Effective Hybrid Test Program Development for Software-Based Self-Testing of Pipeline Processor Cores

Tai-Hua Lu, Chung-Ho Chen, and Kuen-Jong Lee

***Abstract*— This paper presents an effective hybrid test program for the software-based self-testing (SBST) of pipeline processor cores. The test program combines a deterministically developed program which explores different levels of processor core information and a block-based random program which consists of a combination of in-order instructions, random-order instructions, return instructions, as well as instruction sequences used to trigger exception/interrupt requests. Due to the complementary nature of this hybrid test program, it can achieve processor fault coverage that is comparable to the performance of the conventional scan chain method. The test response observation methods and their impacts on fault coverage are also investigated. We present the concept of micro observation versus macro observation and show that the most effective method of using SBST is through a multiple input signature register connected to the processor local bus, while conventional methods that observe only the program results in the memory lead to significantly less processor fault coverage.**

***Index Terms*—Fault coverage measure, fault observation method, hybrid SBST, processor testing, software-based self-testing (SBST).**

I. Introduction

With the rapid advances in semiconductor manufacturing technology, more and more processors are now being integrated into a system-on-a-chip (SoC) design. However, the poor controllability and observability of these embedded processor cores produces testability problems. Traditional test methods that rely on built-in scan chains usually require large chip area overhead and may induce significant performance degradation. This problem is especially serious for high-speed pipeline processors where all flip-flops in pipelined registers and register files may have to be scanned if full scan is required. As a result, software-based self-testing (SBST) methodologies that require no scan chains have received much attention recently [1], [2].

SBST test programs can be developed deterministically [1], [3], [4], randomly [5], [6], or both [2]. Some of these methods also make use of design-for-testability (DFT) hardware to assist program execution or result observation [1], [5], [6]. Various efforts for SBST automation, including automating module level constraint extraction based on the register transfer level (RTL) description of the circuit [7] or test routine description [4] are also presented. In manufacturing testing, SBST can employ a low cost automatic test equipment (ATE) to initiate the processor through cache systems [8].

Our previous work has presented a deterministic SBST methodology that uses multiple-level abstractions of the processor information for test program development [1]. Although good fault coverage has been achieved compared to other SBST methods, many faults related to processor control or glue logic have experienced poor test quality and as a result, the processor fault coverage is still 4%–5% less than that of a full scan approach.

Manuscript received April 03, 2009; revised August 14, 2009. This work was supported in part by the National Science Council, Taiwan under Grant NSC 96-2220-E-006-011 and Grant NSC 97-2221-E-006-250-MY3, and by the Program for Promoting Academic Excellence of Universities in Taiwan.

The authors are with the Department of Electrical Engineering and Institute of Computer and Communication Engineering, National Cheng-Kung University, Tainan 701, Taiwan (e-mail: aaron@casmail.ee.ncku.edu.tw; chchen@mail.ncku.edu.tw; kjlee@mail.ncku.edu.tw).

Digital Object Identifier 10.1109/TVLSI.2009.2036184

This paper presents a new hybrid SBST methodology that combines our previous deterministic program with an effective basic-block based random program. Many test strategies, such as mechanisms to test exceptions and interrupts, undefined instructions, and processor modes, have been developed for the random program. Experimental results show that this SBST method can boost the processor fault coverage to a level that is close to the results of scan-based approaches. Specifically, more than 98% single stuck-at fault coverage is obtained when evaluated on both the miniMIPS and ARMv4 instruction set architecture (ISA) processor cores. To our best knowledge, this level of performance has not yet been seen in any previous SBST studies.

To validate the fault coverage measurement used in this study, we also investigate the various test response observation methods and their impacts on the fault coverage. The principal contributions of this paper thus are the following: 1) a practical and viable SBST methodology that is capable of achieving high processor fault coverage, comparable to that of a full scan chain; 2) many test program development strategies which are critical to improving fault coverage, but often neglected; and 3) three fault observation methods that can be used for SBST, and a demonstration of their impacts on processor fault coverage.

