Technique for Template Generation for Simultaneous Testing of Multiple Identical Functional Units in Super-scalar Architecture

Rahul Raj Choudhary
Govt. Engineering College, Bikaner (Raj) India

Gayaprasad Sinsinwar
Govt. Engineering College, Bikaner (Raj) India

Aditi Kajala
MITS Laxmangarh (Raj) India

Pooja Bhardwaj
Govt. Engineering College, Bikaner (Raj) India

ABSTRACT
At-speed testing has emerged as dominant test requirement in the era of high speed microprocessors. Since the conventional testing techniques prove to be incompetent, Instruction-Based Self-Testing (IBST) has been proposed as an effective alternate to those conventional techniques for at-speed testing of high performance microprocessors. The Superscalar architectures, with vast functionality and exceptionally high speed have become the central integral part of modern high speed digital systems. However, testing superscalar microprocessors using this approach faces serious challenges, due to the out-of-order execution with multiple functional units and in-order commit behaviour. This paper discusses the test program generation procedure (template based) for multiple identical functional units in a superscalar architecture. Procedures for delay fault testing, which make sure that generated test vectors are applied in the correct order to test each testable path, are developed. The preliminary work has been presented in EWDTS[1]

1. INTRODUCTION
The exponential acceleration can be seen in the chip technology and such rapid advances in VLSI technology and aggressive design methodologies are resulting into development of extremely dense and complex devices. The extensive performance, vast functionality and high order of reliability have become the prime expectations of consumers. Modern computer systems and systems-on-a-chip (SOCs) are built around very high-speed processors, in order to meet the increasing consumer demand of high performance and rich functionality with quick turnaround time. Modern high performance microprocessors use superscalar architecture; they are designed for very high frequency operations, and are implemented in nano-technology. Design reuse is being proved as the only way that allows designers to keep pace with the pace of technological developments. Though it reduces time to market and design effort but, on other hand, it introduces test difficulties. The conventional stuck-at faults are have lost their relevance and instead of this delay faults and crosstalk faults are becoming increasingly important to keep pace with the rapid increase in the speed of modern digital circuits. External tester is absolutely incapable and infeasible for At-Speed testing because of its inherent inaccuracy and cost. The built-in self-test (BIST), which is widely used self-testing technique, is a structural testing methodology. BIST provides a good quality test but, due to the need of design change, possibility of excessive power consumption that may result into burn out of chips, and unacceptable performance loss and area overhead, it is also not a feasible alternate for testing high-performance and excessively dense processor cores. Further, it may be unacceptable to use hardware BIST for testing an optimized high-performance, low-power core, embedded deep inside an SoC due to its poor and limited accessibility and its inability to deal with design changes.

The newer concept for testing, program-based self-testing (also known as software-based self-testing) can alleviate the problems of both external tester and structural BIST. It bridges instruction-level test with the low level fault model. Program-based self-testing (IBST) uses processor instructions to deliver the test patterns and collect the test responses and provide the facility to apply tests in functional mode. Thus, being this technique inherently non-intrusive, it does not contribute for area and performance overheads and it is well suited test methodology for the testing of processor cores embedded deep inside an SoC. Additionally, the test programs developed for this method can also be used for online periodic testing to improve the processor reliability and provide the facility for testing in the field.

Our paper concentrates on the test issues related with the superscalar processors and faced by IBST. Further program generation technique has been developed and depicted for instruction-based self-testing of superscalar processors. The superscalar processor uses out of order execution, and extensive use of design reuse in order to achieve higher performance, this makes the instruction-based testing (IBST) difficult. The paper has been organized on following discussions:

- Discussion of superscalar concept along with the emergence of the test challenges for the instruction-based testing of superscalar architecture
- Test procedures are developed which compel the processor scheduler to make sure that generated test vectors are applied in correct and desired order to the various multiple identical functional units.

The Section 2 describes the previous work made in the area of instruction-based self-testing. Section 3 discusses the test challenges and explains basic overview of superscalar architecture. Section 4 discusses various examples to develop template generation procedures. Finally, we conclude with section 5.

