On the on-line functional test of the Reorder Buffer memory in superscalar processors

Authors: Di Carlo, S. ; Sanchez, E. ; Sonza Reorda, M.

Published in the Proceedings of the IEEE 16th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS), 2013

DOI: 10.1109/DDECS.2013.6549785

On the On-line Functional Test of the Reorder Buffer Memory in Superscalar Processors

S. Di Carlo, E. Sanchez, M. Sonza Reorda
Dipartimento di Automatica e Informatica, Politecnico di Torino, Torino, Italy
{stefano.dicarlo, ernesto.sanchez, matteo.sonzareorda}@polito.it

***Abstract*— The Reorder Buffer (ROB) is a key component in superscalar processors. It enables both in-order commitment of instructions and precise exception management even in those architectures that support out-of-order execution. The ROB architecture typically includes a memory array whose size may reach several thousands of bits. Testing this array may be important to guarantee the correct behavior of the processor. Proprietary BIST solutions typically adopted by manufacturers for end-of-production test are not always suitable for on-line test. In fact, they require the usage of test infrastructures that may be expensive, or may not be accessible and/or documented. This paper proposes an alternative solution, based on a functional approach, which has been validated resorting to both an architectural and a memory fault simulator.**

Keywords—microprocessor testing, software-based self-test, embedded memory test, on-line test

I. INTRODUCTION

The widespread use of embedded microprocessors coupled with the demand for increased functionality and higher performance are pushing out-of-order superscalar microarchitectures into the embedded microprocessor space. This trend is also embracing safety related applications (e.g., the automotive, railways and aerospace domains [1][2]). For these application domains, both accurate end-of-manufacturing test and periodic test procedures during the operational phase (*on-line test*) are mandatory requirements. These requirements are enforced by several standards and regulations (e.g., ISO 26262 and DO-254) that specify the fault coverage figures that must be attained with respect to permanent faults.

When considering processor-based systems, either implemented within an electronic board or in the form of a System-on-Chip (SoC), functional test approaches, also referred to as Software-Based Self-Test (SBST) methods [3], may represent a promising solution. 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.

Functional test methods have several advantages when compared to traditional Design for Testability (DfT) or Built-In Self-Test (BIST) techniques. 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. Second, 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 the operational phase using DfT techniques may be complex. Detailed information about available DfT structures is often missing. Moreover, these structures might be not accessible during operational conditions or, even if they can be accessed and activated in this phase, their usage may be prevented by resource and time constraints imposed to the test.

While providing several advantages, 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. Starting from the first attempt 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].

This paper focuses on the on-line test of the Reorder Buffer (ROB) existing within superscalar processors. The ROB plays an important role within processors supporting out-of-order speculative execution. It guarantees that instruction commitment (i.e., the phase in which the result produced by the instruction is written to its destination) is performed in-order, and that exceptions are managed in a precise manner [9]. Since the ROB function is implemented in rather different ways in different processor architectures, for the purpose of this paper we selected the solution adopted by the widely spread SimpleScalar simulator [15]. The ROB is based on a memory block organized in entries. Each entry corresponds to an executed instruction waiting for being committed. Since the size of the ROB is usually in the order of some tens of entries, its internal memory capacity is usually in the order of some thousands of bits.

In this paper we propose a test algorithm easily translatable into a program suitable for the on-line test of the ROB internal memory. The overall idea is to map the set of fault primitives required to sensitize and detect typical memory fault types [10] to a proper sequence of processor instructions. One of the major contributions of the paper is in describing how to perform read and write operations on the ROB memory words following a predefined order, thus enabling the translation of a generic March algorithm into a corresponding test program. The full set of single-cell and double-cell memory faults typically considered in traditional memory testing approaches are considered. The complexity of the resulting test algorithm grows quadratically with the ROB size. 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 is also that it does not need to be executed as a whole. In fact, it can be split in fragments to 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.

