with the price of area overhead and possible performance degradation, and [17] uses deterministic BIST, which encodes deterministic test patterns in pseudorandom test patterns using additional on-chip hardware. Both approaches have been shown to be

effective on a number of industrial circuits, provided that these circuits are made BIST-ready beforehand.

While logic BIST may perform well on industrial application specfied integrated circuits (ASICs), its feasibility on microprocessors is yet to be investigated. First, the design changes needed for making a microprocessor BIST-ready may come with unacceptable cost, such as substantial manual effort and significant performance degradation. In addition, microprocessors are especially random-pattern resistant. Due to the timing-critical nature of microprocessors, test points may not be acceptable as a solution to this problem, as they could introduce performance degradation on critical paths. Deterministic BIST, on the other hand, may lead to unacceptable area overhead, as the size of the on-chip hardware for encoding deterministic test patterns depends on the testability of the circuit [17].

An alternative to hardware-based self-testing techniques such as BIST is software-based self-testing, which involves the testing of microprocessors using processor instructions. Whereas hardware-based self-test must be applied in the nonfunctional BIST mode, software-based self-test can be applied in the normal operational mode of the processor without requiring any design changes or the insertion of any additional hardware structures.

Software-based testing has a long history as an ad hoc technique for testing processors. Computer systems are regularly equipped with software programs to perform in-field testing. The tests done are typically used for checking the functionality of the system, but not for detecting manufacturing defects. Functional validation suites have been used regularly to perform manufacturing testing of processors [18]. However, its application relies on external testers and its results in terms of manufacturing fault coverage are low, as the validation suites are not targeted at structural faults. To reach a desirable fault coverage, functional validation suites are often supplemented with deterministic structural tests or additional handcrafted tests.

Recently, researchers have started investigating functional tests specifically designed for manufacturing testing [19]–[21]. Some propose to apply the tests with external testers [19], others allow the processors to test themselves with self-test programs [20], [21]. A common characteristic of these approaches is to apply randomized instructions to the processor under test. However, although processors are more amenable to random instruction tests than to random-pattern tests, it is difficult to target structural faults by applying random instructions at the processor level.

In this paper, we first analyze the strengths and limitations of current hardware-based self-testing techniques by applying a commercial logic-BIST tool to a simple processor core as well as a complex commercial processor core. Specifically, we demonstrate the design changes needed for making these processors BIST-ready. The design changes require substantial manual effort and can lead to unacceptable performance degradation. The need for design changes can be elevated in the case of processors, as processors are random-pattern resistant due to their complex controls.

Since the need for self-testing is most acute for high-performance processors, we propose a new software-based self-testing methodology for processors that uses a software tester embedded in the processor memory as a vehicle for applying structural tests. The software tester consists of a program for generating pseudorandom test patterns (test pattern generation program) and a program for applying these patterns (test pattern application program). Since software has the advantage of programmability and flexibility, the test pattern generation program can be used to generate desirable random test sets on-chip without any hardware overhead. In addition, the test pattern application program enables on-chip test application by guiding the test patterns through the complex control structure of the processor rather than with the help of scan chains and boundary-scan chains as is done in the case of hardware-based logic BIST.

To circumvent the low fault coverage associated with random-pattern testing of processors, our approach first determines the structural test needs of the components in the processor, which are usually much less complex than the full processor and, hence, much more amenable to random-pattern testing. At the processor level, the instructions of the processor are used to apply the tests to each component at-speed. Since the instructions satisfy the complex control flow of the processor, the flow of test data to/from the component under test will not be impeded, as in the case of hardware BIST applying random patterns to the entire processor.

The proposed approach is *fundamentally different* from any previous approaches that apply functional tests using randomized instructions [19]–[21]. Unlike previous approaches, it aims at structural faults from the very beginning by preparing structural tests for the *components* in the processor. Moreover, the instructions in the software tester are not randomly chosen, but carefully crafted in order to deliver the previously prepared structural tests to the desired components.

In Section II, we first analyze the issues associated with current hardware-based logic BIST by applying a commercial logic-BIST tool to two processor cores. We will then describe the proposed software-based self-testing methodology in Section III. The performance of the software-based approach is evaluated and compared to that of the logic-BIST tool.

II. Motivation—Experiences with Applying Logic BIST to Processors

As we have discussed before, at-speed testing of gigahertz processors can be enabled by self-test. Current hardware-based logic-BIST techniques are based on the application of pseudorandom test patterns. Although the low fault coverage due to the random-pattern resistance of the circuit under test and the area overhead introduced by the BIST circuitry can be improved by various techniques, it has been pointed out by many that the application of logic BIST requires the circuit under test to be BIST-ready [4], [16], [17]. For designs such as cell-based ASICs, logic BIST may be able to achieve high fault coverage with a reasonable amount design changes [4]. However, for complex designs like microprocessors, the amount of design changes and the resulting area and performance overhead might not be acceptable. This is because: 1) such designs *intentionally* incorporate internal tristates and bidirectionals such as partially decoded muxes and pass gates, which cannot be easily handled

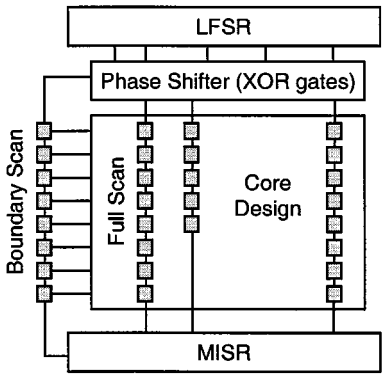


Fig. 1. BIST circuitry inserted by the logic-BIST tool.

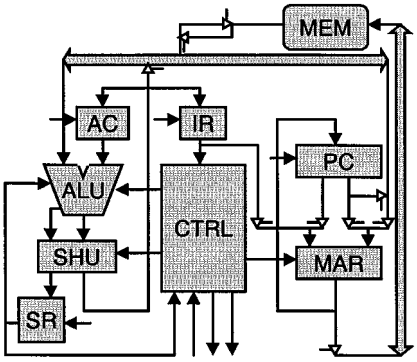


Fig. 2. PARWAN processor core.

by logic BIST [22]; 2) the timing-critical nature of these designs often requires nonfull-scan or sequential test techniques, whereas logic BIST typically builds upon full-scanned designs [23]; and 3) the test logic insertion performed by commercial tools is done at gate level, whereas these designs' netlists are at the transistor level.

As a motivation for the proposed software-based self-testing methodology, we describe our experiences of applying a commercially available logic-BIST tool to two processor cores: PARWAN and picoJava-II. We present the results in terms of fault coverage and area overhead as well as the design changes required for making the processors BIST-ready.

As shown in Fig. 1, the logic-BIST tool uses a STUMPS-based BIST architecture [23]. Instead of relying on an external tester for applying tests to the scan chains, the logic-BIST tool generates test vectors on-chip using a linear feedback shift register (LFSR). The outputs of the LFSR are connected to the scan chains through a phase shifter composed of XOR gates, which is designed to reduce the linear correlation among scan chains. The outputs of the scan chains are compressed on-chip using a multiple-input shift register (MISR). The user may choose to insert a certain number of test points to improve the fault coverage.

