About on-line functionally untestable fault identification in microprocessor cores for safety-critical applications

R. Cantoro, A. Firrincieli, D. Piumatti, M. Restifo, E. Sanchez, M. Sonza Reorda
Politecnico di Torino, Dip. Automatica e Informatica
Torino, Italy

***Abstract*[1]— When microprocessor cores are used in safety-critical applications, in-field test must be performed to reach the target reliability figures. In turns, the in-field test must be organized so that it achieves a sufficient fault coverage. The fault list to be considered for computing the fault coverage should only include testable faults, i.e., faults which may cause a failure in the operating conditions. Hence, single permanent faults that in the operating conditions cannot be excited, or do not propagate to any output, or both, should be removed from the list. These faults, called *on-line functionally untestable faults*, require a significant effort to be identified. The contribution of this paper is twofold. From one side, it reports experiments, showing that in typical embedded safety-critical systems their number is often far from being negligible, and depends on many parameters, including the application code run by the processor. Secondly, it provides a semi-automated approach to their identification. Experimental results on a representative microprocessor are reported.**

Keywords—component; formatting; style; styling;

I. INTRODUCTION

When an electronic system is used in a safety-critical application, the probability that a fault may cause a failure must be evaluated and compared with the target reliability figures. In several domains standard and regulations (e.g., IEC 61508 for industrial systems, DO-254 for avionics, ISO 26262 for automotive) mandate well-defined procedures for reliability assessment and define target reliability figures.

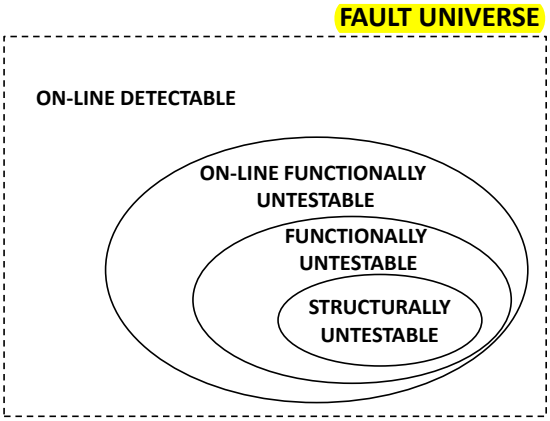

Fig. 1. On-line fault categories and their relationship [4].

It is common to adopt some in-field test solutions, which can be activated during the operational phase and are able to detect possible permanent faults before these may produce any failure. These solutions may be based on Design for Testability techniques (such as BIST) or on self-test functional approaches (such as Software-based Self-test [2]), or on a combination of the former ones. The quality of these solutions is measured first of all in terms of achieved fault coverage with respect to the adopted fault model(s). Clearly, the higher the level of safety that the application must achieve, the higher the fault coverage which is required. The fault coverage figure is computed with respect to a fault list which as a first step includes all possible faults. However, permanent faults that cannot produce any failure in the operational mode (denoted as *untestable faults*) can be removed from this list.

This category of single permanent faults includes several groups, which are illustrated in Fig. 1:

1. *Structurally untestable faults*, i.e., faults for which a test does not exist even if the combinational block where the fault is located is fully controllable and observable. Examples of faults belonging to this category include faults that cannot be tested due to some redundancy in the combinational logic. If a gate-level description of the device is available, an ATPG tool can identify some of these faults.
2. *Functionally untestable faults*, i.e., faults that do not belong to the previous group, but cannot produce any failure due to the sequential behavior of the circuit, for example, in the case the circuit cannot reach all possible states. Several works proposed techniques to automatically identify these faults, either in a generic circuit [5][6][7] or specifically in a CPU [8].
3. *On-line functionally untestable faults*, i.e., faults that do not belong to the previous groups, but cannot produce any failure in the operational conditions the target device works in. As an example, all faults related to the debug circuitry in a processor belong to this group, since debug facilities are not used during the normal behavior. In [4] we already reported some examples of faults belonging to this category, and showed that their number is typically not negligible. In this paper, we extend the number of faults belonging to this category by also considering

[1] This work has been supported by the European Union through the H2020 project no. 637616 (MaMMoTH-Up).

