17th Asian Test Symposium

On-Line Instruction-checking in Pipelined Microprocessors

Stefano Di Carlo
Department of Control and Computer Engineering, Politecnico di Torino, Torino, Italy.
E-mail: stefano.dicarlo@polito.it

Giorgio Di Natale
Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier UMR 5506, France.
E-mail:giorgio.dinatale@lirmm.fr

Riccardo Mariani
Yogitech s.p.a., Pisa, Italy
E-mail:
riccardo.mariani@yogitech.com

***Abstract*— Microprocessors performances have increased by more than five orders of magnitude in the last three decades. As technology scales down, these components become inherently unreliable posing major design and test challenges. This paper proposes an instruction-checking architecture to detect erroneous instruction executions caused by both permanent and transient errors in the internal logic of a microprocessor. Monitoring the correct activation sequence of a set of predefined microprocessor control/status signals allow distinguishing between correctly and not correctly executed instructions.**

I. Introduction

Information technology drastically changes our world. Economies are shifting from the industrial age of steel and cars to the information age of computer networks and ideas [1]. Microprocessors play a central role in this revolution. They are commodity products and represent an essential part of computers, cars, train controls, aircrafts, cellular phones, and personal digital assistants, to name just a few. Embedded microprocessors in daily surroundings are gaining more recognition than in the past and this trend will continue in the next years [2].

VLSI and microprocessors performances have increased by more than five orders of magnitude in the last three decades. As technology scales down we face new challenges such as process variations, single event upsets (soft-errors [3]), and device degradation ([4][5]) leading to inherent unreliability of digital systems. With the massive use of microprocessors in commodity applications we can expect wider sectors of the electronic industry to be demanding for on-line testing solutions in order to ensure the welfare of the users of electronic products [6][7].

Traditional approaches based on massive redundancy such as the ones proposed in [8][9][10][11][12][13][14][15][16][17][18][19] may not be suitable for applications with strong constraints in terms of power consumption and budget. Therefore, there is a need for new fault tolerance methods that can be implemented at a reasonable cost.

This paper proposes an on-line testing methodology for pipelined microprocessors based on concurrent instruction checking, extending the methodology proposed in [20]. The goal of the proposed solution is to detect errors appearing in the control unit of a pipelined microprocessor by monitoring a subset of the microprocessor's control and status signals. The proposed approach targets both transient and permanent faults and it is able to on-line detect faults that lead to a variation of the activation sequence of the monitored signals. One of the main advantages of the proposed schema, with respect to the solution proposed in [20], is a precise methodology that allows identifying the minimum set of signals to monitor in order to reach the desired level of fault detection, thus minimizing the overall hardware overhead.

The paper is organized as follows. Section II introduces the proposed instruction checking technique, whereas section III details the algorithm for the selection of the minimum number of signals to monitor. Section IV describes the results of the application of the instruction checking to an open source microprocessor core, and finally section V summarizes the main contributions of the work and concludes the paper.

II. Instruction Checking

Figure 1 shows the basic architecture required to implement the proposed instruction checking methodology. It comprises four main blocks: (i) the microprocessor under test (MUT), (ii) the system memory containing the program to execute and its data, (iii) the system bus used to fetch instructions and data from the memory, and (iv) the Instruction Checking Module (ICM), an external block able to understand whether the microprocessor correctly executes the instructions fetched from the system bus.

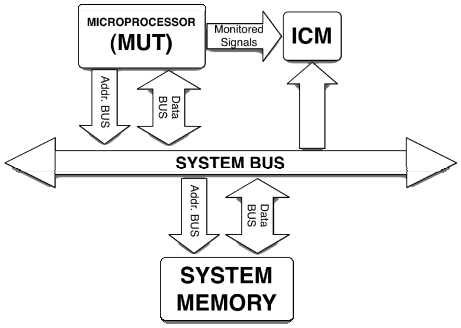

Figure 1. Instruction checking architecture.