We first applied the logic-BIST tool to a simple accumulator-based microprocessor named PARWAN [24] (Fig. 2). It includes the following components: arithmetic logic unit (ALU), accumulator (AC) unit , controller (CTRL), instruction register (IR) unit, program counter (PC) unit, memory address register (MAR) unit , shifter unit (SHU), and status register (SR) unit.

TABLE I
PARWAN: FULL SCAN VERSUS LOGIC BIST

	Area [gate count]	# Test patterns	Fault Coverage
Original. ckt	888	--	--
Modified ckt	812	--	--
Full Scan*	909	640	89.39%
Logic BIST*	2185	32767	88.69%
Logic BIST**	2246	32767	97.34%

*On the modified circuit.

**On the modified circuit with test points.

The synthesized version of PARWAN contains 888 equivalent NAND gates and 53 flip flops. The data bus is 8-b wide, shared by both `data_in` and `data_out`. The address bus is 12-b wide. Accesses to both buses are controlled by tristate buffers.

Several design changes were needed for making PARWAN BIST-ready. As the MISR signatures can be corrupted by undefined values, logic-BIST tools do not accept circuits with possible bus-contention problems [4]. Moreover, logic BIST requires the circuit under test to be free of bidirectional input–output (I/O) pins [22]. Hence, before applying the logic-BIST tool, we manually modified the circuit description of PARWAN. The modifications include: 1) splitting all bidirectional pins into separate I/O pins; and 2) replacing all tristate buffers with selectors. In addition, we insert test points to improve the testability of the circuit.

Table I shows the results of the logic-BIST tool on PARWAN in comparison with the full-scan results. The rows contain the following information:

1) statistics on the original PARWAN circuit;
2) statistics on the modified PARWAN circuit;
3) the results of full scan on the modified circuit;
4) the results of applying the logic-BIST tool to the modified circuit;
5) the results of the logic-BIST tool on the modified circuit with test points (3 control points and 11 observation points).

For the two experiments with the logic-BIST tool, we divided the 53 flip flops of PARWAN into five scan chains. The logic-BIST tool automatically chooses an 18-b LFSR as the pattern generator. The columns contain the following information: the areas of the circuits in terms of the number of equivalent NAND gates, the number of test patterns applied during each test, and the final fault coverage on collapsed faults. Notices that the areas reported do not include routing area.

As shown in Table I, without the help of additional test points, the logic-BIST tool is able to achieve a fault coverage comparable to that of full scan. Due to the insertion of the BIST circuitry, logic BIST requires a larger area overhead than full scan. Since the size of the BIST circuitry stays relatively constant, the relative area overhead is expected to diminish quickly as the design size increases [4]. Note that the fault coverage achieved by logic BIST can be significantly enhanced by the use of test points.

In addition to the PARWAN processor, we have applied the logic-BIST tool to picoJava-II (Fig. 3), a soft microprocessor core [25].

TABLE II
PICOJAVA-II: FULL SCAN VERSUS LOGIC BIST

	LFSR size	MISR size	# Test points		Area overhead	# Test patterns	Fault coverage
			Control	Observe			
Full Scan	--	--	--	--	11.13%	12,736	95.54%
Logic BIST - 1	24	41	0	0	13.06%	32,767	58.81%
Logic BIST - 2	24	41	100	100	13.29%	32,767	82.53%
Logic BIST - 3	32	41	100	100	13.30%	32,767	82.93%
Logic BIST - 4	24	41	100	100	13.30%	1,000,000	84.11%

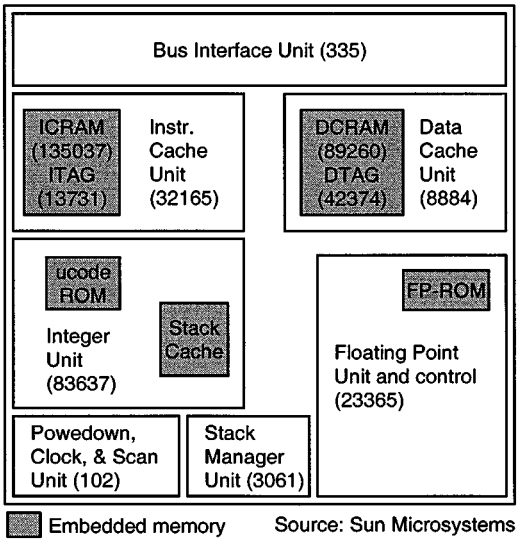


Fig. 3. PicoJava-II processor core.

The picoJava-II processor core is a hardware implementation of the Java Virtual Machine. It is a stack-based 32-b processor with 300 instructions. It contains six pipeline stages and can execute up to four instructions in one cycle. As a soft core, it is distributed in the form of synthesizable Verilog description, including seven technology-dependent embedded memory blocks. We have synthesized it down to the gate level using a commercial logic-synthesis tool. The synthesized picoJava-II core contains 167 I/O ports and 6801 flip flops. Fig. 3 shows the component areas in terms of the number of equivalent NAND gates. The total area is 127 887 for the logic components and 313 989 for the embedded memory components. The total number of faults in the logic blocks is 532 527. Using a commercial fault simulator, it takes 20 s to simulate one test cycle if all faults are included in the fault list.

Table II shows the results of applying the logic-BIST tool to the logic part of the picoJava processor in comparison with the results of full scan. The values of area overhead and fault coverage reported in the table are with respect to the logic part of the processor core.

As shown in Table II, we have conducted four experiments with the logic-BIST tool in order to study the effect of different parameters on the fault coverage. Given a fixed LFSR/MISR configuration, the logic-BIST tool allows the user to set the number of test points. Our experimental results show that the insertion of test points leads to a significant improvement in fault coverage. The fault coverage can be improved further by increasing the size of the LFSR and the number of random patterns. However, the improvement is marginal.

The design changes required for making picoJava BIST-ready are as follows. First, embedded memories were bypassed with scan flip flops in the test mode [4] as otherwise they could become sources of undefined values (X generators), leading to the corruption of MISR signatures. In addition, a number of combinational loops, which did not exist in the functional mode, were formed when random test patterns were applied. The signals in a combinational loop may toggle when the loop is activated, causing the generation of undefined values. The combinational loops can be broken with the help of control points. The breaking of the combinational loops, as well as the insertion of the memory bypass circuits, had to be performed manually.

We would like to point out that the logic-BIST tool we had used does not necessarily incorporate all recently conducted research works in hardware-based logic BIST. The fault coverage and the area overhead can possibly be improved with the application of recent research results in BIST, such as deterministic BIST [5], [6], weighted random patterns [7]–[9], and the reuse of existing circuits [5], [6], [10]–[12]. However, a common problem remains with all logic-BIST techniques: the circuit under test have to be made BIST-ready. For complex high-end microprocessors, this requires substantial manual design effort, significantly delaying the product's time to market. In addition, the design changes may induce intolerable performance degradation.

