Software-Based Self-Test of Processors under Power Constraints

Jun Zhou, Hans-Joachim Wunderlich

Institute of Computer Architecture and Computer Engineering, University of Stuttgart
Pfaffenwaldring 47, D-70569 Stuttgart, Germany

Abstract

Software-based self-test (SBST) of processors offers many benefits, such as dispense with expensive test equipments, test execution during maintenance and in the field or initialization tests for the whole system. In this paper, for the first time a structural SBST methodology is proposed which optimizes energy, average power consumption, test length and fault coverage at the same time.

Key words: Test program generation, processor test, low power test

1 Introduction

The principle of the SBST involves generation, storage and execution of a test program. Its main advantages are reusability at all the stages of the system life cycle, dispense with expensive test equipment, and low design for testability (DfT) overhead.

Originally, SBST was implemented as a purely functional test, which did not consider the structure of the core under test (CUT) but relied merely on the instruction set architecture (ISA) [1, 2]. It was mainly applied to systems with discrete components. However, a structural fault model was not explored, and test quality could not be assessed sufficiently. The advent of embedded processors and core-based designs exhibits more options. With the availability of the structure of a processor, structural automatic test pattern generation (ATPG) can be combined with SBST. Thus, either core vendors may deliver test programs validated by fault simulation, or users of soft cores may generate such programs.

SBST techniques based on emulated LFSR schemes were proposed and analyzed in [3]. In [4], a method for translating module level structural test patterns into processor level instructions was presented, and appropriate control and observation sequences were generated. In [5] a method for test program synthesis maximizing structural coverage was proposed. Recently, Li et al. introduced a deterministic functional self-test scheme based on emulated LFSRs [6, 7]. The previous work was improved leading to shorter test times and less memory requirement in [8]. Corno et al. presented a method for test program synthesis based on genetic algorithms [9, 10]. Most of the previous work targeted stuck-at faults, only recently were SBST schemes developed for detecting delay faults, e.g. [11, 12].

Structural testing maximizes switching activities within a circuit, which leads to an increased power consumption. Special care has to be taken in order to avoid reliability problems, reduced yield or even overheating. As a result, numerous DfT schemes are proposed to limit test power dissipation [13, 14, 15]. Without circuit modification, power can also be reduced through an appropriate test scheduling, proper test pattern optimization or system frequency decrease. The SBST operates under the functional mode so that it meets the peak power specification of the system. Nevertheless, it is of great significance for reliability reasons to optimize programs causing above-average switching activities. If such a test is deployed autonomously, not only average power but also energy consumption has to be optimized.

On the other hand, there are many research efforts on compiler techniques, which optimize code to restrict or minimize power [16, 17]. These techniques are inadequate for test programs, as they are implemented to retain program semantics. Program semantics is only subordinate from a test viewpoint where the main objective is to ensure or even maximize fault coverage, while power is reduced. For the first time, the presented paper combines structural SBST and software optimization for testing processors under power constraints. A novel method for test program synthesis is proposed which tackles fault coverage, test length, energy and average power consumption at the same time.

We arrange the rest of this paper as follows. The next section presents an overview of the methodology and describes the method of automatic test program generation. Section 3 details an algorithm on software power optimization which does not affect fault coverage. We apply the approach to a 32-bit RISC processor in Section 4. A short conclusion is provided in Section 5.

2 Test program synthesis

The overall methodology contains two procedures. First, based on the gate-level structure of a target processor, a test program is generated to provide maximal fault coverage with a short test length and low switching activities. Afterwards, the instruction sequence is optimized in order to minimize power consumption without sacrificing fault coverage. Though the stuck-at fault model is adopted in this paper, the method itself is applicable for any other combinational fault models, e.g. delay faults which require test pairs or sequences.

Fig. 1 elaborates the steps of test program synthesis. A standard ATPG generates structural test patterns for combinational modules, which are subject to a compaction process. The greedy algorithm takes fault coverage and correlation into account: at each time a pattern is selected which detects the largest number of faults yet undetected. If more alternatives are available, the one with the highest correlation measured by bit transitions is picked. The compaction ends once the fault coverage reaches that of the original test set. This step results in both test data reduction and potential energy and power savings without loss of fault coverage. In the end, the test set is mapped into instruction sequences using a template-based test program synthesis.