Every time the MUT fetches an instruction, the same instruction is also fetched by the ICM. The ICM continuously monitors the waveforms produced by a selected set of control and status signals coming from the microprocessor. Either MUT's external or internal signals (when available) can be monitored by the ICM. By comparing these waveforms with a set of pre-calculated ones obtained by simulation, the ICM tries to identify erroneous instruction executions. The proposed approach covers errors

1081-7735/08 $25.00 © 2008 IEEE
DOI 10.1109/ATS.2008.47

in the control unit of the MUT that manifest as a modification of the activation sequence of the monitored signals. Errors appearing in the data processed by the instructions that do not modify the activation sequence of the monitored signals are not covered. Nevertheless, this type of error has been already addressed in literature resorting to data redundancies [21] [22].

The implementation of the instruction checking architecture of Figure 1 presents two major challenges:

- *The identification of the best set of signals to monitor*. The higher is the number of monitored signals, the higher is the number of recognized instructions. Nevertheless, the complexity of modern microprocessors and the very high number of available signals requires the identification of an optimal subset guaranteeing the highest coverage with the lowest complexity. This task is not trivial and requires opportune algorithms;
- *The definition of an ICM architecture keeping the hardware overhead as low as possible*.

The following subsections will introduce a set of basic concepts needed to address the proposed challenges.

A. Microprocessor signal classifications

Considering a generic microprocessor, the full set of available *Observable Signals* (*OS*) can be classified based on the effort required for their external monitoring:

- *External Signals,* corresponding to the external pins of the microprocessor. They can be directly routed from the MUT to the ICM and therefore they are fairly easy to monitor;
- *Internal Signals,* corresponding to the internal nets of the microprocessor. Their observation is only possible when the microprocessor design is available and modifiable since it requires the introduction of additional pins. They should be considered only if really required, i.e., if their contribution to the final coverage justifies the cost of the microprocessor design modification.

Based on this classification, for each signal $s \in OS$ it is possible to define a Signal Observation Cost (*SOC)* taking into account the cost to route the signal from the MUT to the ICM. The design of the ICM requires the identification of the minimum set of observable signals that maximizes the fault detection capabilities and minimizes the global observation cost.

B. Microprocessor pipeline and data path

Almost all modern microprocessors use pipelines to enhance their performance. Even if each microprocessor family presents different implementations, this paper considers a generic 5-stages pipeline. The proposed model is general enough to be mapped on several commercial microprocessors. Moreover, the proposed instruction checking architecture is not limited to this pipeline model and can be easily extended to more complex architectures The five stages pipeline performs the following tasks:

- *Instruction fetch*: the instruction is fetched from memory and placed in the instruction cache;
- *Decode*: the instruction is decoded;
- *Operands fetch*: the instruction operands are read from the register file/memory.
- *Execute*: the operation is executed;
- *Write-back*: the instruction results are written back to the register file or to the memory.

Depending on the microprocessor architecture, *Operands fetch* and *Execute* operations may require more than one clock cycle.

The organization of the pipeline leads to a partition of the blocks composing the microprocessor data-path. Figure 2 shows an example of a generic microprocessor data-path partitioned into five different areas corresponding to the five stages of the pipeline. According to this partitioning we can easily suppose that, during the execution of a program, each stage of the pipeline drives a distinct set of signals not driven by other stages. This partitioning is extremely important to identify the signal waveforms generated by each instruction as will be detailed in section III.A.

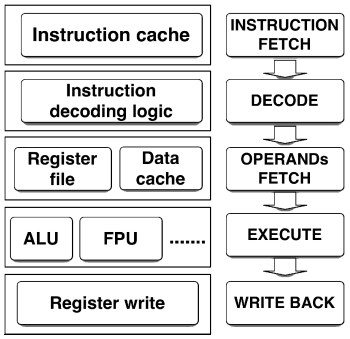

Figure. 2. Generic data-path partitioning.

C. Signal-fingerprints

The concepts introduced in sections II.A and II.B lead to the conclusion that in a pipelined microprocessor each instruction identifies a particular activation sequence of the set of observable signals *OS*.