The rest of this paper is organized as follows. Section II presents the SBST supporting environment and fault observation methods. Section III presents the proposed SBST methodology. Section IV describes the experimental system and evaluation results. Finally, Section V gives the conclusions of this paper.

II. SBST Testing Infrastructure

A. Simple ATE Design for SBST

We first clarify the test infrastructure of our SBST environment which includes the use of a simple ATE system board. The ATE board provides the necessary system support for testing the processor core embedded in the SoC chip. Before testing, the ATE management processor loads the test code from flash memory into the main memory of the ATE board. The processor under testing then copies the self-test instructions through the chip's external memory interface to the internal memory of the SoC chip. The execution of test instructions is done from the main memory so as to speed up the test process and enable the observation of the execution results directly through the processor's local bus. The processor's on-chip caches are disabled when executing SBST from the main memory. Thus some faults in the cache unit are not tested in this work. The detection of these faults requires further research and is considered in our other work [11]. The ATE system can also use a time-out mechanism should the processor under testing fail to respond during testing.

An intuitive way for SBST to determine if the processor is successfully manufactured is to check the written results in the main memory. However, since only the store instructions of the processor can write results to the memory, a written datum in the memory is actually the "lumped" result of many processor clock cycles before the execution of the write. We call this fault effect observation method the "macro" observation. Another feasible way to observe test results is to employ a multiple-input signature register (MISR) at the output bus of the processor. This can capture the processor outputs cycle by cycle and thus is called the "micro" observation method since it checks every detail for faulty output detection. Understandably, the way of how and where to observe the effect of the test code affects the processor fault coverage measured as will be more detailed in Section II-B.

B. Fault Observation Methods for SBST

We will compare three observation methods in this paper. The first method is to observe the processor output bus responses on both edges

of the processor clock. Reading the test response in this "micro" way allows the best possible processor fault coverage that a test program can achieve.

A variant micro observation method only captures the responses on either the positive edge or the negative edge of the processor clock. To emulate this observation method assuming on the positive edge in fault simulation, we use a tri-state controlled buffer to mask the test response when the clock signal is high. As a result, the test response is only observed on the rising edge of the clock.

Unlike the "micro" observation method, which captures cycle-based or edge-based hardware signals, the "macro" observation method observes the written results in memory for fault coverage evaluation. Nevertheless, it is possible to evaluate macro observation-based fault coverage using a micro observation method. To emulate this macro observation method, this study develops a fault simulation model that simply extends the mask circuit concept. That is, to measure the fault coverage based on the written results in memory, the signal mask circuit only allows the hardware response signals that affect the correctness of the store operation to pass for observation. The first type of hardware response signals are the active signals during a write cycle, for instance, the address bus and data bus output signals. During write cycles, these output signals directly impact the written results of the store instructions. The second type of hardware response signals are those that indirectly affect the correctness of the stored instructions. For example, if the instruction address bus outputs an erroneous instruction address, the tested processor will fetch an incorrect instruction. Consequently, this generates an erroneous software response. As a result, the instruction read and control signals are observed during the read cycles.

III. Overview of the Processor Test Program

The proposed test program consists of a deterministic test code that uses a multiple-level abstraction-based methodology as depicted in [1] and a random program based on a basic-block development method to be detailed in this paper. The deterministic test program explores the design information of processor architecture, RTL, and gate-level for different types of processor components. The test routine development methodology applies the most useful information of a certain level to the different parts of the processor core. However, it is impossible to test faults beyond the limited functional coverage of the deterministic program itself. On the other hand, effective random instruction sequences can greatly remedy these deficiencies.

Fig. 1 illustrates the random test routine generation methodology. A basic block is built based on the information abstracted from the processor's architecture model and ISA. Interrupt and exception-generating instructions are randomly inserted between the three types of basic blocks: in-order, out-of-order, and return blocks which will be described later. The program generation flow ends when the target processor fault coverage is reached or saturated.

