Instruction-Level DFT for Testing Processor and IP Cores in System-on-a-Chip

Wei-Cheng Lai, Kwang-Ting (Tim) Cheng
Department of ECE, University of California, Santa Barbara, CA 93106
E-mail: {wlai, timcheng} @yellowstone.ece.ucsb.edu

Abstract

Self-testing manufacturing defects in a system-on-a-chip (SOC) by running test programs using a programmable core has several potential benefits including, at-speed testing, low DfT overhead due to elimination of dedicated test circuitry and better power and thermal management during testing. However, such a self-test strategy might require a lengthy test program and might not achieve a high enough fault coverage. We propose a DfT methodology to improve the fault coverage and reduce the test program length, by adding test instructions to an on-chip programmable core such as a microprocessor core. This paper discusses a method of identifying effective test instructions which could result in highest benefits with low area/performance overhead. The experimental results show that with the added test instructions, a complete fault coverage for testable path delay faults can be achieved with a greater than 20% reduction in the program size and the program runtime, as compared to the case without instruction-level DfT.

1. Introduction

A system-on-a-chip (SOC) device usually contains one or more programmable cores (such as processor cores and DSP cores) and uses buses to connect various programmable and non-programmable IP cores. One possible test strategy for a SOC is to utilize the on-chip programmable cores to test the manufacturing defects on the SOC. Under this test strategy, we view test as an application of a programmable SOC, which reuses on-chip resources for test purpose. This strategy minimizes the addition of dedicated test circuitry for DfT or self-test. We refer to this self-test strategy as *functional self-test* or *embedded-software-based self-test* [1][2].

For high-speed circuits, self-testing has clear advantages over testing relying on external testers. On-chip clock speed is increasing dramatically while the tester's Overall Timing Accuracy (OTA) is not. This trend implies an increasing yield loss due to external testing since guard-banding to cover tester errors results in loss of more and more good chips [3]. Self-testing offers the ability to apply and analyze at-speed test signals on chip with greater accuracy than that available on the tester.

Pure embedded-software-based self-testing may not achieve a desired level of fault coverage. Furthermore, the size of the test program may be too large to fit in on-chip memory. The total test application time may also be too long. The low controllability and observability of some wires and registers in an SOC is the key reason for such problems. In this paper, we propose a DfT methodology to improve the test quality of embedded-software-based self-testing by adding a small number of test instructions to enhance the testability of a processor core. We call this methodology as *instruction-level DfT.*

Instruction-level DfT, which inserts test circuitry in the form of test instructions, should be a less intrusive approach as compared to the gate-level DfT technique which attempts to create a separate test mode somewhat orthogonal to the functional mode. If the test instructions are carefully designed such that their micro-instructions reuse the datapath for the functional instructions and do not require any new datapath, the overhead, which only occurs in the controller, should be relatively low. This methodology is also more attractive for applying at-speed test and for power/thermal management during test, as compared to the existing logic BIST approaches. To apply at-speed tests, existing structural logic BIST needs to resolve complex timing issues related to multiple clock domains, multiple frequencies and test clock skews. In contrast, self-testing the devices using instruction sequences allows much more natural application of at-speed tests. At-speed tests are applied by executing instruction sequences that are designed to achieve high path or gate delay fault coverages. Moreover, structural logic BIST applies non-functional, high-switching random patterns thus causes much higher power consumption than normal system operation. Self-testing the devices using the instruction set of processor cores can alleviate such problems.

This work was supported in part by the MARCO/DARPA Gigascale Silicon Research Center (http://gigascale.eecs.berkeley.edu). Their support is gratefully acknowledged.

DAC 2001, June 18-22, 2001, Las Vegas, Nevada, USA.
Copyright 2001 ACM 1-58113-297-2/01/0006.... $5.00

A number of approaches [2][4][5][6][7][8][9] have been proposed to generate a test program to self-test a microprocessor for either stuck-at or delay faults. Shen and Abraham [7] propose an approach for improving the test quality by adding instructions to control the exception circuitry to the processor such as interrupts and reset. With the new instructions, the test program can achieve fault coverage between 87% and 90% for stuck-at faults. This approach cannot achieve a higher coverage because their test program, which is synthesized based on a random approach, is not able to effectively control or observe some internal registers which have low testability.

