Embedded Hardware and Software Self-Testing Methodologies for Processor Cores

Li Chen†, Sujit Dey†, Pablo Sanchez‡, Krishna Sekar†, and Ying Chen†

†Dept. of ECE, University of California at San Diego, La Jolla, CA 92093, USA

‡Dept. of Electronic Technology and System Engineering (TEISA), University of Cantabria, Santander, Spain

lichen, dey@ece.ucsd.edu

Abstract

At-speed testing of GHz 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, which can be used by processors to test themselves at-speed. Currently, Built-In Self-Test (BIST) is the primary self-test methodology available and is widely used for testing embedded memory cores. In this paper, we report our experiences in applying a commercial BIST methodology to two processor cores and analyze the problems associated with the current hardware-based BIST methodologies. We 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.

1. Introduction

As the speed of microprocessors approaches the GHz range, at-speed testing is becoming increasingly critical. However, testers with speed matching the speed of GHz processors will be increasingly costly. According to the 1997 Semiconductor Industry Association Roadmap [1], if the current testing techniques are to be continued, the test equipment cost can rise towards $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 GHz processor can test itself without relying on high-speed, prohibitively expensive external testers.

This work is supported by MARCO/DARPA Gigascale Silicon Research Center (GSRC).

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. While embedded memory components in processors widely use memory BIST techniques, the non-memory (logic) parts of processors typically do not use BIST. This is because while memory BIST performs well due to the deterministic nature of the memory tests facilitated by the regular structure of memory components, logic BIST remains impractical for large designs due to its reliance on random test patterns.

An alternative to hardware-based self-testing techniques like BIST is software-based self-testing. While 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 regularly used to perform manufacturing testing of processors. However, its application relies on external testers and its results in terms of manufacturing fault coverage are low, as functional tests are not targeted at structural faults. Recently, researchers have started investigating self-test techniques for processors using processor instructions. Shen et al., and Batcher et al. have proposed techniques for functional self-testing of processors [3][4]. Both techniques rely on generating and applying random instruction sequences to processor cores. In [5][6][7][8], the processor functionality has been used for on-chip test pattern generation and test response compaction. In [5] and [6], random operations and operands are generated and applied to test the ALUs of DSP cores. In [7] and [8], the processor is used to generate random test patterns, and scan chains are used to apply the test patterns.

In this paper, to analyze the strength and the limitations of current hardware-based self-testing techniques, we have applied a state-of-the-art logic BIST technique to a simple processor core as well as a complex, commercial processor core. Through these applications, we demonstrate some of the general problems associated with logic BIST, such as inability to provide high fault coverage without extensive design changes on the circuit-under-test. These problems 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, 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. The software-based approach has the advantage of programmability and flexibility, which can be used to generate desirable random test sets on-chip without any hardware overhead. In addition, software instructions can enable on-chip test application by guiding 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 techniques.

To circumvent the low fault coverage associated with random pattern testing of processors, our approach first determines the

structural test needs of processor components, 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.

In the rest of the paper, we first evaluate the performance of a hardware-based commercial logic BIST tool (LBIST) on two processor cores. We will then describe the proposed software-based self-testing methodology in Section 3. The performance of the software-based approach is evaluated and compared to that of LBIST.

2. Evaluating Hardware-Based Logic BIST Techniques

We present our experience of applying a commercial logic BIST tool (LBIST) to two processor cores: PARWAN and PicoJava-II. Figure 1 shows the BIST structure inserted by LBIST. Instead of relying on an external tester for applying tests to the scan chains, LBIST generates test vectors on-chip using an LFSR. The outputs of the LFSR are connected to the scan chains through a phase shifter, which is designed to reduce the linear correlation among scan chains. The outputs of the scan chains are compressed on-chip using a MISR. Test points may be inserted to improve the fault coverage.

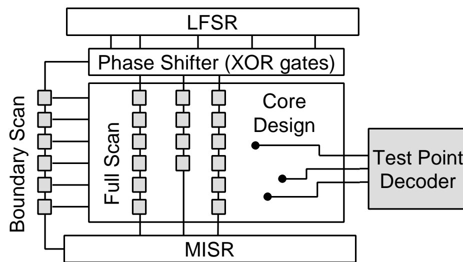


Figure 1. LBIST

As logic BIST relies on the application of random test patterns, the design-under-test often has to be modified extensively to be random pattern testable [9]. The next two sections describe the application of LBIST on the two processor cores, including all design changes needed for making the cores BIST-ready.

