A Functional Approach for Testing the Reorder Buffer Memory

A Functional Approach for Testing the Reorder Buffer Memory

S. Di Carlo, M. Gaudesi, E. Sanchez, M. Sonza Reorda

Dipartimento di Automatica e Informatica, Politecnico di Torino, Torino, Italy

{stefano.dicarlo, marco.gaudesi, ernesto.sanchez, matteo.sonzareorda}@polito.it

Abstract — Superscalar processors have the ability to execute instructions out-of-order to better exploit the internal hardware and to maximize the performance. To maintain the in-order instruction commitment and to guarantee the correctness of the final results (as well as precise exception management), the Reorder Buffer (ROB) may be used. From the architectural point of view, the ROB is a memory array of several thousands of bits that must be tested against hardware faults to ensure a correct behavior of the processor. Since it is deeply embedded within the microprocessor circuitry, the most straightforward approach to test the ROB is through Built-In Self-Test solutions, which are typically adopted by manufacturers for end-of-production test. However, these solutions may not always be used for the test during the operational phase (in-field test) which aims at detecting possible hardware faults arising when the electronic systems works in its target environment. In fact, these solutions require the usage of test infrastructures that may not be accessible and/or documented, or simply not usable during the operational phase. This paper proposes an alternative solution, based on a functional approach, in which the test is performed by forcing the processor to execute a specially written test program, and checking the resulting behavior of the processor. This approach can be adopted for in-field test, e.g., at the power-on, power-off, or during the time slots unused by the system application. The method has been validated resorting to both an architectural and a memory fault simulator.

Keywords – microprocessor testing, software-based self-test, embedded memory test, in-field test

1. Introduction

Final users of embedded processors cores, during the last years, are increasingly requiring new strategies that better suit with in-field testing requirements for these devices, especially if they are used in safety- or mission-critical applications, e.g., in the automotive, railways and aerospace domains [1, 2]. In these scenarios, new standards such as the ISO 26262 for automotive, and DO-254 for avionics, are providing the guidelines to create testing procedures for in-field testing of processor cores, specify the fault coverage figures that must be attained with respect to permanent faults. In this way, any hardware fault that possibly occurred in the system is detected as early as possible, and its negative consequences in terms of system misbehaviors can be prevented or mitigated.

Contemporary, embedded processors for such applications have been required to increase functionalities and performance; then, in nowadays embedded systems, most of the embedded system devises are including out-of-order superscalar microarchitectures able to follow the increasing request for increasing performance and functionalities.

Design for Testability (DfT) or Built-In Self-Test (BIST) techniques may not be suitable for testing embedded processor cores during the normal operation life, since these techniques usually require to modify the original design, and in order to be used, these techniques require for the testing process the support of an external equipment, e.g., an ATE; moreover, using DfT facilities also require to have internal details about the device that manufactures may not share, due to IP protection, with final users. Finally, DfT solutions are also difficult to exploit during the normal processor life, because these techniques seriously affect the actual processor status.

On the other hand, functional test approaches may represent an interesting solution when tackling *in-field* testing. A functional test applies stimuli to the system under test only resorting to its functional inputs, and observes the system's behavior only resorting to its functional outputs. More specifically, the functional test may be implemented resorting to a suitable test program run by the processor under test and expected to produce correct results, e.g., in terms of final content of an output memory area (Software-Based Self-Test or SBST [3]).

Functional test methods may be conveniently used instead of DfT or BIST techniques in some scenarios. First, they do not require any modification of the processor design. This is an important characteristic when third-party processor cores are integrated into a SoC. Secondly, since functional tests exercise the unit under test exactly in the same conditions used during normal behavior, they easily enable at-speed testing while avoiding over-testing. Finally, testing a processor-based system during its operational phase (in-field test) using DfT techniques may be complex. As a conclusion, the functional approach may be particularly attracting for system designers when they want to develop in-field test solutions for their systems.

To the best of our knowledge, it is possible to state that the main drawback of functional test approaches is the difficulty in devising test programs matching existing constraints in terms of duration, size, and fault coverage against a clearly defined fault model. Starting from the first approach to develop functional tests for microprocessors proposed by Thatte et al. in 1980 [4], a rich literature has been produced. Publications propose methods for effectively writing programs for the test of whole microprocessors [5], special modules (e.g., branch prediction units [6], or cache memories [7]), and peripherals components [8].

In this paper, we focus on the test program generation for in-field testing of the memory elements inside the Reorder Buffer (ROB) that may exist within superscalar processors. The ROB is a deeply embedded module that plays an important role within processors supporting out-of-order speculative execution. In particular, the ROB guarantees the in-order instruction commitment (i.e., the phase in which the result produced by the instruction is written to its destination), and is devoted to manage exceptions in a precise manner [18]. Since the ROB function is implemented in rather different ways in different processor architectures, for the purpose of this paper we selected the solution implemented by the widely adopted architectural simulator called SimpleScalar [15]. The ROB is based on a memory block organized in entries. Each entry corresponds to an instruction

waiting for being committed. Since each entry is composed of different fields, storing several information about the instruction, and the size of the ROB is usually in the order of some tens of entries, the total ROB memory size is usually in the order of some thousands of bits.

In this paper, we propose a test algorithm which is particularly suited for the in-field test of the ROB internal memory. Following the algorithm one can easily write a test program to be run during the operational phase, either periodically (e.g., during the time slots left idle by the system application) or during specific situations (e.g., at power-on or power-off). By observing the processor behavior during the test program execution and by looking at the produced results it is possible to detect possible hardware faults affecting the ROB memory.

A preliminary work targeting only one field of the ROB was presented in [9], which only addressed one of the several fields composing each ROB entry (i.e., the *value* field). In this paper we extend that preliminary work to test the other fields in the ROB memory, specifically focusing on the test of the *address* field, which presents some not trivial access problems. Moreover, this paper presents some more comprehensive results in terms of size and duration of the resulting test program. Remarkably, we also present here how to extend the proposed algorithm to the *identifier* field in the ROB, that (to the best of our knowledge) is always available in these hardware structures. Other ROB fields are not considered here, since these are highly dependent on the actual implementation of the superscalar processor, and are not always present in the ROB.