In this paper, we propose a DfT methodology which systematically adds test instructions to an on-chip processor core. The new instructions can improve the testability of a processor core, reduce the size of the test program, and reduce the run time of the test program (i.e., reduce the test application time). To decide which instructions to add, we first analyze the testability of the processor. If a register in the processor is identified as hard-to-access, we add a test instruction to access the register directly. In addition, we observe that, in the test program, some code segments appear repeatedly. We identify such frequently appeared (hot) segments and add a few test instructions to reduce the size of the hot segments. Test instructions can be added to speed up the processes of preparing the test vectors by the processor core, retrieving the responses from the on-chip core under test, and analyzing the responses (by the processor core). Our experimental results show that test instructions can reduce the program size and program running time by about 20%.

The rest of the paper is organized as follows. Section 2 illustrates the concept of embedded software tester and test program synthesis. Section 3 shows the analysis of the testability of a processor core and a synthesized test program. Section 4 focuses on the instruction-level DfT techniques. Experimental results are presented in Section 5. Section 6 concludes the paper.

2. Embedded-Software-Based Self-Test

A processor core in a SOC design can be configured as a pattern generator, a test application controller or a response analyzer simply by running different programs. For example, consider an exemplar SOC design shown in Figure 1. It has two programmable cores, a DLX processor core [10] and a DSP core. There are three on-chip cores: a memory core, core A and core B. All cores are connected by a PCI bus. The Virtual Component Interface (VCI) and the PCI wrapper provide a common interface for a core to communicate with the underlying bus architecture [11][12]. Since DLX implements memory-mapped I/O, portions of the address space are pre-assigned to the non-memory cores. Therefore, DLX can send data to a core by writing the data to the corresponding address of the core.

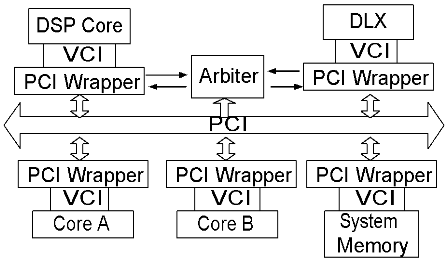


Fig. 1. A SOC design

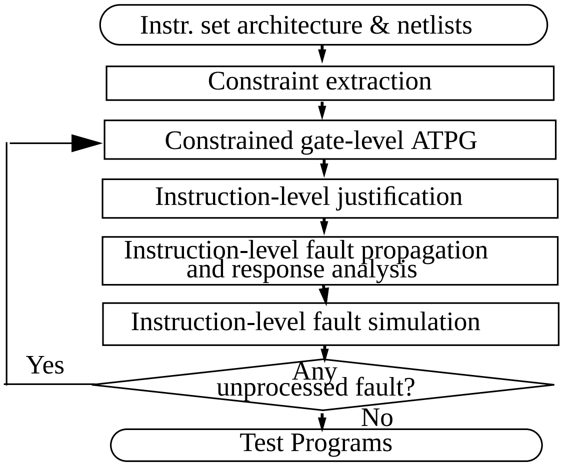

Instr. set architecture & netlists
Constraint extraction
Constrained gate-level ATPG
Instruction-level justification
Instruction-level fault propagation and response analysis
Instruction-level fault simulation
Yes
Any unprocessed fault?
No
Test Programs

Fig. 2. Test program synthesis process

To self-test the DLX core, we can first load the test program from an external tester into the on-chip memory. Then, the DLX core executes the test program at-speed. After the DLX core has been tested, we can use the DLX core to test other on-chip (non-programmable) cores by running additional test programs. Here, it is assumed that each PCI wrapper implements a scan buffer which is connected to the scan chain of the IP core. Each PCI wrapper also has a mode register which can set the core in test or functional mode. The scan buffer and the mode register are all memory-mapped. With this hardware support, the DLX processor can use normal memory read/write operations to configure the core in test (or normal) mode, send scan vectors to the core, and read responses back for analysis.