978-1-5386-1472-3/18/$31.00 ©2018 IEEE

faults that cannot produce any failure, due to the specific application code executed by the CPU. In the ISO26262 terminology, these faults are called "safe faults application dependent".

Identifying untestable faults is crucial, because it allows to remove them from the fault list and to focus the test efforts towards the testable faults, only. Moreover, knowing the list of untestable faults may permit reducing the effects of over-testing phenomena, which are known to reduce the yield, and thus the profit of semiconductor and system companies.

In the past, there have been several efforts to develop effective solutions to the automatic identification of structurally and functionally untestable faults. In general, techniques for identifying on-line functionally untestable faults are less mature, and most of the work is still done manually, often in the frame of the so-called FMEA (*Failure Mode Effects Analysis*). Some preliminary discussion about possible methods for functionally untestable fault identification was given in [4] and [9], based also on the observation that limitations in the usable address space [3] may increase the number of on-line functionally untestable faults.

In this paper, we make some further steps in this direction. We focus on the typical scenario characterizing a special purpose system, i.e., a system built to perform a single application. This means that the processor included in this system only executes a single piece of code, written to perform the target application, which remains the same during the whole operational life. The first contribution of this paper is to show that the number of on-line functionally untestable faults existing in the embedded processor is increased by the fact that the application code is fixed, and cannot cover all possible scenarios the processor has been designed for. In other words, given a certain application code and all possible input data set, it may happen that some internal resources are never accessed: hence, all faults associated to these resources belong to the class of on-line functionally untestable faults. As a trivial example, if the application code never uses multiplication, the faults associated to the multiplier becomes untestable, and can thus be removed from the list used to compute fault coverage. Faults related to other resources (e.g., those supporting test or debug) may also belong to the same category. A similar analysis is reported in [1], where the goal is to identify gates in the processor that do not play any role in such a scenario, and can thus be removed from the design. In our case, the processor design is not changed, but we adopt a similar approach to identify these gates, because they are associated with on-line functionally untestable faults.

The second contribution of the paper is to propose a semi-automated and scalable method, able to identify a good percentage of the on-line functionally untestable faults. The availability of such a method can clearly reduce in a significant manner the amount of time and effort to perform the reliability analysis required by several standards and regulations.

In order to provide the reader with a test case where the method is applied, we selected a representative processor, considered a few application codes, and applied our method to identify the functionally untestable faults. Results show that the number of on-line functionally untestable faults that can be identified by the method is generally not negligible, accounting for up to 30% of the total number of faults. Our method, although not able to identify all of them, appears to be able to significantly reduce the effort for their identification.

The rest of the paper is organized as follows: Section II introduces some basic definitions: first, the controllability is introduced, then, the cone definition is explained with some motivations. Section III better details how the proposed method works with emphasis on the cone extraction algorithm and the logic gate activity extraction. Section IV shows the results obtained by running different applications code on a widely used low-power microprocessor.

II. Background and Motivations

This section provides the reader with the required information about the controllability metric and partitioning of the circuit in cones. This knowledge allows to fully understand the following sections.

A. Controllability

The concept of controllability C has been defined in [10] as the probability that a random input vector for a combinational block forces a given line l to the value 1 (C^l=1) or 0 (C^0=1). The controllability value depends on the logic function implemented by the block and can hold any value inside the interval [0:1].

Controllability values of the inputs are usually equal to 0.5 both for logic zero and logic one. However, when some special conditions hold, these values may be changed accordingly.

When the controllability value is 0, it indicates that it is not possible to set the line to a specific value. For example, C^l(A) = 0 means that line A cannot assume the value 1.

Generally speaking, the controllability of a line holds values in the range [0:1], and it rarely holds the values 1 or 0.