Def. 1: Given an Instruction (*I*), and a set of *Monitored Signals* ($MS \subseteq OS$), the *signals-fingerprint* of *I* is a $m \times n$ matrix *M* where *m* is the number of pipeline stages, $n = |MS|$ is the cardinality of *MS* (i.e., the number of monitored signals), and each element $M[i, j] \in \{0,1,-\}$ represents the value of the j^{th} monitored signal in the i^{th} stage of the pipeline. The '-' symbol means that the signal is not driven in the pipeline stage. ■

Eq. 1 shows an example of signals-fingerprint for the proposed 5-stages pipeline and a set of seven *MS*.

$$M = \begin{array}{ccccccc} 1 & 0 & - & - & - & - & - \\ - & - & 1 & - & - & - & - \\ - & - & - & 1 & 1 & - & - \\ - & - & - & - & 1 & 1 & - \\ - & - & - & - & - & - & 0 \\ s_1 & s_2 & s_3 & s_4 & s_5 & s_6 & s_7 \end{array} \begin{array}{l} \text{Clk}_1 \text{ (Fetch)} \\ \text{Clk}_2 \text{ (Decode)} \\ \text{Clk}_3 \text{ (Op.Fetch)} \\ \text{Clk}_4 \text{ (Execute)} \\ \text{Clk}_5 \text{ (Write}-\text{Back)} \end{array} \quad (1)$$

Depending on the target microprocessor, the number of clock cycles required to execute the instructions composing the instruction set can be constant (e.g., five clock cycles in a 5-stages pipeline) or not. In the identification of the signals-fingerprints this possibility must be taken into account. Signals-fingerprints allow an easy classification of the microprocessor instructions.

Def. 2: An "*Instructions Class"* is a set of instructions identified by the same signals-fingerprint. ■

Instructions belonging to the same class cannot be distinguished by the ICM without monitoring additional signals. There is no way to guarantee a correlation between instruction classes, and groups of instructions with the same functionality (e.g., arithmetic, branches, etc.). Nevertheless, since instructions with same functionality are supposed to use the same functional blocks and therefore involve the same set of control signals, we can likely expect a correlation.

Lemma 1: *A microprocessor instruction is covered by the instruction checking if, by observing the selected set of monitored signals (MS), it is possible to distinguish between the given instruction and all the remaining ones.* ■

D. ICM generic architecture

Based on the definition of signals-fingerprint (Def.1), Figure 3 proposes a generic ICM implementation. The ICM reads the *OPCODE* of each instruction fetched from the system bus. The *Instruction Decoder* is a look-up table; it decodes the *OPCODE* and provides the values expected on *MS* (signals-fingerprints). Depending on the pipeline stage driving the signals, opportune delay blocks are inserted to guarantee the synchronization with the pipeline. At each stage of the pipeline, the expected signals and the actual signals produced by the processor are compared. In case of mismatch the $go/\overline{nogo}$ output notifies the event to the system. Interrupts do not represent a limitation for this architecture. The ICM analyzes the execution of a single instruction at a time, no matter the type of program currently executed (i.e., user program, system call, interrupt handler, etc.).

The ICM focuses on errors inside the microprocessor only, it does not consider errors appearing on the bus or on the internal memory. The problem of on-line testing the system bus and the system memory using error detection/correction codes is a well-known problem and many solutions can be found in literature. The system bus and memory are therefore considered error free.

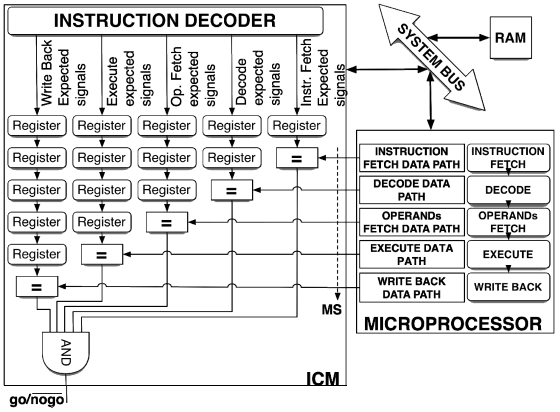

Figure. 3. Generic ICM Architecture.

III. SELECTION OF MONITORED SIGNALS

This section addresses the selection of the minimum set of microprocessor signals the ICM has to monitor in order to reach the desired level of instructions coverage (see Lemma 1). The selection process has to trade-off between the number of signals to monitor, and the cost of the selected signals (see section II.A). It includes two phases: (i) the signals-fingerprints generation computed for each instruction and for the whole set of observable signals, and (ii) the signals-fingerprints analysis. The following subsections detail the two phases.