Depending on the specific implementation, the ROB may include several different fields; for the purpose of this paper we mainly focus on the most important ones of them, which are always present: *value, address,* and *identifier*. The overall idea is to map the set of fault primitives required to sensitize and detect typical memory faulty behaviors [10] to a proper sequence of processor instructions. In this way., we can transform a generic March algorithm into the corresponding test program, which will perform on the ROB memory the same sequence of read/write operations of the original March algorithm, thus achieving its same fault coverage. We will show that the complexity of the resulting test algorithm grows quadratically with the number of ROB entries. Nevertheless, since the ROB size is usually limited, the test program is still limited in terms of size and duration. A major characteristic of the resulting test program lies in the fact that it does not need to be executed as a whole. In fact, it can be split in fragments that can be executed independently at different times (i.e., with reduced cost in terms of stopping and resuming the test, as well as checking its results), thus better matching the strict time constraints imposed when testing a system in its operational environment.

Considering end-of-production testing, the approach proposed in this paper is clearly not so attracting to be performed by semiconductor companies, which can resort to more effective solutions based on DfT. However, the approach may be useful for embedded systems designers, which use processors or SoC devices from third companies. In this case the engineer in charge of developing the in-field test often does not have detailed information on the internal architecture of the processor, but only knows its Instruction Set Architecture. The approach could also be of interest for the in-field test of SoC devices, since this scenario shares several characteristics with the previous one: SoC designers often use processor cores from third parties that can hardly be modified, and whose in-field test may be performed resorting to the functional approach.

The proposed algorithm has been validated resorting to both an architectural and a memory fault simulator. Results show that the proposed algorithm allows to detect the most relevant faults (with the exception of a few faults that are functionally untestable). We also report results allowing to evaluate the cost of the approach in terms of test program size and duration.

The paper is organized as follows: the next Section reports some background about the ROB architecture and behavior. Section III describes the functional approach we propose for generating suitable programs for its test; Section IV describes the experiments we performed and report some data we gathered. The last Section draws some conclusions.

2. Background

In order to better exploit the Instruction Level Parallelism existing in almost every program, superscalar processors support out-of-order execution. This approach, in combination with dynamic scheduling, enables the execution of each instruction as soon as the required functional unit is free and the values of the input operands are available. However, this mechanism can affect the correctness of the computation (e.g., due to Write After Write hazards and imprecise exception handling).

The ROB, or other structures playing a similar role, is mainly intended to guarantee that, despite the out-of-order execution, the completion of each instruction (in particular the phase in which results are written in the target destination) is performed in-order. In this way, Write After Write hazards cannot arise and precise exception handling can be easily implemented. The ROB also plays a key role in speculative execution [11].

A ROB is a memory organized in *entries*, each composed of several fields including: (i) an *identifier* of the instruction, (ii) the *value* produced by the instruction, and (iii) the memory address or register target, where the actual instruction value must be written when the instruction is committed. The ROB is accessed during different phases of the execution of an instruction:

- During the *issue* phase, the processor assigns the instruction to the next free ROB entry. If no entry is available a stall arises. ROB entries are assigned to instructions following the instruction issue order. The ROB is therefore organized as a First-In First-Out (FIFO) buffer, whose key is the order of each entry (i.e., instruction) in the code. Concurrently, during the issue phase, the *identifier* field is written using the type of instruction allocated in the ROB entry. This information is usually related to the instruction type, e.g., branch instructions, ALU instructions, etc.
- When an instruction completes its execution, the produced result is written in the *value* field of the associated ROB entry together with all information items required to identify the target location, that are written in the *address* field of the same entry. In the case of *load* / *store* instructions the *address* field is updated with the calculated effective address.
- At each clock cycle, the circuitry associated to the ROB checks whether the oldest instructions in the ROB (according to the issue order) have completed their execution. If yes, the instructions are *committed*, executing some specific actions according to the instruction type, that is recognized by reading the instruction *identifier*. Summarizing, the produced values are written to the assigned target locations, registers or memory locations. To do this, the value

contained within the *address* field is read and then used to identify the target location for the result (corresponding to a memory location).

- When a conditional branch instruction is executed, the result is compared with the branch prediction. If a mis-prediction occurs, all instructions following the branch and already allocated in the ROB are aborted and removed from the ROB.
- When the input operand of an instruction is produced by another instruction that has been executed but not yet committed, the corresponding value is stored in the ROB, only. To avoid a stall the processor reads this value from the ROB, and not from the Register File. It then forwards it to the functional unit, which can thus start its execution.

Summarizing, the processor accesses the different fields that belong the ROB during the following steps:

- The *value* field:
  - In the issue phase, by allocating entries to instructions according to the FIFO mechanism. Moreover, issued instructions may *read* input data from the ROB if the data were produced by not yet committed instructions;
  - At the end of the execution phase of a generic instruction X, to *write* output data into the value field (and others) of the ROB entry associated to X;
  - In the commit phase, to *read* the value field and *write* its value in the instruction target location.
- The *address* field:
  - At the end of the execution phase of a Load or Store instruction X, to write the effective address calculated by X into the address field of the ROB entry associated to X;
  - In the commit phase, to read the address field and send this data to the Load/Store unit.
- The *identifier* field:
  - During the issue phase: the identifier field of the ROB entry is written with the information regarding the instruction type;
  - During the commit phase: the identifier field is read in order to complete the instruction according to its type.

The other fields, which are highly dependent on the actual ROB implementation, are not detailed here.

It is worth mentioning here that the ROB is typically used in processors supporting the issue, execution and completion of multiple instructions at the same clock cycle. For this reason a ROB is typically organized as a multiple-port memory, to which multiple instructions can access concurrently from different stages.

3. Proposed Approach

According to the assumptions in the previous Section, the ROB can be treated as a memory composed by several entries, each including different fields. In particular, in this work we will mainly focus on two specific fields, namely the *value* and *address* fields, and we will propose a method allowing to write a test program that, when run by the processor, can detect any hardware

faults affecting the ROB memory bits storing them. Moreover, a brief dissertation is also performed in order to detail how to extend the proposed approach to the *identifier* field in the Reorder Buffer. Normally, within the ROB are present other fields (in particular, some flag fields), whose number and role often change depending on the target processor, and that can be tested by extending the approach in a rather straightforward manner.

More in details, our approach is mainly based on providing rules that allow to transform whichever March algorithm into a test program executing the same sequence of read/write operations on the targeted bits of the ROB memory. In this way the fault coverage achieved by the test program is basically the same that would have been provided by the March algorithm, if we could apply it by directly acting on the ROB memory inputs and observe its outputs.

Let us denote by n the number of entries of the ROB and by m the number of bits composing the *target* (corresponding either to value, or address, or whichever else) field. Let us assume that the memory arrays storing different fields are physically independent, and that the whole ROB can be modeled as an $n \times m$ memory array.

