Software-Based Self-Test of Set-Associative Cache Memories

Software-Based Self-Test of Set-Associative Cache Memories

Authors: Di Carlo S., Prinetto P., Savino A.

Published in the IEEE Transactions on Computers Vol. 60 ,No. 7, 2001, pp. 1030-1044.

DOI: 10.1109/TC.2010.166

Software-Based Self-Test of Set-Associative Cache Memories

Stefano Di Carlo, *IEEE Member* , Paolo Prinetto, *IEEE Member*, Alessandro Savino

Abstract—Embedded microprocessor cache memories suffer from limited observability and controllability creating problems during in-system tests. This paper presents a procedure to transform traditional march tests into software based self test programs for set-associative cache memories with LRU replacement. Among all the different cache blocks in a microprocessor, testing instruction caches represents a major challenge due to limitations in two areas: (i) test patterns which must be composed of valid instruction opcodes; and (ii) test result observability: the results can only be observed through of the results of executed instructions. For these reasons the proposed methodology will concentrate on the implementation of test programs for instruction caches. The main contribution of this work lies in the possibility of applying state-of-the-art memory test algorithms to embedded cache memories without introducing any hardware or performance overheads and guaranteeing the detection of typical faults arising in nanometer CMOS technologies.

Index Terms—Memory testing, microprocessor testing, cache memories, Software Based Self Test.

1 Introduction

The industry trends for improving microprocessor performances impose the adoption of large hierarchical memory subsystems including up to four levels of cache memories [1], [2]. Cache memories are high-speed buffers used to temporarily hold information that is likely to be used during the execution of programs. As a result, larger and larger portions of the die areas of microprocessors are nowadays devoted to cache memory blocks, e.g., multiple levels of instruction and data caches, translation look-aside buffers, and prediction tables. For instance, 30% of the Alpha 21264 chip area, 50% of the Pentium 4 chip area, and 60% of the StrongARM chip area are allocated to cache structures [3], [4].

Cache memories are in large part based on Static Random Access Memory (SRAM) arrays. At feature sizes below $65nm$, SRAM cells are strongly impacted by local variability and random defects [5]. Cache memories will therefore be affected by both typical SRAM fault mechanisms and control logic specific faults. Testing is thus essential to guarantee the quality of next generation computers [6]. However, compared to SRAM testing, cache memories provide limited controllability and observability of internal memory arrays.

Even though Built-In-Self-Test (BIST) proved to be very useful to test memories, the area overhead for relatively small blocks such as caches of embedded processors is not negligible [7]. Software Based Self Test (SBST) represents a very attractive test alternative for microprocessor testing [8], [9]. SBST does not aim at replacing traditional BIST approaches, but at increasing and supplementing them to reach higher test quality. During SBST the processor executes self-test programs from the on-chip cache or from the system's memory. The primary advantage of SBST is its non-intrusive nature and its capability of using processor resources and Instruction Set Architecture (ISA) to test the processor itself. It minimizes the need for high-cost functional testers, enables at-speed testing, and can be re-used for periodic on-line testing [10]. Moreover, testing is performed in the processor's normal operation mode without extra power consumption or area overheads.

SBST approaches have been proposed by many research and development groups [11], [12], [13]. Nevertheless, in most of them, cache memory testing is only vaguely mentioned. SBST of cache memories requires several interaction between the processor and the main memory where the test program is stored. While this does not represent a problem for microprocessors embedded into a complex system-on-chip (SoC) where the main memory is usually available at test time, it may introduce strong limitations for testing stand-alone processors where the main memory should be available on the test board, or at least properly emulated by the test equipment.

This paper proposes a methodology to exploit the ISA of a processor to translate generic march tests [14] into SBST programs for set-associative cache memories. Besides concentrating on the definition of a general test program, the paper focuses on defining guidelines to produce test programs customized for specific microprocessors, and for specific classes of faults. Among all the different cache blocks in a microprocessor, testing instruction caches represents a major challenge due to limitations in two areas: (i) test patterns which must

S. Di Carlo, P. Prinetto, and A. Savino are with the Department of Control and Computer Engineering, Politecnico di Torino, Corso Duca degli Abruzzi 24, Torino, Italy. E-mails: {stefano.dicarlo, paolo.prinetto, alessandro.savino}@polito.it.

Manuscript received x xxx. xxxx; revised xx xxx. xxxx; accepted xx xxx. xxxx; published online xx xxxx xxxx.
Recommended for acceptance by xxxx.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2009-06-0250.
Digital Object Identifier no. xxx

be composed of valid instruction opcodes; and (ii) test result observability: the results can only be observed through of the results of executed instructions. The proposed methodology will therefore concentrate on testing this type of memories. The main contribution of this work lies in the possibility of applying state-of-the-art memory test algorithms to embedded cache memories without introducing any hardware or performance overheads. While this work focuses on testing the cache memory arrays, we will also show how the generated test programs guarantee a reasonable fault coverage on the control logic. The results of this research have been used to implement a test program for the instruction cache of the LEON3 microprocessor [15].

The paper is organized as follows: Section 2 presents related works in the field of cache memory testing, while Section 3 introduces basic concepts about cache memories. Section 4 shows how to build a march test for a set-associative cache memory, while Section 5 shows how to translate this type of march test into a SBST program. Section 6 applies the proposed test strategy to the LEON3 microprocessor, and finally Section 7 summarizes the main contributions of the paper and outlines future works.

2 Related Works

Previous solutions for in-system testing of cache memories fall into two main categories: hardware-based solutions, and software-based techniques.

2.1 Hardware-based solutions

Hardware-based solutions mainly rely on SRAM self-testing techniques.

Pei et al. [16] propose an exhaustive study of new functional fault models for drowsy SRAM cache memories and a new march test (March DWOM) designed for memory BIST implementations.

Bhunia at al. [17] propose the modification of the cache structure to improve IDDQ (quiescent supply current) testing sensitivity.

Tan et al. [18] propose a memory BIST device able to apply an improved March C [14] to L1 and L2 caches.

Agrawal et al. [19] propose a process variations tolerant cache architecture to improve yield in nanoscale technologies that resorts to an internal BIST facility to identify faulty cache lines.

The main drawback of these solutions is that they require considerable modifications of the initial cache design. Moreover, to limit hardware and performance overheads, they implement simple march tests only, not considering emerging SRAM fault models [20].

2.2 Software-based solutions

Several solutions for cache testing have been proposed as part of microprocessor SBST. They consider embedded cache memories and face the problem of their limited accessibility. Raina et al. [21] present a random approach for testing the PowePC microprocessor cache. Verhallen et al. [22] introduce a systematic approach to on-chip cache testing as part of the memory subsystem of a microprocessor, and apply it to the Intel i860. Bhavsar at al. [23] suggest a solution for SBST of the data cache of a microprocessor, whereas the instruction cache is tested by BIST techniques. The main drawbacks of these solutions are that cache testing is only outlined and only simple test algorithms (e.g., March B [14]) are proposed.

Sosnowski [24] generalizes the approach proposed in [22] introducing a march-like test algorithm taking into account whether the cache is used to store data or instructions. This solution has the main advantage of considering both data and instruction caches. Moreover, it can easily be translated into as SBST program. Its main drawback is that it proposes a predefined test algorithm, and its extension to more complex fault models is not clear. This solution is improved in [25] exploiting available microprocessor circuitry to increase test observability (e.g., performance monitors and on-line error detectors).

Tuna et al. [26] focus on developing a test program for the data array of an instruction cache, providing an interesting analysis of the test cost in terms of program size and test time. Unfortunately, both the directory array of the cache and the control circuitry are neglected. Moreover, the implemented test algorithms only target simple fault models such as stuck-at faults and transition faults.

An SBST algorithm devoted to test the cache control circuitry is proposed in [27]. This technique relies on an accurate timer to validate cache memory operations. Whenever an accurate timer is not available an alternative solution based on the use of an Infrastructure Intellectual Property core (I-IP) is proposed [28].

Some publications propose the transformation of march-like test sequences to test cache memories. Al-Harbi et al. [29] present a methodology to translate generic march tests into test sequences for the directory array of cache memories. The procedure allows the same fault detection of the original test to be preserved but it is limited to direct mapped caches with write-back policy. Although the authors mention that this translation can be extended to set-associative memories, they do not show how this could be done, and they neglect both write-through cache memories and the transformation of translated march tests into SBST programs. In [30] we introduced a methodology to translate generic march tests into equivalent versions for testing the memory arrays of set-associative cache memories with both write-back and write-through policy. While the paper provides a theoretical foundation to build efficient test procedures, the transformation of the obtained test algorithms into an SBST program for instruction caches is not considered.

To the best of our knowledge, a general methodology for writing SBST programs for the instruction cache of a microprocessor, taking into account user defined fault

lists, is still missing.

3 Cache Memory Overview

Cache memories speed up the microprocessor memory access by storing recently used data [31]. During every reference to main memory, the cache checks whether it already stores the requested information (*cache hit*), or not (*cache miss*). In the former case the information is directly delivered to the processor, while in the latter case it is pulled up from the underlying memory subsystem. The cache internal organization comprises two memory arrays managed by a cache controller circuitry (Fig. 1). The *data array* holds the actual cached information, e.g., programs' instructions or data, while the *directory array* is a small and fast memory that stores portions of the address of cached memory locations (*tags*). To minimize the number of requests to the memory subsystem the minimum allocation unit of the data array, called *cache line,* holds a set of nW memory words.

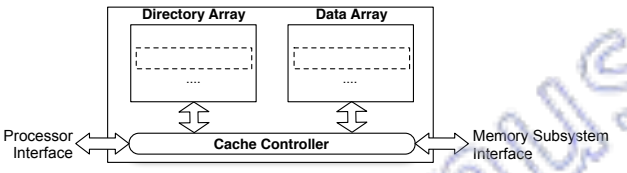

Figure 1. General cache architecture

The way a cache stores data depends on its mapping schema, the most used ones being *set-associative*, and *direct-mapped*. A set-associative cache memory (Fig. 2) is organized into nS sets, each one containing nL cache lines also referred to as *ways*. Each main memory location is mapped to a set, and can be stored in any of its nL cache lines. To deal with this schema a N bit memory address $A = [t, \alpha, \beta]$ is split into three portions:

- the *offset* (β), represented by the $O = log_2(nW)$ less significant bits of A, identifying a specific word of the cache line,
- the *index* (α), represented by the middle $I = log_2(nS)$ bits of A, identifying the set where the desired information can be stored, and
- the *tag* (t), represented by the $T = N - I - O$ most significant bits of the address, identifying the content of the directory array used to tag the cached information.