Figure 2 shows the general flow of synthesizing a test program for testing a processor or an IP core. The detailed description of a test program synthesis (TPS) algorithm can be found in [2]. Given the instruction set architecture, the netlist of the processor core and the netlists of the on-chip cores, the TPS algorithm first extracts a set of constraints capturing correlations among input/output (I/O) signals and registers/flip-flops of the processor. These constraints are used in the subsequent gate-level ATPG process to rule out those test vectors which cannot be produced in the functional mode. Then, a constrained structual gate-level ATPG (for stuck-at or delay faults) is used to generate deterministic tests for a target fault. The generated test vector, which meets the imposed constraints, specifies required values at the inputs and the registers/flip-flops. Next, an instruction-level justification process synthesizes a sequence of instructions which bring the circuit to the state required by the test vector. In the next step, the instruction-level response analysis process synthesizes a sequence of instructions to propagate the fault effects in registers/flip-flops to memory and possibly further compress them into signatures. The above procedure is repeated until all faults have been examined.

3. Testability Analysis and Test Program Analysis

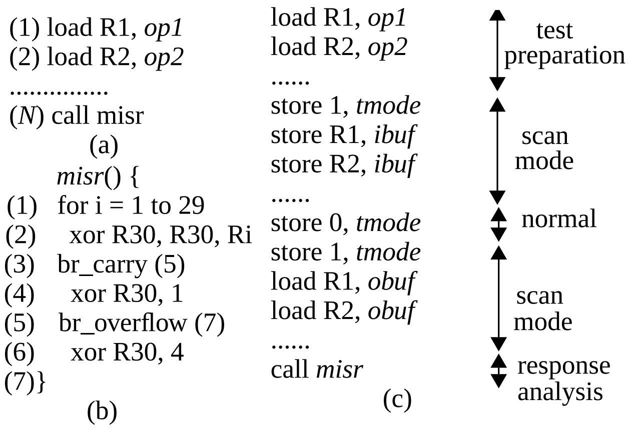
To identify good candidate test instructions, we first apply testability analysis to the processor. For registers and exception circuitry which have low accessibility, test instructions are added to increase their accessibility. We further analyze the synthesized test program. It is observed that many program segments appear repeatedly in the test program. We can add test instructions to transform those repeated code segments into smaller and faster code segments.

3.1. Testability analysis of a microprocessor core

In general, instructions can be classified into three categories: 1) data movement instructions, 2) ALU instructions and 3) branch instructions. Data movement instructions move data from memory to register (load), register to memory (store), and register to register (move). ALU instructions such as addition and subtraction, perform arithmetic and logical operations on operands. Branch instructions such as *jump* and *conditional jump* transfer the program control to a target address specified in the instruction operand.

We determine the testability of a register based on the availability of data movement instructions between registers and memory. We define a register as *fully controllable* if there exists a sequence of data movement instructions which can move the desired data from memory to the register. Similarly, we define a register as *fully observable* if there exists a sequence of data movement instructions to propagate the register data to memory. Given the micro-architecture of a processor core, we can identify those registers which are fully controllable or fully observable. For registers not fully controllable/observable, new instructions can be added to improve their accessibility.

For example, general purpose registers are fully controllable and fully observable since a load/store instruction can move data between the registers and memory. Another examples of fully controllable registers are program counter (PC) and memory address register (MAR) since we can use a "jump" instruction to access them. On the other hand, status registers (SR) have poor controllability because setting up the desired data in SR usually requires specific combination of instruction and operand sequences. The observability of status registers is generally poor because only conditional branches can propagate the errors in the status register to data registers and memory. Registers buried deeply inside the pipeline may have accessibility problems as well. To set up the desired data in these registers, it may be necessary to justify them through long pipeline stages until it reach a fully-controllable register. This justification process could be very complicated and thus, slow down the test program generation process. Therefore, we can add test instructions to directly load data into these registers.



```
(1) load R1, op1
(2) load R2, op2
...............
(N) call misr
```