A. Supporting Mechanism for Random Program Execution

When running a random program, the instruction fetch and data access addresses are unpredictable, leading the processor to fetch unknown instructions and access unknown data. Therefore, a simple hardware device called a test shell is placed between the processor core and system bus as illustrated in Fig. 2. This device enables sequential execution of random programs. The test shell is only enabled when the processor executes the random test program. As a random-to-sequential address converter, the test shell generates the sequential address either to get the instruction in the test code or the data the test code uses from the memory regardless of what address the processor generates.

To support interrupt testing, this study selects specific undefined instructions (illegal instructions) in the processor ISA to trigger interrupt

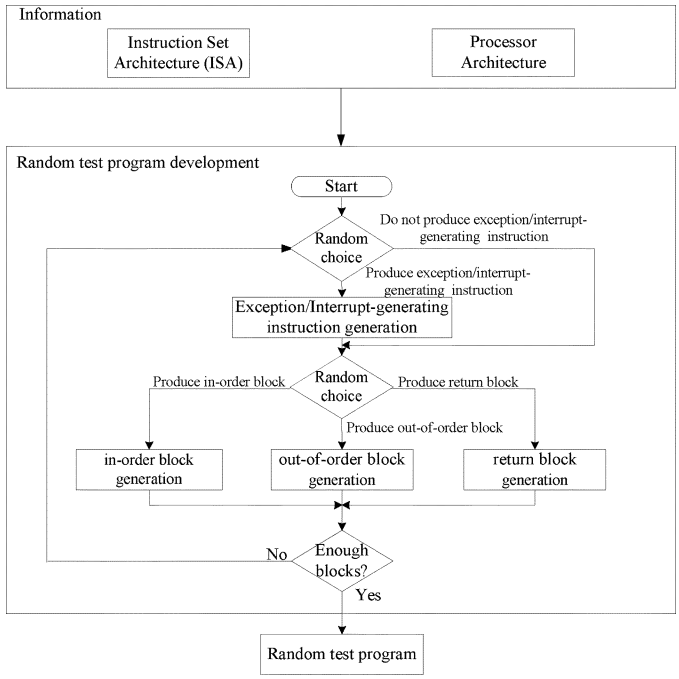

Fig. 1. Random test program generation.

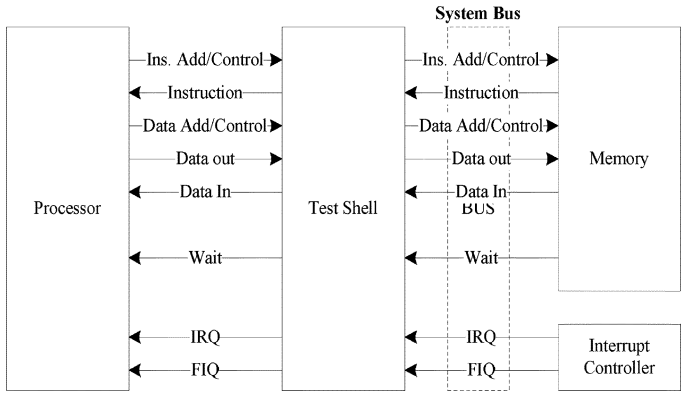

Fig. 2. Test shell used for sequential execution of random program.

events through the test shell. When a designated undefined instruction is fetched, the test shell recognizes it and triggers the predefined interrupt event. The test shell can be implemented as part of a common bus wrapper and thus introduces very little penalty for memory access time. During processor core testing, the processor's on-chip caches and memory management unit (MMU) are disabled so that all accesses go to the main memory through the test shell. SBST can also apply to the on-chip caches and MMU. Interested readers can refer to the work in [11].

B. Basic Block Structure

The main advantage of using basic blocks to form a test program is a systematic approach to specifying the required constraints for a basic block, that is, the type of instructions present in a basic block and the instruction sequence order in a basic block. To build a basic block, the processor instruction set is first classified based on functional operations into types of memory accesses, arithmetic logic units (ALUs), branches, and status register accesses.