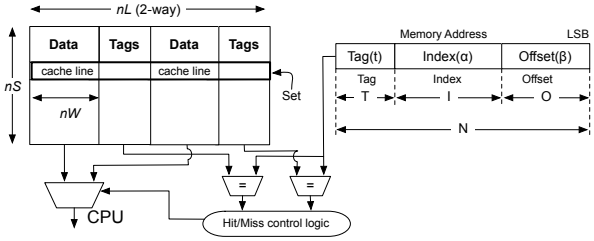

Figure 2. 2-way set-associative cache memory

Every time the microprocessor accesses a memory word with address $W = [t_w, \alpha_w, \beta_w]$, the set α_w is searched associatively for t_w. If a cache hit occurs, the desired memory word is identified in the target cache line through the offset β_w. Otherwise, (cache miss) the cache line containing the desired information is loaded from the memory subsystem. If a free cache line in the target set is available it is used to store the desired information; if not, a cache line is discarded and replaced by the desired one. Among the different policies used to identify the cache line to replace in case of cache miss we consider the LRU (*Least Recently Used*) replacement algorithm that discards the least recently used lines, first.

A direct-mapped cache is a special instance of a set-associative cache with $nL = 1$. In a direct-mapped cache each memory address is associated with a single cache line. This simplifies the cache structure (no need for associative search into the set) at the cost of reduced efficiency.

Several read and write policies can be implemented in a cache memory for both cache hits or misses. Whenever a cache miss occurs during a read operation, the following policies can be implemented:

- *read-through*: while the target cache line is loaded, as soon as the desired word is available it is directly transferred from the main memory to the CPU, and
- *no-read-through*: the target cache line is first loaded into the cache, and then the required information is transferred to the CPU.

There are two common policies on write hits:

- *write-back*: when the system writes to a cached memory location it only updates the cache content. When the cache line is replaced by a new one the updated words are *written back* into main memory. This type of cache provides high performances because it saves on time-consuming write cycles to memory;
- *write-through*: when the system writes to a cached memory location it updates both the cache and the main memory at the same time. This solution provides worse performances than write-back, but it is simpler to implement and guarantees consistency between cache and main memory. It is therefore particularly suited for multiprocessor architectures.

The write miss has also two common options:

- *write-allocate*: when the system writes to a non-cached memory location it loads the block into the cache and then starts a write-hit action, and
- *no-write-allocate*: when the system writes to a non-cached memory location it modifies the block directly into the main memory without loading the block into the cache.

All combinations of hit and miss policies are feasible. Write-back caches usually use write-allocate miss policy whereas write-through caches often use no-write-allocate policy.

4 Applying March Tests to Cache Memories

March tests are probably the most used class of memory test algorithms [14], [32], [33]. They detect faults by performing write and read&verify operations on the target memory.

A *march test* is a sequence of *march elements* (ME) composed of memory operations applied to each cell of the memory. After all operations of a ME have been applied to a cell, they are applied to the next one determined by the considered *addressing order* (AO) ($\Uparrow$ for the ascending AO, $\Downarrow$ for the descending AO, $\Updownarrow$ for any AO).

The application of march tests to cache memories is not trivial due to limited controllability and observability of the internal arrays. At a low level, the memory arrays are usually organized into banks of SRAM cores. This organization influences the way faulty behaviors manifest, and therefore the way test algorithms have to be applied. To cope with the worst case this paper considers single bank memory arrays. In this case all cells of the core may equally participate to different coupling fault mechanisms.

Let us consider a set-associative cache organized into nS sets, each containing nL cache lines. To apply a march test to this memory the correct AO should be generated to "*march*" both through the sets of the cache, and, for each set, through the different cache lines. Given a generic memory address $A = [t, \alpha, \cdot]$, where the offset β is not specified since the entire cache line is considered as a single memory word, the index α allows us to address the sets of the cache in the desired order.

More complex is addressing cache lines of a single set since they are directly managed by the cache replacement algorithm. Considering an empty cache with LRU replacement strategy (see Section 3) nL consecutive write operations on the same set using distinct tags will address nL different cache lines in a sequence, thus filling the set completely. Once the set is full the same sequence of tags can be used to produce several times the same addressing sequence into the set. This strategy can be used to build a specific cache addressing order as follows:

Definition 1: A *way-in-index addressing order (WIIAO)* is a cache specific addressing mechanism allowing to address in the desired way the different sets of the cache and, for each set, the different cache lines. Two WIIAOs can be defined:

- *ascending*, denoted as $\uparrow_{\alpha=0}^{nS-1}\uparrow_{i=0}^{nL-1}(\,)$, where α and i define the address $A = [t_i \in T, \alpha, \cdot]$ on which to apply cache operations, and
- *descending*, denoted as $\downarrow_{\alpha=nS-1}^{0}\downarrow_{i=nL-1}^{0}(\,)$.

In both cases T is an n-tuple of nL distinct tags used to address the cache lines composing a set:

$$T = \langle t_0, t_1, \ldots, t_{nL-1} \rangle \; | t_i \neq t_j, \forall i, j \in [0, nL-1] \wedge j \neq i \tag{1}$$

The addressing sequence generated by the WIIAO changes based on the initial state of the cache, and on the first addressing order used in the test algorithm. Nevertheless, according to the march test theory [14], the addressing sequence identified by the ascending WIIAO is always the opposite of the one generated by the descending one. Fig. 3 shows an example of sequences for a cache with two ways ($nL = 2$) and two sets ($nS = 2$). The values the different tags assume depend on the target memory array, i.e., data array vs. directory array, and will be detailed in the next subsections.

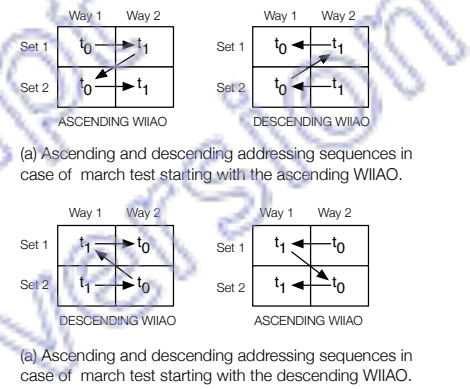

Figure 3. Examples of way-in-index addressing sequences

With the introduction of the WIIAO, the following basic cache operations can be used to build cache memories march tests:

- $w([t, \alpha, (all \mid any \mid \beta)], DB)$: represents a *cache write operation* of DB into the cache line with address $[t, \alpha, (all \mid any \mid \beta)]$. The offset of the address may assume three different values:
  - all: the full cache line is considered,
  - any: any word in the cache line can be used, and
  - β: the word with offset β is considered.
- $r([t, \alpha, (all \mid any \mid \beta], DB)$: represents a *cache read&verify operation*. The cache line with address $[t, \alpha, (all \mid any \mid \beta)]$ is read and compared with the expected pattern DB. Again the offset is used to decide whether the full cache line or a single word of it should be considered.
- $r([t, \alpha, (all \mid any \mid \beta])])$: represents a simple cache read operation. It behaves as a read&verify operation but the result of the read is not verified.

In all the above defined operations DB represents a data background pattern [34], i.e., a generic sequence of bits written in the data array of the cache. For each DB a complemented pattern $\overline{DB}$, obtained from DB by complementing its bits, must also be defined. The size of DB depends whether the corresponding operation is working on a complete cache line or on a portion of it.

Looking at Fig. 3 it is clear that, even if cache lines are organized in sets and ways, the WIIAO allows us to address them linearly, i.e., similarly to the way a SRAM addresses memory words during the execution of a march test. By properly implementing the previously defined cache memory operations it is therefore possible to produce test sequences addressing the same type of faults addressed by traditional march tests, i.e., single cell faults, and coupling faults among cache lines either located in a single set or in different sets. Moreover, by translating march tests developed for word-oriented memories [34], it is possible to detect intra-word faults in a single cache line.

4.1 Data array

By using the WIIAOs and the cache operations defined in Section 4 translating a march test into an equivalent version able to test the data array of a set-associative cache memory is straightforward. Eq. 2 introduces a set of rewriting rules providing a one to one correspondence between the traditional march test notation and the cache memory notation.

$$\begin{aligned} w_1 \rightarrow w([t_i,\alpha,all],DB);\, w_0 \rightarrow w([t_i,\alpha,all],\overline{DB}) \\ r_1 \rightarrow r([t_i,\alpha,all],DB);\, r_0 \rightarrow r([t_i,\alpha,all],\overline{DB}) \\ \Uparrow \rightarrow \uparrow_{\alpha=0}^{nS-1}\uparrow_{i=0}^{nL-1}(\,);\ \Downarrow \rightarrow \downarrow_{\alpha=nS-1}^{0}\downarrow_{i=nL-1}^{0}(\,) \end{aligned} \tag{2}$$

The only consideration about this translation concerns the definition of the set T of tags to use in the WIIAO (eq. 1). Eq. 3 shows an example of tags for a cache with four ways. The value of each tag is not relevant. In this test tags represent an indirect addressing mechanism allowing to address cache lines of a set. The actual addressing is managed by the replacement algorithm (LRU in our case) that allocates cache lines based on the diversity of tags and not on their absolute value. Obviously, faults can influence this mechanism but, since tags are stored in the directory array, they will be considered separately in Section 4.2. The value of each tag can thus be freely chosen to fit the size of the main memory on the target system.

$$\begin{aligned} T = \langle t_0 = \text{"}0\ldots00\text{"}, t_1 = \text{"}0\ldots01\text{"}, \\ t_2 = \text{"}0\ldots10\text{"}, t_3 = \text{"}0\ldots11\text{"}\rangle \end{aligned} \tag{3}$$

This translation schema allows the application of any type of march test to the data array of a set-associative cache memory with LRU replacement.