2. PREVIOUS WORK

At extremely high frequencies, the delay test model and crosstalk fault models have proved their better applicability in comparison with stuck-at faults. The importance of at-speed testing and power consumption issues have given instruction based self testing a significant edge over other techniques like BIST and external tester based testing. The number of built in self testing methodologies [4,5,6,7,8] have been proposed in literature. But these approaches target stuck-at faults for simple non-pipelined processors. For example the approaches proposed by Shen et al. [4] and Batcher et al. [5] are based on the instruction randomization method. Batcher et al. [5] proposed built-in self test method which combines the execution of microprocessor instructions with a small amount of on-chip test hardware which is used to randomize those instructions. The drawback of IRST is that the amount of time spent by IRST in a functional mode is much higher. Also, processors are more amenable to random-instruction tests than to random-pattern tests. It is difficult to target structural faults by applying random instructions at the processor level. Chen et al. [6] uses a software tester, embedded in the processor memory as a vehicle for applying structural tests. Chen et al. [6], Krantis et al. [7], and Kambe et al. [8] generate the structural tests for functional modules under constraints. The method proposed by Kambe [8], also generates multiple templates in effective order to detect faults, which may cover different input spaces, and therefore, different detectable fault sets. Approaches have also been proposed for pipelined processors targeting stuck-at faults [9,10,11]. Chen [9] proposed a template-based approach, which proposes RTL simulation-based techniques, for template reading and selection, and techniques based on the theory of statistical regression for extraction of input-output mapping functions whereas Kranitis [10] and Paschalis [11] proposed approaches based on deterministic test of functional modules. Paschalis [11] proposed to detect both types of permanent and intermittent faults by a small embedded test program with test execution time much less than a quantum time cycle.

Unlike stuck-at fault testing, delay testing is closely tied to the test application strategy. Only a few software-based self-testing approaches [12,13,14,15,16] targeting delay faults have been proposed in literature for simple non-pipelined processors. Lai [13] proposes method to self-test a processor core, by running an automatically synthesized test program which can achieve high path delay fault coverage. The two key features of this methodology are: (1) Path Classification Algorithm to find the functionally testable paths and (2) Constraint Extraction to allow the use of constrained ATPG to generate test vectors for functionally testable paths and test program synthesis. The methodology proposed by Lai [12,13,14], extracts constraints by exhaustively searching instructions and instruction pairs, which can be applied in functional mode. The data path logic and the controller of the microprocessor are considered to identify the functionally testable paths in a microprocessor in [12].
Singh et al. [15] proposed a systematic approach for the delay fault testing of processor core, using its instruction set for this purpose. They [15,16] proposed an efficient graph theoretical model to model a simple processor by an Instruction Execution Graph (IE graph), which is then used for constraints extraction to generate test vectors. These generated vectors can then be applied as instructions to test a processor. They also extended their approach to include testing of pipelined and superscalar processors [17,18]. They [17] introduced first work towards modelling of pipeline behaviour for testing of a microprocessor in functional mode. The methodology develops a systematic approach to test delay fault of such processor cores by utilising the instruction set of the processor itself. The graph model and the associated methodology proposed in [17,18], models the static pipeline behaviour, where instructions progress in lock step fashion. Hence, it is not suited for the modelling and testing of a dynamic pipelined architecture such as a superscalar processor. Superscalar processors use buffers and queues to support out-of-order execution. Indeed, as pointed out in this paper, the test application strategy plays a key role for the testing of superscalar processors. Hatzimihail et al. [20] used performance counters to detect erroneous cache misses. The methodology investigates the effects of performance faults in speculative execution units and proposes a generic, software-based test methodology, which utilizes available processor resources - hardware performance monitors and processor exceptions, to detect these faults in a systematic way. Theodorou et al. [21] have used special performance instructions and performance monitoring mechanism to overcome cache tag testability problem. The methodology considers new realistic fault models, on the basis of Fault Primitives (FPs). It applies March Write Operations in a low cost way by taking advantage of the cache pre-fetching mechanism that modern microprocessors and embedded processors include. I. Pomeranz and Reddy [22] have proposed a state variable selection algorithm to eliminate the requirements, for golden response storage. This extends the output response comparison scheme for identical sequential circuits, in order to increase the fault coverage and reduce the fault latency of an unknown input sequence. The extension is based on using state variables, in addition to primary outputs as part of the output response comparison scheme. A recent study by Sinsinwar et al. [1] proposed the the conceptual methodology for testing multiple copies of identical functional units in superscalar architectures whereas Aditi et al. [2] proposed output response comparison scheme for identical sequential circuits for delay test using static transition probability which allows to make selection independent of the input sequence.

This paper presents a specific mechanism to test identical Floating Point units available in multiple copies in superscalar processors. This facilitates testing such multiple units in functional mode of operation targeting delay faults, which is not addressed in the literature as per best of our knowledge.