Including instructions for observation, each basic block contains every type of the instructions, except for the exception and interrupt-generating instructions. This study defines three types of basic blocks: in-order, out-of-order, and return basic blocks. The rationale behind the in-order basic block is to build a program module similar to a common program basic block by including every possible instruction category. This type of basic block characterizes the instruction sequence, which has an explicit data dependency. Specifically, the basic block begins with two load instructions that employ random source operands (first load) and random destination registers (second load), respectively. Then two instructions from the ALU category are randomly selected and a RAW hazard is enforced between them. The results, including program status registers are subsequently written back for observations. Lastly, a branch instruction using a randomly selected target address is executed to complete the in-order block. Unlike an in-order basic block in which the instruction sequence is predefined, an out-of-order basic block generates the instruction sequence randomly, even though it also contains instructions of each type and the same number of instructions as an in-order block. The basic block also includes the no-operation, which is deemed as an instruction, for the purpose of sequence diversity. In this way, an out-of-order basic block produces a random instruction sequence that is almost impossible to obtain in deterministic test programs and thus helps test the corner cases and difficult-to-test faults.

The third type of basic block is called a return block, which is mainly used for processor mode switching. While an exception or interrupt brings the processor to a privilege mode, the return block brings the processor back from the privilege mode to the user mode. To test the processor modes, which may incorporate the use of complicated register banks, a return block triggers the testing on the access control of the shadow registers in various processor modes, including those for program status registers. The return block, interrupts and exception-generating instructions all activate the circuitry for processor mode switching, which is a critical function in modern operating systems.

C. Exception/Interrupt Testing

In general, a processor may have different kinds of external interrupts and internal exceptions. Such processors may contain shadow registers used in various privilege modes. To test these registers and their access control logic, either the external-interrupt pins must be enabled or some exception-generating instructions must signal the processor to enter a privilege mode. Testing interrupts can be a complicated issue for an SBST-based methodology, especially for pipelined processors. To assist SBST testing for processor interrupt mechanisms, this study defines interrupt-request instructions through the undefined instructions of the processor, that is, the reserved op-codes. The test shell snoops the fetching of an interrupt-request instruction to generate the interrupt request. In this way, the processor ISA does not change. This design simply takes advantage of undefined instruction bit patterns and recognizes the chosen ones to generate various external interrupt requests. If a processor has no undefined instructions, an alternative is to specify a particular short sequence of instructions or memory addresses for test shell snooping.

The instruction fields of the undefined instructions or the exception requesting instructions (SWI, for instance) may include immediate or don't-care values. The proposed test program also pays attention to these fields by using random values to improve the test results.

IV. Experimental System and Results

This study uses ModelSim [12] for logic simulation, Design Analyzer [13] for synthesis, and TurboScan [14] based on the stuck-at fault model for fault simulation. Experiments are conducted on an ARMv4 ISA pipelined processor core implemented by us [1] and the miniMIPS processor [9]. The fault coverage of the hybrid test program is reported by executing the deterministic program and the random program subsequently based on the structurally testable faults which are obtained by removing the undetectable faults.

TABLE I
ARMv4 and miniMIPS Processor Fault Coverage (F.C.)

Synthesis Case	Gate count/ # of faults	F.C.% (Det.)	F.C.% (Random)	F.C.% (Hybrid)	F.C.% (Full Scan)
ARMv4 (0.35 u/ 50Mhz)	63,281/ 152758	93.72 (1743 Inst.)	96.66	98.04	98.15 (681,061)
ARMv4 (0.18 u/125Mhz)	45,960/ 137222	92.62 (2415 Inst.)	95.43	96.93	97.73 (673,855)
mMIPS (0.35 u/50Mhz)	58,540/ 141668	93.63 (2365 Inst.)	96.99	98.46	99.05 (668,777)
mMIPS (0.18 u/125Mhz)	50,325/ 150474	92.89 (2435 Inst.)	97.02	98.13	99.04 (679,382)

A. Processor Fault Coverage

The two targeted processors and the test shell shown in Fig. 2 were synthesized using the TSMC 0.35- and 0.18-μm libraries, respectively. The test shell has a gate count of 1938 and a critical path delay of 0.7 ns for TSMC 0.35-μm technology and 1766 gates and 0.3 ns delay for 0.18 μm. Clearly the test shell introduces negligible area overhead. The critical paths of the test shell are only exercised during random program testing. For normal operation the delay can be at most a multiplexer delay which can be further reduced by integrating the test shell into the AMBA wrapper of the processor. Therefore the performance impact of the test shell should be much smaller than scan design that is required for each stage of the pipeline.