Fig. 4-b shows the translation of the SOA March C- - [35] of Fig. 4-a.

4.2 Directory array

The first important thing to highlight when testing the directory array of a cache is that both write and read operations can be only performed in an *indirect* way working on the data array. Tags represent the actual test patterns. They need to change properly in order to produce the desired marching sequence. Eq. 4 introduces a preliminary set of rewriting rules to translate march tests for the directory array (the symbol '$-$' denotes that the value written in the cache line is not relevant).

$$\begin{aligned} w_1 \rightarrow w([t_i,\alpha,all],-);\, w_0 \rightarrow w([\overline{t_i},\alpha,all],-) \\ r_1 \rightarrow r([t_i,\alpha,all],-);\, r_0 \rightarrow r([\overline{t_i},\alpha,all],-) \\ \Uparrow \rightarrow \uparrow_{\alpha=0}^{nS-1}\uparrow_{i=0}^{nL-1}(\,);\ \Downarrow \rightarrow \downarrow_{\alpha=nS-1}^{0}\downarrow_{i=nL-1}^{0}(\,) \end{aligned} \tag{4}$$

Again, the definition of the set T of tags to use in the WIIAO is the key element of this translation. Since tags are used as test patterns, according to the march test theory, both a pattern (tag $t_i \in T$), and a complemented pattern (tag $\overline{t_i}$) have to be defined. Eq. 5 shows an example of tags for a cache with four ways.

$$\begin{aligned} T = \langle t_0 = \text{"}0\ldots00\text{"}, t_1 = \text{"}0\ldots01\text{"}, \\ t_2 = \text{"}0\ldots10\text{"}, t_3 = \text{"}0\ldots11\text{"}\rangle \\ \overline{T} = \langle \overline{t_0} = \text{"}1\ldots11\text{"}, \overline{t_1} = \text{"}1\ldots10\text{"}, \\ \overline{t_2} = \text{"}1\ldots01\text{"}, \overline{t_3} = \text{"}1\ldots00\text{"}\rangle \end{aligned} \tag{5}$$

Unfortunately, the translation proposed in eq. 4 is somehow limited by the cache replacement mechanism. Two are the main limitations:

1) It is not possible to perform two write operations, or a read followed by a write operation, with different tags on the same cache line. In fact, the first access to the line (either read or write) marks it as the most recently used (MRU) in the set. Based on the LRU policy any tentative of writing a new tag not already contained in the set will cause the LRU line to be replaced (i.e., written). This line will be for sure different from the one marked as MRU;
2) It is not possible to write cache lines belonging to a set in reverse addressing order. Cache lines of a set are written according to the LRU replacement process. Let us consider a cache with two ways. According to Def. 1, two consecutive write operations on the same set, performed with distinct tags, will address two different cache lines. At the end of the sequence the first addressed line will be the LRU

(a) SOA March C - -

$$\underset{M_0}{\Uparrow(w1)};\ \underset{M_1}{\Uparrow(r1,w0,w1)};\ \underset{M_2}{\Uparrow(r1,w0)};\ \underset{M_3}{\Uparrow(r0,w1,w0)};\ \underset{M_4}{\Uparrow(r0)}$$

(b) Data Array Cache SOA March C - -

$$\begin{aligned} M_0 &: \uparrow_{\alpha=0}^{nS-1}\uparrow_{i=0}^{nL-1}(w([t_i,\alpha,all],DB)) \\ M_1 &: \uparrow_{\alpha=0}^{nS-1}\uparrow_{i=0}^{nL-1}(r([t_i,\alpha,all],DB), w([t_i,\alpha,all],\overline{DB}) \\ &\quad w([t_i,\alpha,all],DB)) \\ M_2 &: \uparrow_{\alpha=0}^{nS-1}\uparrow_{i=0}^{nL-1}(r([t_i,\alpha,all],DB), w([t_i,\alpha,all],\overline{DB})) \\ M_3 &: \uparrow_{\alpha=0}^{nS-1}\uparrow_{i=0}^{nL-1}(r([t_i,\alpha,all],\overline{DB}), w([t_i,\alpha,all],DB), \\ &\quad w([t_i,\alpha,all],\overline{DB})) \\ M_4 &: \uparrow_{\alpha=0}^{nS-1}\uparrow_{i=0}^{nL-1}(r([t_i,\alpha,all],\overline{DB})) \end{aligned}$$

Figure 4. Translation of SOA March C - - for the data array

one. Any attempt of writing a new tag will address this line, thus reproducing the same sequence and making impossible to reverse the addressing order.

A simple solution to tackle with these problems is to introduce a set of additional read operations to artificially modify the access time of the cache lines of a set, thus obtaining the desired addressing sequence. For this purpose, we introduce an additional march test operation called *reordering* (RO) defined as:

$$RO(\alpha P_t) = \{r([t_k, \alpha, any], -), \forall t_k \in P_t\} \quad (6)$$

where P_t is the set of tags stored in the set α associated to cache lines with access time older than the one associated to t. This operation is implemented as a sequence of read operations on those cache lines with access time older than the one containing the tag t. At the end of the sequence the line containing t will be therefore the LRU one assuring the possibility of performing a write operation on it. Reordering operations should be included before each write operation of the march test with the following two exceptions:

- if the first march element starts with a write operation and has an ascending addressing order, the write operation does not require the reordering;
- if the march element begins with a write operation and its addressing order is equal to the addressing order of the previous march element, the first write operation does not require the reordering.

The reordering obviously increases the march test complexity. Nevertheless, the number of additional read operations is proportional to the number nL of cache lines per set that is usually lower then the number of sets of the cache. A possible way to keep this overhead as low as possible is to use Single Order Addressing (SOA) march tests [35] to save the re-ordering sequence at the beginning of march elements starting with write operations.

In addition to the addressing problem, read&verify operations represent another critical element of the proposed translation schema. In fact, the value of a tag is not directly re lable. However, faults into the directory array lead to a set of stored tags different from the expected one. Any attempt of reading the content of lines associated to a missing tag will generate an unexpected cache miss that can be used as a fault detection condition. The problem is therefore shifted to the identification of unexpected cache misses.

In general a read operation can detect a cache miss if there exists a difference between the content of the target cache line and the content of the corresponding main memory locations. In this situation, in case of miss, the value read from the main memory is different from the one expected from the cache and the miss can be easily identified. Depending on the write policy of the cache this condition can be implemented in two different ways.

In a cache with write-through policy, the cache content is always consistent with the memory content. This is the worst situation. The only way to introduce a difference between the cache and the main memory is to temporarily disable the cache, and to directly write the desired value in main memory. Most of the modern microprocessors (e.g., Pentium, PowerPC, Sparc) allow this operation using particular instructions. This, in turn, requires to modify the translation rules proposed in eq. 4 according to eq. 7, where the notation $[\cdot]_M$ identifies an operation performed directly in main memory without using the cache. In this new translation schema data background patterns (DB) are used to identify a cache miss.

$$\begin{aligned} w_1 &\rightarrow w([t_i, \alpha, all], DB) \\ w_0 &\rightarrow w([\overline{t_i}, \alpha, all], \overline{DB}) \\ r_1 &\rightarrow \left[w([t_i, \alpha, all], \overline{DB})\right]_M r([t_i, \alpha, all], DB) \\ r_0 &\rightarrow \left[w([\overline{t_i}, \alpha, all], DB)\right]_M r([\overline{t_i}, \alpha, all], \overline{DB}) \end{aligned} \quad (7)$$

In the case of write-back the policy already comprises the possibility of having the cache and the main memory in an inconsistent state. In this type of caches the memory is written back only in case of replacement. Let us consider the following operation $w([t_i, \alpha, all], DB)$ that writes DB and t_i in a cache line of the set α. Whenever this cache line is replaced by a new one owing a different tag t_j the main memory at address $[t_i, \alpha, all]$ is updated with DB. If now a new operation $w([t_i, \alpha, all], \overline{DB})$ on the same line but with different pattern $\overline{DB}$ is performed, the cache is updated while the main memory at address $[t_i, \alpha, all]$ remains equal to DB thus creating the required difference.

This situation can be generated by the following modifications to the translation rules proposed in eq. 4:

- each write operation must use a data background pattern complemented with respect to the one used by the previous write of the same type, e.g., a march test containing the following operations $\Uparrow\ w_0 \dots w_0 \dots$ will be translated into $\uparrow_{\alpha=0}^{nS-1} \uparrow_{i=0}^{nL-1} \left(w(\left[\overline{t_i}, \alpha, all\right], \overline{DB}) \dots w(\left[\overline{t_i}, \alpha, all\right], DB)\right)$, and
- all main memory locations involved in the test should be initialized with a data background pattern complemented with respect to the one used by the first write operation of the march test. This can be performed by introducing a set of initialization march elements.

Given this constraints, the read&verify can be implemented by reading the cache line and checking that the obtained value is the one expected in the cache and not the one contained in the main memory. Some microprocessors also provide hardware features to count the number of cache misses both in the data and in the instruction cache, thus providing an efficient solution to reduce the test complexity. For example, the Intel Pentium family provides a set of performance monitoring facilities (`RDMSR` instruction) that include the possibility

of counting the number of cache misses during the execution of a sequence of instructions [3]. Obviously this solution is applicable both to write-back and write-through caches.

Fig. 5 shows the translation of the SOA March C-- for testing the directory array of both write-through (a) and write-back (b) cache memories. Fig. 6 shows the application of the first portion of the translated march test to a write-back cache memory with $nS = 1$, and $nL = 2$ where a faulty entry in the directory array transforms the tag t_0 into a different tag t_0^* during a write operation. Elements in bold identify addressed cells. Steps 1 to 4 execute the initialization march elements M_{-2} and M_{-1}. They initialize the memory with a pattern $\overline{DB}$ complemented w.r.t. the one used in M_0. Steps 5 and 6 execute M_0. The fault is sensitized in step 5. Finally Step 7 detects the fault since the read operation returns C instead of the expected value DB. C represents here a value used to initialize the main memory when the test starts.