The computation of the controllability *C* for all lines in a logic block can be performed by starting from the input block and then proceeding towards its outputs level by level. For each gate, we can compute the controllability of the output line by knowing the type of the gate and the controllability values of its inputs. The following equations hold for OR/AND gates:

For an *n* input OR gate

$$C^0(N) = 1 - C^1(N)$$

$$C^1(N) = 1 - \prod_{i=1}^{n} C^1(x_i)$$

For an *n* input AND gate

$$C^0(N) = 1 - \prod_{i=1}^{n} C^0(x_i)$$

$$C^1(N) = 1 - C^0(N)$$

For a NOT gate

$$C^0(N) = C^1(x)$$

$$C^1(N) = C^0(x)$$

where N is the output signal of the gate under exam and x_i are its inputs.

Fig. 2 provides an example and the relative expansion formulas of controllability are developed in the following.

$$C^0(X_1) = C^0(X_2) = C^0(X_3) = 0.5$$

$C^1(X_1) = C^1(X_2) = C^1(X_3) = 0.5$
$C^1(a) = C^1(b) = 1 - (0.5 \cdot 0.5) = 0.75$
$C^0(a) = C^0(b) = 1 - C^1(a) = 0.25$
$C^0(c) = 1 - (0.25 \cdot 0.25) = 0.9375$
$C^1(c) = 1 - C^0(c) = 0.0625$

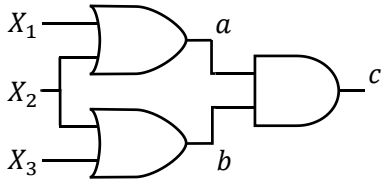

Fig. 2. Example of controllability computation.

B. Cone Partitioning Algorithm

This sub-section describes the cone definition used in this work that has been inspired by [11]. A *cone* in a combinational block is the set of all gates that are directly or indirectly fed by a given input signal.

Fig. 3 illustrates an example of a combinational block and the cone associated to an input signal. The CONE starts from input pin X and arrives up to output O_1 and output O_2.

The cone extraction process is based on the Cone Partitioning Algorithm (CPA). The CPA is based on a Breadth-First-Search over the graph representation of the combinational block netlist. Figure 4 depicts the result of the CPA on the example proposed in Fig. 3.

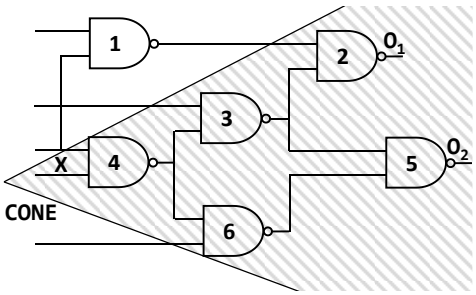

Fig. 3. Example of a cone.

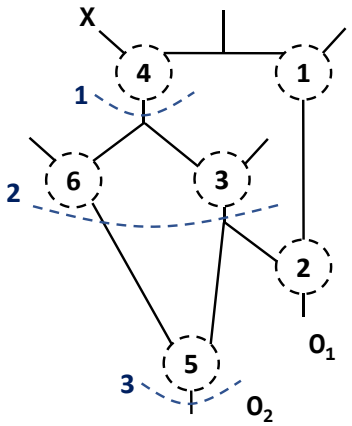

Fig. 4. CPA example related to the example of Fig.3.

C. Motivations

In today safety critical applications, it is important to provide an appropriate methodology able to assure the highest system reliability; however, most of the techniques available today do not differentiate the faults that should be targeted during on-line testing. In fact, the research literature is still missing for a methodology able to clearly identify faults that are untestable during the mission operation.
In our proposed methodology, given a generic program, we can distinguish two main parts:

- The Application Code: the binary image corresponding the compiling of the application code (e.g., written in C or assembly) that performs the speficic application. Such a binary image is typically fixed after the compiling step.

- The Data Set: possible data (e.g., variables, values coming from the otuput, sensors, files, etc.) used by the application code for its operation. Data can be stored inside the main memory or inside some internal registers of the processor core. Contrarily to the application code, data can be modified after the compile step.