The proposed test algorithm implements a deterministic sequence of read/write operations on the ROB entries, in order to implement the March test depicted within the next section. Firstly, we take into account the case of the *value* field: a writing operation within this field happens when the instruction associated to the entry completes its execution: in this stage, the produced value is written in the corresponding ROB entry. The execution order of the write operations closely follows the one in which instructions complete their execution. The value written in each ROB entry is read when the corresponding instruction is committed. The instruction result is written to the target destination (either a register or a memory location) and the instruction is removed from the ROB, thus freeing the corresponding entry. Since the ROB implements a FIFO strategy, the order of read operations strictly follows the order instructions are issued and assigned to the ROB.

The *value* field of the ROB entry associated to an instruction Y whose execution has been completed but still not committed is also read when an instruction X requires an operand produced by Y.

In the case of the *address* field, instead, a write operation arises when the Load or Store instruction associated to the ROB entry computes the effective address of the memory location from/to which data are read/written. This computation is performed when the instruction is written in the ROB if the address is already known at this time; otherwise, its computation is delayed in order to wait for the execution of the instruction(s) responsible for the generation of the address.

The *address* field of the ROB entry is read when the instruction is executed and it performs the corresponding read or write operation through the Load/Store Unit.

Finally, to test the *identifier* field of each ROB's entry, a write operation arises when the issue phase of an instruction is performed; during this operation, a value able to univocally identify each instruction presents in the ISA of the processor is written within the field. The *identifier* field is read when the instruction execution is ended and, with respect to the execution order, it stands for committing in memory the calculated value.

In the following we will introduce the algorithms related to the detection of detect single-cell and double-cell (i.e., coupling) faults in a memory, and then we will describe how to write test programs, for the previously introduced fields, able to reproduce these test conditions on the ROB. For sake of simplicity, in this section we will assume that the ROB memory is only accessed by one instruction per stage per clock cycle. However, this assumption can be removed without impacting the effectiveness of the proposed algorithm.

3.1. Single- and double-cell fault test requirements

Let us start by considering faults affecting a single *m*-bit word *W* of the memory array under test, and denote by A a *m*-bit test pattern for W, and with $\bar{A}$ the corresponding complemented pattern. From the literature we can easily derive the operations (denoted as *Fault Primitives*, or FPs[1]) required to test the different faults affecting single cells in the memory [8]. They are summarized in Table 1.

Table 1 – Single-cell Fault Primitives

Fault	***FP***		***Fault Model***
SF	(1) $<\bar{A}/A/->$	(2) $<A/\bar{A}/->$	State fault
TF	(1) $<\bar{A}w_A/\bar{A}/->$	(2) $<Aw_{\bar{A}}/A/->$	Transition fault
WDF	(1) $<\bar{A}w_{\bar{A}}/A/->$	(2) $<Aw_A/\bar{A}/->$	Write destructive fault
RDF	(1) $<\bar{A}r_{\bar{A}}/A/A>$	(2) $<Ar_A/\bar{A}/\bar{A}>$	Read destructive Fault
IRF	(1) $<\bar{A}r_{\bar{A}}/\bar{A}/A>$	(2) $<Ar_A/A/\bar{A}>$	Incorrect read-fault
DRDF	(1) $<\bar{A}r_{\bar{A}}/A/\bar{A}>$	(2) $<Ar_A/\bar{A}/A>$	Deceptive RDF

Secondly, we can address faults affecting pairs of memory cells (denoted as *aggressor* and *victim*, respectively) and report the corresponding FPs (see Table 2). In this case we denote by A and V the two *m*-bit test patterns for the aggressor entry and the victim entries of the ROB, respectively, and with $\bar{A}$ and $\bar{V}$ the corresponding complemented patterns. Looking at Table 2, double-cell faults (usually denoted as *coupling faults*) can be grouped in two categories based on the type of sensitizing operation:

1. **Group 1**: faults that are sensitized by an operation/state on the aggressor cell and a state on the victim cell (CFds, CFst)
2. **Group 2**: faults that are sensitized by a state of the aggressor cell and an operation on the victim cell (CFtr, CFwd, CFrd, CFir, CFdrd).

Table 2 – Double-cell Fault Primitives

Fault	***FP***		***Fault Model***
CFst	(1) $<\bar{A};\bar{V}/V/->$ (3) $<A;V/\bar{V}/->$	(2) $<\bar{A};V/\bar{V}/->$ (4) $<A;\bar{V}/V/->$	State coupling fault
CFds	(1) $<xw_y;\bar{V}/V/->$	(2) $<xw_y;V/\bar{V}/->$	Disturb

[1] FP=<S/F/R> where S is the sequence of operations required to sensitize the fault, F is the observed faulty behavior that deviates from the correct memory behavior and R, in case of a read operation, is the read result.

Fault	*FP*		*Fault Model*
	(3) $< xr_x;\overline{V} / V / - >$	(4) $< xw_x;V / \overline{V} / - >$	coupling fault
CFtr	(1) $< \overline{A};\overline{V}w_V / \overline{V} / - >$	(2) $< \overline{A};Vw_{\overline{V}} / V / - >$	Transition coupling fault
	(3) $< A;\overline{V}w_V / \overline{V} / - >$	(4) $< A;Vw_{\overline{V}} / V / - >$	
CFwd	(1) $< \overline{A};\overline{V}w_{\overline{V}} / V / - >$	(2) $< \overline{A};Vw_V / \overline{V} / - >$	Write destructive coupling fault
	(3) $< A;\overline{V}w_{\overline{V}} / V / - >$	(4) $< A;Vw_V / \overline{V} / - >$	
CFrd	(1) $< \overline{A};\overline{V}r_{\overline{V}} / V / V >$	(2) $< \overline{A};Vr_V / \overline{V} / \overline{V} >$	Read destructive coupling fault
	(3) $< A;\overline{V}r_{\overline{V}} / V / V >$	(4) $< A;Vr_V / \overline{V} / \overline{V} >$	
CFir	(1) $< \overline{A};\overline{V}r_{\overline{V}} / \overline{V} / V >$	(2) $< \overline{A};Vr_V / V / \overline{V} >$	Incorrect read coupling fault
	(3) $< A;\overline{V}r_{\overline{V}} / \overline{V} / V >$	(4) $< A;Vr_V / V / \overline{V} >$	
CFdrd	(1) $< \overline{A};\overline{V}r_{\overline{V}} / V / \overline{V} >$	(2) $< \overline{A};Vr_V / \overline{V} / V >$	Deceptive read destructive CF
	(3) $< A;\overline{V}r_{\overline{V}} / V / \overline{V} >$	(4) $< A;Vr_V / \overline{V} / V >$	

The conditions to test faults of group 1 are: (1) initialize the victim cells to a given value, (2) sensitize the fault by performing the three possible sensitizing operations (a non-transition write, a transition write and a read) on the aggressor cell, (3) read out the content of the victim cells to check if some of them changed their status.