5 From March Test to SBST

The march test translation methodology proposed in Section 4 provides a theoretical framework to build march tests ready to be used on a generic cache memory. Nevertheless, additional effort is required to translate them into SBST programs. While the translation for a data cache by mapping each cache read/write operation into a sequence of load/store instructions with opportune addresses and data patterns is str..ghtforward, building a test program for an instruction cache is a challenging problem. The proposed approach exploits the Instruction Set Architecture (ISA) of the microprocessor to build basic read and write operations. For the sake of generality we will consider a generic ISA containing instructions commonly available on most of the modern microprocessor families. A specific and detailed implementation for a target case study will be provided in Section 6.

From the test program point of view the instruction cache is a read-only memory: instructions are fetched from the memory subsystem into the cache during the program exec tion and never written-back into memory. Each cache line (CL) is therefore composed of a sequence of valid instructions and can be represented as a n-tuple:

$$CL =< i_1, i_2, \ldots, i_z >, i_i \in ISA \quad (8)$$

Depending on the microprocessor architecture, the number z of instructions that fit a cache line can be fixed (RISC microprocessors) or variable (CISC microprocessors). This strongly affects the way backgrounds patterns (Sections 4.1 and 4.2) used to build march tests can be defined, and therefore the achievable test coverage. Moreover, read and write operations have a completely different meaning w.r.t. the one they have in a data cache.

A cache line is written whenever the microprocessor executes an instruction stored in a non-cached memory location (cache miss). This event forces the cache to load the full line (see Section 3). A write operation can be therefore implemented by forcing the test program to enter a non-cached memory area. A read operation occurs when cached instructions are actually executed. To verify the result of a read operation *(read&verify)*, instructions should be defined in such a way to produce unambiguous final results that, compared with the expected ones, allow to perform the verification. The implementation of the read&verify operation represents one of the main issues for the translation of a cache march test into a SBST program.

Additional issues are posed by the correct implementation of the addressing mechanism. During the execution of the test program, when the last instruction of a cache line is completed, the program moves to the next one. This involves accessing a new cache line whose address may violate the desired marching sequence. It is therefore mandatory to structure the test program in such a way to precisely respect the structure of the original march test. A very efficient solution to this problem consists in splitting the test program into two sections, as reported in Fig. 7:

1) a *control program* implementing the addressing sequence and the read verifications. It must be stored in a *not-cacheable* memory area to avoid corruption of test patterns applied to the cache, and
2) a *test patterns* section storing the actual patterns, i.e., sequences of instructions to be loaded in cache representing the actual test operations. This code must be stored in a *cacheable* portion of the main memory.

Being able to control the cacheability of different memory areas is a key requirement for the implementation of an efficient SBST program. Modern microprocessors provide different mechanisms to implement this feature, thus making this solution widely applicable to several real cases. The program partitioning proposed in Fig. 7 can be either logical or physical depending on the way cacheability is controlled on the microprocessor.

Given this structure each march element of a cache march test can be translated into a control program composed of two nested loops implementing the WIIAO, containing, for each memory operation a call to a specific function:

- *write(tag,index,DB)*: based on the target cache line it forces the control program to jump to the memory area (*target*) containing the appropriate test pattern (DB). The target is calculated based on both the tag and the index involved in the operation. The target cache line should be organized in such a way to contain instructions able to immediately return the control to the write function after the cache line being completely loaded in cache. A cache line can be therefore split into two portions (eq.9): (i) a Test Instructions Sequence (TIS) including instructions required, when executed, to perform a read operation, and (ii) a Control Instructions

(a) Directory array SOA March C - - for write-through cache

$$M_0 : \uparrow_{\alpha=0}^{nS-1}\uparrow_{i=0}^{nL-1} \left(w([t_i,\alpha,all],DB)\right)$$
$$M_1 : \uparrow_{\alpha=0}^{nS-1}\uparrow_{i=0}^{nL-1} \left(\left[w([t_i,\alpha,all],\overline{DB})\right]_M, r([t_i,\alpha,all],DB), RO(P_{\bar{t_i}}), w(\left[\overline{t_i},\alpha,all\right],\overline{DB}), RO(P_{t_i}), w([t_i,\alpha,all],DB)\right)$$
$$M_2 : \uparrow_{\alpha=0}^{nS-1}\uparrow_{i=0}^{nL-1} \left(\left[w([t_i,\alpha,all],\overline{DB})\right]_M, r([t_i,\alpha,all],DB), RO(P_{\bar{t_i}}), w(\left[\overline{t_i},\alpha,all\right],\overline{DB})\right)$$
$$M_3 : \uparrow_{\alpha=0}^{nS-1}\uparrow_{i=0}^{nL-1} \left(\left[w(\left[\overline{t_i},\alpha,all\right],DB)\right]_M, r([\bar{t_i},\alpha,all],\overline{DB}), RO(P_{t_i}), w([t_i,\alpha,all],DB), RO(P_{\bar{t_i}}), w([\bar{t_i},\alpha,all],\overline{DB})\right)$$
$$M_4 : \uparrow_{\alpha=0}^{nS-1}\uparrow_{i=0}^{nL-1} \left(\left[w(\left[\overline{t_i},\alpha,all\right],DB)\right]_M, r(\left[\overline{t_i},\alpha,all\right],\overline{DB})\right)$$

(b) Directory array SOA March C - - for write-back cache. M_{-1} and M_{-2} are the initialization march elements required to implement read&verify operations for write-back caches

$$M_{-2} : \uparrow_{\alpha=0}^{nS-1}\uparrow_{i=0}^{nL-1} \left(w([t_i,\alpha,all],\overline{DB})\right)$$
$$M_{-1} : \uparrow_{\alpha=0}^{nS-1}\uparrow_{i=0}^{nL-1} \left(w(\left[\overline{t_i},\alpha,all\right],\overline{DB})\right)$$
$$M_0 : \uparrow_{\alpha=0}^{nS-1}\uparrow_{i=0}^{nL-1} \left(w([t_i,\alpha,all],DB)\right)$$
$$M_1 : \uparrow_{\alpha=0}^{nS-1}\uparrow_{i=0}^{nL-1} \left(r([t_i,\alpha,all],DB), RO(P_{\bar{t_i}}), w(\left[\overline{t_i},\alpha,all\right],DB), RO(P_{t_i}), w([t_i,\alpha,all],\overline{DB})\right)$$
$$M_2 : \uparrow_{\alpha=0}^{nS-1}\uparrow_{i=0}^{nL-1} \left(r([t_i,\alpha,all],\overline{DB}), RO(P_{\bar{t_i}}), w(\left[\overline{t_i},\alpha,all\right],DB)\right)$$
$$M_3 : \uparrow_{\alpha=0}^{nS-1}\uparrow_{i=0}^{nL-1} \left(r([\bar{t_i},\alpha,all],\overline{DB}), RO(P_{t_i}), w([t_i,\alpha,all],DB), RO(P_{\bar{t_i}}), w(\left[\overline{t_i},\alpha,all\right],DB)\right)$$
$$M_4 : \uparrow_{\alpha=0}^{nS-1}\uparrow_{i=0}^{nL-1} \left(r(\left[\overline{t_i},\alpha,all\right],DB)\right)$$

Figure 5. Translation of SOA March C - - for the directory array

Step	Operation	Cache Memory				Main Memory				
		T	D	T	D	$[t_0,\alpha,all]$	$[t_1,\alpha,all]$	$[\overline{t_0},\alpha,all]$	$[\overline{t_1},\alpha,all]$	$[t^*,\alpha,all]$
1	$M_{-2} : w([t_0,1,all],\overline{DB})$	$\boldsymbol{t_0^*}$	$\boldsymbol{\overline{DB}}$	?	?	C	C	C	C	C
2	$M_{-2} : w([t_1,1,all],\overline{DB})$	t_0^*	$\overline{DB}$	$\boldsymbol{t_1}$	$\boldsymbol{\overline{DB}}$	C	C	C	C	C
3	$M_{-1} : w(\left[\overline{t_0},1,all\right],\overline{DB})$	$\boldsymbol{\overline{t_0}}$	$\boldsymbol{\overline{DB}}$	t_1	$\overline{DB}$	C	C	C	C	$\boldsymbol{\overline{DB}}$
4	$M_{-1} : w(\left[\overline{t_1},1,all\right],\overline{DB})$	$\overline{t_0}$	$\overline{DB}$	$\boldsymbol{\overline{t_1}}$	$\boldsymbol{\overline{DB}}$	C	$\boldsymbol{\overline{DB}}$	C	C	$\overline{DB}$
5	$M_0 : w([t_0,1,all],DB)$	$\boldsymbol{t_0^*}$	$\boldsymbol{DB}$	$\overline{t_1}$	$\overline{DB}$	C	$\overline{DB}$	$\boldsymbol{\overline{DB}}$	C	$\overline{DB}$
6	$M_0 : w([t_1,1,all],DB)$	t_0^*	DB	$\boldsymbol{t_1}$	$\boldsymbol{DB}$	C	$\overline{DB}$	$\overline{DB}$	$\boldsymbol{\overline{DB}}$	$\overline{DB}$
7	$M_1 : r([t_0,1,all],DB)$	$\boldsymbol{t_0^*}$	$\boldsymbol{C}$	t_1	DB	$\boldsymbol{C}$	$\overline{DB}$	$\overline{DB}$	$\overline{DB}$	$\overline{DB}$
8	$M_1 : r([t_0,1,all],DB)$	t_0^*	C	$\boldsymbol{t_1}$	$\boldsymbol{DB}$	C	$\boldsymbol{\overline{DB}}$	$\overline{DB}$	$\overline{DB}$	$\overline{DB}$

Figure 6. Example of test application for a write-back cache with $nS = 1$, $nL = 2$

Sequence (CIS) able to jump back to the control program. CIS represents the actual target address of the jump firing the write operation. The way TIS and CIS are implemented and organized in a cache line depends on the target memory array and will be detailed in the next subsections.

$$CL = \langle \overbrace{< i_1,\ldots,i_j >}^{TIS}, \overbrace{< i_{j+1},\ldots,i_z >}^{CIS} \rangle \quad (9)$$

- *read(tag,index,DB)*: similarly to the write operation the control program resorts to a jump instruction to force the execution of a certain memory area. Since read instructions operate on cache lines previously initialized by a write operation, and therefore already loaded in cache, the execution of the cache line induces a read operation. The same CIS introduced in the write operation allows returning to the control program and starting the verification of the execution results.
- *reordering*(*tag,index*): this function implements the reordering operation RO defined in eq. 6. Since it simply requires to access a set of cache lines in order to update their access time, it can be implemented by a sequence of jumps to the CIS of each cache line of P_{tag} (see eq. 6). Since all the considered lines will already be stored in cache, this operation will just modify their access time leaving the cache content untouched.