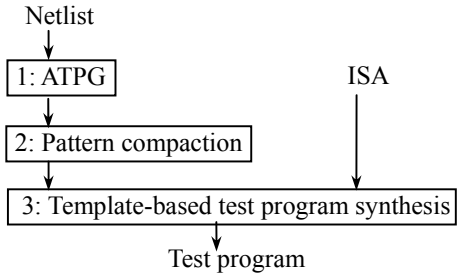

Fig. 1: Methodology for test program synthesis

The synthesis targets the modules which contribute to the major part of the processor area and structural fault coverage. For a standard processor, these are the arithmetic logic unit (ALU), the register file (RF) and the program counter (PC), as part of the control unit. We leave out test generation for the other components of the control unit which constitute deeply sequential logic and sufficient fault coverage for these parts requires special design for testability means, e.g. [18]. In the following paragraphs, we explain the essential code and the template structures similar to [7].

ALU:

Structural patterns for the ALU can be directly parsed into equivalent instructions. Suppose an ALU with two 32-bit inputs and 3 control bits leads to the format (A:<1:32>, B:<33:64>, S:<65:67>). Following this format, we extract an instruction whose operands are defined by the first 64 bits and the opcode is specified by the last 3 bits. Results are saved in general-purpose registers and fully observable. However, observing status signals needs particular instructions. For instance, instruction "JZ" (jump-if-zero) is able to access the status signal "Z", which indicates whether or not the relevant input is zero, and hence is used to make "Z" observable. Fig. 2 represents the common structure for applying an ALU pattern using instruction sequences with parameters in brackets.

```
1)  set      $[x], [A]          ;[A] parsed from <1:32>
2)  set      $[y], [B]          ;[B] parsed from <33:64>
3)  [op]     $[z], $[x], $[y]   ;[op] parse from <65:67>
4)  save     $[z], ([c_res])    ;make the result ob-
                                ;servable
5)  set      $[f], 0            ;set the flag
6)  jz       $[x], [status]     ;access the status sig-
                                ;nal, jump to "status" if
                                ;x is zero
7)  set      $[f], MAXINT
8)  status:
    save     $[f], ([s_res])    ;make the status ob-
                                ;servable
```

Fig. 2: Generating test instructions for the ALU

Program Counter (PC):

Program addresses, used as targets for branch instructions, are patterns for this unit. The generalized code in Fig. 3 applies a provided PC pattern for testing.

```
1)  set      $[x], [PC]         ;load the target
2)  jump     $[x]               ;apply the pattern
3)  .org     [val_pc]           ;[val_pc] parsed from the pc
                                ;pattern
    PC:
    save     $[x],([res])       ;make the target observable
```

Fig. 3: Generating test instructions for the PC

Register file (RF):

A standard memory test, for instance the modified algorithmic test sequence (MATS) designed for stuck-at faults of memories [19], is able to achieve high fault coverage for the RF. A short MATS type test is implemented this way:

1) set all the registers to 0;
2) R_0 = <01…01>;
3) execute $R_i = R_i + R_0$, where i is from 1 to (n-1);
4) run $R_i = R_i + R_i$, where i is from (n-1) to 0.

Even more complex march tests may be implemented, but we are already likely to detect many faults in the RF

as a by-product of the ALU and the PC test if we allocate registers in a uniform way.

The above mentioned tests are altogether embedded into templates, each of which applies an ALU and a PC pattern. The coverage for the RF is maximized by regulating register allocation during template instantiation. PC responses are accumulated to reduce memory accesses [20]. In the head template in Fig. 4, whose instance is at the beginning of the final program, an initial value is loaded into a register specially for the PC test. Then an ALU pattern is applied using the code conforming to the structure in Fig. 2. The last two instructions apply the next pattern to the PC.

```
.org [PC_pattern0]
PC0:
   set    $[x], [ini_pc]     ;setup the context for pc test
   … …                       ;ALU test conforming to the
                             ;structure in Fig. 2
   set    $[e], [PC1]        ;load the next pc pattern
  jump    $[e]               ;apply the pattern
```

Fig. 4: pseudo code of the head template

An intermediate template bears the similar structure as that of the head, only replacing the first instruction with the one, in our case "Add", for accumulation of PC responses. Its instances appear in the middle of the test program. Eventually, the accumulated response is stored in the instance of an tail template.

We generate the test set, containing both deterministic and random patterns, for the PC with the same size as that of the ALU. The former are designed in a way to cause transitions of the PC as many as possible, while the latter are only generated when the number of deterministic PC patterns is still less. The test program is generated by stepwise template instantiation where all the values for parametric fields are determined either by patterns or by register allocation.