(a)

```
      misr() {
(1)   for i = 1 to 29
(2)     xor R30, R30, Ri
(3)   br_carry (5)
(4)     xor R30, 1
(5)   br_overflow (7)
(6)     xor R30, 4
(7)}
```

(b)

```
load R1, op1
load R2, op2
......
store 1, tmode
store R1, ibuf
store R2, ibuf
......
store 0, tmode
store 1, tmode
load R1, obuf
load R2, obuf
......
call misr
```

(c)

Fig. 3. Common structure in test programs

The exception circuitry (e.g., interrupt and reset circuitry) of a processor receives signals from external devices. The signals in this circuitry cannot be directly controlled by any instruction which could result in low fault coverage. We can add test instructions to improve the testability of such circuitry.

3.2. Analysis of a synthesized test program

A test instruction can be added to optimize the test program in terms of program size and program run time. We try to identify repeated common segments in the test program and make these segments as short and efficient as possible using test instructions. Since the TPS algorithm in Figure 2 iteratively synthesizes the code for each fault, the resulting test program shows many similar code structures. These common code structures are good candidates for optimization using test instructions.

Figure 3(a) shows an example of a common code structure in the test program for testing a fault inside the processor. First, the program requires two load operations to read the desired operands *op1* and *op2* into the CPU registers. It applies a sequence of instructions (not shown in the figure) to activate the fault and the responses are captured in the CPU registers. At the end, a response analysis subroutine *misr* (shown in Figure 3(b)) is invoked. The *misr* subroutine computes the response signature (always stored at R30) by applying exclusive-or operations iteratively to values in all registers including the status register (SR). Since the values in SR can only be observed using conditional branch instructions, *misr* needs a sequence of branch statements (e.g., line 3 to line 6 in Figure 3(b)) to retrieve the data from SR. For example, at line 3, the branch instruction will jump to line 6 if the carry bit in SR is logic one.

Figure 3(c) shows a common program structure for testing a fault in an on-chip (non-programmable) IP core assuming the IP core has fully-scanned. It consists of five steps: (1) In the test preparation phase, the desired scan vectors (e.g., *op1* and *op2*) are retrieved from memory into the CPU registers (e.g., R1 and R2) using a sequence of load instructions. (2) The CPU configures the IP core to scan mode by writing a logic one into a memory-mapped core register "*tmode*". Then the CPU starts sending the scan vectors from the registers into the scan input registers "*ibuf*" of the core under test using store operations. The scan input registers are used to alleviate the speed gap among the CPU, bus, and the IP core. Data on these buffers is shifted serially into the scan chain of the IP core during the scan mode. Similarly, there are scan output buffers "*obuf*" which can receive output responses shifted out from the scan chain. Both buffers (*ibuf* and *obuf)* are mapped to memory addresses. (3) The CPU sets the IP core in the normal mode for one clock cycle by writing a logic zero to register *tmode.* The responses are captured in the scan chain. (4) The CPU starts loading the responses from *obuf* to CPU registers. (5) A response analysis subroutine (i.e., *misr*) is invoked to analyze the responses.

As it can be observed, the test programs in Figure 3 execute a lot of consecutive loading instructions to move a set of data from memory to CPU registers. Therefore, we can add a new instruction to speed up these loading operations. We also observe that the response analysis subroutine is the most frequently visited code segment. Therefore, we can use a test instruction to optimize the response analysis subroutine to reduce the program run time.

4. Instruction-level DfT

In adding new instructions, the existing hardware should be "reused" as much as possible. To reduce the area overhead, we should avoid adding extra buses or extra registers while implementing a new instruction. In fact, in most cases, a new instruction can be added by introducing new control signals to the datapath without adding extra hardware to the datapath.