Both the directory array and the data array test programs require full access to the complete addressing space of the microprocessor in order to generate the required set of tags, regardless the real main memory size. Modern microprocessors implement memory management facilities (such as Memory Management Units) that can be exploited to overcome this limitation by introducing a *virtual* memory addressing space. Full access to a wide main memory space can be therefore provided even if a small amount of physical memory is available.

Before detailing the test implementation for the two directory arrays, it is worth mentioning that, during the execution of the test, faults into the cache may lead to program's exceptions (e.g., illegal instruction, access violation, etc.). Software exceptions should be therefore considered as the evidence of a fault, and proper exception handling routines should be configured to deal with these conditions.

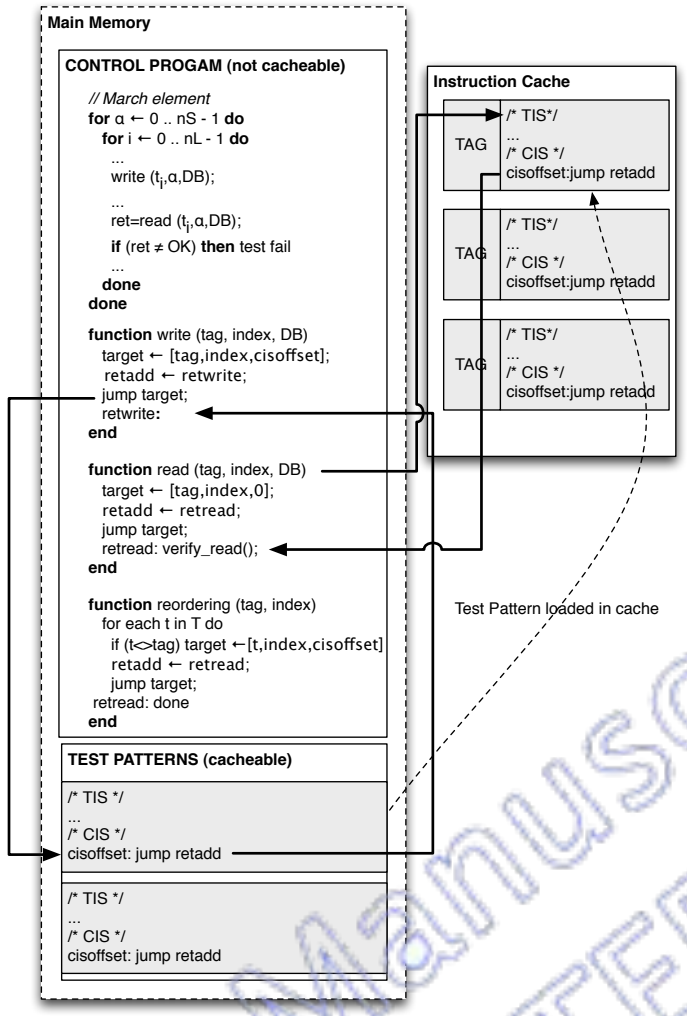

Figure 7. Test program structure

5.1 Data Array Test Patterns

Testing the data array of the instruction cache implies the definition of background patterns composed of microprocessor instruction opcodes and operands. According to the march test theory for each background pattern DB a complemented pattern $\overline{DB}$ should be defined, as well. This, in turn, requires the identification of two $TISs$ (eq. 9) having two properties: (i) the two sequences should be coded by two complemented patterns of bit (TIS, $\overline{TIS}$) and (ii) the execution of each sequence must provide a distinct result.

Fig. 8 graphically shows the structure of two generic sequences of instructions respecting the above mentioned properties: the two sequences of bits are complemented and their execution provides distinct results observable through registers `r0` and `r3`.

The better way to define these sequences is to resort to instructions with immediate operands that provide the maximum freedom in manipulating the instruction code. When choosing the instructions composing the test patterns the internal memory architecture should be also considered. In particular bit swizzling techniques

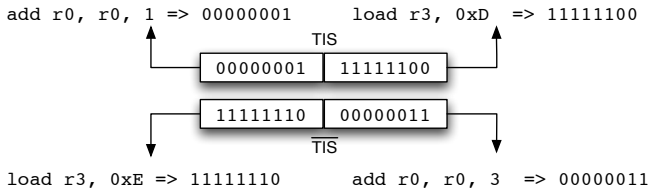

Figure 8. TIS example for the data array

may lead to physical organization of the memory bit different from the logical ones. Test patterns should be constructed respecting march test properties at the physical level.

Whenever the ISA of the microprocessor does not allow the definition of two complemented TIS the test should be executed several times using different patterns to reach the maximum fault coverage, thus increasing the overall test complexity. Nevertheless, modern microprocessors (see Section 6), offering very extensive ISAs, usually provide test designers with enough alternatives for constructing the desired TIS.

Even if an efficient TIS can be defined the CIS reduces the coverage of the march test since it does not respect the properties required to behave as a march test pattern. Both single cell faults and coupling faults affecting the portion of the cache line containing CIS may escape. To cover the full cache line and therefore to maintain the original march test coverage, the test should be executed more than once, properly changing the position of CIS into the cache line. Fig. 9 shows three different configurations able to cover coupling faults between cells belonging to two cache lines, and single cell faults (in this case only two out of three configurations are required). Dashed arrows represent couplings between portions of the lines that cannot be covered by the configuration, while solid arrows represent couplings that can be covered by the configuration. Applying all these configurations together with the WIIAO, both single cell faults and coupling faults among cache lines, either belonging to the same set or to different sets, can be detected.

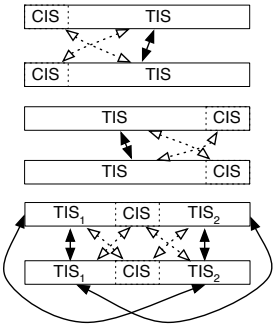

Figure 9. Cache line organization

In order to show how the different configurations influence the execution of the test, Fig. 10 shows the execution of a march test designed to cover a specific

type of idempotent coupling fault expressed by the fault primitive $< 0^a 0^v, w_1^a / 1^v / - >, a < v$. The first execution places CIS at the beginning of the cache line. In this configuration when, during a read operation, TIS is executed, CIS of the next cache line is used to return to the control program. This introduces a partial read of the next cache line that does not reduce the efficiency of the test. In this case, in the memory initialization step, all locations not touched by the test but adjacent to a TIS have to be filled with CIS. In the second execution CIS is placed at the end of the cache line. This is the simplest situation, since the test execution exactly follows the march test flow without introducing extra read operations. The third execution represents the most complex situation. CIS is placed between two $TISs$. This modifies the execution of the read operation that starts executing TIS_1, jumps back to the control program through CIS, and then jumps again to the test to execute TIS_2. After executing TIS_2 a partial read operation of the following cache line is introduced to reach the next CIS and to return to the control program.

The proposed implementation still presents a major problem: how performing two consecutive write operations on the same cache line? Since the instruction cache is a read-only memory the only solution to overcome the problem is to force the cache to replace the desired line by a new content. Two solutions are possible:

1) the control program modifies the content of the main memory to reflect the new DB, if required, and the target cache line is invalidated to force the replacement of its content by the new one. This is a very efficient solution. However, it requires the possibility of invalidating single lines of the cache and disabling the data cache to access the main memory from the control program. In alternative, prefetch instructions may be used. The main problem here is that some of these functionalities might not be available in all microprocessors;
2) different copies of test patterns can be stored in different portions of the main memory. This solution does not require specific cache control instructions,
3) and the invalidation is not required since the pattern is written using a different address. The cache replacement algorithm directly influences the operations. The reordering operation (Section 4) is used to actually invalidate a cache line by changing the access time of the remaining ones.

Both solutions are possible, depending on the microprocessor's ISA. In the former case, once the control program replaces the code and invalidates the cache line, a write operation loads the new pattern in cache. In the latter solution each pattern is stored in a different memory area, thus defining a set of *"equivalent cache patterns"* to be used in consecutive write instructions. Two addresses A_i ($[t_i, \alpha_i, \beta]$), and A_j ($[t_j, \alpha_j, \beta]$) can be used to store equivalent cache patterns iff $\alpha_i = \alpha_j$, and $t_i \neq t_j$.

5.2 Directory Array

The implementation of an SBST program for the directory array does not face the same problems of the data array since, in this case, instructions do not represent test patterns. The code loaded in cache should only be able to produce the desired sequence of tags and to detect faulty conditions. Being the instruction cache a read only memory, similarly to a write-through cache, its content is always consistent with the content of the main memory. According to Section 4.2 faults can be therefore detected by artificially creating a difference between the cache and the main memory content. The general structure of read and write instructions previously presented in this section is therefore still valid:

- *write*: the control program uses a jump instruction to force the cache to load a memory location chosen according to its address. The target location should simply contain an instruction able to immediately jump back to the control program;
- *read*: the control program jumps to a cache line previously loaded into the cache and executes its content. The cache line should contain a set of instructions producing an unambiguous result followed by a CIS to return to the control program. Before performing the read operation, according to Section 4.2, the content of the main memory should be modified writing a set of instructions producing a result different w.r.t. the one produced by the instructions loaded in cache. By checking the result of the computation the control program can verify if the correct tag was loaded (i.e., the instructions in cache have been executed), or not (i.e., a cache miss replaced the cache content with the one stored in memory). To distinguish among different tags the instructions executed during the read operation must produce different results for each defined tag.