As an example, let us consider a matrix multiplication program, as the one depicted in Fig. 5. Such a program makes use of two matrices A and B to compute the resulting matrix C. In this example, the application code is the algorithm performing the matrix multiplication, while the data set is composed of the values allocated in the input matrices A and B.

The purpose of this work is to analyze the effect of any variation in the program data that can affect the testability of faults. For the first time, our work aims at classifying testable faults according to such analysis, and provides figures about on-line functionally untestable faults in a processor running a well-defined application.

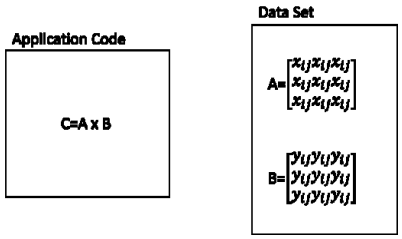

Fig. 5. Example of application code and data set

III. Proposed Method

The approach proposed in this paper examines the effects of the program execution on the processor netlist in order to obtain the set of on-line functionally untestable faults. This classification leads to a reduction in the number of faults that must be detected and relaxes the on-line test requirements of a safety-critical system. Fig. 6 shows and summarizes the schema adopted in the proposed strategy.

First, a topology analysis is performed aiming at:

- identifying the netlist elements possibly connected to a fixed signal by the synthesis process (a);

- extracting the cones for each input of each combinational block (b).

Then, a logic simulation (c) of the processor while running the application is performed. The logic simulation records the circuit activity, which provides information regarding the toggle activity (d) for all signals in the netlist. If feasible, the logic simulation can be repeated a number of times with different data sets.

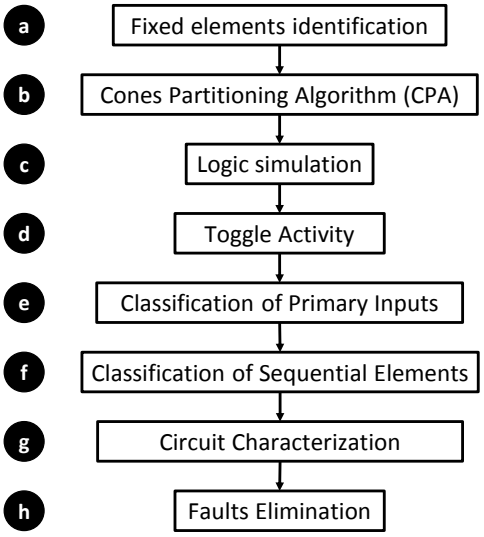

Fig. 6. Proposed method schema.

After, the circuit Primary Inputs (PIs) are classified (e). PIs can assume a constant value or can change during the operational life. Consequently, if a PIs has toggled during the simulation, then the PIs is marked as NOT-FIXED, else it is marked as FIXED.

In the case of the flip-flops (FFs) the concept is slightly different, since they correspond to internal signals. A FF can toggle or not depending on the input constraints, on the program code and on the data set the program works on. For this reason, the FFs are grouped in the following categories:

- *FIXED* sequential elements (type F), which include all FFs that never toggled during the application execution, and will never toggle, no matter the considered data sets;
- *POTENTIALLY NOT-FIXED* sequential elements (type PNF), which include all FFs that never toggled during the application execution, but may toggle in the case a different data set is considered;
- *NOT-FIXED* sequential elements (type NF), which include all FFs that toggled during the application execution.

The circuit FFs are classified (f) on types F and PNF manually, out of those that never toggled during the logic simulation(s), based on the analysis of the processor RT-level model. Examples of FFs belonging to the PNF type include those inside the interrupt management block, which are marked as PNF since an external event can cause them to toggle.

This kind of classification is a conservative approximation and it was also applied to other blocks: for example, given a register from the register file block, if a toggle activity is recorded for at least one bit, while the other bits did not toggle, the whole register is marked as type PNF.