In summary, although it is an attractive solution to the problem of at-speed test, the application of a logic-BIST tool on the two processor cores shows its potential disadvantages. Due to their complex control structures, processors are highly random-pattern resistant. An acceptable fault coverage cannot be achieved by simply applying random test patterns to the entire processor, as certain internal control signals need to be set properly to ensure the free flow of test data. Through our experiences described above as well as through experiences documented in [4] and [16], extensive design changes may often be needed to make the processors random-pattern testable, adding to the area/delay overhead. In addition, certain violations that do not happen in the functional mode such as bus-contentions and the forming of combinational loops could occur during the application of random test patterns. To avoid these violations, additional design changes need to be made, making the insertion of logic BIST even more difficult. Due to the diversity of the designs, not all design changes can be automated with the existing commercial logic-BIST tools. Thus, many of these design changes have to be carried out

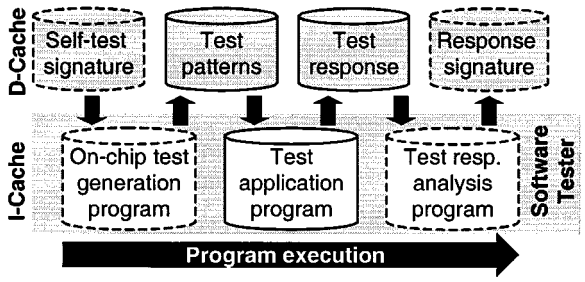


Fig. 4. Self-test methodology.

manually, significantly adding to the design time. In addition, the design changes may cause performance degradation, making it necessary to redesign in some cases.

III. Software-Based Self-Test Methodology Targeting Structural Faults

Unlike hardware-based self-testing, software-based testing is nonintrusive since it applies tests in the normal operational mode of the circuit. Because of the programmability of software, software-based testing enables the use of configurable random-pattern generation programs without requiring any test overhead. Moreover, software instructions have the ability of guiding test patterns through a complex processor, avoiding the blockage of the test data due to nonfunctional control signals as in the case of hardware-based logic BIST.

Given the advantage of software-based testing, we propose a novel software-based processor self-testing methodology that delivers *structural* tests to the components of the processor using processor instructions. Our self-testing scheme includes two steps. The test preparation step includes the generation of realizable structural tests for components of the processor and the encapsulation of component tests into self-test signatures. The self-testing step involves the application of the component tests using a software tester, which consists of an on-chip test pattern generation program, a test pattern application program, and a test response analysis program, as shown in Fig. 4. The self-test signatures and the programs contained in the software tester can be loaded to the processor memory with a low-speed tester prior to the application of the test. During the application of the tests, the on-chip test generation program emulates a pseudorandom pattern generator and expands the self-test signatures into test patterns. The test patterns are applied to components by the on-chip test application program at the speed of the processor. The test application program also collects the test responses and saves them to memory. If desired, the test responses can be compressed into response signatures using the test response analysis program. The responses are stored into the processor memory and can later be unloaded and analyzed by an external tester. Note that we assume the processor memory has been tested with standard techniques such as memory BIST before the application of the test. Thus, the memory is assumed to be fault free.

By targeting the structural test need of less complex components, the proposed method has the fault coverage advantage of deterministic structural testing. Since component test application and response collection are done with instructions instead of with scan chains, it requires no area or performance overhead and the test application is performed at-speed. Most importantly, by shifting the role of external testers from applying tests to loading test programs and unloading responses, it enables at-speed testing of gigahertz processors with low-speed testers.

In the following sections, we describe the above two steps in detail using the PARWAN processor (Fig. 2).

A. Component Test Preparation

During the component test preparation step, we develop structural tests for individual components of the processor, such as the ALU, SHU, and PC. Component-level fault simulation is used for evaluating the preliminary fault coverage of these tests.

Component tests can either be stored or generated on-chip, depending on which method is more efficient for a particular case. If tests are to be generated on-chip, we characterize the test need of the component by a *self-test signature*, which includes the seed S and the configuration C of a pseudorandom number generator as well as the number of test patterns to be generated N. The self-test signatures can be expanded on-chip into test sets using a pseudorandom number generation program. Multiple self-test signatures may be used for one component if necessary. Thus, our self-test methodology allows the incorporation of any deterministic BIST techniques that encode a deterministic test set as several pseudorandom test sets [5], [6]. A low-speed tester can be used to load the self-test signatures or the predetermined tests to the processor memory prior to the application of test.

One of the challenges of component test preparation lies in the generation of *realizable* component tests. That is, the component tests must be deliverable with the software tester. Since the delivery of component tests relies on processor instructions, it is impossible to deliver some test patterns. To avoid producing undeliverable test patterns, component tests must be generated under the constraints imposed by the processor instruction set. Note that the inability to apply all possible input patterns to a component does not necessarily map to a low fault coverage as it is possible to detect all faults by choosing test patterns from the constrained input space. If, however, a fault can only be detected by test patterns outside the allowed input space, by definition the fault is redundant in the normal operational mode of the processor. Thus, there is no need to test for this fault.

In the subsequent sections, we first define the input and output constraints imposed by the instruction set. We then provide methods for modeling these constraints during test generation. In Section III-A3, we propose an iterative method for preparing component tests under the instruction-imposed constraints. An example on the results of the component test preparation step is shown in Section III-A4.

1) Instruction-Imposed Constraints: The constraints imposed by the processor instruction set can be divided into input constraints and output constraints, which are determined by the instructions for controlling the component inputs and the instructions for observing the component outputs.

The input constraints define the input space of the component allowed by instructions. For a component C, let I_c be the set of instructions for controlling the inputs of C, i be an instruction

TABLE III
SHU: SPATIAL CONSTRAINTS IMPOSED BY INSTRUCTIONS

Instr	asl	asr	v	c	z	n	s (data_in[7])	data_in[6:0]
lda	0	0	0	0	1 if	n = s	x	x
and	0	0	0	0	data_in =		x	x
add	0	0	v = c ⊕ s	x	0000 0000		x	x
sub	0	0	v = c ⊕ s	x			x	x
asl	1	0	0	0			x	x
asr	0	1	0	0			x	x

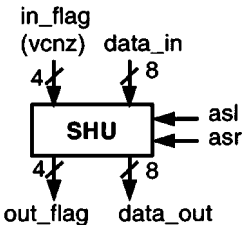


Fig. 5. SHU.

such that $i \in I_c$. $V_{i,c}$ is the set of input vectors to component C allowed by instruction i. The input space of C allowed by I_c is therefore $V_c = \bigcup_{i \in I_c} V_{i,c}$. Let T_c be a test set for C. T_c is said to be *realizable* by the instructions in I_c iff $T_c \subseteq V_c$. On the other hand, a fault f is said to be *undetectable* in the functional mode allowed by I_c if it can only be detected by a set of test vectors T such that $T \subseteq \bar{V}_c$.

The output constraints define the subset of component outputs observable by instructions. They are used in the component-level fault simulation to determine the preliminary fault coverage of the component tests. Since the component test responses are to be collected by instructions, a fault is undetected at the chip level if its resulting errors fail to propagate to any observable outputs. Therefore, during component-level fault simulation, errors propagating to unobservable outputs should not be accounted for toward the detection of any fault. One way to enforce this is to remove all unobservable outputs from the output list during fault simulation.

We will next use one component of the PARWAN processor, SHU, to illustrate the types of constraints imposed by the instruction set.