A. Signals-fingerprints generation

For each microprocessor instruction *I,* the corresponding signals-fingerprint considering the full set *OS* of observable signals has to be computed. Each signals-fingerprint is computed by simulating the MUT and by analyzing the waveforms (*simulation dump*) produced on the *OS*. For each instruction *I* the generation of the simulation dump requires the execution of the small test program proposed in Figure 4. This sequence of instructions, designed for a generic *N*-stages pipeline, guarantees the target instruction traverses all the *N* stages of the pipeline.

The simple use of simulations is not enough for the signals-fingerprint computation. In fact, at a certain time *t*, the value of the observed signals depends on all the instructions currently loaded in the MUT pipeline, not only on the single target instruction. It is therefore mandatory to know, for each signal $s \in OS$, the stage of the pipeline in charge of its control (see section II.A). This information can be obtained by analyzing the internal structure (e.g., the VHDL/Verilog description) of the microprocessor.

From the simulation dump of each instruction and from the knowledge of the pipeline signals partitioning it easy to define the activation matrix *M* that identifies each signals-fingerprint (see Def. 1).

```
NOP
NOP  N
...
Instruction
NOP
NOP  N
...
```

Figure. 4. Test program for the simulation dump generation on a MUT with a N-stages pipeline.

B. Signals-fingerprints analysis

The signals-fingerprints analysis is the algorithm needed for the identification of the minimum set of signals to monitor during the instruction checking.

Given the signals-fingerprint of each instruction computed over the full set of *OS* (see section III.B), and an upper bound on the number of selectable signals (#*SS*) that can be monitored by the ICM, the algorithm identifies the best solution in terms of monitored signals $MS_i \subseteq OS$ for I ranging between 1 and #SS. Each solution MS_i is the set of *i* observable signals that minimizes cost function of Eq.2.

$$CF(MS_i) = C_1 \cdot f_1(\#IC, \#I) + C_2 \cdot f_2() + C_3 \cdot f_3() \qquad (2)$$

where:

- $f_1(\#IC,\#I)$ measures the number #*IC* of Instructions Classes identified by the signals composing the solution over the number #*I* of available instructions (Eq. 3). The higher is #*IC*, the higher is the coverage (see Lemma 1) of the solution. f_1 is a linear function equal to 0 in case #*IC* = #*I* (100% of coverage) and equal to 100 in case of #*IC*=1 (0% of fault coverage);
- f_2 measures the observation cost of the solution calculated as the sum of the Signal Observation Costs (SOC) of the selected signals (see section II.A) normalized to 100 (Eq. 4). The SOC depends on the application and evaluates the effort needed to route the signal from the processor to the ICM. Low cost signals will be best candidates for being included in the solution. Signals with null cost will be always selected;
- f_3 measures how the instructions classes identified by the solution are balanced in terms of number of instructions (Eq. 5 and Eq. 6). A solution identifying two classes each one composed of two instructions will be preferred (less cost) to a solution with two classes with one and three instructions respectively (higher cost). This measure is defined as the standard deviation of the cardinality of each instructions class $|IC_i|$ over the standard deviation of the worst distribution normalized to 100. The worst distribution consists in #*IC*-1 classes including a single instruction and a single class including the remaining ones (#I-#IC-1).

$$f_1(\#IC,\#I) = 100 - \frac{100}{(\#I-1)}(\#IC-1) \qquad (3)$$

$$f_2() = \sum_{\forall \text{signal } s_j \text{ in solution}} SOC_{s_j} \frac{100}{\max(SC_k)_{\forall \text{solution k}}} \qquad (4)$$

$$f_3() = \underbrace{\sqrt{\sum_{i=1}^{\#IC} \frac{\left(|IC_i| - \overline{|IC|}\right)^2}{n}}}_{\text{standard deviation}} \cdot \frac{100}{\sigma_{WD}} \qquad (5)$$

$$\sigma_{WC} = \sqrt{\sum_{i=1}^{\#IC-1} \frac{\left(1 - \overline{|IC|}\right)^2}{n} + \frac{\left((\#I - \#IC - 1) - \overline{|IC|}\right)^2}{n}} \qquad (6)$$

The three constants C_1, C_2, and C_3 can be freely defined and they allow weighting the different parameters depending on the specific application.