2.1 Case study I: PARWAN processor core

We first applied LBIST on a simple accumulator-based microprocessor named PARWAN [10] (Figure 2). It includes the following components: Arithmetic Logic Unit (ALU), Accumulator Unit (AC), Controller (CTRL), Instruction Register Unit (IR), Program Counter Unit (PC), Memory Address Register Unit (MAR), Shifter Unit (SHU), and Status Register Unit (SR). The synthesized version of PARWAN contains 1785 equivalent

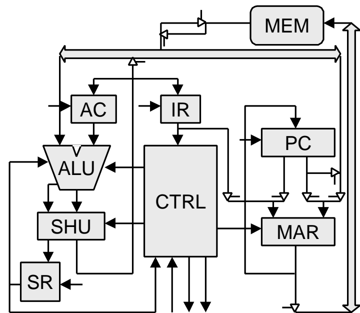


Figure 2. PARWAN processor core

NAND gates and 53 flip-flops. The data bus is 8-bit wide, shared by both `data_in` and `data_out`. The address bus is 12-bit wide. Accesses to both buses are controlled by tri-state buffers.

Next we describe the design changes we have made on PARWAN in order to make the application of LBIST effective. As the MISR signatures can be corrupted by undefined values, logic BIST tools do not accept circuits with possible bus-contention problems [9]. Moreover, LBIST is not able to insert boundary scan to a circuit containing bi-directional pins. Hence, before applying LBIST, we had to manually modify the circuit description of PARWAN. The modifications include: (1) splitting all bi-directional pins into separate I/O pins, (2) replacing all tri-state buffers with selectors, and (3) inserting test points to improve the testability of the circuit.

Table 1 shows the results of LBIST 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 LBIST on the modified circuit, and (5) the results of LBIST on the modified circuit with test points (3 control points and 11 observe points). For the two LBIST runs, we divided the 53 flip-flops of PARWAN into 5 scan chains. LBIST automatically chooses an 18-bit 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 delay of the circuits in *ns*, 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.

Table 1. Parwan: full scan vs. LBIST

	Area [gate count]	Delay [ns]	# Test patterns	Fault Coverage
Original. ckt	888	70.06	--	--
Modified ckt	812	82.50	--	--
Full Scan*	909	82.87	640	89.39%
LBIST*	2185	104.42	32767	88.69%
LBIST**	2246	104.42	32767	97.34%

* On the modified circuit
** On the modified circuit with test points

As can be observed from the table, both full scan and LBIST techniques can achieve comparable fault coverage without the help of additional test points. However, LBIST requires much higher area and delay overhead than full scan due to the need for numerous BIST circuitry, such as the LFSR, the MISR, the boundary scan chain, the phase shifter, and the BIST controller. Note that the fault coverage of LBIST can be significantly enhanced by the use of test points.

2.2 Case study II: PicoJava-II processor core

PicoJava-II (Figure 3) is a soft microprocessor core provided by Sun Microsystems through the Sun community licensing program. This implementation of the Java Virtual Machine is a stack-based 32-bit microprocessor with 300 instructions. It contains six pipeline stages, and can execute up to 4 instructions in one cycle. The instruction set includes most of the Java bytecodes and some C-oriented instructions. The core users can configure the instruction/data caches and the FPU.

The synthesizable description of PicoJava has 46376 lines of Verilog code and includes 7 technology-dependent embedded memory blocks. To obtain the gate-level implementation, we had to modify all the synthesis and timing scripts to be able to adapt to the synthesis tool and the technology library used in our design flow.

The synthesized PicoJava-II core contains 167 I/O ports and 6801 flip-flops. Figure 3 shows component areas in terms of the number of equivalent NAND gates. The total area is 127887 for the logic components and 313989 for the embedded memory

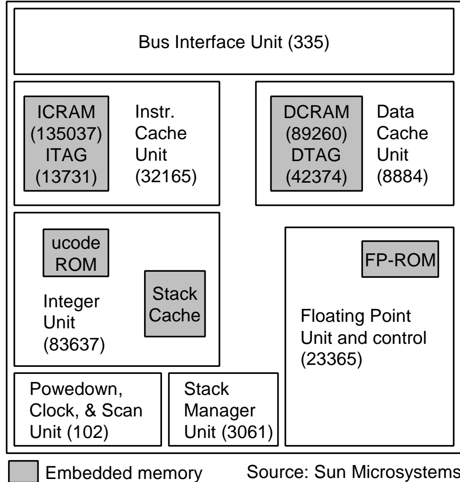


Figure 3. PicoJava-II processor core

components. The total number of faults in the logic blocks is 532,527. Using a commercial fault simulator, it takes 20 seconds to simulate one test cycle, if all faults are included in the fault list.