The approach proposed in this paper is clearly not suitable for the end-of-manufacturing test performed by processor manufacturing companies, which can resort to more effective solutions based on DfT. However, the approach may be useful for system companies, which buy processor or SoC devices from third companies. In this case the engineer in charge of developing the on-line test often does not have detailed information on the internal architecture of the processor, but only knows its Instruction Set Architecture.

The proposed algorithm has been validated resorting to both an architectural and a memory fault simulator. 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: section 2 reports some background about the ROB architecture and behavior. Section 3 describes the functional approach we propose for generating suitable programs for its test; section 4 reports some data about the experiments we performed. Section 5 draws some conclusions.

II. BACKGROUND

In order to better exploit the Instruction Level Parallelism existing in almost every executed 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 in other processors, 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 id of the instruction, (ii) the value it produced, and (iii) the target where this value must be written when the instruction is committed (corresponding either to a register or to a memory location). The ROB is accessed during different phases of the execution of an instruction:

- During the *issue* phase, the processor assigns the instruction to a 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.
- 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.
- 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*, i.e., the produced values are written to the assigned target locations.
- 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, instead of the Register File. It then forwards it to the functional unit, which can thus start its execution.

To summarize, access to the ROB is performed (for sake of simplicity we mainly refer to access to 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 the X;
- In the commit phase, to *read* the value field and *write* its value in the instruction target location.

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.

III. PROPOSED APPROACH

According to Section II, the ROB is composed of several entries, each comprising different fields. For the purpose of this paper we will focus on a specific field, namely the *value* field, and we will propose an algorithm for its functional testing. The other fields, whose number and role often change depending on the target processor, can be tested by extending the approach in a rather straightforward manner.

Let us denote by n the number of entries of the ROB and by m the number of bits composing the value field. Using this notation the ROB internal memory can be modeled as a $n \times m$ memory array whose test is the target of this paper.

The proposed test algorithm implements a deterministic sequence of read/write operations on the ROB entries. In the case of the *value* field, a write operation arises when the instruction associated to the entry completes its execution: in this stage, the produced value is written in the corresponding ROB entry. Write operations are therefore executed following the order 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 following we will first recall the test conditions required to detect single-cell and double-cell (i.e., coupling) faults in a memory, and then will outline an algorithm 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.

A. *Single- and double-cell fault test requirements*

Let us denote by A and V 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. From the literature we can easily derive the operations (denoted as *Fault Primitives*, or FPs[1]) required to test faults affecting single cells in the memory [8]. They are summarized in Table I.

TABLE I. 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 consider address faults affecting pairs of memory cells (denoted as *aggressor* and *victim*, respectively) and report the corresponding FPs (see Table II).

Looking at Table II, 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 II. DOUBLE-CELL FAULT PRIMITIVES

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

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.

B. *Test algorithm*

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 $V/\bar{V}$ in all victim entries and then $A/\bar{A}$ in the aggressor entry;
2. Write $V/\bar{V}$ in all victim entries and then $A/\bar{A}$ in the aggressor entry;

[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.

3. Read the content of all entries starting from the aggressor to detect faults of group 1;
4. Write $V / \bar{V}$ in all victim entries and then $A / \bar{A}$ in the aggressor entry;
5. Read all victim entries two times (if possible) to detect all faults of group 2.

To prove that BBB is able to detect the FPs introduced in Section III.A we focus on double-cell faults reported in Table II. 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 BBB by writing $V / \bar{V}$ in all victim entries and then $A / \bar{A}$ in the aggressor entry. Step 2 of 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 depend on the actual patterns written in the entries during steps 1 and 2. If for instance in both steps 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 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 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 specific 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 ($<A;Vw_{\bar{V}}/V/->$) is sensitized. Step 4 terminates with a write operation on the aggressor entry. Again 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.