Given the number *i* of signals to include into a solution ($1 \le i \le \#SS$) the identification of the set of signals that minimizes CF can be performed using two different approaches:

- *Exhaustive*: the entire set of combinations of the observable signals into groups of *i* signals is evaluated. For each combination CF is evaluated and the solution with minimum cost is selected. The number of the possible solutions to evaluate is defined in Eq. 7.

$$\#Solutions = \binom{\#OS}{i} \qquad (7)$$

- *Greedy*: the selection of a new signal is done considering the solution obtained so far, and adding the signal that locally minimizes CF. The steps of the greedy algorithm are summarized in Algorithm 1.

Algorithm 1: Greedy Algorithm

```
1.  MS ← ∅
2.  while (|MS| < i) do
3.    MinCost ← ∞, newsignal ← NULL
4.    foreach (s ∈ OS ∧ s ∉ MS) do
5.      if (CF(MS ∪ s) < MinCost) then
6.        MinCost ← CF(MS ∪ s), newsignal ← s
7.      end if
8.    done
9.  MS ← MS ∪ newsignal
10. done
```

IV. Experimental Results

We implemented the proposed instruction checking architecture for the open source Plasma CPU core [23]. This processor supports all MIPS I™ user mode instructions with

the exception of unaligned load and store operations that are patented. The architecture of this processor is sketched in Figure 5. The processor has a simplified 3-stages pipeline.

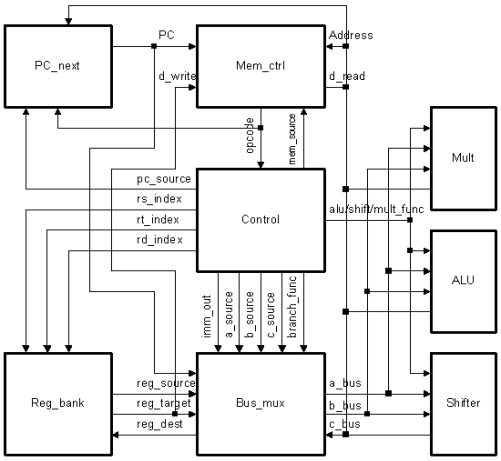

Figure. 5. Plasma CPU architecture.

We successfully generated the full set of signals-fingerprints for all the available instructions according to the procedure introduced in section III.A. We considered an initial set of 28 observable signals (*OS*) including all signals controlled by the microprocessor control unit.

Being *#OS* not too high, we applied the exhaustive selection algorithm proposed in section III.B with #SS equal to 28. The result of the algorithm is that by monitoring 21 signals over the 28 available we are able to reach the maximum instructions coverage with the minimum cost. The execution of the exhaustive algorithm required about 2 hours on a Intel Core Duo with 1GB of RAM.

The selected solution identifies 56 instruction classes over 60 instructions. Only 3 classes contain more than 1 instruction. In particular, the following instructions cannot be distinguished by the ICM:

- ADDI and ADDIU
- ADD, ADDU and DADDU
- SUB and SUBU

As expected there is a strong correlation between instructions classes and instructions grouped by their functionalities.

All selected signals are driven in the execute stage (2nd stage) of the pipeline. This characteristic strongly reduces the complexity of the ICM and thus the final overhead. The instruction coming from the bus is fetched by the processor and at the same time by the ICM. The instruction is decoded inside the ICM in order to generate the set of signals that the processor should generate in order to correctly execute that instruction. Two additional signals (coming from the processor) are used to synchronize the execution of long instructions, i.e., instructions that require more than one clock cycle to be executed (e.g., multiplication instructions and load/store instructions performed with a slow access memory).