3 Power optimization

The proposed method reaches high fault coverage based on knowledge about the gate level structure which can also be used for estimating switching activities of internal nodes of a circuit. If the layout is known as well, we can even take into account capacitances and specify power more accurately by weighted switching activities. In both cases we get a much better estimation of dynamic power dissipation than by merely considering transitions in registers at the program model level [16].

The test program synthesis described above considers already switching activities of the registers. Now, we tune two more factors to optimize a test program. One is the order of the instructions and the other one is defining unspecified bits in instruction words.

3.1 Test-oriented reordering

Test-oriented reordering of instructions offers more degrees of freedom than traditional optimization where program semantics plays an important role. We explain this difference with the help of the code below:

```
… …
1)   set     $0, 512
2)   set     $1, 65535
3)   add     $2, $1, $0
4)   save    $2, (result_0)
5)   set     $2, 324
6)   set     $4, 790
7)   sub     $2, $2, $4
8)   save    $2, (result_1)
… …
```

The dependence graph (DG) of the shown code in Fig. 5 (a) displays the dependencies imposed by program semantics. Since most of the order is fixed, there are only few degrees of freedom for reordering, for example exchanging instruction 1 and 2, and putting 6 somewhere before 7.

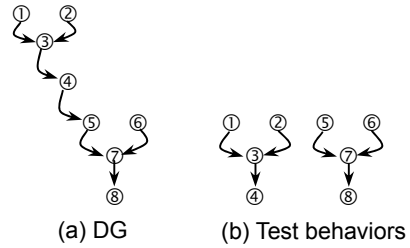

Fig. 5: program-oriented and test-oriented reordering

However, viewing the same code as part of the ALU test, we divide instructions into units, where a pattern is loaded, applied, and relevant responses are stored. Such a unit forms a test behaviour. For the above code, two test behaviours are extracted in Fig. 5 (b). Obviously, fewer restrictions exist, as reordering is possible within or even across test behaviours if no conflict is introduced.

For computing the power consumption during optimization, we have two options: simulation-integrated or model-based. The former uses a gate level power analysis tool for controlling test program optimization. This approach is computationally expensive, and as an alternative, the second method builds a reference model for a target ISA in advance [21]. Once created, the model is reusable for code optimization, and the real power savings rely much on the quality of the model.

Here we illustrate steps towards constructing the power model. We use Hamming Distances (HD) as a measurement characterizing transitions due to an instruction pair (i, j). The switching activities of the instruction register (IR), the RF and the ALU are easily predictable. For example, transitions T_{IR} at the output of IR are related to instruction coding, and those for the RF, T_{RF}, and the

ALU ,T_{ALU}, depends on runtime data. For an n-bit instruction word, T_{IR} is:

$$T_{IR}(i,j) = HD(C_i, C_j) = \sum_{m=0}^{n-1} (C_i^m \oplus C_j^m) \quad (1)$$

For a RF with two outputs, Q0 and Q1, let $Q0_i$ and $Q1_i$ be the values of output registers of instruction i, $Q0_{i\to j}$ and $Q1_{i\to j}$ be the temporary values due to (i, j). Then we model T_{RF} as:

$$T_{RF}(i,j) = HD(Q0_i, Q0_{i\to j}) + HD(Q0_{i\to j}, Q0_j) + HD(Q1_i, Q1_{i\to j}) + HD(Q1_{i\to j}, Q1_j) \quad (2)$$

Transitions of the ALU depend not only on the inputs (A, B, S) but also on the function it operates. For this reason, we consider transitions both at the inputs and the output (Q):

$$T_{ALU}(i,j) = HD(A_i, A_j) + HD(B_i, B_j) + HD(S_i, S_j) + HD(Q_i, Q_j) \quad (3)$$

Hence, the problem to model average power consumption A(i, j) due to inter instruction effects [17] for a given instruction pair (i, j) is finding relations between A(i, j) and the above depicted factors. We can base the work on *regression analysis*, which is a standard statistical method to investigate relationships between *dependent* and *independent variables*. In our case, the dependent variable is A(i, j), and independent variables are T_{IR}, T_{RF} and T_{ALU}, that is:

$$A(i,j) = f(T_{IR}, T_{RF}, T_{ALU}) + \varepsilon \quad (4)$$

ε represents approximation discrepancy. If a linear relationship is assumed, we further specify the above formula as:

$$A(i,j) = \beta_0 + \beta_1 T_{IR}(i,j) + \beta_2 T_{RF}(i,j) + \beta_3 T_{ALU}(i,j) + \varepsilon \quad (5)$$