3. SUPERSCALAR ARCHITECTURE AND TEST CHALLENGES

Our work is focused at delay fault testing of superscalar processors having the features of out of order execution and design reuse. The objective is to generate test sequences that can be applied in the functional mode of operation for multiple identical functional units, using path delay path model [19]. Singh et al. [18] presented the preliminary work in this sequence. They modeled the superscalar (dynamic pipeline) behaviour for the purposes of testing of a superscalar processor. This work describes some of the important issues that are relevant to testing superscalar architectures, highlighting some of the differences between simple pipelined and superscalar architectures. The superscalar and pipelined architectures are discussed in brief

3.1 Pipeline Vs Superscalar Processors

Pipeline concept was introduced to increase throughput and achieving instruction level parallelism. The scalar pipelines are characterized by a single instruction pipeline of *k* stages. All instructions traverse through the same set of pipeline stages, regardless of their different types. At any one point of time only one instruction resides in each of the pipeline stages and the instructions advance through lock step fashion. On the other hand, superscalar processors go beyond just a single-instruction pipeline by being able to simultaneously advance multiple instructions through the pipeline stages and further achieve exceptional speed of execution. Greater concurrency of processing multiple instructions for higher instruction execution throughput is achieved by incorporating multiple functional units, often quantified as instructions per cycle (IPC). Another fundamental character of the superscalar processors is their ability to execute instructions in an order different from the order specified by the original program which is termed as 'out of order execution'.

Various superscalar organizations have been proposed. Typically the superscalar organization consists of instruction fetch and branch prediction unit, decode and register renaming unit, instruction issue unit, execution unit, re-ordering and commit unit.

For this work the simple organization of a superscalar processor which uses distributed reservation station for each functional unit. Re-order buffer (ROB) is used to re-order the instructions. Figure 1 shows the organization of such superscalar processor. This particular organization is used to explain the various concepts, to keep discussion simple. Further these concepts can easily be generalized to other organizations.

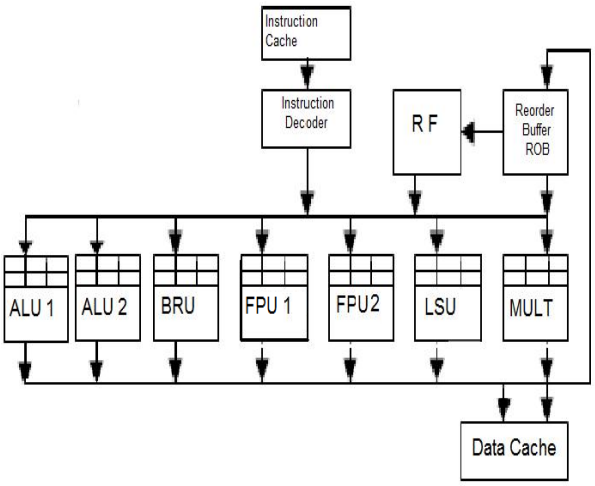

Fig 1: Organization of Superscalar Processor

We have implemented superscalar above shown processor to demonstrate the concept. This implemented organization uses a branch history table with 2 history bits to predict branch. It fetches two instructions and commit two instructions per cycle. Execution unit has 7 functional units (2 ALU, 1 Multiplier, 1 Branch unit, 2 FPU and 1 Load Store unit). Every unit has its own reservation station with 2 entries. Re-Order Buffer (ROB) is implemented as a circular queue with 16 entries. In order to demonstrate the concepts, we assume the fetch width of the processor as 4 instructions.

3.2 Superscalar Test Challenges

The designs reuse and out of order are the prominent characters which extend various facility as well as challenges for testing. In a superscalar processor, the instructions in the sequence may be executed on a different functional unit and possibly in different order of instructions. Further, superscalar architecture uses buffers and queue, which makes it a challenging task to ensure that a given instruction resides at a given location in the buffer or queue with appropriate data at a given time. This leads to a situation where the application of test at desired functional unit becomes very difficult.