A block diagram of SHU is shown in Fig. 5. The input signals include `data_in`, `in_flag`, and the shifting signals from the controller. Signal `in_flag` includes four bits: `v`, `c`, `z`, and `n`, which denote overflow, carry, zero, and negative, respectively. The shifting signals includes two bits, `asl` and `asr`, which denote arithmetic shift left and arithmetic shift right.

The constraints imposed by the processor instruction set can be divided into two types. We define constraints which can be specified in a single time frame as *spatial constraints* and constraints spanning over several time frames as *temporal constraints*.

For SHU, the instructions for controlling the inputs include `lda` (load accumulator), `and`, `add`, `sub`, `asl` (arithmetic shift left), and `asr` (arithmetic shift right). The spatial constraints imposed by these instructions are shown in Table III, where `s` is the sign bit of `data_in` and x denotes "don't-care."

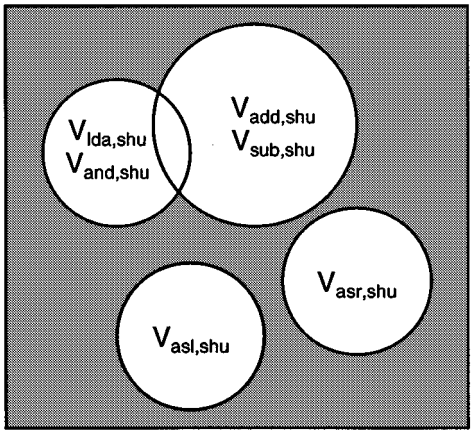


Fig. 6. Input space of SHU allowed by its spatial constraints.

As shown in Table III, `asl` and `asr` cannot both be one. For arithmetic instructions, `v`, `c`, and `s` must obey the relation of $\mathtt{v} = \mathtt{c} \oplus \mathtt{s}$. For other instructions, `v` and `c` are zeros. For all instructions, `z` and `n` must be consistent with the value of `data_in`. In addition to the constraints shown in Table III `s` and `data_in [6:0]` must obey another constraint so that `data_in [7:0]` can never be 10 000 000. In summary, the spatial constraints on SHU can be expressed by the following set of Boolean equations:

1) $\mathtt{z} = \mathtt{!(data\_in[7]} + \mathtt{data\_in[6]} + \cdots + \mathtt{data\_in[0])}$, and
2) $\mathtt{n} = \mathtt{data\_in[7]}$, and
3) $\mathtt{!asl{*}!asr{*}(!v{*}!c} + \mathtt{!(v} \oplus \mathtt{c} \oplus \mathtt{s))} + (\mathtt{asl}{*}\mathtt{!asr} + \mathtt{!asl} * \mathtt{asr}){*}\mathtt{!v{*}!c} = 1$, and
4) $\mathtt{data\_in[7]} * \mathtt{!(data\_in[6]} + \cdots + \mathtt{data\_in[0])} = 0$.

The input space of SHU defined by its spatial constraints can be illustrated by the Venn diagram shown in Fig. 6, where the allowed input space is shown in the white area.

The temporal constraints on SHU are imposed by the sequence of instructions that apply tests to SHU. The sequence includes three steps: 1) loading data to be shifted into the AC; 2) shifting data stored in AC and store the shift result temporarily in AC; and 3) storing the shift result into memory for later analysis. As shown in Fig. 7, the application of one test pattern involves three passes through the SHU. To account for fault aliasing, temporal constraints need to be modeled during component test generation.

Previously, Tupuri *et al.* and Vishakantaiah *et al.* have proposed a methodology to systematically extract structural con-

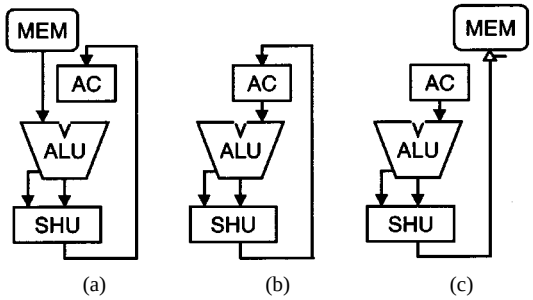


Fig. 7. Hardware paths involved in testing the SHU. (a) Loading to AC. (b) Shifting. (c) Storing to memory (MEM).

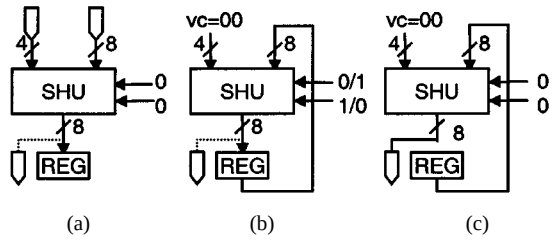


Fig. 8. Circuit for modeling temporal constraints on SHU. (a) Loading to AC. (b) Shifting. (c) Storing to MEM.

straints for components of a processor from the processor description in hardware description language (HDL) [26]–[28]. However, it should be noted that not all architectural constraints can be extracted from structural descriptions [29]. Thus, in the future, we will investigate on enhancing existing structural constraint extraction methods to extract constraints imposed by the instruction set, as is required in our work.

The output constraints for SHU define the list of outputs observable by instructions. The data output of SHU `data_out` can be directly observed by the store instruction. The status output `out_flag` can be indirectly observed by branch instructions. Therefore, for this particular example, there are no output constraints and no outputs need to be removed from the output list during the component-level fault simulation.

2) Constraint Modeling: Having described the constraints imposed by the processor instruction set, we will now describe the modeling of these constraints during component test preparation.

If component tests are generated by automatic test pattern generation (ATPG) , spatial constraints can be specified during test generation with the aid of the ATPG tool. As an alternative, spatial constraint can be specified with a virtual constraint circuit proposed in [26].

If random tests are used for components, random patterns can only be used on independent inputs. In the case of SHU, these would be `data_in` and `c`. Inputs such as `z`, `n`, and `v` can be derived from these inputs. It is inconvenient to assign random patterns to instruction-related signals, such as the shifting signals. Therefore, they are fixed when random patterns are applied to other inputs. The fixed value of the instruction-related signals may be changed if necessary.

The temporal constraints of SHU can be modeled using the three-phase sequential circuit shown in Fig. 8. The three phases correspond to the three instructions for applying tests to SHU, which are loading data into AC, shifting, and storing AC content to memory. Notice that the data inputs and flag inputs of SHU are only connected to the primary inputs in the first phase when the AC content is loaded from the memory. The data outputs of SHU are only connected to the primary outputs in the third phase when the test response is stored to memory. The shifting signals in these two phases are set to zeros. The `v` and `c` flags are set to zeros in the second and the third steps since neither the shift instructions nor the store instruction can set them to one. At any phase, the inputs to SHU must also obey the spatial constraints we have described before.

As described in Section III-A1, even the spatial constraints alone can be complicated for a component as simple as a 1-b shifter, which are only controlled by six instructions. For a more complex component that can be controlled by many more instructions, the constraints can be much more complicated, drastically increasing the complexity of constrained test generation. Based on the fact that input constraints are simpler for a single instruction than for a large number of instructions, in the following we propose an iterative method for generating tests under the constraints imposed by a set of instructions.