$\beta_0, \ldots, \beta_3$ are coefficients to be determined by the process of multiple regression analysis based on samples. A sample is an observation of the dependent and the independent variables. Equation (1) to (3) facilitate calculation for the values of independent variables. According to [17, 21], we explore Equation (6) to obtain values for the dependent variable, where A_{i+j} stands for the measured average power consumption of the loop (i, j), while A_{Base_i} and A_{Base_j} are the basic average power when instruction i or j executes stand-alone:

$$A(i,j) = A_{i+j} - \frac{1}{2}(A_{Base_i} + A_{Base_j}) \quad (6)$$

The reference power model is built by the following steps below:

- simulate programs consisting loops of individual instruction to generate basic power costs A_{Base_i};
- simulate programs consisting loops of instruction pairs to get values for power costs A_{i+j};
- use Equation (1), (2), (3) and (6) to create samples by considering each possible instruction pair;
- perform multiple regression analysis to determine values for coefficients $\beta_0, \ldots, \beta_3$.

Since we work at the gate level, the model accordingly describes gate-level power consumption, offering closer estimation than the bit-transition model at the program level. A comprehensive statistical analysis for validating the model and parameter was done in [21].

3.2 Unused bit setting

Many existing publications use don't-care bits to reduce transitions in test patterns [22]. A similar idea is explored in our work at the instruction level. Fig. 6 exemplifies a specification for an Add operation in SPARC v8 [23].

Assembly Language Syntax			
add	reg_{rs1},	reg_or_imm,	reg_{rd}

Format 1:

10	rd	000000	rs1	i=0	*unused(zero)*	rs2
31	29	24	18	13	12 4	0

Format 2:

10	rd	000000	rs1	i=1	imm
31	29	24	18	13	12 0

Fig. 6: Specification of Add in SPARC v8

According to its format specification, ADD computes "r[rs1]+r[rs2]" if the 13th instruction bit is set to 0, else an immediate is used, and "r[rd]" holds the result. Every bit in Format 2 has its particular meaning, specifying either the operation code or relevant operands. On the contrary, in Format 1, 8 bits starting from Bit 5 to Bit 12 are defined as "unused" and assigned by default zeros. However, in some cases it is more power efficient, if other values are assigned to those bits. Our second optimization factor takes advantage of this observation.

Given the ISA, we are able to identify don't-cares (DCs) of each instruction under consideration. For an instruction pair (i, j), where i is fully specified, we annotate DCs of j in a way that switching activities due to (i, j) are minimized. The reference model built previously can be used once again during DC specification.

3.3 The overall optimization methodology

Before optimization starts, the raw code is partitioned into blocks, each of which is an instance of templates. Then the overall algorithm works in two phases on a block basis:

- Test-oriented greedy reordering. First, test constraints are extracted with the guidance of template structures. Then we identify unscheduled in-

structions that do not violate extracted test behaviors and for each of them, based on the power model, we compute their costs with respect to the last scheduled code. Finally the instruction with the minimal cost gets scheduled.

- DC specification for the instruction to be scheduled.

The overall process ends if all instructions are scheduled.

4 Experimental results

We apply the entire methodology to a 32-bit RISC processor [24]. It is mainly made up of the following functional components: the ALU, the RF with 32 general-purpose registers, the PC, the IR, the control unit. The design contains 14,244 logic gates including 1,088 sequential cells. Fault simulation reports 48,784 uncollapsed stuck-at faults.

The evaluation framework is an extension of [6] to support measurements of gate-level fault coverage and energy consumption in terms of switching activities. Test bench integrates the gate-level processor core and the test program binary. During simulation, two kinds of information are recorded. Primary inputs of the processor are captured and translated into patterns which are later fault simulated to evaluate the structural fault coverage. Switching activities during simulation are tracked in a file named *Value Change Dump* (VCD). Based on such file, we can used commercial tools, e.g. Primepower [25], for energy estimation, if layout information of the technology is available. As an alternative, it is straightforward to sum up recorded transitions and treat the final number as energy consumption for the simulated code directly.

Moreover, we set up our experiments as follows:

1) Perform ATPG for the ALU alone to provide the deterministic patterns. The test set achieves 99.90% fault coverage of this module;

2) Implement a MATS test for the RF, which is used together with programs generated in step 3 to 6;

3) Synthesize the test set generated in step 1 into a test program (Ori.TP), which reflects the rudimentary qualities of our algorithm;

4) Compact ALU patterns in step 1, and then synthesize the new set into a program (Compacted);

5) Reorder the code generated in step 4 under test constraints. The new code is later referred as Reordered;

6) Specify DCs of the code in step 5. The new code is termed as X-filling.