4. PROPOSED METHODOLOGY

The methodology has been proposed in EWDTS [1] and further in this section first this has been discussed with development of program generation for identical floating point units. The same examples [1] have been used and subsequently the new program generation for floating point units has been developed. In this section we consider the paths that transfer the data between architectural registers, data and address part of the pipeline registers, buffers and queues. We proposed test program generation for the delay fault testing of the processor. We use the SIE graph to generate the test templates. We assume that any instruction can follow any other instruction. Data forwarding take place through the multiplexers. So, the data that can be received by forwarding path can also be received by the normal paths. As discussed earlier that due to multiple identical functional units any test cannot be applied to the systems. Therefore we developed templates that can make sure that the test can be applied in functional mode. He we are focused on the template generation. The data for the template can be generated by using an constraint ATPG under the functional constraint or random data can be used [1].

In order to support high IPC and out of order execution, superscalar processors are implemented with reservation stations (buffers) and re-order buffer (queue). These allow a processor to run instructions out of order while maintaining the program order. Instruction based testing faces serious challenges due to the out of order execution with multiple functional units and in-order commit behavior, because it is the processor scheduler who decides the order of instruction execution, on the fly, and not the program that executes on the processor. This means that even if we have a test vector sequence generated under architectural constraints, when we apply such a sequence, there is no guarantee that the sequence will indeed be executed by the same functional unit for which it was meant to be. In fact, in a superscalar processor, the instructions in the sequence may be executed on a different functional unit and possibly in different order of instructions. He we can use the fact that the processor has identical functional units, hence, we can compare the response of the two functional units. *Therefore we need not to store the golden response*. Further, superscalar architecture uses buffers and queue, which makes it a challenging task to ensure that a given instruction resides at a given location in the buffer or queue with appropriate data at a given time. We explain this in the following example.

Example1: Consider a 4 instruction wide fetch superscalar implemented with 2 ALU, 1 Multiplier, 1 Shifter, 2 FPU, 1 Load, 1 Store and 1 Branch Unit, where every unit has individual reservation station with 2 entries, and ROB has 32 entries. Processor instructions are represented as (I Rd, Rs1, Rs2) where I specifies operation, Rd is the destination, and Rs1 and Rs2 are the

two source operands. Let a path through ALU be tested by an instruction sequence ADD followed by SUB. This path is from the reservation station to the reorder buffer. Let the desired operands be placed in the registers R2 and R3 for the ADD instruction and in registers R6 and R7 for the SUB instruction. Since we have two identical ALU, we can test those together and can compare the result. Conventionally, we apply the test vectors in the following sequence:

I_1: ADD R1, R2, R3 -- processor schedules this instruction to ALU1
I_2: SUB R5, R6, R7 -- processor schedules this instruction to ALU2
I_3: ADD R8, R9, R10 -- processor schedules this instruction to ALU1
I_4: SUB R11, R12, R13-- processor schedules this instruction to ALU2
I_5: SUB R14, R5, R11 -- processor compare the two results
I_6: BNEQ R14, Fault -- processor notifies fault

The processor may schedule instructions I_1 and I_2 to two different ALUs. Therefore, this sequence will not apply the desired test to any of the ALUs. We will get the correct result after execution of instruction I_5 in spite of having a faulty path, because the fault is not excited. One possible partial solution to this problem is concurrent testing of two ALU's by the following program segment.

I_1: ADD R1, R2, R3 -- processor schedules this instr. to ALU1
I_2: ADD R21, R2, R3 -- processor schedules this instr. to ALU2
I_3: SUB R5, R6, R7 -- processor schedules this instr. to ALU1
I_4: SUB R25, R6, R7 -- processor schedules this instr. to ALU2
I_5: SUB R11, R5, R25, -- processor compares the two results
I_6: BNEQ R11, Fault -- processor notifies fault

This can apply the test sequence to both of the ALUs provided that these instructions are aligned, i.e., all these 4 instructions are fetched simultaneously. We can achieve this by having branch instruction before this set. Now, these instructions can be applied in our desired order. But, reservation station has two entries and first two instructions will be placed in the first entries of respective reservation stations and next two instructions will be placed in the second entries of the corresponding reservation stations. Therefore, the transition cannot be launched and path remains untested. A possible partial solution of this problem is to insert two instructions between I_2 and I_3 which are being scheduled to some other functional units. Therefore, the partial solution which can test the path from the first entry of reservation station to ROB is:

I_1 JUMP 2000H
I_2: 2000H ADD R1, R2, R3 -- processor schedules it for ALU1 (stay at 1st position in RS)
I_3 ADD R21, R2, R3 -- processor schedules it for ALU2 (stay at 1st position in RS)
I_4: MULT R10, R11, R12 -- processor schedules it for Multiplier (Filler instr.)
I_5: SRA R13, R14, R15 -- processor schedules it for Shifter (Filler instruction)
I_6: SUB R5, R6, R7-- processor schedules it for ALU1 (stay at 1st position in RS)
I_7: SUB R25, R6, R7-- processor schedules it for ALU2 (stay at 1st position in RS)
I_8: SUB R11, R5, R25, -- processor compares the two results
I_9: BNEQ R11, Fault -- processor notifies fault

This way, we can make sure that the transition will be created and can be propagated. This simple example demonstrates how to develop a test sequence and its importance. The situation becomes even more complex when we consider feedback paths in the out of order execution engine.