The conditions to test faults of group 2 are: (1) initialize the victim cells to a given value, (2) initialize the aggressor cell to a given value; (3) for each victim cell sensitize the fault by performing the three possible sensitizing operations (a non-transition write, a transition write and a read) followed by (4) a read operation to detect the fault.

3.2. Test algorithm

This sub-section introduces the proposed ROB test algorithm. The algorithm consists of three passes: the former is for testing the *value* field, then the *address* field, and finally the *identifier* field. Let us focus on the *value* field first.

Considering an *n* entries ROB, the test conditions defined by the considered FPs can be matched by a test program implementing the following sequence of operations, denoted as *basic building block (BBB)*.

1. Write $V / \overline{V}$ in all victim entries and then $A / \overline{A}$ in the aggressor entry
2. Write $V / \overline{V}$ in all victim entries and then $A / \overline{A}$ in the aggressor entry
3. Read the content of all entries starting from the aggressor to detect faults of group 1
4. Write $V / \overline{V}$ in all victim entries and then $A / \overline{A}$ in the aggressor entry

5. Read all victim entries two times to detect all faults of group 2.

To prove that the above BBB is able to detect the FPs introduced in the previous sub-section we focus on the double-cell faults reported in Table 2. Detection conditions for single-cell faults are in general simpler and included in those required for double-cell faults [10]. Let us consider faults of group 1 (i.e., CFds, CFst). To sensitize these faults we need first to initialize the ROB entries. This is performed in step 1 of the BBB by writing $V / \bar{V}$ in all victim entries and then $A / \bar{A}$ in the aggressor entry. Step 2 of the BBB is the first step in which faults are sensitized. First, all victim entries are again initialized with $V / \bar{V}$. These redundant write operations are required since the ROB applies a FIFO strategy. Therefore, to write a new value in the aggressor entry that was the last written during step 1 we need first to write all victim entries. Secondly, the aggressor cell is written with $A / \bar{A}$ to sensitize the faults. The sensitized FPs depends on the actual patterns written in the entries during steps 1 and 2. If for instance in both steps the victim and aggressor entries are respectively written with patterns A and V, FP3 of CFst and FP2 with non-transition write of CFst are sensitized. Step 3 of the BBB starts reading the aggressor entry. This represents the last sensitizing operation for group 1 faults and is required to sensitize FP3 of CFds. At this point all possible sensitizing operations have been executed. By reading out all victim entries it is possible to detect if any fault occurred.

The remaining two steps of the BBB are required to address faults of group 2 (i.e., CFtr, CFwd, CFrd, CFir, CFdrd). All these faults are sensitized by a state of the aggressor entry and an operation on the victim entry. When reaching step 4 the memory is already initialized. By performing a write operation on all victim entries FPs belonging to CFtr and CFwd models can be sensitized. Again, the sensitized FP depends on the applied test patterns. If the aggressor entry was initialized with A, the victim entries were initialized with V and a $w_{\bar{V}}$ operation is performed on each victim, FP4 of CFtr ($< A; V w_{\bar{V}} / V / - >$) is sensitized. Step 4 terminates with a write operation on the aggressor entry. Once more, this operation is required to cope with the FIFO policy of the ROB.

In the last step of the BBB (step 5) all victim cells are read 2 times. With the first read operation faults sensitized during step 4 can be detected. Moreover, this operation sensitizes CFrd, CFir and CFdrd FPs and detects CFrd, CFir FPs. The second read operation is able to detect CFdrd FPs. In fact, in this case the fault is sensitized by the first read but observed only when the entry is read again. Actually, this last step cannot be performed on the address field; this is due to the fact that it is not possible to read two times the address field from the same ROB entry.

The BBB must be executed 6 times changing the combination of the test patterns in the victim and aggressor cells during steps 1, 2 and 4 in order to address all selected FPs. Finally, everything must be executed *n* times considering every element of the ROB as the aggressor cell.

The main characteristic of this test algorithm is that write instructions always follow the same order: first all victim cells are written, followed by the aggressor cell. This behavior can be reproduced on the ROB by forcing the processor to execute a code fragment composed of:

- an instruction named I1 characterized by a long execution time (e.g., DIV) and a result equal to $A / \bar{A}$;

- *n-1* instructions (named I2 to In) characterized by a short execution time (e.g., ADD), a result equal to $V / \bar{V}$, and one of the input operands corresponding to the output operand of the previous instruction (except for the first).

For the purpose of analyzing the behavior of the ROB during the execution of this fragment, we can identify the following phases:

- *Issue phase*: all instructions of the fragment are issued. At the end of this phase the ROB includes one entry devoted to I1 (corresponding to the aggressor entry), and all other entries devoted to instructions I2 to In (corresponding to the victim entries).
- *Execute phase*: the short instructions I2 to In finish their execution before I1 finishes. This means that during this phase I2 to In rapidly finish their execution one after the other. As soon as one of them finishes its execution, it writes the produced result in the ROB. Immediately after, the following instruction reads this value, enters execution, and repeats the same operation. However, instructions I2 to In cannot immediately commit, since they wait for the commit of I1. During this phase each ROB cell (apart from the one associated to I1) undergoes a write, followed by a read operation due to the data dependency between consecutive instructions; with the exception of the ROB cell corresponding to In, where the read is not performed. When at last the execution of I1 finishes, I1 writes its result to the associated ROB slot.
- *Commit phase*: when finally I1 completes its execution, it commits. The value written in the corresponding ROB entry is read and written in the target destination, thus executing a new read operation. All other instructions (I2 to In) can now also commit. The values written in the corresponding ROB slots are thus read and written in the target destinations (i.e., *n-1* registers).

The above code fragment can be exploited to force the processor to perform on the ROB the operations mandated by the Basic Building Block.