Afterwards, the circuit is characterized (g) considering all the information obtained by the previous steps. The controllability of the fixed PIs and type F FFs is set to the appropriate value (0 or 1). The controllability of the remaining PIs, PNF and NF FFs are set to 0.5. At this point, it is thus possible to use the previously proposed method and compute the controllability values for all the lines belonging to the circuit.

Once the controllability evaluation ends, a log file is produced, which reports all gates having a controllability value (C^0 or C^1) equal to 1. For example, if the log file reports that $C^1(A) = 1$, it means that the gate A was never set to the value 0 during the execution of the target application. The log file defines all the possible fault locations that cannot be forced to the opposite value. hence, the corresponding faults can be marked as on-line functionally untestable.

The last step of the method depurates the identified faults from the initial fault list (h).

IV. Experimental Results

The proposed methodology has been experimented on the open-source low-power processor openMSP430 available through OpenCores [12]. The implementation of the method resorts to a TCL script for Synopsys Design Compiler. In the following, we will first provide some details about the experimental setup and then report the results related to the execution of some selected application programs.

A. Experimental setup

The openMSP430 processor is a synthesizable 16-bit microcontroller core written in Verilog. It is compatible with the Texas Instruments' MSP430 microcontroller family and can execute the code generated by any MSP430 toolchain in a nearly cycle accurate way [13]. The core has some embedded peripherals like a 16x16 HW Multiplier, Watchdogs, and Timers. We synthesized the openMSP430 resorting to Synopsys Design Compiler, using the NanGate 45nm Open Cell Library [14]. The size of the resulting gate-level design is approximately 15k Equivalent Gates, including 834 sequential elements. The uncollapsed stuck-at fault list accounts for 51,744 faults. The reader should note that the size and complexity of this CPU module is comparable with many similar CPU modules used in safety-critical embedded applications, e.g., in the automotive domain. Table I reports the distribution of stuck-at faults over the main CPU sub-modules. Synopsys TetraMax has been used for the identification of the structurally untestable (UT) faults reported in the table.

TABLE I STUCK-AT (SA) FAULTS DISTRIBUTION IN THE OPENMSP430

Sub-module	Total SA faults	Structurally UT faults
clock_module	2,180	86
debug	8,340	206
execution_unit	18,434	300
frontend	6,268	190
mem_backbone	3,512	78
multiplier	9,936	130
sfr	602	34
watchdog	1,568	76
glue logic	904	0
(whole CPU)	51,744	1,100

In order to apply the proposed method and to identify the on-line functionally untestable faults, 4 benchmarking programs have been selected, whose characteristics in terms of memory footprint, and execution time are reported in Table II. The selected programs are described in the following:

- *Arithmetic*: the program makes use of arithmetic operations, including multiply instructions. It is written in assembly.
- *Matrix multiplication*: it performs the operation $C=A\cdot B$, where A and B are 3x3 integer matrices. It is implemented in C.
- *Quicksort*: it is an efficient sorting algorithm, implemented in C.
- *CoreMark*: it is a synthetic benchmark that measures the performance of central processing units (CPUs) used in embedded systems [15]. The code is written in C and contains implementations of the following algorithms: list processing (find and sort), matrix manipulation (common matrix operations), state machine (determine if an input stream contains valid numbers), and CRC.

TABLE II CHARACTERISTICS OF THE SELECTED BENCHMARKS

Program	Size [kB]	Duration [#clock cycles]
Arithmetic	26.9	2,943
Matrix multiplication	13.6	4,517
Quicksort	36.4	5,426
CoreMark	61.0	1,490,023

B. Results

Logic simulation of the selected benchmarks has been performed using Mentor QuestaSim. During each logic simulation, a Value Change Dump (VCD) file has been generated, which reports the values of each net of the gate-level circuit at any time. The toggle activity of the processor has been then derived from the VCD file using, a second time, Mentor QuestaSim. By running the script implementing the proposed method, and the processor faults have been classified according to the taxonomy described in the previous section. The results on the selected benchmarks are reported in Table III. The computational time (column 2 in Table III) refers to the time required to run the overall flow on a single core of a Xeon processor running at 3.2 GHz. The computational time includes the logic simulation, the toggle activity computation, and the fault classification. In all cases, the time required by our tool for the fault classification is negligible, while the overall computational time (which is in the order of few minutes up to few hours in the worst-case scenario of the CoreMark benchmark) is dominated by the time for logic simulation.