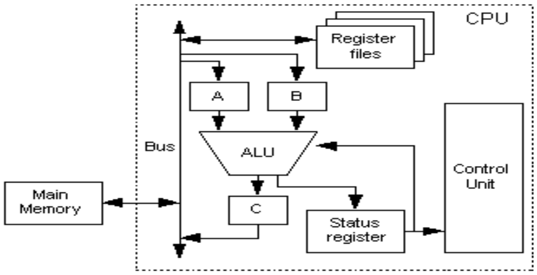


Fig. 4. An example processor

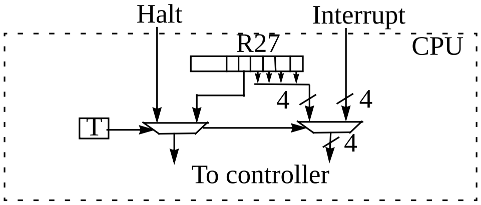


Fig. 5. DfT for exception circuitry

Figure 4 shows an example processor core. It consists of an ALU, a register file, a status register (SR) and a controller. After the testability analysis, we find that the status registers have low controllability and low observability. In addition, exception circuitry such as interrupt vectors, halt, and reset are very difficult to control. We add the following instructions to improve the testability of the processor:

(1) *Move SR to Rn* (*s2r*): This instruction can move the data from the status register to any general purpose register (Rn). Data in SR are propagated through an existing data path from SR to ALU, to register C, to the target register Rn. This instruction improves the observability of the status register and thus, can simplify the instruction-level fault propagation process.

(2) *Move A to SR* (*r2s*): This instruction can move the data from a general purpose register A to the status register. Similarly, we reuse an existing data path (register A, to ALU, to SR) to load the values from register A to SR. This instruction improves the controllability of the status register. It can be used to simplify the instruction-level justification process.

(3) *Read exception signals from register Rn*: This instruction allows the processor to take the exception signals from a general purpose register rather than from external devices. A DfT for exception circuitry has been proposed in [7]. However, this approach does not consider reusing any existing hardware on the chip. In contrast, our DfT method try to reuse the existing hardware as much as possible. The DfT architecture that we propose for handling the exception circuitry is shown in Figure 5. Here, without loss of generality, we select R27 as the register which provides an alternative source of exception signals to the controller. By controlling the register T, which is a 1-bit register, we can select which signals should be fed to the controller. We can use a data movement instruction to set the desired values in R27 before switching the exception signal sources from the external devices to R27. In this approach, we only need to add one extra instruction which can write

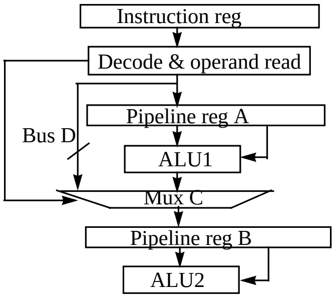


Fig. 6. DfT for pipelined design

value into register T. Adding this instruction will allow self-testing of faults in the exception circuitry.

In Figure 3, we have shown some common program segments in a test program. To reduce the program size and improve the program run time, we can add the following instructions:

(1) *Consecutive load to R_i and R_j* (*load2*): This instruction can read two (or more) consecutive words from a memory address (stored in another register R_k) and load them into registers R_i and R_j, respectively. A consecutive load needs three words in memory (one for the instruction itself and two for the operands). In contrast, two load instructions require four words (two for the load instructions themselves and two for the operands). By replacing two load instructions with a consecutive load, the processor retrieves fewer words from memory and thus, reduces the program size and run time. For example, in Figure 3(a) and Figure 3(c), we can replace the code segments that have two load operations with a consecutive load operation. Therefore, the data retrieval time for memory-to-processor transmissions and core-to-processor transmissions is reduced.

(2) *Signature computation* (*xor_all*): To improve the runtime performance of the signature computation subroutine, we can add an instruction which performs a sequence of exclusive-or operations on all CPU registers. For the example shown in Figure 4, we can add an instruction which iteratively moves data from a general-purpose register to register B, performs an xor operation at ALU, and forwards the results latched in register C to register A until all registers are processed. Note that replacing a sequence of xor instructions in the response analysis subroutine with *xor_all,* which helps reduce the runtime, does not significantly reduce the size of the test program. This is because the test program contains only one copy of the signature analysis subroutine.