6 Testing the LEON Instruction Cache

To prove the concepts introduced in this paper, we applied the proposed test methodology to the LEON3 microprocessor designed by Gaisler Research [15]. It implements a SPARC V8 compliant architecture that can be customized by enabling/disabling hardware modules. Our implementation includes a full integer unit required to compute test addresses, no data-cache, basic cacheability features managed by the AMBA controller, and a 2-way set-associative instruction cache with 32 sets and 8 words per cache line. The cache implements a no-read-through miss policy, i.e., data loaded from the main memory are first cached and then provided to the processor (see Section 3). Each word is composed of 32 bits for a total of 2 KB of data array. This small cache allowed us to reduce the test simulation time even if the proposed implementation is independent on the actual size of the cache. The data cache has been disabled to avoid data latency when the control program requires to update the program's code. This solution simplifies

	Execution 1				Execution 2				Execution 3		
Operation	**Set**	**Tag**	**Data**		**Tag**	**Data**		**Tag**	**Data**		
$M_1 : w\left([t_0, 0, all], \overline{DB}\right)$	0	t_0	CIS	$\overline{TIS}$	t_0	$\overline{TIS}$	CIS	t_0	$\overline{TIS_1}$	CIS	$\overline{TIS_2}$
	1	-	-	-	-	-	-	-	-	-	-
$M_1 : w\left([t_0, 1, all], \overline{DB}\right)$	0	t_0	CIS	$\overline{TIS}$	t_0	$\overline{TIS}$	CIS	t_0	$\overline{TIS_1}$	CIS	$\overline{TIS_2}$
	1	t_0	CIS	$\overline{TIS}$	t_0	$\overline{TIS}$	CIS	t_0	$\overline{TIS_1}$	CIS	$\overline{TIS_2}$
$M_2 : r\left([t_0, 0, all], \overline{DB}\right)$	0	t_0	CIS	$\overline{TIS}$	t_0	$\overline{TIS}$	CIS	t_0	$\overline{TIS_1}$	CIS	$\overline{TIS_2}$
	1	t_0	CIS	$\overline{TIS}$	t_0	$\overline{TIS}$	CIS	t_0	$\overline{TIS_1}$	CIS	$\overline{TIS_2}$
$M_2 : w\left([t_0^*, 0, all], DB\right)$	0	t_0^*	CIS	TIS	t_0^*	TIS	CIS	t_0^*	TIS_1	CIS	TIS_2
	1	t_0	CIS	$\overline{TIS}$	t_0	$\overline{TIS}$	CIS	t_0	$\overline{TIS_1}$	CIS	$\overline{TIS_2}$
$M_2 : r\left([t_0, 1, all], \overline{DB}\right)$	0	t_0^*	CIS	TIS	t_0^*	TIS	CIS	t_0^*	TIS_1	CIS	TIS_2
	1	t_0	CIS	$\overline{TIS}$	t_0	$\overline{TIS}$	CIS	t_0	$\overline{TIS_1}$	CIS	$\overline{TIS_2}$
$M_2 : w\left([t_0^*, 1, all], DB\right)$	0	t_0^*	CIS	TIS	t_0^*	TIS	CIS	t_0^*	TIS_1	CIS	TIS_2
	1	t_0^*	CIS	TIS	t_0^*	TIS	CIS	t_0^*	TIS_1	CIS	TIS_2

write Operation

read& verify operation

read operation

$< 0^a 0^v, w_1^a / 1^v / - >, a < v$

$M_1 : \uparrow_{\alpha=0}^{1} \uparrow_{i=0}^{0} \left(w([t_i, \alpha, all], \overline{DB})\right);$

$M_2 : \uparrow_{\alpha=0}^{1} \uparrow_{i=0}^{0} \left(r([t_i, \alpha, all], \overline{DB}), w([t_i, \alpha, all], \overline{DB})\right);$

Figure 10. March test execution example on a cache memory with $nS = 2$, $nL = 1$

the code without introducing strong assumptions since cache disabling is commonly implemented in modern microprocessors. Internally the memory is organized with a memory bank for each way. The directory array includes a tag portion and a set of 8 validity bits, one for each word of the cache line. No parity bits have been included since they are only available in the fault tolerance version of the LEON3 processor. The data array is organized as a single word array where 8 consecutive words identify a cache line.

The LEON3 microprocessor has some differences with respect to the SPARC V8 specifications that affect the way the test is implemented. It is impossible to invalidate single cache lines. Any time the `flush` instruction is executed both the data and the instruction cache are fully invalidated. For this reason multiple write operations must be implemented by means of equivalent test patterns as proposed in Section 5.1. This actually proves the benefit of having a test methodology not strongly based on specific microprocessor facilities. The cache line replacement is also affected by a side effect on cache lines containing branch instructions. Cache lines load and instructions execution is a concurrent process: as soon as a instruction is available in cache it is executed, while the loading process continues. If the instruction is a branch jumping out of the cache line, its execution immediately stops the loading process, leaving the line content in a not-valid state. The write test pattern proposed in Section 5.1 should therefore be modified to always execute all instructions. This actually turns it into the equivalent of a read operation without verifying the results.

6.1 Test implementation

The most critical part of the data array test implementation is the definition of the instructions implementing TIS and $\overline{TIS}$. This in turn requires a careful analysis of the microprocessor's ISA . Those instructions whose correct execution can be unequivocally verified by looking at the result of the computation should be identified and characterized based on the used resources. Among this set two instructions coded with complemented sequences of bits and operating on different resources should be selected, if possible, to correctly distinguish between the execution of TIS and $\overline{TIS}$. This operation can be performed resorting to an object dumper tool. The use of different operands can be exploited to make instructions producing unique results. Fig. 11 shows the selected patterns for the LEON3 microprocessor:

1) `sethi`: sets the high-order 22 bits of the register provided as second operand using the immediate value given as first operand. The macro `%hi` extracts the most significant 22 bits from its 32 bits argument;
2) `ldsha`: loads into the third operand register a signed 32 bits half-word from an Alternate Space Identifier (ASI) specified by the first two operands (registers `%l5`, and `%l7`). An ASI is a logical memory partition used in the virtual memory management to better control indirect memory addressing.

TIS uses 6 out of 8 instructions available in the cache line, thus leaving two instructions to implement CIS. In the LEON3, for optimization purpose, each jump is always delayed after the execution of the immediately following instruction. This forces us to implement *CIS* with two instructions: the actual jump followed by a `nop` (*no operation*) (see Fig. 11).

Faults in the data array may legally modify opcodes or operands of instructions contained in the patterns, or create illegal configurations rising traps (e.g., illegal instruction, privilege instruction, memory address not aligned, data store error, etc.). In the first case, since TIS and $\overline{TIS}$ operate on different registers, to verify their execution it is enough to check their target registers w.r.t. the expected values, e.g., `cmp %i4, %l2` for TIS and `cmp %l2,%o0` for $\overline{TIS}$, where `%i4` and `%l2` contain the expected value of `%l2` and `%o0`, respectively. In the

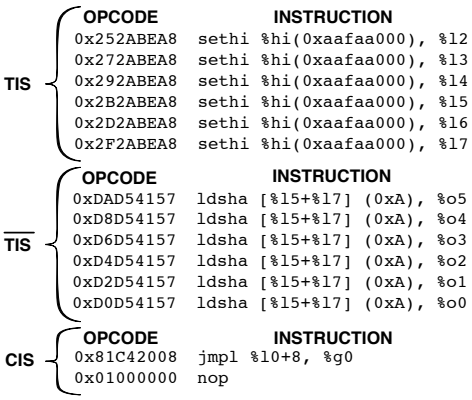

Figure 11. Data Array Patterns

second case, whenever an exception is fired, the program jumps to a trap handler whose address is defined in the LEON3 trap table. To detect these situations all handlers in the trap table should point to an error function able to either stop the test or record the fault.

To better understand the data array test mechanisms, Listing 1 provides the implementation of the data array write function (see Section 5.1). The control program uses the register `%i2` to pass the position of the target CIS to the function. The `prepare_not_TIS` sequence (lines 19 to 23) prepares the registers for the correct execution of $\overline{TIS}$ if required, while the `write_it` sequence (lines 25 to 56) executes the correct jump to the main memory. Each `jmpl` (jump and link) instruction uses its first operand as target address, and saves its current address into the second operand (`%l0` in our case). This address is used by CIS to correctly return to the control program (see CIS in Fig. 11).

According to Section 5.2, the implementation of the directory array test program is straightforward. The only critical point is being able to detect faults by creating a difference between the content of the cache and the content of the main memory. This translates here into the definition of two different sequences of instructions, each producing a different result, to be stored in cache and in main memory, respectively. The LEON3 ISA allows the implementation of this mechanism with a very compact pattern (DB) that exploits the `jmpl` instruction (Fig. 12). A write operation is implemented by jumping to a memory area containing *DB* after storing the return address in `%l0`. In the correct pattern the `jmpl` instruction saves its address in `%l7` before actually performing the jump, while in the incorrect one `%g0` forces the instruction to skip this operation. By checking `%l7` the execution of the proper instruction can be verified.