3) Method for Preparing Component Tests Under Constraints: Our method for preparing component tests under the constraints imposed by a set of instructions is as follows. For each instruction that can be used to control the component, we perform constrained test generation within the input space allowed by this instruction. We repeat this process on other instructions, until 1) we have exhausted all instructions or 2) we have successfully generated tests for all nonredundant faults.

Fig. 9 shows a flowchart illustrating the proposed method, where C is the component under test. In the initialization step, the set of instructions to be processed (I) is initialized to the set of instructions for controlling $C(I_c)$. The list of undetected faults (F) is initialized to the fault list of $C(F_c)$. V_c and T_c, which denote the previously covered input space and the previously generated test set, are both initialized to $\oslash$. During each iteration of the test generation process, an instruction i is chosen from the set of unprocessed instructions (I). The input space allowed by this instruction $V_{i,c}$ is compared with the previously covered input space V_c. If $V_{i,c} \subseteq V_c$, the instruction i is skipped, as the inclusion of this instruction does not expand the input space for component C. Otherwise, we perform constrained test generation for the list of undetected faults (F) under the constraints imposed by instruction i. The resulting test set and the list of newly detected faults are used to update T_c and F. In addition, V_c is updated to include the newly covered input space and i is removed from the list of unprocessed instructions (I). We repeat this process until either I or F becomes empty.

The resulting test set T_c has the following two properties.

Property 1: Assuming the tests generated under the constraints imposed by any *single* instruction i achieve the maximum possible fault coverage in the functional mode allowed by i, T_c can achieve the maximum possible fault coverage in the functional mode allowed by I_c. Thus, T_c detects any faults detectable by V_c.

Property 2: Any test vector in T_c can be realized by at least one instruction in I_c.

Formal proofs for the two properties can be found in the Appendix.

TABLE IV
COMPONENT TESTS FOR THE ALU

Instruction	Test patterns		
	alu_code	in1	in2
lda	100	(11111111, 01100011, 82)	ZZZZZZZZ
sta	110	ZZZZZZZZ	(11111111, 01100011, 82)
cma	001	ZZZZZZZZ	(11111111, 01100011, 35)
and	000	(11111111, 01100011, 98): odd	(11111111, 01100011, 98): even
sub	111	(11111111, 01100011, 24): odd	(11111111, 01100011, 24): even
add	101	(11111111, 01100011, 26): odd	(11111111, 01100011, 26): even

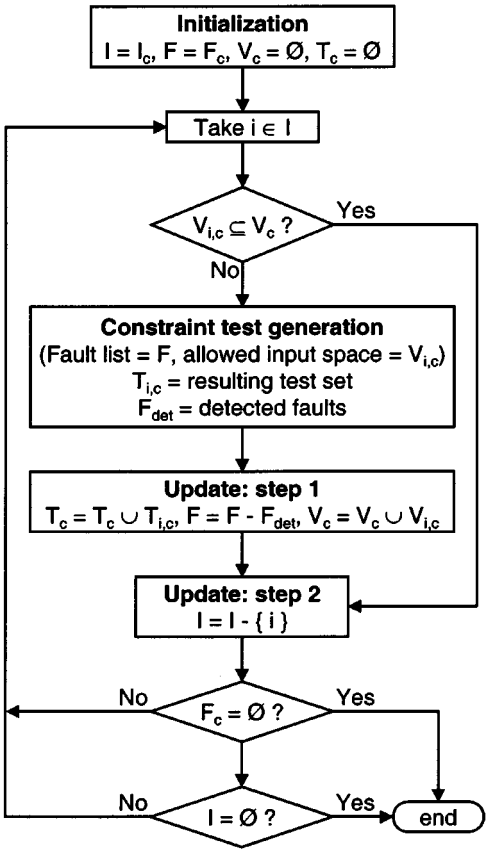


Fig. 9. Iterative method for generating tests under the constraints imposed by a set of instructions.

We do not propose any method for selecting I_c in this paper. However, in practice, the selection of I_c can be performed by circuit simulation, during which it is possible to identify which instructions cause the inputs of a component to change. In addition, to reduce the complexity of the constrained test generation step, the instructions in I_c can be prioritized by the simplicity of instruction-imposed constraints with the higher priorities assigned to the instructions with the simpler constraints. In case an acceptable fault coverage is achieved with the first few instructions in I_c, there is no need the move on to instructions with overly complex constraints.

As shown in Fig. 9, in the beginning of each iteration, we screen out instructions that cannot bring in new input space by checking whether $V_{i,c} \subseteq V_c$. This problem is co-NP- complete. Since it is used for reducing the number of instructions to be processed, the screening step is not mandatory. The requirement of this step can be relaxed to screen out only the instructions that obviously cover the same input space as any previously processed instruction. (e.g., in Table III, we can easily identify that `lda` and `and` cover the exact same input space.) This compromise substantially lowers the complexity of the screening step.

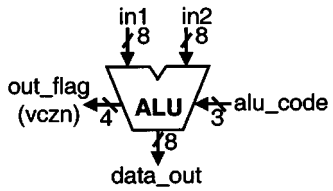


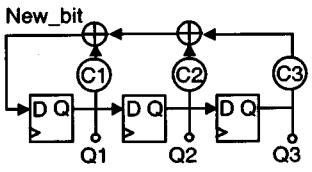
Fig. 10. ALU.

4) Example on the Results of the Component Test Preparation Step: As we have mentioned in the beginning of Section III, the results of the component test preparation step can either be expressed in the form of actual test patterns or in the form of self-test signatures. The self-test signatures can be loaded to the processor memory before the application of the test and expanded into test patterns by an on-chip test pattern generation program.

We now use the ALU (Fig. 10) of the PARWAN processor to illustrate the results of component test preparation in terms of the self-test signatures (Table IV). The ALU contains two 8-b data inputs (`in1` and `in2`) and one 3-b control input (`alu_code`). As shown in Fig. 2, `in1` is connected to the databus between the memory and the processor and `in2` is connected to the output of the accumulator.

The first column in Table IV shows the instructions for controlling the inputs to the ALU. They are load data from the memory to the accumulator (`lda`), store data from the accumulator to the memory (`sta`), compliment accumulator (`cma`), bit-wise AND (`and`), substraction (`sub`), and addition (`add`). Columns 2–4 show the input constraints imposed by these instructions, as well as the self-test signatures prepared for the unconstrained inputs. For a constrained input, the constraint is expressed in terms of a fixed value. For an unconstrained input, the self-test signature is expressed in a triple containing the following components: the seed of the pseudorandom pattern generator S, the configuration of the pseudorandom pattern generator C, and the number of pseudorandom patterns used N.

We now explain the test prepared for the ALU when instruction `sta` is used for applying the tests. In this case, the value of `alu_code` is constrained to 110. The value of `in1` is constrained to high-impedance Z as the tristate buffer from the memory to the databus is disabled when the data is stored from the accumulator to the memory Fig. 2. The value of `in2` is unconstrained; therefore, a self-test signature can be used to



(a)

```
Q = S
Do N times
  begin
  AC = Bitwise-and(C,Q);
  New_bit = Parity(AC);
  Q = New_bit:(Q >> 1);
  end
```

(b)

Fig. 11. Hardware and software implementation of LFSR. Self-test signature: (C, S, N). (a) Hardware implementation. (b) Software implementation.

specify the pseudorandom patterns that will be applied to `in2` during on-chip self-test. The self-test signature is expressed as (11111111, 01100011, 82), which means the seed and the configuration of the pseudorandom pattern generator are set to 11111111 and 01100011, respectively, and 82 patterns are used.