7) In the absence of program synthesis algorithms e.g. [8], we conduct a sequential ATPG for comparison. Since the fault coverage is too low to accept, a program with randomized instructions is added. The number of test cycles for them together is equal to that of Ori.TP generated in step 3.

Throughout the steps above, ATPG and program synthesis last only a few seconds, while compaction accounts for the major part of the time required. The statistics on test are reported in Tab. 1. The column of Ori.TP indicates that the test program achieves high fault coverage for the targeted modules as well as for the entire processor. Column four exhibits the identical data for the cases of Compacted, Reordered and X-filling, which imply we successfully retain the fault coverage during the two-stage optimization for low power. Another conclusion is also drawn that with pattern compaction, we reduce 30% test cycles without distinct impact on the overall fault coverage. The combination of sequential ATPG and the randomized program only yields low test quality. Sequential ATPG alone leads to 18.14% processor fault coverage with 1,457 test cycles, while the improvement by the randomized program, which contributes to a much larger portion of cycles, is not remarkable.

Tab. 1: Comparison in fault coverage and test cycles

		Ori. TP	Optimized	ATPG + randomized
Fault coverage	ALU	99.90%	99.90%	2.87%
	PC	92.38%	92.38%	66.40%
	RF	98.46%	98.03%	17.96%
	Proc.	96.70%	96.33%	19.64%
Test cycles	No.	13,886	9,628	13,886
	Ratio	1.0	0.6935	1.0

Tab. 2 details the outcome regarding energy and average power consumption. The last four rows clearly show a persistent reduction of these two parameters through every optimization step. In particular, row three indicates that compaction not only results in shortening test length, which means energy reduction as well, but also in minimizing average power dissipation by concerning switching activities of registers throughout the process. Moreover, as shown in the first row, our methodology also outperforms the combination of the sequential ATPG and the randomized program in this aspect. According to the experiment, the sequential ATPG patterns causes 1,215,280 transitions in total, which consequently bring the average power consumption (transitions per test cycle) to 834.10. Meanwhile, the randomized program consumes average power of 525.40, which is higher than any of our test programs but much lower than those structural patterns. The reason for this observation is, the structural test activates signal transitions as many as possible, while high correlation among instructions ensures much lower power consumption even for the case where random instructions with random data are used. Hence we come to the conclusion that the SBST is low power in nature.

Tab. 2: Comparison in energy and average power consumption

	Energy		Average power	
	Trans.	Ratio	Trans. per cycle	Ratio
ATPG + randomized	7,745,492	1.0	557.79	1.0
Ori. TP	6,205,081	0.8011	446.86	0.8011
Compacted	4,081,133	0.5269	423.88	0.7599
Reordered	3,904,869	0.5041	405.57	0.7271
X-filling	3,710,389	0.4790	385.37	0.6909

5 Conclusion

In this paper, we present a method to synthesize test programs, where fault coverage, test length, energy and average power dissipation are addressed at the same time. Our experimental results indicate that this methodology achieves high structural fault coverage by targeting at a subset of modules of a processor. The program optimization algorithm reduces energy and average power without sacrificing fault coverage. Our future work focuses on adapting the algorithm to the prevailing superscalar architecture, where features like out-of-order execution pose challenges to program synthesis as well as software optimization for low power.

6 References

[1] S. M. Thatte and J. A. Abraham, "A Methodology for Functional Level Testing of Microprocessors", in *Proceedings of the International Symposium on Fault-Tolerant Computing*, 1978, pp. 90-95.

[2] D. Brahme and J. A. Abraham, "Functional Testing of Microprocessors", IEEE Transactions on Computers, Vol. C-33, June 1984, pp. 475-485.

[3] S. Hellebrand, H.-J. Wunderlich, and A. Hertwig, "Mixed-Mode BIST Using Embedded Processors", in *Proceedings of the IEEE International Test Conference*, 1996, pp. 195-204.

[4] R. S. Tupuri and J. A. Abraham, "A Novel Functional Test Generation Method for Processors using Commercial ATPG", in *Proceedings of the IEEE International Test Conference*, 1997, pp. 743-752.

[5] W. Zhao and C. Papachristou, "Testing DSP Cores Based on Self-Test Programs", in *Proceedings of Design Automation and Test in Europe*, 1998, pp. 166-172.