The resulting test program can be summarized as follows:

1. execute I1 to In to initialize the ROB (step 1 of the Basic Building Block);
2. execute I1 to In to sensitize CFst and CFds that are sensitized by operations on the aggressor cell (step 2 and 3 of the Basic Building Block);
3. execute *n* store instructions writing the *n* target registers into memory and thus making the results of the previous steps observable. According to the SimpleScalar model, in the execution phase a store instruction writes into its ROB entry the value to be moved to memory; thus, the ROB entry value is not changed during the commit phase;
4. execute I1 to In to sensitize CFtr, CFwd, CFrd, CFir, CFdrd that are sensitized by operations on the victim cells (step 4 and 5 of the Basic Building Block);
5. execute *n* store instructions, moving the values of the *n* target registers into memory;
6. repeat steps 0 to 4 six times with different values A, $\bar{A}$, $V / \bar{V}$ for instructions I1 to In
7. repeat steps 1 to 6 *n* times by allocating a different slot to the “long” instruction I1 (which can be achieved by just executing a “dummy” instruction before executing again step 0). In this way we can test faults activated by each possible aggressor cell.

The algorithm is completed by checking whether all values written into memory during the algorithm execution comply with the expected ones.

A generic step of this algorithm oriented to test a 6-entries ROB, is depicted in Table 3. The ROB field targeted here is the *value* field. It performs two write cycles to excite faults (in this example, *victims* write zeros and *aggressor* writes a one), and one read cycle after which values are checked to identify faults:

Table 3 – Read / Write operations order for testing the *value* field

#	instr														
1	AGGR				w(1)	r()						w(1)	r()		
2	VICT	w(0)				r()			w(0)				r()		
3	VICT		w(0)				r()			w(0)				r()	
4	VICT		w(0)				r()			w(0)				r()	
5	VICT			w(0)				r()			w(0)				r()
6	VICT			w(0)				r()			w(0)				r()
step		1	2	3	4	5	6	7	8	9	10	11	12	12	14

The *aggressor* (shortened for convenience with the acronym AGGR) instruction is a multiplication instruction, which is the first to be allocated in the ROB. While this first instruction is executed, the *victim* (shortened with the acronym VICT) instructions (e.g., ADD instructions) are allocated to the other entries of the ROB. These preliminary steps are not depicted in Table 3, since the main goal of this table is to show how the algorithm performs the different steps defined by the corresponding March test. It is important to notice that Table 3 does not show an additional phase devoted to read the results of the first set of instructions in the middle of steps 3 and 4. This reading phase is not depicted here, since in any case these reading operations do not affect the March test development.

The execution time, in terms of clock cycles, of the ADD instructions is usually much shorter of that of the multiplication instructions, and since these instructions use different functional units, victims can write the final result of the ADD instructions inside the *value* field of their entry in the ROB (step 1, 2 and 3 in Table 3), while the multiplication is being waiting to be committed. As soon as the aggressor instruction (i.e., the multiplication one) finishes the execution phase (step 4), the write cycle is repeated. At the end of the second cycle, finally, all the prevised errors that can be detected whit this particular configuration entry/values should be excited and then the algorithm passes to the read phase. At step 11, the result of second aggressor is written in the *value* field of its ROB entry and then it is immediately read to enter the commit phase. After reading the value from the *value* field of the aggressor entry, all the other values from each *value* field of the victim entries are read, in order to commit themselves.

The presented algorithm, used to test the *value* field, corresponds to the execution of $6 \times 6 \times n^2$ instructions. Hence, the total complexity of the proposed algorithm (in terms of number of instructions) is $O(n^2)$. Given the fact that the size n of the ROB is limited (typically in the order of some tens of entries) this complexity still leads to relatively short and fast test programs.

The proposed algorithm still does not detect coupling faults between bits in the same ROB entry. Following [12], to cover also these faults we can simply add to the algorithm a few more steps:

- in the first step *n* instructions are executed, writing a result value corresponding to a given pattern X to the ROB, and then reading and moving it to a register;
- in the second step the target register values are transferred to observable memory locations resorting to *n* store instructions;
- the two steps are repeated substituting X with its complement pattern $\overline{X}$;
- these three steps are repeated $1+\log_2 m$ times, being *m* the size of the *value* field, each time using a different data background pattern. At the first iteration $X = 00 \rightleftharpoons 00$ and $\overline{X} = 11 \rightleftharpoons 11$; at the second iteration $X = 00 \rightleftharpoons 11$ (i.e., a word composed of *m*/2 0 bits and *m*/2 1 bits) and $\overline{X} = 11 \rightleftharpoons 00$ (i.e., the opposite of X); at the last iteration $X = 10 \rightleftharpoons 10$ (i.e., a word composed of m alternated 0 and 1 bits) and $\overline{X} = 01 \rightleftharpoons 01$ (i.e., the opposite of *X*).

It is worth mentioning here that the ROB is typically used in processors supporting the issue, execution and completion of multiple instructions at the same clock cycle. For this reason a ROB is typically organized as a multiple port memory, to which multiple instructions can access concurrently from different stages. Multiple port memories introduce a set of additional faulty behaviors related to the presence of more than one port to those listed in Table 1 and Table 2. Nevertheless, several publications [13, 14] proved that March-like test sequences like the one proposed in this paper, designed to test single port memories, can be easily adapted to cover multi-port specific fault models by properly selecting the port on which operations are performed. Therefore, extending the proposed test method also to the multi-port scenario does not represent a significant issue.

Considering again an *n* entries ROB, the test procedure for the *address* field is slightly different than the previous one. In fact, in this case the test program must implement the following sequence of operations:

1. Write $V / \overline{V}$ in all victim entries and then $A / \overline{A}$ in the aggressor entry;
2. Write $V / \overline{V}$ in all victim entries and then $A / \overline{A}$ in the aggressor entry;
3. Read the content of all entries starting from the aggressor to detect faults of group 1;
4. Write $V / \overline{V}$ in all victim entries and then $A / \overline{A}$ in the aggressor entry;
5. Read all victim entries to detect some faults of group 2.

To test the *address* field, it is important to initialize all victim cells with a defined value and then write the aggressor value in the right entry of the ROB. The proposed algorithm follows the structure of the one used for testing the *value* field, but with some important differences. The *address* field is only accessed during the execution of memory read and write operations, corresponding to *load* and *store* instructions, respectively.

In the second step of this algorithm, the reading phase is executed only once. This can be seen as an intrinsic limitation of the architecture/instruction set, because it is impossible to access and read twice the same value from an address field within the ROB.