Table I shows the fault coverage of the ARMv4 and miniMIPS processors under the micro observation method with various test programs and synthesis cases. The second column shows the processor gate count and stuck-at fault numbers. Depending on the cell-based library used and the experimental conditions, the same processor core can have different synthesized gate-level results, and thus, a different number of faults. The third column shows the processor fault coverage obtained using our previous deterministic programs [1], which ranges from 92.62% to 93.72%. The fourth column is the processor fault coverage when running the random program only, which consists of about 1.1×10^6 instructions or 10^5 basic blocks. These blocks can roughly be split into three equal parts of in-order blocks, out-of-order blocks, and return blocks. The fifth and sixth columns show the results of the hybrid test program and scan chain, respectively.

The proposed hybrid test program obtains a full processor fault coverage that is close to the level of a full scan chain. Note that since this is at-speed testing, the processor runs the test program in processor speed. As a result, testing time is not an issue, and neither is the test program storage with the use of the ATE system board illustrated previously. The sixth column also indicates the number of test cycles for scan chain testing.

Using only one type of the basic blocks, the random program (10^5 blocks) achieves processor fault coverage of 90.3% for in-order blocks, 91.6% for out-of-order blocks, and 89.1% for return blocks. The experiments in this study indicate that the random program generating flow depicted in Fig. 1 is an effective basic block deployment strategy.

Table II shows the breakdown of processor fault coverage using an ARMv4 processor core as an example. The random test programs performed well for the decoder and the access control parts of the functional modules. The hybrid test program performed better than the scan chain method in testing the exception handling unit. This may be due to the specific design of the pipelined exception unit, which handles the exceptions of instruction abort, undefined instruction, data memory abort, and external interrupts. For precise interrupt design, the former

TABLE II
BREAKDOWN OF THE ARMv4 PROCESSOR FAULT COVERAGE (125 Mhz, 0.18 μm)

Component	Number of faults	Determinist. (2415 Inst.)	Random (1.1million)	Hybrid (1.1million)	Scan chain
Register file & access control	55780	93.59%	96.41%	97.49%	96.83%
ALU	35510	96.65%	98.58%	98.63%	98.55%
Shifter	3974	98.92%	99.92%	99.92%	99.93%
Memory access unit	16416	90.50%	94.47%	94.47%	98.57%
Instruction fetch unit	2342	79.80%	84.42%	84.59%	93.08%
Decoder	3018	87.17%	91.98%	94.07%	96.36%
Status registers & access control	7658	78.87%	92.99%	94.40%	95.48%
Coprocessor access unit	1410	80.35%	81.56%	88.79%	93.98%
Exception handling unit	292	86.64%	83.22%	89.38%	85.35%
Other	10822	91.02%	92.55%	97.38%	95.79%
ARM9-v4 compatible processor	137222	92.62%	95.43%	96.93%	
ARM9-v4 compatible processor (full scan)	145586				97.73%

two exceptions are recognized in the fetch stage and decode stage, respectively, and passed through the pipeline registers, while the latter two are directly fed into the unit for exception processing. The scan-based method is effective in testing the former two exceptions since the signals are buffered through the pipeline registers. On the other hand, the SBST method assisted by the test shell design sends test patterns and receives responses through input and output ports directly, so some more faults of the circuits without the buffered registers can be detected for the latter two cases. For some of the units, the hybrid test program has much lower coverage such as the instruction fetch unit in ARMv4 and the memory access unit in miniMIPS (not shown here). The reason for this appears to be the abundant address space which may not be fully explored by the test program, while the scan chain method can generate much more effective test patterns using automatic test pattern generation (ATPG) in these cases.