Notice that the self-test signatures containing the same seed and configuration are reused throughout the tests for the ALU, although different numbers of test patterns are used for different cases. Moreover, when instructions with two unconstrained inputs are used (e.g., `add`), one self-test signature is shared by both inputs. During the on-chip self-test phase, which we will explain in detail in Section III- B, the self-test signature are used to construct an array containing pseudorandom patterns. The array elements with odd indices will be used as test patterns for one input (`in1`) and the elements with even indices will be used for the other (`in2`).

As far as output observability is concerned, the ALU contains an 8-b data outputs (`data_out`) and a 4-b status output (`out_flag`). As we will explain in Section III- B2, both outputs can be observed by instructions. Therefore, both outputs can be used as primary outputs during the component-level fault simulation.

The result of the component-level fault simulation shows that the ALU tests in Table IV are expected to achieve a fault coverage of 98.81% on the ALU.

B. On-Chip Self-Test

The second step of our software-based self-test scheme is on-chip self-test, which uses an embedded software tester for the on-chip generation of component test patterns, the delivery of component tests, and the analysis of their responses (Fig. 4).

1) Test Generation Program: If tests are to be generated on-chip, we expand the component self-test signatures determined during component test preparation into test sets using a pseudorandom number generator. Fig. 11 illustrates this process. A software program emulating a hardware LFSR can be used as the pattern generator. The software LFSR leads to no test overhead and can be reused to generate any LFSR configurations. The configuration of the LFSR is determined by a self-test signature, which includes the characteristic polynomial C, the initial state S, and the number of test patterns to be generated N.

2) Test Application Program: According to Property 2 of T_c, since the component tests are developed under the constraints imposed by the processor instruction set, it will always be possible to find instructions for applying the component tests. An example is shown in Table IV.

```
0              lda addr(y)   //load AC
1              add addr(x)
2              sta data_out  //store AC
3              lda 11111111
4              brav ifv      //branch if overflow
5              and 11110111
6   label ifv  brac ifc      //branch if carry
7              and 11111011
8   label ifc  braz ifz      //branch if zero
9              and 11111101
10  label ifz  bran ifn      //branch if negative
11             and 11111110
12  label ifn  sta flag_out
```

Fig. 12. Observing status outputs.

On the output end, special care needs to be taken for collecting component test response. Data outputs and status outputs have different observability and should be treated differently during response collection. Here, we illustrate the propagation of status outputs with the ALU (Fig. 10) in the PARWAN processor.

The ALU has four status outputs: overflow `v`, carry `c`, zero `z`, and negative `n`, which can be observed by the instruction sequence in Fig. 12. Instructions 0–2 apply a test vector to ALU. The status outputs become available after instruction 1. Instructions 3–11 create an image of the status outputs in the accumulator. First, an all-one vector is loaded to the accumulator. If `v` is one, the all-one vector is left untouched. Otherwise, a 0 replaces the 1 at the fourth bit from right. Other status bits are treated similarly. After the execution of instruction 11, an image of the status output is created in the accumulator. Instruction 12 stores this image to memory.

In general, although there are no instructions for storing the status outputs of a component directly to memory, the image of the status outputs can be created in memory using conditional instructions. This technique can be used to observe the status outputs of any components.

C. Experimental Results

In this section, we report the application of the software-based self-test methodology. Before we report our experimental results, we describe the test evaluation framework we have developed and used to evaluate the fault coverage achieved by the software test program.

To evaluate the fault coverage of a test program on the processor under test, we have established the test evaluation framework shown in Fig. 13. The assembler takes the test program and prepares a very high-speed integrated-circuit HDL (VHDL) test bench containing the initialized instruction memory and data memory. The VHDL simulator takes the design description, runs the test bench, and captures the input signals to the processor. These are the test vectors to be applied during fault simulation. Finally, the fault simulator computes the fault coverage.

During component test preparation, pseudorandom tests were prepared for the ALU. A total of 205 test patterns were used. The

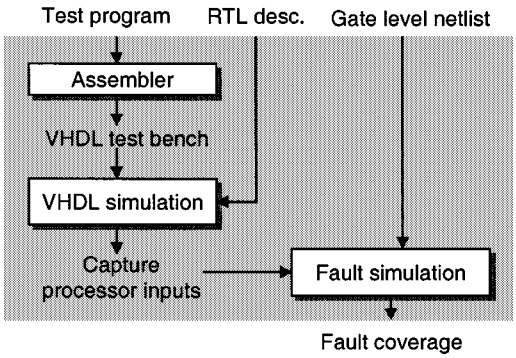


Fig. 13. Test evaluation framework.

expected fault coverage is 98.81%. Deterministic tests were prepared for SHU and PC. 40 test patterns were used for SHU and 12 for PC. The expected fault coverage is 99.27% for SHU and 85.00% for PC. We were unable to obtain full coverage for these components due to the existence of constraints imposed by the instruction set. No tests were generated for other components, as they are not easily accessible through instructions. We expect them to be tested intensively during the test for the targeted components.

Table V shows the statistics on various programs contained in the software tester including the test pattern generation program and the test application programs for ALU, SHU, and PC. For each program, we show the number of instructions included in the program, the size of the program in bytes, and the execution time in the number of processor cycles. A *low-speed* tester can be used to load the test programs into the processor memory. During the application of the self-test program, an external tester is not required to be hooked up to the processor for supplying the test patterns and monitoring the test responses. Therefore, the tester time is not determined by the execution time, but by the size of the test programs, which in this case is only 1129 B.

The complete self-test program achieved an overall fault coverage of 91.42% on the original PARWAN circuit, which includes tristate buffers. Notice that the proposed method does not require the processor outputs to be monitored by an external tester during the application of self-test. The test response is collected after the test by unloading the component test response stored in memory. In general, if a conventional fault simulator is used for evaluating the fault coverage of the proposed method, only primary outputs related to memory should be observed. This includes address outputs, data outputs, and read/write signals for the memory.

The component fault coverages along with the processor fault coverage are shown in Table VI in which DP I/F denotes the datapath interface, and CPU I/F denotes the CPU interface. The component fault coverages are obtained from the full-processor fault simulation, not from the component fault simulation. Notice that the DP I/F mainly consists of buses and tristate buffers. The fault coverage for this unit is low as its testability is reduced by the presence of the tristate buffers [30].

Table VII shows a comparison between the results of the proposed software-based self-test technique and conventional testing techniques such as full-scan and logic BIST (Table I). Note that full-scan and logic BIST have to be applied to the modified version of the original PARWAN circuit, which includes no bidirectional I/O pins or tristate buffers. The relative area overhead of logic BIST is large due to the small size of the original circuit. With a reasonable amount of area overhead, full-scan is able to achieve an acceptable fault coverage. However, it is not able to apply at-speed testing without the help of a high-performance tester. Being self-test techniques, logic BIST and the proposed self-testing technique are both able to apply at-speed test without relying on high-performance testers. Logic BIST, however, may not be suitable for high-performance processors due to the their intolerance to area and delay overhead. With the proposed software-based self-test technique, self-test can be conducted in the normal functional mode of the processor. Therefore, no test overhead is needed.