Adding test instructions to the programmable core does not improve the testability of other non-programmable cores. Therefore, instruction-level DfT cannot increase the fault coverage of the non-programmable cores. However, we can use the consecutive load instruction (*load2*) and signature computation instruction (*xor_all*) to optimize the test programs for testing the non-programmable cores. In other words, we reuse the same set of test instructions added for self-testing the programmable cores to reduce the size and run time of the test programs for testing other non-programmable cores.

For pipelined designs, we can also add instructions to control the registers buried deeply in the pipelines. Figure 6 shows the structure of a pipelined design. Suppose the pipeline register B is very difficult to control. We can add a test instruction, an extra bus (bus D), a mux (mux C), and a mux control signal to enable loading data directly from a general-purpose register to register B. When the test instruction is decoded and its operands are available on bus D, the test instruction will enable mux C to select bus D as the signal sources for the pipeline register B.

It is not necessary to add test instructions to control every pipeline register. This is because some pipeline registers are relatively easy to control using instructions. For example, in Figure 6, after an instruction is decoded, the operands will be latched by pipeline register A. We can set up the desired values in register A by controlling the operand values of the instruction. Thus, there is no need to add a test instruction to control register A.

5. Experimental results

We have applied our method to two simple microprocessor cores: Parwan processor [13] and DLX processor [14]. The implementations of both processors are non-pipelined. Parwan is an 8-bit processor with 1,810 gates. DLX is a 32-bit processor with 18,865 gates.

We apply our methodology to both processors. For PARWAN, we add two instructions for reading and writing its status registers and one instruction for fast computation of signatures. For DLX, we also add two instructions for reading and writing its status register, a signature computation instruction and a consecutive read instruction based on the analysis of the synthesized program.

Table 1 compares the test programs synthesized without test instructions [2] against those with test instructions. All test programs target path delay faults. We show the test program length (in bytes), the execution time of the test program (in clock cycles), the fault coverage for testable path delay faults, the area (in 2-input NAND gate equivalents), and the test generation time for both approaches. We also show the reduction ratio (in parenthesis) in term of the program length, program run time and the test generation time. For example, for the DLX processor core, our approach reduces the program size by 15%, reduces the run time by 21%, improves the fault coverage to 100%, reduces

Table 1.
Results of the test programs for testing processors

		prog. len (bytes)	run time (cycles)	coverage%	Area	CPU (s)
PARWAN	[2]	12,586	78,386	99.8	1,729	1.76
	DfT	8,333 (-34%)	47,601 (-39%)	100	1,810 (+ 4.7%)	1.21 (-31%)
DLX	[2]	141,776	463,185	96.3	18,865	203
	DfT	120,232 (-15%)	367,237 (-21%)	100	19,165 (+1.6%)	123 (-39%)

Table 2.
Results for test programs for a DLX core to test the ISCAS-89 cores

		# test vectors	prog. len (bytes)	run time (cycles)	CPU (s)
s1238	w/o DfT	1220	48,992	125,926	0.61
	DfT	1220	39,184 (-20%)	94,154 (-24%)	0.48 (-21%)
s5378	w/o DfT	952	152,432	229,492	2.11
	DfT	952	110,512 (-27%)	149,524 (-34%)	1.29 (-38%)
s9234	w/o DfT	244	48,912	70,088	0.64
	DfT	244	36,192(-26%)	45,688 (-35%)	0.39 (-39%)
s38584	w/o DfT	37382	35,139,192	42,540,776	524
	DfT	37382	27,812,288 (-21%)	30,429,008 (-28%)	306 (-42%)

the test generation time by 39% and increases the area by 1.6% when compared to the results in [2].