TABLE III FLIP-FLOPS CLASSIFICATION

Program	Comp. time [min]	#FF type F	#FF type PNF	#FF type NF
Arithmetic	3	38.88%	16.15%	43.66%
Matrix multiplication	3	28.35%	23.33%	47.01%
Quicksort	3	38.88%	34.09%	25.60%
CoreMark	120	28.35%	14.23%	56.10%

The reader can notice that the number of FFs that are classified as type F is not negligible and ranges between 28% and 39%. Moreover, only one half of the FFs (about 44% in the worst case) are classified as type NF, meaning that they toggle and that the corresponding faults can be safely marked as functionally testable, while a considerable portion of the remaining FFs (between 16% and 23%) are classified as type PNF. We classified them as potentially testable by using different data in the program, as well as a manual analysis of the processor model. It is also worth noting that the percentage of type F FFs may change significantly when moving from one program to another.

The implementation of the proposed methodology based on the CPA algorithm and the FF classification has permitted to derive the amount of on-line functionally untestable faults in the processor, as reported in Table IV. Clearly, such faults are program-dependent, contrarily to structurally untestable faults, which are derived by the analysis of the circuit topology.

TABLE IV STUCK-AT FAULTS CLASSIFICATION

Program	Testable		On-line functionally untestable	
	faults	%	faults	%
Arithmetic	39,258	75.87	12,486	24.13
Matrix multiplication	39,546	76.25	12,198	23.57
Quicksort	35,930	69.43	15,814	30.56
CoreMark	39,600	76.53	12,144	23.47

The results in Table IV show that the number of on-line functionally untestable faults is not negligible (more than 20% of the total number of faults). This means among the other things that:

- none of these faults is able to produce any failure during the execution of the mission application;
- any functional test method (such as Software-based Self-test, as an example) can safely remove those faults from the fault list used for the test generation;
- other test methods (such as Logic BIST) which will test them are performing overtesting, i.e., they are discarding potentially good devices (i.e., not corrupting the behavior of the mission application).

Moreover, it is worth noting that the number of on-line functionally untestable faults may vary significantly depending on the program executed by the processor. This makes the analysis proposed in this paper and the method to identify them particularly important from a practical point of view.

Finally, if we compare the figures of Table IV with those provided in [1], we see that our method is able to identify a significant percentage of the on-line functionally untestable faults, despite its lower complexity and computational complexity.

The distribution of on-line functionally untestable faults on each processor sub-module is reported in Table V.

The figures in this table show that:

- the different modules can be associated with quite different numbers of on-line functionally untestable faults, both in terms of absolute value and in percentage;
- the debug module is clearly a major contributor, since it does not play any role during the operational phase;
- the execution unit is also associated to a significant number of on-line functionally untestable faults, and this number may change from one program to another depending on the kind of operations performed;
- modules related to the memory access may produce a variable number of on-line functionally untestable faults, depending on the size and location of the data and code memory areas;
- test structures also contribute to the number of on-line functionally untestable faults: since some of them (e.g., the scan chains) are not used during the operational phase, some of the faults associated to them belong to the class of on-line functionally untestable faults.