In this work, we assume that the processor is able to access the whole addressable space of memory, thus allowing to read and write any combination of bits into the address field of the each ROB entry. In order to access them in the wished order, we need a particular instruction able to block the

commit phase of the aggressor instruction until all victim values are written within the *address* field of all the ROB entries. This goal is achieved using an instruction that requires a long execution time and that needs a sufficiently high number of processor's clock cycles to compute the result. In this implementation of the algorithm a *multiplication* instruction is used for this purpose. The multiplication instruction requires a long computational time; the algorithm uses the result of the multiplication instruction to compute the address of the memory location in which the *store* instruction (acting as aggressor) writes. In this way the algorithm stops the store instruction (and the following ones) inside the ROB, due to the need to respect the commit order of all instructions available within the ROB.

The method used here is similar to the one used in the previous section for testing the *value* field of the reorder buffer. As mentioned in the initial part of this section, the difference is the usage of *load* or *store* instructions to excite the right memory cell, related to the *address* field of each entry of the ROB.

The algorithm to test the *address* field can be divided in two phases:

1. In the first phase the algorithm performs the functional test of the *address* fields of *n*-2 ROB's entries: in particular, this phase initializes and tests *n*-2 victim cells using a cell as aggressor to excite faults and another cell to block the procedure and wait that all victim cells are initialized. This algorithm's phase is described in Table 4: while the multiplication instruction is computing the address in memory in which the aggressor store instruction writes the proper value, the aggressor and all the victims are fetched inside the ROB. The aggressor waits the result from multiplication, while the victim store instructions calculate and write addresses within the address field (step 1 and 2 in Table 4) and wait for the commit phase until the multiplication and the aggressor instruction do their own commit.

   Once the aggressor's address is calculated (step 3), the corresponding entry receives the value inside the address field (step 4) and then uses this value to perform the commit in memory (step 5). Finally, read operations are executed on victim entries (step 6 and 7).

Table 4 - Read / Write operations order for testing the *address* field (phase I)

#	instr	Read/Write operations order						
1	MULT			w(1)	r(1)			
2	AGGR				w(1)	r()		
3	VICT	w(0)					r()	
4	VICT	w(0)					r()	
5	VICT		w(0)					r()
6	VICT		w(0)					r()
Step		1	2	3	4	5	6	7

2. The second phase is due to an intrinsic issue within the algorithm targeting the *address* field. Since with the algorithm described above for testing the address field an entry of the ROB is always present that cannot be designated as victim (the one with the multiplication instruction), it is mandatory to perform another cycle in order to test it. Table 5 shows the proposed

sequence of read and write operations, using an example ROB with 6 entries. The sequence includes

- a multiplication instruction
- a series of dummy instructions (chosen among those faster to perform, to prevent further delays) to avoid entries already tested
- a store instruction affecting the victim entry
- a store instruction affecting the aggressor entry, which waits for the result of the multiplication as address value.

During this phase, the victim entry must correspond to the entry on which the multiplication instruction in the previous cycle acts, and the aggressor instruction has to be within the same entry used in the previous cycle. Actually, reversing the order between the aggressor and the victim (with respect to the previous step) introduces a modification inside the algorithm. The main difference is that in this step the value of the victim address field is read before or simultaneously to the aggressor one.

Table 5 - Read / Write operations order for testing the *address* field (phase II)

#	instr	Read/Write operations order							
5	VICT			w(0)					r()
6	AGGR					w(1)			r()
1	MULT				w(1)		r()		
2	NOP	*					*		
3	NOP		*					*	
4	NOP		*					*	
step		1	2	3	4	5	6	7	8

In order to test the *identifier* field in the Reorder Buffer, it is necessary to recall that this field is only written during the instruction issue phase, and read during the instruction commit. Thus, no other reading or writing accesses are made to this field during other stages of the instruction cycle. Moreover, these ROB fields are always read and written in order.

With the aim of creating the *basic building block* (BBB) for this ROB field, it is again important to use different latency instructions. However, even using different execution latencies, the access times are not interchangeable; thus, we need to modify our BBB as follows:

1. Write $V / \overline{V}$ in all victim entries and then $A / \overline{A}$ in the aggressor entry
2. Write $V / \overline{V}$ in all victim entries and then $A / \overline{A}$ in the aggressor entry
3. Read the content of all entries, letting the aggressor at the end (detecting faults of group 1)
4. Write $V / \overline{V}$ in all victim entries and then $A / \overline{A}$ in the aggressor entry
5. Read all victim entries (detecting most of the faults of group 2).

As a consequence,write accesses always follow the same order: first all victim cells are written, followed by the aggressor cell. However, the aggressor instruction is not the one having a long execution time but the last issued instruction. This behavior can be reproduced on the ROB by forcing the processor to execute a code fragment composed of:

- instruction I0, characterized by a long execution time (e.g., DIV). This instruction is devoted to freeze the ROB until the aggressor instruction acts;
- *n-2* instructions (named I1 to In-2) that are in this case the victim instructions, characterized by a short execution time (e.g., ADD), and an opcode able to generate in the ROB *identifier* a value equal to $A / \bar{A}$;
- instruction In-1, which is the aggressor instruction characterized by a short execution time (e.g., SUB), and an opcode able to generate in the ROB *identifier* a value equal to $V / \bar{V}$.

It is important to highlight here that one of the main issues when implementing this test programs is the identification of proper instruction opcodes that enable to implement a pattern and a complemented pattern. Nevertheless, as also discussed in [7], modern microprocessors tend to provide enough alternatives within their instruction set that usually enable to fulfill this requirement.

Our considerations here are the following: the first long delay instruction aims at guaranteeing that only at the end of the execution of this instruction all the reading accesses are made. Thus, the actual aggressor instruction is the last one (In-1), since all the other instructions are written before it. Thus, once all identifier fields (I1 – In-2) are written in the ROB, the first instruction (the long delay one) is read, and then, all the others, ending this step with the aggressor one.

It is necessary to carefully select appropriate instructions able to produce the values $A / \bar{A}$, and $V / \bar{V}$ in the ROB identifiers as described in [7].

Considering the final test coverage obtained by this BBB, since it is not possible to perform two sequential reads without a writing access in between, the final coverage is similar to the one obtained by the BBB tackling the address field. In particular, this means that the proposed algorithm is not suitable for testing the faults belonging to the fault model called CFdrd, since double read operations are not allowed for this field.