The whole system (processor, ICM, memory and bus) has been synthesized with RTL Compiler (Cadence) [24]. The area of the circuit is reported in Figure 5. A first good result of the proposed architecture is that the area overhead is not high. Indeed, the ICM counts of 193 cells, with an overall overhead equal to 1,43%.

```
====================================================
Generated by: Encounter(r) RTL Compiler v06.10-s007_1
Generated on: Jan 30 2007  11:21:24 AM
Module: ICM_system
Technology libraries:   c35_CORELIB 2.0
c35_IOLIB_4M 1.9
Operating conditions:   _nominal_ (balanced_tree)
Wireload mode:          enclosed
====================================================

Instance             Cells   Cell Area   Net Area
-----------------------------------------------------
  ICM_system          7199     1155245     172701
    u1_cpu            7005     1138574     166140
      u4_reg_bank     3685      829265      76230
      u8_mult         1287      132787      26667
      u2_mem_ctrl      477       46810       6237
      u7_shifter       472       32360       9279
      u6_alu           251       28028       4905
      u5_bus_mux       349       26954       4608
      u1_pc_next       194       24424       2268
      u3_control       212       12230       2934
    u3_ICM             193       16598       2880
```

Figure. 5. Synthesis results.

In order to understand the effectiveness of the proposed architecture we set up a fault injection campaign to calculate the detection capability of the ICM in presence of a Single Event Upset (SEU) appearing into an internal flip-flop of the microprocessor.

The experiments consist in injecting bit-flips in the microprocessor register while it is executing a small assembly program designed to extensively use the complete instruction set. The program is composed of 72 instructions. Bit-flips are injected during all possible clock cycles from the first executed instruction to the last one.

Over 6900 injected SEUs, only 368 produced an error at the output of the microprocessor. That means that all the other bit flips targeted “not alive” flip-flops, i.e., their value was not relevant for the application or their value was lately re-written without reading it. Over the 368 relevant SEUs the ICM has been able to detect 286 faults, reaching a coverage percentage of 76,63%.

Figure 6 summarizes the coverage capability related to the processor’s functional blocks. It’s interesting to note that while the fault coverage is quite high for internal registers, the coverage of the Program Counter’s flip-flops is not very high. This is normal because this technique does not target control flow errors but it aims at the detection instructions not correctly executed. A fault in the program counter would most likely lead to a control flow error, i.e., the next executed instruction will not be the correct one. Anyway, this technique guarantees that the instruction fetched by the processor is correctly executed (even if it’s not the expected

instruction to be executed). Without considering faults affecting the program counter, the ICM is able to detect 89,70% of faults.

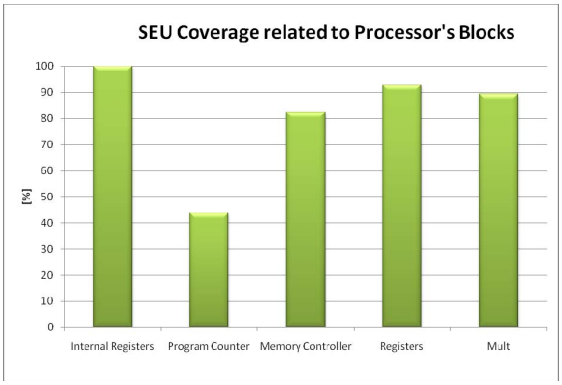

Figure. 6. SEU Coverage related to Processor's functional blocks.

V. CONCLUSIONS

This paper proposed an on-line testing methodology for pipelined microprocessors. It aims at detecting errors appearing in the control unit of a pipelined microprocessor. The overall idea is to monitor a set of control signals of the target microprocessor identifying erroneous sequences of activations. The main problem in applying this technique is the identification of the set of signals to monitor in order to guarantee a high level of fault detection vs. a low complexity of the hardware introduced to perform the check. We proposed an algorithm to identify the minimum set of signal to monitor in order to reach a certain level of detection.

We successfully applied this technique to an open source processor obtaining promising fault detection coverage with a very reduced hardware overhead.

REFERENCES

[1] "A survey of the world economy: The hitchhiker's guide to cybernomics", The Economist, September 28 1996.

[2] K. Sakamura, "Farewell message", IEEE Micro Vol. 22, Issue 6., 2002, pp. 2.

[3] N. Seifert, X. Zhu, L.W. Massengill, "Impact of scaling on Soft-Error Rates in Commercial Microprocessors", IEEE Transaction on Nuclear Science, Vol. 49, No. 6, Dec. 2002.

[4] S. Borkar er al., "Parameter Variations and Impact on Circuits and Microarchitecture", 40th Design Automation Conference, DAC03, IEEE CS Press, 2003, pp. 338-342.