4.1 Test of Forwarding Logic paths:

The forwarding logic paths are responsible to forward data to the instructions residing in the RS without going through commit stage. These paths dominate the normal paths, i.e, a test for the forwarding paths can also test the normal paths. Hence, normal paths can be tested along with forwarding paths by using observation sequence for normal paths. An instruction sequence (I_{v1}, I_{v2}, I_{v3}) can test a path from RS (i^{th} entry) to the same RS (j^{th} entry) if it is applied in the following manner. This instruction sequence will test both normal paths and forwarding paths.

Example 4: Let a processor has two ALU units with 2 entry reservation station and the fetch width is 4. Assume, ADD instruction followed by SUB instruction is a test instruction pair. Paths from the 1st entry in RS to the 2nd entry in RS can be tested by the following sequence.

I_1: J 2000H -- instruction for the alignment
I_2: 2000H ADD R1, R2, R3 -- Instruction I_{V1}
I_3: ADD R21, R2, R3 -- Instruction I_{V1}
I_4: SW R1, 100(R9) – Filler instruction
I_5: SW R21, 104(R14) – Filler instruction
I_6: SUB R4, R5, R6 – Instruction I_{V2}
I_7: SUB R24, R5, R6 – Instruction I_{V2}
I_8: ORA R7, R4, R8 – Instruction I_{V3}
I_9: ORA R17, R24, R8 – Instruction I_{V3}
I_9: SUB R11, R7, R17 – Comparison of results
I_9: BNEQ R11, Fault – Notifies fault

4.2 Testing Identical Floating Point Units:

As the method above illustrated, the sequence of instructions can be generated for other functional units. Assuming two floating units, we propose the following program for simultaneous testing of these identical units.

I_1: J 2000H -- instruction for the alignment
I_2: 2000H ADDF F1, F2, F3 -- Instruction I_{V1}
I_3: ADDF F21, F2, F3 -- Instruction I_{V1}
I_4: SW F1, 100(F9) – Filler instruction
I_5: SW F21, 104(F14) – Filler instruction
I_6: SUBF F4, F5, F6 – Instruction I_{V2}
I_7: SUBF F24, F5, F6 – Instruction I_{V2}
I_8: ORA F7, F4, F8 – Instruction I_{V3}
I_9: ORA F17, F24, F8 – Instruction I_{V3}
I_9: SUB F11, F7, F17 – Comparison of results
I_9: BNEQ F11, Fault – Notifies fault

The templates can be generated for different combinations of the instructions. Since this is a manual process, and for generation of generalized scheme, applicable for other processors, the

development of automated facility is intended. Further automation is aimed as our future work.

5. CONCLUSION

This paper highlighted the issues that are pertinent to testing of identical functional units in a superscalar architecture in the functional mode of operation. Since the application of test at desired functional unit is a big issue, we have developed the test templates that can force the processor scheduler to execute program in our desired order. Hence, these procedures can apply test vectors in the functional mode of operation. As it is applicable in the functional mode of operation, it can be used for online testing and need not to save the golden response or standard response of test. This work is the ongoing work and the evaluation of the methodology and automation of the methodology are the future directions. The different programs aimed at test of different type of functional units, have been developed and this may further lead to an integration of these techniques in order to have a complete test procedure.

6. ACKNOWLEDGEMENT

The authors acknowledge the valuable inputs from Dr Virendra Singh, Super Computer Education and Research Centre, Indian Institute of Science, Bangalore, India.

7. REFERENCES

[1] Gayaprasad Sinsinwar, Rahul Raj Choudhary, Aditi Kajala, Virendra Singh,"Test Program Generation for Simultaneous Testing of Multiple Identical Functional Units in Super-scalar Architecture", IEEE East-West Design and Test Symposium (EWDTS) 2010, St. Petersburg, Russia, Sep 2010, pp. 195-199.