TABLE V
STATISTICS ON THE SELF-TEST PROGRAM

	TPG*	Test application			Total
		ALU	SHU	PC	
# instructions	46	213	243	73	575
Prog. size [bytes]	87	424	471	147	1129
Exec. time [cycles]	87764	37686	11604	595	137649

*Test pattern generation program.

To prove the effectiveness of our software-based self-test methodology on complex problems, we are now in the process of applying it to the picoJava processor core. A preliminary self-test program of 2050 instructions has been applied to its floating-point unit, which has an area of 23 365 equivalent NAND gates. A fault coverage of 81.18% has been achieved on the floating-point unit.

IV. CONCLUSION AND FUTURE DIRECTION

In conclusion, although hardware-based logic BIST may be an effective solution for testing ASICs [4], [17], we have demonstrated some of its disadvantages in testing complex designs such as microprocessors. Due to its reliance on pseudorandom test patterns, logic BIST must be accompanied by design changes required for making a circuit random-pattern testable. The design changes can be overly complex for designs like microprocessors, leading to not only significant increase in design time, but also unacceptable performance degradation. We have proposed a novel software-based self-testing technique that enables at-speed self-testing using the functionality of the processor under test. Structural faults are targeted during the self-test while the functionality of the processor is used as a vehicle for applying structural tests. We have demonstrated the effectiveness of the proposed method on a simple microprocessor. The advantages of the proposed technique include enabling at-speed testing with low speed testers as well as achieving high fault coverage without sacrificing area or performance. By breaking up a complex system into manageable pieces and targeting at individual components, we expect to apply this technique to large processors and systems in the future. Currently, by applying it to the picoJava processor core, we are expecting to extend the proposed self-test technique to address issues related to complex architectural features such as pipelining and superscalar.

TABLE VI
FAULT COVERAGE [%]

Component fault coverage										Processor
AC	IR	PC	MAR	SR	ALU	SHU	CTRL	DP I/F	CPU I/F	fault coverage
99.33	98.61	89.16	97.22	98.88	98.48	94.08	88.26	71.57	97.14	91.42

TABLE VII
COMPARISON WITH CONVENTIONAL TESTING TECHNIQUES

	Area overhead	Fault coverage	At-speed?	External tester?
Full Scan*	11.95%	89.39%	N	Y
Logic BIST*	169.09%**	88.69%	Y	N
Proposed technique	0	91.42%	Y	N

*On the modified circuit without test points.

**Biased by the small size of the original circuit.

The high fault coverage and low test overhead make the proposed software-based self-test methodology an attractive solution for testing high performance processors. However, we acknowledge that compared with traditional design-for-test techniques, the proposed technique is particularly challenging because it requires a certain amount of architectural knowledge on the processor under test. For instance, it requires the knowledge on the subset of instructions for accessing a particular component inside the processor. Furthermore, if ATPG is to be used to generate tests for this component, the constraints imposed by these instructions must be given or extracted. At the current stage of our research, the collection of relevant architectural knowledge and the generation of the test program still rely on manual effort. Nonetheless, the application of the proposed method demonstrates its great potential as a viable alternative for testing high-performance processors. In the future, we will be working toward the automation of the proposed method in order to make it a feasible solution for general processors.

APPENDIX
FORMAL PROOFS ON THE PROPERTIES OF T_c

Property 1: Assuming the tests generated under the constraints imposed by any *single* instruction i achieve the maximum possible fault coverage in the functional mode allowed by i, T_c can achieve the maximum possible fault coverage in the functional mode allowed by I_c. Thus, T_c detects any faults detectable by V_c.

Proof: Let I_p be the set of instructions used in the test generation step. ($I_c - I_p =$ the set of skipped instructions.)

To prove by contradiction, we would like to find an f such that f is detectable by V_c, but not T_c. As shown in Fig. 9, $T_c = \bigcup_{i \in I_p} T_{i,c}$. Therefore, $\forall i \in I_p$, there is no $T_{i,c}$ such that f can be detected by $T_{i,c}$ (Statement 1).

As shown in Fig. 9, instruction $j \in I_c - I_p$ iff $\forall v \in V_{j,c}, \exists i \in I_p$ such that $v \in V_{i,c}$ (instruction j is skipped only if all vectors in $V_{i,c}$ had been covered by previously processed instructions). Therefore, $V_c = \bigcup_{i \in I_c} V_{i,c} = \bigcup_{i \in I_p} V_{i,c}$. Since f is detectable by V_c, $\exists i \in I_p$ such that f is detectable by $V_{i,c}$ (Statement 2).

According to Statements 1 and 2, $\exists i \in I_p$ such that f is detectable by $V_{i,c}$, but not $T_{i,c}$. Since f is not detected by T_c, $f \in F$, which is the final list of undetectable faults (Statement 3).

The assumption in Property 1 can be expressed as follows. $\forall i \in I_p$, if f is previously undetected and f is detectable by $V_{i,c}$, f must be detectable by $T_{i,c}$. This leads to a contradiction with Statement 3. Hence, f does not exist and Property 1 holds.■

Property 2: Any test vector in T_c can be realized by at least one instruction in I_c.

Proof: As shown in Fig. 9,$T_c = \bigcup_{i \in I_c} T_{i,c}$. By definition, $V_c = \bigcup_{i \in I_c} V_{i,c}$. As a result of constrained test generation, $T_{i,c} \subseteq V_{i,c}$. Therefore, $T_c \subseteq V_c$.

Since $V_c = \bigcup_{i \in I_c} V_{i,c}$, for $\forall v \in V_c, \exists i \in I_c$ such that $v \in V_{i,c}$. That is, any vector in V_c can be realized by at least one instruction in I_c.

Since $T_c \subseteq V_c$, any vector in T_c can be realized by at least one instruction in I_c. □

ACKNOWLEDGMENT

The authors would like to thank P. Sanchez and K. Sekar for their help with the picoJava processor core.

REFERENCES

[1] Semiconductor Industry Association, *The National Technology Roadmap for Semiconductors*. San Jose, CA: SIA, 1997.
[2] V. D. Agrawal, C. J. Lin, P. Rutkowski, S. Wu, and Y. Zorian, "Built-in self-test for digital integrated circuits," *AT&T Tech. J.*, vol. 73, no. 2, pp. 30–39, Mar. 1994.
[3] Y. Zorian, "A distributed BIST control scheme for complex VLSI devices," in *Proc. 11th VLSI Test Symp.*, Atlantic City, NJ, Apr. 1993, pp. 4–9.
[4] G. Hetherington, T. Fryars, N. Tamarapalli, M. Kassab, A. Hassan, and J. Rajski, "Logic BIST for large industrial designs: Real issues and case studies," in *Proc. Int. Test Conf.*, Atlantic City, NJ, Sept. 1999, pp. 358–367.
[5] S. Hellebrand and H.-J. Wunderlich, "Mixed-mode BIST using embedded processors," in *Proc. Int. Test Conf.*, Washington, DC, Oct. 1996, pp. 195–204.
[6] R. Dorsch and H.-J. Wunderlich, "Accumulator based deterministic BIST," in *Proc. Int. Test Conf.*, Washington, DC, Oct. 1998, pp. 412–421.
[7] H.-J. Wunderlich, "Self test using unequiprobable random patterns," in *Dig. Papers. 17th Int. Symp. Fault-Tolerant Computing*, Pittsburgh, PA, July 1987, pp. 258–263.