B. Impact of Random Test Program Size and Observation Methods

We also study the effect of random test program size and fault observation methods on processor fault coverage. Table III lists the fault coverage of various test programs with three observation methods, observing doubled edges and the positive edge both with MISR, and the data written in the memory. First, we observe that increasing the random test program size above $55\,000 \times 20$ (about 1.1×10^6 instructions) produces relatively small improvements in fault coverage and thus results of program size beyond that are not reported. Both MISR observation methods obtain the same fault coverage since the clocking methodology of the target processors used triggers at the rising edge, and thus faults can only propagate at the rising edge. The macro observation method, which tries to emulate the observation method based on the written results of the test program, performs relatively poorly in fault coverage measurement. Since only those hardware signal cycles that are related to the correctness of the written results are observed, the fault coverage obtained using this method is 4%–8% less than that provided by the other two models. These results indicate that observing the written results from the memory used by an SBST test program is not sufficient for fault coverage assessment since many faults simply "slip away" without notice. In other words, a full-spectrum hardware response observation method is required, such as the MISR method.

TABLE III
FAULT COVERAGE OF THE TEST PROGRAMS USING DIFFERENCE OBSERVATION METHODS (ARMv4 ISA PROCESSOR, 125 MHz)

Test program	Instruction count	Doubled edge (MISR, micro)	Positive edge (MISR,micro)	Written data (macro)
Deterministic	2415	92.62%	92.62%	85.34%
Random	55,000	93.99%	93.99%	89.78%
Random	55,000 × 5	94.99%	94.99%	90.63%
Random	55,000 × 10	95.34%	95.34%	91.05%
Random	55,000 × 20	95.43%	95.43%	91.19%
Hybrid	55,000	96.38%	96.38%	92.09%
Hybrid	55,000 × 5	96.73%	96.73%	92.62%
Hybrid	55,000 × 10	96.85%	96.85%	92.74%
Hybrid	55,000 × 20	96.93%	96.93%	92.81%

TABLE IV
COMPARISONS OF VARIOUS SBST METHODS

	CPU	Methodology	Gate count/ number of faults	F.C.%
[2]	OpenRISC 1200 (Soft core)	Hybrid	44476 + 2021 state elements / 186209	92.30
[4]	MiniMIPS (Soft core)	Deterministic	32,817/N.A.	95.08
[4]	OpenRISC (Soft core)	Deterministic	35,657/N.A.	90.03
[5]	16-bit DLX (Soft core)	Random + DFT hardware	27,860/43,927	92.50 94.85
[7]	EX1 module of Xtensa processor	Deterministic	N.A./ 24,962	95.20
[10]	ARM920T (Hard core)	Functional testing + DFT hardware	N.A./ N.A.	90.00
Our work 1	ARMv4 ISA (Soft core)	Hybrid	45,960/137222 (0.18 u/ 125MHz)	96.93
			63,281/152758 (0.35u/ 50MHz)	98.04
Our work 2	MiniMIPS (Soft core)	Hybrid	50,325/150474 (0.18 u/ 125MHz)	98.13
			58,540/141668 (0.35u/ 50MHz)	98.46

C. Comparisons With Other Works

Table IV compares the processor fault coverage of various SBST testing methodologies. In [4], the proposed SBST method focuses on the pipeline logic and automation of test program generation. In [7], the study only shows the fault coverage based on functionally testable faults for a large logic module, called EX1, which was extracted from the Xtensa processor. In [10], the ARM processors were tested using functional testing approach. Table IV shows that the proposed hybrid test program clearly presents itself as a viable approach for practical use either as an effective alternative or enhancement for the manufacturing test of microprocessors.