Listing 2 reports the read&verify function designed for the directory array test. It first prepares the return value (register `%o0` cleared at line 10) and the verification register (register `%l7` cleared at line 11). Then it jumps to the target cache line executing its code and verifies

```
1  !Function write_pattern:
2  !input:
3  ! - 1st (%i0): line address
4  ! - 2nd (%i1): pattern selection
5  !              (0 = TIS, 1 = not TIS)
6  ! - 3rd (%i2): CIS position
7  !              (0 = BEGIN, 1 = MIDDLE
8  !              2 = TOP)
9  ! - 4th (%i3): buffer address
10 !output: no output
11
12 write_pattern:
13 save %sp, -0x40, %sp    !Context Saving
14
15 cmp %i1, 0          !check %i1
16 bne write_it        !to select the pattern
17 nop                 !to use
18
19 prepare_not_TIS:            !Initialize the
20 set 0x000000FF, %l2         !buffer pointed
21 clr %l7                     !by %l5 + %l7
22 mov %i3, %l5                !used by ldsha
23 stha %l2, [%l5 + %l7] 0xA !in not_TIS
24
25 write_it:            !start write...
26 cmp %i2, 0          !check CIS position
27 be _j_at_end   !if 0 CIS at END of CL
28 nop
29
30 cmp %i2, 2          !if 2 CIS at BEGIN of CL
31 be _j_at_begin
32 nop
33
34 _j_at_middle:      !if 1 CIS at MIDDLE of CL:
35 jmpl %i0, %l0      !jump to TIS1
36 nop
37
38 _j_at_middle_part2: !complete the MIDDLE
39 add %i0, 20, %i0    !case by jumping
40 jmpl %i0, %l0       !to TIS2
41 nop
42
43 ba _exit            !return to the control
44 nop                 !program
45
46 _j_at_begin:       !CIS at BEGIN of CL:
47 add %i0, 8, %i0    !jump to TIS placed
48 jmpl %i0, %l0      !immediately after CIS
49 nop
50
51 ba _exit           !return to control
52 nop                 !program
53
54 _j_at_end:         !CIS at END of CL:
55 jmpl %i0, %l0      !jump to TIS then
56 nop                 !return to control program
57
58 _exit:             !return to control program
59 ret                !and restore
60 restore            !context
```

Listing 1. Data Array Write Function

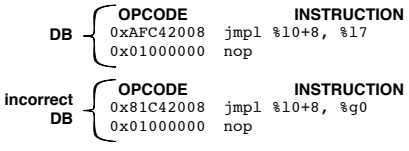

Figure 12. Directory Array Patterns

the cache line address stored in `%l7`. In case of error it jumps to a proper error handling routine (lines 15 to 20). Lines 22 to 27 implement the case of correct verification with the function returning to the control program.

According to the translation theory in Section 4.2 the test of the directory array requires the definition of a

```
1  ! Function read_and_verify_tag:
2  ! input:
3  !   - 1st (%i0): tag address
4  ! output:
5  !   - 1st (%o0): correctness
6
7  read_tag:
8  save %sp, -0x40, %sp ! Context Save
9
10 init_proc: !clearing
11 clr %o0     !output register (%o0)
12 clr %l7     !and check register (%l7)
13
14 jmpl %i0, %l0   !read the line by
15 nop              !jumping on it
16
17 check_return:   !veriry if %i0 = %l7
18 cmp %i0, %l7    !the db was used
19
20 bne on_error    !if incorrect go
21 nop             !to error (%o0 = 0)
22
23 on_success:     !if correct
24 mov 1, %o0      !%o0 = 1
25
26 on_error:
27 ret                      !return and save
28 restore %o0, %g0, %o0 !the output
```

Listing 2. Directory Array Read & Verify Function

```
1  void reordering (unsigned int *
       in_set_tag_list,unsigned int tag,
       unsigned int nL) {
2   int i = 0;
3   while (i < nL) {
4    if (in_set_tag_list[i] != tag)
5     asm ("jmpl %%l7, %%l0;\n" "nop;\n"
6      : "a"(in_set_tag_list[i]));
7    i++;
8  }}
```

Listing 3. Reordering function

proper set of tags. Working with a cache with two ways, two different tags, and their complemented values, have been defined according to eq. 10.

$$T = \langle t_0 = "0\ldots00", t_1 = "0\ldots01" \rangle; \\ \overline{T} = \langle \overline{t_0} = "1\ldots11", \overline{t_1} = "1\ldots10" \rangle \quad (10)$$

Listing 3 shows how the reordering function has been implemented for both the data and the directory array. For each line contained in the target set (line 3), if the stored tag is not the target one (line 4) the function jumps to the CIS of the related cache line (line 5) whose address is loaded in `%l7` (line 6). This updates the cache line's access time.

6.2 Test Coverage and Test Time evaluation

Using the proposed test patterns we translated six different state-of-the-art march tests both for the data and the directory array of the LEON3 cache: SOA Mats+ [35], SOA C -- [35], B [36], U [37], LR [38], and SS [38]. Table 1 reports the test length in terms of CPU cycles and the test size in terms of number of instructions for each test program. The number of instructions refers to the control program since test patterns are treated as data values written in memory during the initialization steps. The test time is quite high, especially if compared with the performance that could be obtained by an embedded memory BIST. Most of this time overhead is generated by the interaction between the control program and the test patterns, and by the fact that the test should be executed three times changing the position of CIS. While test patterns have been carefully designed to guarantee the desired fault coverage, the control program has been partially written in ANSI C language and several optimizations can be introduced to strongly reduce its execution time. The number of instructions is also mainly influenced by the optimization flags of the C compiler and it is not directly related to the original complexity of the march test. Bigger march tests gain more advantage from the optimization that reduces the internal loops.

All test programs have been simulated on a VHDL model of the LEON3 microprocessor using ModelSim [39]. Fault simulation has been performed using the RASTA memory fault simulator. RASTA is a fault primitive based memory fault simulator developed at Politecnico di Torino, whose main characteristics have been published in [40]. All signals at the boundary of the memory arrays have been recorded while performing the ModelSim simulation of the test program. The same signals have been used as test patterns in RASTA to fault simulate the target memory array against the selected faults. The original march test has been also fault simulated on a memory with the same characteristics of the target memory array to allow comparison of the fault coverage. Table 2 shows the test coverage for both the data and the directory array in terms of covered fault models (e.g., 1/2 for TF means that 1 out of 2 possible types of TF are fully covered by the test). The coverage of the original tests has always been preserved.

Considering the directory array, we additionally fault simulated the test coverage on the validity bits portion of the array. Table 3 summarizes the final results. Even if not specifically addressed at design time the test provides a reasonable coverage on the validity bits. Faults are partially covered by the test since these bits are continuously involved in the test operations. Being the coverage not complete it cannot be provided in terms of covered fault models. The percentage of Table 3 indicates the amount of fault instances for a given class that have been correctly detected. The test can be improved to reach 100% fault coverage on single-cell faults by modifying the DB used in the directory array test program (Fig. 12) in order to completely fill the cache lines of the data array. This assures that all validity bits assume the values 0 and 1 at least once. This can be easily implemented by reusing the data array test patterns proposed in Fig. 11 where TIS and $\overline{TIS}$ represent the test pattern to load in cache and the incorrect pattern to load into main memory, respectively. A wrong validity bit forces the cache to load the wrong instruction from the main memory, thus allowing to detect the faulty condition looking at the result of the execution.

Table 1
Test Length and Execution Time

March	Original Complexity	Data Array			Directory Array		
		#instructions	execution cycles	test time (at 800Mhz)	#instructions	execution cycles	test time (at 800Mhz)
SOA Mats+	5n	3.644	715.983	0,89s	3.360	128.600	0,16s
SOA C --	10n	3.732	1.030.035	1,28s	3.432	217.572	0,27s
U	13n	3.808	1.232.303	1,54s	3.488	284.323	0,36s
LR	14n	3.824	1.367.770	1,71s	3.492	310.916	0,39s
B	17n	3.856	1.506.249	1,88s	3.532	404.620	0,50s
SS	22n	3.944	1.922.804	2,40s	3.608	413.243	0,52s

Table 2
Test Coverage

March/FFM	SOA MATS+		SOA C --		U		LR		B		SS	
	Data	Dir	Data	Dir	Data	Dir	Data	Dir	Data	Dir	Data	Dir
SF	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2
TF	1/2	1/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2
WDF	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	2/2	2/2
RDF	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2
DRDF	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	2/2	2/2
IRF	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2
CFst	4/8	4/8	8/8	8/8	8/8	8/8	8/8	8/8	6/8	6/8	8/8	8/8
$CFds_{xw}$	3/8	3/8	8/8	8/8	8/8	8/8	8/8	8/8	7/8	7/8	8/8	8/8
$CFds_{xw\bar{x}}$	3/8	3/8	8/8	8/8	8/8	8/8	8/8	8/8	8/8	8/8	8/8	8/8
$CFds_{xwx}$	0/8	0/8	0/8	0/8	0/8	0/8	0/8	0/8	0/8	0/8	8/8	8/8
CFtr	2/8	2/8	8/8	8/8	8/8	8/8	8/8	8/8	4/8	4/8	8/8	8/8
CFwd	0/8	0/8	0/8	0/8	0/8	0/8	0/8	0/8	0/8	0/8	8/8	8/8
CFrd	4/8	4/8	8/8	8/8	8/8	8/8	8/8	8/8	4/8	4/8	8/8	8/8
CFdrd	0/8	0/8	0/8	0/8	0/8	0/8	0/8	0/8	0/8	0/8	8/8	8/8
CFir	4/8	4/8	8/8	8/8	8/8	8/8	8/8	8/8	4/8	4/8	8/8	8/8

Table 3
Validity Bits Test Coverage (%)

March/FFM	SOA MATS+	SOA C --	U	LR	B	SS
SF	98	98	98	98	98	98
TF	93	93	93	93	93	93
WDF	87	87	87	87	87	87
RDF	100	100	100	100	100	100
DRDF	100	100	100	100	100	100
IRF	100	100	100	100	100	100
CFst	78	78	78	78	78	78
$CFds_{xw}$	56	56	56	56	56	56
$CFds_{xw\bar{x}}$	41	41	41	41	41	41
$CFds_{xwx}$	45	45	45	45	45	45
CFtr	51	51	51	51	51	51
CFwd	71	71	71	71	71	71
CFrd	78	78	78	78	78	78
CFdrd	53	53	53	53	53	53
CFir	78	78	78	78	78	78

Finally, we fault simulated the cache controller circuitry during the application of the proposed test programs. Results provided an average test coverage of 81% on stuck-at faults. While this result is not enough for high quality test programs, it represents a starting point on which custom test programs such as the one proposed in [27] can be applied to detect those faults that escaped this phase.