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 result equal to $A / \bar{A}$;
- *n-1* instructions (named I2 to In) characterized by a short execution time (e.g., ADD), 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 the values written into memory during the algorithm execution comply with the expected ones. It is worth to note that the above algorithm must not necessarily be executed as a whole, but may be split in parts to be executed separately. In particular, it is composed of small independent parts (corresponding to steps 1 to 3 and 4 to 5) that can possibly be executed at different times. This is an important characteristic when functional test is executed in-field. In this situation, small time slots are periodically allocated to execute test procedures on the system. When a test slot begins, the current state of the processor is saved and then the test procedure is executed. At the end of the slot the original state of the processor is finally restored. Being suitable to be split into small chunks is a valuable property for a test procedure in order to execute the test even when small time slots are required [2].

The presented algorithm corresponds to the execution of $5 \times 6 \times n^2$ 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 $\bar{X}$;
- 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 $X = 00 \leftrightharpoons 00$ and $\bar{X} = 11 \leftrightharpoons 11$; at the second iteration $X = 00 \leftrightharpoons 11$ (i.e., a word composed of *m*/2 0 bits and *m*/2 1 bits) and $\bar{X} = 11 \leftrightharpoons 00$ (i.e., the opposite of X); at the last iteration $X = 10 \leftrightharpoons 10$ (i.e., a word composed of m alternated 0 and 1 bits) and $\bar{X} = 01 \leftrightharpoons 01$ (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 I and Table II. 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.

IV. Experimental Results

In order to validate the proposed approach we resorted to SimpleScalar [9], 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), 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.

In Table III we report the characteristics of the test programs developed for ROBs of different sizes. The table reports in the first column the number of ROB entries, the second column contains the memory occupation in bytes required by the test program, the third column shows the number of instructions, and finally, the last column indicates the number of clock cycles required to execute the test program. It is important to mention that we suitably set the Simplescalar parameters in order to minimize the impact of cache and TLB misses in terms of clock cycles. This is possible to reach by modifying the Simplescalar configuration parameters e.g., data cache miss/hit latencies, through a configuration file.

TABLE III. Test characteristics 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

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 memory occupation grows following a quadratic trend depending on 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) that actually only doubles in the cases reported in Table III.

Interestingly, for ROBs composed of more than 32 entries, the number of available general purpose registers may represent a limitation that 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 program has been evaluated by modeling all operations performed on the ROB by the proposed test algorithm into the RASTA memory fault simulator [16]. Table IV shows the outcome of the fault analysis considering different dimensions of the ROB.

TABLE IV. FAULT COVERAGE

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%

As expected, regardless of the ROB dimension, we obtained 100% fault coverage on all instances of single-cell faults and double-cell faults with the exception of CFdrd that was not fully covered. This confirms that all requested coverage conditions have been respected during the implementation of the algorithm. The not-full coverage of CFdrd 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) is not respected for the last entry of the ROB, preventing a 100% coverage of this type of faults. This coverage penalty is mitigated when the ROB size increases, and, in general the overall coverage figure can be considered satisfactory even for the smaller ROB size.

V. CONCLUSIONS

The Reorder Buffer is a key component in modern superscalar processors; testing the memory within this component is therefore crucial for the correct behavior of the processor. When resorting to DfT solutions (e.g., based on BIST) is not possible (e.g., when the test has to be performed in the field), the functional approach can be the only viable alternative. This paper proposes an algorithm allowing to write a functional program for the test of the ROB memory, to be used for on-line test of a processor or a SoC including a processor core.

Given the constraints in its access (a ROB is a FIFO buffer) it is not possible to straightforwardly adopt a March algorithm. Therefore, the proposed algorithm is based on a sequence of operations, allowing to test both single- and double-cell faults. The algorithm 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 algorithm 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).

The authors are now working towards removing the current limitations of the proposed algorithm 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 et al., «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 et al., «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 et al., «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] J. L. Hennessy and D. A. Patterson, «Computer Architecture: A Quantitative Approach», Morgan Kaufmann, 2011.

[10] S. Di Carlo e 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 et al., «Specification and Design of a New Memory Fault Simulator,» in *IEEE 11th Asian Test Symposium, pp.92-97*, 2002.