[5] F. Irom, F.H. Farmanesh, G.M. Swift, A.H. Johnston, G.L. Yoder, "Single-event upset in evolving commercial silicon-on-insulator microprocessor technologies", IEEE Transactions on Nuclear Science, Volume 50, Issue 6, Part 1, Dec. 2003, pp:2107 – 2112.

[6] M. Nicolaidis, Y. Zorian, "On-Line Testing for VLSI – A Compendium of Approaches", Journal of Electronic Testing, Volume 12, Numbers 1-2, 1998, pp. 7-20.

[7] S. Kim, A. K. Somani. "On-Line Integrity Monitoring of Microprocessor Control Logic.", 2001 International Conference on Computer Design, ICCD 2001, 23-26 Sept. 2001, pp.:314-319.

[8] S. S. Yau, F. Ch. Chen, *"An Approach to Concurrent Control Flow Checking"*, IEEE Transaction on Software Engineering, Vol. SE-6, No. 2, pp. 126-137, 1980.

[9] R. Leveugle, T. Michel, G.Saucier, *"Design of Microprocessors with Built-In On-Line test"*, 20th International Symposium on Fault-Tolerant Computing (FTCS-20), pp. 450-456, 1990.

[10] A. Mahamood, E. J. McCluskey, "*Concurrent Error Detection Using Watchdog Processor - A Survay*", IEEE Transaction on Computer, Vol. 37, No. 2, pp. 160-174, 1988.

[11] D. J. Lu, "*Watchdog processors and VLSI*" in Proc. Nat. Electron. Conf., vol. 34, 1980, pp. 240–245.

[12] A. Mahmood and E. J. McCluskey, "*Watchdog processor: Error coverage and overhead*" in Digest, 15th Ann. Int'l. Symp. Fault-Tolerant Computing (FTCS-15), 1985, pp. 214–219.

[13] M. Namjoo, "Techniques for Concurrent Testing of VLSI Processor Operation", International Test Conference (ITC-82), pp. 461-468, 1982.

[14] M.A. Schutte, J.P. Shen, D. P. Siewiorek, Y. X. Zhu, "Expermental Evaluation of Two Concurrent Error Detection Schemes", 16th International Symposium on Fault Tolerant Computing (FTCS-16), pp. 138-143, 1986

[15] K. Wilken, J.P. Shen, "Continuous Signature Monitoring: Low-Cost Concurrent Detection of Processor Errors", IEEE Transaction on Computer Aided Design and Systems, Vol. 9, Issue 6, pp. 629-641, June 1990.

[16] T. Michel, R. Leveugle, G. Saucier, "A New Approach to Control Flow Checking without Program Modification", 21th International Symposium on Fault-Tolerant Computing (FTCS-21), pp. 334-341, 1991.

[17] Shambhu Upaddhyaya, Bina Ramamurthy, "Concurrent Process Monitoring with No Reference Signatures", IEEE Transaction on Computer, Vol. 43 no. 4, pp. 475-480, April 1994.

[18] G. Miremadi, J. Ohlsson, M. Rimen, J. Karlsson, "Use of Time and Address Signatures for Control Flow Checking", 5th IFIP Working Conference on Dependable Computing for Critical Applcation (DCCA-5), pp. 113-124, 1995.

[19] X. Delord, G.Saucier, "Control Flow in Pipelined RISC Microprocessor: The Motorola MC88100 Case Study", Workshop on Real Time (Euromicro '90), pp. 162-169, 1990.

[20] S. F. Daniels. "A concurrent test technique for standard microprocessors". In Digest of Papers, COMPCON Spring 83, pages 389-394, San Francisco, February 1983.

[21] N. Oh, S. Mitra, E.J. McCluskey, "ED4I: error detection by diverse data and duplicated instructions", IEEE Transactions on Computers, Volume 51, Issue 2, Feb. 2002 Page(s):180 – 199.

[22] A. Benso, S. Di Carlo, G. Di Natale, P. Prinetto, "A watchdog processor to detect data and control flow errors", 9th IEEE On-Line Testing Symposium, 2003. IOLTS 2003. 7-9 July 2003, Kos (Greece), Page(s):144 – 148.

[23] http://www.opencores.org

[24] http://www.cadence.com