Finally, it is worth to note that the above algorithms must not necessarily be executed as a whole, but may be split in parts to be executed separately. In particular, the proposed algorithms are all composed of small independent parts (corresponding to steps 1 to 3 and 4 to 5) that can possibly be executed at different times. Basically, it is possible to execute two separated tests procedures, able to independently address faults of group 1 from those of group 2 (Table 1 and Table 2). Clearly, while the execution of each part is shorter than the execution of the whole algorithm, we must also take care of the cost of the initialization phase, which strongly changes depending on the scenarios. This is an important characteristic when the functional test is executed in-field. In this situation, the test may be executed during the idle times of the application: small time slots are periodically allocated to execute the test. When a test slot begins, the current state of the system is saved and then the test procedure is executed. At the end of the slot the original state of the system is finally restored. Being suitable to be split into smaller chunks is a valuable property for a test procedure in order to execute the test even when small time slots are required [2]. However, the test engineer must carefully evaluate whether the cost for saving/restoring the system status before/after the test execution makes it convenient to split the whole test in chunks or not.

4. Experimental Results

In order to validate the proposed approach we resorted to SimpleScalar [18], an open-source processor architectural simulator widely used for computer architecture research and teaching. SimpleScalar can implement a ROB of arbitrary size (called *Register Update Unit*, or *RUU*), it can emulate several instruction sets (Alpha, PISA, ARM, x86), and (since its source code is available) it can be modified to monitor and store the internal state of the processor, and its ISA is easily expandable to include new instructions.

The PISA architecture has been selected for our experiments, and SimpleScalar has been set to use a variable length ROB. In order to check the correctness of the method, the SimpleScalar C code has been modified to store some additional data during the simulation, allowing to record each time an access is performed to the ROB. We then wrote the code of the proposed algorithm, and checked that it performs the expected sequence of accesses to the ROB.

Since the Simplescalar is a simulator, it is important to describe the hardware configuration and the values we assigned to all its parameters for the experiments described in this work. Parameter values are listed in Table 6.

Table 6 – Simplescalar configuration parameters

Simplescalar configuration parameter	Values
# instructions fetch queue size (in insts)	2
Branch predictor type	BTB
Predictor BTB size	1,024
Instructions decode B/W (insts/cycle)	2
Instructions issue B/W (insts/cycle)	2
Run pipeline with in-order issue	FALSE
Register update unit (RUU) size	8 / 16 / 32
Load/Store queue (LSQ) size	8
Total number of integer ALU's available	4
Total number of integer multiplier/dividers available	1
Total number of memory system ports available (to CPU)	2
Total number of floating point ALU's available	4
Total number of floating point multiplier/dividers available	1
Extra branch mis-prediction latency (in clock cycles)	1
L1 and L2 instruction/data cache hit and miss latencies (in clock cycles)	1
Instruction/data TLB miss latency (in clock cycles)	1

In Table 7 and Table 8 we report the characteristics of the test programs developed for ROBs of different sizes, for testing the *value* and *address* field, respectively. The two tables report in the first column the number of ROB entries, while the second column contains the memory size in bytes of the test program; the third column shows its number of instructions, and finally the last column indicates the number of clock cycles required by its execution.

All the executed programs were manually written. Since the algorithm performs the same operations for each entry of the ROB, it is sufficient to write the block for testing a single entry and

then repeat it *n* times (*n* is the number of entries composing the ROB); to skip an entry, and use the next one as the aggressor cells, we inserted a useless operation (NOP) between two instructions block.

When evaluating the cost of the algorithm in terms of execution time when it is supposed to be used for in-field applications we must consider not only the time required by the algorithm for its execution, but also the extra cost due to the penalties generated by the cache miss it is expected to generated. However, the latter figure is hard to quantitatively evaluate, since it strongly depends on the characteristic parameters of the cache and memory systems. For this reason in our experiments we suitably set the SimpleScalar parameters in order to minimize the impact of cache and TLB misses in terms of clock cycles as depicted the last rows in Table 6.

Table 7 – Test program characteristics for the version addressing the *value* field, for different ROB sizes

ROB size [# entries]	Memory occupation [# bytes]	Executed instructions	Time [clock cycles]
8	6,32 K	1,575	2,137
16	24,9 K	6,623	5,682

Table 8 – Test program characteristics for the version addressing the *address* field, for different ROB sizes

ROB size [# entries]	Memory occupation [# bytes]	Executed instructions	Time [clock cycles]
8	9,17 K	2,347	6,338
16	36,23 K	9,275	12,673

As the reader can notice, the experimental results validate what has been reported in the paper in terms of program complexity for 8 and 16 entries ROBs. As expected, the number of instructions and the memory occupation grow following a quadratic trend with respect to the number of entries in the ROB. However, the program execution time does not follow the same pace, since it mainly depends on the long execution time instructions (called in these experiments *I1* and requiring 20 clock cycles); the total execution time actually only doubles in the cases reported in Table 7 and Table 8.

For ROBs composed of 32 entries or more, the number of available general purpose registers in the SimpleScalar simulator (32) represents a limitation that currently prevents us from applying the proposed approach in the form proposed here. However, this obstacle may be circumvented by also exploiting the floating-point registers available in the processor at the expense of slightly more complex test programs.

The fault coverage of the proposed test programs has been evaluated by modeling all operations performed on the ROB by the proposed test algorithms into the RASTA memory fault simulator [16]. Table 9 and Table 10 show the outcome of the fault analysis considering the test program for the *value* field and for the *address* field, with different dimensions of the ROB.

Table 9 – Fault Coverage of the test program for the *value* field

ROB size [# entries]	Single-Cell FPs	CFst, CFds, CFtr, CFwd, CFrd, CFir	CFdrd
8	100%	100%	92.85%
16	100%	100%	96.66%
32	100%	100%	98.38%

Table 10 – Fault Coverage of the test program for the *address* field

ROB size [# entries]	Single-Cell FPs (except DRDF)	DRDF	CFst, CFds, CFtr, CFwd, CFrd, CFir	CFdrd
8	100%	0%	100%	0%
16	100%	0%	100%	0%
32	100%	0%	100%	0%

As expected, regardless of the ROB dimension, we obtained 100% fault coverage on all instances of single-cell faults and double-cell faults both for *value* and *address* field, with the exception of the CFdrd faults in the case of the value field and CFdrd and DRDF faults in the case of the address field, that were not fully covered. This confirms that all requested coverage conditions have been respected during the implementation of the algorithm. In the case of the value field, the incomplete coverage of CFdrd faults is due to the impossibility of performing two consecutive read operations on all victim cells of the buffer. As reported in Section III, our test program first reads each entry of the ROB with the exception of the last entry since each short instruction uses as operand the outcome of the previous instruction that is stored in the ROB. All entries (including the last one) are then read again during the commit phase when the content of the ROB is written in the target location. Therefore, the CFdrd sensitizing condition (i.e., two consecutive reads), in the case of *value* field, is not matched for the last entry of the ROB, preventing a 100% coverage of this type of faults. In the case of the *address* field, instead, this kind of faults and also the corresponding single cell faults (DRDF) are never detected because it is impossible for a program to execute two consecutive read instructions on the address field of the same entry of the ROB. Therefore, this kind of faults belongs to the largest class of Functionally Untestable Faults [17].