Table 2 shows the results of applying full scan and LBIST on the logic part of the PicoJava processor core. The values of area overhead and fault coverage reported in the table are with respect to the logic part of the processor core.

Full scan is able to achieve a fault coverage of 95.54%, with an area overhead of 11.13%. The area overhead is an under-estimate, as it does not include the routing overhead of scan-chains.

Prior to the application of LBIST, several design changes had to be made in order to make the PicoJava processor core BIST-ready. First, embedded memories were bypassed with scan flip-flops in the test mode [9], 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.

In the first LBIST experiment (LBIST-1), an LFSR of size 24 was used. A primitive polynomial was chosen by LBIST to configure the LFSR. The number of random test patterns used was 32,767. The fault coverage was unacceptably low (58.81%), with an area overhead higher than that of full scan. The increase in the area overhead was caused by the insertion of the BIST circuitry, as well as the design modifications required for making PicoJava BIST-ready.

To improve the testability of the design, we performed the second LBIST experiment (LBIST-2), in which the BIST circuitry in LBIST-1 was augmented by 100 control points and 100 observe points. This leads to a significant boost in the final fault coverage (an additional 23.72% compared to LBIST-1). The insertion of the test points leads to a slight increase in area.

We have also investigated the effect of other BIST parameters on the resulting fault coverage, including the LFSR size and the number of random patterns used in test.

In the third LBIST experiment (LBIST-3), the BIST configuration in LBIST-2 was modified so that a larger LFSR (32-bit) was used, leading to a fault coverage improvement of 0.40% over LBIST-2. As the increase in the LFSR size does not lead to a significant improvement in fault coverage, we did not increase the size of the LFSR further.

In the final LBIST experiment (LBIST-4), the exact same BIST configuration in LBIST-2 was used, but the number of random patterns applied was increased from 32,767 to 1,000,000. This leads to a fault coverage improvement of 1.58% over LBIST-2. Compared to LBIST-2, LBIST-4 caused a slight increase in area overhead, as a larger pattern counter was used.

The results in Table 2 show that LBIST was not able to achieve a very high fault coverage on PicoJava, even after the insertion of a large number of test points. The fault coverage can be improved by increasing the size of the LFSR and the number of random patterns. However, the improvement is marginal. Aside from the low random-pattern-testability of a large design, the mechanism used for breaking the combinational loops could also contribute to the low fault coverage of LBIST on PicoJava.

In summary, although it is an attractive solution to the problem of at-speed test, the application of LBIST on the two processor cores shows its potential disadvantages. Processors, due to their complex control structures, 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. Design changes are often 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.

3. A Software-Based Self-Test Methodology Targeting at Structural Faults

Unlike hardware-based self-testing, software-based testing enables the use of random pattern generation programs with various configurations without introducing any test overhead. Moreover, software instructions will be able to guide test patterns through the complex processor, avoiding the blockage of the test data due to non-functional control signals.

In this paper, we propose a novel software-based processor self-testing methodology that delivers structural tests to components using processor instructions. Our self-testing scheme includes two steps. The test preparation step includes the generation of realizable tests for components of the processor and the encapsulation of component tests into self-test signatures. The self-testing step involves the use of 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 Figure 4. If self-test signatures are used, an on-chip

Table 2. PicoJava-II: full scan vs. LBIST

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

test generation program emulates a pseudo random pattern generator and expands the signatures into test patterns. The test patterns are applied to components by an 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 a test response analysis program. The responses are stored into memory and can later be unloaded and analyzed by an external tester.

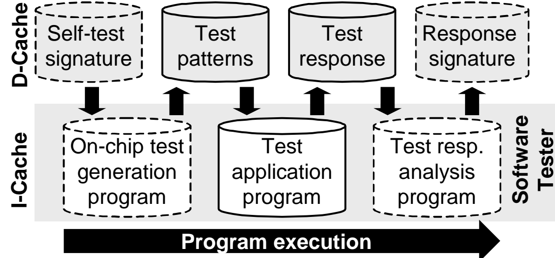


Figure 4. Self-test methodology

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 GHz processors with low speed testers.