[2] Aditi Kajala, Gayaprasad Sinsinwar, Rahul Raj Choudhary, Jaynarayan Tudu, Virendra Singh, "On Selection of State Variables for Delay Test of Identical Functional Units", IEEE East-West Design and Test Symposium (EWDTS) 2010, St. Petersburg, Russia, Sep 2010, pp. 200-203.

[3] S.M. Thatte and J. Abraham, "Test generation for Microprocessors", IEEE Trans. on Computers, Vol. C-29, No. 6, June 1980, pp. 429-441.

[4] J. Shen and J.A. Abraham, "Native Mode Functional Test Generation for Processors with Applications to Self-Test and Design Validation", Proc. of the International Test Conference 1998, pp. 990-999.

[5] K. Batcher and C. Papachristou, "Instruction Randomization Self Test for Processor Cores" Proc. of the VLSI Test Symposium 1999, pp. 34-40.

[6] L. Chen, and S. Dey, "Software-Based Self-Testing Methodology for Processor Cores", IEEE Trans. on CAD of Integrated Circuits and Systems, Vol. 20, No.3, March 2001, pp. 369-380.

[7] N. Krantis, A. Paschalis, D. Gizopoulos, and Y. Zorian, "Instruction-Based Self-Testing of Processor Cores", Journal of Electronic Testing: Theory and Application (JETTA) 19, 2003, pp 103-112.

[8] K.Kambe, M.Inoue, and H. Fujiwara, "Efficient Template Generation for Instruction-Based Self-Test of Processor Cores", Proc. of Asian Test Symposium, 2004, pp. 152-157.

[9] L. Chen, S. Ravi, A. Raghunath, and S. Dey, "A Scalable Software-Based Self-Test Methodology for Programmable Processors", Proc. of Design Automation Conference 2003, pp. 548-553.

[10] N.Krantis, G.Xenoulis, A.Paschalis, D.Gizopolous, and Y.Zorian, "Application and Analysis of RT-Level Software-Based Self-Testing for Embedded Processor Cores", Proc. of International Test Conference, 2003, pp 431-440.

[11] Paschalis, and D. Gizopoulos, " Effective Software-Based Self-Test Strategies for On-Line periodic Testing of Embedded Processors", Proc. of Design and Test in Europe 2004, pp 578-583.

[12] W.-C. Lai, A. Krstic, and K.-T. Cheng, "On Testing the Path Delay Faults of a Microprocessor Using its Instruction Set", Proc. of the VLSI Test Symposium 2000, pp. 15-20.

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

[14] W.-C. Lai, and K.-T. Cheng, "Instruction-Level DFT for Testing Processor and IP Cores in System-on-a-Chip", Proc. of the Design Automation Conference 2001, ACM Press, NY, 2001, pp. 59-64.

[15]V. Singh, M. Inoue, K.K. Saluja, and H. Fujiwara, "Instruction-Based Delay Fault Testing of Processor Cores", Proc. of the International Conference on VLSI Design 2004, pp 933-938.

[16] V. Singh, M. Inoue, K.K. Saluja, and H. Fujiwara, "Delay Fault Testing of Processor Cores in Functional Mode", IEICE Trans. on Information & Systems, Vol. E-88D, No. 3, March 2005, pp. 1-9.

[17] V. Singh, M. Inoue, K.K. Saluja, and H. Fujiwara, "Instruction-Based Delay Fault Testing of Pipelined Processor Cores", Proc. of International Symposium on Circuits and Systems 2005.

[18] V.Singh, M.Inoue, K.K.Saluja, and H.Fujiwara, "Program Based Self-Testing of Superscalar Microprocessors", Proc. of North Atlantic Test Workshop (NATW), 2005.

[19] Krstic and K.-T. Cheng, Delay fault testing for VLSI circuits, Kluwer Academic Publishers, 1998.

[20] M. Hatzimihail et al., A Methodology for Detecting performance Faults in Microprocessor Speculative Execution Units via Hardware Performance Monitoring", in Proc. of International Test Conference [ITC], 2007.

[21] G. Theodorou et al., "A Software Based Self-Test Methodology for In-System Testing of Processor Cache tag Arrays", in Proc. of International On-Line Testing Symposium (IOLTS) 2010

[22] Pomeranz and S.M. Reddy, "Selecting State Variables for Improved On-line Testability Through Output Response Comparison of Identical Circuits", in Proc. of International On-Line Testing Symposium (IOLTS) 2010.