We synthesize several test programs for the DLX core to test other on-chip (non-programmable) cores based on the test delivery mechanism in Section 2 (the details of the mechanism can be found in [15]). Under this mechanism, the processor core supplies the scan vectors to the on-chip core simply by issuing memory read operations. We use ISCAS-89 benchmark circuits as the on-chip cores and synthesize a test program to apply the delay test vectors derived in [16]. Table 2 shows the characteristics of the test programs with and without the support of test instructions. The third column shows the number of test vectors (including the scan vectors and the input vectors) applied to the cores. The last column shows the test program generation time in seconds. In these experiments, we use the same set of test instructions added for self-testing the DLX core to help prepare the test vectors in DLX, retrieve responses from the benchmark cores and analyze the responses in DLX. For s9234, without the instruction-level DfT support, a test program with 48,912 bytes can deliver 244 test vectors to the s9234 core in 70,088 clock cycles. With the test instructions, the same amount of scan vectors can be delivered and analyzed using a program with 36,192 bytes. This program can be completed by DLX in 45,688 clock cycles. On average, the added test instructions can reduce the program size by 23% and program run time by 29%.

6. Conclusions

In this paper, we present an instruction-level DfT methodology by adding new instructions to an on-chip microprocessor core. With the added test instructions, embedded-software-based self-testing can achieve a higher fault coverage, shorter test generation time and smaller and faster test programs with a very low area overhead. Our experimental results show that the proposed DfT methodology can reduce both the program length and the program run time by 20% at the cost of 1.6% area overhead for a couple of example processor cores.

7. References

[1] W.-C. Lai, A. Krstic, and K.-T. Cheng. On Testing the Path Delay Faults of a Microprocessor Using its Instruction Set. *VLSI Test Symp.*, pages 15-20, 2000.

[2] W.-C. Lai, A. Krstic, and K.-T. Cheng. Test Program Synthesis for Path Delay Faults in Microprocessor. *Proceedings of ITC*, pages 1080-1089, 2000.

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

[4] L. Chen and S. Dey. DEFUSE: A Deterministic Functional Self-Test Methodology for Processors, *IEEE VLSI Test Symp*.pp. 255-262, May 2000.

[5] D. Brahme and J.A. Abraham. Functional Testing of Microprocessors. *IEEE Transactions on Computers*, vol. C-33, pages. 475-485, 1984.

[6] F. Distante and V. Piuri. Optimum Behavioral Test Procedure for VLSI Devices: A Simulated Annealing Approach. *Proceedings of the IEEE International Conference on Computer Design*, pages 31-35, 1986.

[7] J. Shen and J.A. Abraham. Native Mode Functional Test Generation for Processors with Applications to Self Test and Design Validation. *Proceedings of International Test Conference*, pages 990-999, 1998.

[8] K. Batcher and C.A. Papachristou. Instruction Randomization Self Test For Processor Cores. *VLSI Test Symposium*, pages 34-40, 1999.

[9] J. Lee and J.H. Patel. Architectural Level Test Generation for Microprocessors. *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems*, 13(10):1288-1300, October 1994.

[10] D. A. Patterson and J. L. Hennessy. *Computer Organization & Design: the Hardware/Software Interface*, Morgan Kaufmann, San Mateo, California, 1994.

[11] PCI Special Interest Group, *PCI Local Bus Specification, Revision 2.2*, Porland, Oregon, Dec. 1998.

[12] Virtual Socket Interface Alliance. *VSI Alliance Virtual Component Interface Standard(OCB 2 1.0)*. March, 2000.

[13] Z. Navabi, *VHDL: Analysis and Modeling of Digital Systems. McGraw-Hill*, New York, December 1997.

[14] M. Gumm. *VLSI Design Course: VHDL-Modelling and Synthesis of the DLXS RISC Processor.* University of Stuttgart, Germany, December 1995.

[15] J.-R. Huang, M. K. Iyer, and K.-T. Cheng. A Self-Test Methodology for IP Cores in Bus-based Programmable System-on-a-chip. *VLSI Test Symp.,* 2001.

[16] K.-T. Cheng, S. Devadas, and K. Keutzer. Delay-Fault Test Generation and Synthesis for Testability Under a Standard Scan Design Methodology. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(8):1217-1231, August 1993.