[6] L. Chen and S. Dey, "DEFUSE: A Deterministic Functional Self-Test Methodology for Processors", in *Proceedings of the 18th IEEE VLSI Test Symposium*, 2000, pp. 255-262.

[7] L. Chen, S. Ravi, A. Raghunathan, and S. Dey, "A Scalable Software-Based Self-Test Methodology for Programmable Processors", in *Proceedings of the 40th Design Automation Conference*, 2003, pp. 548-553.

[8] N. Kranitis, G. Xenoulis, D. Gizopoulos, A. Paschalis, and Y. Zorian, "Low-Cost Software-Based Self-Testing of RISC Processor Cores", in *Proceedings of IEEE Design, Automation & Test in Europe Conference*, 2003, pp. 10714-10719.

[9] F. Corno, M. Sonza Reorda, G. Squillero, and M. Violante, "On the Test of Microprocessor IP Cores", in *Proceedings of IEEE Design, Automation & Test in Europe Conference*, 2001, pp. 209-213.

[10] F. Corno, G. Gumani, M. Sonza Reorda, and G. Squillero, "Fully Automatic Test Program Generation for Microprocessor Cores", in *Proceedings of IEEE Design, Automation & Test in Europe Conference*, 2003, pp. 1006-1011.

[11] W.-C. Lai, A. Krstic, and K.-T. Cheng, "Test Program Synthesis for Path Delay Faults in Microprocessor Cores", in *Proceedings of the 2000 IEEE International Test Conference*, 2000, pp. 1080-1089.

[12] V. Singh, M. Inoue, K. K. Saluja, and H. Fujiwara, "Software-Based Delay Fault Testing of Processor Cores", in *Proceedings of the 12th Asian Test Symposium*, 2003, pp. 68-77.

[13] Y. Zorian, "A Distributed BIST Control Scheme for Complex VLSI Devices", in *Proceedings of IEEE VLSI Test Symposium*, 1993, pp. 4-9.

[14] S. Gerstendörfer and H.-J. Wunderlich, "Minimized Power Consumption for Scan-Based BIST", in *Proceedings of the IEEE International Test Conference*, 1999, pp. 77-84.

[15] P. Girard, "Survey of Low-Power Testing of VLSI Circuits", *IEEE Design and Test of Computers*, vol. 19, no. 3, June 2002, pp. 82-92.

[16] C. Lee, J. K. Lee, T. Hwang, and S.-C. Tsai, "Compiler Optimization on Instruction Scheduling for Low Power", in *Proceedings of the 13th International Symposium on Systems Synthesis*, 2000, pp. 55-60.

[17] Vivek Tiwari, Sharad Malik, Andrew Wolfe, and Mike Tien-Chien Lee. "Instruction Level Power Analysis and Optimization of Software", *Journal of VLSI Signal Processing Systems*, vol. 13, Aug./Sep. 1996, pp. 223-238.

[18] S. Hellebrand, H.-J. Wunderlich, "Synthesis of Self-Testable Controllers", in *Proceedings of European Design Automation Conference* (EDAC/ETC/EuroAsic), Paris, France, Mar. 1994, pp. 580-585

[19] A. J. van de Goor, *Testing Semiconductor Memories, Theory and Practice*. Gouda, the Netherlands: ComTex Publishing, 1998, ISBN: 90-80-427616.

[20] Ken Batcher and Christos Papachristou, "Instruction Randomization Self Test for Processor Cores", in *Proceedings of the 17th IEEE VLSI Test Symposium*, 1999, pp. 34-40

[21] Marc Schuller, "Study of the Switching Activity of RISC-Processors exemplified by the Leon-Processor", *Thesis No. 2042*, 2002, Faculty of Computer Science, University of Stuttgart, Germany (in German)

[22] R. Sankaralingam, R. R. Oruganti, and N. A. Touba, "Static Compaction Techniques to Control Scan Vector Power Dissipation", in *Proceedings of the 18th IEEE VLSI Test Symposium*, 2000, pp.35-40.

[23] "The SPARC Architecture Manual, Version 8", SPARC International, Inc., available under the URL: http://www.sparc.com/standards/V8.pdf

[24] Specification of the RISC processor is available at: http://www.iti.uni-stuttgart.de/~bartscgr/hapra05/hapra_skript_06062005.pdf

[25] "Data Sheet: PrimePower Full-Chip Dynamic Power Analysis for Multimillion-Gate Design", Synopsys, Inc.