7 Conclusions

This paper addressed the problem of applying march test algorithms to SBST of set-associative instruction cache memories. The main contribution of the presented work is the possibility of applying state-of-the-art test algorithms to embedded cache memories without introducing any hardware or performance overhead. Experimental results obtained by constructing test programs for the LEON3 microprocessor show that it is possible to preserve the fault coverage of the original march tests. Experimental results also consider control blocks of the cache such as validity bits and control circuits, providing reasonable coverage also on these blocks. Additional fields that might be included in a microprocessor cache (e.g., parity bits or error correction codes) have not been explicitly considered in this work. Similarly to validity bits, their coverage should be evaluated considering the specific implementation of the target cache memory and, whenever required, the test program should be improved to cover undetected faults.

References

[1] R. Stacpoole and T. Jamil, "Cache memories," *Potentials, IEEE*, vol. 19, no. 2, pp. 24–29, Apr/May 2000.

[2] L. J. Henessy and A. D. Patterson, *Computer Architecture*, 3rd ed. Morgan Kaufmann Publishers, 2003.

[3] D. Bhandarkar and J. Ding, "Performance characterization of the Pentium Pro processor," in *Third International Symposium on High-Performance Computer Architecture, 1997*, 1-5 Feb 1997, pp. 288–297.

[4] S. Manne, A. Klauser, and D. Grunwald, "Pipeline gating: speculation control for energy reduction," in *Proceedings of the 25th Annual International Symposium on Computer Architecture, ISCA98*, Jun-1 Jul 1998, pp. 132–141.

[5] S. Mukhopadhyay, H. Mahmoodi-Meimand, and K. Roy, "Modeling and estimation of failure probability due to parameter variations in nano-scale srams for yield enhancement," in *Symposium on VLSI Circuits, 2004*, June 2004, pp. 64–67.

[6] S. Hamdioui, Z. Al-Ars, A. van de Goor, and M. Rodgers, "Linked faults in random access memories: concept, fault models, test algorithms, and industrial results," *IEEE J. Technol. Comput. Aided Design*, vol. 23, no. 5, pp. 737–757, May 2004.

[7] D. Gizopoulos, A. Paschalis, and Y. Zorian, *Embedded Processor-Based Self-Test*. Springer press, 2004.

[8] A. Krstic, W.-C. Lai, K.-T. Cheng, L. Chen, and S. Dey, "Embedded software-based self-test for programmable core-based designs," *IEEE Design & Test of Computers*, vol. 19, no. 4, pp. 18–27, 2002.

[9] J. Sosnowski, "Software based self-testing of microprocessors," *Journal of System Architecture*, vol. 52, pp. 257–271, 2006.

[10] A. Apostolakis, D. Gizopoulos, M. Psarakis, and A. Paschalis, "Software-based self-testing of symmetric shared-memory multiprocessors," *IEEE Trans. Comput.*, vol. 58, no. 12, pp. 1682–1694, 2009.

[11] L. Chen, S. Ravi, A. Raghunathan, and S. Dey, "A scalable software-based self-test methodology for programmable processors," in *Design Automation Conference, DAC 2003*, June 2003, pp. 548–553.

[12] N. Kranitis, A. Paschalis, D. Gizopoulos, and G. Xenoulis, "Software-based self-testing of embedded processors," *IEEE Trans. Comput.*, vol. 54, no. 4, pp. 461–475, April 2005.

[13] A. Benso, A. Bosio, P. Prinetto, and A. Savino, "An on-line software-based self-test framework for microprocessor cores," in *Proc. International Conference on Design and Test of Integrated Systems in Nanoscale Technology DTIS 2006*, 2006, pp. 394–399.

[14] A. J. van de Goor, "Using march tests to test srams," *IEEE Des. Test. Comput.*, vol. 10, no. 1, pp. 8–14, Mar. 2004.

[15] A. GAISLER. Leon3 processor. [Online]. Available: http://www.gaisler.com

[16] W. Pei, W.-B. Jone, and Y. Hu, "Fa[illegible] modeling and detection for drowsy sram caches," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 26, no. 6, pp. 1084–1100, 2007.

[17] S. Bhunia, H. Li, and K. Roy, "A high performance iddq testable cache for scaled cmos technologies," in *Proc. 11th Asian Test Symposium (ATS '02)*, 2002, pp. 157–162.

[18] P. J. Tan, T. Le, K.-H. Ng, P. Mantri, and J. Westfall, "Testing of ultrasparc t1 microprocessor and its challenges," in *Proc. IEEE International Test Conference ITC '06*, 2006, pp. 1–10.

[19] A. Agarwal, B. Paul, H. Mahmoodi, A. Datta, and K. Roy, "A process-tolerant cache architecture for improved yield in nanoscale technologies," *IEEE Trans. VLSI Syst.*, vol. 13, no. 1, pp. 27–38, 2005.

[20] S. Hamdioui, Z. Al-Ars, and A. van de Goor, "Testing static and dynamic faults in random access memories," in *Proceedings 20th IEEE VLSI Test Symposium, 2002. (VTS 2002).*, 2002.

[21] R. Raina and R. Molyneaux, "Random self-test method applications on [illegible]werPC™microprocessor caches™microprocessor caches," in *8th Great Lakes Symposium on VLSI.*, 19-21 Feb 1998, pp. 222–229.

[22] T. Verhallen and A. van de Goor, "Functional testing of modern microprocessors," in *[3rd] European Conference on Design Automation, 1992. Proceedings.*, 16-19 Mar 1992, pp. 350–354.

[23] D. Bhavsar and J. Edmondson, "Test strategy of the alpha axp 21164 microprocessor." in *IEEE International Test Conference*, 1994.

[24] J. Sosnowski, "In-system testing of cache memories," in *International Test Conference*, 21-25 Oct 1995, pp. 384–393.

[25] ——, "Improving software based self-testing for cache memories," in *2nd International Design and Test Workshop, 2007*, Dec. 2007, pp. 49–54.

[26] M. Tuna, O. Garcia, and M. Benabdenbi, "Software-based self-test strategies for memory caches of risc processor cores," in *IEEE Latin American Workshop LATW'07*, 2007.

[27] W. Perez H, J. Medina, D. Ravotto, E. Sanchez, and M. Reorda, "Software-based self-test strategy for data cache memories embedded in socs," in *Proc. 11th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems DDECS 2008*, 2008, pp. 1–6.

[28] W. Perez, J. Velasco, D. Ravotto, E. Sanchez, and M. Reorda, "A hybrid approach to the test of cache memory controllers embedded in socs," in *Proc. 14th IEEE International On-Line Testing Symposium IOLTS '08*, 2008, pp. 143–148.

[29] S. Al-Harbi and S. Gupta, "A methodology for transforming memory tests for in-system testing of direct mapped cache tags," in *16th IEEE VLSI Test Symposium*, 26-30 Apr 1998, pp. 394–400.

[30] S. Alpe, S. Di Carlo, P. Prinetto, and A. Savino, "Applying march tests to k-way set-associative cache memories," *European Test, 2008 13th*, pp. 77–83, May 2008.

[31] B. Jacob, S. W. NG, and D. Wang, *Memory Systems: Cache, DREAM, Disk*. Morgan Kaufmann Publishers, 2008.

[32] S. Di Carlo and P. Prinetto, "Models in memory testing," *Models in Hardware Testing*, pp. 157–185, 2010.

[33] A. Benso, A. Bosio, S. Di Carlo, G. Di Natale, and P. Prinetto, "March test generation revealed," *Computers, IEEE Transactions on*, vol. 57, no. 12, pp. 1704 –1713, dec. 2008.

[34] A. van de Goor and I. Tlili, "A systematic method for modifying march tests for bit-oriented memories into tests for word-oriented memories," *IEEE Trans. Comput.*, vol. 52, no. 10, pp. 1320–1331, 2003.

[35] A. van de Goor and Y. Zorian, "Effective march algorithms for testing single-order addressed memories," in *Design Automation, 1993, with the European Event in ASIC Design. Proceedings. [4th] European Conference on*, 22-25 Feb 1993, pp. 499–505.

[36] A. J. van de Goor, *Testing Semiconductor Memories: theory and practice*. John Wiley and Sons, Inc, September 1991.

[37] A. van de Goer and G. Gaydadjiev, "March u: a test for unlinked memory faults," *Circuits, Devices and Systems, IEE Proceedings -*, vol. 144, no. 3, pp. 155–160, Jun 1997.

[38] S. Hamdioui, A. van de Goor, and M. Rodgers, "March ss: a test for all static simple ram faults," in *Proc. IEEE International Workshop on Memory Technology, Design and Testing (MTDT 2002)*, 2002, pp. 95–100.

[39] M. Graphics. Modelsim - advanced simulation and debugging. [Online]. Available: http://model.com/

[40] A. Benso, S. Di Carlo, G. Di Natale, and P. Prinetto, "Specification and design of a new memory fault simulator," in *Proceedings of the 11th Asian Test Symposium, 2002. (ATS '02).* IEEE, 2002, pp. 92–97.

Stefano Di Carlo received the MS degree in computer engineering and the PhD degree in information technologies from the Politecnico di Torino, Torino, Italy. Since 2008, he has been an assistant professor in the Department of Control and Computer Engineering, Politecnico di Torino. His research interests include DFT, BIST and dependability. He is a Golden Core member of the IEEE Computer Society and a member of the IEEE.

Paolo Prinetto received the MS degree in electronic engineering from the Politecnico di Torino, Torino, Italy. He is a full professor of computer engineering in the Department of Control and Computer Engineering, Politecnico di Torino, and a joint professor at the University of Illinois, Chicago. His research interests include testing, test generation, BIST, and dependability. He is a Golden Core member of the IEEE Computer Society. He is a member of the IEEE and a member of the IEEE Computer Society.

Alessandro Savino received the MS degree in computer engineering and the PhD degree in information technologies from the Politecnico di Torino, Torino, Italy. Since 2009, he has been a post-doc at the Department of Control and Computer Engineering, Politecnico di Torino. His main research topics are microprocessor test, and software based self test.