5. Conclusions

The Reorder Buffer is a key component in modern superscalar processors; therefore, testing the memory within this component with respect to possible hardware faults affecting it is crucial for guaranteeing the correct behavior of the processor. When resorting to DfT solutions (e.g., based on BIST) is not possible (e.g., because the DfT structures are not accessible and/or documented, as it sometimes happens when the test has to be performed in the field, and its development is up to the system company), the functional approach can be the only viable alternative. This paper proposes an approach based on developing and running a functional program for the test of the ROB memory, to be used for the in-field test of a processor or a SoC including a processor core. The approach has been described referring to the *value*, *address,* and *identifier* fields of the ROB.

Given the fact that this module is deeply embedded in the processor, and due to the constraints in its access (a ROB is a FIFO buffer) it is not possible to straightforwardly apply a March algorithm for the test of the ROB memory. Therefore, the proposed approach is based on identifying a sequence of instructions, that execute the wished sequence of read and write operations on the target memory array, thus allowing to test both single- and double-cell faults. The method is particularly suitable for a test performed during the operational phase, since it can be executed both as a whole, or split in small independent pieces.

The method correctness has been validated resorting to the SimpleScalar simulator, while its fault coverage capabilities with respect to the major fault types have been first evaluated theoretically (working on the required fault primitives), and then experimentally (resorting to a memory fault simulator).

Through this paper, we have shown a method for generating programs to test those parts of the ROB memory storing the abovementioned fields. Some faults cannot be detected in this way, due to the difficulty of accessing the ROB, so that in certain cases it is not possible to apply the stimuli required to excite them.

The authors are now working towards removing some of the current limitations of the proposed algorithm (e.g., in terms of the required number of registers) and extending it to the test of the circuitry surrounding the ROB memory.

References

[1] I. Bate, P. Conmy, T. Kelly and J. McDermid, «Use of Modern Processors in Safety-Critical Applications», *Computer Journal*, vol. 44, no. 6, pp. 531-543, 2001.

[2] A. Benso, S. Di Carlo and A. Savino, «Software-Based Self-Test for Reliable Applications in Railway Systems». In: *Railway Safety, Reliability and Security: Technologies and Systems Engineering* / Francesco Flammini. IGI Global, Hershey (PA), pp. 198-220. ISBN 9781466616431.

[3] L. Chen and S. Dey, «Software-Based Self-Testing Methodology for Processor Cores», *IEEE Trans. on Computer-Aided Design,* vol. 20, n. 3, pp. 369 - 380 , 2001.

[4] S.M. Thatte and J. A. Abraham, «Test Generation for Microprocessors», *IEEE Trans. On Computers,* vol. 29, n. 6, pp. 429-441, 1980.

[5] M. Psarakis, D. Gizopoulos, E. Sanchez and M. Sonza Reorda, «Microprocessor Software-Based Self-Testing», *IEEE DESIGN & TEST OF COMPUTERS,* vol. 27, n. 3, pp. 4-19, 2010.

[6] E. Sanchez, M. Sonza Reorda and A. Tonda, «On the Functional Test of Branch Prediction Units based on Branch History Table», in *19th IFIP/IFEE International Conference on Very Large Scale Integration and SoC*, 2011.

[7] S. Di Carlo, P. Prinetto and A. Savino, «Software-Based Self-Test of Set-Associative Cache Memories», *IEEE Transactions on Computers,* vol. 60, n. 7, pp. 1030 – 1044 , 2011.

[8] A. Apostolakis, D. Gizopoulos, M. Psarakis, D. Ravotto and M. Sonza Reorda, «Test Program Generation for Communication Peripherals in Processor-Based SoC Devices», *IEEE Design & Test of Computers,* vol. 26, n. 2, pp. 52-63, 2009.

[9] S. Di Carlo, E. Sanchez, M. Sonza Reorda, «On the On-line Functional Test of the Reorder Buffer Memory in Superscalar Processors», *2013 IEEE 16th International Symposium on Design and Diagnostic of Electronic Circuits & Systems (DDECS)*, pp. 36-41, April 2013.

[10] S. Di Carlo, P. Prinetto, «Models in Memory Testing», Springer, 2010.

[11] M.F. Younis, T.J. Marlowe, A.D. Stoyen, G. Tsai, «Statically safe speculative execution for real-time systems», *IEEE. Trans. On Software Engineering*, vol. 25, no. 5, pp. 701-721, 1999.

[12] A.J. Van de Goor, I.B.S. Tlili, S. Hamdioui, «Converting March tests for bit oriented memories into tests for word-oriented memories», *International Workshop on Memory Technology, Design and Testing*, pp. 46-52, 1998.

[13] S. Hamdioui, «Testing Multiple Port Memories: Theory and Practice», PhD Dissertation, Delft University of Technology, Apr. 2001, ISBN 90-9014986-4

[14] A. Benso, A. Bosio, S. Di Carlo, G. Di Natale, P. Prinetto, «Automatic March tests generation for multi-port SRAMs», *3rd IEEE International Workshop on Electronic Design, Test and Applications*, 2006.

[15] [Online]. Available: http://www.simplescalar.com/.

[16] A. Benso, S. Di Carlo, G. Di Natale and P. Prinetto, «Specification and Design of a New Memory Fault Simulator» in *IEEE 11th Asian Test Symposium*, pp.92-97, 2002.

[17] P. Bernardi, E. Sanchez, M. Sonza Reorda, O. Ballan, M. Bonazza, "On-Line Functionally Untestable Fault Identification in Embedded Processor Cores", Design, Automation & Test in Europe Conference & Exhibition (DATE '13), 2013.

[18] J. L. Hennessy and D. A. Patterson, «Computer Architecture: A Quantitative Approach», Morgan Kaufmann, 2011.