[8] F. Brglez, C. Gloster, and G. Kedem, "Built-in self-test with weighted random-pattern hardware," in *Proc. IEEE Int. Conf. Computer Design: VLSI in Computers and Processors*, Cambridge, MA, Sept. 1990, pp. 161–167.
[9] D. S. Ha and S. M. Reddy, "On the design of random-pattern testable PLA based on weighted random-pattern testing," *J. Electron. Test.: Theory Applicat.*, vol. 3, no. 2, pp. 149–157, May 1992.
[10] J. Rajski and J. Tyszer, *Arithmetic Built-In Self-Test for Embedded Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1998.
[11] K. Radecka, J. Rajski, and J. Tyszer, "Arithmetic built-in self-test for DSP cores," *IEEE Trans. Computer-Aided Design*, vol. 16, pp. 1358–1369, Nov. 1997.
[12] S. Cataldo, S. Chiusano, P. Prinetto, and H.-J. Wunderlich, "Optimal hardware pattern generation for functional BIST," in *Proc. Meetings Design, Automation and Test Eur.*, Paris, France, Mar. 2000, pp. 292–297.
[13] G. Kiefer and H.-J. Wunderlich, "Deterministic BIST with partial scan," in *Proc. Eur. Test Workshop*, Constance, Germany, May 1999, pp. 110–116.
[14] S. Wang and S. K. Gupta, "DS-LFSR: A new BIST TPG for low-heat dissipation," in *Proc. Int. Test Conf.*, Washington, DC, Nov. 1997, pp. 848–857.
[15] S. Manich, A. Gabarro, M. Lopez, J. Figueras, P. Girard, L. Guiller, C. Landrault, S. Pravossoudovitch, P. Teixeira, and M. Santos, "Low power BIST by filtering nondetecting vectors," *J. Electron. Test.: Theory Applicat.*, vol. 16, no. 13, pp. 193–202, June 2000.
[16] T. G. Foote, D. E. Hoffman, W. V. Huott, T. J. Koprowski, B. J. Robbins, and M. P. Kusko, "Testing the 400 MHz IBM generation-4 CMOS chip," in *Proc. Int. Test Conf.*, Washington, DC, Nov. 1997, pp. 106–114.
[17] G. Kiefer, H. Vranken, E. J. Marinissen, and H.-J. Wunderlich, "Application of deterministic logic BIST on industrial circuits," in *Proc. Int. Test Conf.*, Atlantic City, NJ, Oct. 2000, pp. 105–114.
[18] C. Stolicny, R. Davies, P. McKernan, and T. Truong, "Manufacturing pattern development for the Alpha 21164 microprocessor," in *Proc. Int. Test Conf.*, Washington, DC, Nov. 1997, pp. 278–285.
[19] R. Rajsuman, "Testing a system-on-a-chip with embedded microprocessor," in *Proc. Int. Test Conf.*, Atlantic City, NJ, Sept. 1999, pp. 499–508.
[20] J. Shen and J. A. Abraham, "Native mode functional test generation for processors with applications to self test and design validation," in *Proc. Int. Test Conf.*, Washington, DC, Oct. 1998, pp. 990–999.
[21] K. Batcher and C. Papachristou, "Instruction randomization self test for processor cores," in *Proc. 17th IEEE VLSI Test Symp.*, Dana Point, CA, Apr. 1999, pp. 34–40.
[22] J. Rajski and J. Tyszer, "Built-in self-test for system-on-a-chip," presented at the International Test Conference 1999, Atlantic City, NJ, Sept. 1999. (tutorial).
[23] P. H. Bardell, W. J. McKenney, and J. Savir, *Built-In Test for VLSI: Pseudorandom Techniques*. New York: Wiley, 1987.
[24] Z. Navabi, *VHDL: Analysis and Modeling of Digital Systems*. New York: McGraw-Hill, 1993.
[25] PicoJava Microprocessor Cores, Sun Microsystems [Online]. Available: http://www.sun.com/microelectronics/picoJava
[26] R. Tupuri and J. A. Abraham, "A novel functional test generation method for processors using commercial ATPG," in *Proc. Int. Test Conf.*, Washington, DC, Nov. 1997, pp. 743–752.
[27] P. Vishakantaiah, J. A. Abraham, and D.G. Saab, "CHEETA: Composition of hierarchical sequential tests using ATKET," in *Proc. Int. Test Conf.*, Baltimore, MD, Oct. 1993, pp. 606–615.
[28] R. Tupuri, A. Krishnamachary, and J. A. Abraham, "Test generation for gigahertz processors using an automatic functional constraint extractor," in *Proc. 36th Design Automation Conf.*, New Orleans, LA, June 1999, pp. 647–652.
[29] P. Wohl and J. Waicukauski, "Test generation for ultra-large circuits using ATPG constraints and test-pattern templates," in *Proc. Int. Test Conf.*, Washington, DC, Oct. 1996, pp. 13–20.
[30] R. Raina, C. Njinda, and R. F. Molyneaux, "How seriously do you take possible-detect faults?," in *Proc. Int. Test Conf.*, Washington, DC, Nov. 1997, pp. 819–828.

Li Chen (S'98) received the B.S. and M.S. degrees in electrical and computer engineering from Carnegie Mellon University, Pittsburgh, PA, in 1996 and 1998, respectively. She is working toward the Ph.D. degree in electrical and computer engineering at the University of California, San Diego.

From January 1997 to July 1997, she was with Advanced Micro Devices, Sunnyvale, CA. Her current research interests include the self-testing and self-diagnosis of microprocessor cores.

Sujit Dey (S'90–M'91) received the Ph.D. degree in computer science from Duke University, Durham, NC, in 1991.

He is currently an Associate Professor with the Electrical and Computer Engineering Department, University of California, San Diego. Prior to joining the University of California at San Diego in 1997, he was a Senior Research Staff Member at the NEC C&C Research Laboratories, Princeton, NJ. He has presented numerous full day and embedded tutorials and has participated in panels on the topics of hardware–software embedded systems, low-power wireless systems design, and deep submicrometer system-on-chip design and test. He is affiliated with the DARPA/MARCO Gigascale Silicon Research Center and the Center for Wireless Communications at the University of California at San Diego. He has authored or coauthored 90 journal and conference papers, one book, and ten U.S. patents. His current research interests include devloping innovative hardware–software architectures and methodologies to harness the full potential of nanometer technologies for future networking and wireless appliances.

Dr. Dey received Best Paper Awards at the 1994, 1999, and 2000 Design Automation Conferences and the 11th VLSI Design Conference in 1998. He was the General Chair, Program Chair, and member of organizing and program committees of several IEEE conferences and workshops.