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

3.1 Component test preparation

During the component test preparation step, we develop structural tests for individual components of the processor, such as the ALU, the SHU, and the PC. Component tests can either be stored or generated on-chip. 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 pseudo random 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 pseudo random number generation program. Multiple self-test signatures may be used for one component if necessary. A low-speed tester can be used to load the self-test signatures or the pre-determined tests to the processor memory prior to the application of test.

3.1.1 Instruction-imposed constraints

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. Thus, component tests must obey constraints imposed by the processor instruction set.

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 Figure 5. The input signals include `data_in`, `in_flag`, and the shifting signals from the controller. `in_flag` include 4 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.

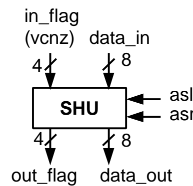


Figure 5. SHU

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 spatial constraints imposed by the processor instruction set include the following:

1. `asl` and `asr` cannot be both 1,
2. `z` and `n` must be consistent with `data_in`, and
3. `v = xor(c, sign_bit(data_in))`.

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 accumulator (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 Figure 6, 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.

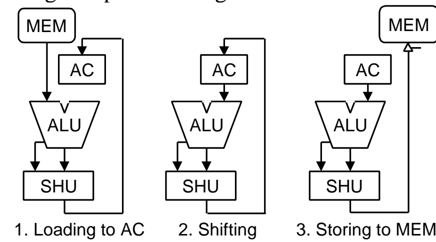


Figure 6. Hardware paths involved in testing the SHU

Tupuri et al. and Vishakantaiah et al. have addressed the issue of constraint test generation in [11][12][13]. They have proposed a methodology to systematically extract structural constraints for components of a processor from the processor HDL description. In the future, we will investigate on enhancing such structural constraint extraction methods to extract constraints imposed by the instruction set, as is required in our work.

3.1.2 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 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 [11].

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 Figure 7. 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 0's. The `v` and `c` flags are set to 0's in the second and the third steps, since neither the shift instructions nor the store instruction can set them to 1. At any phase, the inputs to SHU must also obey the spatial constraints we have described before.

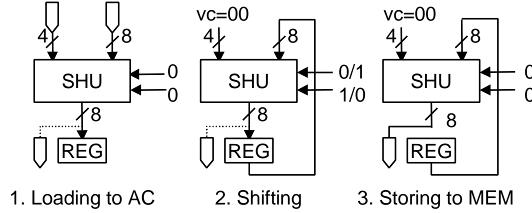


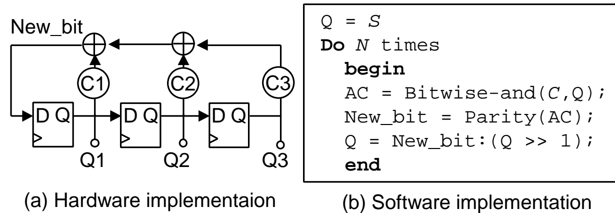
Figure 7. Circuit for modeling temporal constraints on SHU

3.2 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 (Figure 4).

3.2.1 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 pseudo random number generator. Figure 8 illustrates this process. A software program emulating a hardware LFSR could 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 of the LFSR (*C*), the initial state of the LFSR (*S*), and the number of test patterns to be generated (*N*).



Self-test signature: (*C*, *S*, *N*)

Figure 8. Hardware and software implementation of LFSR

3.2.2 Test application program

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.

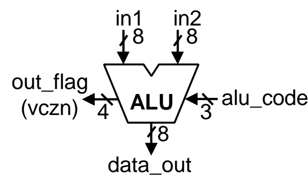


Figure 9. ALU

However, 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 (Figure 9) in the PARWAN processor.

The ALU has four status outputs, `v` (overflow), `c` (carry), `z` (zero), and `n` (negative), which can be observed by the instruction sequence in Figure 10. 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 zero replaces the one at the 4th 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, though 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.

```
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
```

Figure 10. Observing status outputs

3.3 Experimental results

In this section, we report the application of the software-based self-test methodology described above. 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 Figure 11. The assembler takes the test program and prepares a 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.

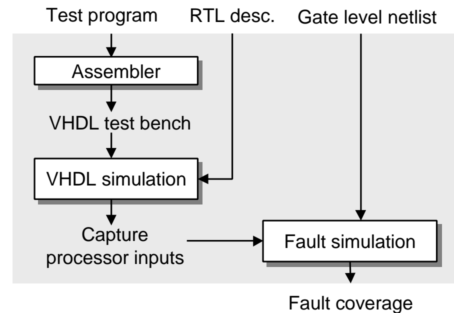


Figure 11. Test evaluation framework

During component test preparation, pseudo random tests were prepared for the ALU. A total of 205 test patterns were used. The 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 3 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 bytes.