TABLE V ON-LINE FUNCTIONALLY UNTESTABLE FAULTS ON OPENMSP430 SUB-MODULES

	Arithmetic	Matrix multiplic.	Quicksort	CoreMark
clock_module	37.11%	37.11%	37.11%	37.11%
debug	65.56%	65.56%	65.56%	65.56%
execution_unit	21.79%	18.91%	17.40%	18.61%
frontend	14.13%	14.25%	19.16%	14.25%
mem_backbone	7.03%	13.72%	7.06%	13.72%
multiplier	5.12%	5.12%	43.41%	5.12%
sfr	14.78%	14.78%	14.78%	14.78%
watchdog	21.11%	21.11%	22.07%	21.30%
glue logic	14.38%	14.38%	14.38%	14.38%
(whole CPU)	24.13%	23.57%	30.56%	23.47%

V. CONCLUSIONS

This paper focuses on the group of single permanent faults inside a processor that can be shown not to be able to cause any failure during the operational phase of a microprocessor-based embedded system. The size of this group depends on the system configuration (e.g., the amount of memory and the memory map) but also on the application program executed by the embedded system.

Identifying the largest possible number of these faults allow better tuning the test process, while still guaranteeing the same reliability level, which is often measured resorting to the Testable Fault Coverage metric. Unfortunately, this task is currently performed in a manual manner within the FMEA process, with clear limitations in terms of required effort and achieved results.

This paper is a first effort towards the development of an automatic method for identifying the on-line functionally untestable faults. The proposed technique still requires some manual steps, but it is shown to be able to identify a significant number of the on-line functionally untestable faults with a limited computational effort.

The reported results have been gathered on a freely available processor core considering only stuck-at faults, but can easily be extended to other processors and fault models.

We are currently working at the development of an improved version of our technique, characterized by full automation and increased performance.

REFERENCES

[1] Hari Cherupalli, Henry Duwe, Weidong Ye, Rakesh Kumar, John Sartori, "Bespoke Processors for Applications with Ultra-low Area and Power Constraints", ISCA '17

[2] M. Psarakis et al., "Microprocessor Software-Based Self-Testing", IEEE Design & Test of Computers, vol. 27, no. 3. May-June 2010, pp. 4-19

[3] P. Bernardi, L. Ciganda, M. de Carvalho, M. Grosso, J. Lagos-Benites, E. Sanchez, M. Sonza Reorda, O. Ballan, "On-line software-based self-test of the address calculation unit in RISC processors", Proc. 17th IEEE Eur. Test Symp. (ETS), May 2012, pp. 1–6

[4] P. Bernardi, M. Bonazza, E. Sanchez, M. Sonza Reorda, and O. Ballan, "On-line functionally untestable fault identification in embedded processor cores", Proc. Design, Autom. Test Eur. Conf. Exhibit. (DATE), Mar. 2013, pp. 1462–1467

[5] J. Raik, H. Fujiwara, R. Ubar, A. Krivenko, "Untestable Fault Identification in Sequential Circuits Using Model-Checking", Proc. IEEE Asian Test Symposium, 2008, pp. 21-26

[6] Syal, M.; Hsiao, M.S., "New techniques for untestable fault identification in sequential circuits", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 5, no. 6, 2006, pp. 1117 – 1131

[7] H.-C. Liang; C. L. Lee; Chen, J.E., "Identifying Untestable Faults in Sequential Circuits", IEEE Design & Test of Computers, Vol. 12 , No. 3, 1995, pp. 14-23

[8] W.-C. Lai; Krstic, A.; Kwang-Ting Cheng, "Functionally testable path delay faults on a microprocessor", IEEE Design & Test of Computers, vol. 17, no. 4, 2000, pp. 6-14

[9] A. Riefert; R. Cantoro; M. Sauer; M. Sonza Reorda; B. Becker, "A Flexible Framework for the Automatic Generation of SBST Programs", IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2016, Volume: 24, Issue: 10, pp. 3055 – 3066

[10] F. Brglez, "On testability analysis of combinational networks," IEEE International Symposium on Circuits and Systems, May 1980, pp. 221-225

[11] D. R. Brasen and G. Saucier, "Using cone structures for circuit partitioning into FPGA packages," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 17, no. 7, pp. 592-600, Jul 1998

[12] https://opencores.org

[13] www.ti.com

[14] www.nangate.com

[15] www.eembc.org/coremark/index.php