D. Application and Limitation

The automation of SBST has been one of the most important long-term goals for processor testing. To help achieve this goal, this paper presents a basic-block based random test program generation method that contributes to the automation effort. Test programs like the deterministic program can be developed ahead and included in a test code template library to help generate and evaluate a useful hybrid test program. In contemporary SoC chips, the ARM and MIPS CPU are widely used processor cores which are often implemented in a pipeline fashion with in-order execution model for embedded applications. Since the proposed test programs work well on both processors, they should be useful in similar pipeline processor cores. For superscalar processors with out-of-order feature, VLIW processors, or larger processors like those used in desktop applications, this can be an interesting study for SBST methods but it is beyond the scope of this paper.

V. Conclusion

This paper presents a high-performance hybrid test program for software-based self-testing of pipeline processor cores. Experiments on two complex real-life pipeline processors with different gate-level implementations show that the random test program and the deterministic test program can nicely compensate for each other for fault detection. Used together, the hybrid test program achieves good processor fault coverage of more than 98%. This study also develops a test shell to cope with random program execution and interrupt testing. To clarify fault coverage evaluation, this study also presents the concept of micro observation versus macro observation for test responses, showing that the most effective method used for SBST is through a MISR connected to the local bus of the processor.

References

[1] C.-H. Chen, C.-K. Wei, T.-H. Lu, and H.-W. Gao, "Software-based self-testing with multiple-level abstractions for soft processor core," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 15, no. 5, pp. 505–517, May 2007.

[2] N. Kranitis, A. Merentitis, and D. Gizopoulos, "Hybrid-SBST methodology for efficient testing of processor cores," *IEEE Des. Test Comput.*, vol. 25, no. 1, pp. 64–75, Jan./Feb. 2008.

[3] N. Kranitis, A. Paschalis, D. Gizopoulos, and Y. Zorian, "Effective software self-test methodology for processor cores," in *Proc. Des. Autom. Test Eur.*, 2002, pp. 592–597.

[4] D. Gizopoulos, M. Psarakis, M. Hatzimihail, M. Maniatakos, A. Paschalis, A. Raghunathan, and S. Ravi, "Systematic software-based self-test for pipelined processors," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 16, no. 11, pp. 1441–1453, Nov. 2008.

[5] K. Batcher and C. Papachristou, "Instruction randomization self test for processor cores," in *Proc. 17th IEEE VLSI Test Symp.*, 1999, pp. 34–40.

[6] P. Parvathala, K. Maneparambil, and W. Lindsay, "FRITS-A microprocessor functional BIST method," in *Proc. Int. Test Conf.*, 2002, pp. 590–598.

[7] L. Chen, S. Ravi, A. Raghunathan, and S. Dey, "A scalable software-based self-test methodology for programmable processors," in *Proc. 40th Des. Autom. Conf.*, Jun. 2003, pp. 548–553.

[8] I. Bayraktaroglu, J. Hunt, and D. Watkins, "Cache resident functional microprocessor testing: Avoiding high speed IO issues," in *Proc. Int. Test Conf.*, 2006, pp. 27–34.

[9] OPENCORES.ORG, "miniMIPS CPU," 2008. [Online]. Available: www.opencores.org/projects/minimips

[10] A. Burdass, G. Campbell, and R. Grisenthwaite, "Embedded test and debug of full custom and synthesisable microprocessor cores," in *Proc. IEEE Eur. Test Workshop*, 2000, pp. 17–22.

[11] Y.-C. Lin, Y.-Y. Tsai, K.-J. Lee, C.-W. Yen, and C.-H. Chen, "A software-based test methodology for direct-mapped data cache," presented at the IEEE 17th Asian Test Symp. (ATS), Sapporo, Japan, Nov. 2008.

[12] Mentor Graphics, USA, "ModelSim SE," version 6.3c, 2007. [Online]. Available: http://www.model.com/

[13] Synopsys Inc., San Jose, CA, "Design Analyzer," version X-2005.09, 2006. [Online]. Available: http://www.synopsys.com/

[14] SynTest Technologies Inc., USA, "TubroScan," version 2.8, 2007. [Online]. Available: http://www.syntest.com/