Table 3. 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

The complete self-test program achieved an overall fault coverage of 91.42% on the original PARWAN circuit, which includes tri-state 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 4, 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 datapath interface (DP I/F) mainly consists of buses and tri-state buffers. The fault coverage for this unit is low as its testability is reduced by the presence of the tri-state buffers [14].

In summary, the proposed method is able to achieve high fault coverage without any test overhead. Most importantly, it enables at-speed testing without any requirement on the performance of the external tester. Since the test is applied in the normal operational mode of the processor, we avoid the problem of creating bus contentions and combinational loops as in the case of scan-based test.

To prove the effectiveness of our software-based self-test methodology on large designs, we are now in the process of applying it to the PicoJava processor core. A self-test program of 2050 instructions has been applied to its floating-point unit, which has an area of 23365 equivalent NAND gates. A fault coverage of 81.18% has been achieved on the floating-point unit.

4. Conclusion

In conclusion, we have demonstrated some of the disadvantages associated with hardware-based logic BIST techniques by applying a commercial BIST tool to two processor cores. In addition to the test overhead required by the insertion of BIST structures, hardware-based logic BIST techniques must be accompanied by design changes required for making the processor-under-test random-pattern-testable. 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 on large processors and systems in the future. Currently, by applying it to a large industrial processor like 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 super-scalar.

References

[1] *The National Technology Roadmap for Semiconductors*, Semiconductor Industry Association, 1997.

[2] V.D. Agrawal et al., "Built-in self-test for digital integrated circuits," *AT&T Technical Journal*, Mar. 1994, pp. 30.

[3] J. Shen and J. A. Abraham, "Native mode functional test generation for processors with applications to self test and design validation," *Proceedings of the International Test Conference 1998*, Washington, DC, Oct. 1998, pp. 990-999.

[4] K. Batcher and C. Papachristou, "Instruction randomization self test for processor cores," *Proceedings of the 17th IEEE VLSI Test Symposium*, Dana Point, California, April 1999, pp. 34 – 40.

[5] J. Rajski and J. Tyszer, *Arithmetic Built-in Self-Test for Embedded Systems*, Prentice Hall, 1998.

[6] K. Radecka, J. Rajski, and J. Tyszer, "Arithmetic built-in self-test for DSP cores," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol.16, no.11, Nov. 1997, pp. 1358 – 69.

[7] S. Hellebrand and H.-J. Wunderlich, "Mixed-mode BIST using embedded processors," *Proceedings of the International Test Conference 1996*, Washington DC, Oct. 1996, pp. 195 – 204.

[8] R. Dorsch and H.-J. Wunderlich, "Accumulator based deterministic BIST," *Proceedings of the International Test Conference 1998*, Washington DC, Oct. 1998, pp. 412 – 421.

[9] G. Hetherington, T. Fryars, N. Tamarapalli, M. Kassab, A. Hassan, and J. Rajski, "Logic BIST for large industrial designs: real issues and case studies," *Proceedings of the International Test Conference 1999*, Atlantic City, New Jersey, Sept. 1999, pp. 358 – 367.

[10] Z. Navabi, *VHDL: Analysis and modeling of digital systems*, New York, McGraw-Hill, 1993.

[11] R. Tupuri and J. A. Abraham, "A novel functional test generation method for processors using commercial ATPG," *Proceedings of the International Test Conference 1997*, Washington DC, Nov. 1997, pp. 743 – 752.

[12] P. Vishakantaiah, J. A. Abraham, and D. G. Saab, "CHEETA: Composition of hierarchical sequential tests using ATKET," *Proceedings of the International Test Conference 1993*, Baltimore, Maryland, Oct. 1993, pp. 606 – 615.

[13] R. Tupuri, A. Krishnamachary and J. A. Abraham, "Test generation for gigahertz processors using an automatic functional constraint extractor," *Proceedings of the 36th Design Automation Conference*, New Orleans, Louisiana, June 1999, pp. 647 – 652.

[14] R. Raina, C. Njinda, and R.F. Molyneaux, "How seriously do you take possible-detect faults?" *Proceedings of the International Test Conference 1997*, Washington DC, Nov. 1997, pp. 819-828.

Table 4. Fault coverage [%]

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